An acceleration architecture for DOpE

Robert Wetzel
rw8@inf.tu-dresden.de
Dresden University of Technology
Operating Systems Group

31.08.2003

Contents

1 Introduction **5**

1.1 XFree86 6

1.1.1 XAA 6

1.1.2 DRI 8

1.1.3 Utah-GLX 8

1.2 OpenGL/ES 8

1.3 DirectFB 8

1.4 Conclusion 9

2 Design **11**

2.1 General Architecture 11

2.1.1 A dedicated acceleration library 11

2.1.2 Splitting into server and client 12

2.1.3 Interface between server and client 13

2.1.4 The IPC overhead 15

2.2 Realtime support for DOpE 15

3 Implementation **17**

3.1 The client-server interface 17

3.1.1 Gfx container 17

3.1.2 Drawing primitives 18

3.1.3 Mapping of gfx container data 18

3.1.4 Mouse handling 18

3.2 Modules 18

3.3 Dispatcher . . . 18
3.4 Configuration . . . 20
3.5 Drivers . . . 20
3.5.1 Driver initialisation . . . 21
3.5.2 Software fallback . . . 21
3.5.3 Caching . . . 22
3.6 The client library interface . . . 22

4 Evaluation **23**
4.1 What to measure . . . 23
4.1.1 Line . . . 24
4.1.2 Filled rectangle . . . 24
4.1.3 Scaled image . . . 24
4.1.4 Measuring on server and client side . . . 24
4.2 Benchmark results . . . 24
4.2.1 IPC and marshalling . . . 25
4.2.2 Clipping . . . 26
4.2.3 Drawing primitives . . . 33
4.3 Models for predicting drawing times . . . 35
4.3.1 Clipping . . . 35
4.3.2 Destination colour-depth . . . 36
4.3.3 Drawing primitives . . . 37

5 Conclusions **41**

Chapter 1

Introduction

The graphical user interface DOpE (Desktop OPerating Environment) is part of DROPS, the Dresden Realtime OPerating System. A screenshot of DOpE is shown in figure 1.1.

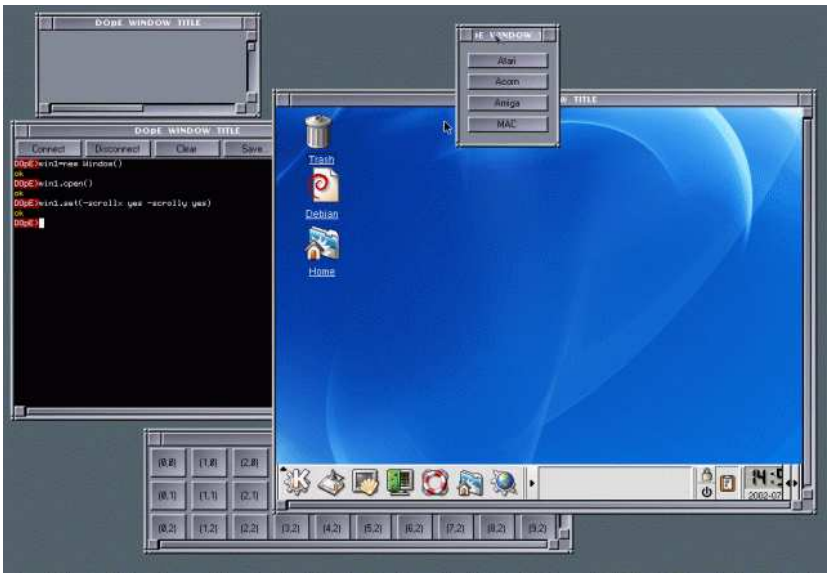

Figure 1.1: Screenshot of DOpE, taken from [2]

DOpE runs under DROPS as well as under Linux. The latter is mainly used for faster development and prototyping of new features. This work concentrates on the DROPS port of DOpE.

Currently, DOpE draws all graphics by using software rendering. The performance bottleneck in the rendering process is the data transfer between system memory and the local memory of the graphics card (see [1], page 48, and 3.5.3 on page 22).

This work is directed at bringing hardware acceleration support to DOpE. A major problem with changing the drawing methods is the change in their time behaviour. DOpE schedules the realtime and non-realtime drawing tasks according to a prediction model. Having a different time behaviour, the new hardware accelerated drawing methods require new prediction

models, which are presented in chapter 4 of this thesis.

There are a number of projects and products aiming at hardware accelerated graphics support. A side-effect of the 3D acceleration of today's graphics cards are advanced features like transparent windows. With the increasing complexity of the graphics hardware, the interfaces of the acceleration libraries grow as well (a nice example for this is DirectX, which evolves with the new features of the hardware). Because the primary goal of this work is to accelerate the output of DOpE, only a very small subset of these graphics functions are needed.

A selection of acceleration architectures is presented in the following sections.

1.1 XFree86

XFree86 is a free (open source) implementation of the X Window System. It is rather complex: the source code is about 60 MB compared to about 2 MB for DOpE. Figure 1.2 visualises the complexity by showing the different parts of XFree together with part of the data flow. With version 4, a framework for 3D acceleration has been added to XFree86, complementing the already available 2D acceleration support. Two parts of XFree86, XAA and DRI, and a third-party module, Utha-GLX, are described further in the next sections.

1.1.1 XAA

The XFree86 Acceleration Architecture, abbreviated as XAA, offers a set of about 3 dozen functions to accelerate 2D output (i.e. drawing lines, filled rectangles, bitblitting). Every XAA graphics primitive has two functions in the interface, one to setup the hardware for the primitive and another one to actually paint the primitive. The paint-function can be called subsequently after setup has been done. For example, drawing a non-filled polygon results in a call to the setup function which sets the line color, line style, and so on, while the drawing function is called for every line the polygon consists of.

Attributes like line color, style, and so on are called context. The amount of data stored in the context depends on the complexity of the drawing functions, i.e. if you want to paint not only solid lines but also dashed ones you have to store the line pattern in the context.

DOpE only needs very basic variants, so most of the XAA primitives offer unnecessary features which add superfluous complexity. For example, XAA offers to paint dashed or dotted lines while DOpE only needs solid (but alpha-blended) horizontal and vertical lines. Due to the very limited amount of context data required, DOpE does not need special setup functions. All the necessary parameters can be handed directly to the drawing functions. Of course, dividing the setup and drawing functions is still an option if more context data is needed.

Another interesting feature of XAA are the flags associated with each primitive. These flags represent the capabilities of the hardware, like support of transparency in bitblits. This allows the X server to recognise cases where it has to draw primitives in software rather than relying on the hardware which is incapable of doing it.

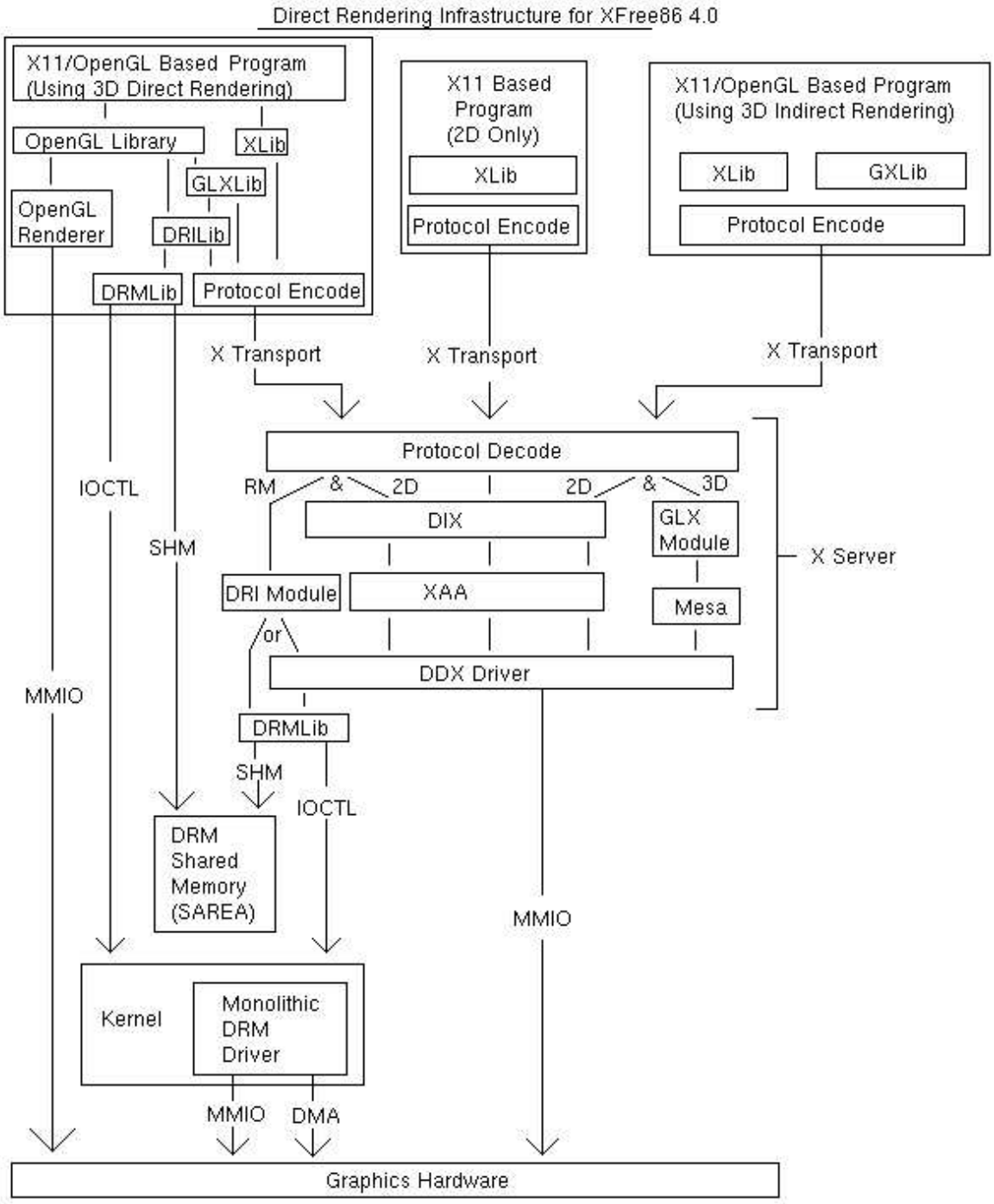

Figure 1.2: Diagram showing the complexity of XFree86 version 4.X; a full explanation of this diagram can be found at [5]

1.1.2 DRI

DRI stands for "Direct Rendering Infrastructure" and "is a software architecture for coordinating the Linux kernel, X window System, 3D graphics hardware and an OpenGL-based rendering engine." [6]. All these components build up a quite complex piece of software. OpenGL has a feature rich interface; the specification (version 1.4) contains more than 300 pages (without GLX and other supplementary specifications).

DRI has been introduced with the version 4 of the XFree86 system and offers a framework to write new, modular device drivers. Older versions of XFree86 lacked the infrastructure to support 3D directly. This led to the development of third party software like the below described Utah-GLX.

If there will be 3D support for DROPS, DRI can serve as a good source of information; for the current goal of accelerating DOpE it is rather useless.

