DEGREE PROJECT 2003:D14

Avdelningen för datavetenskap vid Institutionen för Informatik och Matematik

On the Principles and Future of COM

Featuring

The Random Dot Stereoimage Technology

Anders Alexandersson

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

EXAMENSARBETE

Högskolan Trollhättan · Uddevalla
Institutionen för Informatik och Matematik

Uppsats för filosofie kandidat i Datavetenskap

On the Principles and Future of COM

featuring

The Random Dot Stereoimage Technology

Anders Alexandersson

Examinator:
Docent Stefan Mankefors Institutionen för Informatik och Matematik

Handledare:
Andreas Boklund Institutionen för Informatik och Matematik

Trollhättan, 2003

2003:D14

EXAMENSARBETE

COM — principer och framtida utveckling exemplifierat med RDS teknologin

Anders Alexandersson

Sammanfattning

Föreliggande arbete är en tillämpning av den komponentbaserade principen för programvaruutveckling, med hjälp av COM - Component Object Model från Microsoft. Inledningsvis finns en analys och beskrivning av de grundläggande principerna för COM, varefter dokumentationen av en komplett implementerad Windowsapplikation finns. Applikationen är utvecklad i C++ och bygger på och exemplifierar den inledande COM teorin, och tar samtidigt principerna för Software Engineering i beaktande.

Specifikt är applikationen en RDS (Random Dot Stereoimage) generator, som med hjälp av matematik i tre dimensioner och DirectX skapar äkta tredimensionella bilder på skärmen, vilket är möjligt tack vare människans stereoseende.

Resultatet är en syntes av teori och praktik till en sammanhängande bild av den nutida och framtida komponentbaserade principen för programvaruutveckling, och dess för- och nackdelar.

Utgivare:	Högskolan Trollhättan · Uddevalla, Institutionen för Informatik och Matematik Box 957, 461 29 Trollhättan Tel: 0520-47 50 00 Fax: 0520-47 50 99		
Examinator:	Docent Stefan Mankefors		
Handledare:	Andreas Boklund, HTU		
Huvudämne:	Datavetenskap	**Språk**:	Engelska
Nivå:	Fördjupningsnivå 1	**Poäng**:	10
Rapportnr:	2003:D14	**Datum**:	2003-05-19
Nyckelord:	COM, C++, komponent, DirectX, RDS, Windows, programmering		

DEGREE PROJECT

On the Principles and Future of COM

featuring

The Random Dot Stereoimage Technology

Anders Alexandersson

Summary

This work is an application to the component-based principle of software development, using COM - Component Object Model from Microsoft. Initially there is an analysis and description of the basic principles of COM, where after the documentation of a complete Windows application is presented. The application is developed in C++ and is based on and an example of the initial COM theory, at the same time applying the overall principles of Software Engineering.

Specifically, the application is an RDS (Random Dot Stereoimage) generator, which with help of mathematics in three dimensions and DirectX creates true three-dimensional images on the screen, which is possible due to the stereo vision of Man.

The result is a synthesis of theory and practise into a coherent picture of the present and future principles of component-based software development, and its pros and cons.

Publisher:	University of Trollhättan · Uddevalla, Department of Informatics and Mathematics Box 957, S-461 29 Trollhättan, SWEDEN Phone: + 46 520 47 50 00 Fax: + 46 520 47 50 99		
Examiner:	Docent Stefan Mankefors		
Advisor:	Andreas Boklund, HTU		
Subject:	Computer science	**Language**:	English
Number:	2003:D14	**Date**:	May 19, 2003
Keywords	COM, C++, component, DirectX, RDS, Windows, programming		

Preface

I want to thank my spouse for putting up with me during the creation of this dissertation, fixed the many hours in front of the screen, hardly able to get into contact with by any means.

Also I want to thank my colleagues on the DS programme for valuable points of view.

Contents

Appendix

List of symbols

RDS	Random Dot Stereoimage. A technology for creating 3D images based on 3D geometry and random points on a surface.
COM	Component Object Model - a technology by Microsoft implementing the component principle.
.NET	The successor to COM, but is also a whole platform of software development, and thus more complex a concept.
IDL	Interface definition language - a flexible way of defining interfaces, not language dependent.
DLL	Dynamic link library - code accessed dynamically at execution time of an application, residing outside the EXE itself.
Wrapper function	A function that encapsulates other functions.
DirectX	A Microsoft technology for abstracting video, sound, networking etc.
VRAM	Very fast memory on the video card in a computer, holding the images displayed on the screen.
GUI	Graphical User Interface, a graphical area where user and software meet.

1 Introduction

1.1 Background

In the dawn of the computer age, i.e. half a century ago, computer programs were very small pieces of data, that perhaps calculated some trivial mathematical routine work. As the years have gone by, the programs have become larger and larger, and today reach the multimillion-line size, if not billions of lines of code. At the same time the number of different hardware versions have increased that these applications shall reside and work on. This raises some interesting questions.

1. How should a developer or team of developers be able to handle the massive amounts of code without chaos ensuing?
2. How can you as an application developer, create new versions of your programme without having to send out billions of lines of code when you have only changed a few of them?
3. How can you as a programmer write one single programme that will work on all hardware imaginable?
4. When developing applications in this scale, the same principles are used many times, but why invent the wheel over and over again? Isn't there some way to reuse high quality, well tested code in a good way?

One answer to these questions is COM - the Component Object Model.[1] In chapter 4 we will dwell upon these questions in general, after which we in chapter 5 we will study question number three above in particular, where a fully implemented example is presented in the form of a Random Dot Stereoimage application, presenting in detail how COM is used to answer that question.

1.2 Purpose

The purpose of this work is to present an analysis of the power of the COM paradigm in general and how it can answer the questions above, and specifically how it can be used to get control over the graphical display on the Windows platform, by a complete Windows application example using DirectX [2] - the RDS generator.

1.3 Limitations

I will only study the implementation of COM principles on the Windows platform. There are other suggestions from other companies, to implement the component principle. Examples are Java, and Enterprise Java Beans [3] from Sun Microsystems and XPCOM from Mozilla, [4] which will not be studied.

2 Background of this work

Of all problems that can be solved with COM (see chapter 1), I have, apart from the general analyses in chapter 4, chosen to specifically focus on and study the problem of creating a low-level graphical application, that needs full control over the screen, as needed when e.g. making a simulation, visual effects or the like, due to e.g. performance reasons. The term "low-level" implies that you as a programmer need to directly access the VRAM of the video card, which we will see in detail in chapter 5.

The problem here is that, as mentioned earlier, if you as a programmer need full control over the screen, you are faced with the problem of thousands of graphical devices on the market – each with different configuration. Some have hardware functionality that increases performance, some have not etc. How it is possible to write one single programme that will work on all different video cards?

One answer to this question is the use of the Java technology, which as said will not be discussed in this dissertation. I have instead chosen to study Microsoft's solution to this problem, which is called *DirectX.* We will dwell on the details of this technology in chapter 4.5. As for now we will only say that it is *a complete system of software that abstracts video, audio, input, networking, installation and more.* Moreover it is optimised with regard to performance and robustness compared to the Windows native GDI and/or MCI technologies.[2]

3 Method

3.1 Software engineering principles.

In the development of the application, I have followed the principles recommended by Sommerville [5], where he states that the development of a computer application should follow the same procedures as the development of any other physical engineering.

3.2 Processes

3.2.1 Development processes

According to Sommerville [5] the quality of the product is a result of the quality of the processes involved when creating the product. Therefore a set of processes were defined that should be followed creating the product, that would ensure good quality:

1. Do not write anything sure to be removed later, as temporary comments, or easy, but insecure solutions. Do it right the first time.
2. Always copy-paste code instead of writing manually, to avoid mistyping, and the following difficulty in finding errors.
3. Write keyword "DEBUG" on places that should be removed after all testing is through, to be guaranteed to be found and tested.

4. Never state anything that isn't logically founded, or secured via external sources.
5. Do not use unnecessary complex constructs as settings in separate files etc. since the more complex the structure, the more error prone it will be.
6. Write a log on everything done, to be able to go back in history and find causes of eventual errors, and to avoid inventing the same solution twice.
7. Test a change immediately to have control over cause and effect. If two or more changes have been made, there is no way of knowing which one is the cause of eventual problems.
8. Never hard-code any constants, that need to be changed later, or constants used more than once, to avoid updates of multiple places in the code. Define all constants in a file called *constants.h*, and include it.
9. Follow the system model. No quick solutions.

3.2.2 The SEI Process Capability Maturity Model

I have aimed as much as possible to follow step 2 in the SEI Process Capability Maturity Model [5] - repeatable - in which the following qualifications that can be applied to this work, should be met:

1. Software configuration management.
   Keep track of new versions of the programme, and don't mix them.
2. Software quality assurance.
   Define processes to be followed to guarantee quality.
3. Software project tracking and oversight.
   Create log of all events, problems and their solutions for later reference.
4. Software project planning.
   Use and follow Gantt plan.

3.3 Development model

In the process of developing the application I have created and followed this model, with the analyses by Sommerville [5] concerning software engineering, in mind:

1. Create an overall system architecture, consisting of independent components with single interfaces.
2. Then implement the interfaces of the model by means of prototyping, focusing on the robustness of the interfaces.
3. Last, implement the details inside the components to form a complete application.

3.4 Discussion of the development model

The term *component* is in this context not following the Microsoft COM specification, but only describes conceptually the structure of the model. No specification other than the pure C++ object-oriented principles will be used. The interfaces between the component will be public methods of C++ classes.

This approach has proven to be successful in my previous experience [6], since it guarantees a robust structure, where as little dependence as possible is needed between different areas of the application, and is also recommended by Sommerville [5].

An interesting future project would be to implement the full COM specification, or even the later .NET principle [7] (see chapter 6), with this model, making the application even more flexible, but at this time no such initiative will be made.

3.5 Validation and testing

As to the validation, the application simply serves as an example of an implementation of COM principles. The term *validation* defines that one is building the right product [5] and due to the fact that the application is an example of COM principles, the validation is trivially secured in this case.

The testing of the application can be divided into two steps:

1) Does the RDS algorithm work at all?
2) Once step 1 is OK, does the application work on different configurations of the Windows platform?

Step 1 in the testing process is self-verifying simply because a 3D image will appear on the screen, if the algorithm works.

Step 2 will be verified by testing the application on different Windows configurations and hardware (see chapter 5.6.6).

An overall systematic approach as to the identification of local testing and errors in the code, will be the use of test outputs of values in text files, where cause and effect of errors can be recognized.

3.6 Programming language

The programming language used in developing the application will be C++, to get as much control as possible over the processes of COM. Other languages supporting the COM principle is e.g. Visual Basic, which offers less control of all details.

3.7 COM theory

The theoretical part of this dissertation, where COM theory is analysed and presented is based on literature studies and studies of web resources, where the theory will be broken down into it's basic parts and the connections between those parts clarified, where after conclusions be draw as to the structure of the COM technology.

4 COM fundamentals

4.1 History of software principles

One definition of a computer programme is, that it is an abstract machine that perform algorithmic tasks [8], and such devices can be traced in history all the way back to the Greek and Romans civilizations, although then in mechanical forms. Later in history, around the beginning of the 1900th century – via Pascal, Leibniz and Babbage – the first machine to be controlled using holes in paper cards emerged, and Augusta Ada Byron is today identified as the world's first programmer. In 1890 Hollerith applied this technique to speed up tabulation processes in the U.S. census, a work which actually later led to the creation of IBM.

Not until around 1940 the first electronic devices emerged, one of the first being called Mark I 1944 at Harvard University by Aiken. Other names are the Atanasoff-Berry machine, COLOSSUS and ENIAC. Further development used the transistor and integrated circuits and the computer technology of today saw the light.

In the early abstract machines, the time consuming process of programming the mechanical devices restricted the complexity of the algorithms used. However, as these limitations disappeared the machines were assigned more and more complex tasks.

4.2 Software paradigms

As seen on this brief glance at history, the early programmes were batch processes, that started at the beginning and ended at the end in a linear fashion, let it be a mechanical device or a hole card.

The next step in programming development was the "procedural programming" paradigm, used by e.g. C – developed 1972 by Dennis Ritchie at Bell laboratories [9] – where structures as the "function" is used, being as a black box that made something for you. Programmes were, then, made up by functions that executed independent tasks, and returned control to the main programme when finished.

In the early 1980s the next paradigm in software emerged, namely the object-oriented paradigm, as Bjarne Stroustrup developed C++ [10] – he too at Bell laboratories. The object-oriented approach tries to model real world items into software objects, to create the same conceptual base as in the real world. The objects then interact with each other – as in the real world.

Last, the concept of a *component*, then, is a set of objects that together perform a specific task and is completely independent. The component is accessed via interfaces, the details of which is the subject of the subsequent chapters.

4.3 The component principle

What is a component? The generic and vague notion of a component is an independent chunk of something with some sort of functionality. The simplest example of this is the Lego block. Lego blocks have different sizes and shapes, but they all share the same basic principle, that they can be put together, into almost any shape imaginable. Legoland in Denmark is a good example, where huge creations of all shapes can be found – houses, people, boats etc – all made of a few fundamental Lego components. This illustrates the strength of the component principle, and in the next chapters analyses as to what is needed to simulate this behaviour in a software environment will be made. The laws of assembling Lego-blocks and compiling a computer programme are quite different on a low, detailed level.

4.3.1 Why components?

Why do software engineers want to simulate the behaviour of Legoblocks any way? There are a number of factors motivating the use of the component principle, but the main one is that the development of software is an expensive and time-consuming venture. The ideal situation is expressed like this by a developer at ESRI:

> "In an ideal world, it should be possible to write a piece of code once and then reuse it again and again using a variety of development tools, even in circumstances that the original developer did not foresee. Ideally, changes to the code's functionality made by the original developer could be deployed without requiring existing users to change or recompile their code." [11]

This calls for an elegant software structure as the ideal, where applications are composed by assembling independent components, as in Legoland, that can be replaced or updated simply by plugging out the old and plugging in the new one – without having to change lots of other things.

4.3.2 Early component technologies

As said earlier, the object-oriented paradigm laid the foundation for thinking in terms of independent objects/components that interact with each other. Early attempts were made to create collections of reusable chunks of software, by assembling generic objects – usually developed in C++ – into libraries that could be used by any developer. These were called *class libraries.*[11]

These early tries suffered from some severe problems [11], notably:

1. Problems sharing parts of the system.
   It is very hard to share binary C++ components. Most attempts have only shared source code.
2. Problems with the placement of the components on disk – in object technology called persistence – and updating of components without the need for recompilation.

3. Lack of modelling languages and tools, some also being proprietary.

4.3.3 Analysis of what is needed for true component functionality

At this stage some conclusions can be drawn as to what is needed to generate true component behaviour in a software environment.

1. Components must be independent.
   They must carry all functionality within themselves if they are to be plugged in and out of systems.
2. Components must be used in binary form.
   If they were not, changes to the component interior would force the main programme using the component to be recompiled. If not recompiled, the main programme would not know anything had changed, like changing a blueprint without telling anyone about it.
3. We need some kind of protocol – rules and regulations – which governs how these binary units are handled.

The last point is what COM does.

4.4 The Microsoft Component Object Model – COM

4.4.1 History of COM

The evolution of COM is a bit confusing – due to Microsoft's strange naming conventions. Here follows the history.

One of the first to ever mention the idea of what later became the COM architecture of today, was Anthony Williams in his papers *Object Architecture: Dealing With the Unknown - or - Type Safety in a Dynamically Extensible Class* 1988, and *On Inheritance: What It Means and How To Use It* 1990. [1]

One year later, Microsoft created the *object linking and embedding technology (OLE)* for compound documents, which in turn built on *dynamic data exchange (DDE)* and the *VBX (Visual Basic Extension)* controls from VB 1.0.[1]

In 1993, OLE 2 was released as a successor to OLE 1, and in 1994 OCX or OLE controls as successor to VBX control, at the same time saying that OLE was no longer an acronym, but an overall name for all of Microsoft's component technologies.[1]

In 1996 they renamed some parts related to the Internet of OLE into ActiveX, and then all of the previous OLE as well, i.e. all of Microsoft's component technologies were now called ActiveX, and OLE was degraded to containing the original principle of compound document technology, as in Word, Excel etc. Later the same year DCOM was created as a response to CORBA.[1]

In September 1997, the complete component technologies were once again renamed, this time into COM, and the word ActiveX was diminished and not spoken of very much.[1]

Then, when Windows 2000 was introduced, COM was again renamed into COM+ and DCOM dropped conceptually, as COM+ included the same distributed functionality, i.e. one could call an object residing on a different machine, earlier only possible with DCOM.[1]

Today the complete COM technology is stated to be replaced by the .NET initiative and all other previous frameworks abandoned, but they now coexist during a period of time, DirectX being e.g. COM based as we will see in later chapters. [1]

Thus it is a bit confusing when relating to Microsoft's COM technology, but the term COM will be used to cover the basic principle of the Microsoft component concept.

