Interactive Visualization Methods for Mobile Device Applications

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Joachim Diepstraten

aus Leonberg

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. T. Strothotte

Tag der mündlichen Prüfung: 24.1.2006

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2005

To the two special persons I have met in my life

Contents

Lists

List of Tables

List of Figures

List of Color Figures

Abstract

The number of mobile devices today already outnumbers the installed desktop PCs and this gap will further widen in the future since the market for desktop PCs is stagnating but is still growing for mobile devices. For example, the cellular phone is one of the most widespread devices that can display graphical content and its functionality is constantly increasing.

Even with the increased functionality on recent mobile devices bringing interactive graphics and visualization methods to these devices is not as easy as it might appear to be. Strategies and ideas that work for desktop machines cannot necessarily be transferred to mobile devices because they have very different characteristics. The purpose of this thesis is to exactly address this problem.

The current limitations of these devices – limited processing power, small amount of memory, low bandwidth, limited network capacities, small display area – that hinder the usage of current visualization applications on these devices are identified. In order to overcome them, several different strategies are presented that allow interactive 3D graphic solutions to run on mobile devices. These strategies take advantage of different properties of a mobile device, e.g., network speed. Which strategy should be employed for a graphical application can be decided based on the properties of the device that suit best to the strategy. Moreover, algorithms for presenting data in a resource friendly, user recognizable form but at the same time taking account of the display limitations are introduced by addressing ideas in illustrative and artistic rendering. A special focus in this thesis is on the representation of transparency that plays a significant role in extending the usable space for visualizing information. In this thesis several strategies taken from artistic rendering have been successfully applied to the context of mobile rendering solutions. Additionally, a 3D graphical user interface was designed that is well suited to solve spatial problems in mobile devices and allows developers to build new classes of modern user interfaces.

All the new algorithms explained in this thesis do not only extend the capabilities of bringing interactive 3D graphics to mobile device, but can for example also be used for enhancing the richness of visualization or non-photorealistic rendering methods in any sort of interactive environment.

Zusammenfassung

Die Zahl der mobilen Geräte übersteigt bereits heute die Anzahl der installierten Schreibtisch-PC-Geräte. Es ist zu erwarten, dass diese Kluft in der Zukunft weiter wachsen wird, da die Verkaufszahlen für PC-Geräte momentan stagnieren, während die von mobilen Geräten weiterhin zunehmen. Zum Beispiel kann man momentan das Mobiltelefon als das weitest verbreitete elektronische Gerät, mit dem es möglich ist, grafische Daten anzuzeigen, ansehen und seine Funktionalität nimmt immer weiter zu.

Doch trotz der gestiegenen Funktionalität auf den mobilen Geräten ist es nicht so einfach, interaktive Grafik- und Visualisierungsmethoden für diese Gerättypen zu entwickeln, wie es auf den ersten Blick scheint. Strategien, die auf einem Schreibtischgerät gut funktionieren, sind nicht ohne weiteres auf mobile Geräte übertragbar. Der Grund dafür sind spezifische Eigenschaften der mobilen Geräte. Das Ziel dieser Dissertation ist es genau dieses Problem anzugehen.

Die momentanen Beschränkungen von mobilen Geräten (eingeschränkte Rechenkapazität, niedriges Speichervolumen, niedrige Speicherbandbreite, geringe Netzwerkkapazitäten, kleine physikalische Darstellungsfläche), die den Einsatz von Visualisierungsapplikationen verhindern, werden zunächst identifiziert. Um diese Beschränkungen zu umgehen, werden verschiedene Strategien vorgestellt, die interaktive 3D Grafiklösungen auf mobilen Geräten ermöglichen. Diese Strategien nutzen dabei verschiedene Eigenschaften eines bestimmten Geräts, z.B. Netzwerkkapazitäten. Welche dieser Strategien sich für eine Grafikapplikation anbieten, hängt von diesen Eigenschaften ab. Weiterhin werden Algorithmen vorgestellt, die es ermöglichen Daten in einer ressourcenfreundlichen und vom Benutzer erkennbaren Form darzustellen, zur gleichen Zeit aber auch die eingeschränkten physikalischen Darstellungsmöglichkeiten des mobilen Geräts berücksichtigen. Dabei wird hauptsächlich auf Techniken aus dem künstlerischen und nicht-photorealistischen Bereich zurückgegriffen. Einen speziellen Fokus diese Arbeit liegt auf der Darstellung von Transparenz. Diese kann unter anderem verwendet werden, um die räumliche Darstellungsfläche zu vergrössern.

Zum Abschluss der Arbeit wird eine 3D grafische Benutzeroberfläche für mobile Geräte vorgestellt, die besonders geeignet ist, die räumliche Einschränkung, die durch die kleine Darstellungsfläche auf diesen Geräten besteht, zu erweitern.

Alle in dieser Dissertation vorgestellten neuen Algorithmen erweitern das Spektrum von mobilen Geräten, interaktive 3D Grafik darzustellen. Außerdem können sie ebenfalls eingesetzt werden, um die Reichhaltigkeit von Visualisierungsmethoden durch die vorgestellten nicht-photorealistischen Darstellungstechniken zu erweitern.

Acknowledgements

This dissertation would not exist without the help, support and input of a lot of people. Many thanks and gratitude I have to give to my advisor, Thomas Ertl, in particular for his supportive advice and his confidence in my sometimes not always so obvious ideas.

Several aspects of this work were joint work with other and former colleagues of the VIS department in Stuttgart. Most of all I have to thank Dr. Daniel Weiskopf for the great work together on the Non-Photorealistic display of transparent or semi-transparent surfaces which make up a huge part of this thesis. He always was a constant source of information and inspiration and luckily enough he often managed to organize my weird and unstructured thoughts into a straight line. I also want to thank Simon Stegmaier and Dr. Manfred Weiler on the joint and really efficient work in the image-based remote rendering section of this thesis. And last but not least I would like to thank Dr. Martin Kraus for understanding the vision and putting together the vragments (not raxels!) concept.

There are also other projects that I have worked on during my time at the VIS department, though they are not part of this thesis. One particular I like to mention here is the work on the SFB 627 Project D2 – An indoor navigation system for the blind. This project has largely been co-worked with Dr. Andreas Hub and I would like to thank him for the fruitful time spent together working on this project. He managed to broaden my mind on other aspects besides the topics described in this thesis. He also was a great help together with Mike Eissele to bear all these many NeXus project meetings. Another area of research done during my time at the VIS department and which is also not part of this was about graphics hardware performance that I did together with Mike Eissele. Thank you for clearing up and putting together all these diagrams and for some really thorough and interesting discussion on that topic. Although unfortunately in the end we never had a publication together I still would like to thank Dr. Martin Rotard for working together on several ideas. He always was a very supportive and helpful person at the department for me in many aspects.

Furthermore, I have to thank all the students that I advised on their master thesis and pre-master thesis. They helped to achieve some of my goals and a lot of their work and effort is part of this thesis. In particular these are Martin Görke who implemented great parts of the remote line rendering solution, Gueorgui Ovtcharov that worked on the image based remote rendering, and Harald Sanftmann who did work on the platform independent 3D user interface.

For improving this dissertation I would like to thank my colleagues and friends Paula Novío Mallón, Martin Kraus, Mike Eissele, and Guido Reina.

Many thanks to our secretary Ulike Ritzmann for all her support, and to Hermann Kreppein for his help in purchasing the hardware required for this thesis.

I was really blessed with my room mates Dr. Daniel Weiskopf, Dr. Marcello Magallón, and Guido Reina. Thank you for being great and supportive friends.

Many thanks also the people I have had the pleasure to work – especially the Monday serious work sessions – with at the University of Stuttgart; in particular (in alphabetic order without the ones already mentioned) Katrin Bidmon, Ralf Botchen, Michael Braitmaier, Dr. Klaus Dieter Engel, Dr. Matthias Hopf, Thomas Klein, Dirc Rose, Tobias Schafhitzel, Magnus Strengert, Joachim Ernst Vollrath.

Last but not least a big warm thank you goes to my parents Ann and Paul Diepstraten for supporting me all the time to the best of their possibilities.

Chapter 1

Introduction

With the rapid and multi-faceted growth of technology, mobile devices today, e.g., Personal Digital Assistants (PDAs), cellular phones with more capabilities than just phoning (smartphones) have entered many fields of human life and activity. Throughout this work the term mobile device is associated with devices like PDAs and cellular phones or smartphones since other mobile devices like laptops, notebooks, and TabletPCs currently have almost the same performance and peripheral equipment as desktop machines so they can be treated like those.

Mobile devices also still enjoy a significant growing number of users over the past few years. Along with the increasing computation power of the devices, several new types of applications have appeared on them. These applications can for example be categorized into location-aware applications which depend on the current location of the user in the world. According to this information the user is able to access different services, information, or applications currently attached to her/his location. For example, a user might be interested in a certain street, building or place she/he is currently facing, or even want to have an infield display of scientific data, e.g. medical images and geographical information system data. Another area of growth is mobile entertainment where a merging between traditional dedicated hardware for portable gaming and cellular phones can be observed [Nok05].

Both application groups show a growing demand for displaying 2D and 3D graphical content on these devices. At the same time the user should also be able to manipulate this graphics content to a certain extent. This requires not only the possibility of displaying 3D content but to deliver it to the device in an interactive manner. For two-dimensional graphics content there are two major severe problems to solve. The first one is space arrangement and management due to the small display area of the device and the second one is interaction handling, since these devices regularly have only a simple pen-point interface or maybe a small touch keypad with a few buttons but very seldom a full QWERTY keyboard or a multi button point device. Displaying the 2D content itself is rather un-challenging unless the storage of data is a problem, since the power of the devices is nowadays high enough to handle a significant amount of 2D drawing elements.

In contrast, providing interactive 3D graphics on mobile devices is a rather new and challenging task for researches and developers, since there are certain constraints unique to these device types. Most mobile devices can be characterized by a relatively low resolution and display size that is apart from Tablet PCs about 2-4 inch. They also have a relatively small amount of memory and low memory bandwidth performance throughout the system, relative low computation power compared to desktop CPUs and mostly still lack the availability of 3D graphics accelerators. On the other hand due to their nature of being mobile these devices usually provide some sort of connectivity feature over different types of short, middle, and long-range networks. This capability can of course be used for accessing remote computational resources.

Right now two basic approaches can be distinguished in providing interactive three dimensional content to a mobile device. One concept is to use local device resources and to perform the rendering tasks locally on the client. The idea is to transfer the complete 3D content to the client and display it there. This approach has of course several drawbacks. First the storage capacity of these devices is very limited especially their main memory, so it is very unlikely that complex 3D scenarios can be kept on the device for a longer period. More severe is the fact that right now most of the devices do not have any 3D graphics hardware acceleration and therefore have to rely on a full CPU based transformation and rasterization of graphics primitives instead of exploiting parallelism by a combination of GPU and CPU. These two facts limit this approach to deal only with very basic 3D content. A comparison can be taken to the mid to late nineties of the last century where graphics hardware acceleration was not yet very common in desktop PCs and programs trying to display 3D content relied on highly optimized software rasterizers. Similarly, there are currently highly optimized libraries available for mobile devices providing as fast as possible CPU-based rasterizers or other routines for processing 3D data [Int05b, Tre05b]. Recently a shift can be seen in the world of rendering for mobile devices by the introduction of two low-level graphics API especially suited for mobile and embedded devices [Khr05, Mic05a]. More details on the architecture of these libraries will be given later in this work (see Chapter 2).

Although these technologies allow the development – to some extent – of an attractive graphics application, having more complex 3D scenes is still out of range for mobile devices. This is mainly due to the limits found in the local hardware. These limits will certainly be pushed further in the future when more sophisticated 3D graphics hardware acceleration support is available for the next generation of mobile hardware, storage capacity increases, and access time to it decreases. On the other hand these devices can benefit from their ability to use data networks right now. This allows them to access non-local computational resources and leads to different kinds of remote rendering solutions. In this case a network connection is established between the mobile client and a remote server and basically, data that is more suitable of being processed on the mobile client is sent to it and vice versa, interaction commands are transmitted back to the server. This idea of dividing computation and visualization tasks to overcome local computational limits is not new. Several attempts have been made in the past. A more detailed discussion of these issues will be given later in Chapter 3.

This thesis offers contributions that are not only related to the area of providing interactive 2D and 3D graphics solutions to mobile devices but even extend further in different areas like interactive illustrative rendering, non-photorealistic rendering for cartoon styles, graphics hardware, and graphical user interfaces. In more detail the contributions can be summarized as follows:

- An overview of current mobile graphics hardware and libraries is provided. Different performance evaluation are made to determine the current state of local rendering processing power.
- A remote rendering architecture is described for the usage for mobile devices. This remote rendering architecture is based on image transfer. It works with arbitrary OpenGL applications requiring no changes in the application code itself only in the execution of the application.
- Different performance improvements for image-based remote rendering are discussed and evaluated together with the remote rendering solution. The amount of image data and the corresponding network bandwidth is identified as the main bottleneck for mobile remote rendering. Therefore, different techniques to reduce the amount of data are evaluated including the usage of different rendering techniques taken from the traditional artistic and illustrative rendering. It is shown that a reasonable usage of them can lead to further reduction of image data without increasing the effort in image compression techniques.
- An alternative solution for providing interactive 3D graphics for mobile devices is introduced. It can be classified as a hybrid remote rendering solution and leads to more interactive frame rates but with sacrificing the generality of the image-based remote rendering solution.
- As many real world materials are of transparent nature or at least have the characteristics of being partly transparent, a large part of the thesis is dedicated to techniques for representing transparent and reflective surfaces. The current treatment of transparency in hardware-based scanline rendering is not completely satisfying and often suffers from not being very convincing. More severe, they do not match with the illustrative rendering styles that are discussed for improving the image compression, since in general different illustrative approaches are used. Transparency, however, plays also a significant role in perception and clear representation of scientific and engineering data. Some visualization methods even depend completely on the display of transparency – for example Direct Volume Rendering. In the context of mobile devices transparency can have another major benefit as it allows a more efficient use of display space by overlaying information that would otherwise require more screen space. Screen space is something that can barely be spared on mobile devices. In this thesis several alternative approaches for displaying transparency are introduced for the usage in interactive 3D applications, leading not only to benefits for mobile rendering but also to improvements for other areas of rendering and visualization.
- A new hardware extension is introduced that allows the modification of rasterization positions in the pixel processing stage by the user. This allows several new classes of algorithms and possibilities for both mobile and desktop applications.
- A platform independent 3D Graphical User Interface toolkit is introduced that can run on desktop and mobile devices and helps to increase the information space for mobile user interfaces.

The outline of this thesis is described as follows. First an overview of local rendering methods is given by having a look at the OpenGL|ES and Direct3DMobile rendering pipeline and comparing different implementations of this pipeline concerning performance and therefore their suitability (Chapter 2). This is followed by exploring several remote rendering solutions (Chapter 3). Afterwards strategies and extensions to one of these solutions are discussed in terms of improving general performance by means of different compression methods (Chapter 3.4), decreasing latency by using multi-threading (Chapter 3.4.5), programmable features of modern graphics hardware (Chapter 3.4.6), and how the performance – i.e. compression rates – can be increased even further without much effort by simply using special rendering techniques (Chapter 3.4.3). Suggestion in Chapter 3.4.3 requires to have a look at some modifications on these algorithms for displaying several material classes. Chapter 4 deals exactly with this issue by having a detailed look on different transparency models. The chapter is split into a brief overview of how transparent surface are currently handled in hardware rasterization algorithms (see Chapter 4.1). Then explains different strategies how transparent surface are handled in classical artistic and illustrative rendering and how they can be emulated with the usage of rasterization hardware (see Chapter 4.2 and Chapter 4.4). Chapter 4 ends with a proposal for a new transparency model for realistic rendering that is presented in Chapter 4.5.

Additionally, an alternative remote rendering solutions is presented in Chapter 3.5. Finally the usage of a newly developed platform-independent 3D GUI library kit based either on OpenGL|ES or OpenGL is discussed and it is shown how it can be used in terms of mobile device user interfaces (Chapter 5).

Chapter 2

3D Graphic APIs for Mobile Devices

Until today fast rendering of three dimensional scenes has always relied on the usage of real-time 3D graphics libraries. Realtime 3D graphics libraries are special low or high level APIs designed to take full advantage of dedicated scanline rasterization hardware. This hardware was first included in expensive workstations. In the mid and end of the 90s of the last century, with the decline of workstations and the fast growing market of low cost off-the-shelf graphics accelerators for standard PCs, this situation has changed and now realtime 3D graphics libraries can be found on PCs as well. This process has even gone so far that the evolution of these libraries is nowadays mainly carried by the PC market. In the beginning of this century first efforts have been made to bring realtime 3D graphics libraries again to a new class of devices – mobile and embedded devices.

Before a closer look into the two main competitors of low level graphics APIs is taken, a classification of important concepts and functionality of current graphics hardware is provided that are later used throughout the thesis. At the end of this chapter due to the lack of current hardware support several software implementation of low level APIs for mobile devices are tested to get a notion of the level of complexity they can currently handle.

2.1 Graphics Hardware

Most of all the algorithms and implementations described later in this thesis take advantage of modern graphics hardware. Nowadays one can distinguish between two different types of graphics hardware. The first one is called fixed function hardware and the second one configurable or programmable graphics hardware depending on the level of programmability. Fixed function hardware is the classical form of graphics hardware and it was common until the beginning of this century. Fixed function means that the processing of incoming 3D data to the final image that is displayed on the screen follows a fixed pipeline structure. It can only be influenced by setting a few processing stages or in terms of a rendering API better known as render states. Programmable graphics hardware, on the other hand, provides the application programmer more freedom in respect of influencing this fixed pipeline structure at certain parts in the pipeline by uploading small programs onto the GPU (Graphics Processing Unit). These GPUs can now be abstracted even to some sort of streaming processor where a stream of data are input into the processor operated on a SIMD level and finally a stream of data is output again from the processor. The only major difference between a real stream processor and a GPU that can be seen is, that today's graphics hardware executes small programs where instructions load and store data to local temporary registers rather than to memory. Figure 2.1 shows the current programming model for todays programmable graphics hardware.

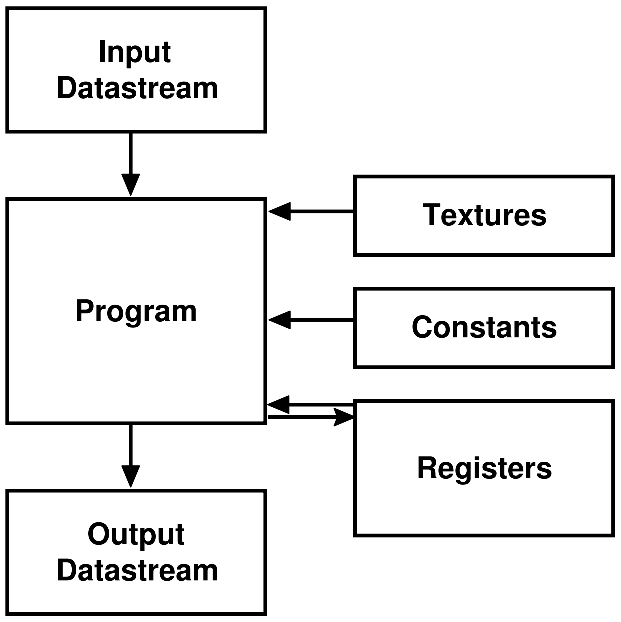


Figure 2.1: Current GPU programming model.

Nonetheless for both types of graphics hardware there is a certain pipeline structure that both more or less follow. The different parts of this pipeline structure are briefly outlined.

2.1.1 Geometry Transfer and Primitives

When displaying 3D content with a low level graphics API like OpenGL first data needs to be transferred from the CPU to the GPU. These data are used to assemble primitives, like triangles, quads, lines, or points. In OpenGL it can be transferred in two different ways. The first one is to send down data for each vertex individually. This is very flexible but also very slow when a large number of vertices is used. Therefore, another method is to send down vertex data chunks and store it in the graphics card memory. This can be achieved by using vertex arrays or vertex buffer objects.

2.1.2 Vertex Processing

Each of the incoming vertices will be processed through a series of various transformations. Traditionally, the transformation of vertices includes typical operations like scaling, translation, and rotation that are performed. Local lighting information is computed using the surface normal and a fixed number of light sources. Finally, the vertices are projected perspectively to the 2D screen coordinates. The flexibility of this approach is severely limited by the few parameters that can be set via render states. These are basically the various transformation matrixes and the parameters of the underlying Phong lighting model. With programmable graphics hardware it is possible to replace this standard processing by an arbitrary user-defined processing via uploading a vertex program to the GPU. However, with these vertex programs it is not possible to break out of the streaming processing nature of the graphics card. More generally, it is not possible to create new or destroy incoming vertices.

2.1.3 Fragment Processing

After vertex processing the rasterization unit converts the geometric primitives into fragments, where each fragment corresponds to a pixel in the final frame buffer that will be displayed on the screen. Fragments are associated with attributes that are calculated through interpolating the outputs of the vertex processing stage such as color, depth, or texture coordinates. The color of a fragment is derived from these attributes. One of the most frequently used and important attributes are textures coordinates. They can be used to look up color or other information in textures. Textures can be seen as a linear ordered memory array on the graphics card that can be classified into different dimensions (1D, 2D, 3D) and different formats (one, two, three, or four color channels with different bit depths, signed or unsigned). With a fixed function hardware pipeline the programmer has almost no influence on the fragment processing. Basically it is only possible through data created in the textures and by different combining strategies between textures if the hardware allows multi-texturing. With programmable graphics hardware this has changed. The color of a fragment can be arbitrarily defined even completely independent of the interpolated data derived from the vertex program by using a user-specified fragment program. The only data of a fragment that are currently impossible to change is its projected screen position and stencil value. However, later in this thesis an extension will be introduced that even allows this feature without changing the general structure of graphics hardware.

2.1.4 Frame buffer Operations

After fragments leave the fragment processing stage they can undergo various tests, like depth test, stencil test, alpha test, and scissoring. If a fragment is not discarded by one of these tests it is written into the frame buffer possibly merging it with the pixel already in there.

If an incoming fragment should not just replace the frame buffer content, blending needs to be performed. Blending is also the only well-known possibility with current graphics hardware to simulate the appearance of transparency. For example the OpenGL specification allows several different types of blending operations that all are basically a linear interpolation between the incoming fragment color and the color already contained in the frame buffer.

As with all other frame buffer operations this part of the rendering pipeline is currently not freely programmable.

2.2 The OpenGL|ES API

The idea to build a small scale and low footprint API for mobile and embedded devices was first set in late 2001. At that time the Khronos group was founded. It was decided to build a low footprint version out of the bigger desktop 3D graphics API well-known as OpenGL. In August 2003 the first specification of OpenGL|ES 1.0 – ES stands for embedded systems – was released. It was later extended in August 2004 to Version 1.1. While version 1.0 is basically derived from OpenGL 1.3 [SA01], 1.1 is derived from OpenGL 1.5 [SA03]. Since it is based on its "bigger brother" OpenGL it follows nearly the same pipeline strategy. Currently in specification phase are OpenGL|ES 1.2 and OpenGL|ES 2.0. The later one should also include support for programmable features as in OpenGL 2.0 [SA04].

When OpenGL|ES was designed, two main considerations were taken into account. The first one is safety in critical environments – like airborne and automotive environments – and the second one is a as small as possible footprint since mobile and embedded devices mainly have a limited storage capacity. Therefore, the OpenGL|ES specification defines two different profiles. They are illustrated in Figure 2.3. The common profile targets a large range of devices that are mostly entertainment related, these can range from PDA, smartphones, consoles, set top boxes to special devices like terminals or kiosks. The biggest difference to the desktop version of OpenGL is that the common profile allows the usage of fixed-point arithmetic throughout the whole pipeline. This design issue was taken as most devices covered by this target range did or still do not have floating-point hardware. Also several functionalities of the core of the common profile has been removed compared to the OpenGL desktop version to minimize instruction and data traffic and to eliminate "un-needed" functionality. Some extended components like OpenGL evaluators, feedback, display lists, and selections have been taken out completely. Table 2.1 summarizes the removed functionality in the three key areas of the pipeline.

The *common lite* profile is focused on devices that have even less resources than the ones for the *common* profile and includes several restrictions. First there is no floating-point anywhere in the pipeline. This means command parameters and vertex data are either fixed-point or integer. It ensures nominally a 16.16 dynamic range in all computations including all transformations. There is no overflow checking or reporting. To shrink the footprint even further it also supports only one form of each command.

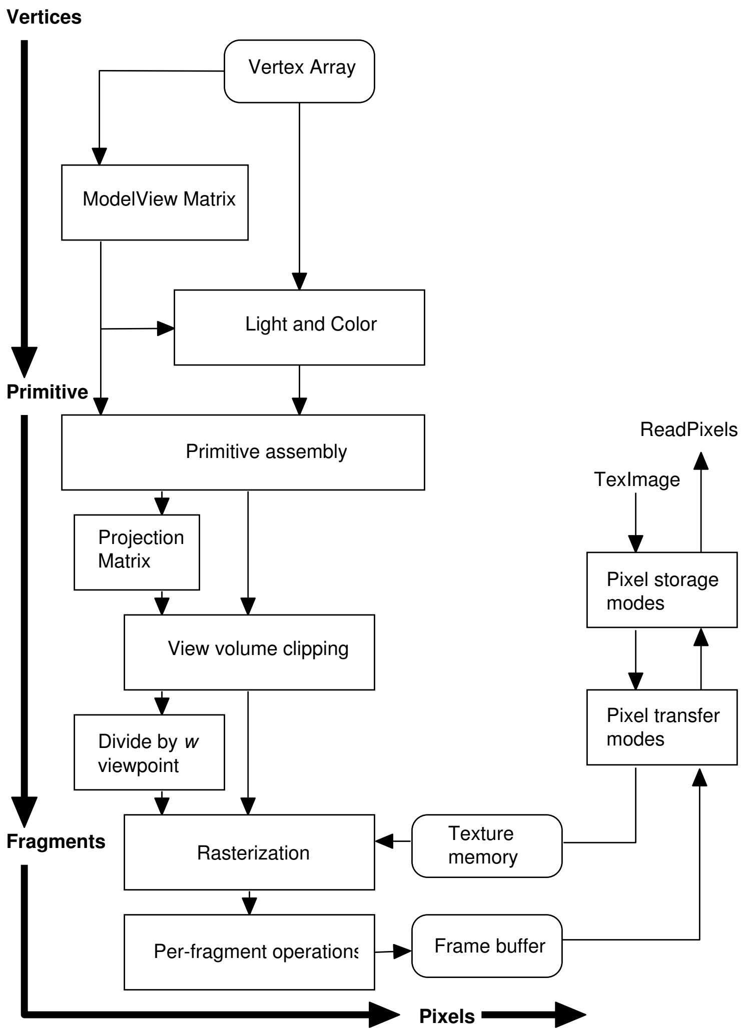


Figure 2.2: A schematic overview of the OpenGL|ES pipeline.

2.3 The Direct3D Mobile API

A few years after OpenGL started to reach desktop PCs, Microsoft started to introduce an 3D graphics API on its own under the DirectX brand for all Microsoft windows operating systems. Until now DirectX is still only available for Microsoft operating systems which lies in contrast to OpenGL that is available on a wide variety of operating systems. Since version 4.2 of Windows CE.NET DirectX – including Direct3D for 3D graphics – was specified as an optional package for the operating system but it was not until Windows CE.NET 5 where a special engineered version for mobile and embedded devices of Direct3D was introduced. This one was simply called Direct3D mobile and similar to OpenGL|ES resembles a small footprint and cut down version of its larger parent desktop version. The rendering pipeline of Direct3D is nearly identical to the one that can be found in OpenGL. The main difference is how the rendering pipeline is accessed or programmed. In OpenGL this is normally done through a set of C-Type functions

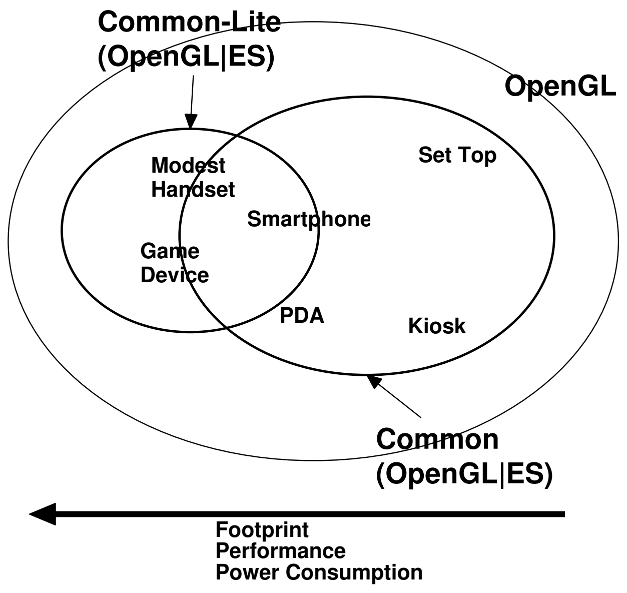


Figure 2.3: Overview of the different OpenGL|ES profile concerning footprint, performance, and power consumption.

whereas in Direct3D everything is done through a set of COM interfaces. This also holds true for both mobile versions of the APIs.

The Direct3D mobile API is roughly based on the version of Direct3D8 with some restrictions in functions and also includes a few features found in Direct3D9. The restrictions of Direct3D mobile compared to Direct3D8 are again similar to the restrictions between OpenGL|ES 1.0 and OpenGL 1.3. Additionally it has no support for any programmable features in the graphics pipeline that are specified in Direct3D8 through vertex and pixel shaders. A second major difference to Direct3D desktop is that it only supports the HAL (Hardware Abstraction Layer) device and not the build-in REF (reference) device driver. And finally the third major difference is that Direct3D mobile allows a fixed-point graphics pipeline.

2.4 Performance Analysis

For some 3D APIs there are already implementations available that run on mobile devices today. Since there is no wide hardware support available these are basically software rasterizers either especially optimized to certain types of devices or just a working class of routines to be used as test bed for applications that will later run on a hardware-accelerated device. In this chapter a look at some of the most well known ones is taken and their performance is measured. Their usage is evaluated for using an location-based visualization tool for different data scenario. Four different test scenarios were picked that provide a good range of possible usage. In the first scenario a middle size 3D mesh is rendered just using standard Gouraud smooth shading with Phong lighting. This should test the render library implementation's ability to display medium

Stage	Removed Functions
Vertex and Geometry Processing	Begin/End Blocks
Vertex and Geometry Processing	Quads, Quad strip, and polygons
Vertex and Geometry Processing	Transpose, Mult, and LoadMatrix
Vertex and Geometry Processing	User clip planes
Vertex and Geometry Processing	Automatic Texture coordinate generation
Vertex and Geometry Processing	Backface material, secondary color
Rasterization	Line Stipple
Rasterization	Polygon Stipple and Polygon Smooth
Rasterization	Polygon mode
Rasterization	Polygon offset
Rasterization	Bitmaps and DrawPixels
Texturing	1D, 3D, Cube Maps
Texturing	Texture clamp and texture borders
Texturing	Combine Environment
Texturing	All image formats except RGBA, packed formats, L and LA
Framebuffer Operations	Accumulation
Framebuffer Operations	CopyPixel
Framebuffer Operations	Draw/ReadBuffer
Misc	Evaluators
Misc	Feedback/Selection mode
Misc	Displaylists

Table 2.1: Functions and features removed from OpenGL|ES in the different rendering pipeline stages compared to OpenGL.

size CAD models on the fly, which is a common usage pattern. The second scenario has less geometry but a lot of textures and a high rasterization load typical for example for volume rendering. The third scenario uses less geometry than the first but multi texturing, a typical scenario for games. And finally the fourth scenario uses simple geometries but many draw calls, similar to a 3D window graphical user interface. As testing machine a PocketPC 2003 device from Toshiba was taken. It has a 400Mhz PXA-263 processor and 128 Mbyte RAM shared between program/data storage and application memory.

2.4.1 Software Rendering Library: Gerbera

Gebera is the name of Hybrid's [Hyb05] OpenGL|ES software renderer. It fully supports the OpenGL|ES Common Profile Specification Version 1.1. A free version of Gerbera is available for non-commercial purposes. It is designed for mobile and embedded devices. It has a portable rasterizer core that runs on most 32-bit processors by supporting multiple different front-ends The system itself is written in Portable C with many portions specifically optimized in ARM assembler. The code for the most time-critical part of the renderer, the pixel pipeline, is even generated on the fly by using a custom JIT (just-in-time) compiler. By doing so it evaluates the active pixel pipeline state and creates routines using an intermediate language. Afterwards a separate compiler performs several optimizations on this intermediate code, for example dead code elimination, constant propagation, register allocation, etc. Finally this optimized pixel pipeline code is processed by a back-end that translates the code into native binary code. It can also perform processor-specific optimizations. According to the authors, the main design goals of the implementation are robustness, stability, and high quality of rendering. The performance of the run-time generated codes should be on par with the best hand-written shader loops. The total code footprint on ARM devices including the OpenGL|ES layer is approximately only 110 kbytes.

Scenario	Rendering time in ms
Many triangles	100
Few triangles, many textures (high overdraw)	1785
Few triangles, multitexturing	260
Several draw calls with two triangles	85

Table 2.2: Rendering performance of Gerbera for the different scenarios.

Table 2.2 shows the measured performance for all four test scenarios. The performance for the first scenario is quite impressive considering that the complexity of the used model corresponded to 11500 gouraud shaded triangles which would lead to a peek performance of around 115000 triangles per second. That is about a factor eight less than what is about to be expected by the first generation of mobile 3D graphics accelerators [Int05a]. The second scenario shows that the software solution is not capable of handling large number of textures and massive overdraw. So applications that use a lot of textures and blending are unlikely to perform well. This definitely rules out for example applications that use texture-based volume rendering – besides the already low memory resources on the device. The performance of the third scenario is slightly surprising as this is a likely scenario for an interactive 3D game title. The scene contains only 1400 triangles and two textures maps. The first one is a sort of lightmap and the second one a surface texture. Both are blended together via multi texturing. Nevertheless, Gerbera is only able to render it with about four frames per second. That is clearly not enough to be considered playable. The last scenario is again handled well by Gerbera showing a 3D GUI system with a few windows and drawable widgets might be feasible even now without any dedicated graphics hardware acceleration.

2.4.2 Software Rendering Library: Vincent

Vincent is an open source non commercial 3D rendering library based on OpenGL|ES. Currently it successfully conforms to the OpenGL|ES 1.0 conformance test. Its main target platforms are PocketPC and smartphones using an Intel XScale PA2xx processor. Vincent is implemented in C++ that maps the C language API calls from OpenGL to the underlying C++ implementation.

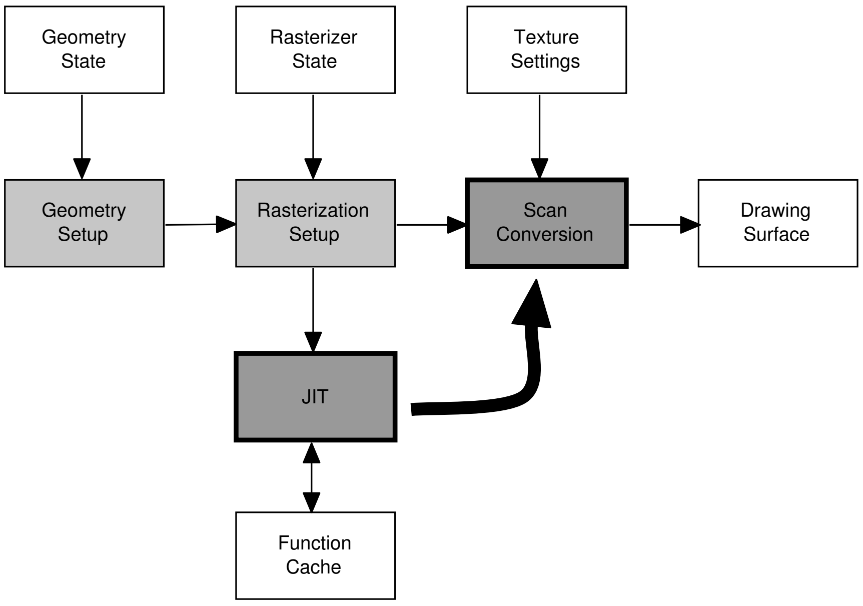


Figure 2.4: Overview of Vincent's runtime code generation method.

Similar to Gerbera it uses a runtime code generation approach to create optimized versions of scan conversion functions based on the current rasterization settings. Figure 2.4 shows an architectural overview of this runtime optimization. In practice each call into glDrawArrays or glDrawElements causes the rasterizer to re-validate and initialize its internal settings before the corresponding sequence of geometric primitives is sent down the graphics pipeline. As part of this re-initialization, the rasterizer employs a code generator to compile an optimized function for the scanline conversion using a JIT compiler that is part of the library. A function cache capturing the most recently used versions of scanline functions ensures that recompilation of function code is kept at a minimum. Table 2.3 shows the performance figures for the same four scenarios as with Gerbera. One thing must be said beforehand that in comparison to Gerbera, Vincent does not run in fullscreen, only in windowed mode and also relies on Windows GDI for rasterization. This might be one of the reasons why the overall performance is between factor of 1.5 – 8 times lower than with Gerbera. But it does not explain everything for example it seems to be incapable of handling simple geometry used in many render batch calls. The JIT Runtime optimization might be the problem here. Thus right now Vincent cannot be recommended for any of the four scenarios.

Scenario	Rendering time in ms
Many triangles	1300
Few triangles, many textures (high overdraw)	2600
Few triangles, multitexturing	360
Several draw calls with two triangles	600

Table 2.3: Rendering performance of Vincent for the different scenarios.

2.4.3 Software Rendering Library: Klimt

Klimt [Wag05] formerly also known as SoftGL although not an official OpenGL|ES implementation it still has a OpenGL|ES compatibility mode and supports nearly the same subset of OpenGL functions as OpenGL|ES 1.0. SoftGL was original developed by VRVIS as part of the Studierstube project [Wie05] – an Augmented Reality toolkit. It has been designed due to the lack of a built-in 3D graphics subsystem on PocketPC. It is a pure software renderer that supports the most important OpenGL API methods for rendering primitives, performing transform & lighting computations, and drawing a video background. The renderer was partially based on existing rasterization code originally developed for PCs without 3D graphics acceleration, but with floating point support on the CPU. The first version was a straight port of this code using compiler-generated floating point emulation. It performed quite well without manual optimizations. Preliminary investigations revealed that performance limitations are due to emulated floating point operations for complex geometry models and due to integer performance in heavy overdraw situations.

Since version 0.4 Klimt has become open source and after version 0.5 the rasterization routines were extended by the option to use Intel's GPP library on the PocketPC. The GPP library provides routines for scanline rasterization and also optimized fixed point arithmetic for devices using Intel's Xscale microprocessor. Table 2.4 illustrates the received rendering times when using Klimt together with Intel's GPP library. Unfortunately only one out of the four benchmark scenarios worked on Klimt. The second one produced unexplainable polygons along with blending artifacts. Further investigations revealed that with the current version – 0.6.1 was the newest

Scenario	Rendering time in ms
Many triangles	405
Few triangles, many textures (high overdraw)	–
Few triangles, multitexturing	–
Several draw calls with two triangles	–

Table 2.4: Rendering performance of Klimt for the different scenarios.

version while writing this thesis – blending does not work at all. The third scenario did not work due to no multi texturing support in Klimt. And finally for the fourth scenario Klimt produced again malformed primitives. Thus a fair testing could not be achieved for these three scenarios. Nonetheless, Klimts performance for the first scenario lies in between what could be observed with Gerbera and Vincent.

2.4.4 Conclusion

Right now software rasterization implementations for the most popular low-level rendering APIs for mobile devices cannot be generally classified as being a practical solution for many types of graphical applications although partially they show very impressive rendering times. Small to medium sized objects with few details and textures are currently manageable at least from a rendering point of view – loading time has not been discussed here. Special sort of applications that only require a few polygons for example in GUI, mockup-views in augmented reality applications seem to be feasible already today. Complex location-based visualization of highly complex and detailed data do not seem to be feasible, unless there is a major pre-processing step to reduce the complexity that allows an interactive viewing on the device itself.

Chapter 3

Remote Rendering Methods

With the advance of todays digital communication networks and the fact that today nearly every computational device seems to be connected to some sort of network a new paradigm of computing has been opened. This paradigm allows devices not only to access its local resources for executing tasks but it is also possible to access any sort of resource on a remote machine for solving these tasks as well. The concept of a distance-based access of computing resources can be applied for visualization or rendering tasks as well.

3.1 Classification

In general when speaking of remote visualization, rendering or distributed graphics today it is possible to classify them into three broad categories:

3.1.1 Render Remote

Render remote solutions are typically image-based where the rendering is performed on a powerful rendering server and afterwards the final generated image is sent over the network to the end user's machine. This solution allows very thin clients without requiring that they have any special hardware support. Furthermore, it can ensure more security since it is not necessary to transfer possible sensitive data from servers to client, e.g. patient data in medical applications, but only some image data which can additionally be encrypted. On the other hand with a large render viewport the network bandwidth will soon become the limiting factor of this solution. For example a viewport size of 1024x1024x24 bit and a rendering speed of 10 frames per second already produce potentially 30 megabytes per seconds of uncompressed image data. This is only manageable with a high speed network. Also the latency of the application will be increased by the latency of the network. Both of these factors can have a serious negative influence on the interactivity of a remote application.

3.1.2 Render Local

Render local solution means that the raw data that should be rendered lies on a distant machine and is transferred locally to the user's machine and rendered there. In this case the geometry is replicated on the local device. This can either be done before an application starts or downloaded just before it is needed. However, several severe problems constrain the usability of this method. Network throughput and the amount of data are mainly responsible for long download times. Under certain circumstances a user has to wait terribly long before the first image will be visible on the device. This makes many interactive applications completely useless. Another problem might occur when the local machine simply has not enough resources either to store the raw data or even to render it fast enough to ensure an interactive workflow.

3.1.3 Shared Visualization or Immediate-mode Rendering

Shared visualization tries to combine features of both render local and render remote in the hope that it will eliminate the problems of each other and only the benefits will remain. The main idea is that remote and local resources should collaborate and negotiate, combining capabilities to produce the final image on the end user device. The most common form of shared visualization is probably immediate-mode rendering. In immediate-mode rendering low-level drawing commands used by the underlying graphics API are issued by the application performing the rendering, but these are not immediately executed. Instead they are sent over the network as a kind of remote procedure call. The actual rendering is then performed by the local device. This functionality is for example supported in the Unix Operating system with the usage of GLX and OpenGL. When executing a render application on the remote machine and rerouting the output display to a local device which runs Unix with GLX support as well, this will automatically happen. Immediate-mode rendering works best when the local device can use all its resources just for the rendering process. It has the benefit that it does not necessarily need to store the whole raw data on the local machine. On the other hand it suffers from the same problem as local render when the amount of data to render is too large to ensure interactivity. It is also very hard to predict the necessary network bandwidth since it depends largely on the installed underlying graphics API. Additionally, progressive techniques, e.g. progressive meshes [Hop96], can be classified as a different form of immediate-mode rendering. Progressive techniques always start with a very basic representation of an object that is slowly refined over time depending on how many resources are available on the client machine. The different representation forms however are generated and stored on the server and transmitted to the client.

3.2 Different Render Remote Solutions

When looking at the capabilities of a mobile device like a PDA or smartphone right now it is rather easy to conclude, at first glance, that there is basically only one practical method for distributed rendering on these kind of devices at least when the dataset that should be displayed interactively has a significant size. Render local can be excluded since there is neither enough memory or storage capacity nor computational resources for doing so. Some sort of shared rendering might be possible, but not in a simple form of immediate-mode rendering as even with a graphics library like OpenGL|ES or Direct3Dmobile on the mobile machine this is not possible since both APIs do not support this feature. This leads to the last option of render remote that will be focused on in the upcoming sections.

As there are several published image based render remote solutions a short overview is given in this chapter to compare them with the proposed solution given in the subsequent chapter. The most straightforward way of remote visualization can be accomplished by using the remote-display capabilities that are a fundamental part of the X Window System [Nye95]. This X-based remote visualization system is illustrated in Figure 3.1.

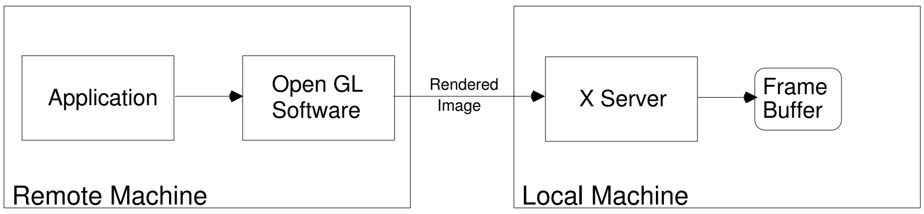


Figure 3.1: Remote visualization using the standard X display forwarding mechanism.

Here, a visualization application runs on the remote machine while a OpenGL software library (MESA) provides all graphics rendering. This OpenGL library produces – using software rendering routines – a sequence of images which is transmitted to the local machine for display. For transmission the standard X protocol is used. This means the images are transmitted individually, frame-by-frame using a sequence of XPutImage commands, that simply transmit the image pixels in an uncompressed, raster-scan fashion. Although this is a completely application transparent solution there are several drawbacks. First it does not take use of graphics hardware acceleration installed in the system since all rendering takes place in software. Second, because the communication takes place via the X Protocol, no compression is applied to the transmitted image data. As a result, the bandwidth required for the communication data and link is relatively high.

Engel et al. [ESE00] proposed a different approach for an image based render remote solution by exchanging the application transparency with taking usage of both hardware accelerated graphics on the server and compressed image transfer. A schematic overview of this approach is illustrated in Figure 3.2. On the server side an OpenInventor or Cosmo3D based application is executed and renders either images on-screen into the frame buffer or off-screen into a pbuffer. On the client system a Java application enables transparent remote access to this server from any window system supporting platform. The received image data sent from the server is decoded, stored as Java2D buffered image and drawn into the frame buffer. All events like mouse and keyboard from the client are transported to the server machine using CORBA methods calls. The server provides an Open Inventor or Cosmo3D interface for these events and passes them to the corresponding functions. Since the system has complete control over the image transfer it allows

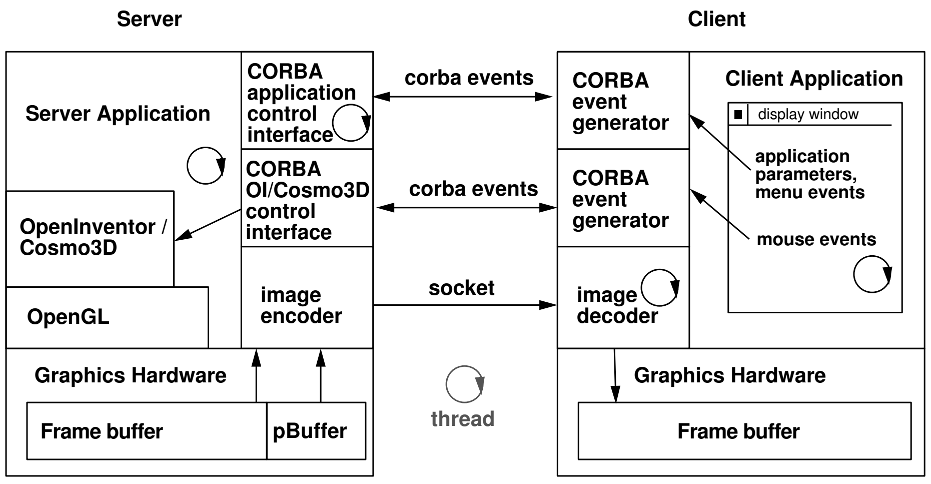


Figure 3.2: Overview of the remote rendering architecture as described by Engel et al. [ESE00].

arbitrary compression schemes. Moreover, the authors investigated several lossless compression algorithms and integrated them into their system.

Ma and Camp [MC00] also proposed a special purpose render remote system for the visualization of time-varying volume data on a parallel computer. Their framework consists of three parts: the display daemon, the render interface that provides each rendering node with image compression and communication to and from the display daemon, and the display interface providing basic functions like image decompression, image assembly and communication to and from the display daemon. The display daemon itself passes images from the renderer to the display and allows the display to communicate with the renderer. Ma and Champ also investigated several compression methods like MPEG, JPEG, JPEG2000, BZIP and LZO.