1.1.3 Utah-GLX

Contrary to XAA and DRI Utah-GLX [7] is a third-party module for XFree86. Similar to DRI, it adds OpenGL/GLX support to the X Window System. It can be used with older XFree86 versions (3.3.x) as well as with XFree86 4.X. Unlike DRI, which runs only on Linux and specific variants of BSD, it was intended to support all platforms XFree86 is running on. This results in worse performance than DRI due to more indirection in the rendering process. Nonetheless, it has the same goal: acceleration of 3D programs based on OpenGL/GLX. Similar to DRI, it can be used as a source of information in case a 3D-API gets implemented for DROPS.

1.2 OpenGL/ES

The progress in computer technology leads to more powerful and smaller devices. Todays handhelds and other embedded systems are powerful enough to do 3D graphics. As the resources are limited, compared to office and home PCs, a special specification for such devices has been created: OpenGL/ES (ES stands for Embedded Systems) [8]. Compared to OpenGL, it lacks more complicated features, keeping implementations of this specification small and simple. Unless the full set of OpenGL features is required (many games will need it for example), OpenGL/ES can be the starting point of a 3D-API implementation for DROPS. It can be used in presentation graphics for example (3D-diagrams, visualisation of complex architectures, etc.).

1.3 DirectFB

DirectFB (FB stands for frame buffer) is aimed at having have a small size and low memory usage, so it can be used in embedded devices as well as on normal desktop machines. The graphics acceleration part is focused on 2D and supports handling of windows (window stack,

event management, etc.), different layers, surfaces and subsurfaces.

Surfaces build the basis of DirectFB. A surface is a rectangular area containing pixel data. In addition to the surfaces, DirectFB offers subsurfaces. A subsurface is a rectangular part of a surface. Changing the subsurface results in changes to the associated surface.

To be able to handle hardware features like overlays (which are often used to display video streams), DirectFB offers the layer-interface. Layers are screens located on top of each other building a layer-stack. For example, the overlays are always located on top of the normal screen.

DirectFB is using the kernel frame buffer driver for setting up the video card, including the access to the frame buffer memory and I/O space. For the actual drawing it uses its own accelerated drawing functions, by disabling the drawing functions in the kernel driver, if possible.

The drawing functions are located in the surface-interface of DirectFB, and most of them are simpler than the corresponding functions in the XAA interface. For example, it does not support dashed lines, only the normal ones, which is much closer to the requirements of DOpE than XAA. Most of the drawing functions are exactly what DOpE itself needs. Other features, like the window management, are handled by DOpE or another application.

1.4 Conclusion

There is other software which offers graphics acceleration, like DirectX, which is of no use to this project as it is not available as source code (one can access the API documentation though). Other projects are more or less similar to one mentioned above, offering 2D or 3D (or both) acceleration.

Although 3D is a nice feature it is also highly complex. As DOpE needs only 2D drawing functions, the most usable projects for this work are DirectFB, and, to some extend, XAA. If 3D is required in the future without the need of full OpenGL compliance, an OpenGL/ES implementation may be sufficient for non-gaming applications, like presentations, data visualisation and so on.

While the separation of setup and drawing functions as done in XAA is not necessary for DOpE, a common way of getting information about the capabilities of the used hardware may be useful. Several capabilities, like whether bitblit is possible from right to left, too, are not needed. This means that a single function for querying capabilities, e.g. support of alpha blending or support of scaled bitblits, should suffice.

The API of DirectFB is a set of interfaces (actually a set of function pointers in C structs) which contain references to the actual functions. This is similar to the module concept used in DOpE. For the current work, the surface interface is the most interesting. Future features of DOpE may require subsurface-like image handling.

Chapter 2

Design

Before and during the design of the hardware acceleration support software, a set of goals has been formulated which influenced the whole design process.

The major goal of adding hardware acceleration to DOpE also implies that a fast way to access the actual drawing functions is required. Besides, the acceleration architecture has to support DOpE in the matter of realtime by supplying DOpE with information about the predicted and the real drawing times.

Like DOpE itself, the software will be divided into modules. This allows a better transition to dynamically linked libraries and more readable and structured source code.

Another goal is the support of multiple graphics adapters at the same time, which has an impact on how the drivers for the hardware have to be written (e.g. splitting of code and data). Additionally, drivers should be easy to write nonetheless, which means most of the data bookkeeping has to be done by a driver framework. This keeps the the drivers clean, easily readable, and replaceable.

2.1 General Architecture

There are many different ways to accelerate the graphics output of DOpE. The simplest solution is a replacement of the current drawing methods. I have done this for a prototype for Matrox G400 compatible cards. Especially in high resolutions, the performance gain was clearly recognisable. Additionally, it supports colour depths of 16 and 32 bit, while the original DOpE only supports 16 bits. The effort of adding the 32 bit mode was small compared to adding software drawing functions for 32 bit to the original DOpE.

2.1.1 A dedicated acceleration library

The good news about the above mentioned solution is its simplicity and the direct way of accessing the hardware (short path between DOpE and the actual drawing functions). The bad news is the hardwired code for one specific graphics card.

This problem can be solved by using an abstract interface between DOpE and the drawing functions similar to the surface-interface of DirectFB. This interface can be implemented by different drivers which support different graphics hardware. During startup, the hardware is detected and the corresponding driver is loaded.

Supporting multiple screens and graphics cards at the same time requires a more abstract interface, together with a set of general functions to organise the handling.

Today's graphics hardware is not suited for handling accesses from different clients. A mediating software has to block all direct access to the graphics card(s), while offering an interface for synchronised access to the hardware.

A library with acceleration support allows one application to use the hardware. If another application needs hardware accelerated graphics, it has to synchronise the requests with the first one. An example of two different applications using hardware acceleration are DOpE running on one screen, while a video is drawn to a second screen at the same time (and the controls of the video program are shown in DOpE).

2.1.2 Splitting into server and client

While DOpE could function as a proxy to hardware accelerated drawing functions, it is not intended to be used like that, and would become much larger in size than it currently is.

A solution to this problem is the partitioning of the accelerator framework into two pieces: A server, which controls the access to the hardware, and a client library, which offers an interface to the applications.

Big client, small server

During the design of the architecture, there turned up several possibilities for dividing the work between the server and the client. The server is a task on its own, while the client is a combination of the actual application (e.g. DOpE), together with a library which is doing the communication with the server.

One approach is to put all the drawing functions into the client, while the server manages the resources. The idea behind this is to have fast access to the drawing functions, while the server controls the resources (frame buffer, I/O ports, IRQ) and offers the necessary functionality for the synchronisation between the clients.

Synchronisation is the biggest problem with this concept. If it is done by the clients, a single malicious or bug-prone client can wreak havoc, without any possibility to stop it. A server-controlled synchronisation does not work either, as the drawing functions are still on the client side (which means they can access the graphics hardware directly).

Furthermore, every client has the same set of drawing functions and no dynamic binding. This will increase the memory usage of the whole system with every accelerated application.

size in dwords	used cycles	time in μs
4	4888	8.146
8	4927	8.211

Table 2.1: Time usage of long IPCs, benchmarked with `pingpong` on a PIII 600MHz

Small client, big server

The severe disadvantage of clients being able to cause havoc leads to a different setup of server and client. This setup has a small client side library (mostly for the communication with the server) and a big server, which contains not only the resource management, but also the drawing functions.

As every call to the drawing functions is delayed by interprocess communication, this approach is slower than the client-based-drawing functions. On the other hand, the server can synchronise every incoming draw-request and prevent clients from messing with data they are not allowed to access.

Splitted Server

One problem of having the graphics card drivers in the server is the possible unstable behaviour of these drivers. This may lead to a total server fault. A way to capsule drivers is to put every single one of them into an own process. While this prevents the general fault of the server in case of an erroneous driver, it introduces another IPC in the path from client to actual drawing function. Table 2.1 shows the cycles spent for a long IPC together with the respective time usage for the test system. If one call to the server results in another one from the server to the driver, the time spent in communication doubles. The maximal possible amount of calls to the server halves for the same amount of time, if the requests are handled one after another. So the idea has been scratched in favour of speed.

Figure 2.1 shows the general design of the chosen approach, together with some internal structure of the server. Every client application uses the "libVideo"-library to communicate with the server in the figure. As the important things are handled by the server (the library only offers an interface to the application and does some data transformation and caching), multiple clients can access the server at the same time. The communication is done between the library and a dedicated communication module on the server side. This module distributes the incoming calls to the various other modules.

2.1.3 Interface between server and client

As the design has been chosen to be a client-server based architecture, the next important problem is the communication between client and server. Regardless of which way the data is transfered (simple IPC or by using a streaming interface like DSI), there are a couple of possibilities to handle the data:

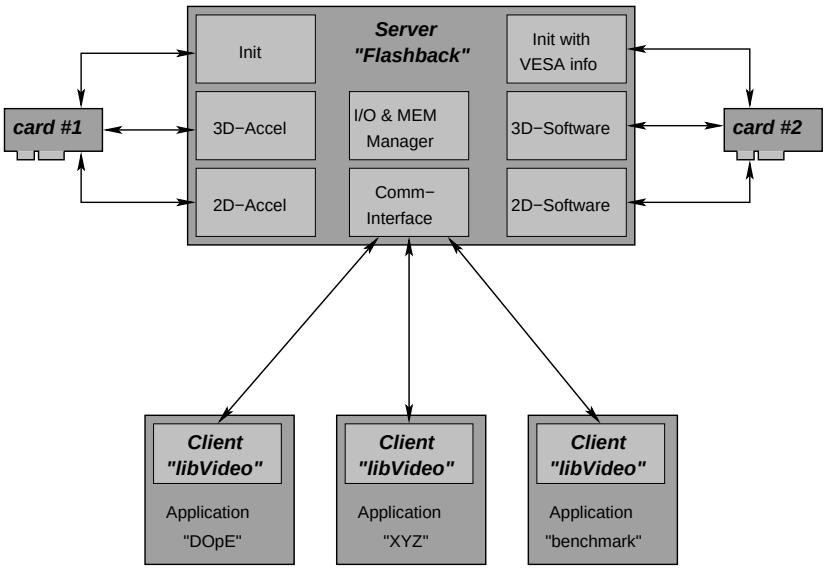

Figure 2.1: Chosen concept of the client and server and how they interact

Predefined interface

There is a predefined interface between server and client, which offers an "abstract" set of drawing functions. These drawing functions are not bound to any special hardware. Thus, every drawing function has to be translated into the necessary hardware commands by each driver in the server.

The upside of this approach is the abstract way of handling graphics. This gives clients a set of graphics functions which they can rely on. Furthermore, it allows easy handling of hardware features like multiple screens.

The downside is the limited interface. As long as all the needed features are supported, that does not pose a problem, but with the more and more complex graphics hardware, additional functionality has to be added to the interface. In the case of 2D the changes are minimal (e.g. addition of alpha-channel); the changes in the area of 3D are by far greater (latest addition are pixel shaders and programmable GPUs). Thus, a generic interface is suitable for 2D (and the goal of accelerating DOpE), while 3D needs an extensible interface.

Hardware specific interface

To access all the fancy features of today's graphics hardware, a hardware specific interface can be established between server and client. That way, the drivers in the server do not have to translate the incoming requests, which allows for faster handling of the requests. If the drivers just stream the requests through to the graphics hardware, the clients have indirect access to the graphics hardware (which is bad because one reason for the distinction between server and client is to prevent this). Unfortunately, the client library must contain all the hardware specific code. The result is a more "big client, small server" like architecture, even with the actual drawing functions located on the server side.

Mixed interface

To counter the problem of only hardware specific protocols, the interface has to be generic or abstract. Hardware specific functions have to be added via an extension mechanism.