4.4.2 COM definition

First of all, it is important to realise that COM isn't a programming language [2]. It is a protocol or standard much like the idea of the protocols in networking. It is an agreement as to how things should be done. In networking the protocols define how e.g. a web page should be treated and transferred, and in COM the protocol defines how to connect one software chunk with another. If everyone uses this protocol, or standard, these chunks of code can be looked upon like Lego blocks that are easily glued together to form the application.

4.4.3 The concept of the interface

The first and most fundamental concept of COM is the *interface.*[2] An interface is simply a *set of functions.* If we use a metaphor for a component - an independent chunk of code that does something - we can see it as a solid box. We don't know what is inside, only what it does. Then it would be of little use if there were no entry points, where input could be made and information retrieved. The interface is just that: an entry point for input and output, to and from the component.

Now, the COM protocol states that all components shall have the same basic interface to begin with, i.e. a set of some common core functions.[2] These functions are defined in an interface called *IUnknown.* The core functions are named as follows:

- QueryInterface()
- AddRef()
- Release()

QueryInterface is used to retrieve other interfaces of the component, AddRef() to increment interface reference count and Release() to decrement interface reference count. The interface reference count is simply an integer keeping track of how many

clients are holding a pointer to that particular interface of a particular COM object. When the interface reference count reaches 0 the COM object destroys itself. This is the fundamental platform from which all COM components are built.

4.4.4 Building an interface

As an example of the creation of an interface, a component that handles graphics and sound for multimedia applications will be studied. We want to have interfaces that can be used to make the component do work. Let's call our first interface *IGraphics* that is the entry point for doing graphics on screen. The COM specification states, as said, that all interfaces shall be base on *IUnknown*, so in order to create the *IGraphics* interface it has to inherit the core functions from *IUnkown,* which in C++ can be made like this, using IDL,

```
Interface IGraphics : IUnknown
{
     virtual int InitGraphics( int mode )  = 0;
}
```

which means that our new interface has four functions: QueryInterface(), AddRef(), Release() and our newly declared InitGraphics(). Note the use of the pure virtual function declaration (declaration assigned 0).

Now we have a graphical interface to our component. We don't know yet how the actual graphical implementation is done, but that is a later problem. For now, we only want to define the entry points to the object. Now let's declare an interface for doing sound, using the same principle as with *IGraphics:*

```
Interface ISound : IUnknown
{
     virtual int InitSound( int driver )   = 0;
}
```

We have now created two interfaces for a component that knows how to handle graphics and sound, by following the COM specification that all interfaces shall be derived from *IUnknown*.

4.4.5 Creating the component itself

Now after we have the interfaces in place, let's create the actual component, called EASY_MULTIMEDIA, like this:

```
class EASY_MULTIMEDIA : public IGraphics, public ISound
{
     public:
```

```
        //Constructor
        //Destructor
    private:
        //Declaration of QueryInterface()
        //Declaration of AddRef()
        //Declaration of Release()
        //Declaration of InitGraphics()
        //Declaration of InitSound()
        //Definition of the reference counter
};

//Implementation of above declared functions
//…
```

We now have a complete component. Note the technique of multiple inheritance of both the IGraphics and the ISound interface to generate a component with two interfaces.

Before we have a look at how our component could be used by another programmer anywhere in the world, just a few words about GUIDs and IIDs.

4.4.6 Global Unique Identifiers (GUIDs) and Interface IDs (IIDs).

The COM specification states [2] that every component and interface thereof must have a unique identifier, i.e. something that makes every interface and component unique in the world. Why is that? Well the idea of the component principle is, as mentioned before, that the component shall be able to be plugged in and out of systems without anything else needs to be changed. But, the programmer that uses the component must have something to use as a "hook" to retrieve an interface on that component. That is what GUIDs and IIDs do. The term IID is used when interfaces are concerned, but it is the same principle. A GUID or IID is a 128 bit vector that makes possible $2^{128} \approx 3.4 \times 10^{38}$ unique ID-numbers which will suffice for a very long time.

The GUID or IID is the unique name of a component or interface, and is the only thing needed to retrieve it and begin doing work with it. The GUIDs and IID are generated by a tool provided by Microsoft called GUIDGEN.EXE, which guarantee that not two generated IDs are the same based on math and probability theory.[2]

An example of a GUID pasted right from the GUIDGEN programme looks like this:

```
// {91EBF1A5-7EF7-11d7-920B-0010A704BFB4}
static const GUID IID_GRAPHIC =
{ 0x91ebf1a5, 0x7ef7, 0x11d7, { 0x92, 0xb, 0x0, 0x10, 0xa7, 0x4, 0xbf,
0xb4 } };
```

The only thing changed is the identifier of the constant name itself, IID_GRAPHIC.

Here is an example in pseudo code of how the GUID can be used inside an implementation of QueryInterface() to return the correct interface:

```
//…
if( requestedInterface == IID_GRAPHIC )
     //return the graphics interface
else
     //return the sound interface
```

The GUIDs are simply compared in an ordinary if-statement.

4.4.7 How to use a component

A programmer would use the following method to access the component and its interfaces, beginning at e.g. *main()* in a C++ console application:

```
//Includes of definition of IUnknown provided by Microsoft
#include unknwn.h

void main(void)
{
     //create the component, often referred to as COM object
     IUnknown *punknown = createComponent();

     //Create pointers to the interfaces that will be used
     IGraphics *pigraphics;
     ISound    *pisound;

     //Query an interface from the base object IUnknown.
     //Note the use of the IID for our graphics interface, which we
     //have provided together with the publication of the component
     //and called IID_GRAPHICS
     punknown->QueryInterface( IID_GRAPHICS, (void **)&pigraphics );

     //pigraphics now point to our graphics interface
     //Init graphics, mode 0 which could be e.g. 1024x768 in resolution
     pigraphics->InitGraphics( 0 );

     //Now query for the sound interface, same principles as above
     punknown->QueryInterface( IID_SOUND, (void **)&pisound );

     //pisound now point to our sound interface
     //Init sound, driver 7, could be 16 bit, stereo
     pisound->InitSound(7);
```

```
    //----- Local code here ------

    //Shutdown after finished program by releasing all interfaces
    pigraphics->release();
    pisound->release();
    punknown->release();
} //End main
```

What is missing in this example is that a complete COM object is compiled as a DLL and is linked in by the application at run-time. As we will see in the next chapters, a programmer doesn't have to worry about this, as the COM specification takes care of these details automatically, using some help functions provided by Microsoft. For the complete picture, here are the steps anyway, that are involved in extracting a COM interface from within a DLL residing on the hard disk [11]:

1. A programmer requests a service of a COM object.
2. The SCM (COM Service Control) looks for the object by searching the Windows registry for the GUID
3. The related DLL is loaded into memory, and a function therein called *DllGetClassObject()* is called, which passes the desired class as the first argument. This is the function that makes a DLL a COM DLL.
4. The SCM calls a function called *CreateInstance* to instantiate the object properly.
5. Finally the desired interface is requested and returned to the programmer and the SCM drops out of the picture, allowing the programmer to talk directly to the COM interface.

4.4.8 Discussion of the principles of COM

We said that COM is a protocol that governs how pieces of software are connected. By the use of GUIDs and IIDs we can access any component in the world and then extract interfaces from them, who contain a set of functions that can do the actual work. This sounds quite complex, but if we analyse the code above, we can draw some interesting conclusions:

First of all the code is native C++. No extras there, which means that what was said earlier, that COM is not a programming language, is correct.

- Conclusion 1: COM doesn't add anything to or demand anything extra from the programming language in which the component is used, proving that COM is simply a protocol that controls how things are done.

Second, the first thing to do when working with COM is to create an object of the class IUnknown, and then query it for new interfaces using IIDs. But the IUnknown is, as we have seen, only a C++ class, let it be abstract and pure virual, but nonetheless nothing but a C++ class. And the QueryInterface() function merely returns a pointer to a class of the type specified by the declaration of that pointer, e.g. IGraphics above.

- Conclusion 2: The starting point in COM, *IUnknown,* is merely a pointer to a set of tables of pointers to functions that do the work.

So, to stripped of all high levels concepts — components, interfaces, GUIDS and IIDs — COM is only a set of rules that controls how to get a pointer to a set of table of pointers to the functionality that someone has written.

4.5 DirectX

4.5.1 Introduction

As said in previous chapters, it is a big problem to write a low-level application due to the fact that there are thousands of manufacturers on the market, each having their own way of doing things. If a programmer should write a programme that worked on all those devices he or she would have to write as many routines as there are devices on the market. That is a total impossibility, it would take thousands of man-years.

Thus something else is needed. The programmer would need to abstract the hardware details, using the same routines and something else should take care of the low-level details. That is exactly what DirectX does.[2] It is possible because of a major engineering effort by Microsoft and all hardware vendors, where Microsoft defined a set of conventions all vendors had to follow when implementing the drivers for their hardware. As long as those standards are followed everything works. A standard call can be made to DirectX, e.g. to plot a pixel on screen, and DirectX knows what device is installed and how to speak directly to it. Thus DirectX has a database with drivers for all devices – video cards, sound cards, input devices, network cards, etc. – that are part of the standard and knows how to speak to it. It works as a multilingual hardware interpreter.

4.5.2 DirectX and COM

How does this relate to COM? Well, DirectX is build up of a number of COM objects. These objects are contained in the system as DLLs after DirectX is installed, following standard Windows procedures. When a programme needs work to be done by DirectX,

the procedure described previously is executed, loading the COM object into memory and retrieving the right interface, which is used to get the work done. [2]

The compile-side of things is a bit more complex, though. As said earlier, the only thing a programmer needs in order to be able to use a component is the GUID or IID. That is also true, but the process of loading DLLs, instantiating COM objects, initialising them etc. requires quite a lot of tedious low-level work as it is dependent on platform. Note that the COM specification is generic and can be used on any platform. [1] Thus, to ease the burden of the programmer, Microsoft provides a set of *wrapper functions* for the Windows platform, that does the tedious work. These have to be included, as we will see in detail in chapter 5, where DirectX is used to access the screen. But, note that these extra library-files do not contain any COM objects, only the help functions that load DLLs etc., so the COM nature of "plug 'n' play" is still preserved.

DirectX contains features that control many things, as 2D graphics, 3D graphics, sound and music, input devices, networking etc., providing hardware acceleration if that specific card supports it. The focus of this work is, as stated earlier, to study the details of graphics and how to create a low-level graphical application, which we will dwell on in chapter 5, so let's take a closer look at the interfaces of DirectX that is responsible for controlling graphics – DirectDraw.

4.5.3 DirectDraw

I have worked with version 6 of DirectX, since the literature I have acquired is based on this version. DirectDraw now exist in version 9, but the basics are the same. New versions of DirectX support more complex concepts, but the old interfaces must remain to guarantee compability [2], and the task of chapter 5 is only to retrieve control over the screen and plot pixels on it. Thus I have chosen to line up the basic principles of graphics based on DirextX 6, which are still present in newer versions.

DirectDraw consists of five interfaces: IUnknown, IDirectDraw, IDirectDrawSurface, IDirectDrawPalette, and IDirectDrawClipper. IUnknown is the same for all COM objects, as described earlier. Note the naming conventions of interfaces with an initial ‘I’, short for ‘Interface’.

IDirectDraw is the main interface that must be created to begin working with DirectDraw and basically represents the video card in the computer. If multiple video cards are installed in the machine, multiple DirectDraw interfaces can be created, each representing one video card. [2]

IDirectDrawSurface is an abstraction of the actual image one wishes to create and view on screen, like a canvas to paint on. There are two kinds of surfaces: primary and secondary. The primary surface usually represents the actual videobuffer being rasterized and displayed by the video card on screen. Secondary surfaces are usually

offscreen and are used to create the next image in an animation when the first is being displayed, a technique called *back buffering,* used to avoid flickering. [2]

IDirectDrawPalette is used when 256 or fewer colors are used, and represents a 256 color palette containing the colors one wishes to use from the millions possible. DirectDraw is equipped to deal with any color space from 1- bit monochrome to 32 - bit Ultra True color.[2]

IDirectDrawClipper helps when clipping rendering outside the valid area on screen, i.e. the window representing the application. The process of analysing if a pixel will be put outside the application window is an expensive process in terms of performance, and the IDirectDrawClipper supports hardware acceleration to do this work.[2]

Once these interfaces are created the following routine is used for accessing the screen: [2]

1. Create the main DirectDraw object and retrieve the latest interface.
2. Create at least a primary surface to draw on using IDirectDrawSurface. If the color mode is 8-bit or less an IDirectDrawPalette is needed.
3. Create a palette using the IDirectDrawPalette interface, fill it with the colors wanted and attach it to the surface.
4. If the application is windowed, create an IDirectDrawClipper that clips everything outside the application window.
5. Draw on the primary surface. The drawn image will be instantaneously visible on screen, as the primary surface resides in the VRAM of the video card.

That is the basics of DirectDraw. Now, let’s get into the details of it with a complete working Windows application, creating random dot stereoimages (RDSs) by plotting pixels on the screen using DirectX and COM.

5 The RDS Generator using COM

5.1 Introduction

I have chosen to illustrate the use of DirectX and COM with the RDS technology. First of all let's have a look at the basic principles of RDS [12] in fig. 5.1:

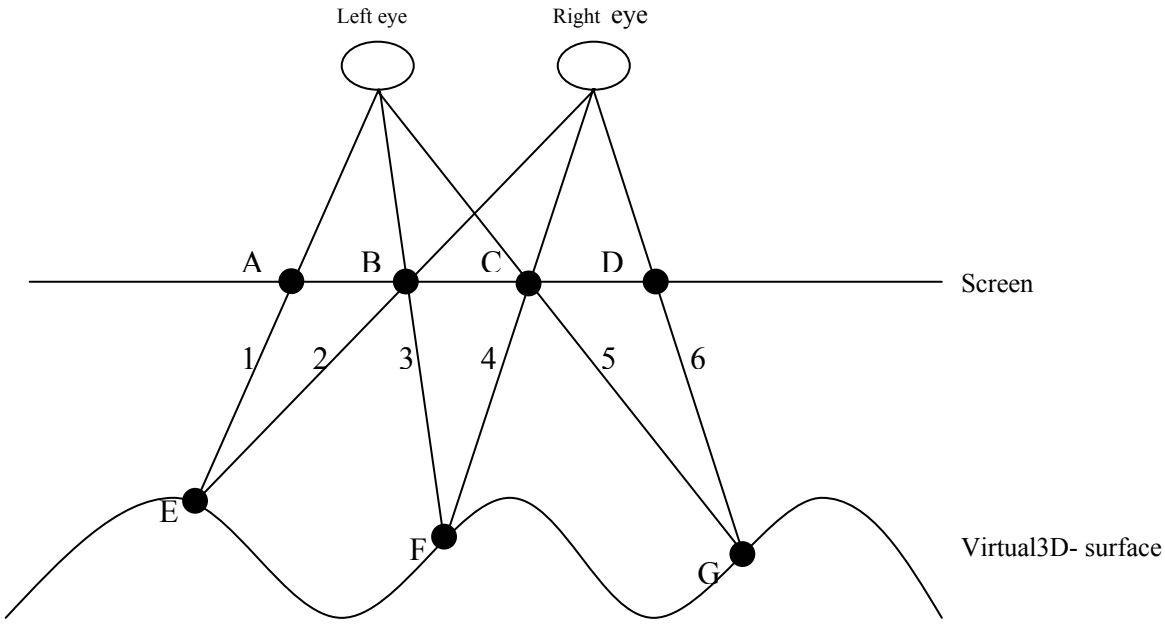


Fig. 5.1

The principle is quite simple. If the person focuses his or her vision in point E in 3D space, i.e. a bit *behind* the screen, at the same time as two pixels are plotted in point A and B, that person would see a *virtual* pixel in point E. The same goes for points F which is generated by pixels in points B and C – note that point B is common for both point E and F – and point G which is generated by pixels in points C and D.

This means that if pixels are plotted in points A, B, C and D, the points E, F and G are generated which create a *virtual surface in 3D space, some distance behind the screen.* The farther apart the pixels are in points A, B, C and D the farther away behind the screen the virtual surface will be. To get it right, however, a detailed algorithm is needed.

5.2 The RDS algorithm

With fig. 5.1 in mind let's have a look at the following algorithm, which describes how to generate the RDS:

1. Get a random point on the surface to begin with. (F)
2. Draw a line to the left eye. (3)
3. Plot a pixel where line intercepts screen. (B)
4. Draw a line from F to right eye. (4)
5. Plot a pixel where line intercepts screen. (C)
6. Draw a line from left eye through C (5) and find interception with surface. (G)
7. Draw a line from G to right eye. (6)
8. Plot a pixel where line intercepts screen. (D)

9. Repeat steps 6 to 8 incrementing the points towards the right until edge of screen is reached.
10. Draw a line from right eye through B (2) and find interception with surface. (E)
11. Draw a line from E to left eye. (1)
12. Plot a pixel where line intercepts screen. (A)
13. Repeat steps 10 to 12 incrementing the points towards the right until edge of screen is reached.
14. Repeat steps 1 to 13 with new initial point until enough points are generated to create a nice looking surface.