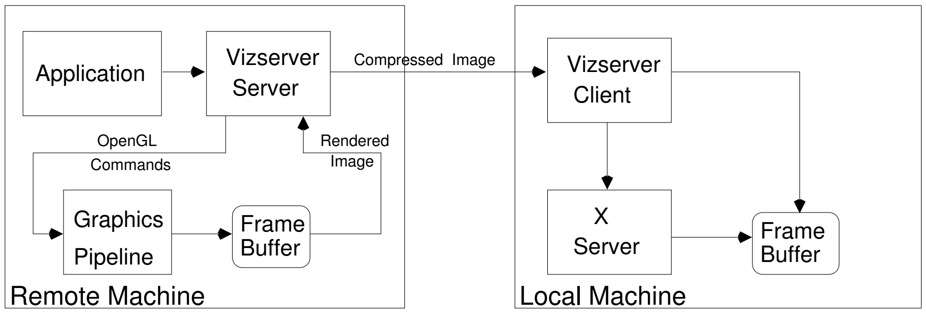


Figure 3.3: Overview of the SGI Vizserver remote rendering architecture.

Sillicon Graphics Inc. even developed a commercial solution called Vizserver [Sil05] for image-based remote rendering. It is designed to provide users with remote access to the graphics pipeline of a Onyx Infinite Reality machine. The user is able to run a rendering application remotely, while the Vizserver captures the rendered imagery output from the application and transmits an image sequence to the local machine for displaying. An important aspect of the

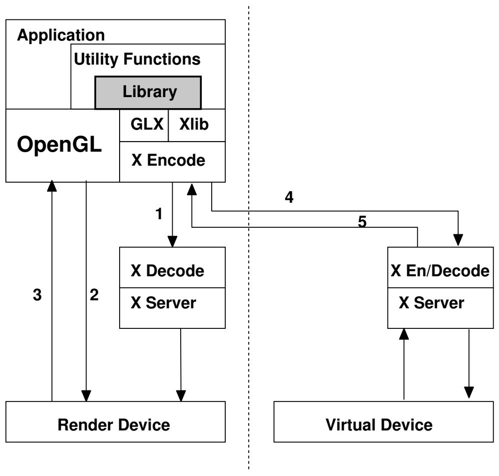


Figure 3.4: Overview of the GLX architecture.

product is that the operation of Vizserver is completely transparent to the application. A diagram of the Vizserver system is illustrated in Figure 3.3. As can be seen from the figure, applications are completely unaware of the presence of Viszerver since all OpenGL commands employed by the application are intercepted and directed to a remote graphics pipeline. Afterwards the server captures the pipeline output from the frame buffer and transmits the resulting images to the Vizserver client. This client is a specially written client and an implementation exists nearly for all major platforms including some mobile device operating systems. Additionally, it has some build-in compression algorithms to reduce the bandwidth required by the network. These are Color Cell Compression (CCC) and Interpolated Cell Compression (ICC). Both, CCC and ICC, are compression schemes designed to provide low latency and require very low computation overhead. They are based upon color-quantization variants of BTC (Block Trunc Coding).

Similar approaches recently also appeared from other classical workstation companies like IBM. IBM's solution is called Deep computing visualization [IBM05] and supports besides what IBM refers as scalable visual networking (SVN) also remote visual networking (RVN). Unfortunately further details on the underlying architecture and especially what compression techniques are used are not provided.

3.3 An Image-based Remote Rendering Solution for Mobile Devices

The remote visualization system used as basis for this work was originally described by Stegmaier et al. [SME02]. Its core idea can easily be recognized by considering the way OpenGL-based applications generate images when the display is redirected to a remote host in a X-based windowing system (see left side of Figure 3.4). All OpenGL commands are encoded using the GLX protocol and transmitted to the remote host as part of the well-known X11 protocol stream. The stream is afterwards decoded by the receiving X server and OpenGL commands are executed exploiting the resources available at the destination hosts. This is what was earlier described as immediate-mode rendering. But this is also exactly what is not desired since current and probably future mobile devices do not support this functionality. Thus, the transmission of OpenGL commands must be suppressed and only the rendered images should be transferred to the host where the user interactions occurs.

Surprisingly, the required modifications of the GLX architecture can be obtained through a single shared object linked to the visualization application at program start-up.

3.3.1 Dynamic Linking

Since graphical applications have to link to a graphics library when accessing graphics hardware functionality, it is not necessary to change each application to have certain functionality. Instead it is also possible to change the library. Library functionality is provided in two formats: as static libraries or as shared libraries. It is up to the programmer to select a suitable format. For shared libraries only a link to the actual functionality is included in the application's executable. Thus, dynamically linked applications allow a replacement of library functionality without relinking. This is necessary for libraries that have functionality which is not portable between different target platforms, e.g. OpenGL.

Most systems that support load-time linking also support a second mechanism sometimes referred to as preloading. Preloading allows to manipulate the stack that is used to resolve function references in dynamically linked applications, usually by setting an environment variable to an object file that redefines the functions to be overridden. Therefore, preloading presents an easy method for non-invasively customizing library behavior. However, completely redefining library functions is not trivial – in particular if the function performs complex tasks.

To cope with this problem, many systems also support run-time linking. With run-time linking, symbol resolution is performed after the program has started instead of during the loading phase. The original functionality can then be recovered by opening the original shared object and determining the address of the relevant function from its name.

3.3.2 Library Architecture

Using the described dynamic linking functionality, a generic remote visualization solution (see Figure 3.5) can be obtained by preloading library functions from two groups.

- Set-up functions like *XOpenDisplay*, *glXChooseVisual*, and *glXMakeCurrent*
- Trigger functions like *glXSwapBuffers*, *glFlush*, and *glFinish*.

The set-up functions make sure that the stream of OpenGL commands is directed to the render server, the trigger functions determine the point of time the rendering has finished and an image

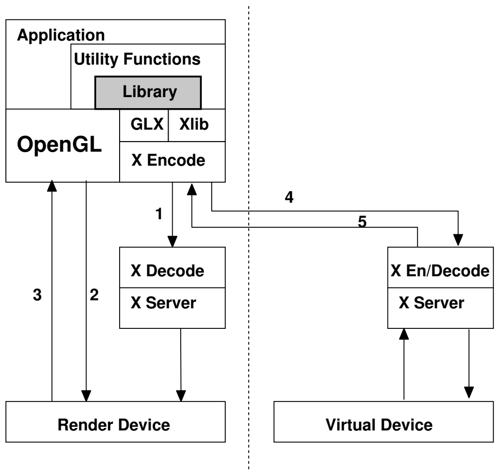


Figure 3.5: System architecture as described by Stegmaier et al. [SME02]. (1) The application issues a GLX request which is sent to the render server. (2) The application issues OpenGL calls, which are handled by the render device. (3) The library reads the contents of the frame buffer and (4) sends it to the interaction server using *XPutImage* request. (5) *XEvents* are sent from the interaction server to the application.

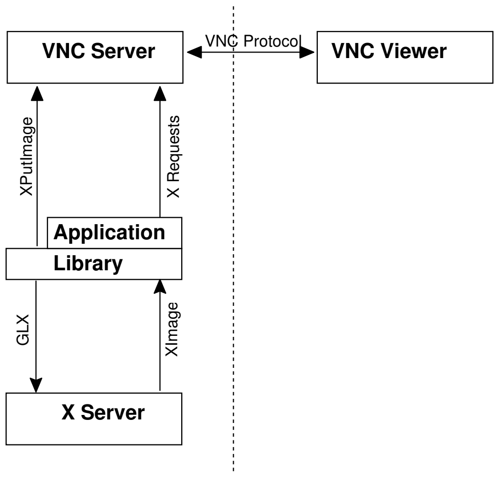


Figure 3.6: System architecture from Stegmaier et al. [SME02] used together with VNC.

update must be sent. If an image needs to be sent, the image is read from the frame buffer with *glReadPixels* and transmitted via *XPutImage* with the display set to a virtual X server.

3.3.3 Evaluation of the previous Solution

The main advantages of this solution are: first, it is application transparent, this means it works with any OpenGL application without rewriting the application. Second, it is simple and can be implemented using a single software component. Therefore, reducing the coding efforts to a minimum, since the handling of the user interface is done completely by the regular X Window system. On the other hand there is a significant limitation with this approach, especially in low bandwidth networks like mobile networks since the core X protocol does not support any compression schemes. There is an extension orientated towards low bandwidth environments called LBX [CFK$^+$96]. For LBX the X consortium adopted several compression algorithms. The most common one is the stream compressor which is an implementation based on the ZLIB [GA05] Library. Additionally, there is a bitmap compression specified and defined by the CCITT Group 4 2D compression algorithm [Int88]. However, by using LBX in a local network no real performance gain could be experienced although a lower network traffic has been observed. The compression algorithm seems to take more time as can be gained by the lower network traffic. This ratio might of course change with extremely low bandwidths like GPRS or ISDN lines. Another way of obtaining stream compression on top of X is to use VNC [RSFWH98]. VNC is a free multi-platform client-server application for displaying and interacting with remote desktops. In contrast to the X protocol, VNC is only capable of sending rectangular frame buffer updates to the client. VNC also provides a variety of simple and specially designed compression algorithms to make this transmission as effective as possible. The Unix variant of the VNC server is based upon a standard X server, which means that on one hand it can communicate with clients using the X protocol but on the other hand it does not implement any X protocol extensions. Figure 3.6 shows how VNC can be used together with this approach. However, using VNC unfortunately embodies some problems. First, since it splits the viewport into rectangular areas for fast updates some rectangular blocks are updated asynchronously. Also with the current implementation of VNC it is hard to trigger it to synchronize with the displaying speed of the VNC client and not to run with full speed. The visual effect that can be recognized by this behavior are jumping frames. This is all related to the fact that VNC was intended for remote controlling graphical user interfaces and not for remote controlling interactive 3D graphical applications. Another drawback is that the integrated compression schemes may be both suboptimal in environments of very low network bandwidth (the compression ratio may be too low to allow interactive work) and very high network bandwidth (where the compression times may be the limiting factor).

3.4 Improvements and Accelerations

From the results given in [SME02] it is noticeable that already at screen sizes beyond 512×512 the frame rates drop rapidly. The situation gets even worse when dealing with slow networks that are very common in mobile computing scenarios like Wireless LAN 802.11b (still the most common network) and Bluetooth or UMTS which have data connection speeds of usually around 1-11 Mbps. This leads to the conclusion that a remote visualization architecture suitable for all kinds of network environments must be able to use arbitrary compression algorithms. This is most easily accomplished by setting up a dedicated data channel (e.g. a TCP/IP connection) for the image transfer and by providing a client application capable of decompression received data (see Figure 3.7). Compared to the previous architecture, the required modifications are only minor – basically, the only difference is that now *XPutImage* is called by the client application

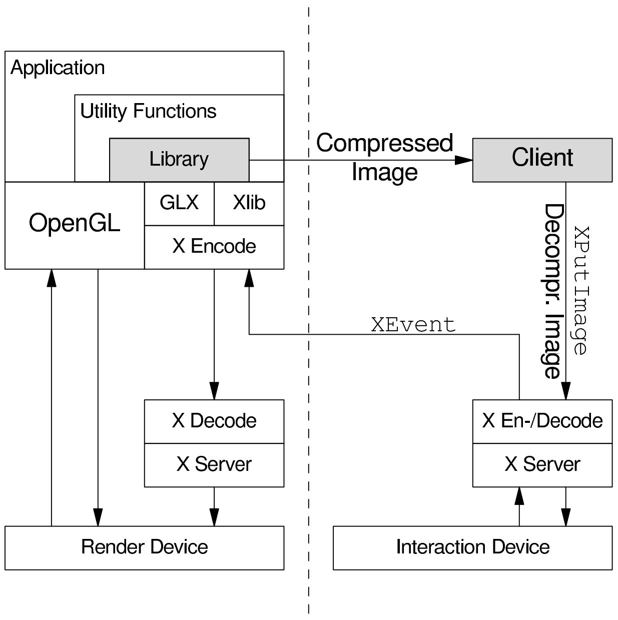


Figure 3.7: Revised two-component architecture of the generic remote visualization system with custom image compression.

instead of the remote visualization library. Once the image data have been received by the client program, the data must be decompressed and drawn into the appropriate window. Since this is already accomplished in the previous implementation, the window identifier is known and can be sent to the client as header information. The following section presents the results of joint work with Simon Stegmaier and Manfred Weiler. It was first published in [SDWE03].

3.4.1 Data Rate Reduction

A basic idea for reducing the data rate without much effort is simply to reduce the amount of data created on the server. Some applications do not need to have a true-color visual – 16 bit or even 8 bit color depths might be sufficient. However, this only helps to reduce the data amount by a factor of two to three which is not enough in most cases. Therefore, avoiding redundant information (from a coding theory's point of view) is more appropriate. Exploiting redundancies for creating a more compact version of the original data is the approach of data compression algorithms.

3.4.2 Software Compression

The field of compression and also image compression is well developed. Continuing progress is being made due to the demand for higher compression performance from an increasing number of application areas. Before deciding which compression algorithm to use; it is helpful to have a look at which criteria are important in a remote visualization environment especially when focused on mobile devices:

- Low cost meaning both short compression and especially decompression time to not increase the latency further than necessary.
- Good image quality.
- High compression rates for fast image transmission even over slow networks.

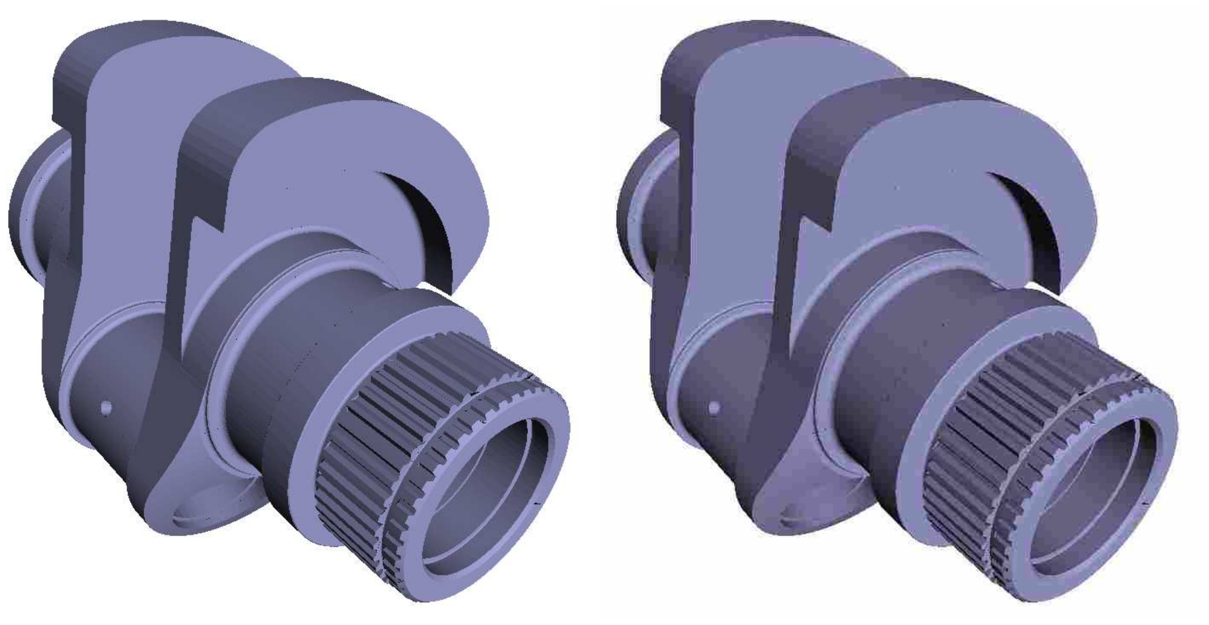


Figure 3.8: Visual quality comparison between a jpeg compressed rendered image (right) and the same image uncompressed (left).

JPEG-2000 [Joi05] is becoming a very popular standard for still image compression since it provides significantly lower distortion with the same compression ratio than for example JPEG. On the other hand, the first requirement eliminates the usage of JPEG-2000 since it is based on wavelet transform and therefore requires a relatively complex computation. It also needs an additional amount of memory which might not be available. Since remote visualization deals with synthetic images, JPEG and other discrete cosine transform algorithms do produce noticeable compression artifacts (see Figure 3.8) unless the build-in quality setting is set at such a high level where the main benefit, the compression ratio, is no longer holding up. Another well-known standard for encoding moving pictures is MPEG [Mov93]. MPEG, which is good for compressing existing videos, is not suited for the interactive setting of remote visualization in which each image is generated on the fly and to be displayed in real time. While using MPEG is not completely impossible, the overhead would be too high to make both the encoding and decoding efficient in software.

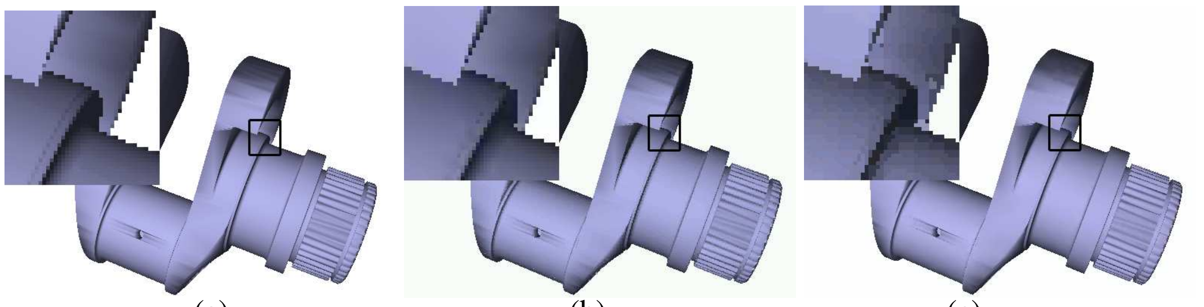


Figure 3.9: Visual quality comparison between different compression techniques. (a) lossless lzo, (b) block-based CCC, (c) transformation-based BTPC.

At the end, the revised remote visualization system includes five different software compression algorithms based on three different groups of compression techniques: ZLIB [GA05] and LZO [Obe05] for lossless compression. ZLIB's compression method is a variation of LZ77. LZO is also a variation of the block-based LZ77/LZ78 compression algorithm. But the main design of LZO was compression and decompression speed which are particularly fast and therefore especially suitable in realtime environments. The third algorithm is called Color-Cell Compression (CCC). It is a simple compression technique based upon a color-quantization variant of BTC (block truncation coding). It has the advantages of requiring very low computation power with a small latency that does not depend on the image. It also achieves a fixed compression ratio of 8:1 and, as previously mentioned, is integrated in SGI's Vizserver. The fourth algorithm is a combination of CCC and afterwards additional encoding with LZO (CCC_LZO). And finally, BTPC (Binary Tree Prediction Coding) is a lossy compressor which on average achieves better performance than JPEG but less than JPEG2000. It uses a binary pyramid, predictive coding, and Huffman coding. BTPC can also be applied losslessly where it is rarely less than 5% worse what the best PNG, GIF, or JPEG-LS can achieve. These five different compression methods are evaluated by examing the compression ratio, the achievable frame rates with different network and client configurations.

	WLAN		Ethernet	
Compression	Ratio/FPS		Ratio/FPS	
none	–	1.0	–	20.7
LZO	2.6-8	5.9-8.9	3.4-17.3	33.4
ZLIB	5.5-16	7.1-8.4	3.0-18.9	17.6
CCC	8	8.4	8.0	32.0
CCC_LZO	12-47	8.5-11.6	17.3-68.1	31.4
BTPC	22-44	3.5	47.3-110.5	11.6

Table 3.1: Frame rates for different scenarios. Left: 802.11b WLAN, right: Fast Ethernet.

Table 3.1 shows the results measured for two different client/server and network configurations. Additionally, Figure 3.9 shows comparison images for the CCC and BTPC compression. For the Ethernet scenario a Gigabit Ethernet was used. The server machine was a Pentium IV 2.8 GHz with a Geforce4Ti4600. The client machine was also a Pentium IV 2.8 GHz with a Geforce4Ti4200. The viewport size was 512×512 with 24 bit color depth. For a Gigabit Ethernet compression – at least for these viewport sizes – does not offer much benefit. The benefit here is fast compression/decompression times instead of high compression ratios. For the WLAN scenario the client was exchanged with an iPAQ 3850 PocketPC running Familiar Linux distribution using an 11 Mbit Wireless LAN. The viewport size was 240×320 and 16 bit color depth. Here compression ratios have more influence on the overall performance except for BTPC which takes too long for compressing and decompressing. Interesting to note is that the lossless compression algorithms are basically nearly on par with lossy compression algorithms in both scenarios, but they have the benefit of higher image quality. Additionally a test with JPEG compression has been conducted using libJPEG. To achieve an equal framerate to the lossless compression algorithms in the WLAN case a quality setting of 10 out of a range between 1-100 had to be set. The visual quality is comparable with what can be seen in Figure 3.8.

3.4.3 Using Different Rendering Styles to Increase Compression Ratios

Another idea to decrease the amount of data that is required for transferring the images to the client is to reduce it already while rendering the image on the server. As most mobile devices have a small display area and a limited resolution and color depth, they cannot benefit from high detail rendering in the first place. It might be better to just concentrate on the most important features of a 3D scene and remove details that normally cannot be seen under these conditions. This follows Tufte's [Tuf97] strategy of the smallest effective difference: "Make all visual distinctions as subtle as possible, but still clear and effective". It can clearly be seen that the artistic and non-photorealistic rendering field provides an excellent domain for following this strategy. Artists and illustrators know exactly on which parts one should concentrate while creating their imagery. Not only do non-photorealistic rendering techniques often increase the information density significantly without increasing the required amount of information space, but often they are a better method to transmit meaningful, non-ambiguous, or even additional information in an image compared to photorealistic methods. That is for example one major reason why they are widely used in technical manuals or medical textbooks. Of course there is also the additional aesthetic and artistic value.

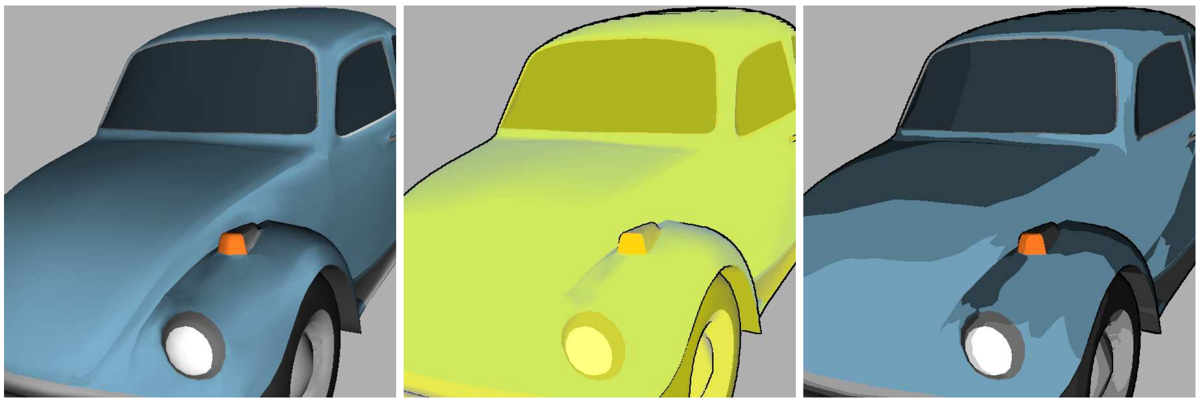
For choosing an appropriate rendering it is reasonable that following criteria should more or less be fulfilled:

- The shape of objects should be recognizable.
- The object shading, material & lighting information should stay intact.

- The color information of objects should be maintained to a certain extent.

- The implementation of the rendering style on the server should not be very complex.

After a short survey we came to the conclusion that some illustrative styles often used in the area of technical illustrations seem to be most suitable for fulfilling the criteria catalog above. In particular cool & warm shading and cel style-like shadings were investigated.



(a) (b) (c)

Figure 3.10: Example scenario taken to compare different data sizes. (a) per-pixel Blinn & Phong rendering (b) cool & warm shading (c) cel shading.

To test the effectiveness of the rendering style on the reduction of the data amount a test scene was taken and rendered using three different rendering styles. In the first case a standard per-pixel Blinn & Phong lighting was used. In the second case a cool & warm shading according to Gooch et al. [GGSC98] was implemented and finally in the third case a hard-shading algorithm for cel shading proposed by Lake et al. [LMHB00] was chosen. Figures 3.10(a)-(c) show the corresponding results. Without using a particular data compression algorithm the amount of data is of course the same for all three generated images, since they have the same amount of pixels and color depth. But the compression factor should differ at least for lossless compression algorithms that according to the previous section are the best choice when considering trade-offs between compression/decompression time, compression ratio, and received image quality.

Table 3.2 shows the different data sizes of the generated 600x600 pixel wide test images using the LZO compression algorithm. As can be seen from the figures in the table, by simply using a cel-style renderer the compression rates can be increased by a factor between eight to nine. And even for the cool & warm shading an increase of a factor of four and above can be achieved. Additionally, the influence of rendering feature lines along with the rendering styles on the data compression was evaluated. When looking at the storage figures for both cases it can clearly be seen that the impact of having feature lines in the image can be considered as negligible .

Type	Size in bytes
Uncompressed	1.080.000
Per-pixel Blinn & Phong rendered LZO compressed	147.358
Cool & warm shaded LZO compressed	30.496
Cool & warm shaded with lines LZO compressed	29.296
Cel shaded LZO compressed	16.316
Cel shaded with lines LZO compressed	16.962

Table 3.2: Comparison of different image data size when using different rendering styles and lossless compression.

3.4.4 Motion Handling

Another idea to reduce the image data even further, is to exploit the human's eye insensitivity to details of moving objects by adjusting the quality parameter of the compression algorithms or downsampling the image to a quarter of its original resolution during user interactions or animations. The server can employ heuristics to automatically detect these situations considering the average number of buffer swaps per time unit. The image is scaled back to its original resolution by the client. Of course, downsampling leads to a definite loss in image quality. Therefore, when the user stops to interact with the application or the current animation is stopped, a high quality version of the last generated image is sent to the client. This approach reduces both the rasterization load of the server and the raw image data to a quarter. Applications that are rasterization bound will, therefore, profit from this solution in double respect.

3.4.5 Latency Reduction

Remote visualization using the described approach involves phases of high CPU utilization as well as phases of high GPU utilization. Four major phases can be identified: image rendering, frame buffer read-out, image compression, and data transmission. Provided that the rendering phase makes extensive use of advanced OpenGL features like vertex objects or display lists and assuming that the frame buffer read-out can be done with a DMA operation, these two phases can be classified as *GPU-dominated* work. The later two *CPU-dominated* phases (compression and transmission) are using the CPU exclusively, potentially consuming resources required by the graphics card driver and consequently slowing down the rendering. This can be alleviated given a suitable parallel architecture. The solution is based on POSIX pthreads and uses two semaphores for synchronizing the exchange of image data. Using a dual-processor AMD Athlon MP2200+ PC as render server, a frame rate increase of about 42% (from 137 FPS single processing to 195 FPS with dual processing) could be experienced. However, parallel processing is just one means for improving latency. An alternative is to reduce the compression time.

3.4.6 Hardware Compression/Decompression

In Chapter 2.1 modern GPUs have been described as a sort of streaming processor but one has to remember that currently in this stream it is not possible to arbitrarily reduce or enlarge the number of input data compared to the output data, only by using some tricks a fixed reduction dimension is possible. This makes it nearly impossible to implement certain algorithms and especially it makes it nearly impossible to implement compression algorithms based on RLE, LZ77/78, or Huffman encoding. Additionally, the architecture is also not very suitable for complicated algorithms where a high random memory access pattern is required or can occur, since this will destroy the performance of the SIMD. This excludes as well compression algorithms based on transform coding (DCT or wavelet) like JPEG or JPEG2000. Additionally, when the decompression should be executed on hardware it should be fast and inexpensive.

Using built-in texture compression

Before using the programmable features of the graphics card one can try to use the already available options a graphics adapter offers. A nice feature of almost every graphics adapter since the introduction of DirectX 6.0, is to handle compressed textures. Texture compression originally intended to reduce the cost of storing textures in the graphics adapters main memory and therefore reducing the costs of RAM could be used for the image compression part in remote visualization as well. The advantage is the very fast and simple decompression scheme since it has to be built into the hardware itself. Also it is very likely that mobile graphics hardware will certainly have these feature as well since they have even higher constraints concerning RAM usage, chip space, and energy restrictions than graphics adapters for desktop PCs. Texture compression and general image compression have many attributes in common but according to Beers et al. [BAC96] there are four factors that should be especially considered when evaluating a texture compression scheme:

- Encoding speed: With the exception of dynamic maps, the majority of textures are prepared well in advance of the rendering. It is therefore feasible to employ a scheme where encoding is considerably slower than the decoding.
- Decoding speed: The accessing of texture data is a critical path in the texturing operation and so the speed of decode is of paramount importance. The decode algorithm should also be relatively simple to keep down the cost of hardware.
- Random access: As objects may be oriented and obscured arbitrarily, it is therefore a requirement to be able to quickly access any random texel
- Compression rate and visual quality: Because the important issue is the quality of the rendered scene rather than the quality of individual textures, some loss of fidelity in the compressed texture is tolerable. Lossy compression schemes offer higher levels of compression than their lossless counterparts, and so the former are generally preferred.

The encoding speed factor lies a bit contrary to the requirements of a remote visualization system as compression should be as fast as possible. On the other hand it is very likely that the server has more computation power compared to the mobile client. Therefore, different texture compression methods were evaluated to analyze their feasibility of their usage in remote visualization applications.

The most common and popular method was introduced by Iourcha et al. [INH] and is nowadays better known in the industry as S3TC^{TM} (or DXTC^{TM} within Microsoft's DirectX 3D interface). It is based on BTC like CCC and also uses 4x4 blocks. As with BTC, each block is completely independent of every other block. Each pixel in this 64 bit block is encoded with two bits which select one of four colors. Two of these colors are stored in the block as 16-bit RGB values while the remaining pair are derived directly from the stored colors. These additional colors are usually 1:2 and 2:1 linear blends of the main representatives. How these two reference colors are found is not further described in the algorithm and it is up to the implementation of the compressor to make this decision. The achievable compression rate is in case of 24 bit RGB values in the original image 6:1. To validate its usage a standalone test was conducted using the DirectX 3D API. The API includes an optimized version of a software DXTC compressor. This compressor can be accessed by simply transforming a texture surface to a compressed texture surface through the provided extension library call *D3DXLoadSurfaceFromSurface*. The achievable frame rates however were disappointing even on a Pentium IV 2.8 Ghz with 2 GB RAM and ATI Radeon 9800XT graphics board. The maximum achievable frame rate was 40 FPS for a $512x512$ size image. This is far below what could be achieved with a software CCC compressor (see Table 3.1). On the other hand decompression is done without any speed penalty since it is completely executed on the graphics hardware. S3TC is therefore a very asymmetric compression algorithm. A recent published texture compression algorithm by Ström and Akenine-Möller [SAM04] that is focused on an implementation on mobile devices or low-cost hardware has already found its way to mobile graphics chips [Oy05]. However the authors noticed that the fastest compression method for this would take 30 milliseconds for a 64x64 size texture on a 1.2 Ghz machine. So this approach seems to be very unpractical for remote visualization.

Implementing a hardware compressor and decompressor on programmable graphics hardware

Since there is a large asymmetry between the compression of texture compression algorithms and their decompression which is available in hardware, another idea is to use the programmable features of the graphics card and implement a compression algorithm. When looking at the previous choice of compression algorithms, the CCC compression proposed by Campbell et al. [CDF$^+$86], seems to be best suited for an implementation on hardware, compared to S3TC it has a better compression rate (8:1), and probably needs fewer scalar instructions. Besides an expected reduction of the compression time, a hardware-based compression has the advantage that only a fraction of the original data have to be read from the frame buffer, making expensive frame buffer reads more bearable.

Understanding the hardware compression requires knowledge of the CCC algorithm. The CCC

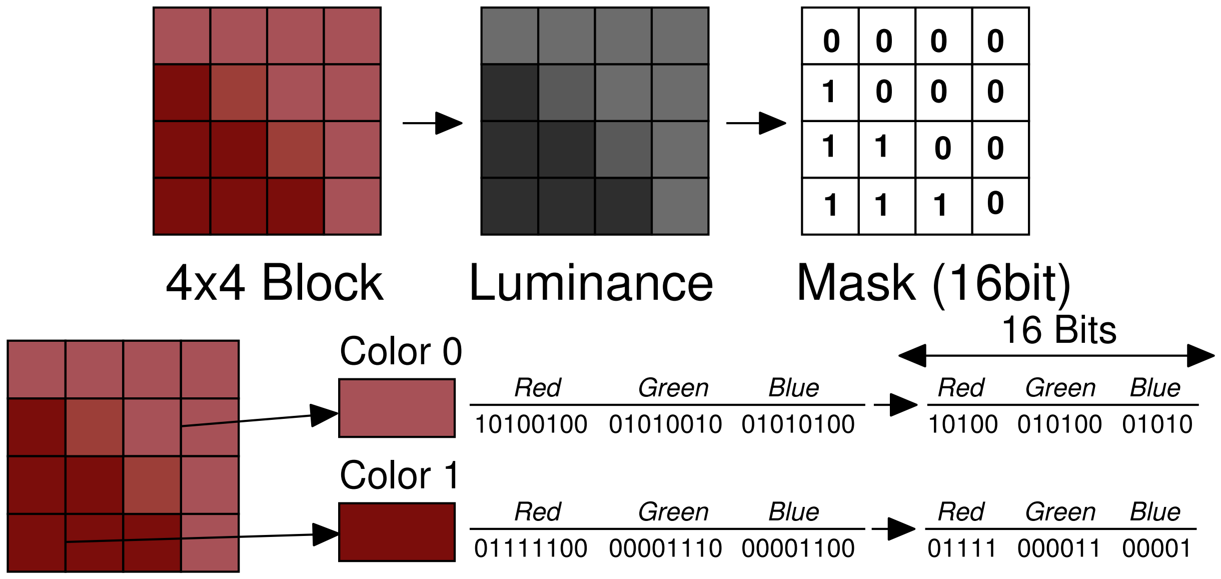


Figure 3.11: Visual explanation of the CCC algorithm. Upper part shows how the bit mask is generated. Lower part shows the encoding of the two colors for each 4x4 pixel block.

algorithm first decomposes the source image given in a 3 bytes/pixel RGB format into independently processed cells of the size of 4x4 pixels. For each cell, a mean luminance L_{mean} is calculated using the well-known luminance equation for RGB color space $L = 0.3R + 0.59G + 0.11B$. Next, all pixels of the cell are divided into pixels with luminance less than or equal L_{mean} and pixels with luminance greater than L_{mean}. A 16 bit bitmask is then created in which pixels with luminance values greater L_{mean} are marked with one (see Figure 3.11). Additionally, two RGB color values are stored per cell which are the arithmetic mean RGB values of the pixel groups. The image is reconstructed from the bitmask and the two colors by setting the first color where the corresponding cell's bitmask is zero, and the second color where the bit is one. Using this approach, a cell can be encoded with $2+3+3 = 8$ bytes. Additionally a quantization of the group colors is applied by storing only 16 bit for the colors in RGB565 format (see Figure 3.11(lower part)), yielding an overall compression ratio of 8:1.

Obviously, the CCC algorithm allows all cells to be processed independently. This matches the architecture of modern graphics adapters with several pixel pipelines operating in parallel very well. The following description relates to an implementation on NVidia GeforceFX graphics card which has been successfully adapted to other architectures as well, these will be discussed in more detail later.

Encoder The hardware-based CCC algorithm renders a quadrilateral with a size of one fourth of the source image in each dimension. Thus, each pixel generated from this quadrilateral maps to one cell of the source image. The color data of the source image is provided by a texture map that is updated in each frame. Since OpenGL allows to efficiently copy data directly from the frame buffer to a texture map, this step hardly involves any overhead compared to the expensive frame buffer read of a software solution. The alternative of using NVIDIA's `render_texture_rectangle` OpenGL extensions [Kil01] that allow the directly definition of a texture from the frame buffer without copying was discarded due to a negligible performance benefit.

To map rendered pixels to cells, a fragment shader is applied that performs 16 lookups in the source image texture for acquiring the cell's color information and for generating the bitmask and the two group colors. The shader implementation is almost a 1:1 mapping of the software encoder since NVIDIA's high level shading language *Cg* [NVI05] was utilized for the hardware implementation.

The bitmask and the two group colors should be ideally written with 16 bit precision to the red, green, and blue component of the frame buffer. Unfortunately, only 8 bit frame buffer precision is currently largely available. A possible solution would have been to use an additional output target and to write the bitmask to the red and green components of the target and the colors to the red/green and blue/alpha channels of the frame buffer, respectively. However, there were no multiple render targets available for the NVIDIA card during the implementation. Another alternative, 16 bit pbuffers, had to be discarded due to huge performance penalties caused by context switches.

For these reasons, three rendering passes are applied. The first pass computes the bitmask, whereas the second and the third passes compute the two colors. Each value is computed with 16 bit, split into a high and a low byte and written to the red and alpha component of the fragment. After each pass, the pixels covered by the rendered quadrilateral are read from the frame buffer. Actually, a two-pass solution is also sufficient: one pass for the bitmask and one pass for both colors. However, this requires a fragment shader for the decoding to separate the colors on the graphics card.

The algorithm was later also transferred to Direct3D9 to have access to multiple render targets (MRT) which are now also available in OpenGL2.0 and to make porting between different graphics architectures easier. Three different implementations of the algorithm written in pixel shader assembler are available. A three pass version using pixel shader 2.0 that runs on the ATI R3xx series. Three rendering passes are needed due to the limitation of possible arithmetic instructions. Therefore, the tasks have to be split into three passes. Two for the bitmask and one for extracting the two colors. The second version uses two passes and is written in pixel shader 2.x (x stays for extended) and can run on both ATI R4xx and NVidia's Geforce 6 architectures. In this version the two phases for generating the bitmask are combined to one. And finally the third version uses only one rendering pass where both the generation of the bitmask and the two reference colors are combined into one shader. It is also written in pixel shader 2.x. The difference in execution speed of all three versions where measured by running the whole compression code in a loop 40 times. The results for an ATI X800Pro card for encoding a 512×512 image are for the three pass technique 12.65 ms for the two pass technique 12.57 ms and for the one pass technique 12.47 ms. It is surprising that the speed gain is rather low since for the one pass method several texture load operations can be saved compared to the other two. The total execution time for the one pass method including copying the result surfaces to main memory takes about 4.5 ms on the X800Pro.

Decoder Whereas the encoder requires the extended fragment processing features of the latest graphics hardware generation, a hardware-based decoding can be achieved basically with OpenGL 1.0 functionality. The reconstruction requires three passes. First, the bitmask data is written into the stencil buffer. In the second pass, a screen-sized quadrilateral is rendered with the *second* color of the CCC algorithm mapped as a texture. The OpenGL stencil-test is exploited to write only pixels with their corresponding bit in the bitmask set. In the third pass, using the *first* color as a texture, only pixels with a corresponding unset bit are written by inverting the stencil-test. This solution took 5.4 ms for decoding a 512 $\times$ 512 image on the test system. The reconstruction can be reduced to a single rendering pass if the interaction server offers multi-textures and extended fragment color combination (as provided since NVIDIA's GeForce256 chip via the `register_combiners` extension [Kil01]). In this case, the bitmask and the two group colors are bound as three textures and a combiner setup routes, either the first or the second color to the fragment color depending on the value of the bitmask texture. This decoding required 5.3 ms which implies that the texture setup is more expensive than the rendering itself.

The color textures can be directly defined from the CCC image stream, provided that the encoded data for the bitmask and the colors are stored consecutively and not interleaved as in the original CCC description. OpenGL intrinsically supports the 5/6/5 quantized color format. All that is necessary is to provide OpenGL with the right offset in the data block. Unfortunately, the bitmask requires more effort. OpenGL does not allow to specify a texture from bit data; therefore, the bitmask first has to be converted to a byte stream with one byte per bitmask bit. Using a fragment shader for the reconstruction, the bitmask texture can also be defined directly from the image stream since the individual bits per pixel can be extracted from the fragment shader on the fly. Thus, as programmable fragment processing becomes a common feature for desktop PCs in the near future, the need for unpacking the bitmask data is eliminated. The fragment shader implementation required 6.8 ms when using three individual textures for the bitmask, the first and the second color. This is slightly slower than using an already unpacked bitmask. However, on systems with slower processors, a significant speed-up from 20.3 ms to 8.9 ms could be measured.

3.5 An Alternative Remote Rendering Method

As it could be seen from the previous section, image compression techniques can help to reduce the amount of necessary image data transferred through the network, but also increase the work load on both sides. Besides, they still require an expensive fullscreen image blit operation on the client to copy data onto the screen. Increasing power and functionality on mobile devices – some devices already have a second generation 2D accelerator, and first generation 3D accelerators are just around the corner – could require a reconsideration of the previous image based remote rendering strategy. It might be wise to re-balance the work load between mobile client and stationary server. In this chapter a new shared visualization approach is described. The original results of this joint work with Martin Görke was first published in [DGE04].

As with every shared visualization approach, the main idea is to split up some of the image generation tasks between client and server with the aim to reduce the necessary network traffic, exploit increasing – otherwise not used functionality – on the client, and to have a more fair balancing between client and server. The idea in this approach is to just render 2D line primitives on the client, that have previously been generated on the server derived from arbitrary 3D scenes. Using line primitives introduces of course some limitations. For example right now we cannot have fully colored and shaded images on the client, instead we have to rely on drawing feature lines of the objects contained in a 3D scene. But according to research by Tufte [Tuf97], in many cases a set of feature lines is sufficient enough to engender a good visual impression of the scene to the viewer. Besides, due to the small screen size, resolution and color depth of mobile devices, the benefit of having color must not always be that great.

3.5.1 Basic Concept

The idea of the remote line rendering system is to split the workload between client and server and at the same time reduce the required network bandwidth for each frame. Of course most of the tasks are still done on the server as the client has limited CPU and especially memory resources, which prohibit to store and to process large amount of data locally. Therefore, the work between client and server is split in the following manner:

Server

The server is doing the complete scene handling, storing all the 3D objects of the scene – if possible – in its main memory and processing them for each render pass. The server also handles the user input events which have been sent by the client, updates the corresponding transformation matrices and starts processing the next frame. It is responsible for finding all the feature lines including the view dependent silhouette lines and transforming, projecting them to 2D window coordinates. Additionally, it has to take care of hidden surface removal. As only line primitives are drawn and no filled polygons with depth information, this cannot be done on the client. Also before sending the extracted 2D lines it does some packaging of lines to linestrips and additional optimization steps.

Feature lines

Numerous works have been published on deciding which lines of a 3D object are important for the human visual perception [ST90, Goo98, MKTH97]. Also various research has been conducted to find these lines efficiently. There are two principle methods for finding these features lines either by applying line detection filters in image space [ST90] or by using the edge information in object space [Goo98, MKTH97, GSG$^+$99, ZI97, BS00]. For object space detection

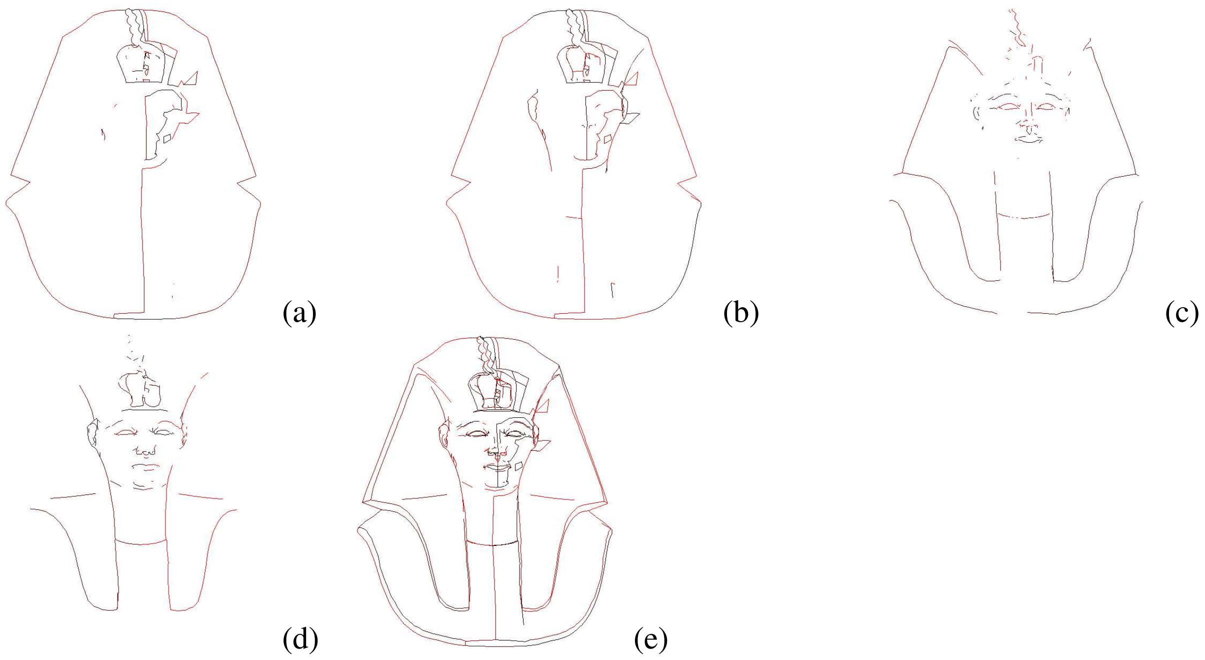


Figure 3.12: Explanation of different classes of feature lines. (a) boundaries, (b) silhouettes, (c) ridges, (d) valleys, (e) all feature lines combined.

the following classes of feature lines can be distinguished. These are found by the server and stored in an extra edge buffer:

- **Boundaries** (see Figure 3.12(a)), these are edges at the boundary of a triangle mesh and occur when the mesh cannot be described through a closed polyhedron. They can be simply found by just looking if the edge in the edge list has two faces or not.
- **Ridges** (see Figure 3.12(c)), these are edges resulting from a bump in the object surface and often are used to emphasize the shape of the surface. To test if an edge belongs to the group of ridges, the dot product between the normals of the two faces $\vec{n}_{face1}$ and $\vec{n}_{face2}$ connected by an edge is computed and tested against a user-defined threshold value.
- **Valleys** (see Figure 3.12(d)) are closely related to ridges and can be detected by the same method. To distinguish between a valley and a ridge edge a second vector is needed. This second vector is taken from the edge of the first face pointing away from the current testing edge. Afterwards another dot product is computed between this vector and the normal of the second face, and if its sign is positive it is a valley edge, otherwise a ridge edge.
- **Silhouettes** (see Figure 3.12(b)), are edges that connect a front-facing face with a back-facing face. They differ from the previous mentioned feature lines as they depend on the position of the viewer and therefore must be computed for each frame. Our silhouette detection routine is very similar to Buchanan and Sousa [BS00]. It iterates over all faces

and decides if the face is back facing. When the face is back facing all the type flags of the edges of this face will be marked as silhouette by using an XOR bit operation. At the end edges marked as silhouette are silhouettes, all other edges are not. This is due to the fact that either this type will never be set, because only front faces have been found, or in the case of two back faces it is unset again through a double XOR operation.

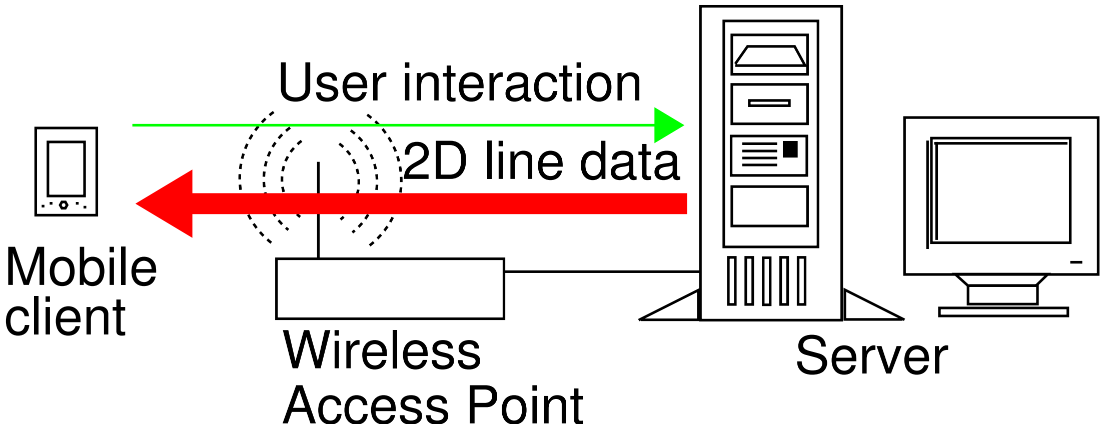


Figure 3.13: Architecture overview. The thickness of the lines represents the amount of data transferred over the network.

Client

A schematic illustration of the remote rendering system is shown in Figure 3.13. The client only has to rasterize the received line packages into its frame buffer and propagate input events back to the server.

3.5.2 Implementation

This section describes how each step is performed for both the mobile client and the stationary server. The mobile client is assumed to be a Pocket PC class type, the server has no general restriction and can be any type that has 3D hardware installed.