One way to do this is to capsule the hardware specific data into a "container". Every time non-generic data has to be transfered, such a container is used.

A different way is used by OpenGL, which offers additional function calls for non-generic hardware features. Some of these have become common, while others are highly specific to a certain hardware. There exist meta-functions to retrieve information about the extensions and what functionality is available.

For the time being, the first (predefined interface) will be used, as there is no need for hardware specific acceleration functions. It is meant as 2D acceleration support; interfaces to support 3D are far more complex and probably need a full server redesign, including the interface.

2.1.4 The IPC overhead

A major factor which influenced the design of the different interfaces and data protocols between server and client is the overhead introduced by the IPCs (InterProcess Communication). I assumed the overhead (additional time consumption) to have serious impact on the total time used for the drawing operations.

In the simplest case, every call to a drawing function results in an IPC. With the above mentioned assumption it is usable for a first prototype. For later development steps other possibilities should be found though.

One possibility is to create a batch of drawing calls, which are delivered as a single request to the server, reducing the amount of IPC calls compared to the one-IPC-per-drawing-function scenario. The server can optimise the execution of the batch as it knows what is coming next.

2.2 Realtime support for DOpE

To schedule the realtime and non-realtime drawing tasks, DOpE uses predictions about the time needed to draw objects like buttons, realtime content (e.g. videos), labels, window borders, and so on. The prediction has to be done by the server, because the underlying prediction models may differ not only between graphics cards, but also between different drawing primitives.

DOpE uses a simple, yet effective way to predict the time for a drawing operation: It assumes a fixed amount of time is required for every drawn pixel that must be carried over the (PCI or AGP) bus to the graphics card. If it has to draw a box of the size 300x300 and this box is clipped to 100x100, it predicts the time costs on the basis of the 100x100 pixels, because other pixels are not drawn. Furthermore, it assumes a linear dependency between amount of drawn pixels and the respective used time (e.g. if 1 pixel needs N seconds, 100 pixels need 100*N seconds). This turns out to be a valid assumption for software graphics routines, as the bus bandwidth dominates the output performance.

This model may become obsolete, or is only half of the truth, if graphics hardware is used to draw. There are a number of factors which play an important role:

- Graphics hardware may or may not clip intelligently by calculating the area to be drawn. A simpler way is by checking every pixel to be drawn or not. In some cases (like 3D with Z-buffer), this is the most effective way to do it, while in 2D it is a waste of time. The check for a pixel to be drawn or not does not need as much time as to check and then draw the pixel. So the model needs information about how many pixels are drawn and how many pixels are clipped.
- Drawing scaled and alpha blended images relies on the texturing unit of graphics cards. The drawing speed depends not only the the texture size, but also on the resulting size, the speed of the texturing unit, and the memory bandwidth between the texturing unit and the memory where the texture is stored. Additionally, it is influenced by optional parameters, like the filtering method, which has impact on the resulting image quality.
- By using advanced features like AGP, bus mastering (the graphics card issues data read/write commands on the PCI/AGP bus instead of the CPU) the time prediction models get even more complicated. In a highly optimised driver, the time used to draw a primitve may depend not only on the parameters of the primitive itself, but also on the previously drawn primitive(s).
- The hardware optimises memory accesses by loading several pixels at once (e.g. 32 byte at once). Sizes which do not fit into this alignment may need additional cycles to get handled correctly, thus decreasing performance. This leads to a trade-off between performance and memory usage, which is influenced by many things, like how much memory is available, whether the performance has influence the user experience, and so on.

Chapter 3

Implementation

While the previous chapter was focused on the concepts and ideas for the acceleration architecture, this chapter highlights some aspects of the actual implementation. First, the client-server interface will be discussed, followed by some selected aspects of the guts of the server itself.

3.1 The client-server interface

As discussed in the last chapter, the implementation is based on a predefined, fixed interface. Additional functionality may require changes to the interface. For the specific task of supporting DOpE with accelerated drawing functions the need will most likely not aris, as the set of the needed functions is known.

3.1.1 Gfx container

During the development of the interface, the concept of "gfx containers" was introduced (gfx as abbreviation for graphics). It has similarities to the surfaces used in DirectFB [4].

Gfx containers contain some sort of graphics, mostly rectangular pixel areas (e.g. images, screens,, etc.), but can also contain a font or a palette. With drawing functions, the content of the gfx containers can be manipulated, including the possibility to draw the contents of an image gfx container into another one (e.g. painting a picture on a screen).

Every gfx container has to be allocated before it can be used. A successful allocation returns an identification number, an ID. This ID is used in all other functions.

Behind the stage, the server associates the client task which allocated the gfx container with the container ID. This enables the server to check for access violations. A client using "random" container IDs will not be allowed to draw to foreign containers, as the calling client task is checked for access rights to the respective container.

3.1.2 Drawing primitives

To draw to the gfx containers, the interface offers a set of drawing primitives:

- solid line
- solid filled rectangle
- gradient filled rectangle
- scaled image
- printing text

As an improvement over the original DOpE, all drawing primitives have an alpha parameter for transparency effects.

3.1.3 Mapping of gfx container data

Due to the limited set of drawing primitives, they are often not very useful (e.g. for palettes or fonts) or not sufficient (e.g. for real time animations). In this case, clients can get the gfx container data mapped into their address space, so they can draw into them in any way they want. As the server does not know about the changes of the data, the client must send an update-call.

3.1.4 Mouse handling

Additionally to the set of functions for management and handling of gfx containers, there exist two more, which manage the hardware based mouse pointer. While one function sets a new shape of the mouse cursor, the other allows setting the current mouse cursor position.

3.2 Modules

Like DOpE, the acceleration server is built out of modules [1]. Every module has a defined calling interface. An instance of this interface is stored together with the module name at the module manager. The naming allows a lookup of specific module interfaces. A small example is illustrated in figure 3.1.

During startup, the module manager initialises all modules. Because the module manager itself needs to access modules during startup, there is a special bootstrap handling for these modules (common memory management and hashtable management for the lookup table).

3.3 Dispatcher

As already shown in picture 2.1 on page 14, there is a module, implementing the above described server interface, which is visible to all clients. This module is called ”dispatcher”.

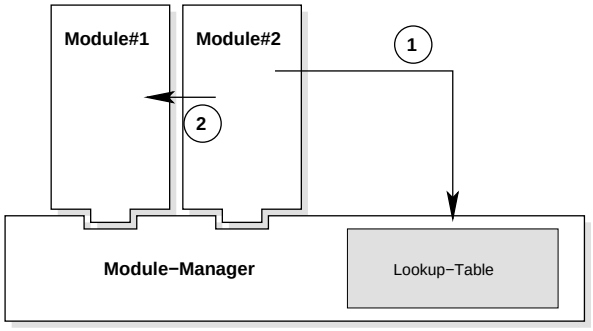

Figure 3.1: Modules and the module manager – (1) module 2 calls the module manager to look up the interface for module 1 (2) after module 2 retrieved the interface it calls the methods in module 1 it needs to get the work done

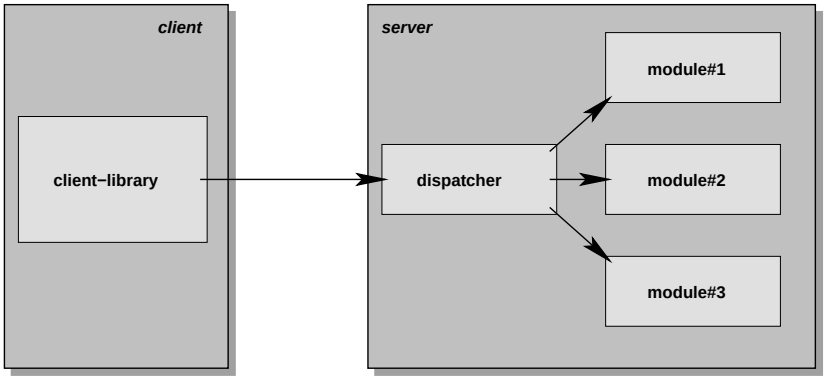

Figure 3.2: Role of the dispatcher

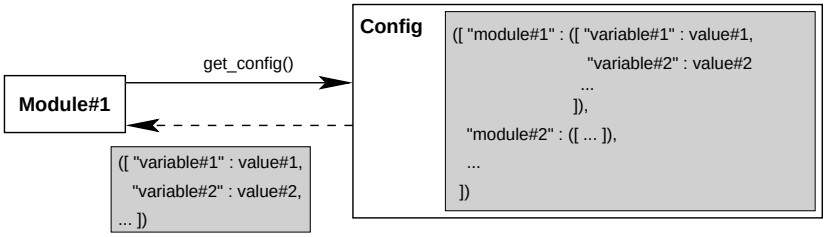

Figure 3.3: Interaction between a module and the config module

Whenever the dispatcher receives an incoming IPC, it calls different other modules to accomplish the task ("dispatching" the incoming call). Instead of having a lot of modules call each other, the dispatcher calls one after another, collecting data from one and giving it to the next. This way, these modules stay independent of each other. This simplifies not only debugging, but also rewrites of modules or reengineering of the code. Figure 3.2 shows where the dispatcher fits in the overall structure.

Because the dispatcher contains the handling of every call to the server from the beginning to the end, a modification can change the way how the server handles incoming calls. Furthermore, this design allows easy addition of time measurement of the drawing functions needed by DOpE for its realtime handling.

3.4 Configuration

During the implementation of the server, an increasing number of variable parameters piled up. These parameters should be configurable, either during startup or at runtime. Thus, I created a central point where all this data is stored, the configuration manager.

This configuration manager is responsible for fetching parameter data from the command line (or configuration files in the future) during startup. It allows other modules to retrieve this data, and change their behaviour according to it.

The configuration data is stored in a two level hashtable-tree, which is similar to the way .ini-files of Windows(tm) store their data. Every module has a set of key-value associations, which is stored under the name of the module in the top-level hashtable (see figure 3.3).

Additionally to the raw configuration data, the configuration module stores the module initialisation order. This information is used by the driver manager (see below for further information) to initialise the modules in the given order.

3.5 Drivers

A main part of the acceleration server are the drivers. They all provide the same interface to the server while managing different graphics hardware. Concerning the drivers there are three important concepts, which are explained below.

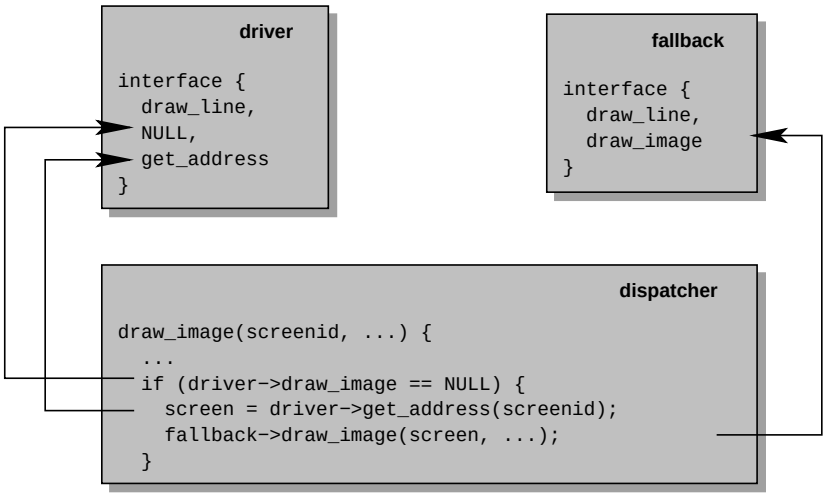

Figure 3.4: Fallback mechanism for software rendering methods

3.5.1 Driver initialisation

A special driver manager collects all the necessary information about the "hardware" (pointers to PCI information of installed VGA devices, multiboot information block which contains VESA information, etc.). When a driver module gets initialised, it registers with the driver manager. The driver manager subsequently calls a certain function in the driver for every dataset of "hardware" information found. This driver function checks the dataset for the hardware it can support and returns an "ok" or "not ok" to the driver manager.