That is the core of the RDS technology. Now, let's have a look at some mathematics in 3D space in order to understand how this basic algorithm is implemented in reality.

5.3 3D space geometry

In order to do any kind of calculations of coordinates, we need to define origo. With fig. 5.1 still in mind let's have a look at a refined version in 3D in fig. 5.2, where a coordinate system is defined emerging from origo:

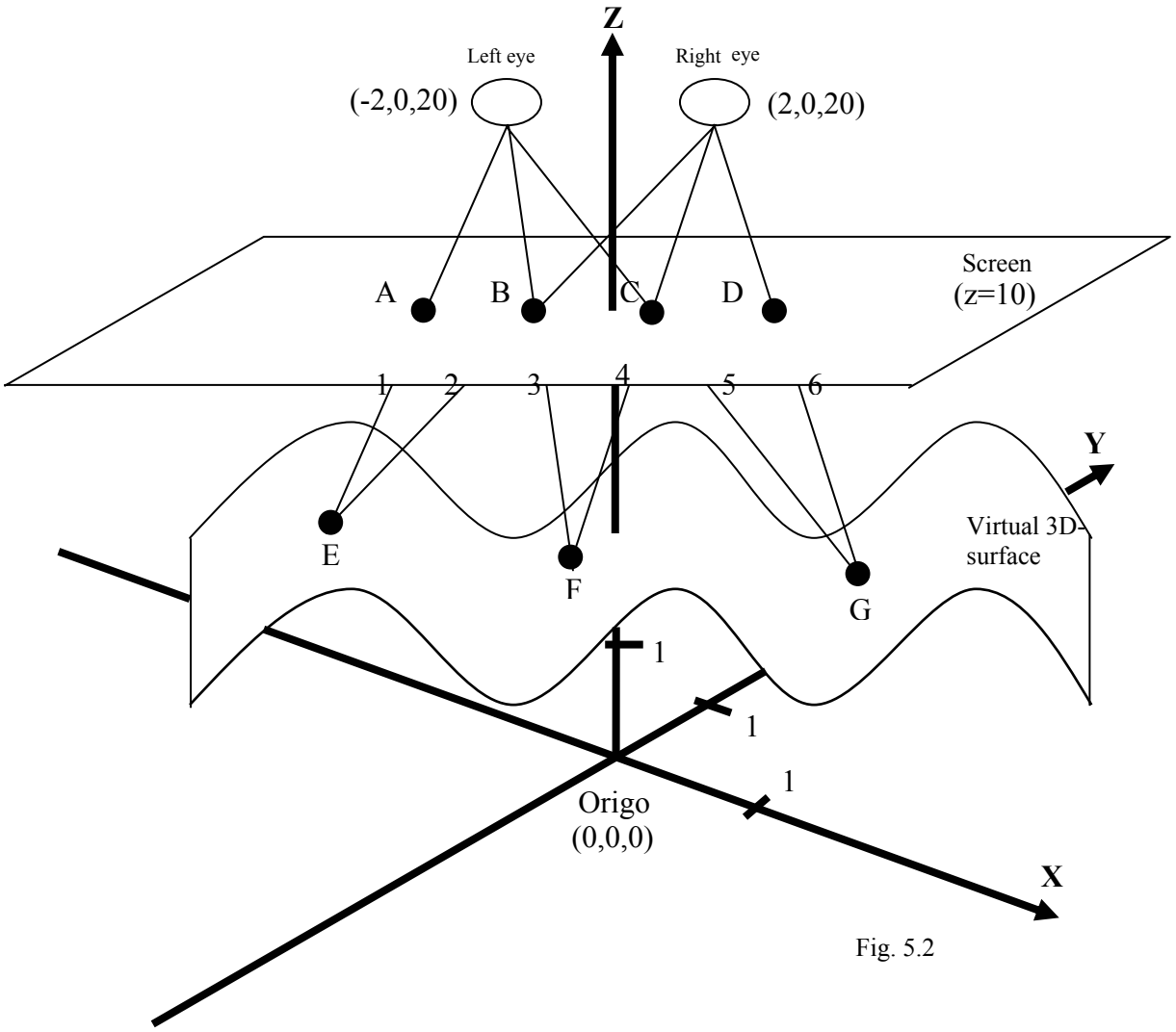

Z
Left eye
Right eye
(-2,0,20)
(2,0,20)
A
B
C
D
Screen
(z=10)
1
2
3
4
5
6
Y
E
F
G
Virtual 3D-
surface
1
1
1
Origo
(0,0,0)
X

Fig. 5.2

The notation z=10 for the screen and z=f(x,y) for the surface we will come back to in a moment.

Now the foundation for drawing lines in 3D space is in place. Now let's look at how to draw in this coordinate system.

5.3.1 Points in 3D space

Points in 3D space are defined as an x-value, a y-value and a z-value, one for each direction in space.[13] See fig 5.2. Thus every point can be described as a triple (x,y,z).

5.3.2 Lines in 3D space

A line in 3D space is something that connects two points in 3D space. Any two points define a unique line, due to the line's linear properties. This is trivial. Thus a line can be defined as a prolonging of the starting point, like this:

$(x,y,z) = P_{init} + t*V_{direction}$, P_{init} being the starting point, $V_{direction}$ being the direction vector of the line and t any real number. The direction vector is a vector from starting point to end point defining the concept of direction in 3D space.

Thus, to conclude, if we have two points A = (x1,y1,z1) and B = (x2,y2,z2), a line between the two is defined as:

(x,y,z) = A + t * V

where V is defined as (x2-x1, y2-y1, z2-z1), and t any real number.[13]

5.3.3 Surfaces in 3D space

A general surface in 3D space can be defined when the z-values of the points on the surface are functions of the x- and y-values of those points. A trivial example is that of a plane parallel to the base x,y plane, which is defined as: z = 0*x + 0*y + K , K being any real constant. K = 10 generates z = 10, which means a plane parallel to the base x,y plane with a z-value of 10, i.e. 10 units above the base x,y plane. (See 'screen' in fig. 5.2)

Generally a surface can be defined as z = f(x,y), i.e. the z-value of the points on a surface in 3D space is a function of the x and y-values of that point. (See 'virtual 3D surface' if fig. 5.2) [13]

That is the 3D theory necessary to understand the RDS technology. Now let's get into the details of how to implement this theory in reality with the help of C++ and COM. But first some Windows programming foundations.

5.4 Windows programming foundations

5.4.1 Introduction

On the Windows platform, information is passed between applications and between application and the OS by sending *messages*. [2] A message is a defined type with fixed parameters, created by Microsoft. Furthermore all C++ Windows programmes are built upon the same basic skeleton, which can be broken down into three basic parts.

1. WinMain()
2. WindowProc()
3. The main event loop

All Windows programmes begin in WinMain() where the application window is defined and created, where after the main event loop is entered. In the main event loop the application begins listening for messages from the user or the system. If e.g. the user clicks a button in the application window, a message is generated and sent to that application. The applications then retrieves that message in the main event loop, which forwards it to the WindowProc() function which is where action is taken depending on the type of message. If e.g. the exit button has been clicked, action is taken to exit the application etc. When all processing is done, the main event loop enters a new round, picking up the next message, and so on. The main event loop exits on a special exit message. [2] Let's analyse these parts more in detail.

5.4.2 The Windows application skeleton

Beginning with WinMain, in pseudocode the works of this function looks like this:

```
WinMain()
{
    State the properties of the application window.
    Register the type of window above with Windows.
    Let windows create the window and show it on screen.

    Enter main event loop and start receiving messages,
    passing them through to WindowProc() for processing.

    When the main event loop exits, due to a quit-message,
    the application is killed.
}
```

Here we have some references to the other two basic parts, WindowProc() and the main event loop. Let's continue with the main event loop, which is nothing more that an ordinary while statement:

```
while( there are messages for me from the system, get them here )
{
     Do some standard formatting of message
     Send the message to WindowProc for processing.
}
```

Last let's have a look at WindowProc():

```
WindowProc()
{
     switch(message)
     {
      Action depending on message type
     }

     Return unprocessed messages to Windows for default
     handling
}
```

That is all information necessary about the Windows programming model at the moment. The details can be seen in appendix A, the source code, where further and more detailed documentation can be found. Now let's proceed with the details of the RDS Generator system model.

5.5 The RDS Generator system model

5.5.1 Requirements

With the RDS Generator, any shape where z = f(x,y) shall be drawn. The shapes supported in this example shall be:

1. An ascended rectangle above the x,y plane.
2. A hill-like surface with its peak at the centre of screen, sloping down in all other directions.
3. Circular waves, as when a stone is thrown into water.
4. Linear waves, as the waves of the ocean.

Note that *any* shape is possible that follows the rule z = f(x,y), and the system shall be easily updated with new shapes, as we will come back to a bit later. These four are just examples, since the focus is on how COM is used within the programme.

An external error log shall be created in the same directory as the application at execute time, where error messages shall be written. The external log is necessary since the

screen is unavailable at times of primary surface lock etc. See source code for details when screen is no long available for error messages. In this way the cause of unexpected exit of the application can still be found.

5.5.2 System architecture

I have chosen to create a three-layered system architecture with a single interface between the layers. This guarantees flexibility in the system as changes in one layer does not at all affect the other layers, as long as the interfaces are unchanged.[5] Again the component principle, although not in the shape of the full COM model.

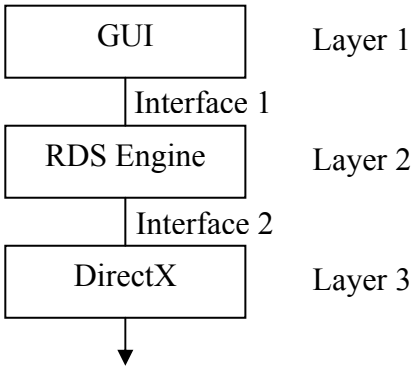


Fig. 5.3

Layer 1 is the graphical user interface (GUI). Here we find routines for setting the properties of the application window, colors, menus etc.

Layer 2 is the RDS engine that knows all about the RDS algorithm.

Layer 3 is DirectX, that knows how to plot the pixels from layer 2 on any video card on the market.

5.5.3 The interfaces of the RDS system model

5.5.3.1 Interface 1

Beginning in the GUI layer, the GUI is drawn on screen and the programme awaits user input from the menus. (See appendix B for screenshots.) When a particular shape is chosen from the menus, the contact with layer 2 is established by the creation of the object *RDSEngine*, and the shape chosen is passed as the argument to this layer, like this:

```
//Establish contact with layer 2.
RDSEngine RDSengine;
```

```
//Trigger the render of shape passed as the argument.
RDSengine.render(waves);
```

Here we see that the only interface to layer 2 is through the method render() of the class RDSEngine, which takes a shape as argument.

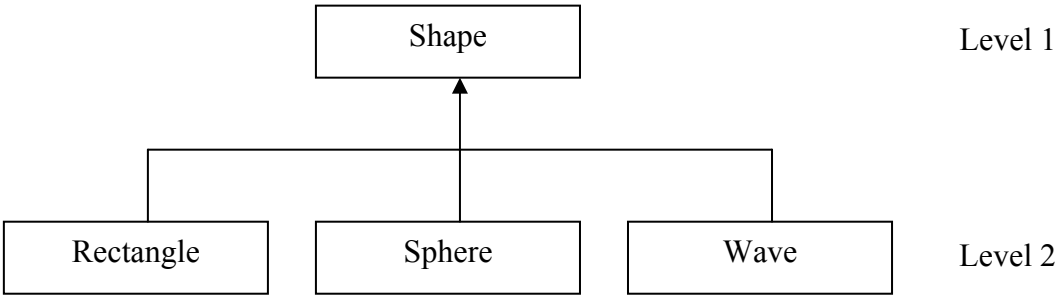


Fig. 5.4

Here I would like to discuss some interesting mechanisms. In order not to brake the single interface, the argument of the method render() must be the same all the time. If two types were possible, we would have two interfaces between the layers, which is bad. Low level changes inside layer 2 then maybe wouldn't be compatible with the other interface. To keep the design clean and robust, we want only one single entry point into a layer. Right, how can I design an interface that takes the same type all the time – to preserve the single interface which creates a robust solution – but still allows me to pass any shape I want for rendering? Different shapes have different properties, and thus need to be of different types!
The solution I found was to use the object-oriented mechanisms of inheritance and virtual functions.
Say that I define a class Shape, which contains only a function *getZvalue()*, which defines a shape in 3D space. (See previous chapters.) I then create tangible shapes like rectangles etc. by letting them inherit from this more abstract class Shape, like this:
The inherited shapes on level 2 *override* the base function getZvalue() and define the function representing that particular shape. The trick here is that any object on level 2 is *still a shape.* Thus, any one of these can be passed as an argument to render(), but still carry there own individual properties, extending base class shape.
This technique is called dynamic binding, and is based on the principle of virtual functions[14], which is the base for generating the behaviour that all objects are treated as a *Shape* in the interface, but when the method getZvalue() of *Shape* is called, the overridden variant, that resides inside the objects of level 2 in fig. 5.4, is actually called. This is indicated by the use of keyword *virtual* preceeding the functions in C++. See appendix A and documentation in the code for how this is implemented in detail.

Thus, by the use of virtual functions and inheritance, any new shape can be defined in layer 1 and simply passed down to layer 2 for rendering.

5.5.3.2 Interface 2

Interface 2 is the interface to directly access the VRAM of the video card installed, with the help of DirectX, to simply plot the pixels in the position calculated by layer 2, i.e. the RDS engine.

From within layer 2, connection is established to layer 3 by the creation of an object of class *DirectX*, which is a class I have defined myself, which we will see in detail later.

```
//Establish connection with layer 3.
DirectX directX;
```

The pointer to VRAM on the video card is retrieved from the DirectX object, like this,

```
UCHAR *video_buffer = directX.getVideo_buffer();
```

and a necessary memory pitch integer like this:

```
int mempitch = directX.getMemPitch();
```

The mempitch is a low-level DirectX detail needed to get the right y position on screen. This is due to the fact that memory in VRAM is linear, and the mempitch integer contains the right number of steps to increment the pointer in VRAM in order to get to the (linear) position in memory that represents the y-coordinate. That number of steps is different between different video cards, but DirectX returns the right one based on the card installed. [2]

With this stated, a pixel can now be plotted in position x,y on screen by the syntax:

```
video_buffer[x+y*mempitch] = plotcolor;  //plotcolor = 0,1,2…255
```

The single connection between layers 2 and 3 is, then, defined by the use of the object *DirectX*, from which we can retrieve the necessary variables to plot a pixel in any position on screen.

5.6 Discussion of problems and arguments for solutions

In this section I will present the problems that arouse during the development and discuss my solutions.

5.6.1 Virtual base plane initialisation

At first I only got pixels in a band across the screen, and it seemed impossible to get pixels plotted all the way up to the top of the screen, and down to the bottom of the screen. See fig. 5.5:

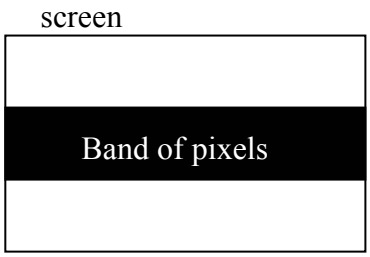


Fig. 5.5

I finally realised that I had missed the following mechanism:

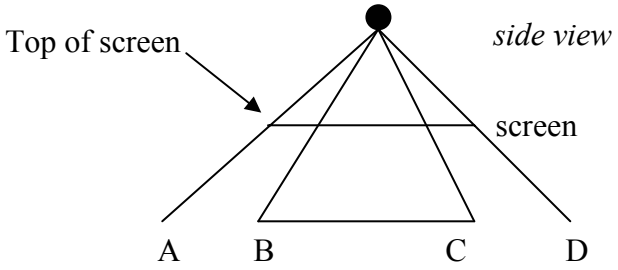


Fig 5.6

I initially only recognised the valid area to generate initial x,y values from as BC, when it actually is AB, i.e. the virtual base x,y plane is wider than the coordinates at the borders of the screen. Thus I created a function that calculated the virtual base plane and put the result in a global variable called *virtual_YMAX,* which I then used in the calculations in the RDS algorithm. It solved the problem. The use of a global variable is motivated, as the value can be seen as a constant, once calculated. It needs to be calculated as it is dependent on position of the eyes in the coordinate system, the position of the screen etc. See appendix A for details.

5.6.2 Blurred picture

I realised early that the RDS technology is very sensitive. The depth of any virtual point is a function of the distance between two pixels on screen, and if those pixels are only 1 pixel too far from each other, the virtual point generated will be experienced as being half a centimetre above or beneath the rest of the surface. That results in is quite a

disturbing blurred picture, which looks double, as some points are below and some above the correct surface.