Client

Pocket PC clients currently have a very small screen resolution which is usually 240x320 pixels and 16 bit color depth. Hardware-accelerated 3D graphics is not yet available for Pocket PCs, some 2D acceleration chips showed up recently in a few devices but are not a general requirement and therefore still not very wide spread. The well-known interface for text- and graphics output of Windows, GDI, is also available on Windows CE Pocket PC clients and can be used in the same fashion as on normal Desktop PCs. Right now DirectX is not available on Mobile Windows for PocketPC, but for graphically intense applications like games, Microsoft released the Game API (GAPI) [Mic05b] for Pocket PCs. This Game API is very limited and it only provides direct access to the frame buffer for blitting. No abstract operations for drawing primitives or copying regions in the buffer are provided. Thus other people have developed several 3rd party libraries [Vik05, Inm05, Tre05a] which are built on top of GAPI providing a broader range of functionality.

Interesting to note is the significant performance difference between GDI and GAPI, a simple full size backbuffer clear with bitblt takes about 21.79 ms on a iPAQ 3850 Pocket PC 2002 Arm 211 MHz device. While using GAPI the same operation takes about 6.14 ms. To support possible subsequent 2D acceleration in GDI an implementation for both interfaces has been realized. For 2D line rasterization the well-known Bresenham algorithm [Bre65] was implemented in GAPI and in GDI the standard *PolyLine()* function was used.

After each frame has been rendered on the client, a synchronization message is sent to the server to acknowledge the receipt of the next frame. Also stylus interactions and button events will be sent uninterpreted directly to the server.

Server

In the remote rendering solution the server still has a lot of work to do. Before anything can be displayed to the client, the 3D data has to be loaded, typically from hard disk. Its content is stored in an object list. An object again contains a list of vertices, edges, an index face set of the mesh, and a list of faces. After loading, all edges of each mesh are extracted and stored in a corresponding list. With the help of this edge list the feature lines of the objects are extracted.

After finding all the feature lines the next step for the server is to remove all those lines which are not visible to the viewer. We decided to use an image-space algorithm previously described by Northrup and Makosian [NM00]. It has the major advantage that it makes use of the hardware-accelerated depth buffer of the graphics card and, therefore, is very reliable and fast.

In a preprocessing pass all the objects are rendered in their normal solid filled form into the depth buffer. It is wise to use a slight polygon offset while rendering to avoid z-aliasing in the next pass. Next, all previously determined feature lines are rendered into depth and color buffer with each feature line having a unique color ID. To ensure that this color ID is unaffected until rasterization all lighting computations should be turned off.

The transformation of the feature lines to 2D window coordinates is done automatically when a 3D hardware library like OpenGL or Direct3D is used. The only problem is to get this 2D data back from the library to send it over the network to a client. To do this we developed two different methods.

Method I The first method uses a special function only available in OpenGL – the feedback buffer. When using the feedback mode of OpenGL, primitives are not rasterized like the usual

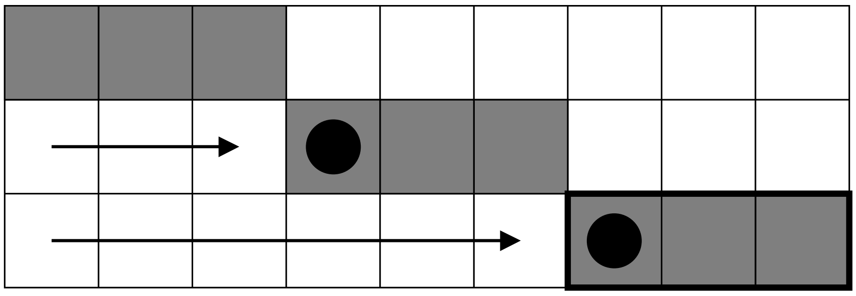


Figure 3.14: Scenario of what might happen when using bottom up, left to right scanline scanning for finding visible line segments.

way but rather the data of the transformed coordinates of each vertex are stored in a special buffer. In this so-called feedback buffer they can be read back by the application. For each feature line the transformed starting and ending point in window coordinates and each color ID is stored in the feedback buffer. In the next step the scene is rendered in normal rendering mode. Now the data from the feedback buffer is taken and a scanline conversion of each feature line is performed. But instead of writing pixels to the image plane, pixels are read from the previous generated image and their color is compared with the current color ID. If it matches the color, the current line will be marked as visible. Due to occlusion, lines can be disconnected by other lines. During the scan conversion of the line a visible segment will be generated for each interruption that occurs. At the end all these line segments resample the visible lines, and their start- and endpoints are stored in a list.

Method II The previous method requires two rendering passes, first the pass for feedback mode and then another pass for rendering the lines in the output image. (Feedback mode and normal mode cannot be executed at the same time in OpenGL). Additionally, a software line rasterization is required to find the visible lines in the output images. To avoid the double rendering and scanline rasterization we propose a second different method. In this method the output image is analyzed and the visible line segments are determined line by line.

For each rendered feature line there is a list of visible segments. If during the scanning of the image a color ID is found it will first be checked if there is already a visible segment of the corresponding feature line belonging to this color ID. When there is already a segment belonging to the line the pixel will be tested if it is adjacent to the last found segment. In case it does, the segment is extended by the current pixel position. Otherwise, if there is no segment, or the last segment is too far away a new segment is generated.

There is a certain problem when scanning the image from bottom to top and left to right with lines that have a slope smaller than zero and greater than minus one. Figure 3.14 shows such a scenario. In this figure a segment has been found at the bottom line, but the marked pixel of the next line is too far away from the marked pixel of the currently found segment. In this case unnecessary segments might be introduced. To avoid these unwanted segments, first all segments of each line are temporarily stored until the end of the current scanline is reached. Afterwards all temporary segments are connected to each other. Alternatively it is possible to check the current segment after each extension to see whether it can be connected to the previous one.

Number of Lines	230	820	1530	2400
Ratio m_1/m_2	1.53	0.97	0.42	0.37

Table 3.3: Comparison of the efficiency between Method I and Method II for getting the 2D line data.

Depending on the number of feature lines either method I or method II is faster. Table 3.3 shows the ratio between the two methods for different numbers of lines.

To reduce the network bandwidth even further it is wise to merge single lines to polygon lines if possible. To do this the eight nearest neighboring pixels of each line endpoint are tested in the rendered image if they contain a pixel with a different color ID. If a pixel with a different color ID is found, and the corresponding line is not yet part of the polyline, the location of its endpoints are tested. If one of the endpoints lies also in its 3x3 pixel neighborhood the line will be added to the polyline group.

Afterwards the polylines resulting from this process can be simplified by using a technique described by Douglas and Peucker [DP73].

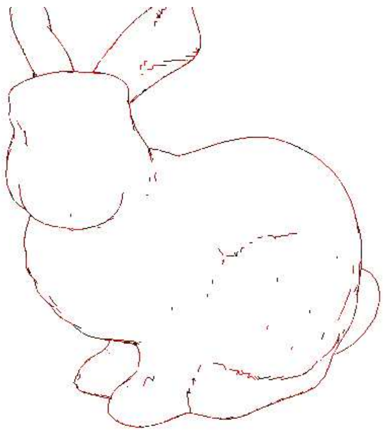
3.5.3 Results

The server component of our remote rendering architecture is implemented in OpenGL using C++ and runs on the Win32 platform. A minimal GUI was implemented on the server to provide the user some control options for rendering. For example, the user can decide which feature lines should be extracted on the server. It also has a control output viewport and some additional statistics like current frames per second, connected client IP, etc.

The client has been implemented in Embedded Visual C++ 3.0 and runs on all Pocket PC 2000 and Pocket PC 2002 compatible machines. A rendering interface for GDI and GAPI [Mic05b] is provided and the user can choose at startup which interface should be used. Later a second version has been ported to Embedded Visual C++ 4.0 to support newer models running under Mobile Windows 2003 and use optimization for later processors like Intel's XScale PA-Series.

When the client connects successfully to the server it switches to full screen mode. Basic navigation like rotate, zoom, and translate have been implemented using stylus and button input.

To measure the performance of the solution two different test scenes on two different client devices have been examined. The first scene (see Figure 3.15(a)) uses a model of the well known Stanford Bunny; it has 27808 vertices and 52649 faces. Its feature lines basically consist of silhouettes and a few border lines leading to very few lines. The second test scene (see Figure 3.15(b)) is a model of the Statue of Liberty, it has 19006 vertices and 37186 faces. The line


(a)

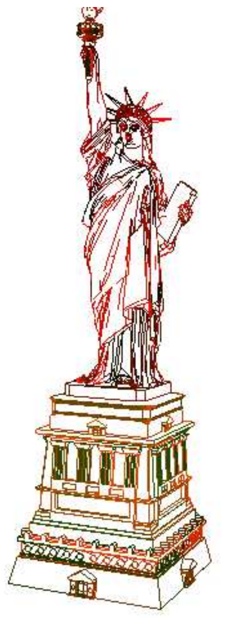

(b)

Figure 3.15: The three different test scenarios in their line representation. (a) Stanford Bunny model, (b) Statue of Liberty.

Scene	Device	Resolution	FPS	
			GAPI	GDI
Bunny	HP iPAQ 3850	QVGA	33-36	15-17
Bunny	Toshiba e800	QVGA	33-36	23-26
Bunny	Toshiba e800	VGA	23-24	10-11
Liberty	HP iPAQ 3850	QVGA	20-22	5-6
Liberty	Toshiba e800	QVGA	20-24	8-10
Liberty	Toshiba e800	VGA	15-17	5-6

Table 3.4: Frame rates achieved on different mobile clients with different screen resolutions.

representation contains a high amount of border and valley lines. The overall 2D lines per frame to represent the individual scenes depend largely on the current viewpoint. In the case of the bunny it is approximately 270 lines in average and for the Statue of Liberty about 1500 lines. The first client device was a HP iPAQ 3850 PocketPC using PocketPC 2002. The second client was a newer Mobile Windows 2003 device from Toshiba the e800. An interesting special feature of the Toshiba e800 is the support of the new high resolution mode supporting native VGA resolution and not just QVGA. With this option the scalability of the approach to higher resolutions can be tested. For the server a Pentium IV 2.8 GHz installed with a Radeon 9700Pro was used.

Table 3.4 shows the received frame rates for both systems and scenarios. Additionally, another test run was conducted but this time the throughput of the access points in the wireless network were artificially slowed down to a 1Mbit TxRate to test how well the approach will behave for slower networks, expecting similar performance of next generation of wide-area mobile networks, for example UMTS. The achieved frame rates for this using the standard QVGA resolution were 23 FPS for the Stanford Bunny and 10 FPS for the Statue of Liberty. When looking at the frame rates for the normal bandwidth case of a Wireless LAN, it seems that they are not limited by the network bandwidth but rather how fast either the server can produce the necessary lines or the client can actually draw them. This is a nice indication that this approach is going into the right direction, by shifting the problems of remote rendering from the usual network bandwidth issue to other areas. These problems are more likely to be solved with the next generation of hardware, because CPU and GPU power increases a lot faster than network bandwidth. It also can be seen that the performance gap between the GAPI and the GDI implementation increases with the number of lines, that have to be drawn at each frame on the client. The GDI implementation definitely seems to be a bottleneck right now. Also since the Toshiba e800 has a second generation 2D graphics acceleration chip installed, it cannot really benefit from it or at least nothing appears to be accelerated. Also the frame rates seem to scale quite well with higher resolutions whereas an image based approach would probably quarter its performance thus the loss in this approach is much less.

With the decreased network bandwidth of 1 Mbit/s, the system is likely running into the point where the network is actually the limiting factor. However, 10-23 FPS are still respectable frame rates and lie in an interactive range. This means this solution should perform quite well with next generation mobile wide-area networks.

Task	Time ($\approx$ in ms)	
	Bunny	Liberty
Silhouette detection	6.0	1.5
Visibility	5.6	30.0
Extract Poly lines	1.1	11.9
Simplify	0.4	3.4
Package Data	0.3	4.7

Table 3.5: Amount of time spent on the server for each task of the two different test scenarios.

To examine the performance drain in the high bandwidth case we profiled the server to check where most of the time is spent. Table 3.5 shows the resulting figures. The timings uncover some interesting behavior. In the case of the bunny, the limitation appears clearly to be on the clients ability to draw the 2D lines. (The total processing time for the bunny is $\approx$ 13.5ms resulting in possible achievable frame rates of $\approx$ 74 FPS).

Not surprisingly, most of the time on the server is spent either on visibility checking or silhouette detection. Note that the times for visibility checking include a readback from the frame buffer. This operation takes with basic *glReadPixels* $\approx$ 2.8 ms on the Radeon 9700Pro and $\approx$ 1.7 ms on the Geforce FX 5800. Further increase might be expected by using special AGP / PCI-Express memory areas.

3.5.4 Open Issues

There is still room for improvements with this approach on both sides – client and server. Right now on the server most of the time is either spent in the silhouette detection, in the visibility checks or polyline extraction. The polyline extraction cannot be improved that easily. But the silhouette detection and visibility testing could be possibly improved by using an image space approach for feature line detection which can be run completely on the GPU. The advantage would be that the visibility checking is automatically provided and would not require an additional pass as it does right now. However, this option might be traded with a slightly more complicated polyline extraction.

On the client the first choice would be to use a faster line algorithm or strategies described by Chen et al. [CWB02]. Also as the client has 2D vector data some transformation operations could – to some extent – be completely done on the client (like translating and scaling) without requesting new data from the server. Right now this function is not exploited but could become quite useful if the mobile client runs into a network signal loss. It might be interesting to investigate special line compression schemes to further reduce network load. Right now only LZO compression has been tested that leads to a reduction of data by just 3%. Other possibilities could be to use different primitive types for example curved representations.

But the biggest issue which should be addressed in the future is aliasing. Aliasing is a very pressing challenge especially on small screens where it becomes quite noticeable. Unfortunately, on current generation of mobile devices there are not enough resources left to cope with these problems. It might be worth to investigate algorithms for low-cost screen antialiasing especially suited for low power graphics accelerators for example described by Akenine-Möller and Ström in [AMS03].

Chapter 4

Rendering Transparent Surfaces

As already stated in the introduction of this thesis, the display of transparency plays a significant role in computer graphics and visualization. Not only do a lot of real world objects have a transparent or semi-transparent material appearance, but transparency can also often be used to encode additional information for example in a visualization context. Or it can be used to show further details and even provide context information. Transparency in the context of mobile devices also has the benefit to use the limited screen space more efficiently by stacking information. Unfortunately, rendering transparent surfaces is still one of the bigger challenges in hardware-based rasterization graphics. Additionally, in the previous chapter several rendering styles were investigated that help to reduce the amount of transmission data to the mobile client machine. They were selected according to a criteria catalog (see Chapter 3.4.3). Although the selected rendering styles seem to fulfill nearly all of the criteria above, they can fail in respect to transparent and semi-transparent objects.

Therefore, in this chapter different approaches to the visualization of transparent surfaces in different non-photorealistic rendering styles are addressed. Before a more detailed discussion on how these methods can be simulated is presented, a short introduction into how transparency is handled in surface rendering is provided. Additionally, at the end of this chapter improvements to the common computer graphics model are discussed for creating a better impression on the visual appearance of transparent surfaces to the viewer.

4.1 Transparency in Computer Graphics: General

To achieve the most realistic representation of transparency one simply has to take a classical Whitted raytracer [Whi80] and follow the refracted view ray from the hit point on the surface through the object till it hits another object. The refracted ray can be computed by applying physical laws like Snell's law for geometrical optics. Unfortunately, realtime graphics is still based on scanline rasterization that is able to run very efficiently on special dedicated hardware. Although some attempts have been made to incorporate the raytracing algorithm with the programmable features of modern graphics hardware [CHH02, PBMH02] it still does not really show to be a feasible alternative to the traditional scanline rasterization approach. Therefore, more simple solutions to reproduce the impression of transparency in scanline rasterization have been proposed already very early in the days of computer graphics by Newell et al. [NNS73]. They proposed a very basic algorithm that is still most often used in combination with graphics hardware. The original algorithm includes only a linear intensity function and can be summarized as follows. A background image is first rendered and stored. Afterwards the image of the transparent object is painted over this background image. The color of each pixel is determined according to the formula

$$color_{final} = color_{dest} \cdot t + color_{src} \cdot (1-t) \quad , \tag{4.1}$$

where $color_{final}$ is the color value of the final image, $color_{dest}$ the stored pixel value of the background image, $color_{src}$ the color of the transparent object assuming it had been opaque, and t the transparency (0 to 1) of the object. This formula is also known as additive transparency model as will be shown later in this chapter. The method effectively weights the background image with the object being shaded to produce the appropriate combination. The algorithm is very efficient, but its naive implementation runs into trouble when there are multiple transparent and opaque objects in a scene, since a correct rendering order has to be determined to reproduce the correct result. Kay and Greenberg [KG79] extended this simple approach to better capture the transparency impression on curved surfaces. Ideally, the transparency should be related to how much material the light must pass through. A first approximation for curved surfaces would therefore ensure that the amount of transparency decreases near the edges. Kay and Greenberg concluded that this can be accomplished by a function

$$t = (t_{max} - t_{min}) \cdot (1 - (1 - NormZ)^k) + t_{min} \quad , \tag{4.2}$$

where t is the transparency of the current pixel as in equation 4.1, t_{max} the maximum transparency of any point on the surface of the object, t_{min} the minimum transparency of any point on the surface of the object, $NormZ$ the Z component of the unit normal vector of the surface at this point, and k a cosine exponent factor.

With a typical exponent value of two or three, this function produces a relatively constant transparency across most of the object and quickly reduces to zero near the edges. Regardless of the environment configuration, the transparency is basically controlled by the direction of the surface. Different exponents can be used to simulate the transparency of various types of material. For a very thin glass, a thin line of lower transparency along the edge can be obtained by using a larger exponent. Figure 4.1 compares visually the linear transparency model with Kay and Greenberg's non-linear algorithm.

4.2 Transparency in Technical Illustrations

In contrast to realistic rendering of transparent materials artists found different, unique and effective techniques and rules for handling transparency in order to communicate the location of

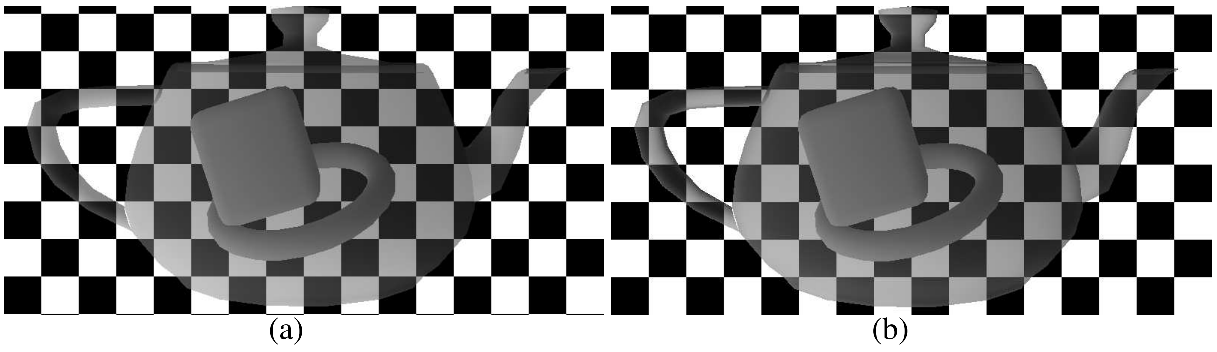


Figure 4.1: Visual comparison between (a) linear and (b) non-linear transparency model as proposed by Kay and Greenberg [KG79]. Note the difference at the silhouettes of the objects.

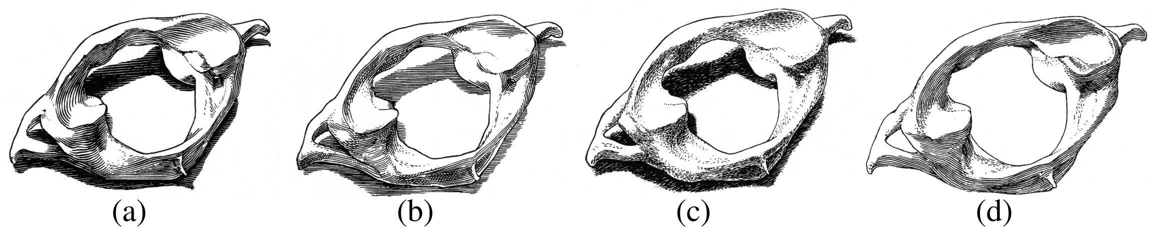


Figure 4.2: Comparison of different ink line styles. (a) eyelash, (b) cross hatch, (c) short line, (d) brush lines. Image taken from [Hod89].

occluding and occluded objects. When looking into the classical literature of scientific and technical illustration [Tho68, Mar89, Hod89] focused on the learning artist it can be noticed that the rendering of transparent surfaces can be divided – with some exceptions – into basically two general classes:

Direct methods

Indirect methods

Direct methods try to visualize the effect of transparency directly by special drawing techniques or surface shading methods. Indirect methods do not visualize the effect of transparency directly but try to express it on a more abstract level to achieve the same effect. To this class for example belong the well-known explosion drawings and cutaway drawings in technical illustrations, that take up a high percentage of all technical illustration styles. In this chapter various direct and indirect methods for visualizing transparency in technical illustrations are discussed and for those methods where a simulated computer graphics implementation exists an implementation approach is discussed in more detail.

4.2.1 Direct Transparency Methods for Line Drawings

In technical illustrations a large group of illustrative styles use just pen & ink as drawing medium and therefore do not reproduce any color information. This is often done to cut down the cost of the print. Figure 4.2 shows some examples of different pen & ink drawing styles. Different methods and algorithms for creating these sort of drawings with the help of computer graphics can be found for example in Strothotte & Schlechtweg [SS02] or more graphics hardware related methods by [FMS02, OMP03]. Shading information in these drawings is either not provided at all or symbolized through different hatching methods.

Phantom lines

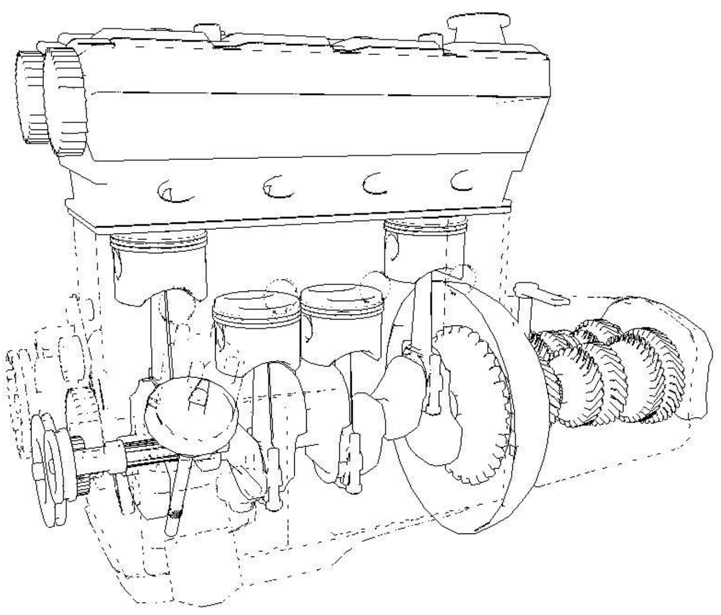


Figure 4.3: Illustration of a transparent surface using phantom lines style.

The simplest method of visualizing transparency is to draw only the outlines of transparent objects. These outlines are rendered in a linestyle different to the other outlines to make them distinct from the opaque objects (see Figure 4.3). This particular linestyle is often classified as phantom lines [Tho68]. Although this technique is very simple and can be applied to a wide-range of illustrative styles, there are certain drawbacks. First, nearly all the details – especially material and surface information – of the transparent object are lost since only its outlines are drawn. Second, no distinguishing between different levels of transparency is possible. Nonetheless, it is a widely used technique among artists, especially for illustrations that do not provide any shading information in the first place. Nienhaus and Döllner [ND04] recently published a graphics hardware-based method for generating blueprint illustrations for architecture and technical illustrations which is a similar approach.

Hatched surfaces

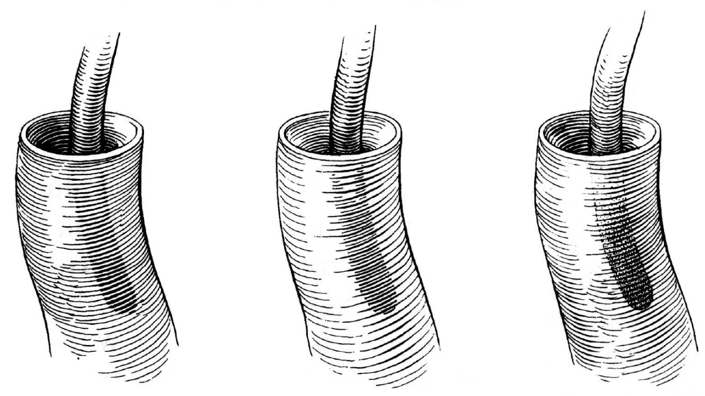


Figure 4.4: An example of different strategies in line drawing to symbolize the appearance of a transparent surface. Image taken from [Hod89].

For hatched line drawings illustrators seem to use three different ways to show the effect of transparency. The first alternative is to thicken the lines of an opaque object lying behind a transparent surface (see Figure 4.4 left). In the second method, the opaque object lying behind the transparent surface is drawn with more lines (see Figure 4.4 middle). Finally, in the third method the opaque object behind the transparent surface is drawn in a different shading technique (see Figure 4.4 right). All three methods have one characteristics in common: when an object is partially occluded by a transparent surface the part which is occluded is drawn different to the part which is not, and the occluded part is faded out as closer it comes to the barrier of the transparent surface.

Hamel et al. [HSS98] described a semi-automatically rendering system for reproducing this sort of drawings. A brief description will be given here. The line drawing routine in their systems seems to be decoupled from the transparency and lighting calculation and uses the final intensity value of the object region. Therefore, this part only concentrates on how to create the right intensity value for the later final rendering step. Working directly on the brightness image is according to the authors a straightforward process: an edge detection in the objectID-buffer (a special output of their 3D OpenGL renderer) between the outer and inner part is executed and later followed by a vectorization that gives a number of lines. These can be used for all pixels of the occluded object to determine the smallest distance d_s. This distance is set in relation to the parameters d_{out} or d_{in} – depending whether the pixels are inside or outside – that define the length of the respective fading areas. The blending equations can be described as a linear interpolation of the brightness values from the picture without (b_i) and with (b_i^T) transparency applied to the pixels on the inside.

$$b_i' = \begin{cases} b_i \cdot (1 - \frac{d_s}{d_{in}}) + b_i^T \cdot \frac{d_s}{d_{in}} & d_s < d_{in} \\ b_i^T & otherwise \end{cases} \tag{4.3}$$

For the outside part a linear scaling is sufficient.

$$b_i' = \begin{cases} b_i \cdot \frac{d_s}{d_{out}} & d_s < d_{out} \\ b_i & otherwise \end{cases} \tag{4.4}$$

The fading will follow the edge between the inner and outer part. Thus if the edge is formed differently for example where the curvature of the edge is different to the curvature of the object, the result will look less satisfying. The logical consequence according to the authors is to define the fading analogously to the above 2D solution in 3D. For each area now a start point, a normal vector indicating the direction, and a length has to be defined. These parameters are set by the user via a pyramid-formed widget.

4.2.2 Direct Transparency Methods for Color Illustrations

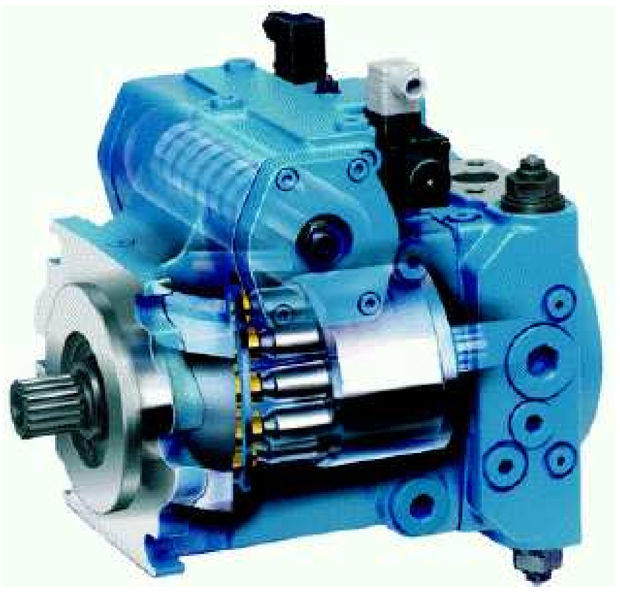


Figure 4.5: An example of traditional technical drawing, showing transparency effects.

An approach for visualizing transparency in color illustrations can also be found in “The Guild Handbook of Scientific Illustration” by Hodges [Hod89]. For color illustrations, Hodges recommends to lighten the color of object regions which are occluded by transparent objects. Hodges proposes thereby following basic rules:

- Strengthen the shade where an opaque object enters a non-opaque object.
- Set the intensity of the opaque object to zero at the edge of the surrounding object and slowly increase its intensity with increasing distance from the edge.

To put it another way, transparency falls off close to the edges of transparent objects and increases with the distance to edges. Besides these fundamental rules in visualizing transparent objects, there are others that are not directly described by Hodges and which focus on the correlation between objects. These precepts are often obvious for an illustrator, but cannot directly be transformed into rules that are appropriate for a computer-based implementation. However, by analyzing real color-shaded technical drawings, such as Figure 4.5, the following simplified rules can be identified:

- Back faces or front faces from the same non-opaque object never shine through.
- Opaque objects that are occluded by two transparent objects do not shine through to the closer transparent object.
- Two transparent objects are only blended with each other if they do not distract the viewer or if they are very close to each other and belong to the same semantic group.

This set of rules are based on the fact that in technical drawings transparency is used to look into objects and to show objects which lie inside or go through non-opaque ones. Often these object are opaque in reality.

For an automatic rendering process which was developed in a joint work with Daniel Weiskopf and first published in [DWE02], the last rule is changed to

- Two transparent objects never shine through each other,

since semantic grouping is something which cannot be achieved without additional user interaction in pre or post processing steps.

Basic rendering approach

From the above traditional methods to show transparency in manual drawings, the following small and effective set of rules can be extracted for a computer-based rendering:

- Faces of transparent objects never shine through.
- Opaque objects that are occluded by two transparent objects do not shine through.
- Transparency falls off close to the edges of transparent objects and increases with the distance to edges.

Based on these rules, the rendering approach is as follows. First, the objects which are blended have to be determined, following the guidelines of the first two rules. This task essentially corresponds to a view-dependent spatial sorting. An efficient and adapted solution to this problem is described later.

Secondly, transparency values have to be computed for a correct blending between transparent and opaque objects according to the third rule. In the subsequent paragraph, a corresponding algorithm is proposed. In what follows, the term α (opacity) value is often used instead of a transparency value. Note that transparency is $1-\alpha$.

View-dependent transparency

To simulate the third rule, the concept of view-dependent transparency is introduced: The α value for blending between transparent and opaque objects depends on the distance to the outline of the transparent object; the outlines of an object projected onto a 2D screen consist of silhouettes lines and thus depend on the position of the viewer.

Silhouette edges can either be determined in 2D image-space or in 3D world/object space. Hamel et al. [HSS98] recommend to use an image space method by establishing an object ID buffer and an edge extraction filter. However, this method is very time-consuming as you first have to render the objects with an ID tag, find the edges, vectorize these edges to 2D lines, and—with the help of these lines—calculate the distance of each pixel to the outline.

A 3D world/object space approach might be favorable since in this case all silhouettes are determined before the rasterization stage. With this approach most of the previously mentioned steps can be avoided. In 3D space, a silhouette is an edge connecting two faces, where one face of the edge is back facing and the other one is front facing. This classification can be formulated as follows:

$$(\vec{n}_1 \cdot (\vec{p}-\vec{o}))(\vec{n}_2 \cdot (\vec{p}-\vec{o})) < 0 \quad , \tag{4.5}$$

where $\vec{p}$ is an arbitrary, yet fixed point on the edge, $\vec{n}_i$ are the outward facing surface normal vectors of the two faces sharing the edge, and $\vec{o}$ is the position of the camera. In a naive implementation, all edges are checked for the above criterion and the detected silhouette edges are stored in a list. This is based on Appel's original algorithm [App67]. The performance of silhouette detection could be improved by more sophisticated techniques, such as fast back face culling described by Zhang et al. [ZI97], the Gauss map method [GSG$^+$99], or by exploiting frame–to–frame coherence.

After extracting all silhouettes, the distance of each vertex to the closest silhouette edge is computed. First, the distances d of the vertex to all silhouette edges are determined according to

$$d = \left\| \vec{v} - \left(\vec{p} + \frac{(\vec{v}-\vec{p})\cdot\vec{e}}{\|\vec{e}\|}\vec{e} \right) \right\|^2 \quad ,$$

where $\vec{v}$ is the position of the vertex, $\vec{p}$ is an arbitrary, yet fixed point on the silhouette edge, and $\vec{e}$ is the silhouette edge. The minimum distance of a vertex to the silhouettes, d_{min}, is used to calculate the α value for the respective vertex of the transparent object. As an example the following approach can be used

$$\alpha = 1 - \left(\frac{d_{\text{min}}}{d_{\text{object,max}}} \right)^k \quad , \tag{4.6}$$

where $d_{\text{object,max}}$ denotes the maximum distance between the center of the object and all surface points; $k \in [0,1]$ is a user-specified falling off term.

The above α value for the transparent surface and another, fixed value for the opaque object is used as weights for blending between transparent and opaque objects. Therefore, the α value of the opaque object determines this object's weight in the final image. Note that the background color itself may be blended with transparent surfaces as well. It is possible to specify different α values for opaque objects and for the background.

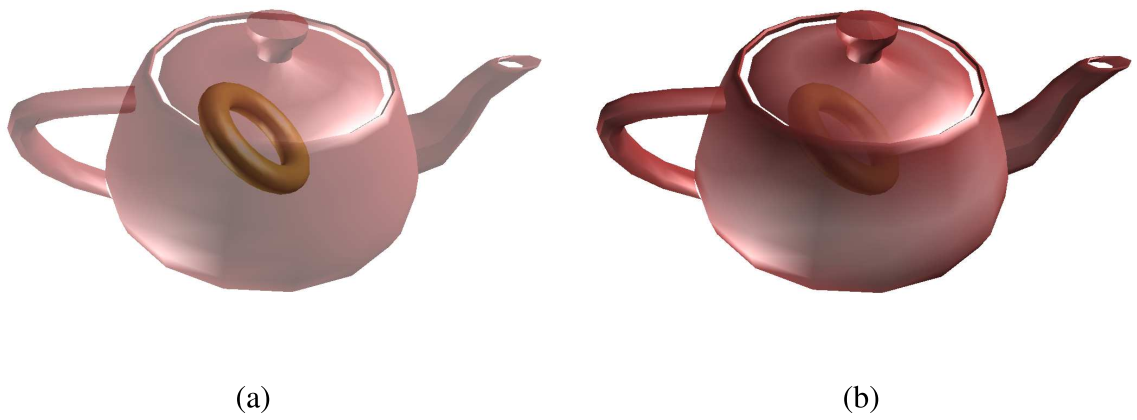
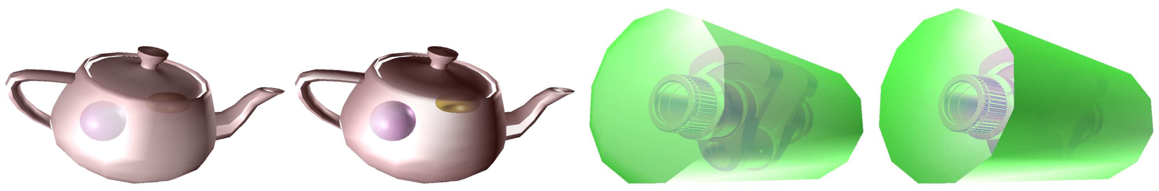


Figure 4.6: Difference between (a) standard transparency blending and (b)view-dependent transparency blending.

Figure 4.6 (a) and 4.6 (b) compare standard linear view-independent transparency to view-dependent transparency.

Finding the closest silhouette edge for each vertex and determine the distance to it is a rather time consuming operation and takes in a naive implementation at least $O(n^2)$. Another drawback is that this operation cannot be executed in the graphics pipeline itself but has to be computed by the CPU for each frame. Therefore, a faster method would be highly desirable. Not really the same but a close approximation at least for highly curved objects is to use an idea which originates from Everitt [Eve00]. The idea is to use a single pass approximation of silhouette edges by means of a vertex program and a simple 1D texture lookup per fragment. This approximation takes advantage of the fact that for silhouette edges the sign of the dot product between view vector and normal vector of the two faces connected by this edge switches its sign, meaning somewhere in between it must be zero. Instead of computing now the dot product between both faces connected by this edge, simply the dot product is computed between each vertex and its corresponding normal $\vec{n}_v$ and the view vector $\vec{v}$. The value of the dot product is stored as texture coordinate and passed further to the rasterization unit. In the fragment processing now the coordinate is used to lookup a color value inside a 1D texture. This 1D texture contains only two colors the color of the silhouette and a neutral background color. The silhouette region itself is marked with a ramp-like function. The advantage of this technique is that it is very easy to implement and really fast. The drawback is that it only works very well with curved objects.



(a) (b) (c) (d)

Figure 4.7: Difference between real closest silhouette finding method and simple approximation. (a) real, (b) approximation, (c) real, (d) approximation.

In the case of the blending value, the ramp function can be exchanged with a desired pseudo distance function. A visual comparison between this approximation and the original closest silhouette finding algorithm is illustrated in Figure 4.7. These figures show that the difference between the fast approximation and the more complex method of finding each silhouette and computing the distance of each vertex to the closest one, is in the case of very smooth objects, not that large. When looking at the teapot scenario the only real difference that can be found is the teapot cap where the approximation method fails to reproduce the same effect. But for non-smooth or more regular formed objects the visual difference can be quite large as can be seen in Figure 4.7(c) and Figure 4.7(d). It should be up to the user to decide if trading speed to strictness is desired.

Depth sorting

Now that the blending parameter between transparent and opaque objects is known, what is left to determine is which objects have to be blended and which objects are hidden and do not contribute to the final image. In full generality, rendering semi-transparent scenes would require view-dependent depth sorting. A large body of research has been conducted on these issues of spatial sorting and visibility determination [Dur99, FvDFH90]. The two main approaches to depth sorting are either based on screen-space or on object-space.

In this work, the focus lies on screen-space algorithms. A major advantage of this approach is that it can exploit dedicated graphics hardware to efficiently solve this problem and can thus avoid operations on the “slower” CPU. Screen space algorithms also benefit from the fact that some specific applications do not require complete depth sorting—like in this case since it is only required to determine the closest transparent objects and the opaque objects directly behind these transparent surfaces.

In all of the following depth-sorting approaches, it is assumed that volumetric objects are defined by boundary surface representations. Furthermore, the surface of a single objects is assumed to be connected, i.e., an object cannot be separated into different disconnected pieces.

Implicit interior boundaries Let us start with a first, quite common scenario. Here, opaque objects may be contained inside the volume of a transparent object. These opaque objects should

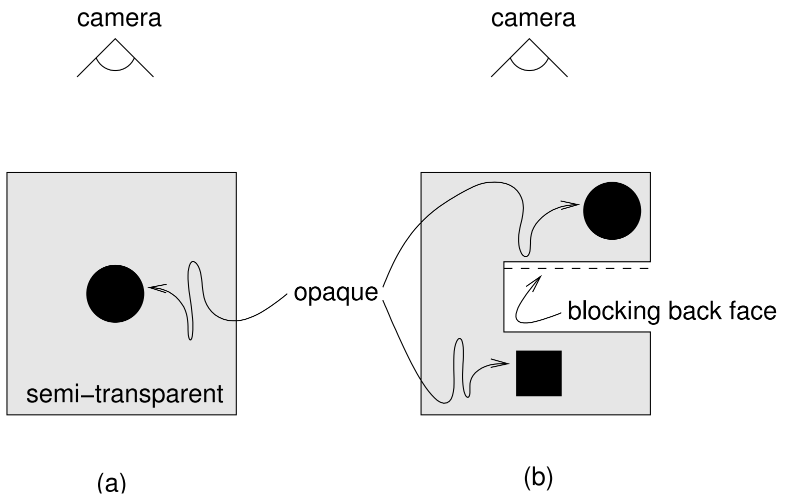


Figure 4.8: Scenario with opaque objects embedded into the volume of a transparent object. The left image shows a convex transparent object, the right image shows a concave transparent object.

be visualized by transparent rendering. Figure 4.8 illustrates this scenario. Here, the interior boundary of the transparent volume is given implicitly by the surface of the surrounded opaque object.

First, let us consider only convex transparent objects as shown in Figure 4.8 (a). Opaque objects, however, may be of arbitrary shape. Following the rules from the beginning of this section, the front-facing surface of the transparent object closest to the camera has to be blended with opaque objects contained within the volume of this transparent object. All objects—both opaque and transparent—which are located further away should be hidden by this nearest transparent object. Note that these invisible objects are not included in Figure 4.8 (a). Finally, opaque objects in the foreground are drawn on top of all other objects further behind.

In a slightly more complicated scenario, concave transparent objects are permitted, as illustrated in Figure 4.8 (b). Surrounded opaque objects should be visible only up to the first back-facing part of the transparent object in order to blend out unnecessary and distracting visual information. To put it another way, only those opaque objects that are closer to the viewer than the nearest back face of the surrounding transparent object are visible. In the example of Figure 4.8 (b), the opaque circular object is visible, whereas the squared object is hidden by a back-facing surface of the surrounding object.

Figure 4.9 shows the rendering pipeline for the scenario illustrated in Figure 4.8. The depth buffer is used to select the correct objects for blending and to hide the unwanted objects. In the first step, back faces of all transparent objects are rendered into the depth buffer only. In this way, the depth buffer contains the distance to the closest back-facing transparent surfaces. In the second step, the front faces and the silhouettes of all opaque objects are rendered to both frame and depth buffers. The depth test rejects all opaque objects lying behind any “transparent” back-facing surface. Finally, the front faces and silhouettes of the transparent objects are rendered and

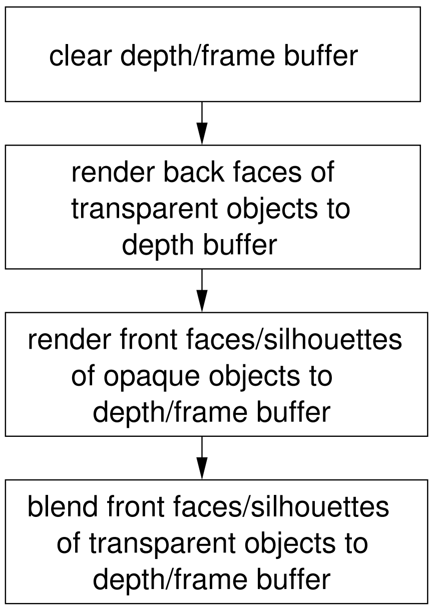

clear depth/frame buffer
render back faces of transparent objects to depth buffer
render front faces/silhouettes of opaque objects to depth/frame buffer
blend front faces/silhouettes of transparent objects to depth/frame buffer

Figure 4.9: Rendering pipeline for opaque objects embedded into the volumes of transparent objects.

blended into the depth and frame buffer, respectively. Here, the depth test rejects all parts of the transparent objects hidden by an opaque foreground object. Blending is applied only at those parts of the frame buffer where a transparent surface is directly visible to the user.

The algorithm can be implemented by only using standard OpenGL 1.2[SA99]. Writing to the depth or frame buffers can be enabled and disabled by `glDepthMask` or `glColorMask`, respectively. This rendering approach comprises only "one and a half rendering passes" because front faces of the opaque objects and both front and back faces of the transparent objects have to be rendered.

Explicit interior boundaries Now let us consider a more complex scenario. Volumetric objects are still represented by boundary surfaces. However, objects may no longer be contained inside the volume of another object. In fact, surrounding transparent objects have to be modeled with respect to both the outside and the inside boundary. Figure 4.10 illustrates this scenario for the example of an opaque parallelepiped inside a transparent cylindrical, mug-shaped object; Figure 4(a) in the color plates shows a color-shaded rendering of a similar scene. This scenario better reflects the properties of many technical 3D data sets, which explicitly represent all boundaries—both inside and outside.

The algorithm from the previous paragraph fails for this scenario, as all surrounded opaque objects are hidden by a back-facing transparent surface. To overcome this problem, another classification of the visibility of opaque objects is necessary: only those opaque objects located between the closest and second-closest front-facing transparent surfaces are visible. Objects – transparent or opaque – lying further behind are hidden.

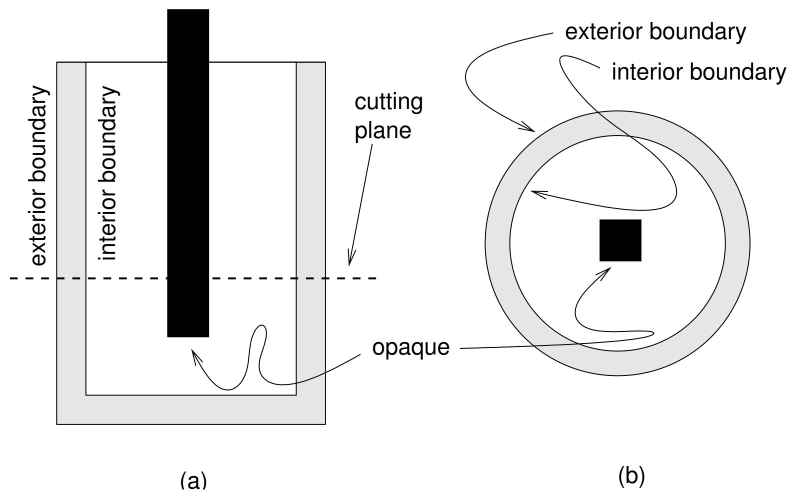


Figure 4.10: Scenario with an opaque parallelepiped inside a transparent mug-like object, whose interior boundary is explicitly modeled. The left image shows a side view with a horizontal cutting plane and the right image shows a top view onto the cutting plane.

Two depth buffers are necessary to perform this depth selection. However, only one depth buffer is directly supported on available graphics hardware. Fortunately, the required behavior can be emulated by means of texture mapping and per-fragment depth operations which are available since the introduction of NVidia's GeForce3.

The basic algorithm is as follows. In the first step, all front-facing transparent surfaces are rendered to the depth buffer; afterwards, the depth buffer contains the depth values of the closest transparent surfaces. As second step, the depth buffer is stored in a high-resolution texture and then the depth buffer is cleared. In the third step, the front-facing transparent surfaces are rendered to the depth buffer except the foremost ones, virtually peeling off the closest surfaces. After this step, the second-closest transparent front faces are stored in the depth buffer. In the fourth step, opaque objects are rendered. The depth test rejects all surfaces, but those lying in front of the second-closest transparent front faces. Finally, just the foremost transparent surfaces are blended into the frame buffer.

The principal idea behind step three is related to Everitt's *depth-peeling* approach [Eve01], making use of dual depth buffers [Die96] and virtual pixel maps [Mam89].

Figure 4.11 shows the details of the actual rendering pipeline. The left part contains the rendering steps, the right part conveys the intended behavior. Only front-facing polygons are drawn in all rendering steps.