The driver manager does not instantly call this driver function. The whole initialisation process is ordered according to data collected by the configuration module. Currently this means the drivers are initialised in the order they appear on the command line. Drivers not mentioned there are not initialised at all.

The driver function not only checks if the hardware information fits one of the supported hardware, it also allocates all the necessary resources of the hardware (like I/O ports and frame buffer) in case it is the hardware it supports.

After the hardware has been initialised, the driver manager gets the number of physical screens the hardware supports and builds up a mapping between the screens and the drivers. This way, the server knows which driver to call, when a specific screen has to be initialised.

3.5.2 Software fallback

Not every hardware supports all of the required drawing functions. To ease the work of driver programmers, these functions can be left out in the calling interface (set to NULL in the interface struct). The dispatcher checks for these missing functions every time and invokes the respective software fallback drawing function instead.

As the drivers keep the information about the address of the frame buffer to themselves, they have a function which returns the address of the screen to be drawn to.

The whole process is visualised in figure 3.4.

3.5.3 Caching

A very difficult topic is the caching of image data. Even today's hardware does not support every possible image format. Those pictures or font images have to be converted to a format the hardware can handle.

The fastest access to image data for a graphics card is to store this data in the local memory of the graphics card. Otherwise, the image data has to be transfered over the PCI/AGP bus, which is much slower (about 45MB/s for writes from CPU to Matrox G550 memory, while the G550 can access its own memory with a couple of Gigabytes per second). Thus, every image handled by the graphics card should be stored in the card's local memory.

One problem with the caching is the information where to cache the image. If a client allocates an image, the server does not know on which card (if there are more than one present) it is going to be displayed. In this case, the caching will be delayed until the first call to the image drawing function.

The other option is to store every image on every graphics card's RAM. As long as both screens always show the same (maybe in different resolutions) this is fine; otherwise, it is a waste of memory. Also, with different amounts of available memory, not all pictures which can be stored on one card can be so on another. These problems led to the above mentioned approach, although it will increase the drawing time by the delay introduced through the caching. With use of advanced features like busmastering, AGP, and write-combining this delay can be reduced compared to a non-accelerated CPU-based write.

Images can be mapped into the address space of clients. This allows them to freely edit the contents of the images. To inform the server about changes done, the client sends an update-message. With the informations sent, the server can update all cached incarnations of the updated image.

3.6 The client library interface

Applications using the acceleration server are linked against the client library interface. This interface is copied almost one to one from the client-server interface making it a mere wrapper around the client-server communication. This results in a very small client library. Furthermore, it allows to change the client-server-interface, while preserving the one to the application. This can be used to optimise the whole server and the client-server-interface for speed without the need to rewrite the applications based on the client-library.

Chapter 4

Evaluation

This chapter continues the discussion of realtime support for DOpE started at the end of chapter 2. Recalling the content of this section, there were two main goals: Adding time measurement to server and client as well as defining models for precalculation of the time needed for certain operations.

The first part was rather easy to implement using functions from the l4util package. As the interface between server and client has been designed with time measurement in mind, there have been no changes to it.

The second, and more important, point, namely defining models for time usage precalculation, is rather complex. I created a special client which does several benchmarks. Based on the outcome of the benchmark, a model of the underlying graphics hardware could be made.

4.1 What to measure

Today's graphics cards offer a lot of image processing features. The Matrox G550 is based on the Matrox G400, which offers alpha blending, bilinear filtering, different filtering for up- or downscaling, Gouraud shading and so on. Not every drawing primitive uses all of these features. In fact, most of these are only used in conjunction with drawing textures (used for drawing scaled images).

Most of these features are not needed by DOpE, or can be treated as optional (like bilinear filtering), and have not been used during the benchmarks. Only the alpha blending has found its way even into the client-server-interface, as it will probably used later on by DOpE (for transparent windows).

More important is the way graphics cards are benchmarked. The more intelligent the hardware is, the more effort has to be put into the chip design. Additionally, it increases the chip size because more sophisticated algorithms require more transistors for the chip (resulting in a higher price, too). This means that not all operations are handled in an optimised way. For example, the clipping can be handled in a line by line or pixel by pixel way instead of precalculating all the borders.

4.1.1 Line

DOpE only uses horizontal and vertical lines, which makes them a subset of filled rectangles with a height or width of 1, respectively. Anyway, if measured on their own, the following parameters have to be taken into account:

- start and end point, resulting in the total amount of pixels (DOpE needs horizontal and vertical lines only)
- clipping rectangle, resulting in clipped and non-clipped pixels
- screen colour depth

4.1.2 Filled rectangle

Drawing filled rectangles with alpha blending has the following parameters:

- width and height, resulting in a total amount of pixels
- clipping rectangle, resulting in clipped and non-clipped pixels
- screen colour depth

4.1.3 Scaled image

Drawing scaled images with alpha blending is the most complex of the drawing primitives required by DOpE. This shows up in the parameter list used for benchmarking:

- width and height of scaled image, resulting in total amount of pixels
- width, height and colour-depth of original image
- clipping rectangle, resulting in clipped and non-clipped pixels
- screen colour depth

4.1.4 Measuring on server and client side

For DOpE, the measured times on the client side are interesting. These include marshalling, the IPC and the server-side handling of the request. The measurement on the server-side eleminates the time overhead introduced by marshalling and IPC.

4.2 Benchmark results

The benchmarks have been taken with the following hardware:

primitive non-clipped painted 20000 times	time usage in μs client side	server side	IPC overhead per call in μs	relative
rectangle (100x100)	988400	805114	9.16430	22.76%
rectangle (300x300)	7294479	7102544	9.59675	2.70%
image (100x100)[a]	2068985	1865234	10.18755	10.92%
image (400x400)[a]	16310597	16101382	10.46075	1.30%

[a]texture size 300x300

Table 4.1: IPC overhead in for selected primitives

- PIII (Katmai) 600MHz
- Intel 440BX chipset based mainboard (QDI P6440BX/B1)
- 448 MB SDRAM 100MHz
- Matrox G550 AGP with 32MB SD/SG(?)-RAM
- ATI Radeon VE (7000) AGP with 64MB DDR-RAM

The current drivers do not use any special features like AGP, write-combining and so on. Currently, this only has impact on how fast the image data is loaded into the local memory of the graphics board and on how fast the command stream of the Radeon driver can be served (the Matrox is programmed via MMIO, Memory Mapped I/O). Drawing primitives, like lines or filled rectangles, does not need any pixel data which is not already present in the local memory of the graphics card.

The drivers for the G550 and the Radeon do not necessarily need to wait for the card to finish the drawing process. They can send commands to the graphics cards while the graphics hardware is still busy with a previously initiated drawing operation. This feature can be disabled by waiting for the graphics hardware to become idle after a drawing operation has been started. In the following text this feature is called "idle-wait" and is assumed to be used per default. The drivers allow to change this behaviour during compile time.

L4's scheduling supports priorities which has an impact on the measured values. The server as well as the benchmarking client have both been started with the default priority for every benchmark.

4.2.1 IPC and marshalling

During the design phase, IPCs have been assumed to be costly. With the measurement of the times on server and client side, it is possible to give more accurate statements about the absolute and relative costs of marshalling and the IPCs.

Both drawing functions contain more parameters than a short IPC can take on a x86 architecture (6 respective 7 32 bit integers versus 2 32 bit integers for a short IPC), which means long IPCs have been used for every call. While the absolute time is about 9 to 11 μs per call (complete IPC with return value; the image drawing function needs one 32 bit integer more),

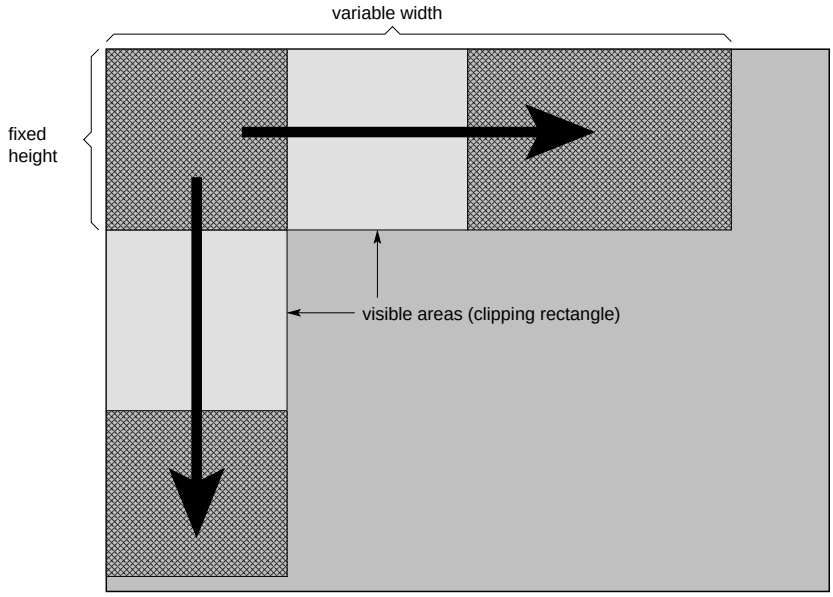

Figure 4.1: Clipping test 1 – test the behaviour before and after passing the clipping rectangle

the relative overhead in relation to the drawing operation itself varies. Especially operations on small sizes (e.g. for drawing button backgrounds, icons, etc.) have a much higher relative loss in time than huge drawing operations (e.g. realtime content, like video playback).

The overhead introduced to IPC and marshalling is not measured in the other following benchmarks. This eliminates some inaccuracy, as the overhead is not exactly the same every time (as shown in the table above).

4.2.2 Clipping

To show the effects of clipping, both cards had to paint filled rectangles with enabled hardware clipping. Every card had to pass 3 different tests:

The first test depicted in figure 4.1 draws clipped bars with a variable width or height, while the other dimension remains at a fixed size. The picture shows this for a fixed height and a variable width. This test shows the behaviour before and after the clipping rectangle has been passed.

The second test is aimed at an unbiased comparision between the variable width and the variable height case. The clipping rectangle is put into the top left corner, as shown in figure 4.2, so in both cases the graphics card has to deal with the same clipping rectangle.

In the third test, the clipping rectangle does not intersect any of the filled rectangles and by this removes the need to draw anything. Picture 4.3 illustrates this.