After analysis of the conversion from virtual coordinates into screen coordinates (see chapter 5.6.4) I realised that the rounding of values from floats to integers generated the blur. Clearly, pixel coordinates are integers since it is not possible to plot half a pixel, and thus a minor fault is inevitable. But the fault could be minimised by plotting the pixel in the rounded integer position and then using that position in the following calculations. One pixel is common to two points, and the position of the previous is used to get the next. (See fig 5.1) Thus I didn't use the exact virtual coordinate in the next calculation, but calculated back the rounded position into virtual coordinates again, and used that value. By doing this, the blur was minimised, and hardly noticeable.

5.6.3 Safety margins

When the RDS algorithm (see chapter 5.2) reaches the edge of the screen, it should stop. But due to the rounding of floats described above, it was possible that a value outside the edge of the screen when calculated back into virtual coordinates, again was inside the valid screen area. If e.g. a point shall be plotted in virtual x-coordinate 5,01 using a resolution of 1024x768 – 5,0 being edge of screen – that coordinate was converted and rounded into screen x-coordinate 1023, since the algorithm for practical reasons plotted the last pixel *on* the very edge if it was out of bounds. The pixel was plotted and the value 1023 was calculated back into virtual coordinates again, this time evaluating to 4,98. Thus the algorithm thought that the point was still valid and tried to calculate the next point. That point was also outside the valid area and again rounded to 1023, since it was the last point and out of bounds. Thus the algorithm never exited. Therefore a safety margin is now used in the termination statement for the algorithm, so the rounding of edge values never occurs.

5.6.4 Conversion virtual coordinates – screen coordinates and back

The conversion equation from virtual coordinates to screen coordinates stems from the following analysis, where W is screen width in pixels, H screen height in pixels, X_{max} max virtual x coordinate, Y_{max} max virtual y coordinate, $X_{virtual}$ the virtual x-coordinate to be converted and $Y_{virtual}$ the virtual y-coordinate to be converted. Note that origo is defined as being always on centre of screen:

$$X_{pixel} = W/2 + (W/2)/X_{max} * X_{virtual} = (W*X_{max} + W*X_{virtual}) / (2*X_{max})$$

$$Y_{pixel} = H/2 - (H/2)/Y_{max} * Y_{virtual} = (H*Y_{max} - H*Y_{virtual}) / (2*Y_{max})$$

Or,

X_{pixel} = position of origo + (number of pixels per unit * units)

Y_{pixel} = position of origo + (number of pixels per unit * units)

For example with a screen resolution of 1024x768, X_{max} of 5 and Y_{max} of 3, top left virtual corner (-5,3) evaluates to

X_{pixel} = 1024*5 + 1024*(-5) / (2*5) = 0

Y_{pixel} = 768*3 + 768*(-3) / (2*3) = 0

i.e. position (0,0), which is correct, and further testing proves the formula to be correct in all other cases as well.

To convert back again, we simply solve for $X_{virtual}$ and $Y_{virtual}$ respectively and get:

$X_{virtual} = (2*X_{pixel}*X_{max} - W*X_{max}) / W$

$Y_{virtual} = (H*Y_{max} - 2*Y_{pixel}*Y_{max}) / H$

With the same example as above we get:

$X_{virtual} = (2*0*5 - 1024*5) / 1024 = (-1024*5)/1024 = -5$

$Y_{virtual} = (768*3 - 2*0*3) / 768 = (768*3)/768 = 3$

i.e. (-5,3) which is correct.

5.6.5 Interception of surface algorithm

Getting the coordinates where a line intercept the screen in the RDS algorithm is straightforward given the equation of the line. One simply solves for the parameter t for the z-value of screen, and then uses this value to calculate the corresponding (x,y,z) coordinate. (See chapter 5.3.2) This is simple because the screen is always on a fixed height.

The rendered surface on the other hand is not, and therefore an algorithm is needed in order to get the interception of the line and surface. (See steps 6 and 10 in the RDS algorithm.) One could argue that the value could be solved analytically and the exact value be found, but the interception routine must be generic and work on any surface renderable. Furthermore layer 2 (see fig. 5.3) should not have to be updated with new analytical solving routines if new shapes are added in layer 1. That would break the component behaviour of the layers, i.e. their complete independence of their

surroundings. And, many equations defining surfaces in 3D space are very complex indeed and are impossible to solve analytically. [13]

Thus an algorithm has been created based on the following basic principle, having step 6 in the RDS algorithm as example:

The starting point is when the line has been created from left eye to point C, and we want to find point G. The line is prolonged a small increment, and the z-values of the ending of the line and the z-value of the surface are compared. If the difference is smaller than a very small acceptable tolerance, the line and surface are reckoned as intercepted.

Now, in order to get as good exactness as possible, the increment must be small. After all, the exactness of the interception is dependent on how big steps we take each increment. The smaller the step, the better the exactness, but also the slower the algorithm. Thus to achieve better performance, big steps are taken until we have just passed the surface and are beneath it – z-value of line is smaller than z-value of surface with a difference dependent on the big step – and then very small steps are taken back again until the difference is smaller than a smaller and more exact tolerance.

Here is the formal interception algorithm, based on the RDS algorithm:

1. Create line from eye to point on screen.
2. Prolong the line a big increment.
3. Compare z-value of line ending, and z-value of surface on the same x,y values.
4. If z-value of line > z-value of surface repeat steps 2 – 3.
5. Line is beneath surface, shorten the line a small increment.
6. Compare z-value of line ending, and z-value of surface on the same x,y values.
7. If difference in z-values is bigger than exact tolerance, repeat steps 5 – 6.
8. Difference is smaller than exact tolerance. Return (x,y,z) interception point.

Note that steps 2 – 4 are very fast, while steps 5 – 7 are slow. The performance of this algorithm is based on the size of the steps, and the interception tolerance which can be set in the file constants.h where all constants controlling the system are located. I have found both very good performance and exactness with a big increment of 0.1 and a small of 0.001 units.

5.6.6 Test results of the application on different Windows configurations

The application was tested on different computers on campus Trollhättan in different configurations to test the functionality of DirectX, and was proven to work very well. On those computers not having DirectX installed the application detected it and notified the user to update DirectX, as planned.

The application was also sent via e-mail to a number of people, testing the application at home on their personal computers, reporting success.

Only on one computer was the application unable to lock the primary surface (see chapter 5.7) although the correct version of DirectX was detected. Since the application works fine on all other computers tested, the cause of the error must be related to the misconfiguration of that particular computer. The application exited securely as planned, reporting the error message in the error log.

5.6.7 Limitations

The smallest floating point type *float* has been used to hold the values of the coordinate system. The reasons for this is performance and effectiveness of memory usage. But, the maximum value to be held by a float is limited, and thus the system is limited to a region around origo. This is not a big problem, since many surfaces have their most interesting behaviour around origo. The tested configuration at present is maximum x-value of 5 and maximum y-value of 3, so it is safe and effective to use floats.

5.7 The details of how COM principles are implemented

Now that we have seen the workings of the RDS technology and the problems related to it, let's now analyse layer 1, DirectX, and see in detail how the COM principles are implemented there to plot pixels.
When the user has chosen a shape to be rendered, that shape is passed down to layer 2, the RDS engine. So far no DirectX is involved. The first thing that happens then in layer 2, is the establishment of the connection to layer 1, by the creation of the object of my own class *DirectX*.

```
DirectX directX;
```

Note that this is my own creation and no Microsoft product. I have simply collected everything that is related to DirectX in the class DirectX, to achieve the layered structure of the system model previously described, with single interfaces between the layers.
Then, for the first time DirectX routines are used by calling my method *initDirectX()*

```
directX.initDirectX();
```

which initialises the Microsoft technology. Let's analyse what happens in tat function.

5.7.1 Initialisation of DirectX

The first call to DirectX looks like this:

```
//First create base IDirectDraw interface
DirectDrawCreate(NULL, &lpdd, NULL);
```

DirectDrawCreate() is one on the previously mentioned *wrapper functions* provided by Microsoft, that do all the low-level details of COM. It loads the COM libraries from DLLs, then creates a base DirectDraw COM object and last initialises it. [11]

At this point a pointer to a base DirectDraw object is held in lpdd. Now, we want to retrieve another interface on that DirectDraw object, by the use of the following call:

```
//Query base IDirectDraw for another interface
lpdd->QueryInterface(IID_IDirectDraw4, (LPVOID *)&lpdd4)
```

This uses the method QueryInterface() defined in IUnknown, to query the object for version 6 of DirectDraw. The naming conventions by Microsoft is quite inconsistent, and version 6 of DirectDraw is actually called DirectDraw4, which can cause some trouble.

After this call, a pointer to the version 6 of the interface to DirectDraw is held in lpdd4. Note the use of the IID for DirectDraw4, a constant called IID_IDirectDraw4, which is the global unique identifier for that interface. The constant is defined in the file ddraw.h provided by Microsoft, like this:

```
DEFINE_GUID( IID_IDirectDraw4,
0x9c59509a,0x39bd,0x11d1,0x8c,0x4a,0x00,0xc0,0x4f,0xd9,0x30,0xc5 );
```

Now we have access to DirectDraw 6 and can begin using the functionality of that interface of DirectX.

Next, we release the old base interface, held by lpdd, since we don't need it anymore by the call to Release() originally defined in IUnknown:

```
//Release the base interface, since I don't need it anymore.
lpdd->Release();
```

Next we use DirectDraw 6 to set the cooperative level of the application. DirectDraw allows the programmer to get full control of the screen in full screen mode, or one can chose to work with a windowed application. We want full screen mode, which looks like this:

```
lpdd4->SetCooperativeLevel( hwnd,
                            DDSCL_FULLSCREEN|
                            DDSCL_ALLOWMODEX|
                            DDSCL_EXCLUSIVE|
```

```
                                DDSCL_ALLOWREBOOT );
```

hwnd is an identifier of the application DirectDraw is to be connected to, where after four flags are stated, logically OR:ed together. This is a common technique in DirectX. [2] The flags above creates an application that doesn't allow any other application to access the screen, but still listens for ctrl+alt+delete sequence. [2]

Now that we have full control over the screen, we set the display mode:

```
//Set Display Mode
lpdd4->SetDisplayMode(SCREEN_WIDTH,SCREEN_HEIGHT,COLOR_DEPTH,0,0);
```

The extra parameters set to zero at the end control the refresh rate of the screen and other extra advanced flags, which we don't use, thus set them to 0. [2]

Then, if 8 bit color is used the palette is created and initialised. This is not vital for the DirectDraw principle and will not be taken up here, but the details are documented in the code inside the *initDirectX()* method. See appendix A. The result is a palette structure which holds the desired 256 colors, here called *palette*, which will later be connected to the primary surface.

Next we define the properties of the surface which we use to draw on later, by filling in the details in the *surface descriptor*: There are lots of options here, [2] but we will only define the surface to be a primary surface, i.e. a surface immediately visible on screen, when draw upon:

```
//Set as primary surface.
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

Next we create the primary surface with help of the surface descriptor by the call:

```
//Create primary surface to draw on.
lpdd4->CreateSurface(&ddsd, &lpddsprimary, NULL);
```

A pointer to the surface is held in *lpddsprimary* after the call.

Then we use this pointer to attach the previously created palette to the primary surface like this:

```
lpddsprimary->SetPalette(lpddpal);
```

The last thing that needs to be done is to lock the surface, for two reasons. First, to block any other process to access the VRAM memory, and second to indicate that memory shouldn't be moved by some internal caching system on the video card. There is no guarantee that VRAM stays in the same place all the time, due to this, but when the primary surface – which is nothing more then a range of VRAM – is locked the

memory is not moved. We can manipulate it, and then unlock it again when we are done. [2] The call looks like this:

```
lpddsprimary->Lock( NULL, &ddsdlock,  DDLOCK_SURFACEMEMORYPTR|
                                      DDLOCK_WAIT,
                                      NULL );
```

The result of the lock routine is saved in *ddsdlock*. We are now approaching the end of the DirectX initialisation. The next two calls give us what we need in order to directly manipulate the screen, via the primary surface. Using *ddsdlock* we retrieve a pointer to the top left corner of the locked primary surface, and the previously mentioned memory pitch.

```
mempitch      = ddsdlock.lPitch;
video_buffer  = (UCHAR *)ddsdlock.lpSurface;
```

5.7.2 Using DirectX

At this point we have a pointer to VRAM and the tool to access x and y coordinates on screen (mempitch), which is what we wanted to achieve with layer 1 in the system model. The syntax for plotting pixels in position x,y is thus:

```
video_buffer[x+y*mempitch] = plotcolor;    //plotcolor = 0…255
```

Now DirectDraw is initialised, the screen is locked and in full control of the programmer, who can manipulate the screen freely, pixel by pixel. The two key parameters *mempitch* and *video_buffer* are extracted from the object directX by the help of the methods *directx.getMemPitch()* and *directx.getVideo_buffer()* in layer 2, where the intelligence and the RDS algorithm are located – the RDS engine.

5.7.3 Shutdown of DirectX

Once all drawing is through and the RDS is completed, the process of shutting down DirectX begins, on command of the user.

The shutdown process begins with the call *directX.closeDirectX(),* where the following call is made:

```
lpddsprimary->Unlock(NULL);
```

which unlocks the primary surface allowing VRAM to be altered and moved around again. Then cooperation level is restored to normal:

```
lpdd4->SetCooperativeLevel( hwnd, DDSCL_NORMAL  );
```

The palette is released:

```
lpddpal->Release();
```

The primary surface is released:

```
lpddsprimary->Release();
```

And finally the entire DirectDraw interface of DirectX is released:

```
lpdd4->Release();
```

And we are back at the standard GUI again. DirectX is out of the picture.

6 The future of COM — .NET

6.1 Introduction

In 2002 Microsoft released the successor to COM, called .NET, which will replace COM, as said earlier. .NET is a wider concept than COM, being a whole new platform for developing software, which is among other things language- and system independent. [15] I will not dig deeper into the principles of .NET, since that would be a dissertation of its own, but only describe the principles of how the creation of components differ from the COM way.

6.2 .NET vs. COM

With this focus in mind, .NET was developed as an answer to problems in the old COM technology, some of which were:

- The need for Global Unique Identifiers registered in the registry
- If the component would be used on several computers, a registry entry of the GUID had to be present on each machine.
- Complex installation due to the GUID principle, involving registering.
- The “DLL Hell” problem, which is problems with incompatibilities between different versions of a component, called by mistake.

The name for a component in .NET is *assembly*. Assemblies take the concept of a component even further, being completely self-containing, carrying all descriptions necessary within itself. Thus no need for GUIDs. The process of installing as assembly is thus merely to copy the assembly - always being a DLL - into the application folder, where it is directly accessible by the application using it. [16]

The “DLL Hell” problem is solved by means of incorporated version number, which is used together with the component name to access it. Thus it can never happen that a component of the wrong version is called by mistake.

6.3 Discussion

The .NET assemblies are easier to manage. They take care of all low-level details one is burdened with using COM [16], no installation is necessary making automated updates of programs possible. And, using .NET opens up the possibility of operating in a platform independent manner, regards taken to the complete .NET framework.[16]

On the other hand, vast amounts of code is present today based on COM, e.g. large portions of Windows itself together with applications like Word and Internet Explorer. Lots of other programmes not from Microsoft also rely on COM. Therefore, at present .NET cannot replace COM. Support is built in for bridging this gap however, using a .NET/COM proxy software, allowing .NET components be used in COM applications and vice versa.[15] But seen in a long perspective, .NET - as clearly stated by Microsoft - will completely replace COM, due to its inter-platform mechanisms and smoother management.

7 Result

The result is a complete Windows application based on the analysed principles of COM, which has complete control over the graphical display with help of DirectX, using the random dot stereoimage technology for creating 3D images on the screen.

8 Conclusion

Although the component paradigm can be less intuitive than the object-oriented paradigm, being more abstract and containing more complex structures, as clusters of related classes, it is my conclusion that the advantages outweigh the disadvantages, especially taking the .NET initiative into consideration. The .NET component principle, that pieces of software can be replaced simply by copying a file into a folder, carries tremendous strength and flexibility. A most important question indeed is the possibility to change the code later on, since it is very hard to foresee everything in the software development process - especially as customer requirements can easily change over time - and the component paradigm makes this possible due to it’s binary form. Furthermore the component principle creates as few connections as possible between different areas of the software, having only a well defined interface as point of interaction, ensuring robustness if changes are needed. Compared to the monolithic structures of the early times of software development, the control of the programmer has increased tremendously with this structural principle.

Another important issue is the reuse of software, and the component paradigm opens up for the possibility to create core functionality encapsulated, to be reused in different situations by many developers. With the object-oriented approach the classes are too specific, and cannot be created generic enough to form a reusable unit. With the component approach that is possible, even though it is an area of research at present, and lot of problems are yet to be solved [17].

Seen as a whole, though, as software gets more and more complex, it is my conclusion that the component paradigm is here to stay and that it will be the technology of the future, since it gives the programmer as much control as possible over the code.

8.1 Recommendation for future work

An interesting future project would be to implement the RDS generator as a fully COM compliant application, layers 2 and 3 being COM objects, accessible to other programmers wishing to use the RDS technology in their applications.

Also in that context, it would be interesting to use the future technology of .NET and the use of assemblies and manifests instead of GUIDs and IIDs to take the step even further into the future.

References

[1] Wikipedia. (2003). *Component object model*. [Electronic]. Wikipedia, the free encyclopedia. Available: < http://www.wikipedia.org/wiki/Component_object_model > [2003-05-18]

[2] LaMothe, A. (1999) *Tricks of the Windows Game Programming Gurus. Fundamentals of 2D and 3D game programming*. Indianapolis, USA: Sams.

[3] Wikipedia. (2003). *Enterprise Java Beans*. [Electronic]. Wikipedia, the free encyclopedia. Available: < http://www.wikipedia.org/wiki/Enterprise_Java_Beans > [2003-05-18]

[4] Wikipedia. (2003). *XPCOM* [Electronic]. Wikipedia, the free encyclopedia. Available: < http://www.wikipedia.org/wiki/XPCOM > [2003-05-18]

[5] Sommerville, I. (2001) *Software Engineering, 6th edition*. USA: Pearson Education Limited.

[6] Development of weather station application in the course Software engineering, 2003, HTU.

[7] Wikipedia. (2003). *Microsoft .NET*. [Electronic]. Wikipedia, the free encyclopedia. Available: < http://www.wikipedia.org/wiki/.NET > [2003-05-18]

[8] Brookshear, J. (2000) *Computer Science, 6th edition*. USA: Addison Wesley Longman, Inc.

[9] Ritchie, D. (1996). *The Development of the C Language*. [Electronic]. Bell Labs/Lucent Technologies Available: <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html> [2003-05-16]

[10] Hitmill (2003). *History of C++*. [Electronic]. Hitmill.com Available: < http://www.hitmill.com/programming/cpp/cppHistory.asp > [2003-05-18]

[11] ESRI (2003). *Introduction to COM*. [Electronic]. ESRI.com Available: < http://arcobjectsonline.esri.com/GettingStarted/IntroToCOM.htm > [2003-05-18]

[12] Chang, P. (1995). *Survey of Stereography*. [Electronic]. Nottingham.ac.uk
Available: < http://www.nottingham.ac.uk/~etzpc/ss/ssa.html > [2003-05-18]

[13] Neymark, M. *Analys i flera variabler*.Linköping: Department of Mathematics.

[14] Deitel & Deitel. *C++ How to program*, USA: Prentice-Hall, Inc.

[15] DrPizza (2003). *.NET and COM*. [Electronic]. Arstechnica.com
Available: < http://arstechnica.com/paedia/n/net/net-5.html > [2003-05-18]

[16] Thakkar, Y. (2001). *.NET Assemblies & Manifests*. [Electronic]. dotnetextreme.com
Available: < http://www.dotnetextreme.com/articles/assemblies.asp > [2003-05-18]

[17] Lectures by Ph.D student R. Torkar, HTU, 2003

Appendix A Source code