The first four boxes implement parts one and two of the basic algorithm. The depth buffer is read to main memory, after having rendered the transparent objects to the depth buffer. Alternatively the depth values can be rendered into a special render target where a short pixel program maps the depth to output. If complex pixel programs and floating point targets are not available, a HILO texture can be used instead where the depth values are transferred as 32 bit unsigned integers. A HILO texture is a high-resolution 2D texture, consisting of two 16 bit unsigned integers per

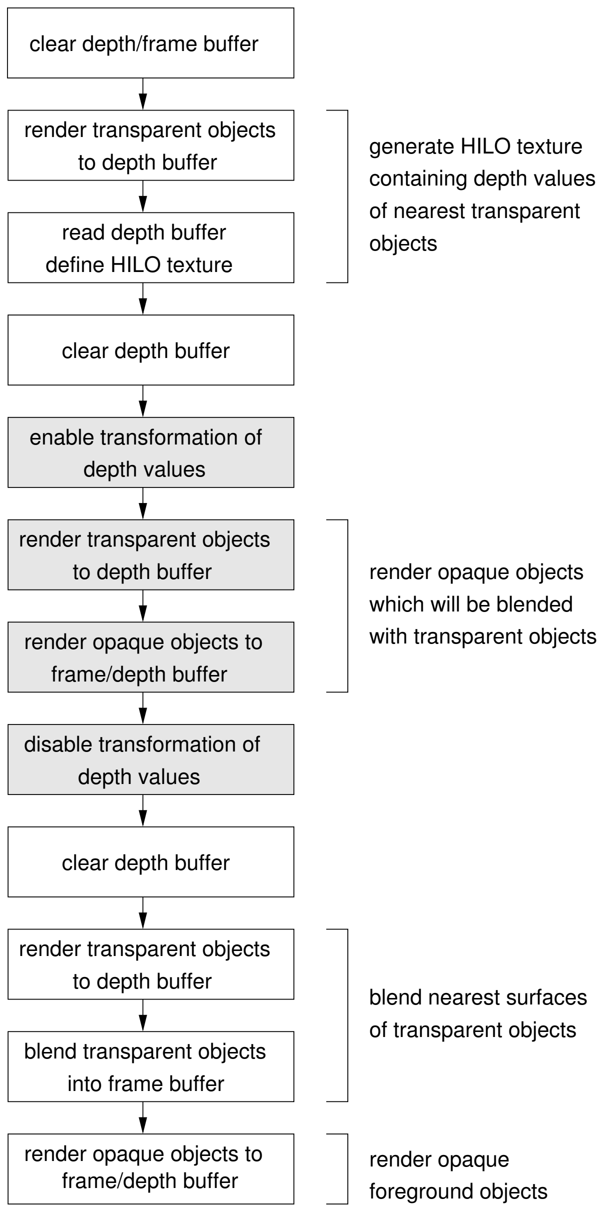

clear depth/frame buffer
render transparent objects to depth buffer
read depth buffer define HILO texture
generate HILO texture containing depth values of nearest transparent objects
clear depth buffer
enable transformation of depth values
render transparent objects to depth buffer
render opaque objects to frame/depth buffer
render opaque objects which will be blended with transparent objects
disable transformation of depth values
clear depth buffer
render transparent objects to depth buffer
blend transparent objects into frame buffer
blend nearest surfaces of transparent objects
render opaque objects to frame/depth buffer
render opaque foreground objects

Figure 4.11: Rendering pipeline for explicitly modeled inside boundaries of surrounding objects.

texel. The original 32 bit depth component can be regarded as a HILO pair of two 16 bit unsigned integers. In this way, a time-consuming remapping of depth values can be avoided—the content of the depth buffer is transferred from main memory to texture memory as is.

The next two boxes implement part three of the basic algorithm. A pixel shader program is enabled to virtually clip away all surfaces that have equal or smaller depth values than those given by the above HILO or floating point texture. This can either be done by a fragment kill operations or by replacing the z value of a fragment by $z - z_{\text{shift}}$, where z_{shift} represents the z value stored in the HILO texture. The gray boxes in Figure 4.11 indicate the scope of the depth transformation. By shifting z values by $-z_{\text{shift}}$, only the fragments with $z > z_{\text{shift}}$ stay in a valid range of depth values—all other fragments are clipped away. As a consequence, the foremost transparent surfaces are "peeled off" and only the depth values of the second-closest surfaces are rendered into the depth buffer.

The last two gray boxes implement part four of the basic algorithm. With the shader program still being enabled, the opaque surfaces and corresponding silhouette lines are rendered into both frame and depth buffer. Only the fragments lying in front of the second-closest transparent surfaces pass the depth test, i.e., only those parts that are supposed to be blended with the transparent surrounding.

Finally, the last four boxes realize part five of the basic algorithm. The first two steps once again initialize the depth buffer with the original z values of the closest transparent surfaces. Based on a depth test for equal z values, just these closest surfaces and silhouettes are blended into the frame buffer. Ultimately, the missing opaque foreground objects are rendered.

The pixel shader and corresponding vertex shader program for the depth operation is straight forward. In the vertex shader the depth value in homogenous clip coordinates is computed for each vertex from the original object coordinates (i.e., after modeling, viewing, and perspective transformations). Clip coordinates are linearly interpolated between vertices during the scan-conversion of triangles, yielding a hyperbolic interpolation in normalized device coordinates and in window coordinates. The division by w is performed on a per-fragment basis. For perspectively correct texture mapping, texture coordinates analogous to the above clip coordinates have to be used. Consequently, the vertex program may only assign clip coordinates per vertex, but not device or window coordinates. With the help of the corresponding texture coordinates which simulate a 1:1 mapping between the fragment position and the texel and its corresponding values previously stored in the HILO or floating point texture the depth value of an existent most front facing fragment can be read and compared to the current depth in window coordinate. The fragment can simply be removed by a fragment kill operation if the difference is equal or below zero. The corresponding assembly code for a Direct3D9 pixel shader 2.0 implementation look like that:

```
ps.2.0
dcl v0                // Incoming color
dcl t0                // Incoming projected homogenous object coords
dcl t1                // Incoming normal in object space
dcl_2d s0             // texture containing depth of closest frontface
rcp r1, t0.w          // homogenous divide
mul r0, t0, r1.x      // do homogenous divide for clip-space coords
add r4, r0, c29.x     // now do clip-space to screen-space transform
mul r4, r4, c29.y     // now do clip-space to screen-space transform
sub r4.y, c29.x, r4.y // correct upside down mirroring
texld r4, r4, s0      // load depth value from texture
sub r5, r0.z, r4.x    // check difference if fragment belongs to front most face
sub r5, r5, c29.w     // add an additional offset in case of inaccuracy or equal
texkill r5            // kill fragment if it's in front
```

scene	# polygons		lines?	FPS	
	total	non-opaque		I	II
4.7(c)	25,192	192	no	128.0	37.0
4.7(c)	25,192	192	yes	45.0	23.0
5(a)	70,955	3,112	no	—	2.85

Table 4.1: Performance measurements of the OpenGL implementation

```
mov oC0, v0        // output to all color channels incoming channel
```

Results

There are two implementations of transparency for technical illustrations. The first one is based on OpenGL [SA99] and on NVidia-specific extensions [Kil01]. User-interaction and the management of rendering contexts in the C++ application are handled by GLUT. The second one is implemented in Direct3D9 and uses newer hardware functionality like vertex and pixel shader version 2.0 and 3.0.

Cool & warm tone-based shading [GGSC98] is implemented as a vertex program. Tone-based shading is combined with black line silhouettes to facilitate the recognition of the outlines of both transparent and opaque objects. The silhouettes are rendered using a hardware approach, as described by Gooch and Gooch [GSG$^+$99].

Additionally, for the first implementation the silhouette edges of transparent objects are computed in 3D world / object space according to Section 4.2.2 to determine the α values. These values are calculated per vertex according to Eq. (4.6). The second version uses the silhouette approximation described earlier.

Figures 4 in the color plates show results generated by both implementation. Figure 4 (a) displays a transparent mug with two boundaries surrounding an opaque box, similarly to the scenario described in Section 4.2.2. Figure 4 (b) shows a similar scene including multiple transparent mugs which are rendered according to the rules described at the beginning of this chapter. Figure 5(a) in the color plates shows a parts of a Lancia engine block as a typical example of a technical illustration. Finally, Figure 5(b) in the color plates show the complete Lancia engine block.

Table 4.1 shows performance measurements for both depth-sorting approaches with the old OpenGL implementation. The method for implicitly modeled interior boundaries is denoted as “I”, the method for explicitly modeled interior boundaries is denoted as “II”. All tests were carried out on a WindowsXP PC with Pentium IV 2.8 GHz CPU and a GeForce4 Ti4600. Window size was 512^2. The first column refers to the figure in which the respective scene is depicted. The second and third columns contain the number of either all polygons or transparent polygons. The fourth column indicates whether silhouette lines are rendered. The fifth and sixth columns

scene	# polygons		lines?	FPS
	total	non-opaque		
4.7(c)	25,192	192	no	380
4.7(c)	25,192	192	yes	240
5(a)	70,955	3,112	no	212
5(a)	70,955	3,112	yes	126
5(b)	139,183	16,810	no	7

Table 4.2: Performance measurements of the new Direct3D9 pixel shader 2 implementation of Method II.

reflect the frame rates for methods "I" and "II", respectively. Note that method "I" does not render the scene in the last row correctly and thus the corresponding frame rate is omitted.

Table 4.2 shows performance measurements for the newer Direct3D9 pixel shader 2 implementation running on the same configuration only with a Radeon 9700Pro . In this case only method II has been implemented. The large increase in frame rates compared to the OpenGL implementation can be explained by many factors. The first one is of course, the faster graphics board. Secondly, no CPU time is spent for finding the silhouettes and the nearest silhouette for each vertex. Third, the OpenGL implementation does not use any optimizations like Vertex Objects or Vertex buffer, whereas in Direct3D9 everything has to be sent down in chunks containing an optional index and a mandatory vertex buffer. Fourth there is no read back or copy from depth buffer necessary since an extra floating point render target can be used. This is very efficient under Direct3D9. Fifth, the rendering pipeline can be simplified when using a fragment kill operation since the depth buffer is no longer trashed and therefore, some render steps shown in Figure 4.11 can be avoided.

4.2.3 Indirect Method: Explosion Illustrations

Explosion illustrations (see Figure 4.12) are a very common class of illustrations found in technical documentation because they are easy to be understood by people without area expertise. Not only do explosion drawings reduce or eliminate occlusion and expose internal parts as wanted for achieving a similar result accomplished by using transparency, but they also convey the global structure of the depicted object, the details of individual components, and the local relationship among them. Explosion drawings often have a distinct axis where the objects are moved apart from. Additionally, an exploded illustration does not consist simply of a set of parts separated in space, leader lines are often drawn between parts indicating the relationship between the exploded part and its "parent".

An automatic generation of explosion illustrations would require knowledge about how affected parts are belonging together. This problem is closely related to assembly planing [Wol89]. A really clever algorithm or approach for generating explosion illustration without major user input

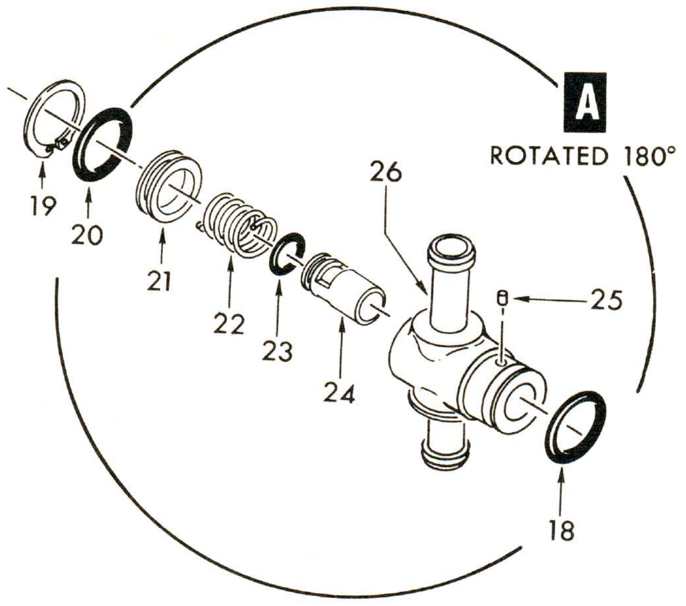


Figure 4.12: A real hand drawn explosion illustration taken from [Tho68].

was described by Raab [Raa98]. The original intention was to use it for object space distortion to magnify specific regions in a 2D or 3D scene. But it can also be used for generating explosion illustrations by selecting appropriate parameters. The method itself relies on a special spatial hierarchy in which each entry in this hierarchy is assigned a certain amount of free space it may occupy during the scaling transformation. Other semi-automatical approach have been discussed in papers [RKSZ94, DC95, LAS04, APH$^+$03].

4.2.4 Indirect method: Cutaway illustrations

Another style frequently used in technical illustrations are cutaway techniques. Cutaway drawings in technical illustrations allow the user to view the interior of a solid opaque object. In these illustrations, entities lying inside or going through an opaque object are of more interest than the surrounding one itself. Instead of letting the inner object shine through the girdling surface, parts of the exterior object are removed. This produces a visual appearance as if someone had cutout a piece of the object or sliced it into parts. Figure 4.13 shows an example of a traditional cutaway illustration. Although parts of the outermost object are removed, its shape and size is still visible. It is also possible to get an idea of the weight, solidness, and material of the outer object based on the thickness of its wall. This can be of particular interest, for example, in instruction manuals or science books. Cutaway illustrations avoid ambiguities with respect to spatial ordering, provide a sharp contrast between foreground and background objects, and facilitate a good understanding of spatial ordering. Another reason for the popularity of cutaway illustrations might

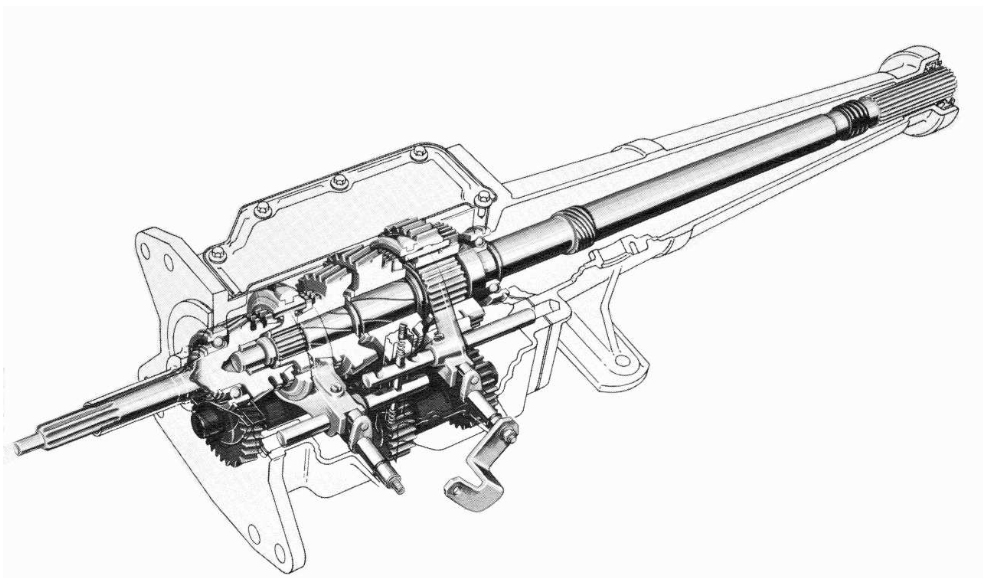


Figure 4.13: A real hand drawn cutaway illustration taken from [Tho68].

be the fact that the appearance of semi-transparent surfaces is hard to simulate with most classical drawing styles for hand-made illustrations. Previous research in this area done by Seligman and Feiner [SF91, SF93] focused on a rule based system where illustrations – including cutaway parts – can be generated automatically via a previously user-defined rule set. In contast to this rule based system in the following part of this chapter, an approach is discussed on how a nearly fully automatic generation of cutaway illustrations can be achieved.

Overview of cutaway illustrations

In this chapter a brief review on how *cutaway* illustrations are traditionally created by illustrators is given in order to extract some requirements for an automatic generation process on the computer. For detailed background information on hand-made technical illustrations in general and cutaway drawings in particular the reader is advised to refer to classical artbooks [Tho68, Mar89].

The purpose of a cutaway drawing is to allow the viewer to have a look into an otherwise solid opaque object. In comparison to direct methods inner object do not shine through their surrounding surface, instead parts of the outside object are simply removed. From an algorithmic point of view, the most interesting question is where to cut the outside object. The answer to this fundamental question depends on many different factors, for example, the sizes and shapes of the inside and outside objects, the semantics of the objects, personal taste, etc. Many of these factors cannot be formalized in the form of a simple algorithm and need some user interaction. Nevertheless, some interesting common properties can be found in many examples of traditional cutaway drawings which allow to automatically generate quite reasonable cutaways.

First it is possible to distinguish between two different subclasses of the general notion of a *cutaway* drawing: *cutout* and *breakaway*. Figures 4.14 (a) and 4.14 (b) show the difference

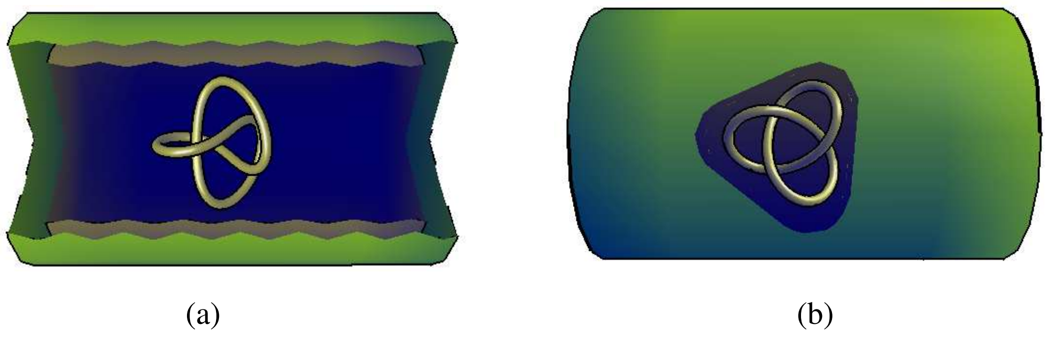


Figure 4.14: Comparison of computer-generated cutout and breakaway illustrations. (a) demonstrates the cutout technique with a jittering boundary. In (b) the breakaway method is applied to the same scene.

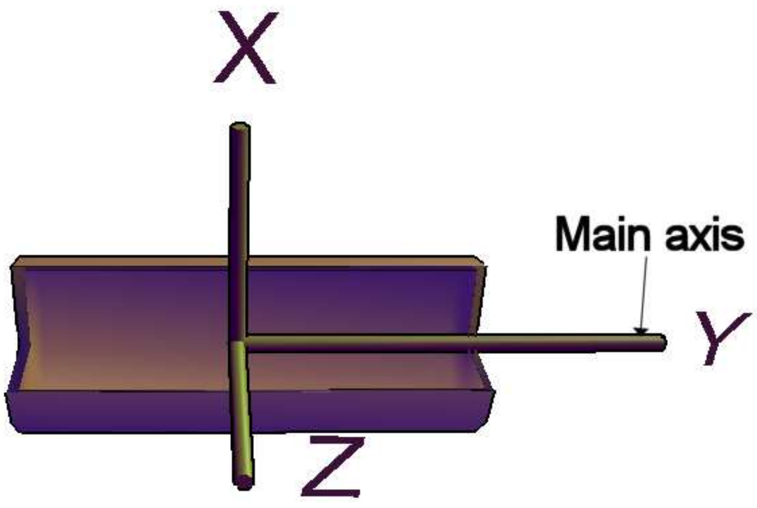


Figure 4.15: Main axis of a sliced cylinder.

between the two subclasses.

Artists and illustrators tend to restrict themselves to very simple and regularly shaped cutout geometry. Often only a small number of planar slices is cut into the outside object; in many cases just two planes are sufficient. The main reason for using simple cutout geometry is to avoid any extra cognitive burden for the viewer if the structure of the objects is already quite complex. The location and orientation of the cutting planes are determined by the spatial distribution of the interior objects and, more importantly, by the geometry of the outside body. Just enough is taken away from the outlying object to allow the observer to view the internal details. From many cutaway illustrations it is possible to conclude that two planes intersecting at an angle between 90 to 140 degrees are sufficient for a wide class of applications. Another common property concerns the location of the slicing planes. The cut through the object of interest often takes place at or around its main axis. The main axis of an object is the axis with the greatest spread see Figure 4.15. For a cutout with two planes, the main axis is identical with the intersection line between the two planes.

A cutout in a technical illustration has not always to be smooth. For example, a saw tooth-like or jittering cutting is often applied to better distinguish between outer and inner objects and produce a higher level of abstraction. Figure 4.14 (a) shows such a jittering cutout for a simple example scene. The image was generated by the computer-based method described later in this chapter. In some examples one can even find several cut-through layers where different kind of materials and thickness occur.

The cutout approach is particularly useful when many objects or large objects are inside and cover a large portion of the interior of the girdling object. In contrast, if only a few small inside objects lie densely to each other, another approach is more appropriate. Here, an illustrator rather breaks a virtual hole into the boundary to show the interior objects. This boundary hole should be just wide enough to see these objects. In this work the method is called *breakaway* —imagine taking a hammer and breaking a hole into the boundary by force. Figure 4.14 (b) shows a simple example of a breakaway illustration. The image was generated by the computer-based technique described in Chapter 4.2.4.

It has already been stated that hand-made illustrations are influenced by various aspects many of which are hard to be formalized for a computer-based processing. Nevertheless, it is possible to extract a small number of rules that lead to quite convincing, fully automatic cutaway drawings. The user is still able to change parameters after the initial automatic construction.

The first, very basic question is: Which objects are potentially subject to cutting? It can be observed that interior objects are not sliced by the cutout geometry. Cutting is only applied to outside objects. Therefore, the first requirement is:

R1 Inside and outside objects have to be distinguished from each other.

Please note that this requirement not only covers scenes with a single outside object, but has to allow for scenarios with several disjoint outside objects and even nested layers of outside objects. Another issue is the shape of the cutout geometry here it is restrict ourselves to a specific class of shapes:

R2 The cutout geometry is represented by the intersection of (a few) half spaces[1].

By construction, this cutout geometry is convex. Very often good results are achieved by:

R2' A cutout geometry that is represented by the intersection of two half spaces.

The next rule determines the position and orientation of the above cutout geometry:

[1]A half space can be represented by the plane that separates the space from its complement.

R3 The cutout is located at or around the main axis of the outside object.

For a cutout with two planes, the intersection line between the two planes lies on the main axis. The angle of rotation around this axis is a free parameter that can be adjusted by the user.

The boundary in technical illustration cutouts do not have to be as smooth as the geometry defined by a collection of half spaces. For example, a saw tooth-like or jittering cutting is often applied to such a simple cutout geometry to produce a higher level of abstraction.

R4 An optional jittering mechanism is useful to allow for rough cutouts.

Finally, the cutout produces new parts of the objects' surfaces at the sliced walls. This leads to:

R5 A possibility to make the wall visible is needed.

This requirement is important in the context of boundary representations (BReps) of scene objects, which do not explicitly represent the solid interior of walls. Therefore, special care has to be taken to make a correct illumination of cutout walls possible.

The other cutaway approach in the form of a breakaway is based on a slightly different set of rules. The first requirement (R1) for a distinction between inside and outside objects is the same as in cutout drawings. However, the shape and position of the breakaway geometry is not based on rules (R2) and (R3), but on:

R6 The breakaway should be realized by a single hole in the outside object.

If several small openings were cut into the outside surface, a rather disturbing and complex visual appearance would be generated. Nevertheless:

R7 All interior objects should be visible from any given viewing angle.

The above rule for making the walls visible (R5) can be applied to breakaway illustrations as well. Jittering breakaway illustrations are seldom and therefore (R4) is not a hard requirement for these illustrations.

In the following two sections, two different rendering algorithms are presented that meet the above characteristics for cutout and breakaway illustrations, respectively.

Implementation: Cutout drawings

Now a class of rendering algorithms for cutout drawings is presented. It is shown how a computer-based process can fulfill the aforementioned rules (R1)–(R5) for the cutout approach.

Classification It is assumed that the classification of objects as interior or exterior (R1) is provided by an outside mechanism. The problem is that a generic classification criterion solely based on the spatial structure of the scene objects is not available. For special cases, objects can be recognized as being inside or outside by observing their geometry. For example, the important class of nested surfaces can be handled by an algorithm by Nooruddin and Turk [NT00]. However, this method does not support exterior objects that have already openings before the cutting process is performed.

A more basic approach is to store the objects in a scenegraph structure; where the classification is based on an additional boolean attribute that is attached to each geometry node of the scenegraph. For practical purposes, geometric modeling of scenes should be performed in an external modeling and animation tool (such as 3D Studio Max or Maya). Since these modeling tools do not directly support the additional classification attribute, other information stored in the 3D file format has to be exploited. For example, the transparency value can be (mis-) used, or the classification is coded in the form of a string pattern in the name of 3D objects. This approach allows the user to explicitly specify interior and exterior objects and to introduce some external knowledge into the system.

Main axis Another issue is the computation of the main axes of the outside objects. Each object is assumed to be represented by a triangulated surface. Information on the connectivity between triangles is not required, i.e., a "triangle soup" can be used.

The algorithm makes use of first and second order statistics that summarize the vertex coordinates. These are the mean value and the covariance matrix [DH73]. The algorithm is identical to Gottschalk et al.'s [GLM96] method for creating object-oriented bounding boxes (OBB). If the vertices of the i'th triangle are the points $\vec{p}^i$, $\vec{q}^i$ and $\vec{r}^i$, then the mean value $\vec{\mu}$ and the covariance matrix C can be expressed in vector arithmetics as

$$\vec{\mu} = \frac{1}{3n}\sum_{i=0}^{n}(\vec{p}^i + \vec{q}^i + \vec{r}^i) \quad ,$$

$$C_{jk} = \frac{1}{3n}\sum_{i=0}^{n}(\vec{p}_j'^i \vec{p}_k'^i + \vec{q}_j'^i \vec{q}_k'^i + \vec{r}_j'^i \vec{r}_k'^i) \quad , 1 \leq j,k \leq 3$$

where n is the number of triangles, $\vec{p}'^i = \vec{p}^i - \vec{\mu}$, $\vec{q}'^i = \vec{q}^i - \vec{\mu}$, $\vec{r}'^i = \vec{r}^i - \mu$. Each of them is a 3×1 vector, e.g. $\vec{p}^i = (\vec{p}_1{}^i, \vec{p}_2{}^i, \vec{p}_3{}^i)^T$ and C_{jk} are the elements of the 3×3 covariance matrix. Actually, the vertices of the original mesh are not used, but the vertices of the convex hull. Moreover, a uniform sampling of the convex hull is applied to avoid potential artifacts caused by unevenly distributed sizes of triangles. These improvements are also described by Gottschalk et al. [GLM96]. The eigenvectors of a symmetric matrix with different eigenvalues are mutually orthogonal. The eigenvectors of the symmetric covariance matrix can be used as an orthogonal basis. Of special interest is the eigenvector corresponding to the largest eigenvalue because it serves as the main axis of the object.

CSG cutout Finally, the cut geometry has to be defined and then applied at the previously determined location. An object-space approach working directly on the geometry could be realized by techniques known from constructive solid geometry (CSG).

The intersection of half spaces (R2) can be realized by a CSG intersection operation working on half spaces. An intersection operation is equivalent to a logical “and” applied to the corresponding elements of the spaces. A half space can be represented by the plane that separates the space from its complement. Therefore, this plane serves as a slicing plane in the cutout approach. Note that the half space is defined in a way that outside objects are removed at all locations of this half space; outside objects are left untouched in the complementary space. The actual cutting process is modeled by a CSG difference operation applied to the cutout geometry and the geometry of the exterior objects. Intrinsic to all CSG operations is the creation of new boundary surfaces at cuts. Therefore, cutout walls are automatically modeled and can be displayed afterwards, as required by (R5).

In the general approach to rule (R2), the number, locations, and orientations of the cutting planes have to be defined by the user. The more restricted rule (R2’) prescribes a fixed number of two planes. Moreover, the intersection line between the planes is fixed by the main axis of the outside object. The only free parameters are the relative angle between the planes and the angle of rotation of the cutout geometry with respect to the main axis. The relative angle can be set to a default value in between 90 and 140 degrees (e.g., to 110 degrees); the default orientation with respect to the main axis can be set to any fixed angle. With these initial values, quite good results are achieved without any user interaction.

For an optional saw tooth-like or jittered cutout (R4), the cutout geometry has to be perturbed, for example, by a displacement mapping technique [Coo84]. Appropriate kinds of displacement maps are presented shortly in the description of texture-based cutouts.

Although all the requirements (R1)–(R5) can be directly mapped to a CSG-based implementation, this approach is not pursued in more detail since the main problem is that CSG boolean operations can be very time-consuming. Therefore, parameter changes are unlikely to work in real time and interactive work is not possible. This is a major drawback because cutout drawings—even if they almost work automatically—need some user interaction to adjust parameters for improved final results. If highly detailed and jittered cutout geometry or complex exterior objects are used, CSG operations become particularly time-consuming. Another issue is rendering time itself. Often a high number of new primitives is introduced by the re-tesselating steps required for precise intersections between objects. The high amount of new primitives could hinder interactive rendering times and might require further object optimization steps. Finally, from a rather practical point of view, often problems of accuracy can be experienced for boolean operations on many commercially available 3D modeling and animation tools. Quite often, “manual” cuts into complex objects lead to not-quite-correct results with respect to the geometry and, in particular with respect to the illumination of the remaining object. All these aspects limit the applicability of the CSG approach for an interactive application. Therefore, other approaches are investigated that make use of graphics hardware acceleration and are based on image-space calculations.

Planar cutout A simple image-space approach is based on the concept of clipping planes and allows for piecewise planar, convex cutouts according to (R2). Each planar element of the cutout geometry is identified by a clipping plane. The exterior object is rendered *n* times where *n* is the number of different planes. In each rendering pass, the respective clipping plane is activated. Afterwards, the interior objects are rendered in a single pass, with clipping planes being deactivated.

The advantages of the clipping-plane based method are its rather simple implementation and its support by virtually any graphics hardware. For example, client-defined clipping planes are already available in standard OpenGL 1.0. A drawback is the increase in rendering costs for multiple rendering passes—especially for more complex cutouts with several cutting planes. Another issue is the restriction to slick cutouts. Jittering boundaries according to (R4) are not possible. Since no explicit modeling of the cutout surface is implemented, the wall cannot be made visible (R5).

Cutout via Stencil test The following screen-space technique exploits the stencil buffer and stencil test to represent the cutout geometry. The advantage of this method is the fact that jittering boundaries are supported. On the other hand, it is restricted to convex exterior objects. Rappoport and Spitz [RS97] demonstrate that a related stencil-based approach can be used to make possible interactive boolean operators for CSG.

The algorithm is similar to an implementation of shadow volumes by means of the stencil buffer [Kil99]. The following extensions are needed for stencil-based cutouts. First, the algorithm has to affect the visibility of objects and thus their z values. Therefore, a mechanism to adjust depth values has to be incorporated. Second, the algorithm has to allow for front and back faces of the exterior convex object because the line-of-sight (from the camera) may have intersections with one front and one back face of the exterior object. The core algorithm is applied twice: once for front and once for back faces.

The details of the rendering algorithm are as follows. In the first step, the front face of the exterior object that should be cut is rendered to the depth buffer; the color buffer is masked out. Afterwards the front faces of the cutout geometry are rendered with the depth test being activated, but without changing the depth buffer entries. A stencil operator increases the stencil value by one each time a fragment passes the depth test. Similarly, the back faces of the cutout geometry are rendered without changing the depth buffer. This time the stencil value is decreased when a fragment passes the depth test. This ensures that pixels of the front face lying inside the cutout geometry have a stencil value of one. All other pixels have a stencil value of zero. Note that we have assumed that the camera is not located inside the cutout geometry. If the camera is within the cutout geometry, the stencil buffer needs to be initialized to one.

In the next step, the depth buffer is cleared and the back face of the exterior object is rendered to the depth buffer in the regions where the stencil value is greater than zero. Like in the second step, the front faces of the cutout geometry and then the back faces of the cutout geometry are rendered with depth test. The stencil value is increased for front faces passing the depth test and decreased for back faces passing the depth test. It is now possible to decide for each pixel if only the back face of the exterior object (stencil value one), both the back and front faces (stencil value zero) or neither (stencil value two) are visible. For the final step the depth buffer is cleared and then the back face of the exterior object is both rendered into depth and color buffers in regions where the stencil value is one. Afterwards the front face of the exterior object is rendered to depth and color buffers in those parts of the screen where the stencil value is zero. Finally, the interior objects are rendered without stencil test.

The advantage of this approach is that complex cutout geometry is supported. In addition, the number of rendering passes does not increase with the complexity of the cutout geometry. Stencil buffer and stencil test are already included in standard OpenGL 1.2. Therefore, the algorithm is widely supported by graphics hardware. Unfortunately, the structure of the exterior object is subject to an important restriction. The object has to be represented by a single convex surface without boundaries. The above algorithm makes explicit use of the fact that (at the most) one front and one back face is cut by a ray originating from the camera. However, many technical 3D data sets contain nested surfaces or explicitly represent all boundaries—both inside and outside. This means they can have more than just one front and back face intersecting the same line of sight. In this case the above algorithm fails because it is no longer guaranteed that all back faces of the exterior object lie behind the cutout geometry. Moreover, the cutout walls are not modeled (R5).

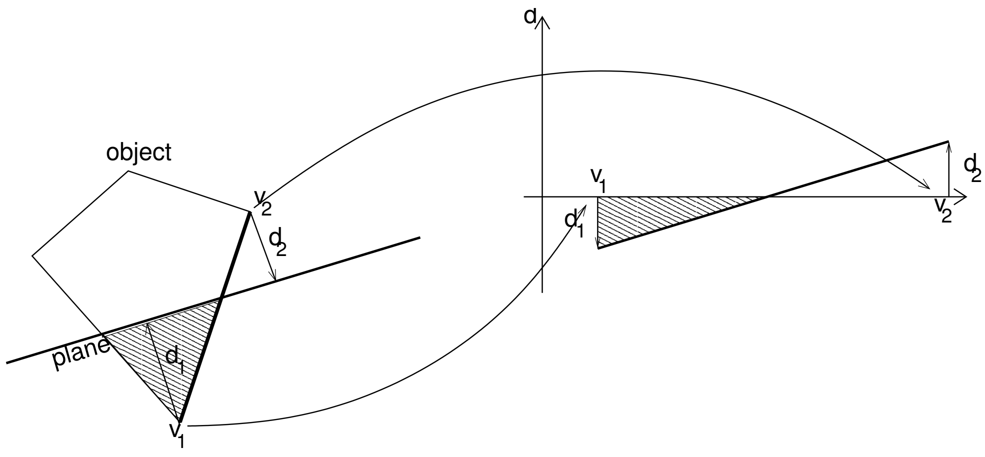


Figure 4.16: Planar cutout based on a linearly interpolated signed distance.

Texture-based cutout To overcome the restriction to convex exterior objects a new rendering algorithm is presented that exploits texture mapping to represent the cutout geometry. The implementation requires programmable transform and lighting, per-fragment operations, and multi-texturing.

First, the basic idea of the approach is illustrated by restricting ourselves to a single cutting plane. The scenario is depicted in Figure 4.16. Let us consider the required operations and tests to allow for a cut into a single triangle. The decision whether a fragment of the triangle lies inside the clipped half space or in the complement space is based on the signed Euclidean distance of the fragment from the plane. It is defined such that fragments with a negative distance d are clipped and fragments with a positive value d pass. The signed distances are computed for each vertex and then interpolated across the triangle to obtain values for each fragment. The per-vertex distances can either be computed on the CPU or by a vertex program in the transform and lighting part of the rendering pipeline. The necessary parameters for the plane equation of the cutout can be provided by passing vertex program parameters.

In this first approach, the signed distance is communicated from the transform and lighting part to the rasterization part of the rendering pipeline via the α value of the primary color. Since color values are restricted to a range $[0,1]$, a transformation of the signed distances is applied so that $d = 0$ (fragment is on the plane) corresponds to $\alpha = 0.5$. During rasterization, the α value is interpolated by scanline conversion to provide a value for each fragment. Finally, the alpha test is enabled to reject fragments with $\alpha < 0.5$, i.e., fragments that lie inside the cutout half space are not written into the depth and color buffers.

Although this α-based approach clearly illustrates the basic idea behind the texture-based cutout algorithm, it has several severe drawbacks that disqualify the α-based method for a real application. The α-based technique can only produce slick cutouts; jittering cutouts are not possible. The α value is very often restricted to a limited resolution, such as 8 or 12 bit. This resolution is inappropriate as a measure for distances and might produce step-like artifacts. Another problem is the interpolation method during scanline conversion. The true distance value d is achieved by a linear interpolation of the vertices' distance values with respect to the coordinate system of the plane. Since affine transformations are compatible with linear interpolations, d could be as well linearly interpolated in world or eye coordinates. Scanline conversion, however, works in screen space—after perspective transformation. Therefore, a linear interpolation of α yields wrong values for the signed distance.

A better solution is to store the signed distance in a texture coordinate. Texture coordinates are better suited than color-components for the following reasons: on most GPUs texture coordinates are implemented as floating point numbers, they are not restricted to the range $[0,1]$, and they have a much higher accuracy. Moreover, texture coordinates can be interpolated in a perspectively correct manner, i.e., a hyperbolic interpolation in screen space corresponds to a correct linear interpolation in object space.

The original signed distance d_{plane} can also be perturbed to allow jittering cutouts. The idea is based on displacement mapping techniques [Coo84]. Figure 4.17 illustrates the displacement of the signed distance to a cutting plane. The final distance is computed by fragment operations according to

$$d = d_{\text{plane}} + d_{\text{perturb}} \quad . \tag{4.7}$$

The perturbation d_{perturb} is stored in a 2D texture and is superimposed onto the original distance. Good visual effects can be produced with fractals, synthetic procedural textures used for clouds and virtual terrain height fields, and real terrain data. In particular, a saw tooth-like boundary

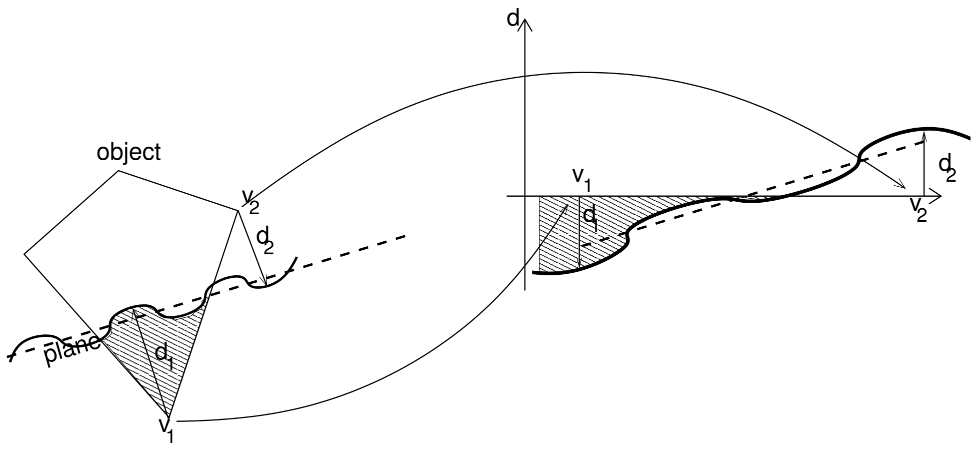


Figure 4.17: Jittering cutout. A perturbation function displaces the original distances to the cutting plane.

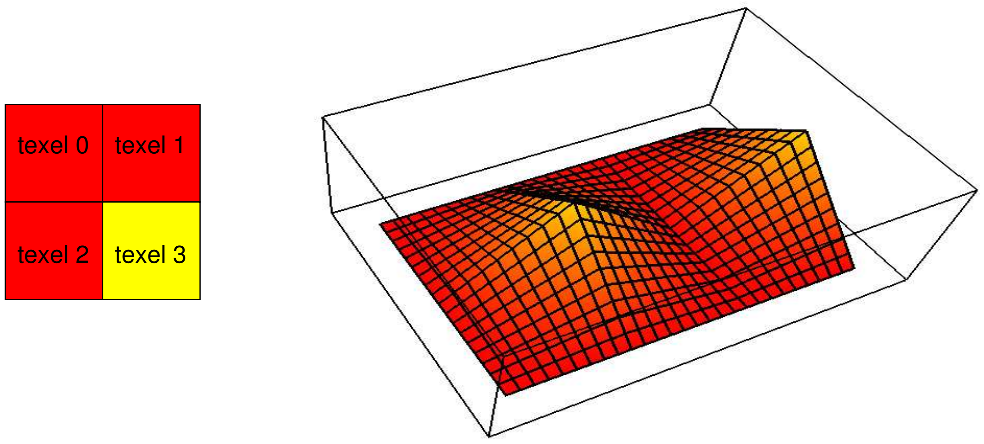


Figure 4.18: A tiny 2×2 example texture used for sawtooth-shaped boundaries.

can be achieved by a basic and memory friendly perturbation texture. Just a tiny 2×2 texture as illustrated in Figure 4.18 is needed. Thanks to texture repeating and bilinear texture interpolation a repeated falloff is generated that leads to the desired visual effect. Figure 4.14 (a) shows an example of a cutout illustration based on this 2×2 perturbation texture.

In the last step a fragment clipping operation has to be executed according to the corresponding distance value. If the value is below zero the fragment has to be clipped, otherwise kept. This can either be done through a texkill command or by setting the alpha value and using the alpha test.

So far only a single perturbed cutout plane is supported. The above texture-based algorithm can easily be extended to several cutout planes. A separate texture coordinate is used for each plane to store the respective signed distance. The same perturbation is applied to each distance value. An additional fragment operation determines the minimal absolute distance value. The following approximation can be used to avoid several texture coordinates and the additional fragment operation. The minimal absolute distance value can also be computed per vertex and the according signed distance can be used as the only texture coordinate. This approximation yields correct results for most cases. If, however, the three different vertices of a triangle do not have the same closest plane, inaccuracies are introduced by interpolating signed distances that are attached to different planes. The smaller the triangle, the smaller is the possible error.

The presented texture-based cutout algorithm meets the requirements (R1)–(R4). The geometry of the interior and exterior objects is not subject to any restrictions. Since the algorithm can be mapped to graphics hardware, interactive frame rates are possible even for complex illustrations. The only drawback is the missing modeling of the cutout walls (R5).

Breakaway illustrations

In this part, a rendering algorithm that meets the requirements for breakaway illustrations is presented. A computer-based process that can fulfill the breakaway-specific rules (R6) and (R7) is discussed. The classification of objects as interior or exterior (R1) is provided by the same mechanism as for the cutout technique.

The basic idea is to clip away those parts of the surrounding object that would otherwise occlude the interior objects as seen from the camera's position. Therefore, this approach is intrinsically view-dependent and allows for (R7). The other requirement is that only a single hole is cut into the exterior object (R6). The convex hull of the interior objects is used as a basis for breakaway illustrations. Just enough is removed from the outside object to make this convex hull visible. The convex hull has two advantages. First, it contains all interior objects. If the convex hull is visible, all interior objects are visible. Second, the projection of the convex hull onto the image plane always yields a convex geometry and cannot contain any holes.

The algorithm for breakaway illustrations works as follows. In a preprocessing step, the convex hull of the interior objects is computed, for example, by the Quick Hull algorithm [BDH93]. The convex hull is extended in all directions by some additional spatial offset. In this way, all interior objects are enclosed with a non-zero minimum distance to the hull. During the actual rendering process the extended convex hull serves as a virtual clipping object. Only those parts of surrounding objects that are not in front of the convex hull are rendered. This is achieved by using the foremost part of the convex hull as a clipping object. Finally, the interior objects are displayed.

The crucial point of the algorithm is the clipping at the foremost surface of the convex hull. A mechanism to clip away objects in front of an arbitrarily shaped object has to be employed. A clipping algorithm that is very similar to one of Diepstraten et al.'s [DWE02] techniques for rendering transparent surfaces and described in more detail in the previous Chapter is used. Alternatively, Everitt's *depth-peeling* [Eve01] could also be used. Both algorithms can be implemented on a GeForce 3.

Figure 4.19 shows the details of the rendering algorithm. In this case the terminology from the standard OpenGL specification and NVidia-specific extensions for the GeForce 3 are used.

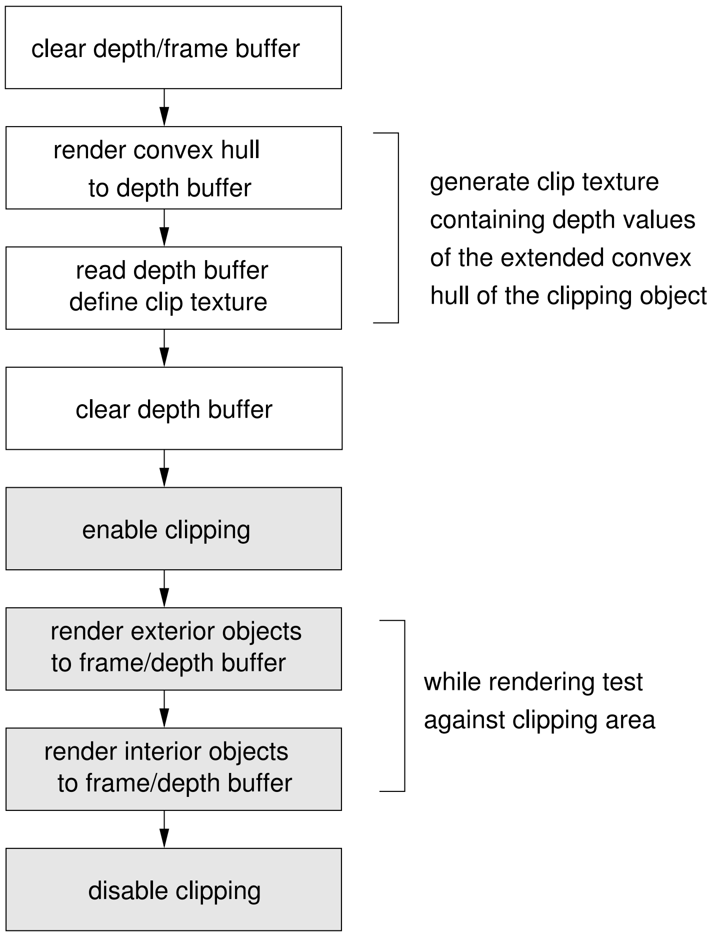

clear depth/frame buffer
render convex hull to depth buffer
read depth buffer define clip texture
clear depth buffer
enable clipping
render exterior objects to frame/depth buffer
render interior objects to frame/depth buffer
disable clipping
generate clip texture containing depth values of the extended convex hull of the clipping object
while rendering test against clipping area

Figure 4.19: Rendering algorithm for the breakaway technique.

In the first four boxes, the clipping area in screen space is determined. First, the depth buffer is initialized with a depth of zero (which corresponds to the near clipping plane of the view frustum). Then, the extended convex hull is rendered twice into the depth buffer. The first rendering pass uses "greater" as logical operation for the depth test, the second rendering pass uses the standard "less" depth test. In this way, the depth buffer contains the depth values of the foremost parts of the convex hull. Furthermore, the depth buffer is still initialized with zero in the areas that are not covered by the convex hull. In the third box, the depth values are transferred into a high-resolution or floating point 2D texture. Alternatively this texture can be used as render target. Then the depth buffer is cleared.

In the fifth box, a fragment program is enabled to virtually clip away all fragments that have equal or smaller depth values than those given by the above clip texture. Then the exterior object is rendered into the frame and depth buffers; all parts in front of the convex hull are clipped away. Finally, the interior objects are displayed and the fragment program is reset to the standard configuration.

This algorithm can take into account more than one "cluster" of interior objects by computing several corresponding convex hulls and rendering them into the depth buffer or floating/HILO render target in step two of the rendering pipeline. It is also possible to allow for surrounding objects with boundary surfaces of finite thickness. Here, separate clipping objects have to be defined for the front and for the back face of the boundary. By choosing a smaller clipping geometry for the back face, it is possible to imitate the effect of cutting through an object of finite wall thickness. In this way, requirement (R5) for visible walls can be met in parts.

Figures 6 (a)–(c) in the color plates show that the cutout illustration can be rendered independently of the drawing style. The pictures show a cutaway drawing of a Lancia Delta 5 engine block using three different rendering styles. Additionally, a 512×512 shadow map is used to simulate shadow casting inside the interior parts, the silhouettes are extracted by using an image filter approach. As filter a 3×3 kernel size Sobel filter is used.

cutaway	lines?	FPS
cutout	NO	18.7
cutout	YES	9.3
breakaway	NO	15.3
breakaway	YES	9.4

Table 4.3: Performance measurements for the Lancia engine model in frames per second. The tests were carried out with the OpenGL Geforce3 specific implementation.

Table 4.3 shows performance measurements for both approaches with the different rendering styles using an OpenGL implementation that uses specific extensions for the Geforce3/4 series. All tests were carried out on a Windows XP PC with Pentium IV 2.8 Ghz CPU and GeForce 4 Ti 4600; viewport size was set to 512×512. The test scene is depicted in Figures 6 (a)–(c) and contains 140113 triangles. Note that no special accelerated data structures like vertex arrays or display lists were used in the measurements. Additionally, both algorithms have been implemented on Direct3D9 using vertex and pixel shader 2.0.

cutaway	lines?	FPS
cutout	NO	148
cutout	YES	92
breakaway	NO	425
breakaway	YES	137

Table 4.4: Performance measurements in frames per second. The tests were carried out with the Direct3D9 pixel shader 2.0 implementation.

Table 4.4 shows the performance measurements for these. They were carried out on a Windows XP PC with Pentium IV 2.8 Ghz CPU and ATI Radeon X800XT; viewport size was also set to 512×512. The difference in the gap between cutout line and breakaway line without rendering the silhouettes, is simply because the type of implementation is different. For the cutout implementation an image space filter was used while for the breakaway implementation additional render passes were carried out to render the back faces in line mode. Interesting to note is also that in Direct3D9 the breakaway method is faster than the cutout while in OpenGL it is the opposite. This is related to the missing copying of data from the depth buffer which is not necessary since the depth values of the convex hull are rendered into an additional floating point target and therefore the later Direct3D9 implementation requires less rendering steps.

