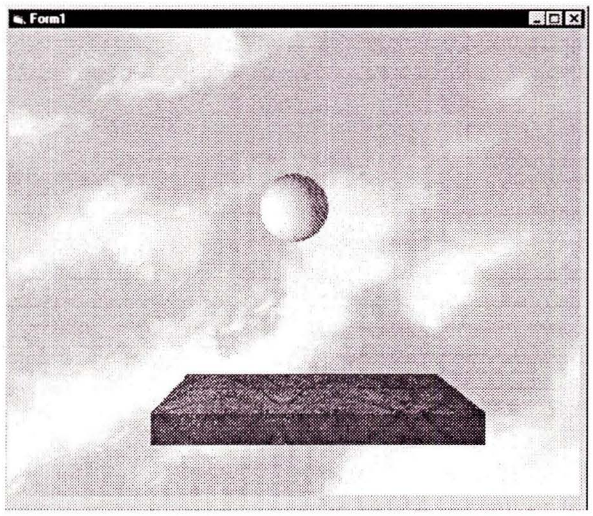
IPO, Center for Research on User-System Interaction
PO Box 513, 5600 MB Eindhoven

25.02.1999

Rapport no. 1201

NUKE - Multi-modal user interface software components

H.F. Moll
M.C. Boschman
J.R. de Pijper
M. v.d. Voort



NUKE - Multi-modal User Interface Software components

Combining DirectX and ActiveX in UI components for the rapid construction of multi-modal interfaces

H.F. Moll, M.C. Boschman, J.R. de Pijper, M. v.d. Voort

This report is released in a slightly different format as Nat.Lab. Technical Note 067/99 of Philips Research.

Authors' address data: H.F. Moll, Philips Natlab, WLp; hfmoll@natlab.research.philips.com

M. C. Boschman, TUE, IPO; m.c.boschman@tue.nl

J.R. de Pijper, TUE, IPO; j.r.d.pijper@tue.nl

M. v.d. Voort, Fontys Hogeschool Eindhoven

Title: NUKE - Multi-modal User Interface Software components

Combining DirectX and ActiveX in UI components for the rapid construction of multi-modal interfaces

Author(s): H.F. Moll, M.C. Boschman, J.R. de Pijper, M. v.d. Voort

Part of project: User interface Software Exploration (NUKE), UCD1

Customer: Philips Research

Keywords: User interfaces, software components, rapid prototyping, multi-modal interfaces, DirectX, ActiveX, COM

Abstract: This report describes software components we constructed in the NUKE project based on Microsoft's DirectX technology. They can be used for rapid construction of multi-modal interfaces.

The report starts with an overview of both COM, the chosen SW component technology and of DirectX. After this overview the specific components that we built are described. We conclude with a section describing the use of these components in various applications.

The NUKE project was a co-operation in 1998 between Philips Research (USIT group) and the IPO to explore the potential of standard software technology for multi-modal interfaces. The created software components are available both at the IPO and the USIT group via their web pages.

Conclusions:

- SW components have the future, also for User Interfaces
- "Beyond the GUI" interfaces become reality (3D, speech, sound, touch on the PC and at home)
- Packaging DirectX in ActiveX components allows rapid prototyping of multi-modal interfaces.

Contents

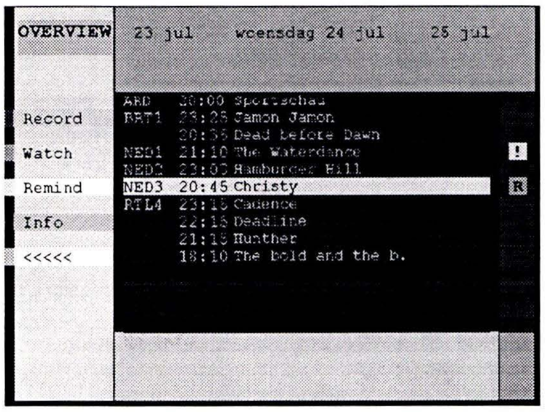
1 Introduction

This document describes the user interface software components that have been created in the NUKE project [1], in collaboration between the IPO and Philips Research. In this project we explored how to use the technical possibilities of Microsoft's DirectX APIs to create a set of easy to use UI components that offer possibilities that go beyond the standard GUI and allow the rapid construction of multi-modal interfaces. Multi-modal interfaces are interfaces that communicate with the user using more then one modality at the same time. An example of a modality is the use of speech for user input, or the use of 2D graphical images for output to the user. The set of UI SW components we developed supports the modalities from DirextX: sound, 3D animated graphics and touch (force feedback).

This document gives an overview of the used technologies (Microsoft COM, ActiveX and DirectX) and a short comparison with other component technologies. After this overview, the UI SW components we created and their usage in several demonstrators and projects are covered. Finally some conclusions and expectations of the future are given. The remainder of this chapter gives the motivations for doing this work.

1.1 Relevance

User interfaces of many (Philips) consumer electronic products are controlled by software. Examples include televisions or set-top-boxes that offer an electronic program guide (EPG) that can replace the paper program guide or a GSM mobile phone with a large menu tree of facilities (see Figure 1 and Figure 2).



```
ARD   20:00 Sportschau
BRT1  23:25 Jamon Jamon
      20:35 Dead before Dawn
NED1  21:10 The Waterdance
NED2  23:05 Hamburger Hill
NED3  20:45 Christy
RTL4  23:15 Cadence
      22:15 Deadline
      21:15 Hunther
      18:10 The bold and the b.
```

Figure 1 a simple EPG

Figure 2 a GSM phone

Such products are growing in complexity and are getting more difficult to use. One approach to this problem is to reduce the set of features, but that also might reduce the competitiveness of the product. Another solution is to make the set of features easy to use by the same technology that made them possible in the first place: embedded software. Currently most software development efforts are just aiming at increasing the set of features. The usability and the user interface of these features are often the last thing being considered. Supporting more natural ways of interaction by using other I/O modalities then just displaying graphics for output to the user and using keys to receive input from the user is one way to improve the usability. In this way the interaction between user and product becomes more natural because it is more similar to the ways humans are used to communicate with each other (using all their senses). See e.g. [2], [3].