```
// constants.h - Constants that control global settings.

//Screen settings.
#define WINDOW_CLASS_NAME "WINCLASS1" //Name of application class
#define SCREEN_WIDTH  1024  //I use 1024x768 since Laptops work best
in this resolution.
#define SCREEN_HEIGHT 768
#define COLOR_DEPTH       8       //256 colors.

//Macros for sensing user pressing keyboard asyncronically, i.e. not
via Windows messages,
//but anytime.
#define KEYDOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 :
0)
#define KEYUP(vk_code)   ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 :
1)

//Max values of virtual coordinate system. See documentation for
details.
#define X_MAX 5
#define X_MIN -5
#define Y_MAX 4
#define Y_MIN -4

//Position of the eyes of the user in the virtual coordinate system.
#define EYE_RIGHT_X 1
#define EYE_RIGHT_Y 0
#define EYE_RIGHT_Z 19

#define EYE_LEFT_X -1
#define EYE_LEFT_Y 0
#define EYE_LEFT_Z 19

//Position of z-coordinate of screen in the virtual coordinate system.
#define SCREEN_Z 9

//Difference betweeen z-values to be considered equality.
#define INTERCEPT_TOLERANCE 0.001
```

```
//How much I shall extend a line each iteration to get surface
intercept. See documentation for details.
#define PARAM_INCREMENT 0.001    //In detail.
#define BIG_PARAM_INCREMENT 0.1  //Get near surface fast.

//Colors on screen.
#define PLOTCOLOR 0                  //Black pixels.
#define BACKGROUND_COLOR 255         //White background.

#define POINTS_TO_PLOT 5000      //How many original random pixels on
screen.
```

```
// directx.h - declaration of my own class DirectX.

//Used for popup message box.
#include <windows.h>

//Native DirectX header.
#include <ddraw.h>

//User defined symbols e.g. screen width etc.
#include "constants.h"

//Using global variable.
extern HWND hwnd;       //From rds.cpp.

class DirectX
{
     private:
     LPDIRECTDRAW  lpdd;                        //Standard DirectDraw
1.0.
     LPDIRECTDRAW4 lpdd4;                   //DirectDraw 6.0 interface
4.
     LPDIRECTDRAWPALETTE lpddpal;     //Palette interface.
     LPDIRECTDRAWSURFACE4 lpddsprimary;    //Primary DirectDraw
surface.
     DDSURFACEDESC2 ddsd;                   //DirectDraw surface
descriptor.
     DDSURFACEDESC2  ddsdlock;              //Will hold the result of
lock function.
     int mempitch;                          //Used to access y-values on
screen.
     UCHAR *video_buffer;                   //Address to VRAM extracted
from DirectDraw4.

     public:
     DirectX();                        //Constructor.
     bool initDirectX();               //Inits.
     void closeDirectX();         //Shuts down.
     int  getMemPitch();          //Extracts mempitch.
     UCHAR* getVideo_buffer();    //Extracts video_buffer.

};
```

```
//rds_res.h - defines for menuitems to be attached.

//Defines for top level menu FILE
#define MENU_FILE_ID_EXIT                       1000

//Defines for top level menu Render
#define MENU_RENDER_ID_RECT                     2000
#define MENU_RENDER_ID_HILL                     2001
#define MENU_RENDER_ID_WAVES                    2002
#define MENU_RENDER_ID_LINEARWAVES              2003

//Defines for top level menu Settings
#define MENU_SETTINGS_ID_RECT_HEIGHT_2          3000
#define MENU_SETTINGS_ID_RECT_HEIGHT_3          3001
#define MENU_SETTINGS_ID_RECT_WIDTH_3           3002
#define MENU_SETTINGS_ID_RECT_WIDTH_4           3003

#define MENU_SETTINGS_ID_HILL_HEIGHT_1          3004
#define MENU_SETTINGS_ID_HILL_HEIGHT_2          3005
#define MENU_SETTINGS_ID_HILL_HEIGHT_3          3006

#define MENU_SETTINGS_ID_CWAVES_FREQ_2          3007
#define MENU_SETTINGS_ID_CWAVES_FREQ_1          3008
#define MENU_SETTINGS_ID_CWAVES_FREQ_05         3009

#define MENU_SETTINGS_ID_LWAVES_FREQ_2          3010
#define MENU_SETTINGS_ID_LWAVES_FREQ_1          3011
#define MENU_SETTINGS_ID_LWAVES_FREQ_05         3012
```

```
//RDSEngine.h - declaration of classes used in the RDS Engine.

//Classes used.
#include "RDSshapes.h"
#include "DirectX.h"

/****************************************** CLASS Point
 A two dimensional point on the x,y plane.
 Use float to save memory.
 ******************************************/
class Point
{
     public:
     float x;
     float y;

     Point();               //Empty constructor, inits to (0,0).
     Point(float, float);   //Constructor.
};

/****************************************** CLASS PointDouble
 A two dimensional point on the x,y plane.
 Native windows functions return doubles.
 ******************************************/
class PointDouble
{
     public:
     double x;
     double y;

     PointDouble();                  //Empty constructor, inits to
(0,0).
     PointDouble(double, double); //Constructor.

};

/****************************************** CLASS Point3D
 A three dimensional point in 3D-space.
 ******************************************/
class Point3D
{
```

```
     public:
     float x;
     float y;
     float z;

     Point3D();                              //Empty constructor, inits
to (0,0,0).
     Point3D(float, float, float);    //Constructor.
};

/****************************************** RDSLine
 A line in 3D space in parameter form.
 ******************************************/
class RDSLine
{
     private:
     Point3D startPoint;
     Point3D directionVector;

     public:
     RDSLine(Point3D startPoint, Point3D EndPoint); //Constructor.
     Point3D getXYZvalue(float param);
     float getParamValue(float z);
};

/****************************************** RDSEngine
 The main engine that generates the RDS
 on screen.
 ******************************************/
class RDSEngine
{
     private:
     //Private member variables.
     Point3D rightEye;  //User eyes goes here, from constants.h
     Point3D leftEye;

     //Private functions.
     Point RDSRand();                                  //Get a random
point in the x,y plane.
     void renderHelpPoints( DirectX directX );  //Plots help points for
user.
```

```
    PointDouble transformToPixelCoords( double x, double y );
    //Converts to pixel coordinates.
    Point3D transformToVirtualCoords(double x, double y);
    //Converts to virtual coordinates.

    Point3D plotNextLeftPoint(Point3D, Shape &, DirectX);
    //Plots all points to the left given the previous.
    Point3D plotNextRightPoint(Point3D, Shape &, DirectX);
    //Plots all points to the right given the previous.

    public:
    RDSEngine();            //Constructor, inits the eye positions of
user, from constants.h
    void render(Shape &);   //Render shape. Takes address (&) to
                                //Shape-object to use dynamic binding
                                //and virtual functions. See
documentation for details.
};
```

```
//RDSshapes.h - Declarations of shapes to be rendered.

/******************************************** CLASS Shape
 The base class from which all other shapes,
 as rectangles, waves etc., inherit.
 ********************************************/
class Shape
{

     public:
     Shape();  //Constructor.

     //The matemathical expression that defines the shape. Returns z-
value from x,y-values.
     //Declared virtual to allow dynamic binding. See documentation for
details.
     virtual float getZvalue(float, float);

};

/******************************************** CLASS RDSRectangle
 A rectangle with variables for side, and
 height over base x,y plane.
 ********************************************/
class RDSRectangle : public Shape//Extends Shape
{
     public:
     RDSRectangle();              //Constructor.
     void setHeight(float);  //Sets the height.
     void setWidth(float);   //Sets the side of rectangle.

     //Overriding base-class virtual function, to define a rectangle.
     virtual float getZvalue(float, float);

     private:
     float height; //Height of rectangle.
     float side;        //Side of rectangle.
};

/******************************************** CLASS RDSHill
 A hill-formed shape that has its peak in
```

```
 origo and slopes towards the base plane
 in all directions.
 ******************************************/
class RDSHill : public Shape //Extends Shape
{
     public:
     RDSHill();                   //Constructor.
     void setHeight(float);  //Sets the height of the hill.

     //Overriding base-class virtual function, to define a hill.
     virtual float getZvalue(float, float);

     private:
     float height;           //The height of the hill.
};

/****************************************** CLASS RDSWaves
 Circular waves, as if a stone is thrown into
 water.
 ******************************************/
class RDSWaves : public Shape    //Extends Shape
{
     public:
     RDSWaves();                       //Constructor.
     void setFrequency(float);    //Sets the frequency of the waves.

     //Overriding base-class virtual function, to define circular
waves.
     virtual float getZvalue(float, float);

     private:
     float frequency;            //The frequency of the waves.
};

/****************************************** CLASS RDSLinearWaves
 Linear waves, as the waves on the ocean.
 ******************************************/
class RDSLinearWaves : public Shape   //Extends Shape
{
     public:
     RDSLinearWaves();           //Constructor.
```

```
    void setFrequency(float);    //Sets the frequency of the waves.

    //Overriding base-class virtual function, to define linear waves.
    virtual float getZvalue(float, float);

    private:
    float frequency;             //The frequency of the waves.
};
```

```
//rds_res.rc - The resource file that holds the menu definitions
//to be compiled into the .EXE file.

#include "rds_res.h"      //Menu id definitions.

MainMenu MENU DISCARDABLE //DISCARDABLE is a necessary standard
keyword.
{
      POPUP "File"
      {
           MENUITEM "E&xit", MENU_FILE_ID_EXIT         //The '&'
stands before the shortcut menu key.
      }

      POPUP "&Render"
      {
           MENUITEM "&Rectangle", MENU_RENDER_ID_RECT
           MENUITEM "&Hill", MENU_RENDER_ID_HILL
           MENUITEM "&Circular Waves", MENU_RENDER_ID_WAVES
           MENUITEM "&Linear Waves", MENU_RENDER_ID_LINEARWAVES
      }

      POPUP "Settings"
      {

           POPUP "&Rectangle"        //Cascading menus. The menu item
is itself a menu.
           {

                 POPUP "&Height"
                 {
                       MENUITEM "2", MENU_SETTINGS_ID_RECT_HEIGHT_2
                       MENUITEM "3", MENU_SETTINGS_ID_RECT_HEIGHT_3
                 }

                 POPUP "&Width"
                 {
                       MENUITEM "3", MENU_SETTINGS_ID_RECT_WIDTH_3
                       MENUITEM "4", MENU_SETTINGS_ID_RECT_WIDTH_4
                 }
           }
```

```
            POPUP "&Hill"
            {

                  POPUP "&Height"
                  {
                        MENUITEM "1", MENU_SETTINGS_ID_HILL_HEIGHT_1
                        MENUITEM "2", MENU_SETTINGS_ID_HILL_HEIGHT_2
                        MENUITEM "3", MENU_SETTINGS_ID_HILL_HEIGHT_3
                  }
            }

            POPUP "&Circular Waves"
            {

                  POPUP "&Frequency"
                  {
                        MENUITEM "High",
MENU_SETTINGS_ID_CWAVES_FREQ_2
                        MENUITEM "Medium",
MENU_SETTINGS_ID_CWAVES_FREQ_1
                        MENUITEM "Low",
MENU_SETTINGS_ID_CWAVES_FREQ_05
                  }
            }

            POPUP "&Linear Waves"
            {

                  POPUP "&Frequency"
                  {
                        MENUITEM "High",
MENU_SETTINGS_ID_LWAVES_FREQ_2
                        MENUITEM "Medium",
MENU_SETTINGS_ID_LWAVES_FREQ_1
                        MENUITEM "Low",
MENU_SETTINGS_ID_LWAVES_FREQ_05
                  }
            }

      } //End Settings
```

```
} //End MainMenu
```

```
//RDSshapes.cpp - implementation of shape code.
#include "RDSshapes.h"
#include <math.h>

//CLASS Shape

Shape::Shape() {}  //Constructor.

/*************************************************************
Shape::getZvalue()
 This function is overloaded in shapes who inherit from
 this class. The base-class function is never used.
 *************************************************************/
float Shape::getZvalue(float x, float y)
{
     return 0;
}


//CLASS RDSRectangle

RDSRectangle::RDSRectangle() //Constructor
{
     height    = 0; //Default height.
     side = 3; //Default width of side.
}

/*************************************************************
RDSRectangle::setHeight()
 Sets the height of the rectangle over x,y plane.
 *************************************************************/
void RDSRectangle::setHeight(float h)
{
     height = h;
}
/*************************************************************
RDSRectangle::setWidth()
 Sets the width of the rectangle over x,y plane.
 *************************************************************/
void RDSRectangle::setWidth(float w)
{
```

```
     side = w;
}

/*************************************************************
RDSRectangle::getZvalue()
 Overriding base-class function.
 *************************************************************/
float RDSRectangle::getZvalue(float x, float y)
{

     if( x>=(-side/2) && x<=(side/2) &&
          y>=(-side/2) && y<=(side/2) )
          return height;
     else
          return 0; //Base level
}




//CLASS RDSHill
//Constructor
RDSHill::RDSHill() { }

/*************************************************************
RDSHill::setHeight()
 Sets the height of the hill over x,y plane.
 *************************************************************/
void RDSHill::setHeight(float h)
{
     height = h;
}

/*************************************************************
RDSHill::getZvalue()
 Overriding base-class function.
 *************************************************************/
float RDSHill::getZvalue(float x, float y)
{

     return (float)exp(-x*x-y*y)*height;
```

```
}



//CLASS RDSWaves
//Constructor
RDSWaves::RDSWaves() { }

/************************************************************
RDSWaves::setFrequency()
 Sets the frequency of the waves.
 ************************************************************/
void RDSWaves::setFrequency(float f)
{
     frequency = f;
}

/************************************************************
RDSWaves::getZvalue()
 Overriding base-class function.
 ************************************************************/
float RDSWaves::getZvalue(float x, float y)
{

     return (float)cos(   sqrt(x*x + y*y)*frequency    );
}



//CLASS RDSLinearWaves
//Constructor
RDSLinearWaves::RDSLinearWaves() { }

/************************************************************
RDSLinearWaves::setFrequency()
 Sets the frequency of the waves.
 ************************************************************/
void RDSLinearWaves::setFrequency(float f)
{
     frequency = f;
```

```
}

/************************************************************
RDSLinearWaves::getZvalue()
 Overriding base-class function.
 ************************************************************/
float RDSLinearWaves::getZvalue(float x, float y)
{

     return (float)(cos((x+y)*frequency)*2);

}
```

```
// RDS Generator - written by Anders Alexandersson
// rds.cpp - main application file.