4.2.5 Hybrid Method: Ghosting illustrations

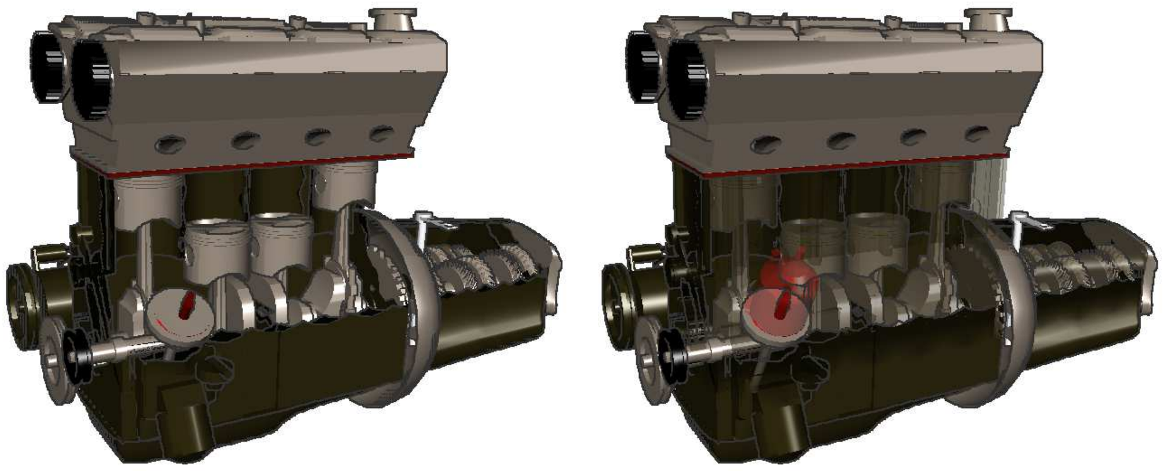

(a) (b)

Figure 4.20: Difference between (a) cutaway and (b) ghosting technique for visualizing transparent surfaces in technical illustrations.

A variation of cutaway illustration can be found as so-called ghosting, ghosted view, or phantom view illustration. They can be classified as hybrid methods as they have characteristics from both illustration groups – direct and indirect. The purpose of ghosting illustrations is the same as with cutaway illustrations to reveal internal components of an object. This is achieved by fading out the exterior skin of the object. From an artist point of view, a ghosted illustration is more complicated than a cutaway illustration. Since ghosting an object requires completion of the entire outside and inside of the subject before the fading or “ghosting” process takes place. The decision to ghost or cutaway is mostly a stylistic one. Although more internal information may be visible in a cutaway version, a ghosted illustration will favor the exterior of the subject which may contain important components, body features, logos, etc. Figure 4.20 shows the visual difference between a cutaway and ghosting illustration for the same scenario.

Since ghosting is very similar to cutaway it basically shares many of the characteristics of cutaway illustration or more particularly: the distraction between interior and exterior object is necessary as well, the ghosting area is often basically represented by the intersection of (a few) halfspaces which are often located at or around the main axis of the outside object. Even the optional jittering can be observed by not having a clear area of ghosting. Only the necessity to visualize the wall is not that important. Additionally, a new rule is added, that the level of transparency increases to a certain maximum level as the exterior skin lies further inside the cutting volume.

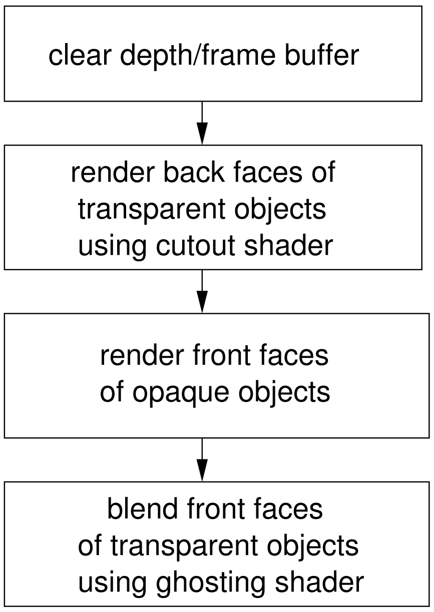

clear depth/frame buffer
render back faces of transparent objects using cutout shader
render front faces of opaque objects
blend front faces of transparent objects using ghosting shader

Figure 4.21: Rendering algorithm for the ghosting technique.

As the rules are nearly identical to the cutout approach, it is possible to produce convincing ghosting illustrations with some modifications in the rendering algorithm. The rendering algorithm consists now of the following steps (see Figure 4.21). In the first step, the transparent surfaces are rendered as with the cutout technique with the same vertex and pixel shader programs, the only difference is that just the backfaces are rendered. This is necessary in order not to blend with the background color when the transparent layer surface will be rendered. Otherwise a back to front drawing order must be ensured which would require a depth sorting of each primitive in the mesh. This is also a major difference to the direct visualization method described in Chapter 4.2.2 since backfaces actually do shine through in ghosting illustrations. In the second step the opaque objects are rendered as usually. In the third and final step the actual "ghosting" process takes place. This is achieved by rendering the transparent objects again, but this time only the front surfaces and using a different pixel shader program with alpha blending activated using the standard additive blending equation. For the alpha value either a scaled version of the original signed distance d_{plane} is used or the perturbed one using Equation (4.7). The scaling should shift the alpha value into the desired range. It is also possible to set a maximum transparency value to prevent that parts of the exterior skin lying in the middle of the ghosting volume to become completely invisible. Performance measurements using the same Direct3D testing environment as for the cutaway implementation showed 140 FPS for the normal rendering without rendering outlines of the objects and 88 FPS with rendering outlines of the object. This means the ghosting method is almost as fast as the cutaway cutout approach, at least in the given scenario. Figure 8 in the color plates show two result images using different surface rendering styles. Even a combination of render styles which is very common in technical illustrations is illustrated.

4.3 An Extension to Graphics Hardware

One might have noticed that for the cutout technique one rule could not be maintained with the described texture based technique at least not for objects having an implicit boundary. This requirement is the visibility of the wall. It is impossible to fulfill this criteria since fragments can only be destroyed and not shifted to a different position with current graphics hardware. To overcome this limitation an extension to current programmable rasterization hardware is proposed. It is shown that this extension does not only help to overcome the limitation in the algorithm of the previous chapter but also allows many other possibilities. The original results of this joint work with Daniel Weiskopf and Martin Kraus was first published in [DWKE04].

Past innovations in the area of desktop computer graphics were and still are mostly driven by the computer games industry and their special demands [Kir98]. It is likely that this will, in parts, change in the future as manufactures are looking into different markets to sell their hardware. These markets might have different demands compared to the ones of the gaming industry. The extension provided in this chapter is simple but effective to current programmable rasterization hardware and should be able to broaden the spectrum of supported algorithms. Programmable relocation of a fragment in the fragment processor makes it possible to implement a completely new class of algorithms, or alternative implementations of currently used algorithms. Experiences of the recent past show that increasing functionality of graphics hardware stimulates researchers in the visualization and computer graphics community to exploit hardware in ways that were never considered by the manufactures in the first place. An early example is the work by Heidrich et al. [HS99]; recent examples are works by Carr et al. [CHH02], Purcell et al. [PBMH02], Krüger and Westerman [KW03], and Hillesland et al. [HMG03].

4.3.1 Architecture

The vragment extension fits very well into the structure of the rendering pipeline on current graphics hardware. Figure 4.22 schematically shows the proposal for a modified pipeline. Green and blue boxes show elements that are already present on today's GPUs and that are (almost) not changed by our extension. The green boxes indicate parts that are related to vertex and primitive processing; the blue boxes concern operations on fragment and pixel level. The additional parts for the vragment extension are marked red.

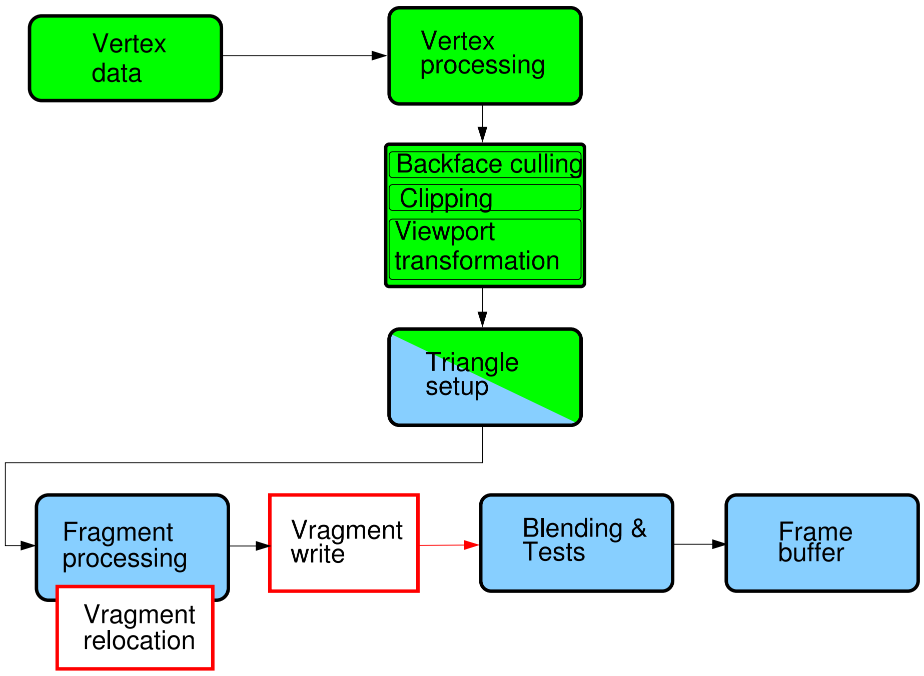


Figure 4.22: Structure of the new proposed rendering pipeline.

The “vragment relocation” module is an add-on to the existing fragment processing unit. It only slightly modifies this unit: A readable and writable register that describes the x and y coordinates of the current fragment in window coordinates is added, along with all the functionality that is available for other registers. Since GPUs already provide numerical operations with floating-point accuracy for a large number of registers, this modification nicely fits into the existing architecture.

The second change concerns the “vragment write” module. This module actually writes the vragment information from the fragment processing at the position previously computed through the vragment relocation module. Before writing, another viewport clipping has to be executed as vragments can be freely moved by a fragment program and maybe even outside the viewport, otherwise memory page faults might occur. Writing pixels in a random order will definitely not come without a penalty because writing vragments to the frame buffer would require a random write access to memory. An exact calculation of the costs for a random write access would require a complete emulation of todays memory technology and caching strategies. Unfortunately, most of them are unknown to outsiders due to intellectual property of different companies. But estimations can be extracted from texture-indirection or dependent texture reads, which are already possible with todays graphics hardware. They provide a random read access to memory and several sources show that dependent texture fetches lead to a significant performance decrease [Rig03, Spi01, DE04a]. On the other hand new memory technology like GDRIII, Rambus and an increasing popularity of dependent texture reads – for example in shading calculations – will hopefully help to decrease this bottleneck in the future. The vragments extension might also benefit from this as the same strategies to speed up random memory reads could also be used for writing. A special treatment could be beneficial when fragment positions are not changed at all (i.e., standard render code) because the same access mechanisms as used in today's graphics boards can be applied. If fragment positions are changed in a coherent manner (e.g. moving in similar directions), write-caching strategies will greatly reduce memory accessing penalties.

4.3.2 A Simulator for Vragments

The simulator for the programmable relocation of fragments utilizes both hardware and software support. It is based on Direct3D9. This API was chosen in contrast to OpenGL or a completely self-written software rasterization, because:

- The extension has only little impact on the rest of the graphics pipeline, and therefore only slight changes to existing GPU support should be introduced, ruling out a self-written software rasterization.
- Only minimal changes to standard pixel shader code is required; thus, the extension does not require modifications of existing fragment programs.
- All features of current graphics chips can be used in the simulator.
- High resolution and / or floating-point frame buffer formats and textures can be accessed without using any special extensions.
- It is easy to switch between a completely software-based and a hybrid software / hardware-based simulation.

The simulator affects the use of the Direct3D API in two ways: first, a parser translates pixel shader code with our extensions to standard Direct3D9 in order to generate a hardware-supported pixel shader program. Second, the actual rendering of primitives has to be called through special routines of the simulator.

Extension to pixel shaders

A few extensions are added to pixel shader versions 2.0 (ps2.0) and 3.0 (ps3.0) to provide control over rasterization positions to a shader programmer. Ps3.0 already provides read access to a fragment's raster position through the register vPos. For consistency, this register name is used in our extensions as well. Note that vPos only stores the x and y coordinates, the z coordinate is stored in oDepth. New version commands – vss.2.0 and vss.3.0 – are introduced to distinguish the shaders from the standard pixel shader versions. The final position of a fragment is set by a mov instruction applied to the vPos register. Also, the vPos register has to be explicitly declared at the beginning of the shader. A valid vragment shader only allows one mov to vPos, which is consistent with the usage of all other output registers oCn (n describes the output target) and oDepth in ps2.0 and ps3.0. The write operation acting on vPos can be freely placed within the shader code, with one exception: it has to be located before the output to the color buffer, i.e., before mov oCn. A tiny vragment shader code in assembler language looks like this:

```
vss.2.0                      // vragment shader version 2.0
dcl v0                       // declare input color
def c0, 10.0, 0.0, 0.0, 0.0  // constant describing a shift
dcl vPos.xy                  // declare fragment position register
add r0, vPos, c0             // add 10 pixels to the x-pos
mov vPos, r0                 // set new fragment position
mov oC0, v0                  // set fragment output color
```

The changes to a conventional Direct3D9 pixel shader program are minimal; a programmer can build on her or his knowledge of GPU programming.

Constraints

The current simulator is subject to some restrictions and constraints concerning shaders. First, the number of available instruction slots is decreased by two for both pixel shader versions because additional operations are required to communicate fragment positions to the simulator. One instruction slot is used for a mov instruction that writes the fragment position to a render target. 16 bit integer targets are used to temporarily store these positions. Since the supported range of values is restricted to the interval [0,1], the coordinates are mapped to this range before they are written to the render target. This operation uses the second additional instruction slot. Moreover, the number of available constant registers is decreased for both pixel shader versions because one constant register is needed to store the user-specified multiplication factor for the above mapping. Finally, one additional texture register is lost for vss.2.0 because ps.2.0 does not provide a readable rasterization position. Therefore, this position register is emulated by a vertex shader program that transfers the window coordinate position of each vertex to texture coordinates. Corresponding positions are obtained for each fragment by linear interpolation during scan conversion.

4.3.3 Benefits of Vragments

An extension to graphics hardware definitely stands or falls with its usefulness for programs running on this hardware. If the only benefit of this extension would be to fix a problem in one algorithm it will likely not be considered. Therefore, several scenarios and algorithms are described and discussed in which this extension is valuable. This collection of algorithms is by no means exhaustive, but rather presents a small glimpse of the possibilities offered by this extension.

Binning algorithms

Collecting and counting algorithms, or in general binning algorithms, are a completely new class of algorithms that become possible on GPUs. This class plays a fundamental role for many tasks in image analysis, image manipulation, and computer vision [SLS00]. Basically, all these algorithms take information given on an input texture and reorder this data into bins. Since the number of bins usually is smaller than the number of input texels, a compactification is achieved. As an example, a histogram of gray values in an image can be computed by a vragment shader program. The image is represented by an input texture. A quadrilateral is rendered with the same size as the input texture, establishing a one-to-one mapping between generated fragments and input texture. Without the vragment extension, these fragments would be rendered at the same position as the input texels. A vragment program, however, allows us to move the fragment to another location that corresponds to the gray value of the image texel. For example, the n bins can be organized linearly in the first column of the output image. Here, the mapping takes the gray value v [0,1] and yields the output position (x,y) = (0, v n). The output color is set to a constant value that represents the integer value one for the output render target. A histogram counts the number of elements belonging to a bin. This counting operation is implemented by additive blending. The vragment shader for this task looks like this:

```
vss.2.0                     // vragment shader version 2.0
dcl_2d s0                   // define sampler
dcl t0                      // define texture coordinate
dcl vPos.xy                 // declare fragment position register
def c2, 0.0, 0.0, 0.0, 0.0  // set some constants
texld r0,t0,s0              // load texel
mul r1, r0.x, c0.x          // compute luminance
mad r1, r0.y, c0.y, r1      // of texel
mad r1, r0.z, c0.z, r1
mul r1, r1, c1.x            // map to corresponding bin
mov r1.yzw, c2.x            // zero y position
mov vPos, r1                // set new fragment position
mov r2, c1.y                // set counter increase
mov oC0, r2                 // write counter increase
```

A more sophisticated counting algorithm allows for the computation of the entropy of an image. The image entropy can be deduced from the entropy of a discrete random variable, which is defined as

$$\begin{aligned} H(x) &= -E_X[\log P(X)] \\ &= -\sum_{x_i \in \Omega_x} \log(P(X = x_i))P(X = x_i) \quad , \end{aligned}$$

where $P(X)$ can be derived from a previously computed histogram. Again, the summation over several inputs is supported by the vragment extension.

Displacements

Other techniques that become possible with freely relocatable fragments are displacement mapping [Coo84] and procedural texture synthesis for hyper texture [PH89]. For doing displacements a vragment shader simply maps each input fragment to a new output coordinate by adding a directional offset to its 3D world coordinate. Afterwards this new world coordinate is transformed back to the window coordinate system. The 3D coordinates of a fragment can be accessed for example by storing the 3D world coordinates at each vertex as texture coordinates. The advantage compared to normal triangle-based displacement mapping approaches is: no time-consuming retesselation is required. The necessary triangle setup stays the same as no new triangles have to be sent down to the pipeline. This also means that the workload does no longer depend on the complexity of the displacement input map but rather on the projection of the displaced object. Displacement mapping is possible with graphics hardware which is capable of vertex shader 3.0, but for good results it requires a fine tessellated surface. But not only full 3D displacements are possible; also 2D displacements in window coordinates can be implemented or even a combination of both. Figure 9(a) in the color plates show the displacement of the 2D stain geometry that is rendered to a texture and then used again as input for a 3D displacement map on the paper surface. Unfortunately besides these advantages there are also some drawbacks. The vragment approach works only well with 3D displacements that have a rather small magnitude otherwise the newly calculated distance between pixels will be too large and holes are starting to pop up. When trying to close these holes a certain overdraw has to be ensured on the input fragment side. At some point the expected cost of tessellation and vertex processing will be lower than the cost of overdrawing. But for displacements with small magnitudes, for example crumpled paper or objects covered by a thin snow layer, this approach works well.

Particle systems

Vragments could also be exploited for simulating and rendering a system of particles [Ree83]. Today, very complex particles systems are still mainly managed and simulated by the CPU and transferred through the entire rendering pipeline. However, latest graphics hardware functionality allows certain types of particle systems [KSW04]. Vragment programs offer an even higher flexibility by the advantage of computing the motion of particles and the final rendering only within the fragment processing stage. Therefore, data transfer between CPU and GPU is obsolete, and the vertex processing is unburdened. In a vragment approach, the particle system is represented by textures that store the current position and any other property of each particle required for simulation and rendering (e.g., color, age, mass, momentum). One part of the vragment program computes the animation by solving the equations of motion on a per-texel basis. This task does not require the vragment extension because each texel is mapped to the very same element of the texture representing the subsequent time step. In the second part, particles are rendered to screen. Rendering requires the extension to move the fragment to the current position of the particle, as previously computed by the simulation step.

Flow visualization

The study of flow fields plays a decisive role in many scientific disciplines, such as computational fluid dynamics (CFD) or meteorological and oceanographical simulation. An important class of visualization techniques computes the motion of massless particles transported along the velocity field to obtain characteristic structures like streamlines, pathlines, or streaklines. Often a dense representation by these lines is chosen. These dense representations are typically implemented by textures [SMM00]. Texture advection is a very popular way of implementing this approach on todays graphics hardware: dense texture representations are moved along the velocity field of the flow. Due to the very nature of graphics hardware, the texture representation for a new time step is based on lookups in the texture of the previous time step. Stated differently, a backward mapping along the velocity field has to be applied. In some implementations, such as Lagrangian-Eulerian advection [JEH00], this causes problems because input noise might be duplicated into several texels of subsequent time steps, leading to a degradation of image quality. A vragment approach makes possible a forward mapping: texels of the dense representation can actively move to their position at the following time step. This process can be regarded as a special case of a particle system in which the motion of massless particles is governed by the velocity of the flow. Forward mapping avoids texel duplication because a single texel is only mapped to another single texel. Typically, not all texels are touched by forward mapping; however, these gaps can be filled with new input noise.

Automatic LOD for fur and fur-like materials

Rendering fur can be very time-consuming, as a lot of individual primitives have to be sent down to the pipeline and transformed and finally rendered. Therefore, Level of Detail (LOD) algorithms for fur and other fur-like materials can release some of the strains lying on the graphics pipeline. One approach would be to use volume textures [LPFH01]. Although this approach can solve some of the problems, it unfortunately introduces new ones. It is well known that rendering volume textures forces a very high memory demand and a high rasterization workload on the graphics board. Using vragments could solve both problems and even provide an automatic LOD mechanism for free. The idea is closely related to the previously mentioned particle systems. As with particles, the geometry of one single hair, brin, blade of grass, etc. is stored in a texture. The size of this texture defines the maximum detail of a single element. While rendering, a furry surface contains a repeated pattern of this texture together with a second offset texture that displaces each individual element to its final position in 3D world coordinates. Whenever the viewer gets closer to the furry surface more vragments will automatically be spent for each individual hair because more texels from the hair primitive are read. In contrast, when the viewer moves away from the surface, only a few vragments will be spent for each individual hair because less texels are read.

Arbitrary curves

Rendering curves, for example Bézier, Hermite, and arbitrary spline curves, is still an essential task in Computer Aided Design. In todays commercial CAD packages curves are either rendered in software or split into line segments and then rendered as individual lines. With the help of vragments the rendering of simple line primitives can be individually changed to rendering directly any arbitrary curve by simply drawing a line primitive, evaluating the desired curve equation inside the vragment program, and place the vragment to its calculated position.

Special effects

An interesting class of applications for vragments could be special effects, in particular for games. Game applications do not have time to spend a lot of computation power on special effects. The goal is visual plausibility, not physical accuracy. In this case vragments are especially well suited for effects where the connectivity of a surface is extremely complicated and time-varying. Examples for this class are explosions, implosion, arbitrary object deformations, cracks, object melting and flowing, etc. With vragments, no connectivity is necessary as each vragment can be treated as a particle and can therefore be moved individually without influencing any neighboring points or primitive groups. Figure 9(b) in the color plate shows an example of an exploding teapot mesh rendered with the vragment simulator. If necessary, new neighboring information besides the automatically defined connectivity through the object definition can be built by grouping fragments with special textures. These are simply mapped onto the object surface. Additionally, it is still possible to use connected and fast renderable primitive groupings like triangle strips or triangle fans.

4.3.4 Open Issues of Vragments

Although there are many possibilities with this proposed extension, unfortunately there are several problems and unclear issues as well. The biggest issue is probably that this extension works only well when connection information on a surface is of less importance in the rendering algorithm. Otherwise gaps between the individual fragments might be introduced especially in the case of 2D and 3D displacements. The gap or hole problem could also be solved by simply rendering more primitives but this would lead to a massive amount of overdraw. More severe – from a hardware point of view – is memory access, since being able to shift a fragment position anywhere freely in the frame buffer would lead to more complex writing patterns and it is unclear how costly these patterns might be on a real hardware. Related to this issue is that current graphics hardware pipeline infrastructure with their many parallel pixel pipelines working on a group of fragments in parallel are often arranged in a block structure to benefit from read access patterns into texture maps and corresponding filtering. It is unclear how the ability to move fragments would affect this processing order. The problem of achieving anti-aliasing is unclear as well. Nowadays this is mostly resolved by applying supersampling schemes. But in the case of vragments this might introduce additional vragments that could be moved to unspecific positions.

Another problem is the order of writing into the frame buffer because the blend/test unit supports both read and write from/to the frame buffer, if the order is changed the final result might also change as well. This problem does not show up in the simulator because the blending has to be done in software, but that might become a problem in pure hardware rendering. Also a interesting question would be the support for different point sizes. Should the vragment write module also handle different point sizes? Or should it only allow single pixels? Different point sizes might be very useful to solve some of the overdraw issues but would also increase the complexity of the unit, as now complete memory blocks might have to be accessed. The final question is how should multiple render targets be treated? Should every target have its own fragment relocation or should every target map to the same location?

4.3.5 Conclusion

Although the proposed extension introduces several challenges that are not fully addressed yet it also provides unique benefits without requiring a complete new design of the current hardware rendering pipeline. It allows new possibilities and new algorithms by slightly extending the current processing scheme of GPUs. Although some of these new proposed algorithms are possible to do with current generation of graphics processors they also suffer from certain limitation and drawbacks, or can only be implemented very inefficiently. Vragments would allow often a more efficient solutions by processing the desired effect in one singular pass instead of going through several vertex and fragment unit processing passes. Additionally the highly effective streaming nature of graphics processors would not be harmed by this extension. It would also allow GPU manufactures to look into more aggressively resource sharing between vertex and fragment processing than nowadays since they nearly would require the same possibilities this could lead to a melting of the two units in the longterm and probably also to a “real” streaming processor. This would give the application programmer more freedom, more possibilities, and less restrictions.

4.4 Transparency and Reflections in Cel Animation

In Chapter 3.4.3 two rendering styles have been classified as being suitable to increase compression ratios in an image based remote rendering environment. Both can be found in technical illustrations but the toon approach is original related – obvious because of its name – to cartoons or more general from cel animation.

The term cel is the short form of celluloid, a clear sheet of plastic where a sequence or part of a sequence of an animation was drawn onto. The transparent quality of the cel allows for each character or object in a frame to be animated on different cels, as the cel of one character can be seen underneath the cel of another. The final image is the result of stacking of individual cels above each other. This is similar to what is nowadays called layers in modern image processing.

Even though the usage of cels was an important innovation in traditional animation as it allowed to reuse parts in several frames and thus saving labor, it is still a difficult and very time-consuming media to work with. Traditional cel animation still requires a large team of highly trained artists to produce only a few minutes of animation. To reduce the required workload in cel animation plenty of research was conducted in the 90s of the last century to speed-up the process by using computer systems. At the beginning of this century many animation houses world-wide have already switched to digital animation systems. These animation systems can be classified into two main categories: 2D systems (for example Animo, RETAS! or CTP), and 3D systems (special toon-shading plug-ins [Gou05, Cam05] for commercial 3D software packages). The 2D systems are composed of a mixture between vector and bitmap editing software having automatic colorization and special build-in camera effects found in traditional animated films. Some of these systems might have been influenced by previous academic research in the case of automated image processing and composting aspects of cel animation [FBC$^+$95, Sha94, Wal81, Rob94].

2D systems are still the most used ones as the artist has the highest control over his production, and even animation experts sometimes cannot tell the difference between 2D digital made and hand-made sequences. On the other hand 3D systems are not yet that widely accepted among artists because they are known to produce a more artificial look compared to 2D systems. But increasing interest for interactive content having a toonish look & feel is pushing the demand for non-interactive and interactive 3D algorithms in this area.

When looking into academic research on 3D toon or cel shading probably the first research paper that focuses on achieving a toonish look through 3D rendering was from Decaudine [Dec96]. Lake et al. [LMHB00] later developed a hard-shading algorithm that is similar to the cel animator's process of painting an already inked cel. Their method can be applied successfully to graphics hardware and therefore allows high interactivity. A recent extension in this area describes a system for rendering stylized highlights in cartoon rendering and animation [AH03].

Similar to technical illustrations, traditional cel animation also have distinct methods to show certain global interference between objects in particular shadow casting, reflections, and refractions can still be found. In the case of cel animation they should help the viewer to emphasize the scene's mood and are important cues to increase the liveliness of the animation. Unfortunately, these inter-object relationships have been barely considered in 3D cel shading systems in the past.

The approach described in the following chapter focuses only on one important class of inter-object relationships, mainly reflecting and basic transparent surfaces. A short version of this was original published in [DE04b]. The method does not consider multiple reflection and refraction between objects because they are not very common in cel animation. This is a huge advantage as therefore, a time-consuming raytracing approach is not required, but rather graphics hardware functions can be used that even allow interactive frame rates. The methods also generally abstract from the normal local object shading and for this reason the proposed algorithms can easily be combined with any already existing 3D toon/cel shading algorithms.

The approach described should be classified as a semi-automatic method. The reasons for using a semi-automatic approach lies in the fact that in many cases the reflective and transparent surfaces in cel animation are influenced by semantic information and do not necessarily follow physical rules. This semantic information must be provided to the system from outside. In rare cases

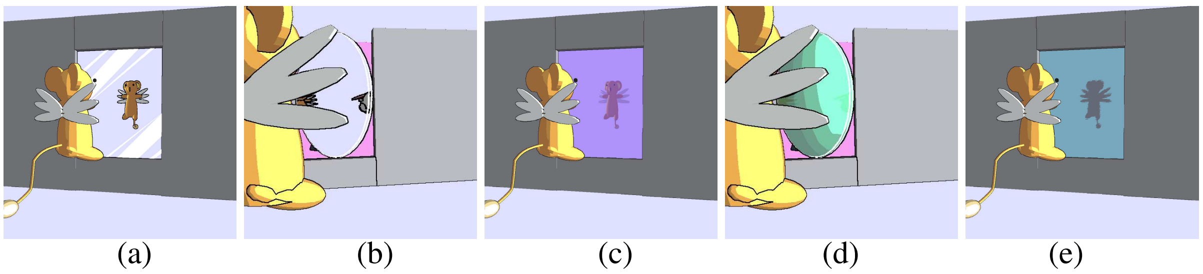


Figure 4.23: Examples of different reflection types in cel animation (a) planar perfect mirror, (b) curved perfect mirror, (c) planar partial reflective, (d) curved partial reflective, (e) water surface.

it could also be extracted automatically and some prototype functions are provided on how this could be achieved. But even these prototypes have parameters that should be given by the user to customize the rendering of the final image to his desired needs.

4.4.1 Analysis of Reflection Types

Although there are many textbooks on cartoon animation and comics, it seems there is not one that deals exactly with reflections and transparency issues. Therefore, the reflection and transparency model discussed in this chapter are driven by observations made while examining several different cel animations works. Around 30 different animation productions dating from the mid 70s to the early 2000s were investigated. The examined pieces were produced by several different commercial animation studios world-wide. Also, they ranged from TV series format to full featured animated films.

A first attribute that can be noticed is a correlation between format and budget of a production and the consideration that is spent on realistic materials including reflective and transparent ones. This might be an indication that reflective and transparent surfaces are time consuming and require a skillful artist to render these materials in a convincing manner.

It can also be observed that there are several different kinds of reflections and transparencies depending on the shape and material of the surface and each of them are expressed in a different style. Figure 4.23 shows example images of different reflection groups. On the other hand there seem to be certain characteristics that apply to nearly all of these groups. For example reflections and transparency in cartoons are seldom physically correct, but rather physically plausible to such a degree that the viewer obtains the feeling and idea of the material she/he is currently seeing. This is especially true for curved objects. Often, parts and details seen in reflecting and transparent images are removed or blended out by using a distinct color for reflected objects or objects lying behind a transparent surface (see Figure 4.24). This is done either due to expenses, to attract the focus of the viewer to important parts in the image, or to express a certain mood.

The illustration of reflecting surfaces in cartoons can basically be categorized into three different main groups. These are perfect mirrors, surfaces that are only partially reflecting and water

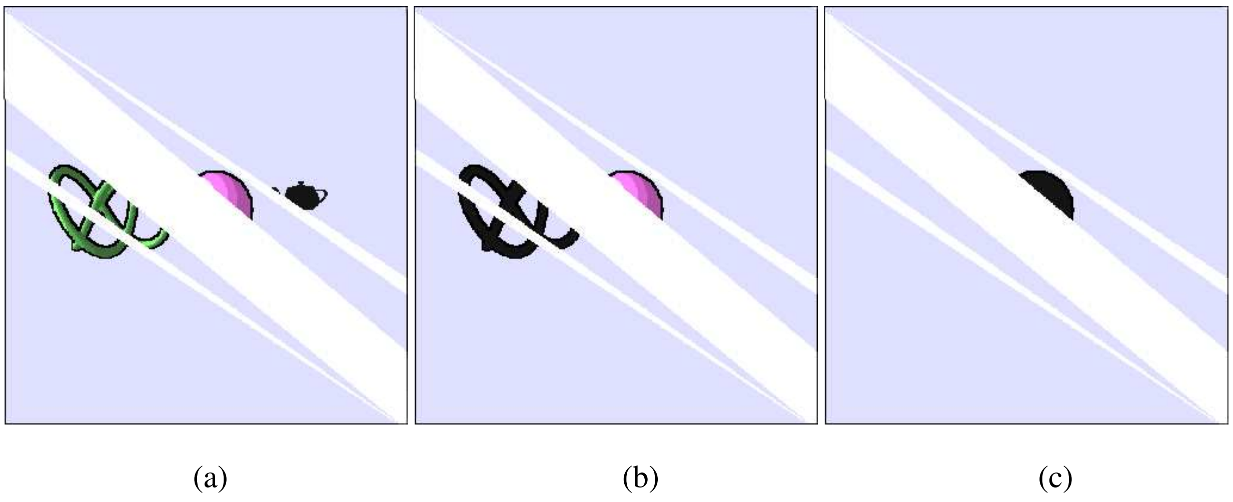


Figure 4.24: Three example pictures demonstrating how details are blended out in reflections. (a) high threshold, except for the teapot everything is fully shaded in the mirror image. (b) middle threshold, teapot disappears, knot torus is no longer fully shaded. (c) low threshold, only the sphere can still be seen but not fully shaded.

surfaces. Except for water surfaces each category can be further sub categorized into planar or curved reflectors.

Type I Perfect mirror Planar mirror surfaces (see Figure 4.23(a)) are among the most often reflective surfaces in cel animations, at least when considering the animation works that have been investigated. Like in reality, objects that are reflected in mirrors are drawn from a mirroring viewpoint. To enforce the feeling of a mirror surface an additional highlight texture is placed on top of the reflected image in the mirror, giving the reflector a shiny appearance as if light is directly bounced off the surface. Notice that the highlight texture also covers pen&ink drawn outlines of reflected objects. In the planar case, this highlight texture consists of a set of streak lines with different thickness. The reason for having this highlight texture is to disambiguate the scenery for the viewer. When using a highlight it is easier for a viewer to understand that there is not just an image placed somewhere in the scene but there is actually a reflective or transmissive object. The highlight texture is not just attached to the surface but appears to move, when either the object location or viewpoint changes.

Curved mirror surfaces – are not surprisingly – seldom found in cartoons as they require a significant amount of skill and time to reproduce convincingly. These surfaces are illustrated nearly in the same manner as planar surfaces having no major perspective distortion – as in reality – but with a different kind of highlight texture. The reflecting image often looks like it was taken with a camera using an extremely large field of view, typically greater than 120°. The highlight texture for curved reflecting surfaces in contrast to the planar case illustrates the curvature of the object. This should help the viewer's recognition of object shapes, especially when the curvature is hard to apprehend without the shading information as for example with perfect mirror surfaces.

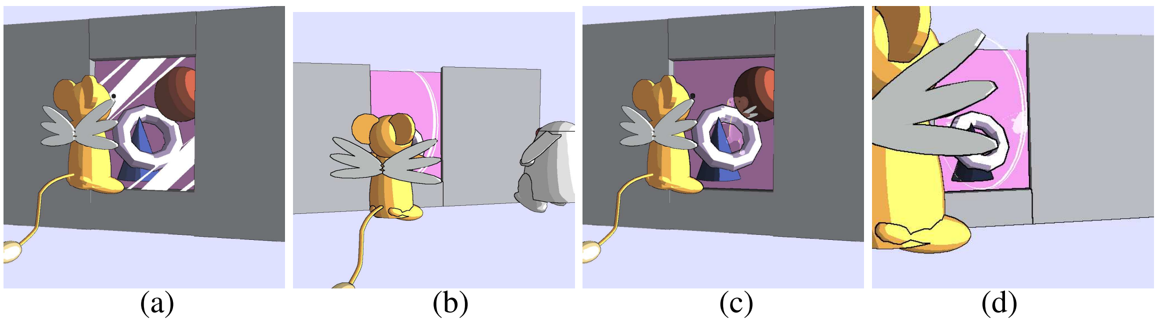


Figure 4.25: Examples of different transparency types in cel animation. (a) planar completely transparent, (b) curved completely transparent, (c) planar partial reflective and partial transmissive, (d) curved partial reflective and partial transmissive.

Surprisingly although these reflection images do not look physical plausible they do not appear to look artificial to the untrained eye.

Type II Partially reflecting objects Planar partially reflecting objects (see Figure 4.23(c)) are illustrated similarly to their perfect mirror counterparts. The differences are: instead of showing only the reflected object, the resulting color is a mixture of the surface color and the reflected color, and no highlight texture is used. The contribution of these two input colors to the final color resembles the reflection coefficient of the reflecting material. The curved version is basically the same as Type I and compared to its planar version also includes the highlight texture.

Type III Water surfaces Water surfaces (see Figure 4.23(e)) are an interesting special case. This reflecting surface type can very often be found in cel animations even in low-class productions. But surprisingly they are largely illustrated in a uniform manner, independent of the production's budget.

The refracting behavior of water is – except for some rare occasions – mostly ignored and only its reflecting part is considered. Water surfaces in cartoons also do not follow physical rules at all, but rather act like a perfect shadow mirror. Shadow mirror means that the regions of the water surface covered by a reflected object are simply darkened. For large water surfaces (like rivers, oceans, pools, ponds, etc.) the shadow reflection is slightly distorted to create a wavy looking surface. This distortion can also change over time for example when something is dropped into the water. The pen&ink outlines of reflected objects are not considered in the reflecting images of the water surface.

4.4.2 Analysis of Transparency Types

In contrast to reflective surfaces, transparent surfaces in cel animation belong to either of two major groups. These groups can further be categorized into planar and curved transparent objects.

The two main groups are completely transparent surfaces, and surfaces that are partially reflective and transparent.

Type IV Completely transparent For completely transparent surfaces (see Figure 4.25(a)) the objects lying behind them are either drawn as usual or details of the objects are blended out by painting them in a single color. Thus, only providing a feeling of the shape of the object. On top of the surface a highlight texture is placed similar to the perfect mirror type. Otherwise, the viewer might think there is no surface in between as it is unrecognizable if the surface is just a painting or actually a transmissive surface.

Curved transparent surfaces are not different to their planar counterpart, as refraction is not considered.

Type V Partially reflecting and transparent Partially reflecting and transparent surfaces are a very challenging task in cel animation, as they require a detailed understanding of the current scenery, of color mixturing, and blending between different layers. The challenge consists of providing a convincing, but yet not confusing image to the viewer. Therefore, special attention has to be taken when blending between the reflective and transmissive part. It can be observed that the blending depends on two major factors. The first factor is a semantic feature and decides which of the two portions is more important and should have a higher contribution to the final image. The second factor is the difference between the color of the reflective image and the transparent image. If this difference is getting too high it is unlikely that a blending between the two parts takes place, instead only one of the parts will be drawn. Which one is defined by the artist. In the second case the contribution of the "important" object is used and lightened or darkened by the amount of the less important ones (see Figure 4.25(c)).

Deciding which outlines to draw can even be more complicated. Either only the outlines of the more important objects are drawn or both in the blending case. In the non-blending case outlines of the less important part crossing the more important one are often not considered.

There is no significant difference between the curved version to the planar one, except that there is a highlight texture placed on top of the surface symbolizing the curvature of the object.

4.4.3 Implementation

In this chapter a class of rendering algorithms is presented to generated a as closely looking resemblance as possible for the different transparency and reflective types described in the previous chapter. The implementation described here tries to do automatically as much as possible. Only at points were semantic input is necessary, desired user input is requested or must be provided through special user-defined functions.

When dealing with transparent and especially reflective surfaces, the best solution would be to use a classical raytracer. Although recent advances on cluster-based raytracing [WSB01] improved the performance significantly, it still requires some further effort before it can be classified as being practical for interactive applications. Therefore, a classical scanline rasterization approach seems to be more suitable and has the advantage that it is possible to exploit latest graphics hardware functionality to ensure high performance. However, for several reasons it is recommended to store the current scene in a hierarchical data structure (for example a scene-graph) and attach certain attributes to each node.

Classification

Before anything can be rendered to the frame buffer, several classification steps have to be made. The initial classification should separate non-reflective objects from reflective and transparent ones. This could be done, e.g., by looking at the material properties of each object. The next step is to classify which type of reflection or transparent surface the object belongs to. This is a much harder task and it is unlikely that this can be done without intervention by the user. One solution is to use certain naming conventions for object ids or stick additional attributes to the objects inside of the modeling tool and search for these while processing the scene.

Rendering from reflectors

Except for type IV surfaces, a reflected image of the scene is required for each reflector. For planar reflectors several rendering techniques for scanline rasterization have been proposed. For an excellent survey on this topic please refer to [AMH02]. In this implementation the method proposed by Diefenbach and Badler [DB97] with some little modification is used. Diefenbach described rendering of reflective surfaces as a multi-pass rendering pipeline. In this pipeline for a reflective surface the entire scene is rendered excluding the mirror surface itself. Diefenbach proposes to draw the mirrored surface with depth-buffering and creating a stencil mask of pixels where the mirror is visible. In the second pass the environment will be rendered from the mirrored viewpoint, drawing only over the previously masked pixels. The virtual camera position is calculated from reflection of an incident line of sight with the plane on which the specular surface lies. Not only does this result in a virtual camera position, but this transformation involves scaling the scene about the Y axis to flip or mirror the image. This scaling is implicit in the transform given in [HH84] that is derived from the equation representing reflected points P_r in terms of the original point P_i and the plane equation $E \cdot P_i = 0$ having the normal N. As $E \cdot P$ provides also the distance from the plane to any point, the reflected point can be expressed by

$$P_r = P_i - 2(E \cdot P_i)\vec{N} \quad . \tag{4.8}$$

The first modification is to add an additional clip plane to make sure that objects lying behind the reflecting surface are not mistakenly drawn while the view is rendered from the mirrored viewpoint. Surprisingly, the original proposed method does not consider this case. The second modification is that instead of using the stencil masking, the scene is rendered from the mirrored view into an offscreen buffer, that will later be used as a texture. This is necessary to allow certain processing steps required for some of the reflection types. For best results the size of the offscreen buffer should roughly match the size of the onscreen buffer in order to reduce magnification artifacts introduced by texturing.

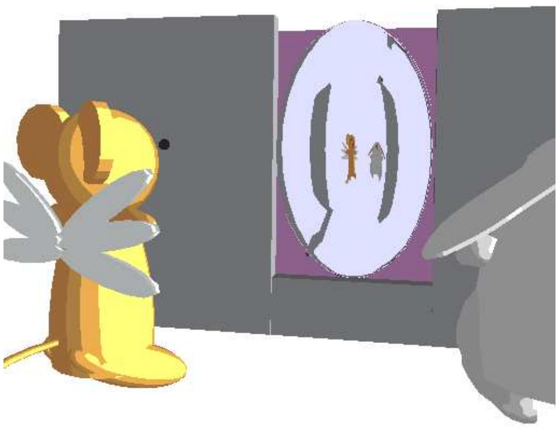


Figure 4.26: Curved reflection created by an environment cube map.

For curved surfaces the standard approach for scanline hardware is to use environment maps. Nowadays, they are mostly used in the form of cube maps since they are available on nearly every hardware. Unfortunately, for the case of toon reflections they did not seem to produce good results as the reflected image looked too distorted (see Figure 4.26). Another idea for rendering curved reflections was first described by Ofek and Rappoport [OR98]. Their idea is to generate a 3D virtual object for every reflector and every potential object that might be reflected into it. This virtual 3D object can be rendered in the usual fashion and produces an image having a visual appearance similar to the object's reflected image. If the depth relationship between all the virtual objects are still correct, the rendered image can then be merged together and by using alpha blending with the reflector image. This method has the nice advantage that the reflected image is generated on object level and not on pixel level and therefore does not underlay any resolution limit. The big disadvantage is definitely the high computing time for generating all the virtual objects and thus was not further considered. Another point for this decision was that physical perfect looking reflections are not really required or even worse, as they might be disturbing the overall impression. In conclusion a proper solution seems to be to use the same technique as with planar surfaces. But unfortunately, it is no longer so easily possible to have a definite reflection transformation as there is no definite surface normal for the plane equation. Thus, the required normal has to come from somewhere else. A good approximation seems to take the normal of one of the bounding box sides. To determine which side, a similar idea as to which 2D texture stack should be used in direct volume rendering [LL94] can be employed. Practically, it is the minimum angle between the current view vector $\vec{V}$ and all the bounding box side normals $\vec{N}$.

As multiple reflections are seldom considered in cel animation, other reflectors or transparent objects are not rendered from the mirrored viewpoint. For all other objects a user-defined function is evaluated which returns a corresponding importance value for this object before rendering. Depending on this value, objects are either rendered in the same manner as from the normal camera viewpoint (with the same lighting conditions and shading algorithm), not rendered at all, or completely rendered in a single user-defined color. Exceptions are water surfaces. Objects reflected in water surfaces are either drawn in a darkened version of the water surfaces color or not drawn at all depending on their “importance”.

Rendering the final result

Depending on the reflection or transparency type of the surface different rendering and shading paths are necessary for generating the final image.

Type I perfect mirror surfaces Perfect mirror surfaces can be rendered by simply using the multi-texturing ability of the graphics hardware. The first texture is the previously rendered reflection image and the second texture is the highlight texture. These two input textures are just added together. It is possible to use bilinear interpolation on the reflection texture if only toon shading is used for surface shading. It will smooth the edges between shading boundaries. This increases the image quality without being noticeable. It would be different if the scene contains hand-painted textures on models as well, bilinear interpolation would lead to noticeable and undesirable blurring.

The highlight texture can either be loaded as a binary image previously made by an artist, or generated using a simple algorithm or methods described in [AH03]. To ensure that the highlight texture changes when the location of the object or the viewer changes, an additional offset is added to the texture coordinates of the texture. This offset is calculated as the difference between the current view vector $\vec{V}$ and the center middle point of the bounding box of the mirror in world space P, weighted by the size of the mirror. For simplicity only the x and y component of the resulting vector are considered.

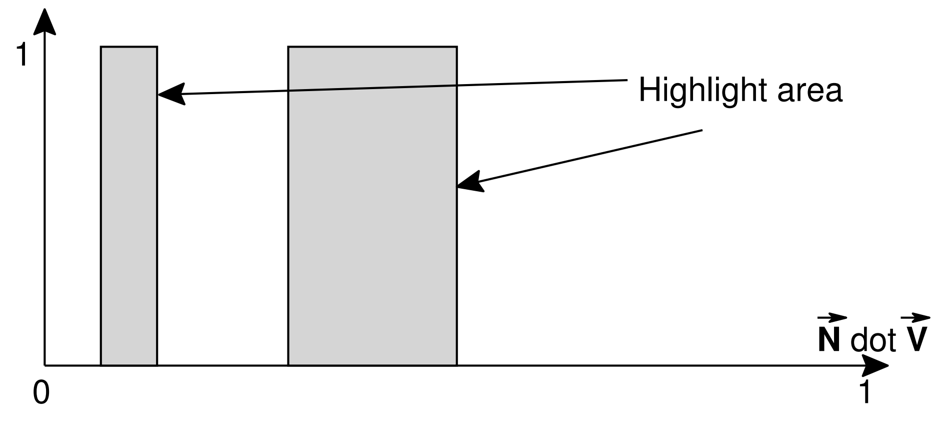


Figure 4.27: Example of a 1D highlight texture used for curved reflections.

Rendering the curved version requires only two significant modifications. The first one is a difference in computing the texture coordinates for the mirror texture. The object texture coordinates for curved mirrors cannot be used, because they would produce irritating results. Instead the texture coordinates are computed for each vertex. For each vertex a planar mapping depending on the previously chosen bounding box side of the reflector is computed. This resembles OpenGL's *glTexgen* behavior using the *GL_OBJECT_LINEAR* mode. The second modification is the generation of the highlight texture. As already mentioned before, the highlight texture for curved reflectors symbolizes the curvature of the object. At the same time, it changes when either the object location or the viewpoint is changed. This means using the local curvature values of the mesh would not satisfy both requirements because they are view-independent. Instead of using the curvature information a little trick can be applied and is often used for generating pseudo silhouette lines in object space in a single rendering pass [Eve00]. This trick makes use of the fact that the dot product between the normal vectors $\vec{N}_1$ and $\vec{N}_2$, of the two faces connected by the edge e, and the view vector $\vec{V}$ at the current edge e switches its sign if it is a silhouette edge. Alternatively, the dot product between $\vec{N}$ and $\vec{V}$ can be used as texture coordinate to look up a value in a $1D$ ramp texture marking silhouette regions. It is worth noticing that this does completely fail for non-smooth objects but in this case that is not a problem, as non-smooth objects will be classified as planar reflectors. Also the ramp texture used for showing highlights has a slightly different gradient than used for finding pseudo silhouettes (see Figure 4.27).