There are two basic ways of implementing clipping:

Scanline based clipping. While drawing the primitive, at the start of every (scan)line there is a check if this line fits into the clipping rectangle or not (checking with y-boundaries

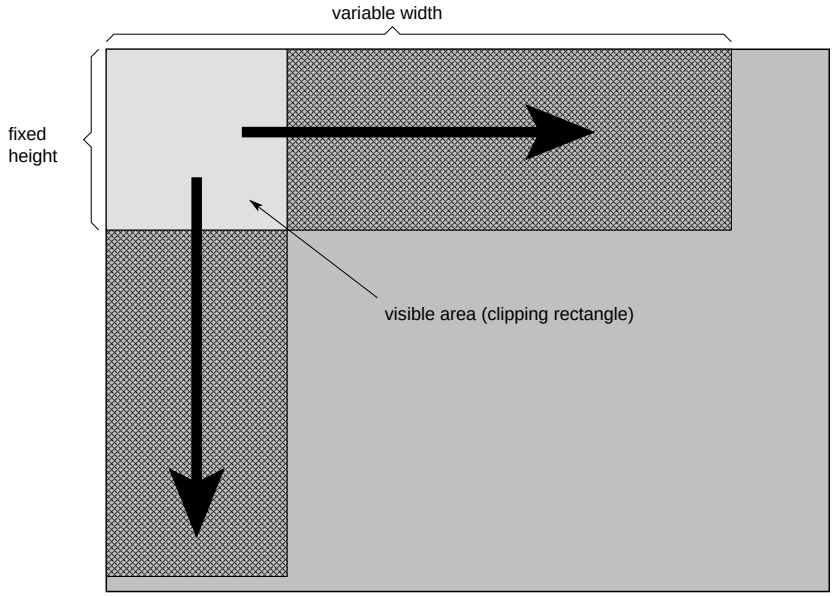

Figure 4.2: Clipping test 2 – comparision between variable width and variable height case

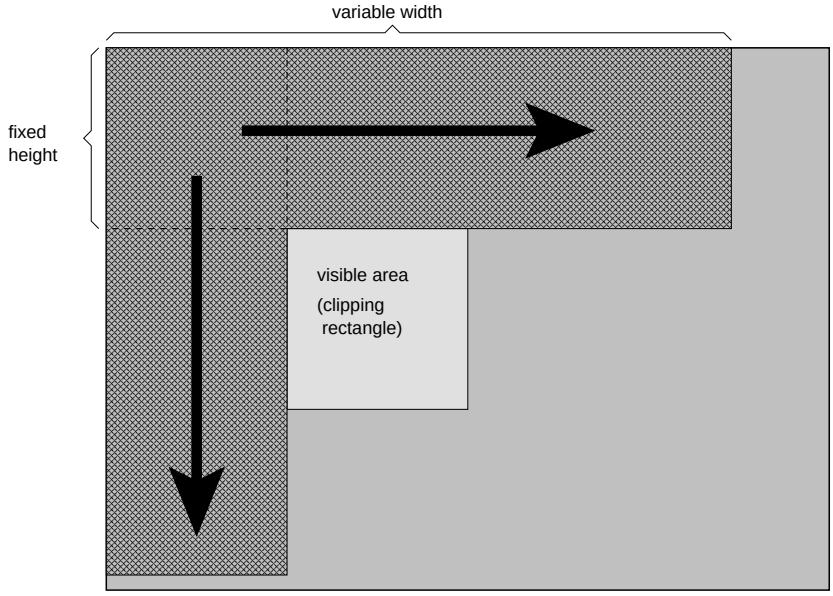

Figure 4.3: Clipping test 3 – behaviour in fully clipped case

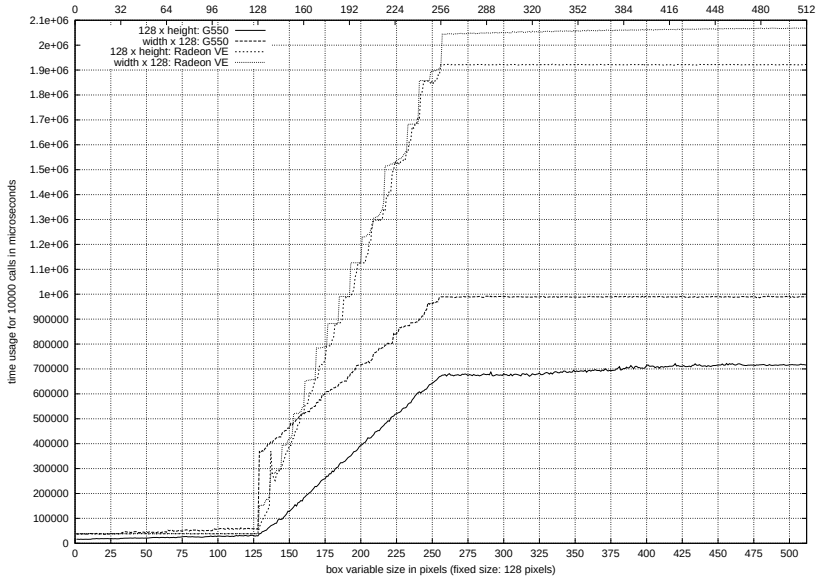

Figure 4.4: Clipping test 1 – G550 and Radeon VE in comparision

of the clipping rectangle). In case it lies between these boundaries, every pixel on this line is checked for containment in the clipping rectangle. This concept is very good for highly complex structures which are drawn in one pass, but inefficient with simple primitives like rectangles. DOpE only needs these simple primitives (lines are technically rectangles with a width or height of 1, filled rectangles and scaled images have a rectangular shape as well). Thus, precalculated clipping is a much better choice, as described below.

Precalculated clipping. Unlike the scanline based algorithm, the parameters for the actual area to be drawn are precalculated. For solid filled rectangles only the new start- and end-point have to be calculated. For lines the intersections with the resulting draw area have to be determined. In the case of scaled images, there are two calculations to be done. First, the drawing area has to be calculated in the way it is handled as with filled rectangles. Secondly, the start- and end-vertex in the original texture must be determined, as the clipping shows only a part of the whole texture.

Clipping test 1

There are a number of interesting things to see in the figure 4.4:

- With respect to the drawing speed, the Radeon VE is slower than the G550. The reason is most probably the memory bandwidth. While the Radeon VE can access the data in 2x64 bit, the G550 has the double bandwidth of 256 bit to its local memory. Both cards draw alpha blended filled rectangles. Thus, the speed shown includes both the read and write speed from and to memory (source and destination pixels have to be blended).
- The time required for precalculated clipping can be modeled with a constant function, $t(x) = const$. The constant value depends on the time needed to do the precalculation.

Figure 4.4 shows four cases of a constant clipping time. Only one of these originates from the G550. The other three show the clipping of the Radeon VE.

- The other four cases show increasing time usage while clipping is performed at the begin and the end of the curves. This can be interpreted as scanline based clipping, but both cards show the effect of precalculated clipping as well. Further investigation with idle-waiting switched off brought up a different picture, which is shown in figure 4.5. Especially in the clipped off area at the beginning, the G550 seems to do precalculated clipping without idle-waiting. This can be a wrong assumption though, as the G550 can still be busy with clipping, while already accepting the next command. Interestingly, it still shows a slightly increasing slope at the end of the variable-height-no-idle-wait curve, while in all other no-idle-wait cases the slope is zero.
- In the first figure (4.4) for clipping test 1 the curves show some other strange things, too:
  - In the variable width curves of the G550, there is an instant rise of the time usage, which I can only assume has its reason in the number of lines to be drawn. As the G550 renders at least the graphics scanline based, there is an overhead for every line. Thus the instant rise in the curve, as one pixel has to be drawn on each of the 128 scanlines at a total box with of 129.
  - The Radeon VE shows a spike at the beginning of the actual drawing process of the variable height curve. This spike shows up on other curves as can be seen below. As in the above case, I can not give any explanation for this behaviour.
  - There are stairs in both the variable width and variable height curve of the Radeon VE. These stairs may have their reason in the caching of pixel data in the 3D pipeline, minimising the accesses to the local memory of the Radeon VE. When the work on one block of pixels has been done, the next block has to be loaded, increasing the amount of time used, which results in the jumps ("stairs"). These effects show up more clearly with a higher amount of drawn pixels (e.g. increasing the number of calls or increasing the size of the drawn area).

Clipping test 2

As said above, this test is meant to compare both the variable width and variable height case on both cards. Figure 4.6 depicts how the G550 and Radeon VE handle it.

Although the clipping of the G550 in the variable width case is done better, the behaviour during the drawing process is much more erratic than in the variable height case. Both cases need about the same time to draw the visible rectangle. In the variable height case, the clipping needs an increasing amount of time, having a difference of about 0.06 seconds in the end.

The curves associated with the Radeon VE clearly show the above mentioned optimisation caused by the pixel cache, as well as the spike at the start. Unlike the G550, both cases need a different amount of time in total, there is a difference of about 0.15 seconds up to about 0.18 seconds between the 512 pixels wide and 512 pixels high rectangle case.

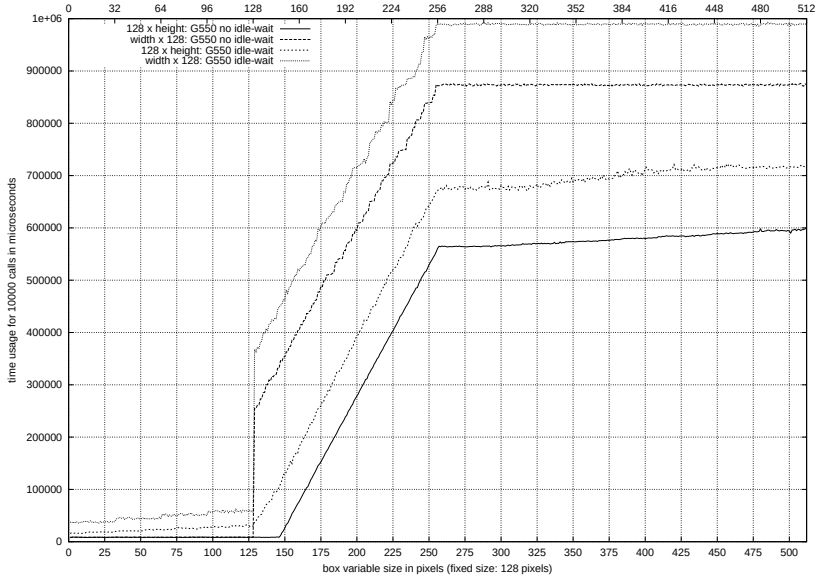

Figure 4.5: Clipping test 1 – G550 with and without idle-waiting

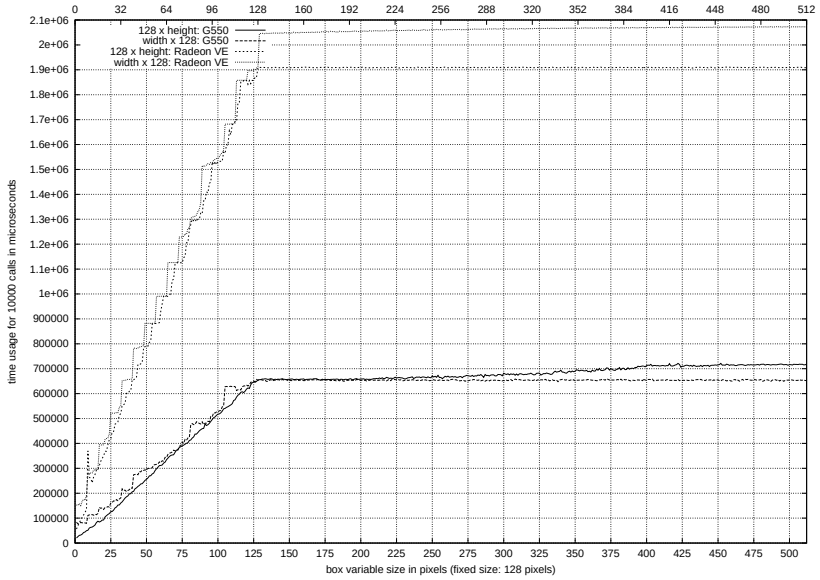

Figure 4.6: Clipping test 2 – comparision of variable height and variable width case under same clipping conditions