//============================================================ Includes
#define WIN32_LEAN_AND_MEAN      //Say no to MFC - unnecessary headers

//Includes.
#include <windows.h>
#include <math.h>
#include <ddraw.h>      //DirectX header
#include "rds_res.h"    //Menu header
#include "constants.h"  //Own constants, screen widths etc.
#include "RDSEngine.h"  //The RDSEngine

//Function declaration.
void initVirtualBase();

//Global variables
HWND hwnd;                   //Generic window handle, used to point to
this appliction.
int virtual_YMAX = 0;   //Virtual max value to pick randoms from.

//User settings will be held here, initialized to default values.
float rect_height_setting        = 2;
float rect_width_setting         = 3;
float hill_height_setting        = 2;
float circularWaves_frequency    = 2;
float linearWaves_frequency      = 1;


//============================================================ Functions
/*****************************************************************
WindowProc()
 This is the main event handler of the application. All messages from
 Windows arrives here for processing, i.e. when a button is clicked,
 the application window moved etc.
 *****************************************************************/
LRESULT CALLBACK WindowProc( HWND hwnd,
                             UINT msg,
                             WPARAM wparam,
                             LPARAM lparam )
```

```
{
     PAINTSTRUCT ps;         //Used in WM_PAINT when updating a window.
     HDC           hdc; //Handle to a device context.

     switch( msg )      //What message is coming in?
     {
          case WM_CREATE:
          {
               //<-- Initializations goes here. -->
               return(0);    //Tell Windows I processed the message.

          }break; //End case WM_CREATE.

          case WM_COMMAND:   //Coming from the menues
          {
               switch(LOWORD(wparam))  //What is the ID of clicked menu
choise?
               {
                    //=======================Handle the File menu.
                    case MENU_FILE_ID_EXIT:
                    {
                         PostMessage(hwnd, WM_DESTROY,0,0);   //Exit
application.
                    } break;

                    //=======================Handle the Render menu.
                    case MENU_RENDER_ID_RECT:
                    {
                         //User requested a rectangle!
                         //Create a rectangle
                         RDSRectangle rect;

                         //Set height to user setting.
                         rect.setHeight(rect_height_setting); //Global
variable.

                         //Set width to user setting.
                         rect.setWidth(rect_width_setting);
     //Global variable.

                         //Render!
```

```
                    RDSEngine RDSengine;
                    RDSengine.render(rect);

               } break;

               case MENU_RENDER_ID_HILL:
               {
                    //User requested a hill!
                    //Create a hill.
                    RDSHill hill;

                    //Set height to user setting.
                    hill.setHeight(hill_height_setting); //Global
variable.

                    //Render!
                    RDSEngine RDSengine;
                    RDSengine.render(hill);

               } break;

               case MENU_RENDER_ID_WAVES:
               {
                    //User requested circular waves!
                    //Create waves.
                    RDSWaves waves;

                    //Set frequency to user setting.
                    waves.setFrequency(circularWaves_frequency);
     //Global variable.

                    //Render!
                    RDSEngine RDSengine;
                    RDSengine.render(waves);

               } break;

               case MENU_RENDER_ID_LINEARWAVES:
               {
                    //User requested a linear wave!
                    RDSLinearWaves linearWaves;
```

```
                    //Set frequency to user setting.

    linearWaves.setFrequency(linearWaves_frequency);    //Global
variable.

                    //Render!
                    RDSEngine RDSengine;
                    RDSengine.render(linearWaves);

                } break;

                //======================Handle the Settings menu.
                case MENU_SETTINGS_ID_RECT_HEIGHT_2:
                {
                    //Global variable.
                    rect_height_setting = 2;
                    MessageBox(NULL, "Height set to 2!",
"Rectangle", 0);
                } break;

                case MENU_SETTINGS_ID_RECT_HEIGHT_3:
                {
                    //Global variable.
                    rect_height_setting = 3;
                    MessageBox(NULL, "Height set to 3!",
"Rectangle", 0);
                } break;

                case MENU_SETTINGS_ID_RECT_WIDTH_3:
                {
                    //Global variable.
                    rect_width_setting = 3;
                    MessageBox(NULL, "Width set to 3!",
"Rectangle", 0);
                } break;

                case MENU_SETTINGS_ID_RECT_WIDTH_4:
                {
                    //Global variable.
                    rect_width_setting = 4;
```

```
                    MessageBox(NULL, "Width set to 4!",
"Rectangle", 0);
                } break;

                case MENU_SETTINGS_ID_HILL_HEIGHT_1:
                {
                    //Global variable.
                    hill_height_setting = 1;
                    MessageBox(NULL, "Height set to 1!", "Hill",
0);
                } break;

                case MENU_SETTINGS_ID_HILL_HEIGHT_2:
                {
                    //Global variable.
                    hill_height_setting = 2;
                    MessageBox(NULL, "Height set to 2!", "Hill",
0);
                } break;

                case MENU_SETTINGS_ID_HILL_HEIGHT_3:
                {
                    //Global variable.
                    hill_height_setting = 3;
                    MessageBox(NULL, "Height set to 3!", "Hill",
0);
                } break;

                case MENU_SETTINGS_ID_CWAVES_FREQ_2:
                {
                    //Global variable.
                    circularWaves_frequency = 2;
                    MessageBox(NULL, "Frequency set to High!",
"Circular Waves", 0);
                } break;

                case MENU_SETTINGS_ID_CWAVES_FREQ_1:
                {
                    //Global variable.
                    circularWaves_frequency = 1;
```

```
                        MessageBox(NULL, "Frequency set to Medium!",
"Circular Waves", 0);
                    } break;

                    case MENU_SETTINGS_ID_CWAVES_FREQ_05:
                    {
                        //Global variable.
                        circularWaves_frequency = 0.5;
                        MessageBox(NULL, "Frequency set to Low!",
"Circular Waves", 0);
                    } break;

                    case MENU_SETTINGS_ID_LWAVES_FREQ_2:
                    {
                        //Global variable.
                        linearWaves_frequency = 2;
                        MessageBox(NULL, "Frequency set to High!",
"Linear Waves", 0);
                    } break;

                    case MENU_SETTINGS_ID_LWAVES_FREQ_1:
                    {
                        //Global variable.
                        linearWaves_frequency = 1;
                        MessageBox(NULL, "Frequency set to Medium!",
"Linear Waves", 0);
                    } break;

                    case MENU_SETTINGS_ID_LWAVES_FREQ_05:
                    {
                        //Global variable.
                        linearWaves_frequency = 0.5;
                        MessageBox(NULL, "Frequency set to Low!",
"Linear Waves", 0);
                    } break;

                }//End switch wparam.

            }break; //End case MW_COMMAND.

            case WM_PAINT:
```

```
          {
               //Validate the window, standard.
               hdc = BeginPaint(hwnd, &ps);
               //<-- Own painting here -->
               EndPaint(hwnd,&ps);

               return(0);     //Tell Windows I processed the message.

          }break; //End case WM_PAINT.

          case WM_DESTROY:
          {
               //Kill the application by sending a WM_QUIT message
               PostQuitMessage(0);
               return(0);     //Tell Windows I processed the message.

          }break;

          default:break;
     }//End Switch

     //Send back messages we do not process to Windows for
     //default processing.
     return ( DefWindowProc(hwnd, msg, wparam, lparam) );

}//End WindowProc

/********************************************** initVirtualBase()
 Special function that initializes the valid x,y plane from which to
 choose random points. See documentation for details.
 ****************************************************************/
void initVirtualBase()
{
     Point3D eyeLevel(EYE_RIGHT_X,EYE_RIGHT_Y,EYE_RIGHT_Z);
     //Coordinate of eyes.
     Point3D screenBorder(0,Y_MAX,SCREEN_Z);
     //Coordinate of screen border.

     RDSLine maxYvirtual(eyeLevel, screenBorder);           //Line
from eyes to screen border.
```

```
     float t0 = maxYvirtual.getParamValue(0);
     //Parameter value for base plane interception.
     Point3D virtualBorder = maxYvirtual.getXYZvalue(t0);
     //Coordinate of border point on x,y plane.

     //Round correctly to integer value.
     int y_int = (int)virtualBorder.y;
     if ( ( virtualBorder.y - y_int ) > 0.5 )
     {
          virtual_YMAX = (int)ceil(virtualBorder.y);         //Global
variable.

     }
     else
          virtual_YMAX = (int)floor(virtualBorder.y);

}

//======================================================== WinMain
/*********************************************************************
 This is the main entry point for all Windows applications. The
 WINAPI declarator makes the startup parameters be passed from
 left to right instead of the normal right to left. Standard syntax.
 *********************************************************************/
int WINAPI WinMain( HINSTANCE hinstance,
                        HINSTANCE hprevinstance,
                        LPSTR lpcmdline,
                        int ncmdshow )
{
     WNDCLASSEX    winclass; //Holds the windowclass I create.
     MSG           msg;      //Generic Windows message.

     //============================================== Fill in Class
     //First fill in the window class structure. Here all configuration
     //takes place when creating a new window for the application.
     winclass.cbSize    = sizeof(WNDCLASSEX);
     winclass.style=    CS_DBLCLKS |  //Tracks double clicks.
                             CS_OWNDC |         //Unique device context
                                                //for each window in
class.
                             CS_HREDRAW |  //Repaints when changed
```

```
                                                  //horizontally.
                              CS_VREDRAW;         //Repaints when changed
                                                  //vertically

     winclass.lpfnWndProc = WindowProc;    //Tell Windows my
eventhandler
     winclass.cbClsExtra      = 0;              //Not used.
     winclass.cbWndExtra      = 0;              //Not used.
     winclass.hInstance  = hinstance; //The handle from Windows
                                                  //passed to WinMain.
     //The application icon goes here.
     winclass.hIcon      = LoadIcon(NULL, IDI_APPLICATION);

     //The cursor for the client area goes here.
     winclass.hCursor      = LoadCursor(NULL, IDC_ARROW);

     //The default background color goes here.
     //winclass.hbrBackground = GetStockObject(GRAY_BRUSH);
     winclass.hbrBackground = (HBRUSH__ *)GetStockObject(GRAY_BRUSH);

     //Attachment of menues goes here.
     winclass.lpszMenuName  = "MainMenu";

     //Name this class.
     winclass.lpszClassName = WINDOW_CLASS_NAME;

     //Choose the small application icon
     winclass.hIconSm      = LoadIcon(NULL, IDI_APPLICATION);

     //============================================ Register & Create
     //Now register the class with Windows, to be able to create
     //the new window. Exit if this process is unsuccessful, since
     //the registration is neccessary for the rest of the application.
     if( !RegisterClassEx(&winclass) )
          return(0);

     //Create the registred window and exit if unsuccessful.
     if (!(hwnd = CreateWindowEx( NULL,    //Extended windows styles,
                                                  //not used. Example is
                                                  //"always on top"
feature.
```

```
                              WINDOW_CLASS_NAME,   //Class to create
from.
                              "RDS Generator",         //Title.
                              WS_OVERLAPPEDWINDOW| //Standard window.
                              WS_VISIBLE,          //Initially
visible.
                              0,0,                      //Initial x,y
                              SCREEN_WIDTH,        //Initial width
                              SCREEN_HEIGHT,       //Initial height
                              NULL,                //Handle to
parent,

     //NULL=Desktop.
                              NULL,                //Handle for menu.
                              hinstance,           //From WinMain.
                              NULL )))                  //Advanced,
not used.
     {
          return(0);
     }

     //Maximixe window and welcome user.
     ShowWindow(hwnd, SW_MAXIMIZE);
     MessageBox(NULL,"Press the ecsape key to cancel RDS render at all
times!", "Welcome",0);
     //Calculate the valid x,y area. See documentation.
     initVirtualBase();

     //========================================== Enter event loop

     while( GetMessage(&msg, NULL, 0,0) )
     {
          //Translate any accelerator keys, standard.
          TranslateMessage(&msg);

          //Send the message to my own WindowProc.
          DispatchMessage(&msg);

     }//end while
```

```
    //=============================================================
//Exit application when the main event loop exits.
    //Exit application like this, standard.
    return(msg.wParam);

}//end WinMain
```

```
//RDSEngine.cpp - Layer 2

#include "RDSEngine.h"
#include <math.h>
#include "constants.h"

//================================================ CLASS RDSEngine

//Constructor
RDSEngine::RDSEngine()
{
     //Initialize eye coordinates
     rightEye.x = EYE_RIGHT_X;
     rightEye.y = EYE_RIGHT_Y;
     rightEye.z = EYE_RIGHT_Z;

     leftEye.x = EYE_LEFT_X;
     leftEye.y = EYE_LEFT_Y;
     leftEye.z = EYE_LEFT_Z;
}

//=====================================================================
Member functions
/*********************************************************************
RDSEngine::render()
 Main engine that generates any Shape-object on screen. See
 documentation for details.
 *********************************************************************/
void RDSEngine::render(Shape &orderedShape)
{
     //Init DirectX.
     //First create a DirectX object.
     DirectX directX;

     //Then init directX, i.e. go to full screen mode. See directX.cpp.
     //If unsuccessful, quit application.
     if( !directX.initDirectX() )
     {
          //MessageBox(NULL, "The application terminated due to DirectX
misconfiguration on your system.\nCause of error can be found in the
file RDSerror.log in samr directory as the RDS.EXE file.", "Error",0);
```

```
            PostMessage(hwnd, WM_DESTROY,0,0);    //Exit application.
            return;   //Go back to GUI and wait for application to exit.
      }

      //Then get the key variables from directX which are
      //mempitch, to access y-coords, and video_buffer, the
      //address to VRAM in which to write directly.
      int mempitch = directX.getMemPitch();
      UCHAR *video_buffer = directX.getVideo_buffer();

      //Plot pixels using the syntax: video_buffer[x+y*mempitch] =
plotcolor;


      //========================================== Start of RDS
Algorithm.
      //See documentation for details of RDS algorithm.


      //Draw help points, i.e. big points on bottom of screen.
      renderHelpPoints(directX);

      //Main loop that plots the pixels.
      for (int i=0; i < POINTS_TO_PLOT; i++)     //POINTS_TO_PLOT
defined in constants.h
      {
                //Check each loop if user aborts by pressing escape.
                if( KEYDOWN(VK_ESCAPE) )
                     break;

                //Get a random 2D point on the XY-plane.
                Point p = RDSRand();

                //Get z-value of shape.
                float z = orderedShape.getZvalue(p.x,p.y);

                //First point on surface
                Point3D surfacePoint1(p.x, p.y, z);

                //========================================== To right
eye.
```

```
            //Draw a line to right eye.
            RDSLine rightEyeLine(surfacePoint1, rightEye);

            //Get the parameter value which corresponds to
interception of screen.
            float paramScreen =
rightEyeLine.getParamValue(SCREEN_Z);

            //Get screen interception. Save for going to the right
later.
            Point3D screenIntercept1 =
rightEyeLine.getXYZvalue(paramScreen);

            //Convert to pixel coordinates. See documentation for
details.
            PointDouble screenPoint =
transformToPixelCoords(screenIntercept1.x, screenIntercept1.y );
            //Convert back again to correct round errors ofr next
point. See documentation for details.
            screenIntercept1 =
transformToVirtualCoords(screenPoint.x, screenPoint.y);

            //Now plot the pixel using DirectX.
            //Cast not dangerous since x and y are whole numbers
from transformToPixelCoords.

    video_buffer[(int)screenPoint.x+(int)screenPoint.y*mempitch] =
PLOTCOLOR;    //PLOTCOLOR defined in constants.h

            //============================================== To left
eye.
            //Draw a line to left eye.
            RDSLine leftEyeLine(surfacePoint1, leftEye);

            //Get the parameter value which corresponds to
interception of screen.
            paramScreen = leftEyeLine.getParamValue(SCREEN_Z);

            //Get screen interception.
            Point3D screenInterceptLeft =
leftEyeLine.getXYZvalue(paramScreen);
```

```
              //Convert to pixel coordinates. See documentation for
details.
              screenPoint =
transformToPixelCoords(screenInterceptLeft.x, screenInterceptLeft.y);
              //Convert back again to correct round errors ofr next
point. See documentation for details.
              screenInterceptLeft =
transformToVirtualCoords(screenPoint.x, screenPoint.y);

              //Plot pixel using DirectX again.

     video_buffer[(int)screenPoint.x+(int)screenPoint.y*mempitch] =
PLOTCOLOR;

              //============================================= Rest of
pixels to the left.