Type II Partially reflecting surfaces Depending on the complexity of the standard surface lighting and shading algorithm the rendering of partially reflecting surfaces can also be done by simply using multi texturing. For the final rendering, the mirror texture is weighted by a certain factor and added to the color value received from the toon shading algorithm. For the curved version the highlight texture is calculated in the same manner as for perfect mirrors.

Type III Water surfaces Water surfaces require more effort especially if the jittering of their surface should be considered. The easiest way to do this, is to add a 2D distortion on the mirror texture. This can be done by reading values from a 2D displacement texture and using these values to shift the texture coordinate for each fragment before looking up the value in the mirror texture. This 2D displacement texture can either be loaded from a previously created bitmap or generated automatically by using a noise function.

Type IV Completely transparent Before rendering completely transparent surfaces one should assure that all other objects have already been rendered to the frame and depth buffer. In a scenegraph-like structure, this can be warranted by sorting the objects by their type. Now, when rendering the transparent object just use the highlight texture as input and render this to the frame buffer with alpha blending activated and the blending function set to the maximum function. The maximum function takes the maximum color component between the source color and destination color. As the highlight texture is binary (either white or black) only the white regions will be drawn in the frame buffer and all other regions will stay untouched.

Type V Partially reflecting and transparent surfaces Unfortunately for partially reflecting and transparent surfaces, the blending is not that easy as for completely transparent surfaces, and it can no longer be done with standard blending functions provided by the graphics hardware. To still be able to render these types of surfaces the blending has to be emulated by a fragment program. There is only a slight problem: current fragment programs do not allow reading content from the frame buffer. Therefore, reading has to be emulated by rendering all the objects that are not transparent in an additional pass to an offscreen buffer. This buffer should have the same size as the frame buffer and is later used as input texture for the fragment program.

To decide which part – the mirror or the transparent – is more “important”, its importance value can be stored in the alpha channel of the buffer. This ensures that the information can be transferred into the fragment program. In the program a compare operation can decide whether it is important or not. Additionally, the intensity of each portion is calculated in the fragment program. Normally in computer graphics the intensity formula for RGB color space $I = 0.59 \times R + 0.3 \times G + 0.11 \times B$ is used but as color mixing in painting media behaves differently to RGB color space, it is better to let the user provide arbitrary weights for each color component. These weights can be set as shader constants from outside. Also the threshold for deciding when not to blend between parts but to use the more important one can be set by the user through a constant. If the difference between both colors is too large, the output color equals the color of the important part, $color_{output} = color_{important}$, otherwise

$$color_{output} = \begin{cases} color_{important} \times k_{sub} \times i_{other} & i_{other} \leq t \quad , \\ color_{important} \times k_{add} \times i_{other} & i_{other} > t \quad , \end{cases}$$

where k_{sub} and k_{add} are two user-defined factors for darkening and lightening, i_{other} is the “intensity” of the less important portion and t an additional threshold value helping to decide when to darken and when to lighten.

Possible importance functions

Importance is the term used for semantic and stylistic control by the artist in cel animations. It tells how objects that are reflected or occluded by transparent surfaces should be treated. As it is an artistic and semantic value, it is hard to get an automatical setting for it. Still in this section some indications on which properties this value could be evaluated are discussed.

The most obvious one would be to use the distance between objects as an indicator. The importance value could be derived from the distance of reflecting objects and their corresponding reflector. The same can be done with transparent objects and objects lying behind these transparent ones. Using distance has a major advantage: it just requires some basic vector math and therefore can be calculated easily and fast. However, the disadvantage relates to the fact that it is not very reliable and often does not make much sense when the camera is very close to the reflector or the reflector is very large.

Thus a better choice could be to use the actual screen size a reflected object would cover in the final image and according to this value choose the importance. This now solves the camera issue, but it still could happen that small but nearby objects would never receive a high importance, and contrary large objects might always have a high importance although this is undesired.

Alternatively, one could think of more exotic importance functions, for example very dynamic objects have a high importance as they draw attention of the viewer on them. Or highly complex objects have a small importance as they take a lot of time to draw for artists, so they rather want to save precious time.

4.4.4 Results

For the normal surface and lighting calculation the same toon-shading approach is used as in Chapter 3.4.3. The implementation was done in Direct3D9. Since Direct3D offers only a very limited support for line rendering, a screen space algorithm to create the object's outlines has been used. The approach taken is very similar to an offscreen G-Buffer [ST90], where the object ids and desired silhouette widths for each fragment are stored. A dilation filter implemented in a pixel shader program is used in a second render pass to extract the silhouettes with the desired widths. The result is afterwards blended over the previous normal shaded image.

Using a screen space algorithm has a nice advantage compared to object space feature line extraction, since it is possible to have arbitrary line widths even across a feature line. However, this advantage is paid with a little trade-off, when dealing with partly transparent and partly reflective surfaces, because it cannot be fully guaranteed that the correct outline will be extracted.

Figure 10 in the color plates shows a sequence of frames with moving objects. From the images it can be seen that the highlight texture as well as the importance of the object are changing over the frames. The objects lying behind the front shield of the car are first shaded in one single black tone, while in the later two frames they appear fully shaded. For the importance function simple distance to viewpoint calculation is used. Figure 11(b) in the color plates shows a more complex scene rendered with the prototype system, it contains two partially reflective and transparent mirrors (from the current viewpoint only one can be seen), the importance of the reflected objects is set extremely low, that is why they just appear to be shadow-like. This scene was used as one of the performance measurement scenes. Figure 11(a) shows the second test scene used for benchmarking. This scene contains five planar mirrors. And finally, Figure 12(a)-(c) in the color plates show a scene containing a curved reflector with different reflecting surface types. Notice the difference to the reflection created when using a cube environment map (see Figure 4.26).

Table 4.5 shows some performance figures for two different test scenarios. The test system was a Pentium IV 2.8 GHz PC using an ATI Radeon X800XT. The render canvas was set to 800×600 pixels. Additionally, two different settings were used for both scenes, one without rendering the lines and one with rendering the lines. The extra effort for rendering lines is surprisingly low since the current implementation nearly has to render everything twice. A little optimization would be to use multiple render targets which would allow the same number of passes with and without rendering lines. The only explanation that no dramatic frame drop can be experienced

# mirrors	# triangles	lines?	FPS
2	192192	YES	33
2	192192	NO	55
5	40829	YES	47
5	40829	NO	58

Table 4.5: Performance measurements for two different scenarios.

might be the early depth rejection of modern cards, since the depth buffer is already filled with the first render pass.

Finally, it must be said that there is still a lot of room for optimization, for example mirror surfaces are not culled, but are rendered each frame even though they might not be visible. One could think of using smaller viewports if the mirror is very far away from the current viewpoint and only takes up a small portion of the screen.

4.5 Improving the Rendering of Transparent Surfaces in Computer Graphics

Going back to the beginning of this Chapter (4.1) where a brief introduction for displaying transparent surfaces in scanline rendering was given, it is now time to take a closer look at the underlying transparency model. When comparing the visual appearance of a real transmissive surface as seen in Figure 13 in the color plates, with an artificial reproduction of the scenario using the transparency model described in [NNS73] – see Figure 13 in the color plates – one will probably notice that they do not really match too well. Certainly a sort of transparent look might be provided but the color of the object inside in the bottle does not nearly match the color appearance of the pencil in the real photograph – although both are of the same red coloring. The transparency value for the artificial rendering was set to 50%, this is definitely not the corresponding value for this material but looked like a good starting point. To achieve a more closer looking resemblance of the scenario seen in the photo, the transparency value has to be decreased to 15% (see Figure 13 in color plates). This means the standard model used in computer graphics for graphics hardware-based rendering seems to be incapable of capturing the real appearance of certain transparent materials. The main question that arises out of this is what makes a transparent surface look transparent?

The answer to this can be found in physical perception of the human vision. The perception of transparency in human vision refers to the phenomenon that an observer does not only see surfaces behind a transparent medium, but also the transparent medium itself. Interestingly the human vision system must probably be able to decompose the image into an underlying surface and a transparent surface as light reaching the vision system after passing through a transparent layer does not contain specific information about the characteristics of the transparent layers. This fact has been used by several researchers conducting research [Met74, BPI84, GSTD90, DCKL97, NSPU99, FE02] on the human visual system and lead to several models of perceptual transparency that refer exactly to this cue. Although these models differ in many respects, a broad distinction can be drawn between them and they can mainly be classified into two categories: additive and subtractive models. In computer graphics research, already in the early days, a common sense was found on how to display transparent surface and mainly the additive model survived. Although it is commonly believed that the predictions of this model cannot be too wrong in most cases, it must be said that it only resembles a rough approximation of the real underlying physical behavior of transparent surfaces. On the other hand the simplicity of this model and therefore its low complexity made it perfect for a hardware implementation. Other interesting works in the computer graphics field concerning research on transparent surfaces try to capture the transparency or refraction of real objects with so called environment mattes and use this gathered information for rendering the final 3D objects [ZWCS99, CZH$^+$00, WFZ02, PD03]. By extending the idea that the transparent surface is not actually infinite thin but rather has a certain thickness, light transport changes to mainly scattering and we enter the field of subsurface scattering, lately an area of active research. With subsurface scattering the common BRDF model is extended to a BSSRDF [HK93, JMLH01, MKB$^+$03].

4.5.1 Transparency Models

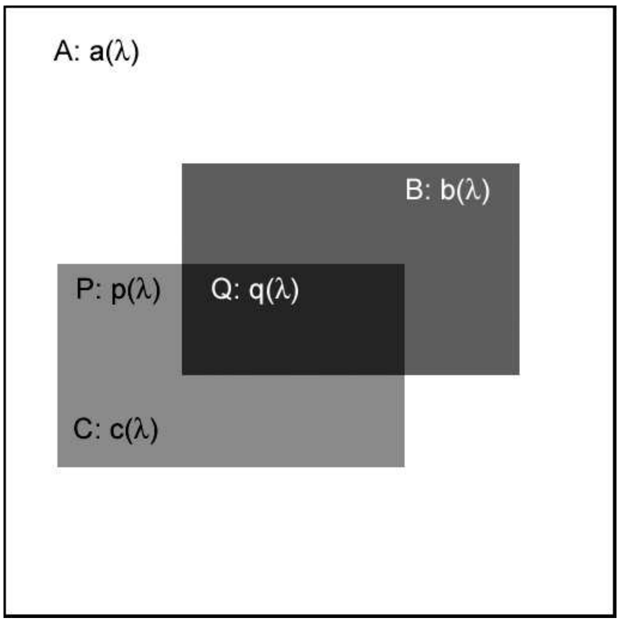


Figure 4.28: Resulting image formed by juxtaposing two squares. The left square C is assumed to be transparent. The right square B and the background surface A are opaque.

When an object is viewed through a transparent medium, spectral characteristics of the light reaching the eye are modified from the original underlying color because of color mixture. In this chapter several models of color transparency are described. Thereby, an image is considered to be formed by juxtaposing two squares as shown in Figure 4.28. It is assumed that the left square (C) is a spatially uniform transparent surface overlapping the right square (B) and a background surface (A). Regions where C overlaps A and B are referred to as P and Q, respectively. Let $a(\lambda)$ and $b(\lambda)$ be surface reflectance of A and B. $p(\lambda)$ and $q(\lambda)$ are apparent surface reflections of P and Q, that are described by the transparent models later as function of $a(\lambda)$ and $b(\lambda)$ and of characteristics of the transparent surface C.

Additive model

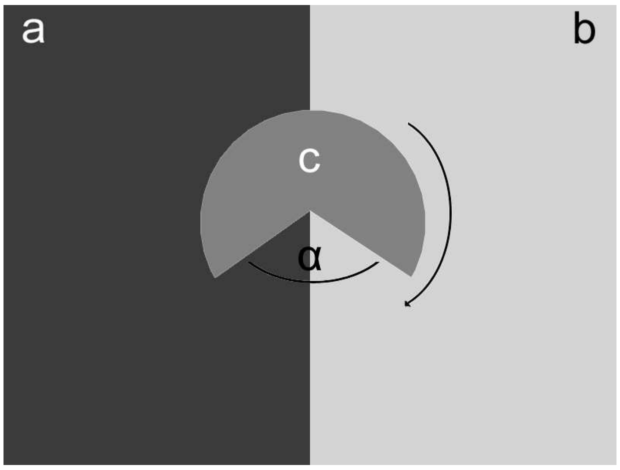


Figure 4.29: Rotating disc as a physical setup for simulating the additive transparency model as proposed by Metelli [Met74].

Metelli [Met74] first proposed a model of transparency based on a physical setup involving a rotating disc with an open sector (or wedge of a relative area α) (see Figure 4.29). When the rotation of this disc is rapid enough, it is perceived as a homogeneous partially transmissive layer. If the surface of the disc has reflectance c, and the underlying surface region has reflectance a, the resulting color mixing can be described by:

$$p = \alpha a + (1-\alpha)c \quad . \tag{4.9}$$

Metelli modeled transparency in terms of Talbot's [Tal34] equation of color mixing between transparent surface and underlying opaque surface, but Metelli used the term color for achromatic colors. Metelli noticed that when only p and a are given in an image it is impossible to determine α and c uniquely. However if there is another surface region with a distinct reflectance b, which is also partially visible through the same disc leading to a similar scenario as illustrated in Figure 4.28, then it is possible to obtain an additional equation.

$$q = \alpha b + (1-\alpha)c \quad . \tag{4.10}$$

D'Zmura et al. extended Metelli's model to the chromatic case leading to the final formulas:

$$p(\lambda) = \alpha a(\lambda) + (1-\alpha)c(\lambda) \quad , \tag{4.11}$$
$$q(\lambda) = \alpha b(\lambda) + (1-\alpha)c(\lambda) \quad , \tag{4.12}$$

where α is putatively related to the perceived "thickness" of the transparent layer. Thus α and $1-\alpha$ correspond to the proportions into which the apparent surface reflectance is divided between the underlying and overlapping surfaces. It is commonly believed that this model works well for transparent media like fog or smoke, where additive color mixture occurs.

Subtractive or filter model

Subtractive color mixture occurs when the object is viewed through absorption media such as liquid or glass. The classical filter model takes internal and external reflections into account. The parameters of the filter are the absorption spectrum $m(\lambda)$ of the filter, the filter thickness x and the refractive index $n(\lambda)$ of the filter material. To make things easier it is assumed that the refractive index is a constant function of wavelength, since it usually varies slightly in the visible range of the spectrum. If the absorption spectrum $m(\lambda)$ and the thickness x of the filter are given, the inner transmittance $\theta(\lambda)$ or the ratio of the amount of light reaching the bottom of the filter to the amount of light entering the filter can be calculated according to Bouger's law: $\theta(\lambda) = \exp^{-m(\lambda)x}$ If the material of the filter has a refractive index that differs to from that of air (≈ 1), then each time the light crosses an air-filter boundary, a certain fraction k of the incident light does not pass the boundary but is reflected. This factor k can be calculated from Fresnel's equations. Assuming unpolarized light and normal incidence, then $k = (n-1)^2/(n+1)^2$. For typical glass filters this value is 0.04.

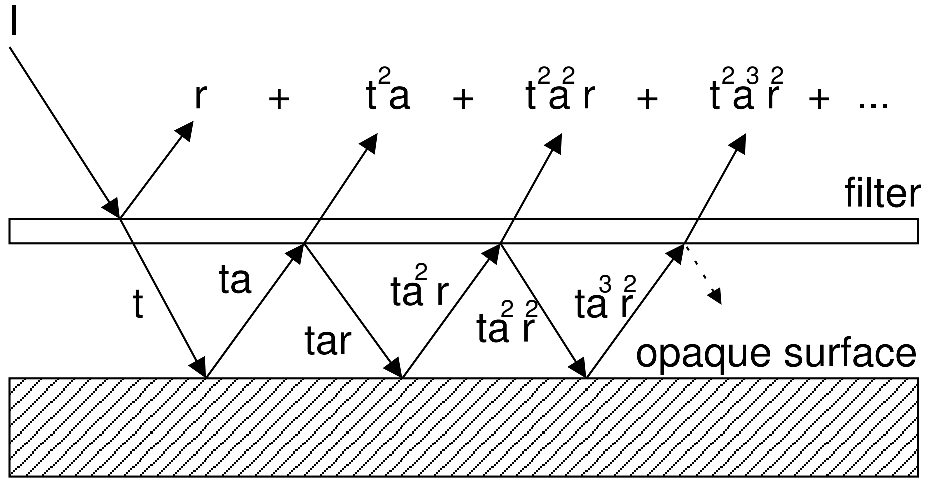


Figure 4.30: Pattern of external reflections.

Since direct reflection occurs at each air-filter interface, multiple inner reflections must be taken into account. Thus the total transmittance $t(\lambda)$, is given by the infinite sum (see Figure 4.30).

The limiting value of this sum is:

$$t(\lambda) = \frac{(1-k)^2\theta(\lambda)}{1-k^2\theta^2(\lambda)} \tag{4.13}$$

Also the total reflection $c_r(\lambda)$ that is eventually reflected from the filter surface C is given by

$$c_r(\lambda) = k + \frac{k(1-k)^2\theta^2(\lambda)}{1-k^2\theta^2(\lambda)} \quad . \tag{4.14}$$

Note that for $n = 1$, $k = 0$ the total transmittance is identical to the inner transmittance $[t(\lambda) = \theta(\lambda)]$ and the total reflection $c_r(\lambda)$ vanishes, leading to the common subtractive transparency model (see Figure 4.30). The virtual reflectance $p(\lambda)$ and $q(\lambda)$ in the regions of the background covered by the filter C thus can be determined by [BPI84]:

$$p(\lambda) = c(\lambda) + \frac{t^2(\lambda)a(\lambda)}{1-c(\lambda)a(\lambda)} \quad , \tag{4.15}$$

$$q(\lambda) = c(\lambda) + \frac{t^2(\lambda)b(\lambda)}{1-c(\lambda)b(\lambda)} \quad . \tag{4.16}$$

Scaling model

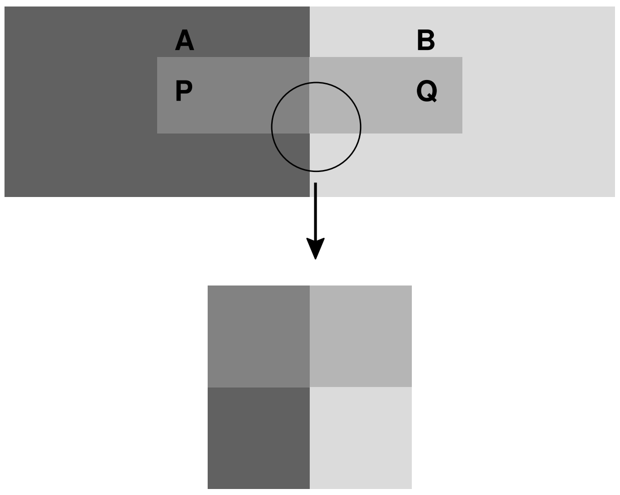


Figure 4.31: Image of an X junction, barrier between two different background colors and their corresponding colors when viewed through a transparent surface.

This rather novel model of transparency was introduced by Fraul and Ekroll [FE02]. It states that the four colors at an X junction should be optimal for an impression of transparency if following relations hold:

$$p(\lambda) = c(\lambda)\left(a(\lambda) + h\frac{(a(\lambda)+b(\lambda))}{2}\right) \quad , \tag{4.17}$$

$$q(\lambda) = c(\lambda)\left(b(\lambda) + h\frac{(a(\lambda)+b(\lambda))}{2}\right) \quad , \tag{4.18}$$

according to Fraul and Ekroll, h is a parameter related to the perceived “haziness” of the filter.

A junction is defined as an abrupt change in luminance contrast of two neighboring surfaces along their connecting edge. An X-junction is illustrated in Figure 4.31.

The scaling model may be understood as a rough approximation of the previously mentioned filter model, in the sense that the scaling describes the color change that occurs when a bipartite background is covered by a physical filter that conforms to the filter model. The authors provide a proof of this assumption in their work.

4.5.2 Proposal of a New Model

The new model proposed here is also based on the previously described filter model with some slight modifications. First the denominators of equation (4.15) and (4.16) are dropped, which are usually close to 1, thus getting $t(\lambda) = (1-k)^2\theta(\lambda)$ and $c_r(\lambda) = k(1-k)^2\theta(\lambda)$ or in the case of $n = 1$ and $k = 0$, $p(\lambda) = c(\lambda) + t^2(\lambda)a(\lambda)$ and $q(\lambda) = c(\lambda) + t^2(\lambda)b(\lambda)$. Looking at Figure 4.30 again one can notice that by using only the first order terms this approximation of the physical filter model can only be classified as a basic approximation. It can easily be improved by extending it to the higher ordered terms. Of course the question arises how many back reflections should be included to not incommensurate to the required complexity. This can be examined by making a simple example calculation. Lets assume the filter has a transmittance in the best case of 90%, the opaque object is completely white and the filter also has a white color. Using RGB color space as wavelength parameter, the third order contribution would be (for simplification we only consider the luminance contribution) $t^2(=0.9^2)a^3(=1.0^3)r^2(=0.1^2) = 0.0081$, one could argue that this value is completely negligible. The second order contribution would be $t^2(=0.9^2)a^2(=1.0^2)r(=0.1) = 0.081$ this is still $\approx 8\%$ which cannot easily be considered as negligible. Therefore, the proposed transparency model includes this second order term as well, leading to following equations:

$$p(\lambda) = c(\lambda) + t^2(\lambda)a(\lambda) + t^2(\lambda)a^2(\lambda)c(\lambda) \quad , \tag{4.19}$$

$$q(\lambda) = c(\lambda) + t^2(\lambda)b(\lambda) + t^2(\lambda)b^2(\lambda)c(\lambda) \quad . \tag{4.20}$$

Additionally, when looking at Figure 4.30 the second term $a^2(\lambda)$ does not necessarily have to have the same base color as $a(\lambda)$ due to the reflection occurring at the filter surface but the reflected ray from the lower boundary of the filter could also bounce further away that even $a^2(\lambda)$ could be $a(\lambda)b(\lambda)$, this leads to the final transparency formulas:

$$p(\lambda) = c(\lambda) + t^2(\lambda)a(\lambda) + t^2(\lambda)a(\lambda)b(\lambda)c(\lambda) \quad , \tag{4.21}$$

$$q(\lambda) = c(\lambda) + t^2(\lambda)b(\lambda) + t^2(\lambda)b(\lambda)a(\lambda)c(\lambda) \quad . \tag{4.22}$$

4.5.3 Comparison of Models

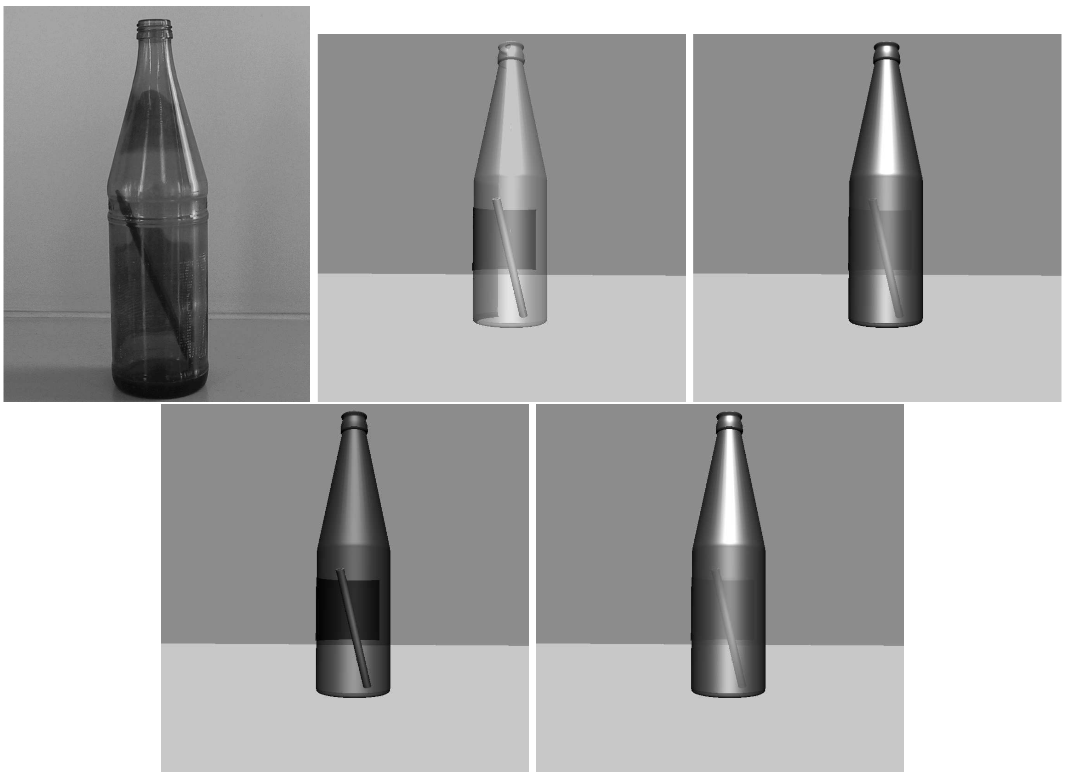


Figure 4.32: Comparison between different transparency models in the achromatic case. Upper left, real photo of a bottle and afterwards generated illuminance images. Upper middle, rendering of a bottle using the standard additive model. Upper right, rendering of a bottle using the subtractive model. Bottom left, rendering of a bottle using the scaling model. Bottom right, rendering of a bottle using the proposed novel model.

For comparison of the previously described transparency models, several different test cases were generated, consisting of three different scenarios:

- An achromatic transparent object and an achromatic opaque object lying inside.
- A chromatic transparent object and a chromatic opaque object lying inside.
- Several layers of chromatic transparent objects using different filter colors and a chromatic opaque object lying behind.

In the last scenario there are three transparent filters stacked behind each other. Its exact configuration and filter colors are shown in Figure 4.33. The resulting images of the chromatic and multi-layer case can be found in the color plates (Figures 14 & 15), whereas for the achromatic case are illustrated here (see Figure 4.32). Additionally, the real photograph was converted to a gray scale image for comparison. The α value was set to 0.5 for both single transparent scenarios and to 0.6 for the multi-transparent scenario. The h parameter used in the scaling model was set to the same value (0.5).

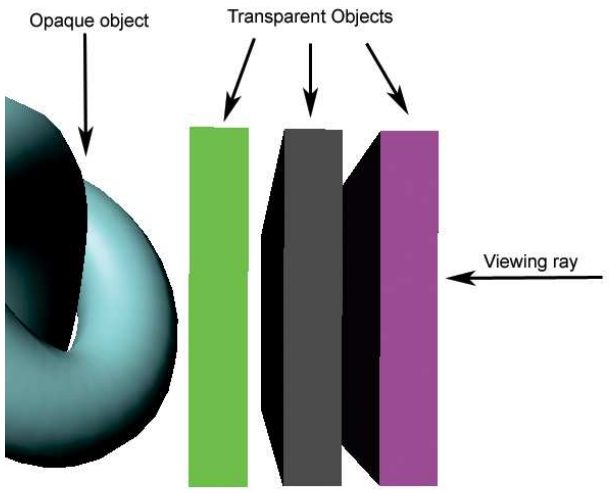


Figure 4.33: Arrangement and color of the filters for the third test scenario.

For the first case – the achromatic one – the additive transparency model found in computer graphics although it produces some sort of transparency appearance to the viewer, it does not really capture very well the visual appearance of looking through a glass surface at least not in this particular case when comparing it to the photograph. The subtractive transparency produces a different, in this case more milky, visual appearance of this scenario, so does the novel model. Actually the visual results between both appear very similar. That is not very surprising as the subtractive transparency model can be classified as an approximation of the novel model. A little bit surprising are the results received from the scaling model: there is a transparency feeling of some sort and the brightness appears to be closest to the photo but its transparency appearance can be classified as slightly odd. At least the viewer does not really get the feeling that the stick lies inside of the bottle. Analyzing the reasons why the scaling model appears like this, showed that contrary to what the authors claim, the parameter h in their model does not really produce a haziness effect, at least not with a computer graphics implementation working on RGB color space. It could be stated as even worse: nearly no evidence could be found that the parameter h has any influence on the final visual appearance of the image.

For the second case – the chromatic one – things do not appear to change dramatically. The additive transparency model is far off from the reality at least with the α value setting of 0.5. The novel model and subtractive model produce again the best visual resemblance to the photo. The scaling model could be classified as slightly better than the additive model with this setting.

An interesting case is provided by the last test case. The additive, subtractive, and the novel model achieve quite similar results when ignoring the fact that the additive version appears brighter than the other two. But the image resulting from the scaling model looked very different and much darker compared to the other models. Curious about the result given by the scaling model, further investigations have been made and it seems that for each transparency layer the brightness is significantly lower compared to all other models and this difference sums up to the final perceived image.

A second issue one might notice when comparing the results from all the test cases is that the difference between the novel model and the subtractive model are not that huge and one could jump to the conclusion that the improvement is not justified. Figure 16 and Figure 17 in the color plates show a direct comparison between the novel model and the subtractive model. It is obvious from the pictures that at high α values meaning for a high transparency both seem to converge to the nearly identical visual result. This however changes when the α value decreases. At a value of 20% there is a significant visual difference. The new blending algorithm appears more milky and less lucent than the subtractive version.

4.5.4 Implementation

The standard transform & lighting pipeline of modern graphics adapters allow different blending operations and blending factors for the destination and source color. However, there is no possible combination of blending operation and blending factors that would allow the implementation of a previously described transparency model other than the standard additive one. Therefore, the only choice one has right now is to use – if possible – programmable features of the graphics adapter. One will soon notice that none of the current graphics hardware supported rendering APIs, nor available graphic adapters allow programmable blending in the rasterization unit. But it is common nowadays that graphic chips have programmable fragment and vertex processing units. With the help of these it is possible to implement all of these transparency models by accepting several drawbacks and limitations.

The only major problem someone faces when trying to implement the novel model, the subtractive, or scaling model is to get access to the currently stored destination color in the frame buffer. Neither the different OpenGL extensions for programmable rasterization units nor the Direct3D9 specification of pixel shaders allow reading data from the frame buffer at the current fragment position, though it is possible to emulate such a frame buffer reading by using textures. The viewport mapping can either be done by using a special access register to the fragment position that for example is available in Direct3D9 in pixel shader version 3.0. Or if these are not available, by calculating them through the means of a vertex program and pass them into the pixel program via texture coordinates.

Having the viewport position of the fragment implementing the transparency models is now very straightforward. In the first pass all the opaque objects are rendered to the back buffer and to an additional render target having the same size. Notice that this requires a graphics adapter allowing multiply render targets. If this function is not available alternatively it is possible to render the opaque object as usual to the back buffer and copy the results into an additional texture surface afterwards. In the next pass one simply renders the transparent surface and uses the previous additional render target as input texture.

An arbitrary lighting calculation is used to determine the color of the transparent surface. Additional parameters like the transparency or haziness factor are either passed into the shader through shader parameters or if non-uniform parameters are desired through an additional input texture mapped onto the object with standard texture mapping. The different transparency equations from Chapter 4.5.1 are implemented by using the standard arithmetic instructions available in shader model 2 or 3. Unfortunately, since spectral data for arbitrary surface materials are currently not widely available, the implementation described here replace the wavelength parameter λ with common RGB values. Also Fraul and Ekroll do not provide further details on how the $b(\lambda)$ in equation (4.17) for the scaling model is usually determined it has been assumed to be just the neighboring texel of the background input texture. This is a questionable solution but it is the natural and most computational effective choice and leads to the desired results in most cases.

In the novel model, $b(\lambda)$ of equation (4.21) is calculated by a cheap approximation of refraction that is discussed in more detail by Kay and Greenberg [KG79] and is also known as refraction mapping. Kay and Greenberg compute a corresponding refraction ray according to Snell's Law. But they only take the x/y component to use it as offset to the current fragment position. That again is used as texture coordinate for looking up the color value in the background texture. Additionally, a thickness parameter can be used that is multiplied to the offset. This thickness parameter can either be a constant set through a parameter or arbitrary on the surface by using an additional input texture storing the thickness.

Of course even this is also only a basic approximation since the reflected ray from the background might be refracted on the transparent surface layer again before it reaches the observer's eye. It is possible to include this effect by storing the normal of the underlying object, using the flipped normal of the transparent object at the current point and calculate the real ray position. But this effect has not been included in the current implementation.

The described implementation works perfectly if only one transparent surface or many but non-overlapping transparent surfaces are contained in a scene. Otherwise there is not only the typical problem of requiring a full depth sorting or similar technique [Mam89, Die96], but when using a graphics hardware implementation as described above one also has the additional problem that basically you have to render each transparent surface or polygon in a single pass. In the worst case this is a n-times pass algorithm, where n is the number of transparent polygons. This is necessary as right now it is not possible to bind a texture to a texture sampler unit and have it assigned as render target at the same time. Although some hardwares do not explicitly prohibit this and some people have actually used this trick [Gre03], it must be said that this only works in rare cases as the outcome of the operation is undefined and its behavior can change with the next revision of the card or even with the next driver version. Therefore, this cannot really be classified as a practical solution.

Unfortunately unless upcoming hardware either allows programmable blending, reading from frame buffer, or simultaneous read and write to a texture, arbitrary multiply blending cannot be solved in a satisfactory manner. Luckily this problem is likely to be addressed with WGF (formerly known as DirectX Next).

Chapter 5

Three-dimensional Graphical User Interface for Mobile Devices

Graphical user interfaces for mobile devices set a complete new challenge compared to desktop user interfaces where it seems that a common sense has been found more than 20 years ago with the first invention of a two dimensional windowing system developed by Xerox Parc. This first idea was later extended to what is known today as the WIMP metaphor (Windows, Icons, Menues, Pointers) and desktop metaphor. Both found their way to everyday computers first on Apple Macintosh computers, later to the graphical windows system for Unix XWindows and also to Microsoft Windows. The WIMP and desktop metaphors are still seen as the state of the art of todays graphical user interfaces in most commercial software products. Mobile interfaces on the other hand have undergone a more rapid evolution than desktop interfaces, basically because some of the lessons already learned on the desktop could be transferred. However, external factors and limitations of these devices often make it impossible or sometimes not necessarily useful to transfer all the technology, methods and ideas from desktop computing to mobile computing. It is often better to build more simple interfaces or search for different interaction ideas that seem more suitable for the task.

5.1 Why Three-dimensional Graphical User Interfaces?

3D user interfaces are a very controversial subject and there are many debates in the academia – and even outside of it – over their overall usage. But in order to understand the complexity and all the diverse arguments for and against three-dimensional user interfaces, first it is necessary to separate different classes of applications. The first class that will only be briefly discussed in this chapter are applications that are designed to handle real three dimensional input through special input or natural input devices like for example in Augmented Reality or Virtual Reality environments. Examples for applications using these pure 3D interfaces are medical, architectural applications, product design, and scientific visualization [SC92, ZHR$^+$93, SZH94, DH98].

The other class of applications is your typical desktop application like word processing, web browsing, spreadsheet, etc.

The amount of controversy that a 3D user interface is necessary and useful for the first class of applications is much lower than for the second class of applications. This is not very surprising because these are typical applications where the user interacts and manipulates 3D content so it is only natural to have a 3D interface as well. Even though, surprisingly most of todays user interfaces for these 3D applications still take little advantage of the third dimension's added power. There are many reasons for this underutilization of 3D interfaces. The most obvious one is probably just a practical reason since many interaction techniques must be created from scratch, since there are only few toolkits for 3D interaction techniques. It is not easy to develop such a toolkit unless more stable and good metaphors for 3D interfaces have been found then what is known to date. 3D interfaces also complicate interface design and implementation, since the interface must take into account such issues as richer collection of primitives, attributes and rendering styles, multiple coordinate systems, viewing projections, visibility determination, lighting, and shading. But even for this sort of applications the design of interfaces are controversially discussed [Shn03]. Some designers believe that the closer an interface resembles the real world, the easier the usage. However, this is a dubious proposition since user studies showed that disorienting navigation, complex user actions, and annoying occlusions can slow performance in the real world as well as in 3D interfaces. The classic example is the ill-considered use of 3D features in which simple 2D presentations would do a better job like adding a third dimension to bar charts that might slow users and even mislead them. It is often pointed out that flexibility in a user-interface design environment is a double-edged sword, allowing novel and useful interfaces as well as novel and useless interfaces.

5.1.1 2D versus 3D on Desktop Computers

As already previously noted by the mid-1980s graphical user interfaces had converged on the desktop metaphor with overlapping windows. This desktop metaphor describes a 2 1/2 D GUI. It is 2D because visual elements are two-dimensional, i.e. they lie in a plane that is defined by 2D coordinates. They are flat and contain only planar regions. It is 2 1/2 D because when visual elements overlap, they obscure each other according to their priority. This desktop metaphor has changed little since it was created. On the other hand, the way computers are used has changed significantly over the years. A growing range of applications and online services have widened the range of applications to new real-world activities. Studies like [RvDR$^+$00] show that there are certain leads that suggest that this desktop metaphor can be improved by adding a third dimension to it. Robertson et al. show that placing documents and applications in 3D space helps users to remember where they are during later retrievals. On the other hand studies like [CM02, Coc04] actually deny this claim by saying that performance deteriorates as the freedom to locate items in the third dimension increases and that 3D interfaces can be perceived as more cluttered and less efficient than 2 1/2 D. Other problems with 3D interfaces for desktop applications are related to frequent navigation problems encountered in 3D interfaces and the need for better input techniques. It is true, that a 3D drag-and-drop operation on a window might require a lot more concentration and effort than its 2D equivalent. Also other people point out that reading, writing, and drawing cover a fair amount of our uses of computers and are almost always associated to 2D surfaces. So one can easily jump to the conclusion that 3D interfaces are inadequate for these tasks. However, when dealing with real physical objects, whether 2D or 3D, we perceive them and manipulate them in a 3D world. Why should not the same hold true for the digital world?

Besides the obvious looks cool factor of full 3D GUIs for desktop application there are also some real benefits that may be gained out of it. One of the major problems of 2 1/2D GUI is window management. Windows housing applications and documents, may overlap and obscure each other. However, people often want to switch between different tasks and applications. A consequence out of this and the physical limited screen space given by the display technology is to constantly manage the position and layout of windows. Windows need to be constantly arranged and rearranged to get access to the particular window that houses the task or information one needs at a given point in time. This phenomenon has also been described as window trashing [DAHC86]. A number of solutions to this problem have been proposed to overcome or at least reduce the amount of window trashing. These include tiled layouts, virtual desktops, and fish-eye views [Fur86]. Whereas the most popular solution that has also been integrated in many windows management systems are virtual desktops or a hardware variant multi-displays. A different solution to this problem was proposed by Robertson et al. [RvDR$^+$00]. In their system, called The Task Gallery, a redirection mechanism for hosting existing Windows application in a 3D workspace is used that does not require a recompiling of the application. By taking advantage of the powerful graphics model of Direct3D, the authors created a 3D window manager that better takes into account the human perception and spatial cognition. In Task Gallery the current task is displayed on a stage at the end of a virtual gallery. It contains opened windows for that task. Other tasks are placed on the walls, floor, and ceiling of the same gallery. This also allows the user to view multiple documents simultaneously, a task that is not that easily achieved with the other solutions and in a normal 2 1/2D desktop environment requires many steps that need to be executed to do so. Similar approaches can be found from Leach et al. [LaQG$^+$97] and Topol [Top00]. The later one discusses how a 2D windowing system can be integrated into a Virtual Reality workbench environment.

Other aspects could be that rotation of windows can play an important role in future since they could be used to better differentiate between windows. In that case, similar orientations could be used to indicate that a group of windows are related in some way (e.g. they belong to the same application or they work on the same files or other resources). Rotations of individual objects also make it possible to create interfaces for horizontal displays, that might be particularly interesting for single-display groupware situations.

A work that is very similar to the 3D Graphical User Interface toolkit that is described in this Chapter was done by Rusdorf et al. [RLWB03]. It describes a 3D toolkit acting like a 2 1/2D widget set but is built through real 3D resources. However, their work focuses more on the usage in a Virtual Reality environment.

Besides this academical research on 3D user interface for desktop applications and their pros and cons, there is also a move by the industry away from the traditional desktop metaphor. One can currently notice a modernization of the underlying graphics libraries that drive the windowing and graphical user interface systems. This demand is basically coming from the desire to have more fancy looking interfaces and also the growth of common off the shelf graphics hardware in the end 90s on desktop machines. This potential is currently sparsely addressed in standard 2D desktop applications.

Quartz Compositor

The window system of the new Apple Mac OS X operating system is based on three different libraries (illustrated in Figure 5.1). Quartz for 2D graphics, OpenGL for 3D graphics, and QuickTime for dynamic media. A fourth component, the Quartz Compositor, is responsible for the composition and display of graphics rendered with these three libraries.

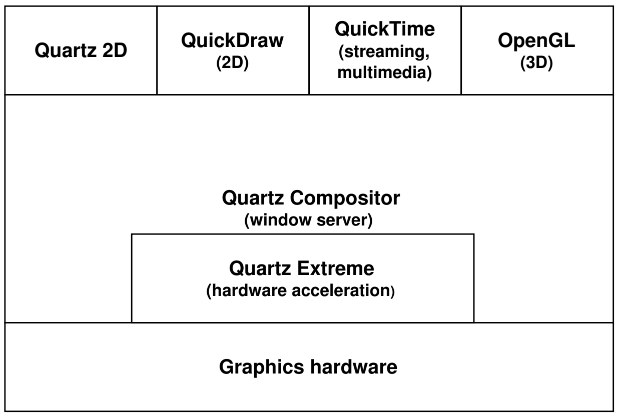

Quartz 2D
QuickDraw (2D)
QuickTime (streaming, multimedia)
OpenGL (3D)
Quartz Compositor (window server)
Quartz Extreme (hardware acceleration)
Graphics hardware

Figure 5.1: Architecture overview of MacOS X Quartz.

Quartz offers high-quality screen rendering and printing. It is based on Adobe's Portable Document Format (PDF) graphics model and features a rich number of advanced 2D graphics capabilities such as spline-based curves, text rotation, transparency, and antialiased drawing. This allows the usage of semi-transparent menus and controls, drop shadows, or fading effects that are now available to all standard Mac OS graphical user interface applications.

The Quartz Compositor is based on the idea that the window system can be considered as a digital image compositor. Quartz, OpenGL, and QuickTime graphics are rendered into off-screen buffers that are then used as textures to create the actual on-screen display. Since Mac OS X v10.2 the Quartz Compositor includes Quartz Extreme which is just another OpenGL application. As such, it can take full advantage of hardware-accelerated graphics functions to transform windows in real-time before compositing them. Examples of transformation include alpha blending, color fading, or geometric transformations, as the scale and genie effects shown when the windows are minimized.

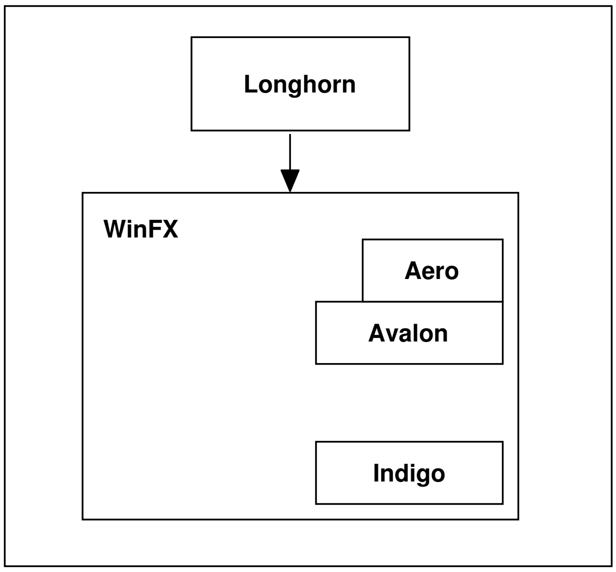


Figure 5.2: Possible architecture overview of Windows Longhorn.

Windows Longhorn

Microsofts next generation operating system called Longhorn will also introduce significant changes in the underlying windowing system similar to Quartz from Mac OS X. Although Longhorn will probably not arrive before the year 2006 some facts are already known. Figure 5.2 shows a schematic overview of Longhorn. The interesting part is WinFX. WinFX will replace Windows Forms and the Win32 API. It will distinguish between a 2D and a 3D drawing mode. The 3D drawing mode will be part of Avalon that provides a list of 3D primitives that are based on a scenegraph-like structure containing transformations, picking, intersection testing, lighting, and animations. Avalon will be considered as a high-level 3D API for GUI programming. The 3D scene itself will be captured by a camera and projected onto a viewing surface. Additionally, it will provide semi transparency and the possibility to use programmable shaders.

A user interface via Avalon can either be programmed by using a language based on XML called XAML, alternatively it will be possible to program it directly with a common programming language like C++. Avalon itself will be based on the latest Direct3D version that was previously rumored as DirectX Next but is now code named as WGF (Windows Graphics Foundation).

Java Looking Glass

Project Looking Glass is an initiative introduced by Sun Microsystems Inc. to build a 3D Window Management system based on Java. It runs both on Solaris and Linux. Project Looking Glass wants to introduce a new paradigm to the desktop called the paper paradigm but currently it is not further described what is actually meant by that. Similar to Longhorn, applications can be displayed as 3D windows, that can be rotated along both the vertical and the horizontal axis. It is also possible like in Quartz or Longhorn to use both 2D and 3D applications. 2D applications do not require a change.

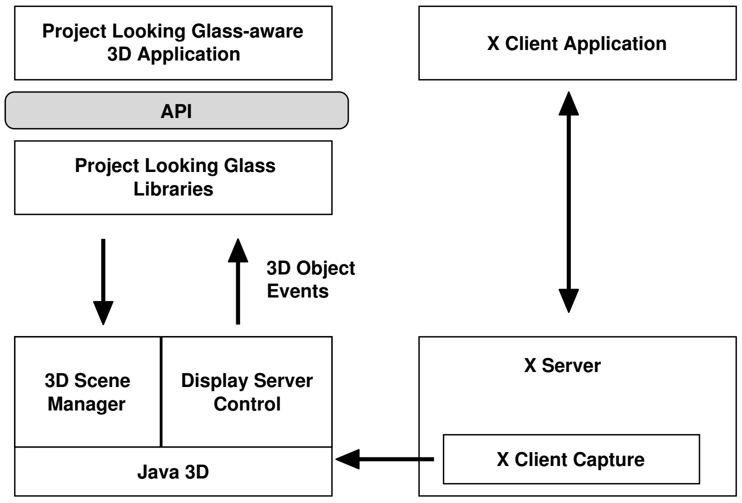


Figure 5.3: Architecture overview of Project Looking Glass.

Figure 5.3 illustrates the architecture of Looking Glass. Like Longhorn the bottom of the graphics API is defined by a 3D scenegraph. In this case it is the already existing Java3D scenegraph [Sun05]. Java3D itself uses a low-level rendering API like Direct3D or OpenGL.

Advanced features of Looking Glass will be the possibility to switch between different desktops by simply animating a rotating camera. It is also possible to use the backside of windows for useful information. For example preference settings can be accessed by turning a window onto its back.

X Windows Render and RandR

X Windows was designed with the aspect that any user-level application can act as a window manager. This led to the introduction and production of many several window managers. However, all of them are based on the original X graphics model and therefore do not differ in their operation principle. Most windows remain rectangular and opaque, very little use is made of the advanced features of modern graphics hardware.

The X Rendering extension (Render) [Pac01] and the Resize and Rotate extension (RandR) [GP01] were designed to address many of the shortcomings of the original X rendering architecture. Combined, these two extensions provide image compositing operators and glyph management and allow applications to resize, rotate, reflect, and change the refresh rate of an X screen on the fly. With XFree86 4.3.0 parts of the Render extension were included into the common Xserver, providing antialiased text drawing and image composition. On the other hand basic functions of the Render extension such as affine transformations of images remain to be implemented in existing X servers.

5.1.2 2D versus 3D on Mobile Devices

As already stated in the introduction of this chapter, mobile devices have undergone a more rapid evolution process than desktop computers because many lessons have already been learned. Surprisingly, the first mobile device was already introduced in the year 1984 from Psion that was closely related to a calculator with some extended functionality. It had only a display capable of displaying a maximum of 16 characters but a full functional keyboard. The first breakthrough was achieved in 1993 with the Psion3a that had a limited grey scale display which was already capable of displaying icons and other window controls and had a nearly full functional QWERTY keyboard. In the late 90s, driven by further miniaturization and larger display area, things started to change at least on the PDA sector. Due to space constraints the keyboard interface was nearly completely removed except a few additional important functional buttons and a stylus interface was introduced. Along with this hardware change came also a change in the software and especially in the graphical user interface. Richer iconical and graphical widget interfaces were introduced to ensure the need of less textual input. Also a way had to be found to replace the textual input which was previously done through the keyboard by other means. Since computational power on these devices was still spare and hand writing recognition was also not a completely solved issue, a more basic and easier to recognize drawing language called Graffiti was introduced. The problem with Graffiti and hand writing recognition in general is that either a mode switch is necessary to indicate the system that you want to enter text now and not to pin point on a certain area, or a special free screen area has to be reserved, minimizing the already limited screen space on these devices even further.