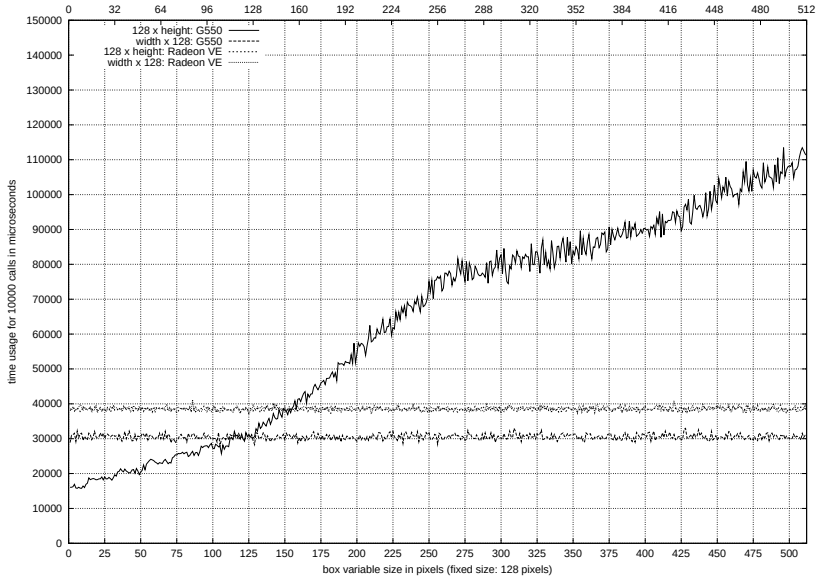

Figure 4.7: Clipping test 3 – behaviour in the fully clipped case

Both cards have an instant increase in the drawing time in the variable width case, which seconds my assumption about the scanline based rendering process mentioned above. Both variable height curves have a much smaller starting offset when the first scanline is drawn.

Clipping test 3

In the last clipping test, the visible area was outside the area the rectangles were drawn to, so not a single pixel had to be drawn at all. The resulting curves are shown in figure 4.7.

The noise on the curves are mostly due to measurement errors and effects of the scheduling (at least for the G550, as it waits in an poll-loop for the G550 to become ready again; the Radeon VE does the idle-wait in hardware).

The Radeon VE always needs the same time to handle the clipping, at about 3.8 μs for a call. The G550 handles the clipping in the variable width case even faster than the Radeon VE at about 3 μs.

The big difference is in the variable height curve of the G550. This curve shows a scanline based clipping algorithm:

- The curve can be split into three parts, where the first and the last one have about the same slope, whereas in the middle part the time usage increases much faster.
- The first part of the curve slowly increases until it intersects with the variable width case at about 128 pixels height, where the same rectangles are drawn.
- The higher increase in time usage during the height of 129 to 256 pixels is caused by the fact the lines match the y-boundaries of the clipping rectangle. Thus, every pixel

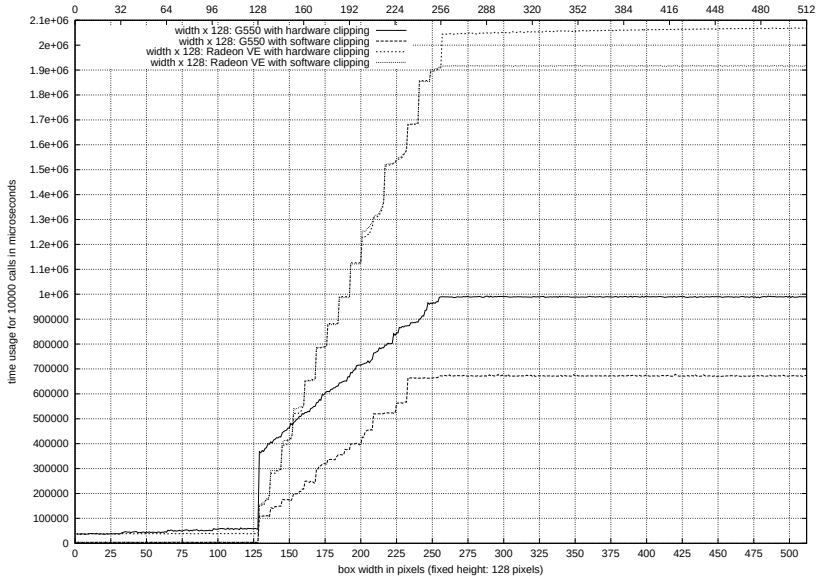

Figure 4.8: Software clipping vs. hardware clipping shown with clipping test 1, variable width case

on these lines has to be checked if it gets displayed or not. After the y-boundaries have been passed, the slope is similar to the one at the beginning.

The three tests yield some interesting results:

- The G550 seems to do scanline-based clipping. Tests with the idle-waiting switched off show otherwise, but this might be a result of the pipelined architecture and the FIFO(s) used to buffer incoming requests.
- Contrary to the G550, the Radeon VE probably does precalculated clipping, which is slower for smaller structures (as can be seen in the last test (figure 4.7)). The advantage is the easy predictability of the clipping.
- The partially strange behaviour of both cards, as well as the overall flaw of the clipping of the G550, lead to the conclusion that a predictable clipping method must be used which is based on precalculated clipping.

Software clipping

With the very dissatisfying results of the hardware clipping, especially from the G550, I tested how software clipping performs. The result turned out to be very interesting; two examples are shown in the figure 4.8.

In general, the software clipping was much faster than done in hardware, regardless of the algorithm used in hardware. Secondly, the overall time usage decreased. In this special, case the performance gain for the G550 is about 150% for full width rectangles (width = 512 pixels).

Although hardly to be seen, the software clipping uses a very small amount of time to calculate the clipping. It is about 0.3 to 0.4 μs, compared to about 4 μs for the hardware based clipping.

In the remarks about the clipping tests, I wrote about the instant rise of the time usage in the variable width clipping test 1 of the G550. Interestingly, this jump is much less with software clipping, compared to the hardware clipping case.

The effects on the Radeon VE are much smaller; especially during the drawing process both curves are more or less identical. The differences between hardware and software clipping show up at the beginning and the end of the curve, gaining a bit of performance over the hardware clipped case.

As a conclusion all drivers should use software clipping. Because the software is really fast at it I doubt even today's graphics cards are able to outperform it. Furthermore, modern processors are even faster than the one used in the test computer.

This brings up the question why hardware clipping is used at all. While simple primitives, like rectangles and lines, can easily be precalculated, complex 3D sceneries increase the effort for clipping drastically. Under these conditions I guess the hardware clipping is much more efficient than any software clipping, besides the technical problems to implement this.

4.2.3 Drawing primitives

Filled rectangles

These tests have been done on both the Radeon and the Matrox card, which allows an additional comparision of the drawing speed. To measure the drawing speed, the cards had to draw a variable width, non-clipped rectangle with a height of 512 pixels at 16 bits per pixel and at 32 bits per pixel colour-depth.

The results are shown in figure 4.9. In comparision, the G550 paints the alpha blended solid filled rectangles much faster than the Radeon VE (about three times faster). This is mainly caused by a twice as wide data bus to the local memory (256 bit vs. 2x64 bit of the Radeon VE) and probably different clock cycle speeds (no data about the G550 available) as well as the type of used RAM on the graphics cards.

Clearly visible are the optimisations done by the Radeon VE (pixelcache in 3D-pipeline), but also some regular anomalies in the behaviour of the G550. The overall dependency between the amount of pixels and the time used to draw them is linear, though. This is modified by a factor influenced by the colour-depth of the destination.

Scaled images

The major difference between the filled rectangle and the scaled image primitive is the texture required for the latter one. Thus, additional data has to be transfered from memory, influencing the time required to draw the scaled image. For the benchmarks, the textures have been stored in the local memory of the graphics card for fastest possible access.

Figure 4.10 shows the time usage for the variable width case. Noticeable are the spikes,

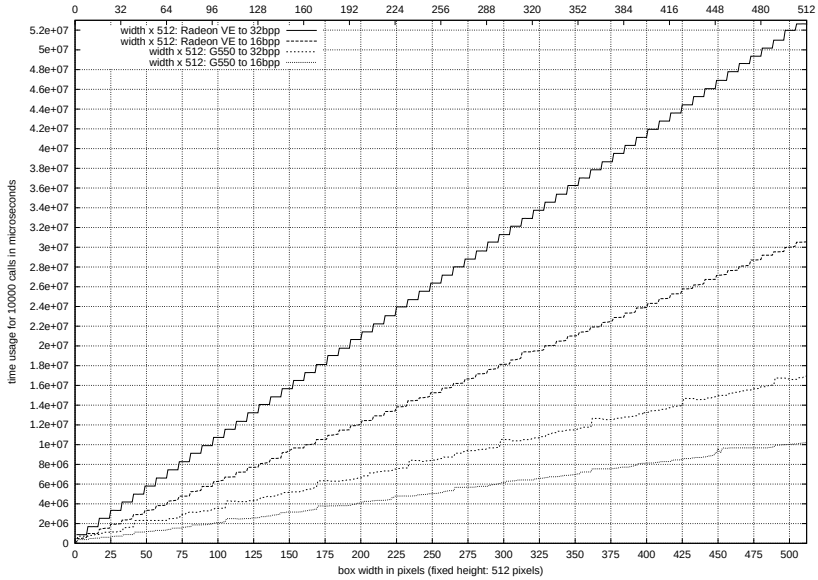

Figure 4.9: Fill test with G550 and Radeon VE in 16 and 32 bpp

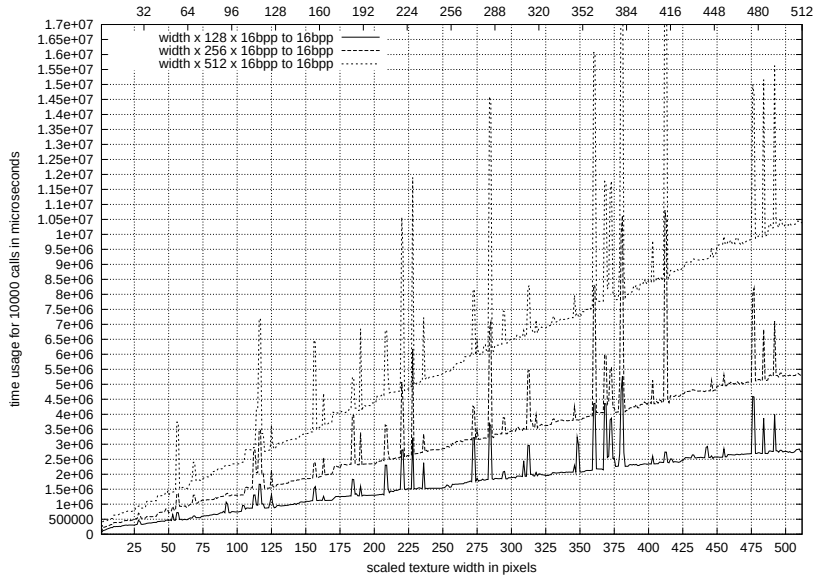

Figure 4.10: Scaled image time consumption based on a 128x128x16bpp texture (G550), variable width case