              //Plot all points to the left of screenIntercept1
              //The extra 0.5 is a safety margin because the values
are
              //rounded and sometimes the limit X_MIN is never
reached.
              //See documentation for details.
              while ( screenInterceptLeft.x > X_MIN+0.5 )
              {
                   screenInterceptLeft =
plotNextLeftPoint(screenInterceptLeft, orderedShape, directX);
              }

              //============================================= Rest of
pixels to the right.

              //Plot all points to the right of screenIntercept1

              //Copy screenIntercept1 into screenInterceptRight for
more logical name in loop.
              Point3D screenInterceptRight(    screenIntercept1.x,
     screenIntercept1.y,     screenIntercept1.z );

              //The extra 0.5 same as above.
```

```
               while ( screenInterceptRight.x < X_MAX-0.5 )
               {
                    screenInterceptRight =
plotNextRightPoint(screenInterceptRight, orderedShape, directX);
               }

     }//End for.

     //Image finished. Wait for user.
     while(!KEYDOWN(VK_ESCAPE)) {}    //Wait until user press escape.

     //======================================== Shutdown DirectX.
     //Shutdown DirectX and return control to standard Windows GUI.

     directX.closeDirectX();

}//End render().


/*****************************************************************
RDSEngine::RDSRand()
 Generates random values on the valid x,y plane.
 *****************************************************************/
Point RDSEngine::RDSRand()
{
     //Local x,y values and negativity variable.
     int       neg;
     float     x,y;

     //Get random (x,y) values.
     x = (float) (rand()%(X_MAX*1000+1));  //A float from 0 to
X_MAX*100.
     x = x / 1000;                              //0 to X_MAX, 3
decimals.

     neg  = rand()%2;                                  //An integer from
0 to 1.
     if(neg)
          x*=-1;                                       //x is now a float
from -X_MAX to +X_MAX, with three decimals.
```

```
     extern int virtual_YMAX;
     y = (float)(rand()%(virtual_YMAX*1000+1)); //Same as x-coordinate
above.
     y = y / 1000;

     neg  = rand()%2;
     if(neg)
          y*=-1;

     Point p(x,y); //Create a Point with the random values.
     return p;

} //End RDSRand()


/*****************************************************************
RDSEngine::transformToVirtualCoords()
 Converts from pixel values to virtual coordinates. Used to avoid a
 blurred image. See documentation for details.
 *****************************************************************/
Point3D RDSEngine::transformToVirtualCoords(double x, double y)
{
     //New 3D point.
     Point3D virtualCoords;

     //Calculate pixel coordinates to virtual. See documentation for
details.
     virtualCoords.x = (float)(x*2*X_MAX-
SCREEN_WIDTH*X_MAX)/SCREEN_WIDTH;
     virtualCoords.y = (float)(SCREEN_HEIGHT*Y_MAX-
y*2*Y_MAX)/SCREEN_HEIGHT;
     virtualCoords.z = SCREEN_Z; //Pixels always on screen level.

     return virtualCoords;

}//End transformToVirtualCoords

/*****************************************************************
RDSEngine::renderHelpPoints()
 Renders two help point for the inexperienced user on bottom of
 screen.
```

```
 ***************************************************************/
void RDSEngine::renderHelpPoints( DirectX directX )

{

     //Exctract directX variables for rendering on screen.
     UCHAR*    video_buffer  = directX.getVideo_buffer();
     int       mempitch      = directX.getMemPitch();

     //Define virtual coordinates for help point.
     extern int virtual_YMAX; //Global variable.
     Point3D helpPoint(0, (float)(-1*virtual_YMAX+1), 0);

     //Draw a line from help point to right eye.
     RDSLine rightEyeLine(helpPoint, rightEye);

     //Get the parameter value which corresponds to interception of
screen.
     float paramScreen = rightEyeLine.getParamValue(SCREEN_Z);

     //Get screen interception.
     Point3D screenIntercept = rightEyeLine.getXYZvalue(paramScreen);

     //Get pixel coordinates.
     PointDouble screenPoint =
transformToPixelCoords(screenIntercept.x, screenIntercept.y );

     //Plot pixel using DirectX!
     //Cast not dangerous since x and y are whole numbers.
     //See transformToPixelCoords for details.

     //Plot 5x5 help point.
     for(int k=0; k<5; k++)
          for(int m=0; m<5; m++)

     video_buffer[(int)(screenPoint.x+k)+(int)(screenPoint.y+m)*mempitc
h] = PLOTCOLOR;



     //Draw a line to left eye.
     RDSLine leftEyeLine(helpPoint, leftEye);
```

```
     //Get the parameter value which corresponds to interception of
screen.
     paramScreen = leftEyeLine.getParamValue(SCREEN_Z);

     //Get screen interception.
     Point3D screenInterceptLeft =
leftEyeLine.getXYZvalue(paramScreen);

     //Get pixel coordinates.
     screenPoint = transformToPixelCoords(screenInterceptLeft.x,
screenInterceptLeft.y);

     //Plot pixel using DirectX!

     //Plot 5x5 help pixel.
     for(int i=0; i<5; i++)
          for(int j=0; j<5; j++)

     video_buffer[(int)(screenPoint.x+i)+(int)(screenPoint.y+j)*mempitc
h] = PLOTCOLOR;

}//End renderHelpPoints

/*****************************************************************
RDSEngine::transformToPixelCoords()
 Convert from virtual pixels to screen coordinates in which to plot
 pixels.
 *****************************************************************/
PointDouble RDSEngine::transformToPixelCoords(double x, double y )
{
     //New x,y point.
     PointDouble xy_pixels;

     //See documentation for detils of conversion.
     xy_pixels.x = ( (SCREEN_WIDTH*X_MAX + SCREEN_WIDTH*x)/(2*X_MAX) );
     xy_pixels.y = ( (SCREEN_HEIGHT*Y_MAX - SCREEN_HEIGHT*y)/(2*Y_MAX)
);

     //Round the x-pixel to nearest integer.
     int x_int = (int)xy_pixels.x;
```

```
     if ( ( xy_pixels.x - x_int ) > 0.5 )
     {
          xy_pixels.x = ceil( xy_pixels.x );
     }
     else
          xy_pixels.x = floor(xy_pixels.x);

     //Round the y-pixel to nearest integer.
     int y_int = (int)xy_pixels.y;
     if ( ( xy_pixels.y - y_int ) > 0.5 )
     {
          xy_pixels.y = ceil( xy_pixels.y );
     }
     else
          xy_pixels.y = floor(xy_pixels.y);


     //Check if we are out of bounds after rounding.
     //Not necessary because of safety margins, but
     //keeping just to be safe.
     if ( xy_pixels.x < 0 )
          xy_pixels.x = 0;
     else
     if ( xy_pixels.x >= SCREEN_WIDTH )
          xy_pixels.x = SCREEN_WIDTH-1;

     if ( xy_pixels.y < 0 )
          xy_pixels.y = 0;
     else
     if (xy_pixels.y >= SCREEN_HEIGHT )
          xy_pixels.y = SCREEN_HEIGHT-1;

     return xy_pixels;

}//End transformToPixelCoords().

/*********************************************************************
RDSEngine::plotNextLeftPoint()
 Plots the next point to the left, given the previous point and
 shape. See documentation for details of algorithm.
 *********************************************************************/
```

```
Point3D RDSEngine::plotNextLeftPoint( Point3D screenInterceptLeft,
                                      Shape &orderedShape,
                                      DirectX directX )
{
     //Draw a line FROM right eye to screenInterceptLeft.
     RDSLine rightEyeLine(rightEye, screenInterceptLeft);

     //Get surface interception.
     bool tooFar = true;                              //true == Z-values
are too far from each other.
     float t = (float)(1+PARAM_INCREMENT); //t==1 == screen intercept.
     Point3D surfaceIntercept;                  //Will hold the surface
interception point.

     //Take big steps, for performance, until we reach just beyond
surface interception.
     while(tooFar)
     {
          //Get z-value of line at length t from right eye.
          surfaceIntercept = rightEyeLine.getXYZvalue(t);

          //Check if we are beyond surface. Calculate z-values for both
line and surface with same (x,y,z).
          if( (surfaceIntercept.z -
orderedShape.getZvalue(surfaceIntercept.x,surfaceIntercept.y)) < 0 )
          {
               tooFar = false;    //Close to interception! Exit loop.
          }
          else
               t += (float)BIG_PARAM_INCREMENT; //Lengthen line towards
surface.

     }//End while


     //We are close. Now iterate in detail.
     tooFar = true;//New loop.

     while(tooFar)
     {
          //Get z-value of line at length t from right eye.
```

```
          surfaceIntercept = rightEyeLine.getXYZvalue(t);

          //This difference will alway be positive, as we are below
surface.
          //See documentation for details.
          if(
(orderedShape.getZvalue(surfaceIntercept.x,surfaceIntercept.y) -
surfaceIntercept.z ) < INTERCEPT_TOLERANCE )
          {
               tooFar = false;    //Interception! Exit loop.
          }
          else
               t -= (float)PARAM_INCREMENT; //Shorten line a bit.

     }//End while

     //surfaceIntercept now holds the new point on surface to draw line
from.

     //Draw a line to left eye.
     RDSLine leftEyeLine(surfaceIntercept, leftEye);

     //Get the parameter value which corresponds to interception of
screen.
     float paramScreen = leftEyeLine.getParamValue(SCREEN_Z);

     //Get screen interception.
     screenInterceptLeft = leftEyeLine.getXYZvalue(paramScreen);

     //Get pixel coordinates.
     PointDouble screenPoint =
transformToPixelCoords(screenInterceptLeft.x, screenInterceptLeft.y );
     //Convert back again to aviod blurred image. See documentation for
details.
     screenInterceptLeft = transformToVirtualCoords(screenPoint.x,
screenPoint.y);

     //Plot pixel using DirectX!

     UCHAR*  video_buffer = directX.getVideo_buffer();
     int       mempitch   = directX.getMemPitch();
```

```
     //Check if we are out of bounds.
     if( !(screenInterceptLeft.x <= X_MIN) )
     {

     video_buffer[(int)screenPoint.x+(int)screenPoint.y*mempitch] =
PLOTCOLOR;
     }

     return screenInterceptLeft;

}//End plotNextLeftPoint()

/*****************************************************************
RDSEngine::plotNextRightPoint()
 Plots the next point to the right, given the previous point and
 shape. See documentation for details of algorithm. Exactly same as
 plotNextLeftPoint but mirrored to the other eye.
 *****************************************************************/
Point3D RDSEngine::plotNextRightPoint( Point3D screenInterceptRight,
                                       Shape &orderedShape,
                                       DirectX directX )
{
     //Draw a line FROM left eye to screenInterceptRight.
     RDSLine leftEyeLine(leftEye, screenInterceptRight);

     //Get surface interception.
     bool tooFar = true;                              //Z-values are too
far from each other.
     float t = (float)(1+PARAM_INCREMENT); //t==1 == screen intercept.
     Point3D surfaceIntercept;                  //Will hold the surface
interception point.

     //Take big steps, for performance, until we reach just beyond
surface interception.
     while(tooFar)
     {
          //Get z-value of line at length t from right eye.
          surfaceIntercept = leftEyeLine.getXYZvalue(t);
```

```
          //Check if we are beyond surface. Calculate z-values for both
line and surface with same (x,y,z).
          if( (surfaceIntercept.z -
orderedShape.getZvalue(surfaceIntercept.x,surfaceIntercept.y)) < 0 )
          {
               tooFar = false;    //Close to interception! Exit loop.
          }
          else
               t += (float)BIG_PARAM_INCREMENT; //Lengthen line towards
surface.

     }//End while


     //We are close. Now iterate in detail.
     tooFar = true;//New loop.

     while(tooFar)
     {
          //Get z-value of line at length t from right eye.
          surfaceIntercept = leftEyeLine.getXYZvalue(t);

          //This difference will alway be positive, as we are below
surface.
          //See documentation for details.
          if(
(orderedShape.getZvalue(surfaceIntercept.x,surfaceIntercept.y) -
surfaceIntercept.z ) < INTERCEPT_TOLERANCE )
          {
               tooFar = false;    //Interception! Exit loop.
          }
          else
               t -= (float)PARAM_INCREMENT;//Shorten line a bit.

     }//End while

     //surfaceIntercept is the new point on surface to draw line from.

     //Draw a line to right eye.
     RDSLine rightEyeLine(surfaceIntercept, rightEye);
```

```
     //Get the parameter value which corresponds to interception of
screen.
     float paramScreen = rightEyeLine.getParamValue(SCREEN_Z);

     //Get screen interception.
     screenInterceptRight = rightEyeLine.getXYZvalue(paramScreen);

     //Get pixel coordinates.
     PointDouble screenPoint =
transformToPixelCoords(screenInterceptRight.x,
screenInterceptRight.y);

     //Convert back again for exactness in next point. See
documentation for details.
     screenInterceptRight = transformToVirtualCoords(screenPoint.x,
screenPoint.y);

     //Plot pixel using DirectX!

     UCHAR*  video_buffer = directX.getVideo_buffer();
     int      mempitch   = directX.getMemPitch();

     //Check if we are out of bounds. See documentation for details.
     if( !(screenInterceptRight.x >= X_MAX) )
     {

     video_buffer[(int)(screenPoint.x)+(int)(screenPoint.y)*mempitch] =
PLOTCOLOR;
     }

     return screenInterceptRight;

}//End plotNextRightPoint

//============================================================ CLASS
Point
//Constructors.
Point::Point() {x=0; y=0;}
Point::Point(float x_init, float y_init)
{
     x = x_init;
```

```
     y = y_init;
}

//============================================================ CLASS
PointDouble
//Constructors.
PointDouble::PointDouble() {x=0; y=0;}
PointDouble::PointDouble(double x_init, double y_init)
{
     x=x_init;
     y=y_init;
}

//============================================================ CLASS
Point3D
//Constructors.
Point3D::Point3D() {x=0;y=0;z=0;}
Point3D::Point3D(float x_init, float y_init, float z_init)
{
     x    = x_init;
     y    = y_init;
     z    = z_init;
}

//============================================================ CLASS
RDSLine
RDSLine::RDSLine(Point3D starts, Point3D ends) //Constructor.
{
     //Calculate the line.
     startPoint = starts;
     directionVector.x = ends.x - starts.x;
     directionVector.y = ends.y - starts.y;
     directionVector.z = ends.z - starts.z;

}


/*****************************************************************
RDSLine::getXYZvalue()
 Returns a point given a certain parameter value.
 See documentation for details.
```

```
 ***********************************************************************/
Point3D RDSLine::getXYZvalue(float param)
{
     //Equation of a line in 3D space: v = startPoint +
t*directionVector;
     Point3D xyz;
     xyz.x = startPoint.x + param*directionVector.x;
     xyz.y = startPoint.y + param*directionVector.y;
     xyz.z = startPoint.z + param*directionVector.z;

     return xyz;

}

/***********************************************************************
RDSLine::getParamValue()
 Returns a parametervalue given a certain z-value. See documentation
for details.
 ***********************************************************************/
float RDSLine::getParamValue(float z)
{
     //Return the parameter value of z-value input.
     return (z - startPoint.z)/directionVector.z;

}//End getParamValue().
```

```
//directX.cpp - directX calls, layer 1.

#include "DirectX.h"
#include <fstream.h>

//Constructor.
DirectX::DirectX()
{
     lpdd           = NULL;
     lpdd4               = NULL;
     lpddpal             = NULL;
     lpddsprimary   = NULL;

     //ddsd has to be initialized later, since it depends on the init
     //process.

     //ddslock has to be initialized later, since it depends on the
init
     //of ddsd.

     //mempitch
     mempitch = 0;

     //video_buffer
     video_buffer = NULL;

}//End Constructor.

//=======================================================initDirectX()
bool DirectX::initDirectX()
{
     //Open error log to write error messages in, in case of problems.
     //Since no installation is required, I have no idea of where the
     //user opens the application. The file is therefore created in the
     //same directory as the application (no path).
     ofstream error("RDSerror.log", ios::out);

     //First create base IDirectDraw interface
     if (FAILED(DirectDrawCreate(NULL, &lpdd, NULL)))
     {
```

```
          MessageBox(NULL, "Could not create base IDirectDraw
Interface.\nThis application requires DirectX 6.0\nPlease update your
system and try again.", "Error",0);
          return false; //Quit further init of DirectX. Go back to
     }

     //Query base IDirectDraw for later interface
     if (FAILED(lpdd->QueryInterface(IID_IDirectDraw4, (LPVOID
*)&lpdd4)))
     {
          MessageBox(NULL, "Could not get latest interface.\nThis
application requires DirectX 6.0\nPlease update your system and try
again.", "Error",0);
          return false; //Quit.
     }

     //Release the base interface, since I don't need it anymore.
     lpdd->Release();
     lpdd = NULL;  //Set to NULL for safety.