Due to external restrictions – where the most severe one is definitely the rather limited screen space but also lower computing power and less main memory – 2 1/2D desktop user interfaces with their rich widget sets cannot be simply transferred to mobile devices. A common solution to this problem is to restrict the complexity of the interface or even to change known paradigms like the window paradigm. Nowadays, most mobile graphical user interfaces do not work as a windowing system as it is commonly known but rather applications act like cards, where each card uses nearly the whole screen area. Therefore, it is not possible to minimize or change the size of a window but only suspend it. This makes the window managing very simple but unfortunately also actually increases the feeling of window trashing. Since to return to a previously started application now requires either to start the application from new – the operating system normally automatically detects if the same application is already running and calls back the suspended one – or through a task manager. Solutions against window trashing that work for desktop interfaces do not necessarily work that well. Especially multi screen display is out of question since it is not possible from a hardware point of view. Also the software method using virtual desktop does not really work too well with such a small display. 3D user interfaces could help to alleviate this problem. With the help of geometric and perspective transformation it is possible to use the limited screen space on these devices even more efficiently. Limited screen space is a problem that will very likely remain. Surprisingly, research on 3D graphical user interfaces for mobile devices outside the Augmented Reality realm is still non-existent.

5.2 Design of a Platform Independent 3D GUI toolkit

The 3D GUI toolkit described in this chapter should fulfill several key requirements that look reasonable enough to follow.

- It should ensure platform independence or at least easy porting between several mobile and even desktop platforms. All efforts described in Chapter 5.1.1 are bound to one or a few platforms and none of them will likely work on a mobile device.
- It should carefully use 3D techniques like occlusion, lighting, shadows, and perspective.
- It should avoid unnecessary visual clutter, distraction, contrast-shifts, and reflections.
- It should keep text readable.
- It should simplify pointer movement or reduce to none if possible.
- It should simplify object movement.
- It should enable users to construct visual groups to support spatial recall.
- It should not surprise users with completely new interaction concepts but experienced 2 1/2D desktop users should find it easy to use.
- It should offer some sort of x-ray vision to users that they are able to see beyond objects.

5.2.1 General

To accomplish the first and probably one of the most important criteria, the GUI toolkit is designed with OpenGL|ES as underlying graphics library. OpenGL|ES has the benefit that it is available or will be available soon at the time of this writing on most major mobile device operating systems, like Symbian OS (it is part of the Symbian OS 7.0 standard), Palm OS (announced to be part of the next instalment), and Windows Mobile (several software implementation OpenGL|ES libraries are already available, see Chapter 2.4). But since OpenGL|ES is a subset of OpenGL1.3 without any new commands, a porting to all major desktop operating systems is quite easy to achieve as well.

A different design criteria that addresses the third characteristic, but has practical reasons as well, is that the widgets should consist of very few textures and the appearance should be rendered by using shaded and non-shaded primitives of OpenGL|ES – basically triangles and polylines. Textures are often subject to distortion effects due to magnification and minification. Therefore, they require costly texture filtering operations and even additional mip map levels. All this would require a significant amount of memory, something that mobile devices currently do not have too much to spare.

The appearance of each widget is encapsulated in a few low and high-level primitive drawing calls and decoupled from the logic and events handling of a widget. This should allow an experienced programmer to easily change the appearance to a new look&feel if desired or requested. Also all global rendering states that are necessary for the correct appearance of the single widgets like the camera setting and the lighting parameters are encapsulated in a global scenehandler class.

And finally to satisfy the last criterion the x-ray vision, every widget supports alpha blending to ensure that widgets lying behind can be seen as well.

5.2.2 Camera Control

A three-dimensional camera poses six degrees of freedom, three for translation and another three for rotations. It is generally known that the more degrees of freedom are open to the user the more confuse and complicated gets a navigation. Another problem arises as the common stylus interface for mobile devices only allows the user to operate on two degrees of freedom at one time. This is not a problem in Augmented Reality environments where other means of interaction could be used but for normal desktop operations this is a problem. Nonetheless, it still can be useful to allow camera movements that would correspond to a virtual desktop environment in 2 or 2 1/2D. However, the disadvantage is that certain widgets would only become visible while translating the camera. If the user does not know that there are widgets outside of her/his current viewing range she/he probably will never change the camera position. Rotating the camera would mean that all the widgets have to be aligned on a circle or sphere to ensure that every widget is nearly at the same distance from the camera.

In general, a navigation in a single background color environment would be very complicated. A background texture of a familiar surrounding would help the user. For example she/he could remember that he has placed the text editor next to the door.

To overcome these navigation problems and also to fulfill the criteria of reducing the user movement if possible, the camera is set on a fixed position. This means the orientation and position cannot be changed by the user. The camera is orientated in a fashion that it always points into the positive z axis, the coordinate $(0,0,0)$ corresponds with the upper left corner and the coordinate $(width, height, 0)$ with the lower right corner, where $width$ is the width of the parent window and $height$ the height of the parent window. This camera setting corresponds to a 2D Widget set and thus should not irritate the programmer neither the user.

5.2.3 Occlusion of Windows

Common desktop window systems are 2 1/2D since all the windows have an explicit stacking order. This means windows lying in front occlude windows lying in the back. This stacking order is automatically given in a 3D environment by just using the standard depth buffer and depth test. However, when using only the depth test for determing the stacking order, it can

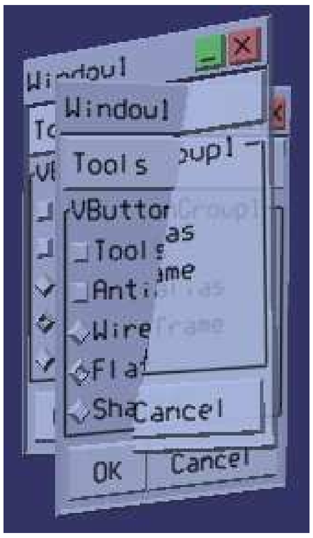


Figure 5.4: Overlapping and occlusion of windows when standard depth test is used in a 3D windowing system.

occur that windows can occlude each other (see Figure 5.4). This produces a rather awkward situation that is not desired and also breaches the previous second rule. In addition, bringing a window that is partially occluded by other windows into the foreground would require to shift the window in the negative z direction until it lies before all the others. In a 2 1/2D windowing system this is simply achieved by clicking on the window you want to set into the front. This problem is solvable by simply decreasing the z coordinate of the window that should come to the front automatically so that it lies in front and occludes all others. Unfortunately after a few of these window switches a window would probably lie behind the camera quite soon. Therefore, it would be necessary to ensure that no window will leave the viewing range of the camera. This is not only time consuming but would also lead to jumping windows while switching between windows and possibly distract the user.

To solve both problems the common stacking order of 2 1/2D is simply kept. Windows that lie at the first position on the stack occlude all the others. Other solution would be possible as well for example like [RvDR$^+$00]. In this solution the user has only one real active window and the other windows are automatically hanged into a free empty space like in a real art gallery. However, such a solution would limit the user in her/his freedom to arrange the windows to her/his desires.

The traditional 2 1/2D stacking order is rather simple to achieve via 3D since it only requires that each child window clears the depth buffer before rendering and signal its parent widget every time when it wants to be drawn last. This ensures that windows cannot overlap with each other and that after selecting a window it appears at front.

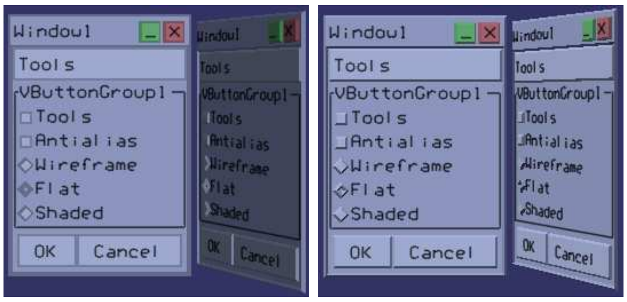


Figure 5.5: Lighting effects when using (left) two fixed directional light sources and (right) a light source that moves along with the window.

5.2.4 Lighting

If actual geometry for the 3D widgets is used instead of textures, a fundamental question is what sort of lighting should be provided to achieve always the best lighting conditions? Should point lights, spot lights, directional lights, or even more exotic light sources be used?

If a light source is set fixed in world coordinates, it is possible that the user or the system can rotate a window into a position where no light will be reflected and the window would appear dark. By using two light sources this can be prohibited when they are positioned correctly. However, at certain angles the widgets would still be illuminated very weakly and this would breach the second rule again. Figure 5.5(left) shows a screenshot with two directional light sources. What can be noticed in this figure is that there is a large contrast difference and also a loss of depth perception.

A solution to the lighting problem is not to set the light source fixed in the world coordinate system but better anchor it relative to the window. Thus, the diffuse lighting component does not change while rotating the windows. The impact of lighting effects is reduced but on the other hand it increases the visibility of widgets (see Figure 5.5(right)).

5.2.5 Antialiasing

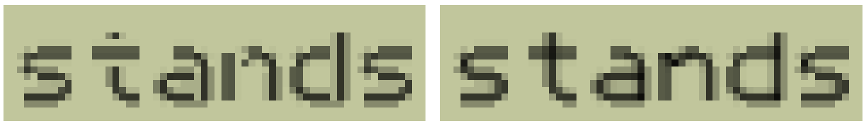


Figure 5.6: Antialiasing of lines using *GL_LINE_SMOOTH* option for font output with (left) depth protection and (right) without depth protection.

Aliasing is a rather pressing issue in 3D applications especially when rendering windows or other widgets that have small details. This is an even larger issue than in 2D, since it is also possible to rotate windows. The common solution for antialiasing is multi- or supersampling that current desktop graphics cards do on request automatically. However, with OpenGL and OpenGL|ES this can not be forced from an application side but rather has to be handled through graphics driver settings. Another solution would be to do the supersampling manually by rendering a larger version of the GUI in an offscreen buffer and later render the result as a small textured quads with texture filtering. But this increases the strain for mobile devices in double respect.

A rather basic method to achieve at least antialiasing for lines is to use the built in filtering of OpenGL *GL_LINE_SMOOTH*. To use this filtering properly in the case of 3D widgets it is necessary that line elements will be drawn last. Additional care has to be taken when line segments cross each other. In this case the depth buffer should be write protected, otherwise artifacts as can be seen in Figure 5.6(left) might occur.

5.2.6 Text Output

Rendering text with OpenGL|ES or even more general with OpenGL is a major problem since OpenGL currently has no native support for it. Still there are several methods for rendering text. The easiest and most common method is to use bitmaps. With bitmaps each character is stored as a bitmap graphics that is loaded into a texture and rendered onto the screen via a textured quad. Unfortunately, bitmaps cannot be scaled or rotated without losing significant quality and again would also require extra memory. The other possibility is to create each character via polygons and render them with standard OpenGL|ES primitives. This is also known as stroke fonts. Although the visual quality of bitmap fonts is better in the planar case still stroke fonts are the better alternatives for rotated windows. A switching between both type of fonts would be possible but might also lead to visual distraction while rotating windows.

5.2.7 Text Input

The input for text dialogs is identical to the common text input mechanism. All text input will be routed to the currently selected widget.

Text marking

For text marking with the stylus there are two principle solutions. The first one is to correspond stylus movements into a certain direction with text cursor movements to a certain direction. This solution is appropriate for 2D widgets, however it does not work as well for 3D widgets, since the text must not necessarily be aligned to the display.

Figure 5.7 illustrates the problem. A user wants to mark the text as shown in the figure. She/he probably expects that when pointing at the word “stands” and moving until “just” that the text would be marked. If the marking of text would be handled like described above this will not be the case. Only the word “stands” will be selected. To mark the text correctly it would be necessary to move the stylus further to the right and down. The problem with this kind of

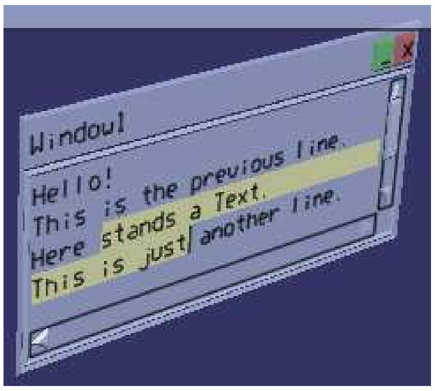


Figure 5.7: Marking a text on a rotated window.

marking is that the input pen does not provide any clue, where the marking ends if the window containing the widget is rotated. This makes it rather difficult for a user to mark the desired text passage.

For achieving the desired text marking behavior it is necessary to compute for the position of the stylus the corresponding position in the text. This can be achieved by computing the intersection between the text plane and a virtual ray with each movement of the stylus.

5.2.8 Event Handling

For event handling an approach similar to Java 1.1 event handling is used. This event handling model is based on the concept of an "event listener". An object interested in receiving events is an event listener. An object that generates events is an event source. The event source maintains a list of listeners that are interested in being notified when events occur, and provides methods that allow listeners to add and remove themselves from this list of interested objects. When the event source object generates an event (or when a user input event occurs on the event source object), the event source notifies all the listener objects that the event has occurred. This is essentially a "callback" model. When a GUI component is created, it is told what method or methods it should invoke when a particular event occurs on it. Java requires to pass special event objects between source and listener because it does not support natively the concept of pointers or more particularly method pointers. Since the toolkit described in this chapter is based on C++, this restriction does not apply. Instead the event source will be passed directly to the listener and can access changed data directly through its public methods.

The key benefit of using the event listener model is that it is very flexible and easy to understand even for inexperienced programmers. However, the events in this toolkit are split between high level events (events that are not a direct result of user input and thus do not appear so frequently) and low level events (events that are direct results of user input, e.g., keyboard and mouse events). Mouse and keyboard events are directly consumed by the widgets by implementing a special KeyboardMouseCallback handler since they appear frequently and no processing time should be wasted for fast response.

Currently, the toolkit implements three types of high-level events where an event listener scheme is used.

Action events

Action events are like in Java AWT or Swing a signal for buttons that something has changed. They correspond to a mouse down and afterwards mouse up event. The event source calls an *actionPerformed* method from the previously subscribed action event listeners.

Adjustment events

Adjustment events are especially well suited for scroll bars or slider controls to signal that the value of the slider has changed. The event source calls an *adjustmentValueChanged* method from the previously subscribed adjustment event listeners.

Component events

Component events are for all widgets telling that their current size or minimum size has changed. In contrast to the previous two event types the event source will not be attached to the event listener but rather a *ComponentEvent* object will be created and sent as parameter. This is done that not only the new sizes can be queried but also what has actually been changed. Otherwise this information needs to be stored at the event source as well.

The event source calls an *componentResized* method with a *ComponentEvent* object as parameter for the previously subscribed component event listeners.

5.2.9 Geometry Management

Each widget has a position and a size specified in 3D space. Widgets can also have children that are drawn inside of their parent widget. These children normally do not require the whole space that is provided by the parent but rather leave an empty border. The starting position of the child window is specified through $GeometryX|Y$. It specifies the distance to the left upper edge of the parent widget. The *width* and *height* specify the size of the internal drawing area of a widget. Figure 5.8 shows the geometrical dependencies of a widget.

Widgets also have a minimum size. A further reduction below this minimum size of a widget is not possible. The dependencies of the different geometrical sizes are exemplary illustrated in Figure 5.9. To react to changes in the size of their children widget or a child widget on his parent widget the previously mentioned component event and component listener is used. This is either called by the parent or child widget depending on where the change in size occurred.

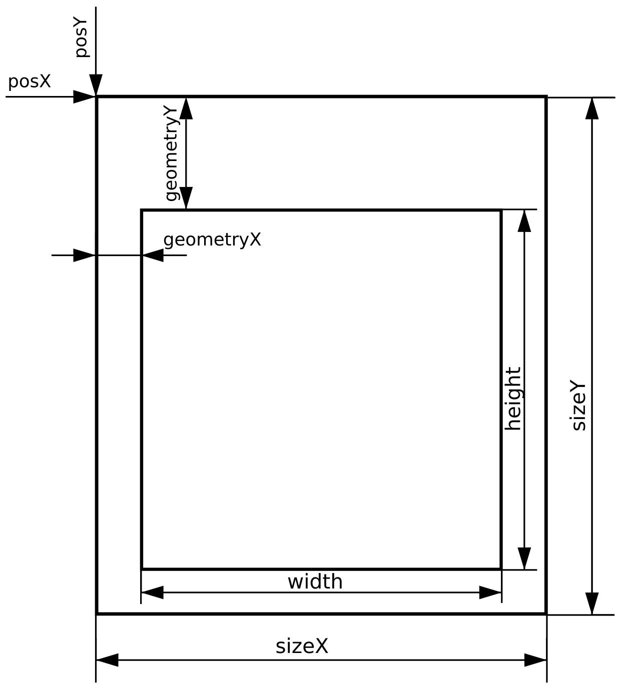


Figure 5.8: Explanation of the different size parameters in a widget.

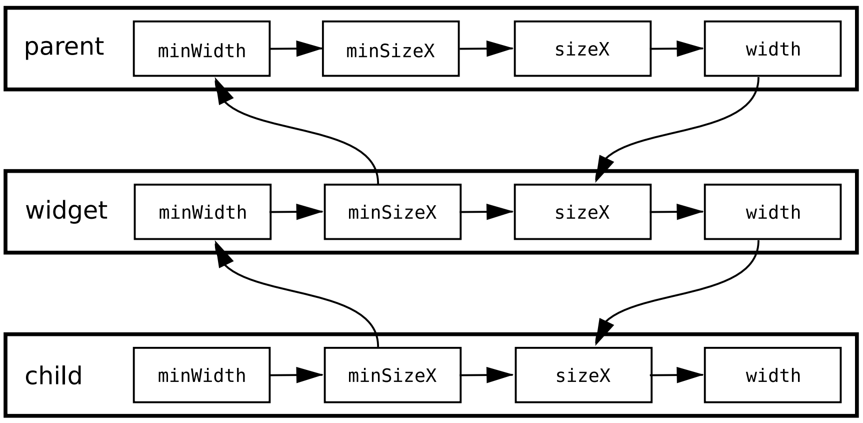


Figure 5.9: Dependencies between the different size parameters.

Figure 5.10 shows how the current and minimum size can be controlled. The thick black arrows correspond to messages that are sent through the component listener. The thin black arrows represent method calls that change the current or minimum sizes and the grey stippled arrows illustrate internal information flow in a widget itself.

The left side illustrates the case when widgets control their current and minimum size on their own. The widget receives changes in the current size of its parent widget. These are used to set the current size of itself when changes occur. It also receives changes of the minimum size of its children. These are used to change its minimum size. This scenario is rather simple, however,

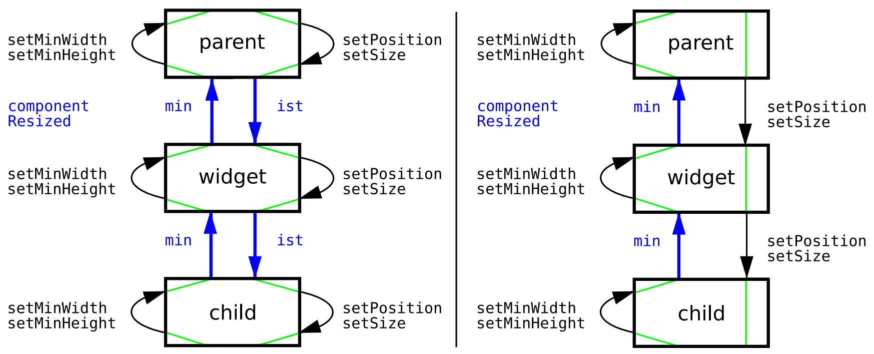


Figure 5.10: Different implementation strategies for a component listener when a resize event occurs.

Attribute	Depends on
position	parent geometry
width	sizeX
height	sizeY
sizeX	minSizeX, parent width
sizeY	minSizeY, parent height
minWidth	children minSizeX
minHeight	children minSizeY
minSizeX	minWidth
minSizeY	minHeight

Table 5.1: Policy and dependency of different size parameters between parent, current, and child widget.

it does not work when a widget has more than one child since the children do not know of each other. The right side of Figure 5.10 shows the solution for this problem. Here the parent controls the position and current size of a widget. The position and current size of the widget and the minimum size of the parent widget are changed according to Table 5.1. Since the parent sets position as well as current position of the children, it is now possible to handle more than one child correctly. The minimum sizes are determined automatically as in the left scenario.

5.3 Results

The 3D GUI toolkit has been successfully implemented on desktop PCs as well as on mobile devices of the PocketPC class. Figure 18 in the color plates shows screenshots of an example GUI generated by the toolkit on both platforms. For rendering on the PocketPC device the

OpenGL|ES software implementation from Hybrid called Gerbera was used (see also Chapter 2.4 for more details on Gerbera). The desktop version has successfully been tested on Win32 platforms as well as on Linux. The desktop version uses Producer [And05] for handling the OpenGL context, and keyboard and mouse handling. Since Producer is available for all major Unix platforms, Windows, and even MacOS a porting of the toolkit should not be any problem. The mobile device implementation uses Microsoft's GAPI [Mic05b] for screen access and the normal Win32 message loop for stylus and button handling.

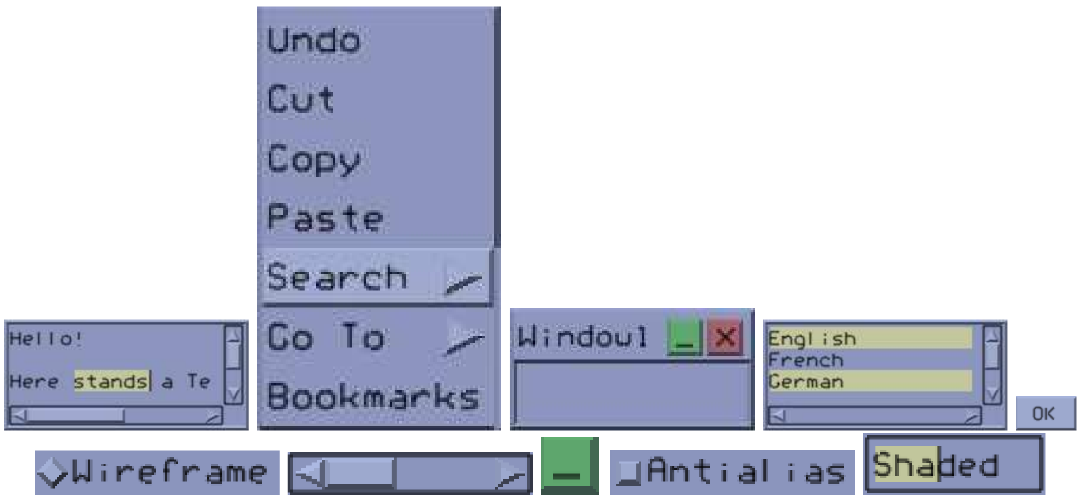


Figure 5.11: Example of different widgets that are available in this 3D GUI toolkit.

The toolkit implements the most common widgets as can be seen in Figure 5.11. But it is designed such that it can easily be extended by new widgets if necessary.

A new GUI can be built with the toolkit in two different ways. The first one is the classical approach by using C++ and invoke all the objects and methods by hand. The other possibility is to define the GUI via a XML description. This XML description is later parsed by an utility program and translated into a ready to compile C++ program. An example of such a XML description is given here:

```
<?xml version="1.0" encoding-"iso-8859-1"?>
<!DOCTYPE application SYSTEM "application.dtd">
<application text="Example program">
  <mainWindow>
    <menuBar>
      <popupMenuItem text="File">
        <popupMenu>
          <postMenuItem text="New">
          </postMenuItem>
          <postMenuItem text="Exit" variable="exitMenuItem">
            <name text="exit"/>
          </postMenuItem>
        </popupMenu>
      </popupMenuItem>
      <popupMenuItem text="Edit">
        <popupMenu>
          <postMenuItem text="Copy" variable="copyMenuItem">
            <name text="copy"/>
          </postMenuItem>
          <postMenuItem text="Paste" variable="pasteMenuItem">
            <name text="paste"/>
          </postMenuItem>
        </popupMenu>
      </popupMenuItem>
    </menuBar>
    <childWindow text="Window1">
      <textEdit text="Hello World" variable="textEdit"/>
    </childWindow>
  </mainWindow>
  <connect from="exitMenuItem" to="application"/>
  <connect from="copyMenuItem" to="textEdit"/>
  <connect from="pasteMenuItem" to="textEdit"/>
</application>
```

This generates a simple application that corresponds to the example seen in Figure 5.12.

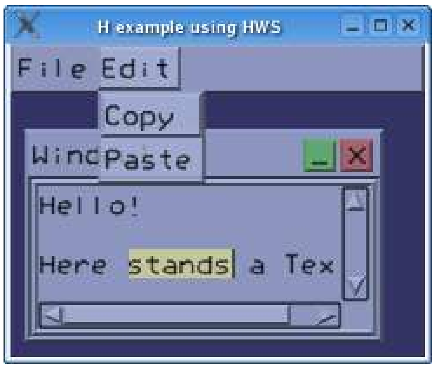


Figure 5.12: Example window that has been generated by the XML description given in this chapter.

Also the performance of both implementations – desktop and mobile device – was tested. On the mobile device the scenario shown in Figure 5.12 was used. The rendering of a frame took about 75 – 150 ms on the Toshiba e840 using Gerbera without any graphics accelerator. This still lies in interactive range and should definitely increase with graphics hardware acceleration. On the desktop a scenario with 100 windows has been tested on a Pentium IV 2.6 GHz using an ATI Radeon 9600 XT. This can be classified currently as a mid-range consumer graphics card. The performance was constantly above 25 FPS. The main bottleneck on the desktop machine is the clearing of the depth buffer, without clearing a performance boost of a factor between 4 and 5 can be achieved. But since a depth buffer clearing is only done within a *ChildWindow* widget many more other widgets can be used without such a high performance penalty.

5.4 Open Issues

There are still some unsolved issues with the current 3D GUI toolkit especially on the mobile device. From the visual standpoint the most serious one is definitely antialiasing that is more noticeable than on desktop machines with their high resolution. Unfortunately, without native hardware support there is not much that can currently be done to avoid it, except for rendering the whole GUI in a higher resolution and using texture mapping. But this would reduce the performance drastically. Unfortunately, even the *GL_LINE_SMOOTH* does currently not work properly with Hybrid's OpenGL|ES software implementation.

On the interaction part there are also some not yet satisfying solutions. For example the interaction with windows needs actually a right mouse button for rotating them – the left mouse button is used for translation. However, since PDAs have a pen interface there is only the left mouse button available. Currently, the right mouse button is emulated by pressing one of the additional hardware buttons on the device. Unfortunately, this interaction appears to be a bit clumsy.

“Good artists copy, great artists steal”

Pablo Picaso 1881-1978

Chapter 6

Summary

6.1 Conclusion

The main question of this thesis was how to bring interactive graphics and visualization solutions to mobile devices especially concerning the display of three-dimensional content. The previous chapters have successfully found different answers to this questions. Currently this goal was mainly achieved by using different strategies of splitting up the rendering tasks between mobile client and a more powerful visualization server. But that is not the only available option. For certain cases the local resources of the device might be powerful enough to complete the rendering task completely on the client device itself – as in case of the introduced 3D Graphical User Interface.

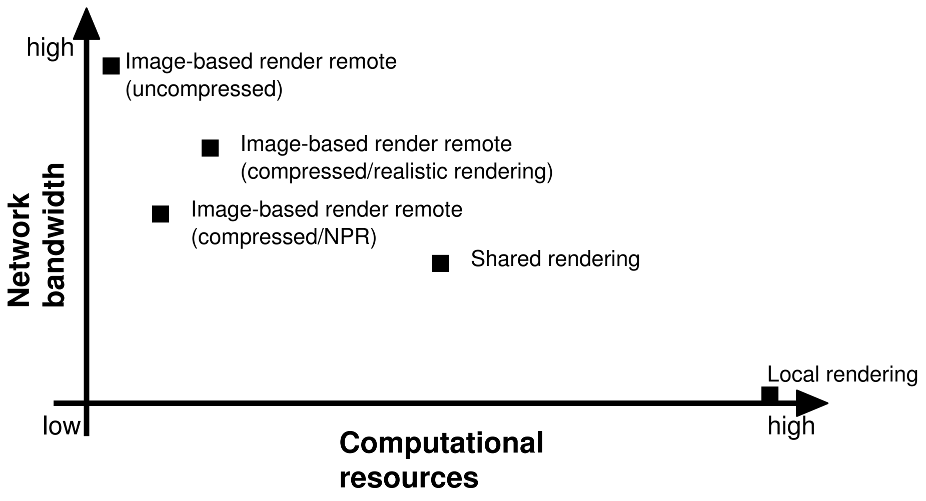
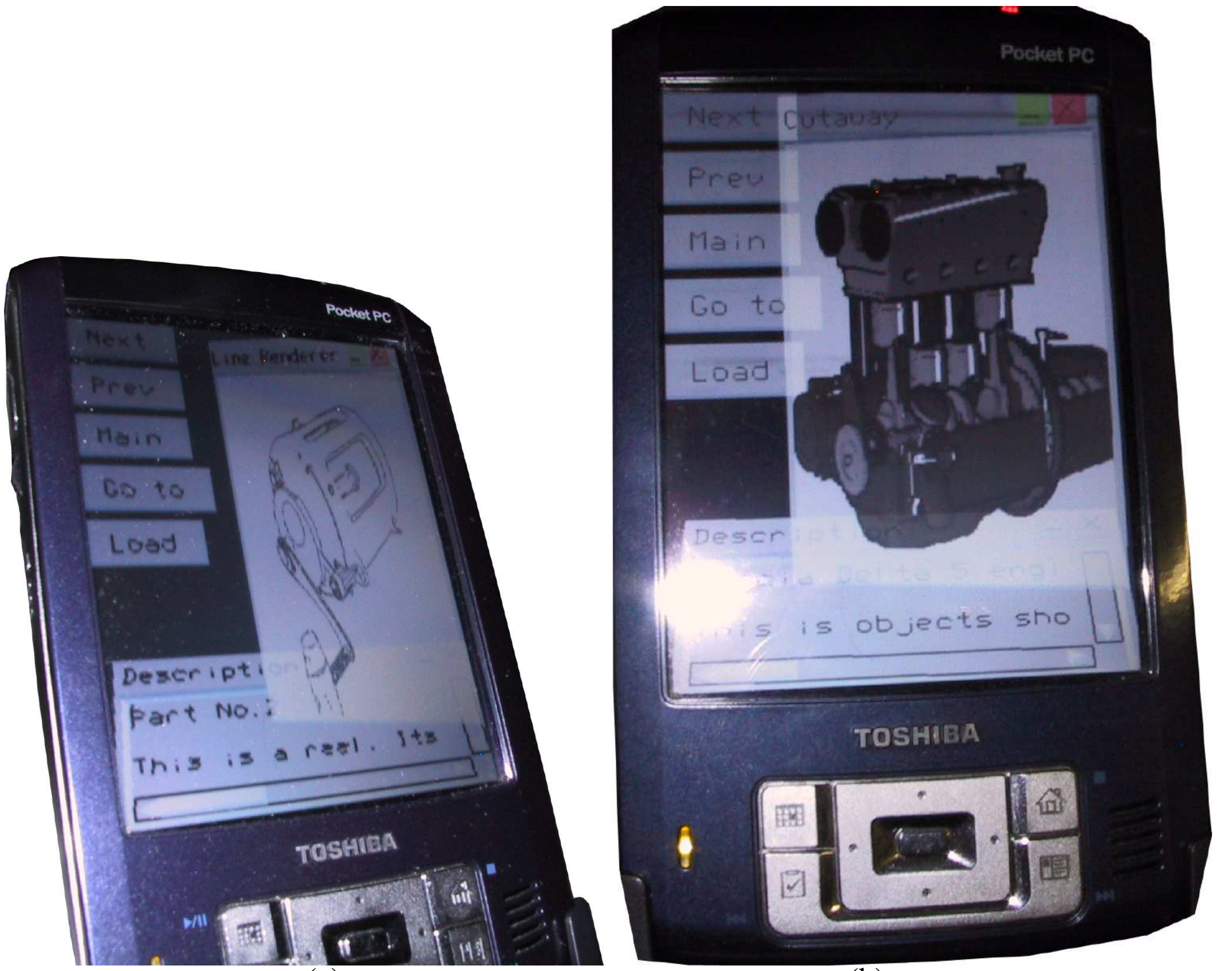

high
Image-based render remote (uncompressed)
Image-based render remote (compressed/realistic rendering)
Image-based render remote (compressed/NPR)
Shared rendering
Network bandwidth
Local rendering
low
high
Computational resources

Figure 6.1: Decision diagram for interactive rendering solutions on mobile device taking into account only the two attributes network bandwidth and computational resources on the mobile client.

Overall it is not possible to extract an universal common solution to this problem as it depends on many different factors. The most crucial ones are definitely the available network bandwidth, trade off between image quality and achievable frame rates, processing capabilities of the mobile client, storage capacity on the client, and even security reasons. What solution to take depends more on the underlying application scenario. Figure 6.1 shows the different possible options by just focusing on the two attributes network bandwidth and computation resources on the mobile device. But as already mentioned other attributes may have an impact as well and this decision graph has to be extended into more dimensions. For example, if the device has a very fast network connection but very low local processing power the image-based render remote solution is probably the solution to take. Depending on the desired frame rates and the graphical content to display either a realistic rendering or a non-realistic rendering on the server can be chosen. If the network connection is not so fast and still high frame rates are desired but the content and application can get along with a simpler form of rendering, the hybrid solution is the best choice. If there is enough local storage capacity and also enough processing power and there are no security restrictions the rendering can even be done locally on the mobile client.

But only considering these three main options is a too limited view on the problem as in some cases a combination of different methods can work out smooth as well as Figures 6.2(a) & (b) demonstrate. In this case a combination of local rendering and remote rendering are used. The 3D GUI elements are rendered locally on the device to achieve a very good usage of space and fast local response. While the complex 3D model is rendered either with the discussed hybrid remote rendering solution or the image based solution. In this particular scenario the remote solution is currently running in a window of the 3D GUI. The same would be possible by using an image-based render remote solution where the complex 3D rendering is performed inside a window of the 3D GUI. Or even a combination of all three, where parts of the scenery are rendered through image-based render remote for example a volume data set. Additional elements in this render canvas could be rendered using the render remote solution since they require fast updates and last but not least the GUI elements and other crucial interactive elements are rendered locally on the mobile client. With all these different methods introduced in this thesis the possibilities for bringing an interactive 3D visualization application to mobile devices are endlessly rich.

Rendering transparent surfaces play an important role not only in standard rendering application for desktop PCs but can also contribute as a major benefit for mobile device rendering solutions. Unfortunately, the standard solutions for dealing with transparent surfaces with current rasterization hardware underly several restrictions that are not fully solved yet. The thesis describes several alternative solutions for displaying transparency that were partially derived from artistic and illustrative style renderings. However, it is not that easy to provide a general instruction on which technique is the best candidate under certain circumstances, since as with many artistic and illustrative rendering techniques this is more an esthetic or stylistic matter. But it can be noticed that artists tend to take a more basic or sometimes also a more abstract solution the higher the complexity of a scenery is becoming. This trend can also be noticed with transparent surfaces. In technical illustrations more often a cutaway or ghosting solution can be seen, especially if the objects are rather complex or several layers of surfaces occlude an interesting part of the object. On the other hand cel animations for example seldomly contain objects with such a high complexity so a more direct approach for displaying transparent surface can be observed. So in



(a) (b)

Figure 6.2: Two example applications running on a PocketPC using a combination of different methods described in this thesis. (a) Line rendering running in a window of the 3D GUI. (b) Image-based remote rendering in a window of the 3D GUI rendering a cutaway view of the Lancia Delta 5 engine.

the end it basically depends on the purpose or application the user is aiming at.

6.2 Summary

In this thesis new methods and the current state-of-the-art how to generate 3D graphics on mobile devices for interactive graphics and visualization applications have been discussed. 3D graphics on mobile devices will play a significant role in the future for many potential applications. Either in the entertainment industry where already today a growing demand for 3D graphics can be seen and is likely to increase even further in the future. But also in other fields like location aware applications where information can be displayed and explained more thoroughly by the usage of interactive 3D graphics leading in its extreme form to an Augmented or Mixed Reality scenario. Other new rather unexplored applications can be made available for example on-the-fly and on-location visualization of information or even visualization of data that is gathered by the device itself by adding sensorial devices.

However, many of the potential applications of these devices are limited by the resources available to the device. These are mainly computation, display, storage, and power restrictions that are likely not going to change tremendously in the near future. The current state of 3D graphics and 3D graphics API on mobile devices were investigated and it could be clearly proved that the current possibilities are not enough for interactive display of scientific data or complex 3D scenarios. Therefore, in this thesis appropriate solutions to bring interactive 3D graphics already today on these devices have been presented.

Built on previous research on remote rendering solutions for desktop machines, remote rendering solutions particularly designed for mobile devices have been addressed. The main bottleneck of these solutions has been exposed and different solutions for reducing or solving this bottleneck have been discussed. Also the potential of programmable graphics hardware has been investigated that could lead to a significant performance increase currently on the server end but also in future on the mobile client device.

A large part of the thesis addressed methods that originally were intended to increase the performance of render remote rendering and compression schemes by the means of non-photorealistic and illustrative rendering through taking advantage of the graphical limitations of the device. These methods help to concentrate on the most important detail in a 3D scenario and remove unnecessary detail. Several modifications in the area of computer generated illustrative rendering were addressed to solve the problem of rendering transparent surfaces. In total three different approaches were investigated. These approaches were derived from the classic literature or handmade images found in the field of technical illustration. They do not only lead to improvements in the context of rendering content for mobile devices but more generally can be used to generate meaningful and realistic technical illustrations from 3D models with very few user interactions in highly interactive frame rates. For one of the approaches a graphics hardware extension was investigated to solve one of its still open issues. This graphics hardware extension however is also able to allow many other new possibilities besides solving this particular problem. A hybrid hardware/software emulator based on Direct3D was built to implement some prototype algorithms that become possible through this extension and also to evaluate its usefulness.

But not only methods for handling transparency in technical illustrations have been investigated, also how transparency and additional reflections are done in cel animation was analyzed. Surprisingly many different groups of reflective and transmissive surfaces could be isolated and a corresponding algorithm for automatically creating a digital reproduction was given.

With these solutions now a corresponding render remote rendering solution can take full benefit of the investigated non-photorealistic rendering styles without losing material information that previously could not be ensured. In the future, these rendering solutions can also be applied to native rendering solutions running on the mobile device itself since most of them do not require a complicated rendering architecture and even are very resource friendly.

Additionally, a novel physical plausible transparency model that can be used in scanline rendering has been introduced. This model is based on the physical filter model. It was compared with the common additive model used in hardware rendering and several other transparency models. It was possible to show that for many scenarios this model results into more realistic and convincing looking images compared to the standard additive blending model.

At the end of the thesis a novel device and platform independent 3D graphical user interface toolkit was introduced. It is based on OpenGL|ES and thus can easily be ported to all major mobile and desktop platforms. With this toolkit it is possible to design graphical user interfaces that address certain, common problems both on desktop as on mobile platforms, like window trashing and efficient usage of screen space. An implementation on a PocketPC was done to show that even right now it is possible to run a 3D graphical user interface on a mobile device without any graphical hardware acceleration. A rich set of widgets has been implemented. Also the toolkit is designed to be easily extended with new widgets if required.

6.3 Outlook

As with many fields and especially in the information technology sector it is really hard to say what is in store for the future and how things will progress. There are so many factors playing a significant role that not all of them can be taken into account and some of them are even irrational. So what will be written in this chapter is a very subjective view on things lying in the future.

What definitely can be said is that the total number of mobile devices worldwide will increase even further, all consumer statistics point into that direction. However, it is very likely that the diversity of different mobile device classes will decrease. Already today a blending between smartphone and PDA can be observed and it is possible that in the long term one of them might disappear. A more interesting outlook at least in terms of this thesis will be two major changes that can be observed. The first one is that 3D graphics hardware acceleration will come or the first generation is already there and it will be a standard feature in all mobile devices in the future. The pressure for having mobile hardware acceleration is similar as to PC desktop in the mid to end 90s of the last century coming from the entertainment industry. This will lead to reconsideration of current rendering strategies for mobile devices. On the other hand, mobile network bandwidth will increase as well. Already a step from 11 Mbit 802.11b wireless networks to 54 Mbit 802.11g/a can be experienced and there is already talk about a 108 Mbit and even higher standard. This brings network performance closely to the level of wired desktop networks right now. This in term favors remote rendering solutions especially render remote since they are easy to implement and do not require much effort on the client side. Along with the increasing bandwidth of wireless networks also the availability of these high bandwidth networks will increase. Likely the most important aspect in the future that will have a large impact on the type of solution for 3D graphics to be used will be power consumption. The installment of 3D graphics accelerators in mobile device will come with certain penalties in power consumption and it will likely make a huge difference on how many and how often features in the 3D accelerator are used to the overall battery time. On the other hand, high speed wireless network adapters will also likely require more power compared to the current ones. So it might be interesting to investigate even further in hybrid solutions where parts of the image or parts of the image generation process are done on the server and send over the network and other parts are done on the client itself. Image-based rendering and progressive mesh reduction techniques will play a more significant role in the future. Along with this will probably be aggressive texture compression since the amount of available memory on a mobile device still will grow much slower than on desktop machines. Alternative rendering pipelines like [TK96] could become interesting as well.

On the other hand, some things will likely not change much in the future. One is screen size and screen resolution: currently a move from QVGA to VGA can be experienced but it is unlikely that it will go beyond SVGA, at least not with the current display technology, unless more exotic display techniques e.g., e-paper become an interactive medium. This makes antialiasing an even more critical issue that is currently the most serious issue besides performance when considering rendering 3D content on the mobile device itself. In render remote rendering this is not a problem since it can easily be addressed by putting all the effort on the server.

Another characteristic that is very unlikely to change dramatically is the way of interacting with the mobile device: the current stylus interface will probably stay because of miniaturization reasons; a real usable QWERTY keyboard will probably not be achievable – it might be more beneficial to increase the space of the display. Maybe more speech recognition input will be used because compared to desktop machines it does not have to compete against the classical keyboard interface that limited the success of speech input. Also the working scenarios with mobile devices are often different.

An interesting issue are also 3D graphical user interfaces they are probably not stoppable even when there are many doubts in the academic field concerning their usefulness. The commercial software industry is already pushing into this area. And 3D GUIs will adapt very fast from desktop to mobile devices.

Taking a future look at the non-photorealistic rendering field, especially concerning the generation of illustrations. There are still a variety of techniques that have not been addressed yet but they are very likely to be addressed in the future since the community has grown a lot over the past years and it seems that more and more graphics research, previously focused in the photo-realistic field, is moving into perceptional and non-photorealistic rendering. Both areas should be able to learn from each other with the goal to gain more inside on how humans perceive visual representations and generate an understanding of color, material, and gestalt.

Right now some non-photorealistic rendering technique nearly perfectly mimic traditional hand crafted renderings. There is only problem that should not be left out of focus: things could become looking too perfect and therefore start to look artificial or appear too cold for the viewer. In art imperfection is often used as a stylistic medium.

Appendix A

Color plates

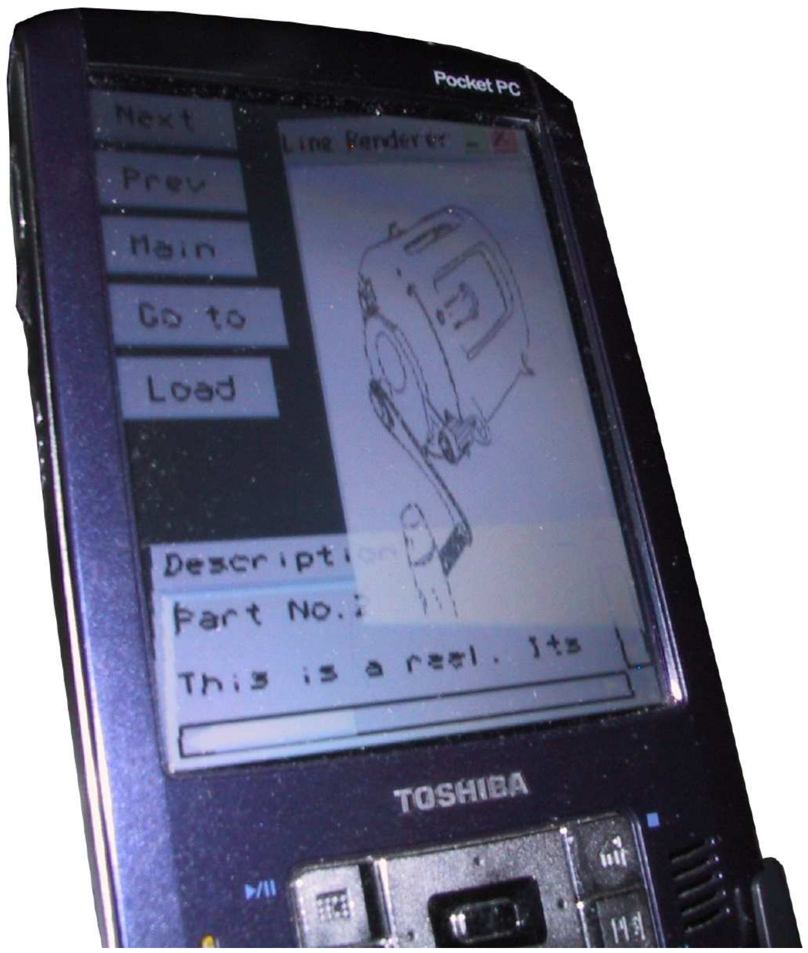
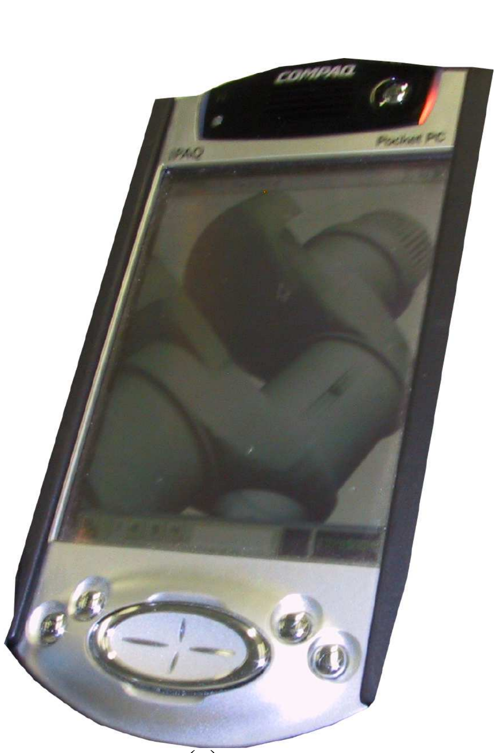
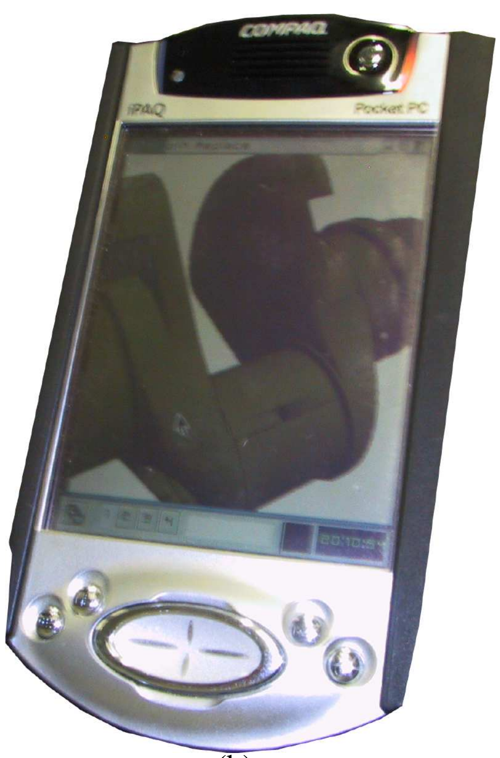


Plate 1: An example application running on a PocketPC using a combination of different methods described in this thesis.



(a)



(b)

Plate 2: Two examples of the image render remote solution running on a PocketPC mobile device. (a) using standard OpenGL lighting. (b) using cool & warm shading with feature lines.