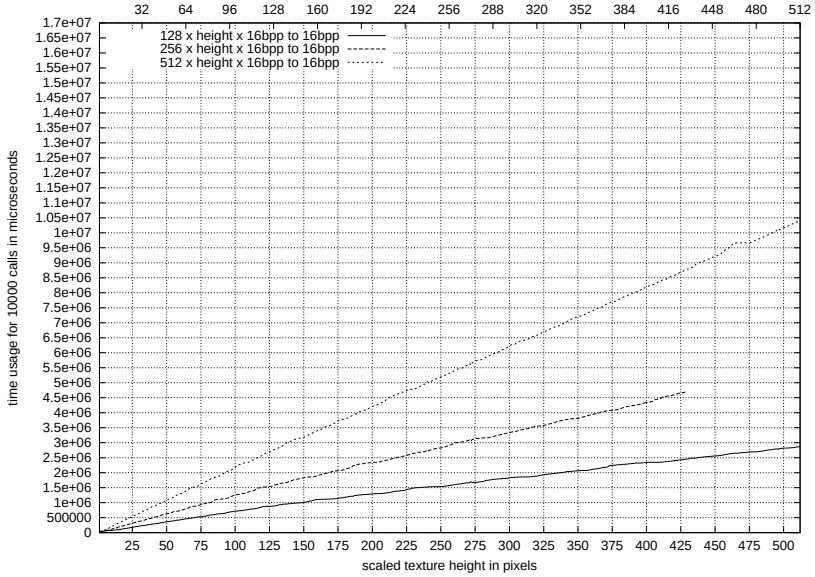

Figure 4.11: Scaled image time consumption based on a 128x128x16 bpp texture (G550), variable height case

indicating much longer drawing times than usual. With the exception of the spikes, the drawing time increases linearly. There is a change in the slope at the beginning, which is better visible in figure 4.15 on page 39. The different curves shown in figure 4.10 are linearly dependent, including the spikes.

Unlike the variable width case, changing the height while keeping the width constant results in linear curves without any spikes as shown in figure 4.11. This explains the linear dependency between the variable width curves above.

Drawing to different pixel depths (e.g. 16 or 32 bpp) results in different drawing speeds, similar to the filled rectangle primitive.

4.3 Models for predicting drawing times

The different benchmarks showed the differences of the drawing speed behaviour, not only between two different graphics cards, but also between different drawing primitives. As a result, there is no general model for all possible combinations without making really pessimistic predictions.

4.3.1 Clipping

Although the graphics cards support hardware based clipping, the tests have shown a significant gain in performance and predictability while using software based clipping. Because it needs a constant time to handle clipping in software it, can be added as an offset to the calculated time usage.

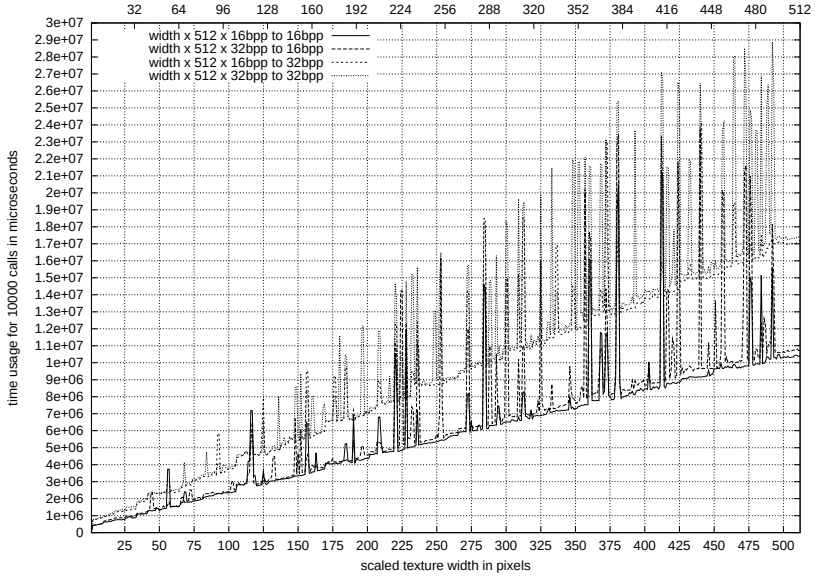

Figure 4.12: Scaled image time consumption based on a 128x128x16 bpp texture (G550), variable width at different depth

card	depth	time in μs
G550	16 bpp	3.9
G550	32 bpp	6.5
Radeon VE	16 bpp	11.6
Radeon VE	32 bpp	20.1

Table 4.2: Approximated drawing time per pixel in different colour-depths; data based on filled rectangle benchmark

Under the mentioned hard- and software conditions, the clipping calculation needs about 0.3 to 0.4 μs for a call. In relation to the time needed for hardware clipping (3 to 4 μs and more per call), and the delay introduced by IPC (9 to 11 μs) this is a rather short time.

4.3.2 Destination colour-depth

As shown with filled rectangles and scaled images, the destination colour-depth plays an important role in the total time used to draw a primitive. The prediction models should be matched to one colour-depth.

Because the drawing speed in different colour-depths is influenced by the hardware architecture of the graphics card, the model for the different colour-depths is specific for each card. Table 4.2 shows some approximated drawing times for different colour-depths.

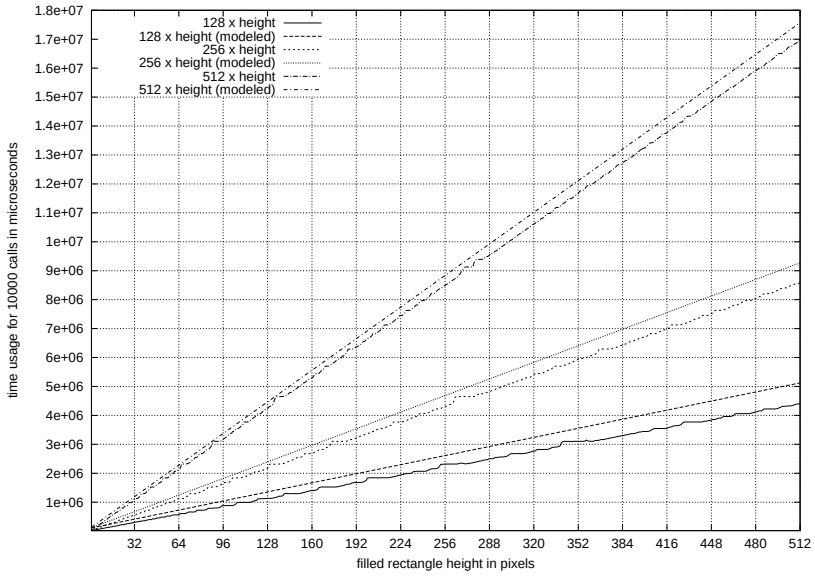

Figure 4.13: Model for G550, filled rectangle at 32 bpp, original and modeled time usage for variable height

4.3.3 Drawing primitives

The figures of the benchmarks show differences between the way primitives are drawn by one graphics card. This ranges from a more or less linear dependency between the amount of pixels to be drawn to a segmented function depending on a couple of parameters with the scaled image primitive.

Filled rectangles

Filled rectangles depend mainly on three parameters: width, height, and colour-depth. Because the clipping time has been reduced to a very small constant, I will not include it in the formulas here. Thus, the resulting function can be denoted as

$$t_{rectangle}(w, h, bpp),$$

where w and h denote the width and the height of the rectangle, respectively.

With a constant width the time usage raises linear, with a constant height it is the same. Thus the resulting function is:

$$t_{rectangle}(w, h, bpp) = a(bpp) + b(bpp) \cdot w + c(bpp) \cdot h + d(bpp) \cdot w \cdot h$$

Based on the benchmark data for different widths and heights the following prediction function has been found for the G550 via trial and error:

$$t_{rectangle}(w, h, 32bpp) = 10\mu s + 6.33ns \cdot (w + 27) \cdot h$$

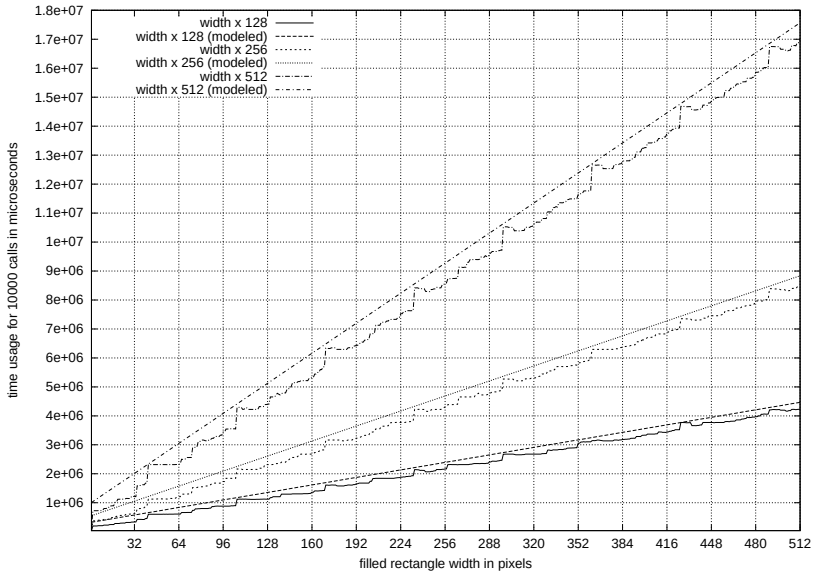

Figure 4.14: Model for G550, filled rectangle at 32 bpp, original and modeled time usage for variable width

The resulting curves are shown together with the data determined by the benchmarks in figure 4.13 and 4.14.

Scaled images (G550 only)

Figure 4.15 shows a variable width case for a 2048x2048 texture. Three different parts of the curve are highlighted. Part 1 and 2 are linear functions, while part 3 has a linear lower bound but a random looking upper bound caused by many spikes. The overall time usage is increasing though.

As already shown in figure 4.11, the dependency in the height is linear, leaving the variable width case as the main influence to the resulting model.

The time prediction model is based on a bilinear function similar to the one for filled rectangles. Because of the visible changes in the slope of the variable width curve this results in a composite function.

A pessimistic approach results in two different bilinear functions. One models the time usage from point A to point B while the other does so from point B to point D:

$$t_{scaledimage}(w,h,bpp) = \begin{cases} A_{width} \leq w \leq B_{width} & : \quad a + b \cdot w + c \cdot h + d \cdot w \cdot h \\ B_{width} < w & : \quad l + m \cdot w + n \cdot h + o \cdot w \cdot h \end{cases}$$

where a to d and l to o depend on bpp similar to the bilinear function shown for filled rectangles.