     //Set cooperation level
     if (FAILED(lpdd4->SetCooperativeLevel(     hwnd,
                                                     //DDSCL_NORMAL|
//Alone!
                                                     DDSCL_FULLSCREEN|
                                                     DDSCL_ALLOWMODEX|
                                                     DDSCL_EXCLUSIVE|
                                                     DDSCL_ALLOWREBOOT
                                                     )))
     {
          MessageBox(NULL, "Could not set cooperation level.\nThis
application requires DirectX 6.0\nPlease update your system and try
again.", "Error",0);
          return false; //Quit.
     }

     //Set Display Mode
     if (FAILED(lpdd4-
>SetDisplayMode(SCREEN_WIDTH,SCREEN_HEIGHT,COLOR_DEPTH,0,0)))
     {
```

```
          //At this stage screen is locked and no messages can be shown
to user.
          //If any error occurs, write error message in error log and
quit. The user
          //will be notified that the error has occured and that
details can be found
          //in the error log.

          error << "Could not set Display Mode.";
          error.close();//Close file.

          return false; //Quit.
     }

     //Create palette with 356 colors
     PALETTEENTRY palette[256];  // Create palette data structure.

     for ( int color=1; color <256; color++ )
     {

          palette[color].peRed   = rand()%256;
          palette[color].peGreen = rand()%256;
          palette[color].peBlue  = rand()%256;

          palette[color].peFlags = PC_NOCOLLAPSE;    //Prevent system
from optimizing.

     }//End for

     //Black is 0.
     palette[0].peRed   = 0;
     palette[0].peGreen = 0;
     palette[0].peBlue  = 0;
     palette[0].peFlags = PC_NOCOLLAPSE;   //Prevent system from
optimizing.

     //White is 255.
     palette[255].peRed   = 255;
     palette[255].peGreen = 255;
     palette[255].peBlue  = 255;
```

```
     palette[255].peFlags = PC_NOCOLLAPSE; //Prevent system from
optimizing.

     //Create palette interface to DirectDraw using the palette.
     if (FAILED(lpdd4->CreatePalette( DDPCAPS_8BIT|
                                              DDPCAPS_ALLOW256|
                                              DDPCAPS_INITIALIZE,
                                              palette,
                                              &lpddpal,
                                              NULL)))
     {
          //See SetDisplayMode for comment.
          error << "Could not create palette.";
          error.close();

          return false; //Quit.
     }

     //Prepare the primary surface to draw on.
     //Microsoft recommends clearing out the structure DDSURFACEDESC2.
     memset(&ddsd,0, sizeof(ddsd));
     //Fill in the size of structure.
     ddsd.dwSize = sizeof(ddsd); //Reciever of pointer will know the
size.

     //Fill in valid datafields in ddsd structure.
     ddsd.dwFlags = DDSD_CAPS;
     //Set as primary surface.
     ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

     //Create primary surface to draw on.
     if(FAILED(lpdd4->CreateSurface(&ddsd, &lpddsprimary, NULL)))
     {
          //See SetDisplayMode for comment.
          error << "Could not create primary suface.";
          error.close();

          return false; //Quit.
     }

     //Attach palette to the primary surface.
```

```
     if (FAILED(lpddsprimary->SetPalette(lpddpal)))
     {
          //See SetDisplayMode for comment.
          error << "Could not attach palette to primary surface.";
          error.close();

          return false; //Quit.
     }

     //NOW WE ARE READY TO PLOT PIXELS!!!

     //ddslock will hold the result of lock function. See class
declaration.
     //Always clear structure memory of DDSURFACEDESC2.
     memset(&ddsd, 0, sizeof(ddsdlock));
     //Always set size of structure.
     ddsdlock.dwSize = sizeof(ddsdlock);

     //Lock the surface to draw on.
     if(FAILED(lpddsprimary->Lock(    NULL,
     //No sub-area of surface.
                                          &ddsdlock,
          //Save result here.
                                          DDLOCK_SURFACEMEMORYPTR|
     //Poiter to top left.
                                          DDLOCK_WAIT,
     //Try until success.
                                          NULL )))
     //Advanced, not used.
     {
          //See SetDisplayMode for comment.
          error << "Could not lock surface.";
          error.close();

          return false; //Quit.
     }


     //Aliases for cleaner code. See class declaration of DirectX for
details.
     mempitch = ddsdlock.lPitch;
```

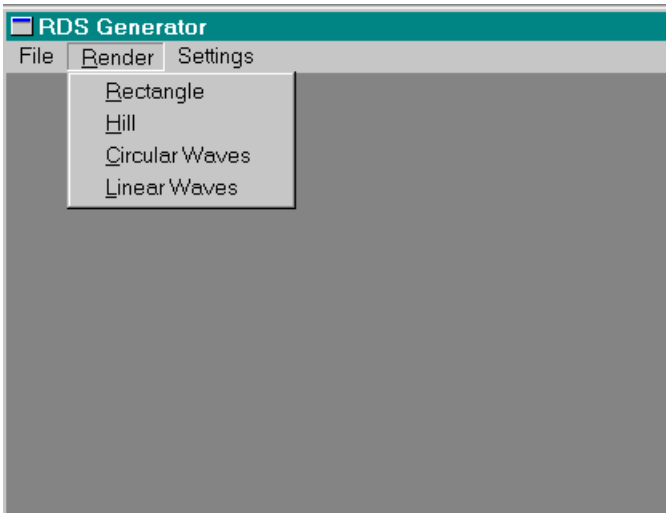
```
     video_buffer = (UCHAR *)ddsdlock.lpSurface;

     //User can now extract mempitch and video_buffer using the public
     //methods, and manipulate the screen coordinates x,y directly
using
     //the syntax: video_buffer[x+y*mempitch] = plotcolor;

     //Set background color
     for( int y = 0; y < SCREEN_HEIGHT-1; y++ )
          for( int x = 0; x < SCREEN_WIDTH-1; x++ )
               video_buffer[x+y*mempitch] = BACKGROUND_COLOR;

     return true;  //Init OK, return succsess!

}//End initDirectX()

//Shuts down DirectX and returns control to standard GUI.
void DirectX::closeDirectX()
{
     //Unlock surface
     if(FAILED(lpddsprimary->Unlock(NULL)))     //NULL for entire
screen.
     {
          //Exit application, if screen cannot be unlocked. Fatal
error.
          PostMessage(hwnd,WM_DESTROY, 0, 0);

     }

     //========================================== Shutdown entire
directX
     //Release objects in reverse order of creation.
     //First reset cooperation level to make
     //application window non-always-on-top.
     if (FAILED(lpdd4->SetCooperativeLevel(    hwnd,
                                                DDSCL_NORMAL  )))
     {
          MessageBox(NULL, "Could not reset cooperation level.",
"Error",0);
          return;   //Quit.
     }
```

```
     //Maximize Window again after DirectX took over.
     ShowWindow(hwnd, SW_MAXIMIZE);

     //Release the palette.
     if (lpddpal)
     {
          lpddpal->Release();
          lpddpal = NULL;
     }

     //Release the primary surface.
     if (lpddsprimary)
     {
          lpddsprimary->Release();
          lpddsprimary = NULL;
     }

     //Release the DirectDraw interface.
     if (lpdd4)
     {
          lpdd4->Release();
          lpdd4 = NULL;
     }

}//End closeDirectX()

//Returns memPitch.
int  DirectX::getMemPitch()
{
     return mempitch;
}

//Returns video_buffer.
UCHAR* DirectX::getVideo_buffer()
{
     return video_buffer;
}
```

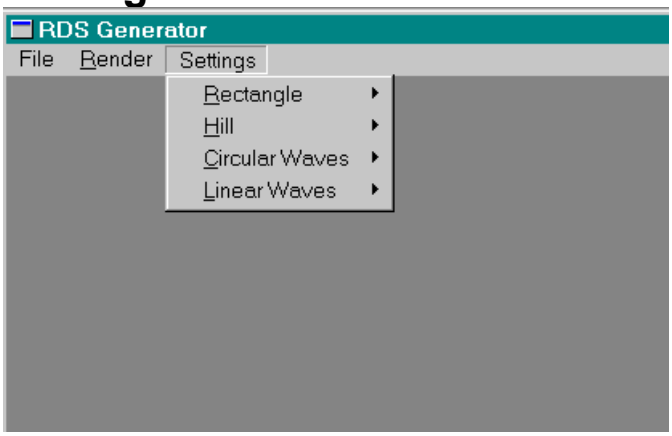
Appendix B Screen shots of GUI

Main

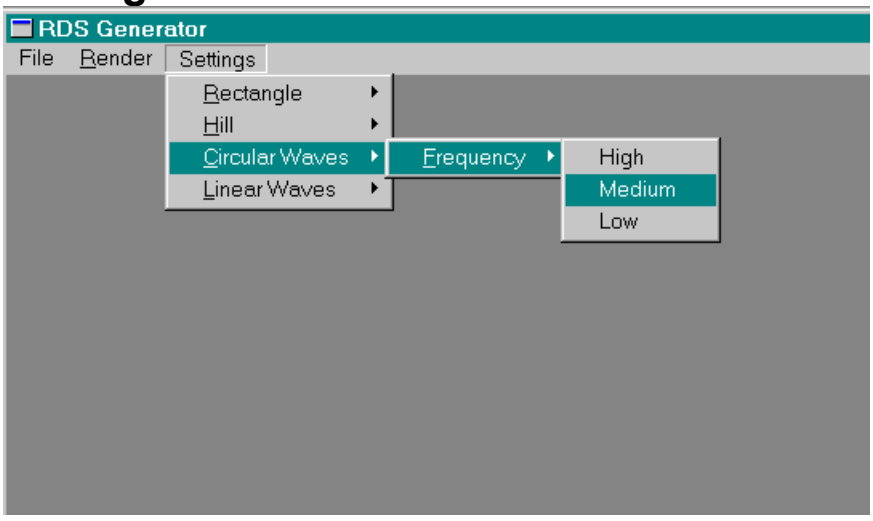
Render menu



Settings menu



Settings menu detail



Appendix C Development Log

2003-03-17 Created skeleton application successfully.

2003-03-18 Created time spreadsheet, documented skeleton application, created system architecture, started accessing base DirectX interfaces.

2003-03-19 Continued trying to access base DirectX interfaces. Application complied , but did not link. Missed external identifier. Manually configured the compiler to import **ddraw.lib** by adding it to the existing libraries to be linked in, solved the problem. Installed latest version of DirectX, 9 to get latest interfaces and performance. Compiler did not find `IID_IDirectDraw4` identifier. Found solution on Google by searching the error message phrase. Added **dxguid.lib** to be imported, since this library tells the linker what `IID_IDirectDrawXXXX` etc means.

2003-03-22 Worked on colors and palettes in DirectDraw. Began accessing the screen using DirectX to be able to plot pixels on it. Succeeded in plotting pixels on the screen. Windows immediately repainted the windows with default background color.

2003-03-24 Studied the architecture of the Windows GUI, the Windows messaging system and application resources. Developed an interface prototype with menus. Succeeded in creating menus, compiling them and attaching them to the window. Symbols go in resource_header.h, menu definitions go in resource_script.rc and are compiled into the .EXE, accessible at all times.
Also implemented the WM_COMMAND to catch choices on the menus.

Split the application on multiple source files. The linker links the multiple files together automatically, no own code necessary to make it work, just one header with prototypes. Put #define statements with screen width etc. in external include file, as need to be included in the multiple source files, e.g. SCREEN_WIDTH is used in two files after the splitting.

2003-03-25 Began creating RDS Engine component. Notes: Member functions are defined outside the class to avoid others from seeing the function implementations.
Analyzed the mathematics to generate 3D shapes. Designed class Shape, which holds the function surface - the only thing that defines a shape. Rectangles etc. will inherit this function and extend with parameters from user. Thought about function pointers and virtual functions, to extract a function from an object.

2003-03-26 Realized that it is not necessary with virtual functions and function pointers. Better to hide everything concerned with shapes inside the object. The only thing the RDS Engine needs is a z-value from (x,y) pairs. The rest - insertion of parameters, optimizing of calculations etc. - is hidden inside the object, and I just call the shape's function that returns a z-value. Thought about having a shape ID to know which shape is coming to the RDS Engine component, but it is not necessary either. Again, the only thing the RDS Engine needs is a z-value!
Worked on inheritance. Collect all shape class declarations in **RDSshapes.h** and all member function implementations in **RDSshapes.cpp -** ALWAYS implement the constructor for an inherited

object, otherwise compilation error. RDS Engine interface OK. Defined to receive a Shape object, but this can be replaced by a RDSRectangle object and it still works the same! NOTE: Rectangle is a reserved word in C++.

2003-03-28 Sending a RDSRectangle still called base-class function render(). Studied virtual functions again and realized that that is what I needed after all. Virtual functions use "dynamic binding" which is the key to calling a base-class but getting the right function depending on inherited object, e.g. I send a Rectangle. It is received as a Shape. I call the rectangles method getZvalue() SEEN AS A SHAPE. I still arrive in the Rectangle's overridden render() method. THIS IS WHAT I NEED! Now I have one single interface between GUI and RDSEngine. No matter what I send to RDSEngine, that Shape's getZvalue is called. I don't even have to know what object it is. The alternative is to use an ID in each inherited Shape an switch it, and then cast to that inherited object's pointer type, and use it in a static manner. Problem: when I add an extra inherited shape in the future, I have to extend the switch statement. EXTREMELY ERROR-PRONE. With dynamic binding and virtual classes I don't have to do anything in the RDSEngine. It can handle anything I send to it, as long as it is a Shape and has the method getZvalue() to extract the 3D surface's Z-value.

2003-03-29 Defined transformation from mathematical coordinations in the virtual 3D space to pixels on screen. Defined formula for rectangle using the member variables for side and height in the RDSRectangle object. Object is passed correctly to the render function. Had problems printing an integer in a messagebox on screen for testing. How to cast an integer to a char?

2003-03-31 Printed testresults in a textfile. Code:

```
ofstream filehandle( "filename", ios::out );
filehandle << "text";
filehandle.close();
```

Made the 3D Engine algorithm, and implemented 3 stages. Had problems with float random numbers. Solution:

```
float      x      =      (float)      (rand()%(X_MAX*100+1));
x = x / 100;
```

I have to cast on the fly, as above.

Created new classes for 3D and 2D points, and Lines. Made methods for manipulating lines in 3D space. Used constants for all values in constants.h.

Also made a transformation algorithm from mathematical points to pixels on screen. I now have the interception of the first line with the screen, and will plot it using DirectX next time.

2003-04-02 Tried to restructure the directX calls into classes, hwnd needs to be global.

2003-04-03 Made hwnd global, works OK. Maybe it is possible to extract it from msg?

2003-04-04 Big problems with passing directX interfaces as parameters when I break down the directX init sequence into separate functions. Decide to start over from when the init is OK in a sequence. Worked more on cooperation levels after directX is finished and I return to GUI. **When I release the primary surface, the screen is back to standard Windows colors again**. Also released the interfacs in reverse order of creation. Before that I set the cooperation level to normal, since otherwise the application window is always on top. Works fine now! All I want to do now is to restructure this

sequence into memberfunctions of a class called DirectX, and make it do the same thing. Found the command to maximize a window. `ShowWindow(hwnd, SW_MAXIMIZE);`
Finished restructuring directX calls into a class. ddsd structures can not be initialized in constructor since they depend on other variables being set first. Took variable for variable up to member variable status, systematically, with success.

2003-04-04 Began implementing the RDS algorithm.

2003-04-05 I have to define functions using Shapes like this: render(Shape &rect) WITH the '&', otherwise the dynamic binding will not work.
Moved eye positions into member variables. Realized that the valid Y-area from which to randomize, is NOT MIN_Y to MAX_Y, but the virtual Y-area that goes beyond the screen. See documentation.

2003-04-07 Tried removing dubble-vision on planes, without success. I tried everything, with constants, no effect. High resolution no difference. Some levels are natural and look fine.

2003-04-08 Tried more on dubble-vision problem, with more decimals in both x and y. IMPOSSIBLE. There seem to be a limit as to the correctness in the RDS-technology when using discrete surface. Print on paper maybe is better.
Quit trying, move on to new shapes. Tolerance 0.1 is NOT enough. 0.001 looks great!
Created three shapes and added help point functionality.
Tried recalculating the pixels to compensate after plotting, a big improvement!

2003-04-12 Made possible user settings, optimized the interception algoritm, commented and cleaned up code.

2003-04-16 Cleaned code, had forgotten to optimize nextRightPoint! Tremendous improvement of performance! Commented code.

2003-04-17 Continued cleaning & commenting code.

2003-05-18 Dissertation finished!