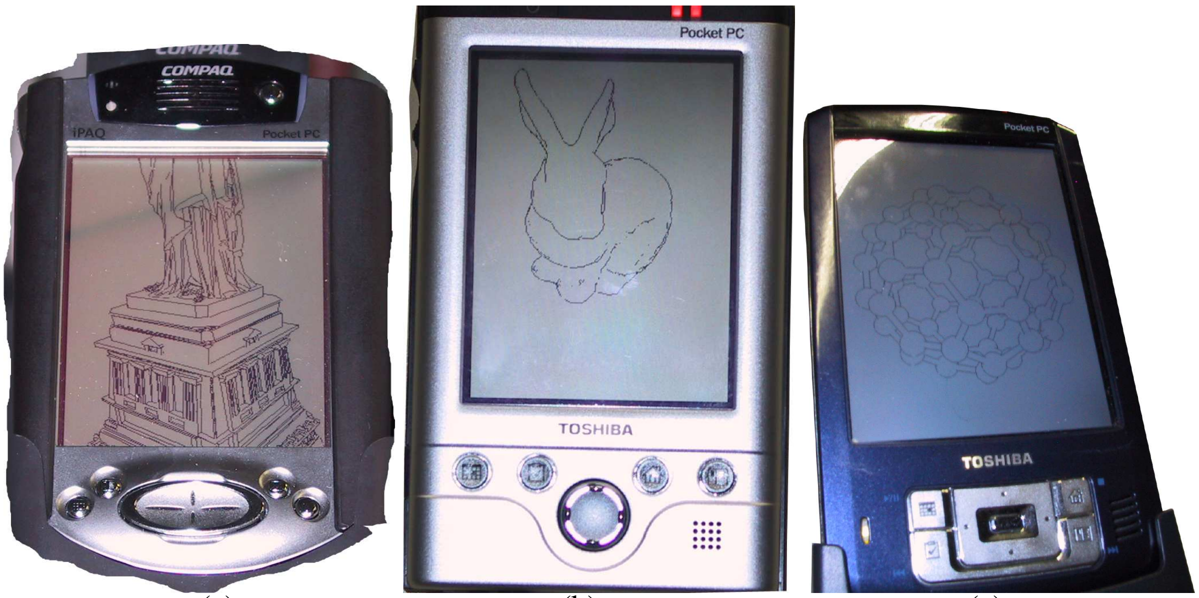

(a) (b) (c)

Plate 3: Three images showing the remote line system actually running on different mobile clients. (a) Statue of Liberty model on a iPAQ 3850. (b) Stanford Bunny on a Toshiba e740. (c) C60 (bucky ball) molecular structure on a Toshiba e840.

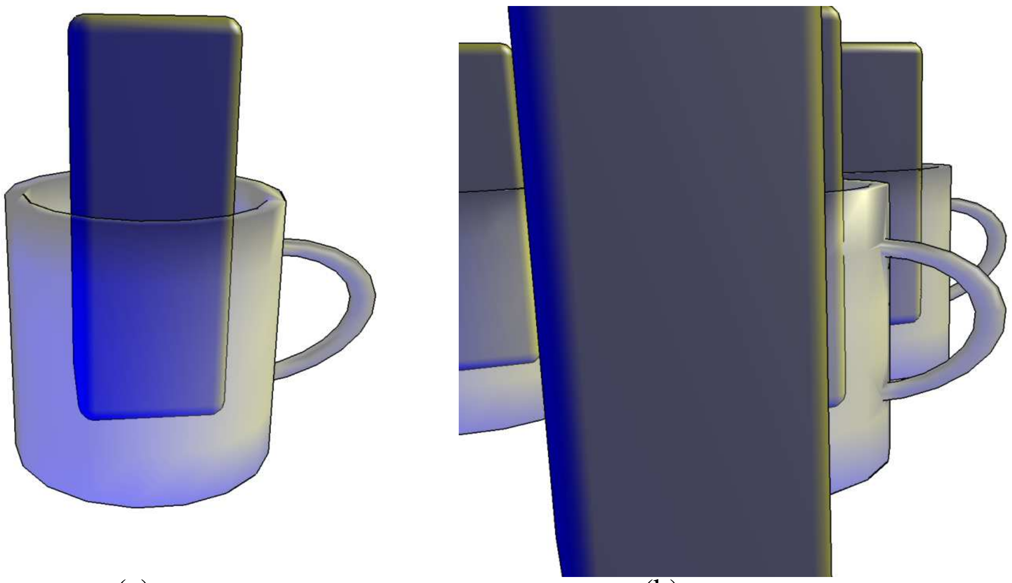
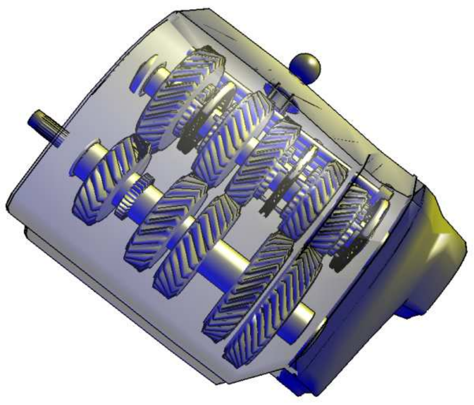
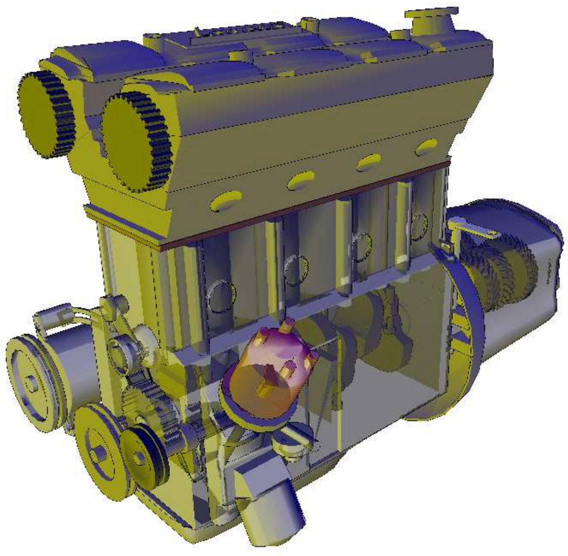

(a) (b)

Plate 4: (a) a mug rendered with the illustrative direct transparency model. (b) several mugs rendered with the illustrative direct transparency model using cool & warm shading and silhouette line rendering.

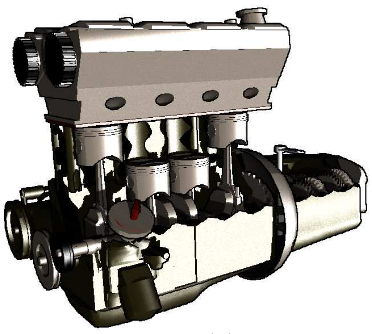


(a)

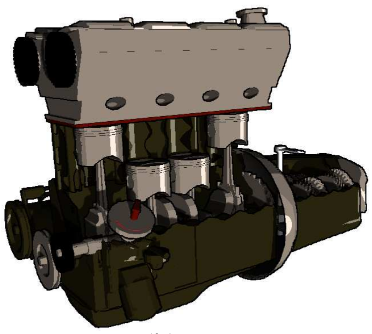


(b)

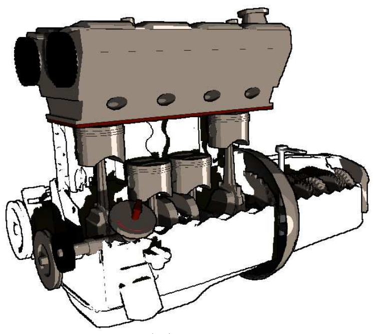
Plate 5: A part of the Lancia Delta 5 engine in more detail, rendered with the illustrative direct transparency model. (b) the complete Lancia Delta 5 engine. Both images use cool & warm shading with silhouette line rendering.



(a)

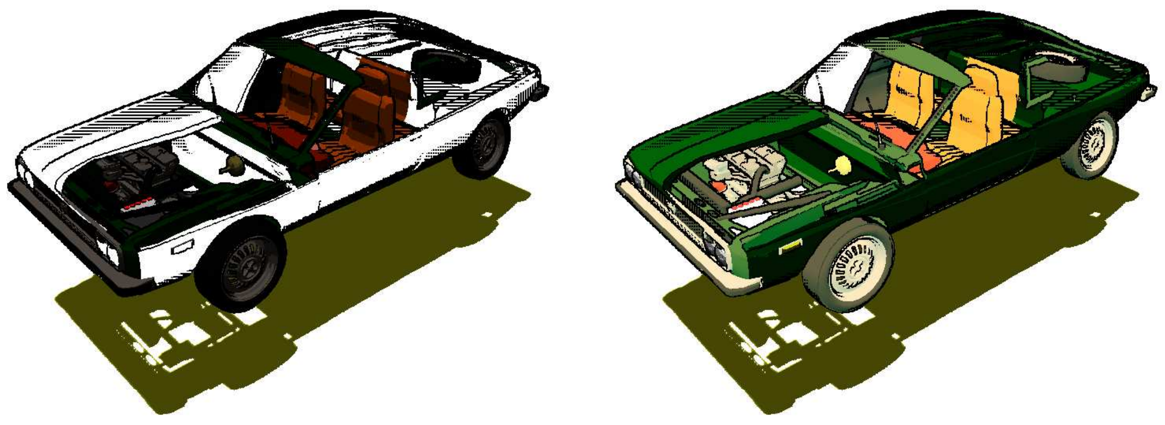


(b)



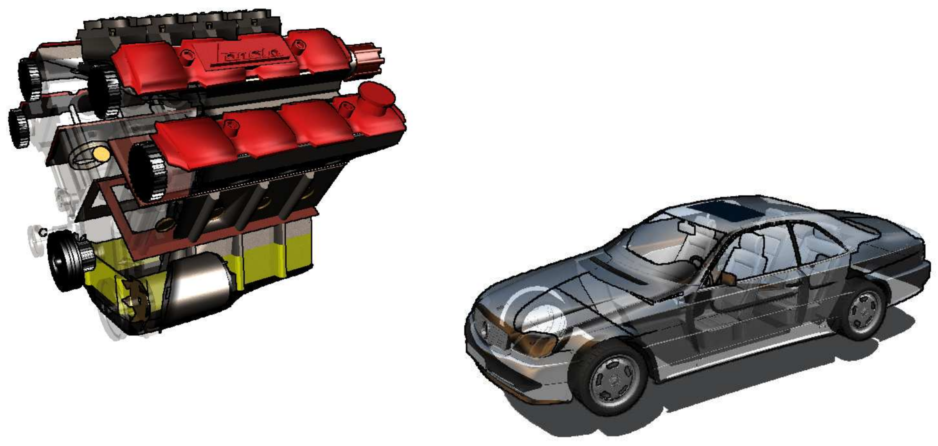
(c)

Plate 6: Three different shaded version of a cutaway rendering of the Lancia Delta 5 engine. (a) airbrush style, (b) cel shaded, (c) mixed between feature line only and toon shaded. All three images also contain shadow casting.



(a) (b)

Plate 7: Two cutaways of a Lancia Delta 5 car model. (a) exterior is rendered in toon shading, interior cool & warm shading. (b) exterior is rendered line shaded only, interior uses cel shading.



(a) (b)

Plate 8: Two ghostaway images. (a) engine block. (b) Mercedes car model.

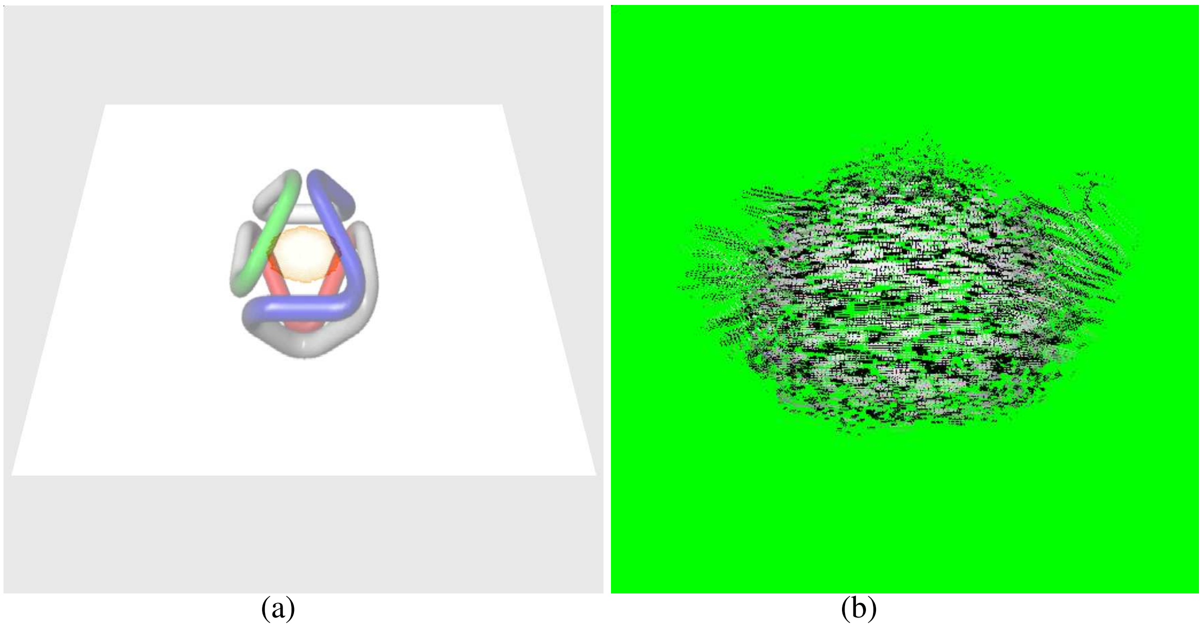


Plate 9: Two results using the vragment extension. (a) Simulation of a stain on the paper using for the 2D displacement of the pattern and the displacement of the 3D paper structure the vragment extension. (b) special effect: an exploding teapot.

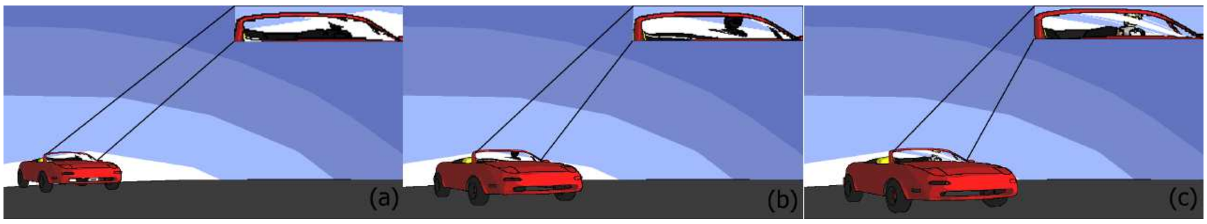


Plate 10: A sequence of a moving car with a transparent front shield. Using the transparency and reflection model for toon shading. In the first frame the car is very far away from the camera. Everything appears to be dark. Notice the moving highlight between the frames. In the third frame the car reached a certain distance to the viewer where some parts of the model that are lying behind the front shield are shaded in full detail.

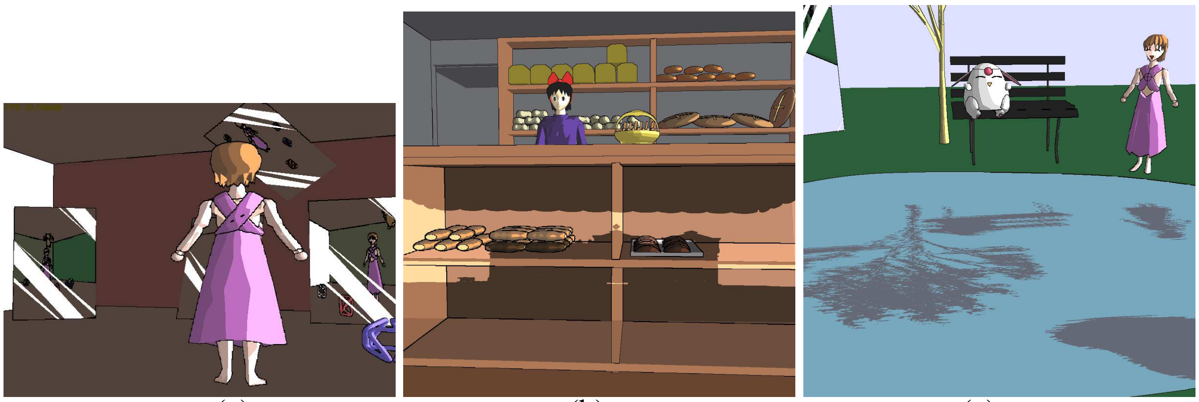


Plate 11: Three different, more complex scenes showing different reflective and transparent surfaces in toon shading. (a) & (b) are the two benchmark scenes used. (c) shows a water surface and a reflector in the back. (a) contains five planar reflectors and (b) contains two patial reflective and transparent surfaces.

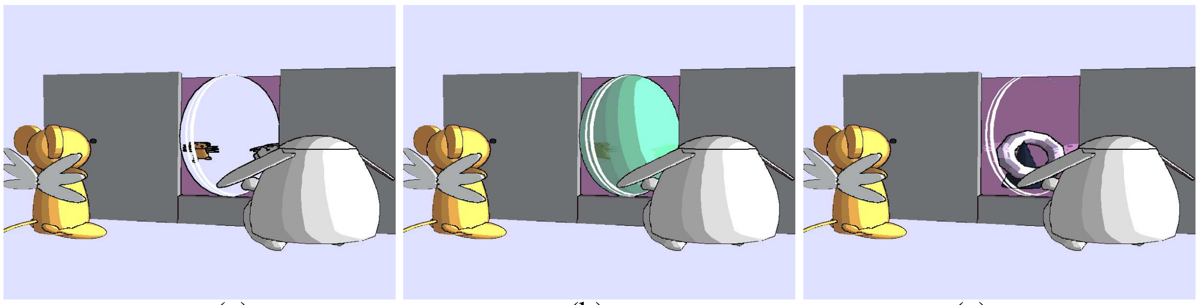


Plate 12: Different curved reflective and transparent objects. (a) perfect curved mirror, (b) partial curved mirror, (c) partial reflective and transparent surface.

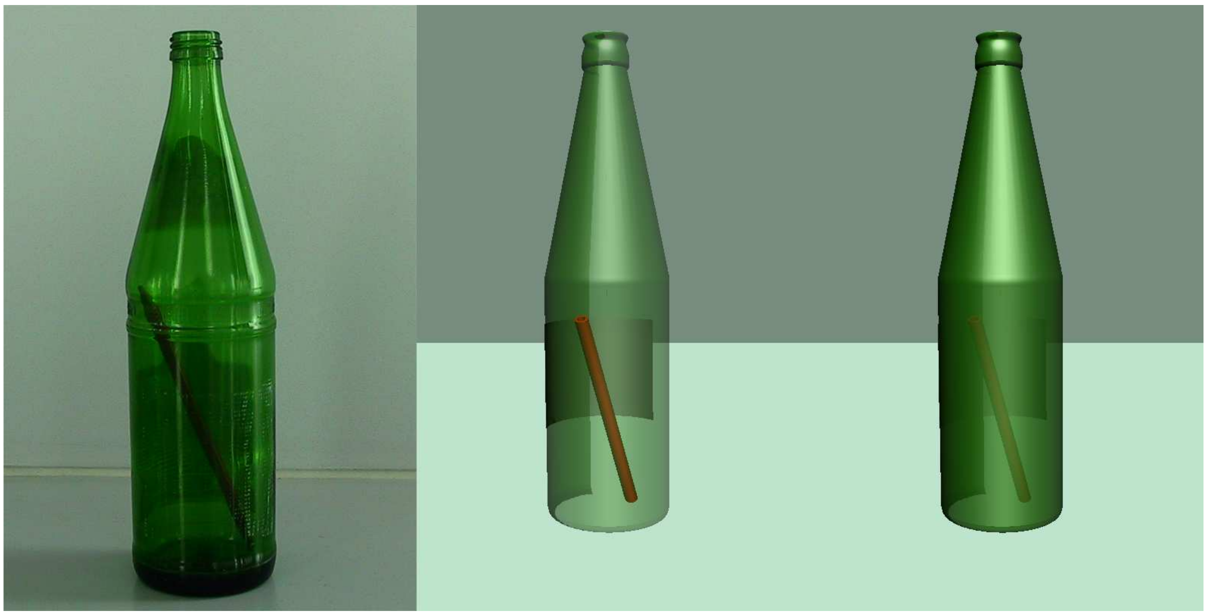


Plate 13: A real transparent bottle with a red pencil inside compared to an artificial rendered version using the standard transparency model and another version with a lower transparency value.

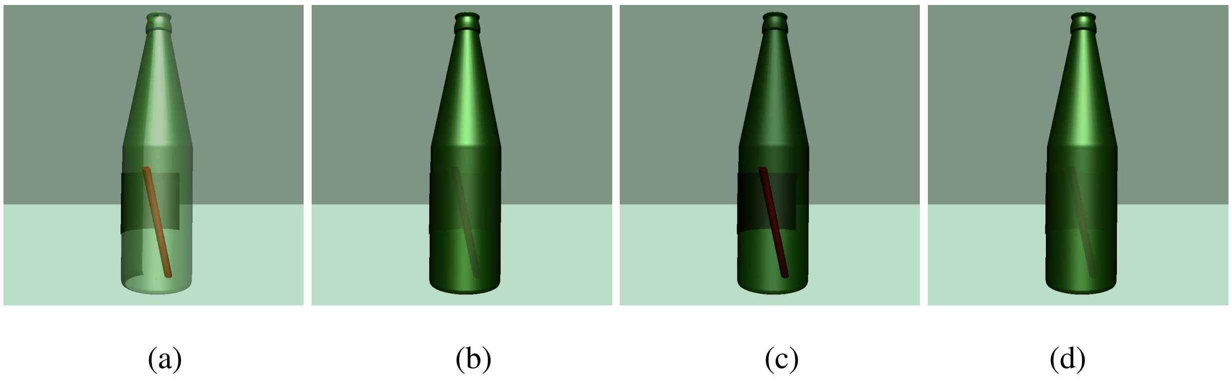
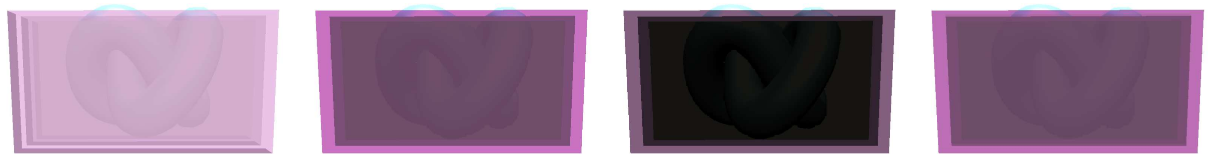
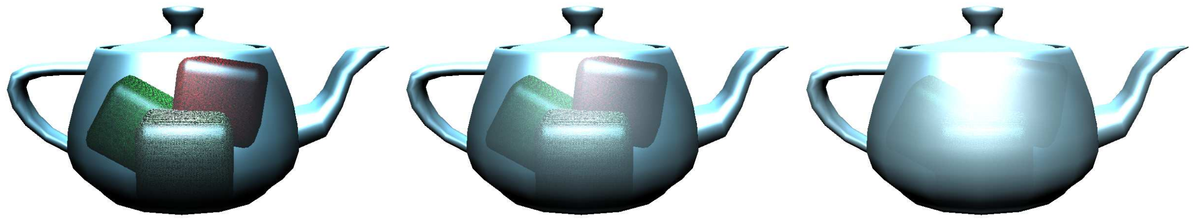


Plate 14: Comparision between the different transparency models for the chromatic test case. (a) additive model, (b) subtractive model, (c) scaling model, (d) proposed model.



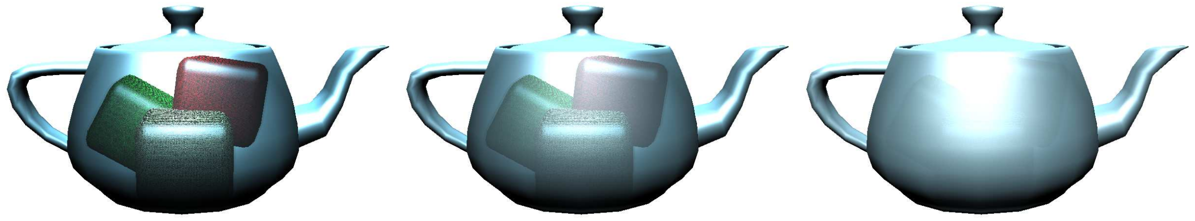
(a) (b) (c) (d)

Plate 15: Comparision between the different transparency models for the multi layer transparent test case. (a) additive model, (b) subtractive model, (c) scaling model, (d) proposed model.



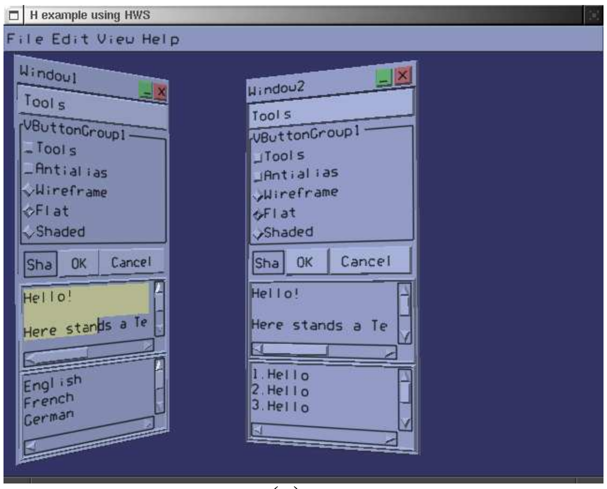
(a) (b) (c)

Plate 16: Rendering of a scene using the subtractive model with different transparency values. (a) 80%, (b) 50%, (c) 20%.

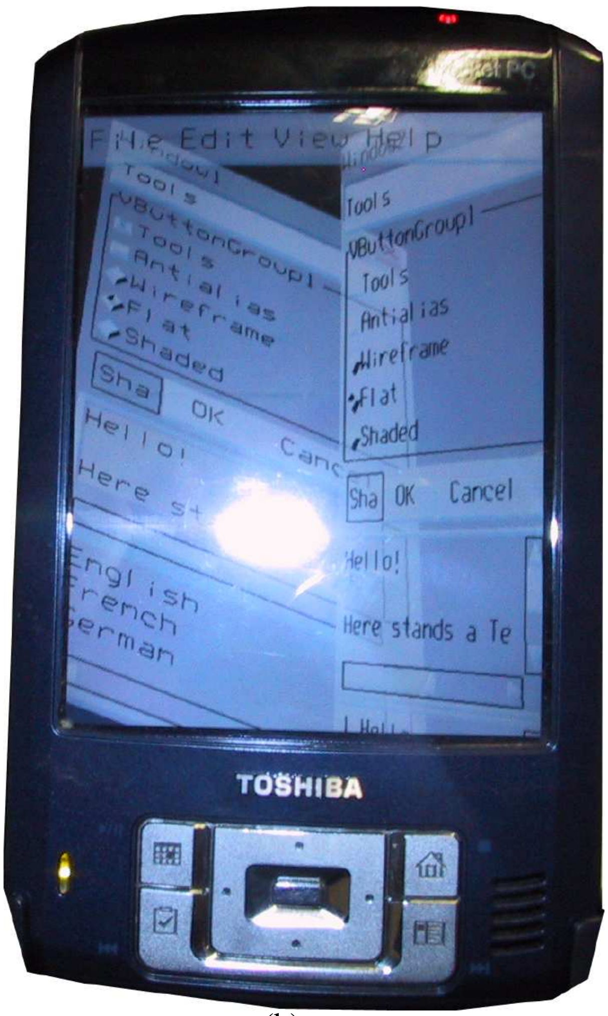


(a) (b) (c)

Plate 17: Rendering of a scene using the subtractive model with different transparency values. (a) 80%, (b) 50%, (c) 20%.



(a)



(b)

Plate 18: A demonstration GUI using the 3D GUI widget set introduced in this thesis on two different devices and platforms. (a) PC using Linux and OpenGL, (b) Toshiba e840 PocketPC using Windows Mobile 2003 and OpenGL|ES.

Appendix B

Bibliography

[AH03] K. Aniyo and K. Hiramitsu. Stylized highlights for cartoon rendering and animation. *Computer Graphics & Aplications*, 23(4):54–61, 2003.

[AMH02] T. Akenine-Möller and E. Haines. *Real-Time Rendering 2nd Edition*. A. K. Peters, 2002.

[AMS03] T. Akenine-Möller and J. Ström. Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Trans. Graph.*, 22(3):801–808, 2003.

[And05] Andes Computer Engineering. Open Producer. Web site: http://www.andesengineering.com/Producer, 2005.

[APH$^+$03] M. Agrawala, D. Phan, J. Heiser, J. Haymaker, J. Klingner, P. Hanrahan, and B. Tversky. Designing effective step-by-step assembly instructions. *ACM Trans. Graph.*, 22(3):828–837, 2003.

[App67] A. Appel. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of ACM National Conference*, pages 387–393, 167.

[BAC96] A. C. Beers, M. Agrawala, and N. Chaddha. Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 373–378. ACM Press, 1996.

[BDH93] B. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hull. Technical Report GCG53, The Geometry Center MN, 1993.

[BPI84] J. Beck, K. Prazdny, and R. Ivry. The perception of transparency with achromatic colors. *Perception & Psychophysics*, 35(7):407–422, 1984.

[Bre65] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):35–40, 1965.

[BS00] J. W. Buchanan and M. C. Sousa. The edge buffer: a data structure for easy silhouette rendering. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 39–42. ACM Press, 2000.

[Cam05] Cambridge Animation Systems. Inkworks render, cel shading plugin for Alias/Wavefront Maya, 2005.

[CDF$^+$86] G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, and .L. A. Leske. Two bit/pixel full color encoding. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 215–223. ACM Press, 1986.

[CFK$^+$96] D. Converse, J. Fulton, C. Kantarjiev, D. Lemke, R. Mor, K. Packard, R. Tice, and D. Tonogai. LBX - Low Bandwidth X Extension, 1996.

[CHH02] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, 2002.

[CM02] A. Cockburn and B. McKenzie. Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 203–210. ACM Press, 2002.

[Coc04] A. Cockburn. Revisiting 2D vs 3D implications on spatial memory. In *Proceedings of the 5th conference on Australasian user interface*, pages 25–31. Australian Computer Society, 2004.

[Coo84] R. L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM Press, 1984.

[CWB02] J. Chen, X. Wang, and J. E. Bresenham. The analysis and statistics of line distribution. *Computer Graphics & Applications*, 22(6):100–107, 2002.

[CZH$^+$00] Y.-Y. Chuang, D. E. Zongker, J. Hindorff, B. Curless, D. H. Salesin, and R. Szeliski. Environment matting extensions: towards higher accuracy and real-time capture. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 121–130. ACM Press/Addison-Wesley Publishing Co., 2000.

[DAHC86] Jr. D. Austin Henderson and Stuart Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph.*, 5(3):211–243, 1986.

[DB97] P. J. Diefenbach and N. I. Badler. Multi-pass pipeline rendering: realism for dynamic environments. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 59–70. ACM Press, 1997.

[DC95] E. Driskill and E. Cohen. Interactive design, analysis, and illustration of assemblies. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 27–34. ACM Press, 1995.

[DCKL97] M. D'Zmura, P. Colantoni, K. Knoblauch, and B. Laget. Color transparency. *Perception*, 26(4):471–492, 1997.

[DE04a] J. Diepstraten and M. Eissele. In-Depth Performance Analyses of DirectX9 Shading Hardware concerning Pixel Shader and Texture Performance. In Wolfgang Engel, editor, *Shader X3*, pages 523–544. Charles River Media, 2004.

[DE04b] J. Diepstraten and T. Ertl. Interactive rendering of reflective and transmissive surfaces in 3D toon shading. In *Proceedings of GI Workshop Methoden und Werkzeuge zukuenftiger Computerspiele '04*, 2004.

[Dec96] P. Decaudine. Cartoon-looking rendering of 3D-scenes. Technical Report 2919, INRIA, 1996.

[DGE04] J. Diepstraten, M. Görke, and T. Ertl. Remote line rendering for mobile devices. In *Proceedings of IEEE Computer Graphics International (CGI)'04*, pages 454–461, 2004.

[DH73] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

[DH98] J. Döllner and K. Hinrichs. Interactive, animated 3D widgets. In *CGI '98: Proceedings of the Computer Graphics International 1998*, pages 278–297. IEEE Computer Society, 1998.

[Die96] P. Diefenbach. *Pipeline Rendering: Interaction and realism through hardware-based multi-passrendering*. PhD thesis, University of Pennsylvania, 1996.

[DP73] D. H. Douglas and T. K. Peucker. Algorithm for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[Dur99] F. Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Université Joseph Fourier, Grenoble I, July 1999.

[DWE02] J. Diepstraten, D. Weiskopf, and T. Ertl. Transparency in technical illustrations. *Computer Graphics Forum*, 21(3):317–325, 2002.

[DWKE04] J. Diepstraten, D. Weiskopf, M. Kraus, and T. Ertl. Vragments - Relocatability as an extension to programmable rasterization hardware. In *Proceedings of WSCG 2004 Short Papers*, 2004.

[ESE00] K. Engel, O. Sommer, and T. Ertl. A framework for interactive hardware accelerated remote 3D-visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 167–177,291, 2000.

[Eve00] C. Everitt. One-pass silhouette rendering with GeForce and GeForce2. NVidia Whitepaper, 2000.

[Eve01] C. Everitt. Interactive order-independent transparency. White paper, NVidia, 2001.

[FBC$^+$95] J.-D. Fekete, E. Bizouarn, E. Cournarie, T. Galas, and F. Taillefer. Tictactoon: a paperless system for professional 2D animation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 79–90. ACM Press, 1995.

[FE02] F. Faul and V. Ekroll. Psychophysical model of chromatic perceptual transparency based on subtractive color mixture. *Journal Optical Society America A*, 19(6):1084–1095, 2002.

[FMS02] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Real-time halftoning: A primitive for non-photorealistic shading. In *Proceedings of Eurographics Workshop on Rendering*, pages 227–231, 2002.

[Fur86] G. W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23. ACM Press, 1986.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Massachusetts, 1990.

[GA05] J.-L. Gailly and M. Adler, 2005. Web site: http://www.gzip.org/zlib.

[GGSC98] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452. ACM Press, 1998.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBB-tree: A hierarchical structure for rapid interference detection. In *Proceedings of ACM SIGGRAPH 96*, pages 171–180, 1996.

[Goo98] A. Gooch. *Interactive Non-Photorealistic Technical Illustration*. PhD thesis, University of Utah, 1998.

[Gou05] D. Gould. Illustrate! cel shading plugin for 3D Studio Max, 2005.

[GP01] J. Gettys and K. Packard. The X Resize and Rotate Extension - RandR. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, pages 235–243. USENIX association, 2001.

[Gre03] S. Green. Summed area tables using graphics hardware. GDC2003 Presentation, 2003.

[GSG$^+$99] B. Gooch, P.-P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, pages 31–38, 1999.

[GSTD90] W. Gerbino, C. I. F. H. J. Stultiens, J. M. Troost, and C. M. M. De Weert. Transparent layer constancy. *Journal of Experimental Psychological Human Perception Performance*, 16(1):3–20, 1990.

[HH84] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 119–127. ACM Press, 1984.

[HK93] P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 165–174. ACM Press, 1993.

[HMG03] K. E. Hillesland, S. Molinov, and R. Grzeszczuk. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. *ACM Trans. Graph.*, 22(3):925–934, 2003.

[Hod89] E.R.S. Hodges. *The Guild Handbook of Scientific Illustration*. Van Nostrand Reinhold, New York, 1989.

[Hop96] H. Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108. ACM Press, 1996.

[HS99] W. Heidrich and H.-P. Seidel. Realistic, hardware-accelerated shading and lighting. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 171–178. ACM Press/Addison-Wesley Publishing Co., 1999.

[HSS98] J. Hamel, S. Schlechtweg, and T. Strothotte. An approach to visualizing transparency in computer-generated line drawings. In *Proceedings of IEEE Information Visualization 1998*, pages 151–156, 1998.

[Hyb05] Hybrid Graphics Ltd. Gerbera, 2005. Web site: http://www.hybrid.fi/main/esframework/index.php.

[IBM05] IBM Corporation. Deep Computing Visualization, 2005. Web site: http://www-1.ibm.com/servers/deepcomputing/visualization/.

[INH] K. Iourcha, K. Nayak, and Z. Hong. System and method for fixed-rate-block-based-image compression with inferred pixel values. US Patent 5,956,431.

[Inm05] Inmar Software. Diesel Engine SDK, 2005. Web site: http://www.3darts.fi/mobile/de.htm.

[Int88] International Telecommunication Union. CCITT volume VII fascicle VII.3, 1988.

[Int05a] Intel Corporation. Intel 2700g multimedia accelerator, 2005. Web site: http://www.intel.com/design/pca/prodbref/300571.htm.

[Int05b] Intel Corporation. Intel graphics performance primitives, 2005. Web site: http://www.intel.com/design/pca/applicationsprocessors/ swsup/gpp.htm.

[JEH00] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Hardware-accelerated texture advection for unsteady flow visualization. In *Proceedings of the conference on Visualization '00*, pages 155–162. IEEE Computer Society Press, 2000.

[JMLH01] H. W. Jensen, S. R. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518. ACM Press, 2001.

[Joi05] Joint Photographic Experts Group. JPEG 2000, 2005. Web site: http://www.jpeg.org/jpeg2000/.

[KG79] D. Scott Kay and D. Greenberg. Transparency for computer synthesized images. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, pages 158–164. ACM Press, 1979.

[Khr05] Khronos Group. OpenGL ES - the standard for embedded accelerated 3d graphics, 2005. Web site: http://www.khronos.org/opengles/.

[Kil99] M. Kilgard. Creating reflections and shadows using stencil buffers (NVidia technical demonstration). Game Developer Conference, 1999.

[Kil01] M. J. Kilgard, editor. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2001.

[Kir98] D. B. Kirk. Unsolved problems and oppertunities for high-quality, high-performance 3D graphics on PC platform. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 11–13, 1998.

[KSW04] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: A GPU-based particle engine. In *Eurographics Symposium Proceedings Graphics Hardware 2004*, pages 115–122, 2004.

[KW03] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[LaQG$^+$97] G. Leach, G. al Qaimari, M. Grieve, N. Jinks, and C. McKay. Elements of a three-dimensional graphical user interface. In *INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 69–76. Chapman & Hall, Ltd., 1997.

[LAS04] W. Li, M. Agrawala, and D. Salesin. Interactive image-based exploded view diagrams. In *Proceedings of the 2004 conference on Graphics interface*, pages 203–212. Canadian Human-Computer Communications Society, 2004.

[LL94] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458. ACM Press, 1994.

[LMHB00] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3D animation. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, pages 13–20, 2000.

[LPFH01] J. L., E. Praun, A. Finkelstein, and H. Hoppe. Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 227–232. ACM Press, 2001.

[Mam89] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, July 1989.

[Mar89] J. Martin. *Technical Illustration: Material, Methods, and Techniques*, volume 1. Macdonald and Co, 1989.

[MC00] K.-L. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Supercomputing*, 2000.

[Met74] F. Metelli. The perception of transparency. *Scientific America*, 230(4):90–98, 1974.

[Mic05a] Microsoft Corporation. Direct3d mobile, 2005. Web site: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcemultimedia5/html/wce50oriDirect3DMobile.asp.

[Mic05b] Microsoft Corporation. Using the pocket pc game api, 2005. Web site: http://msdn.microsoft.com/library/default.asp?url=
/library/en-us/guide_ppc/htm/_games_using_the_gapi_run_times.asp.

[MKB$^+$03] T. Mertens, J. Kautz, P. Bekaert, H.-P. Seidel, and F. Van Reeth. Interactive rendering of translucent deformable objects. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 130–140. Eurographics Association, 2003.

[MKTH97] L. Markosian, M. Kowalski, S. Trychin, and J. Hughes. Real-time non-photorealistic rendering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 415–420, 1997.

[Mov93] Moving Picture Experts Group. MPEG-1, 1993.

[ND04] M. Nienhaus and J. Döllner. Blueprints - illustrating architecture and technical parts using hardware-accelerated non-photorealistic rendering. In *Proceedings of the 2004 conference on Graphics interface*, pages 49–56. Canadian Human-Computer Communications Society, 2004.

[NM00] J. D. Northrup and L. Markosian. Artistic silhouettes: a hybrid approach. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, pages 31–37. ACM Press, 2000.

[NNS73] M.E. Newell, R.G. Newell, and T.L. Sancha. A new approach to the shaded picture problem. In *Proceedings of ACM*. ACM Press, 1973.

[Nok05] Nokia. N-Gage portable cellular phone game console. Web Site: http://www.n-gage.com, 2005.

[NSPU99] S. Nakauchi, P. Silfsten, J. Parkinnen, and S. Usui. Computational theory of color transparency: recovery of spectral properties for overlapping surfaces. *Journal Optical Society America A*, 16(11):2612–2624, 1999.

[NT00] F. S. Nooruddin and Greg Turk. Interior/exterior classification of polygonal models. In *Proceedings of IEEE Visualization 2000*, pages 415–422, 2000.

[NVI05] NVIDIA Corp. Cg Language Specification, 2005. Web site: http://developer.nvidia.com/page/cg_main.html.

[Nye95] A. Nye, editor. *Volume 0: X Protocol Reference Manual*. X Window System Series. O'Reilly & Associates, 4th edition, 1995.

[Obe05] M. Oberhumer, 2005. Web site: http://www.oberhumer.com/opensource/lzo.

[OMP03] T. Strothotte O. M. Pastor, B. Freudenberg. Real-time animated stippling. *IEEE Computer Graphics and Applications*, 23(4):62–68, 2003.

[OR98] E. Ofek and A. Rappoport. Interactive reflections on curved objects. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 333–342. ACM Press, 1998.

[Oy05] Bitboys Oy. Acceleon 3D graphics hardware for Wireless Devices, 2005. Web site: http://www.acceleon.com.

[Pac01] K. Packard. Design and implementation of the X rendering extension. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, pages 213–224. USENIX association, 2001.

[PBMH02] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712. ACM Press, 2002.

[PD03] P. Peers and P. Dutré. Wavelet environment matting. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 157–166. Eurographics Association, 2003.

[PH89] K. Perlin and E. M. Hoffert. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262. ACM Press, 1989.

[Raa98] A. Raab. *Techniken zur Interaktion mit und Visualisierung von geometrischen Modellen*. PhD thesis, School of Computer Science, Otto-von-Guericke University of Magdeburg, 1998.

[Ree83] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 359–375. ACM Press, 1983.

[Rig03] G. Riguer. Performance optimization techniques for ATI graphics hardware with DirectX 9.0. White paper, ATI, 2003.

[RKSZ94] T. Rist, A. Krüger, G. Schneider, and D. Zimmermann. Awi: a workbench for semi-automated illustration design. In *Proceedings of the workshop on Advanced visual interfaces*, pages 59–68. ACM Press, 1994.

[RLWB03] S. Rusdorf, M. Lorenz, S. Wölk, and G. Brunnett. Virtualiti3d (v3d): A system-independent, real time-animated, threedimensional Graphical User Interface. In *Proceedings of the 3rd IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2003)*, pages 955–960. ACTA Press, 2003.

[Rob94] B. Robertson. Disney lets CAPS out of the bag. *Computer Graphics World*, 1994(7):58–64, 1994.

[RS97] A. Rappoport and S. Spitz. Interactive boolean operations for conceptual design of 3D solids. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 269–278. ACM Press/Addison-Wesley Publishing Co., 1997.

[RSFWH98] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[RvDR$^+$00] G. Robertson, M. van Dantzich, D. Robbins, M. Czerwinski, K. Hinckley, K. Risden, D. Thiel, and V. Gorokhovsky. The task gallery: a 3D window manager. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 494–501. ACM Press, 2000.

[SA99] M. Segal and K. Akeley, editors. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, 1999.

[SA01] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification Version 1.3, 2001.

[SA03] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification Version 1.5, 2003.

[SA04] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification Version 2.0, 2004.

[SAM04] J. Ström and T. Akenine-Möller. PACKMAN: Texture Compression for Mobile Phones, 2004. SIGGRAPH 2004 Sketch.

[SC92] P. S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 341–349. ACM Press, 1992.

[SDWE03] S. Stegmaier, J. Diepstraten, M. Weiler, and T. Ertl. Widening the remote visualization bottleneck. In *Proceedings of ISPA '03*. IEEE, 2003.

[SF91] D. D. Seligmann and S. Feiner. Automated generation of intent-based 3D illustrations. *Computer Graphics (Proceedings of ACM SIGGRAPH 91)*, 25(4):123–132, 1991.

[SF93] D. D. Seligmann and S. Feiner. Supporting interactivity in automated 3D illustrations. In *Proceedings of the 1st International Conference on Intelligent User Interfaces*, pages 37–44. ACM Press, 1993.

[Sha94] M. A. Shantzis. A model for efficient and flexible image computing. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 147–154. ACM Press, 1994.

[Shn03] B. Shneiderman. Why not make interfaces better than 3D reality? *Computer Graphics & Aplications*, 23(6):12–15, 2003.

[Sil05] Silicon Graphics, Inc. OpenGL Vizserver 3.0 – Application-Transparent Remote Interactive Visualization and Collaboration, 2005. Web site: http://www.sgi.com/.

[SLS00] M. Seul, L.O'Gordman, and M. J. Sammon. *Practical Algorithms for Image Analysis: Descriptions, Examples and Code*. Cambridge University Press, Cambridge, 2000.

[SME02] S. Stegmaier, M. Magallón, and T. Ertl. A generic solution for hardware-accelerated remote visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '02*, 2002.

[SMM00] A. Sanna, B. Montrucchio, and P. Montuschi. A survey on visualization vector fields by texture-based methods. *Recent Res. Devel. Pattern Rec.*, 1:13–27, 2000.

[Spi01] J. Spitzer. Geforce 3 OpenGL performance. White paper, NVidia, 2001.

[SS02] T. Strothotte and S. Schlechtweg. *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation.* Morgan Kaufmann Publishers, 2002.

[ST90] T. Saito and T. Takahashi. Comprehensible rendering of 3-D shapes. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, volume 24, pages 197–206, 1990.

[Sun05] Sun Microsystems Inc. Java3D Scenegraph. Web site: http://java.sun.com/products/java-media/3D/index.jsp, 2005.

[SZH94] M. P. Stevens, R. C. Zeleznik, and J. F. Hughes. An architecture for an extensible 3D interface toolkit. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 59–67. ACM Press, 1994.

[Tal34] W.H.F. Talbot. Experiments on light. *Philosophical Magazine (3rd series)*, 5:321–334, 1934.

[Tho68] T.A. Thomas. *Technical Illustration.* McGraw-Hill, second edition, 1968.

[TK96] J. Torborg and J. T. Kajiya. Talisman: commodity realtime 3D graphics for the pc. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 353–363. ACM Press, 1996.

[Top00] A. Topol. Immersion of xwindow applications into a 3D workbench. In *CHI '00: CHI '00 extended abstracts on Human factors in computing systems*, pages 355–356. ACM Press, 2000.

[Tre05a] Thierry Tremblay. PocketFrog SDK. http://pocketfrog.droneship.com/index.html, 2005.

[Tre05b] Thierry Tremblay. PocketHAL, 2005. Web site: http://pockethal.droneship.com/.

[Tuf97] E. Tufte. *Visual Explanations.* Graphic Press, 1997.

[Vik05] Viktoria Institute. GapiDraw graphics SDK, 2005. Web site: http://www.gapidraw.com.

[Wag05] D. Wagner. Klimt - the open source 3d graphics library for mobile devices, 2005. http://studierstube.org/klimt/index.php.

[Wal81] B. A. Wallace. Merging and transformation of raster images for cartoon animation. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 253–262. ACM Press, 1981.

[WFZ02] Y. Wexler, A. W. Fitzgibbon, and A. Zisserman. Image-based environment matting. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 279–290. Eurographics Association, 2002.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[Wie05] TU Wien. Studierstube Augmented Reality Project, 2005. Web site: http://www.studierstube.org.

[Wol89] J.D. Wolter. On the automatic generation of assembly plans. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 62–68. IEEE, 1989.

[WSB01] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th EUROGRAPHICS Workshop on Rendering*, pages 277–288. Springer, 2001.

[ZHR$^+$93] R. C. Zeleznik, K. P. Herndon, D. C. Robbins, N. Huang, T. Meyer, N. Parker, and J. F. Hughes. An interactive 3d toolkit for constructing 3D widgets. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 81–84. ACM Press, 1993.

[ZI97] H. Zhang and K. Hoff III. Fast backface culling using normal masks. In *Proceedings Symposium on Interactive 3D Graphics 1997*, pages 103–106, April 1997.

[ZWCS99] D. E. Zongker, D. M. Werner, B. Curless, and D. H. Salesin. Environment matting and compositing. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 205–214. ACM Press/Addison-Wesley Publishing Co., 1999.