This model suffices for small texture sizes. Bigger textures like 2048x2048 can be modeled with three different bilinear functions, one from A to B, the second from B to C and the last

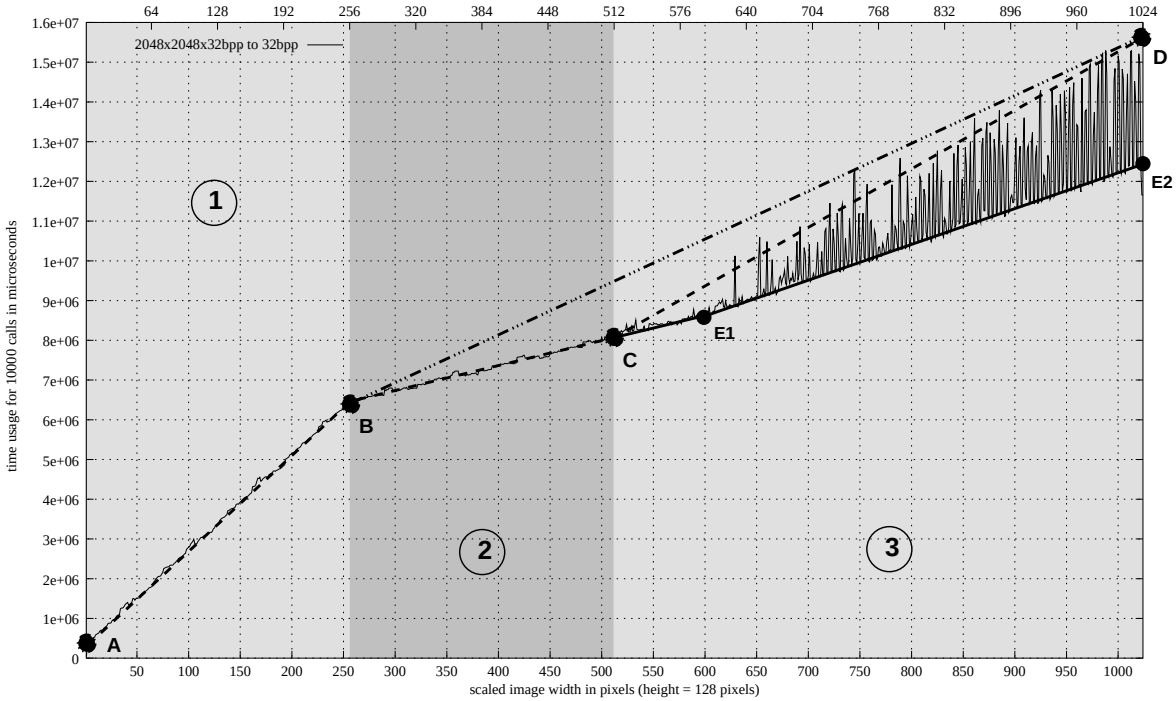

Figure 4.15: Prediction model for G550 scaled images

one from C to D:

$$t_{scaledimage}(w,h,bpp) = \begin{cases} A_{width} \leq w \leq B_{width} & : \quad a + b \cdot w + c \cdot h + d \cdot w \cdot h \\ B_{width} < w \leq C_{width} & : \quad l + m \cdot w + n \cdot h + o \cdot w \cdot h \\ C_{width} < w & : \quad p + q \cdot w + r \cdot h + s \cdot w \cdot h \end{cases}$$

The location of B_{width} and C_{width} can be calculated as follows:

$$\begin{aligned} B_{width} &= \frac{width_{texture} \cdot bpp_{texture}}{32} \\ C_{width} &= 2 \cdot B_{width} \end{aligned}$$

where $bpp_{texture}$ is denoted in bytes per pixel. The G550 can load up to 32 byte at once from its local memory. If the texture is scaled down, not every pixel will be used to display the resulting image. Only one or even no pixel is taken into account if the scaled image width is below or equal to B_{width}. Compared to the case where more than one pixel is contained in a 32 byte block the drawing speed is much lower.

Chapter 5

Conclusions

The primary goal of the presented acceleration architecture is to speed up the drawing primitives of DOpE. This is done by using the hardware accelerated drawing capabilities of graphics cards.

Graphics hardware on the PC is aimed mainly at the entertainment market. It is not built for supporting realtime or multitasking, but for having the best possible average performance. The different benchmarks made this visible.

While the drawing itself has to be done in hardware (this is the purpose of this work), other things are not forced to rely on the graphics hardware. This has been done with the clipping. Tests with hardware based clipping resulted in hard to predict time usage functions. Out of curiousity software clipping has been tested. Surprisingly, it performed much better than hardware clipping. Furthermore, it can be modeled with a simple time prediction model based on a constant time usage.

Determining a model for the actual drawing methods is complicated, because the hardware optimises some tasks, but performs bad in others. At least for both tested graphics cards, it can be assumed the line drawing is based on a linear function, filled rectangles and gradient drawing can be modeled with a bilenear function while drawing scaled images is based on a composite function built up from several bilinear functions.

Calculating specific parameters for the models presented in the last chapter is difficult. To be sure the model is the pessimistic case for all combinations, the whole set of combinations has to be checked against the model. This leads to two conclusions:

- There should be a program which benchmarks a certain primitive for all possible combinations. This is possible to do for the solid lines and filled rectangles. In case of scaled images the number of benchmarks explodes: For every possible texture size (max. texture size for Radeon VE and G550 is 2048x2048) the drawing speed has to be checked for every scaling size and different bit depths of the screen.

- Based on the data collected from the benchmarks and the model given to the program it calculates the parameters and outputs a function similar to the one shown for filled rectangles.

drawing routine	time used
G550	0.27 ms
Radeon VE	0.75 ms
DOpE	5.67 ms
SW-fallback	149.19 ms

Table 5.1: drawing times for one 512x128 filled rectangle at 16 bpp; DOpE without IPC, all others include IPC; the DOpE routine does not support alpha blending

The presented models have to be checked for validity if a driver for not yet supported hardware is written.

An instance of a model is bound to specific hardware. Changing the speed of the graphics processor, the local memory on the graphics card or other components may result in a different instance of a model. A benchmark during startup of the server can initialise the model instances.

The same applies for IPCs. Using a different CPU may result in a different amount of time needed for the communication between client and server.

For the used test system, the IPCs are fast enough to still have a performance gain compared to the old DOpE drawing routines. Table 5.1 shows this. The drawing time for the software fallback supports alpha and works directly on the frame buffer of a graphics card. The alpha-blending requires reading from the frame buffer, which is very slow compared to writing. For example, using software the frame buffer of the G550 can be written with about 45 MB/s while reading only with about 7 MB/s.

The low data transfer rates between system memory and graphics card memory can be improved by using techniques like write-combining, AGP and bus mastering. There are a number of more things, which can be done in the future:

- Currently the VESA driver is based on the software fallback routines. Improving their performance will be useful for all graphics cards without hardware acceleration support.
- For now, only the Matrox G400 and compatible as well as all Radeon 7000 compatible cards have driver support. Adding more chipsets to the supported-list should be one of the long-term goals.
- The server supports multiple graphics cards at once. As a result there are multiple screens available. They can be used independently; for more advanced features like combining two screens to one or displaying part of one screen on another the necessary interface definition is still missing.
- For the time being all drivers have to support the full functionality. Thus, an interface which allows to get information about the capabilities of the graphics hardware is required. This allows DOpE and other applications to enable or disable optional features.
- DOpE needs to be changed so it accesses different models depending on the used hardware and drawing primitive.

- Besides the above mentioned performance optimisations there are more possible:
  - Lets assume the system has two graphics cards installed. If one of these cards is busy but the incoming request is directed at the idle card, this request can be executed even while the other card is still busy. This holds true unless both cards paint to the same memory area (AGP/PCI memory).
  - If a card does not support all primitives and the missing functionality has to be covered by software rendering, soft- and hardware rendering can work independently as long as they do not paint to the same memory area.
  - If clients are not allowed to access other clients graphics data, requests from different clients can be handled in parallel.
  - The communication between server and client can be optimised by using DSI. Collecting drawing tasks on a stack and sending the whole stack at once can be combined with DSI.
- There already have been questions about the 3D support. It requires a rewrite of the server architecture and needs much more complex drivers. The first prototype could implement an OpenGL/ES conform interface, which can be improved step by step to a full featured OpenGL interface later on.

Glossary

AGP Accelerated Graphics Port. As the name suggests, it was created to speed up the data transfer between a host system and the graphics card in comparision to the previously used PCI bus.

API Application Programmers Interface. The software interface offered by frameworks, libraries and so on.

bitblit Operation which moves a rectangular graphics area in memory. Graphics hardware supports different kinds of bitblits, e.g. bitblit with transparency or bitblits with colour expansion, which is used to draw non-antialiased characters. Additionally, the source and the destination can be combined with a raster operation.

bus mastering On data buses like PCI and AGP one device is the master, initiating bus transfers and regulating the traffic. In case of software based rendering, the CPU is the master. But PCI and AGP devices can be programmed to take the bus master role. While the devices manage the data transfer, the CPU is free to do other stuff.

DOpE Desktop Operating Environment. It is a graphical user interface with realtime support. It is part of DROPS. See [1].

DMA Direct Memory Access. See *bus mastering*.

DRI Direct Rendering Infrastructure. Part of XFree86 4.X.

DROPS Dresden Realtime Operating System. See [3].

DSI Direct Streaming Interface. To speed up the interprocess-communication, DSI manages a shared memory area. This memory area is used to store the data transfered between two tasks. This saves the time which is normally spent by copying this data during the IPC-call.

frame buffer Local memory on the graphics card. It is mapped into the address space of the system, so the CPU and other PCI devices can access this memory. The data transfer speed to this memory is limited by the bus the graphics card is connected to. See AGP.

GLX The "glue" between OpenGL and X so one can execute OpenGL applications remotely and still run them accelerated.

GPU Graphical Processing Unit. Based on the term CPU. It is the core of most graphics cards and can perform graphical computations.

IPC Interprocess Communication.

IRQ Interrupt request. This is a way for hardware to notify the CPU that an event occured.

MMIO Memory Mapping Input Output. Hardware like PCI and AGP cards map their programmable registers into the memory space of the CPU, so they can be accessed via normal memory write and read operations.

OpenGL A specification for a 3D API. See `http://www.opengl.org/` for further information.

OpenGL/ES OpenGL for embedded systems. This specification defines a subset of the OpenGL specification, so it can be used in embedded systems. See `http://www.khronos.org/` for more information.

PCI Peripheral Component Interconnect. PCI is a bus for local perihperals in a computer which is widely used in current PC hardware. It was developed as a replacement for the now outdated ISA bus.

VESA Video Electronics Standards Association. See `http://www.vesa.org/`.

VGA Video Graphics Array. It was developed by IBM and supports 640x480 with 16 colours and 320x200 with 256 colours. It has no builtin hardware acceleration features.

write combining Instead of initiating one data cycle for every access to a PCI or AGP card, up to 32 subsequent bytes get cached in the CPU and sent in one burst. The overhead for preparing a data cycle on the PCI or AGP bus is reduced to one time per burst, instead of up to 32 times if single bytes are transfered. Write combining has to be enabled for a certain memory area by configuring the MTT registers accordingly.

XAA XFree86 Acceleration Architecture. Part of XFree86 (see `http://www.xfree86.org/`. The documentation for XAA can be found in the XFree86 sources.

Bibliography

[1] Norman Feske. *DOpE – a graphical user interface for DROPS.* diploma thesis, Dresden University of Technology, Operating Systems Group, 17.10.2002

[2] `http://os.inf.tu-dresden.de/ nf2/dopescreens.html`
webpage containing several screenshots of DOpE

[3] `http://os.inf.tu-dresden.de/drops/`
homepage of the DROPS project

[4] `http://www.directfb.org/`
Homepage of DirectFB

[5] `http://dri.sourceforge.net/doc/dri_control_flow.html`
DRI control flow diagram

[6] `http://dri.sourceforge.net/doc/DRIintro.html`
Introduction to DRI

[7] `http://utah-glx.sourceforge.net/`
Utah-GLX homepage

[8] `http://www.khronos.org/`
OpenGL/ES homepage

Erklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 31. August 2003