1.2 Standard software platforms

The general goal of the NUKE project was to explore and extend the possibilities of user interfaces that go beyond the GUI. We aimed at doing this on standard computer platforms and within standard SW architectures.

A secondary goal derived from this was to build up knowledge and experience with state of the art UI products and tools with a strong focus on SW aspects.

Standard software platforms of today include:

- a PC with an Intel Pentium processor and Microsoft Windows 9X or NT
- a Web browser supporting HTML, JavaScript and Java
- a RISC workstation running a Unix OS and an X-Windows interface

Standard SW platforms are also likely to be used more and more for consumer application (Web Browsing on your TV, Windows CE in your GSM or PDA) and available processing power and memory is growing according to Moore's law, also for CE products.

In NUKE we focused on the PC platform. The PC platform sets the direction for innovations and is still gaining market share, both at the high-end (Windows NT as application-server) and at the low-end (Windows CE in PDAs and cars). We also expect a crossover from PC platform technologies to consumer products, both for hardware (processors, disks, memory modules) and software (Microsoft Windows, applications).

Even on the PC platform alone we see a variety of SW tools, methods and standards with respect to building applications and creating user interfaces. We focused on de-facto standards on this platform:

- COM is the standard SW component technology on the PC
- ActiveX is the standard for creating interactive components in a networked PC environment (Internet, Client/Server, Desktop applications)
- DirectX is the standard set of multimedia APIs on the PC to create high performance games and is also finding usage for other domains (Web, professional applications)

With these choices made we studied the following aspects:

- SW model, SW architecture, SW components - how can we create well-designed applications using these standards?
- Platform limits - what are the memory requirements of using these standards and when do we hit a performance limit?
- Usability in other contexts - what are the options of using these standards in embedded products?
- Comparison to alternatives - how do these standards compare to alternative solutions, both within and outside Philips?

2 COM, ActiveX and other SW Component Technology

2.1 SW component technology

In the early days of computers programmers used to write their applications starting from scratch and they compiled it into one monolithic part of binary code. Applications were dedicated to the programmer's specific problem and re-use was not invented yet. Clearly this was labor intensive and inflexible. When computer platform and programming languages became more and more standardized (Unix, DOS, Mac), programmers began re-using each other's code. This led to the use of SW libraries, which saved effort and increased flexibility. In fact the use of a standard OS was the first form of re-use. SW libraries solved a big part of the problem, but they introduced a version and maintenance problem. New versions of libraries made old applications crash and new applications did not work on old libraries. With the advent of Object Oriented software methods this resulted in using binary SW components with rigid interfaces as a way to build an application from smaller parts. Advantages of this method compared to writing monolithic applications are:

- Applications can be updated after shipping
  If a component contains a bug, only the component needs to be repaired and replaced. See component E in Figure 3.
- Applications can be customized
  If a user has a different implementation of a component, offering the same functionality in a different way, it can replace the component in the application. An example is a different editor component. See "vi" and "emacs" in Figure 3.
- Applications can be developed rapidly
  Ideally an application can be build rapidly by taking components offering required functionality from a component library and gluing them together with a few lines of code and little effort.

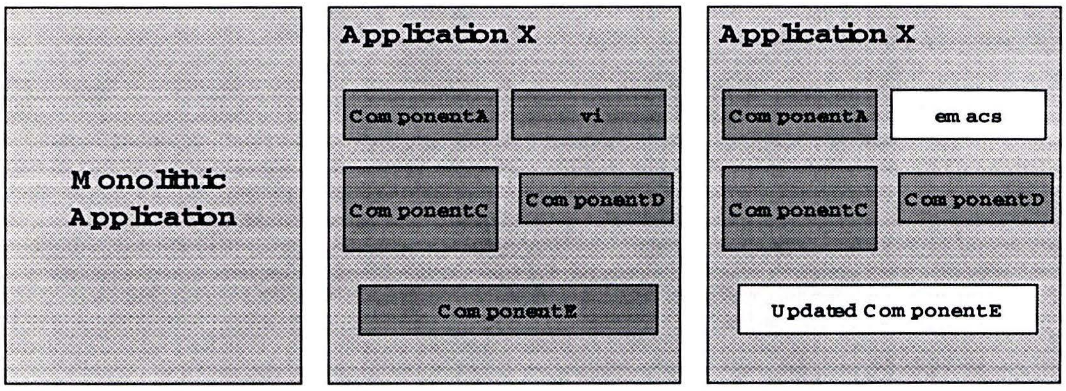


Figure 3 Applications and Components

2.1.1 Requirements on components

In order for software components to really achieve these advantages, we can state the following requirements on SW components:

1. Software components can dynamically plug into and unplug from an application. With this we mean dynamic run-time linking in stead of static, compile time linking. Without this property the users of a component need to recompile and link their applications when a component is changed, which is clearly undesirable (it releases the source code, asks for distribution of tools, etc.).
2. A component should hide all implementation details. This property is also called encapsulation. An important aspect of this, is the used programming language. There are many programming languages and new ones are being developed all the time. We want to use components from the language we chose for our applications and we want to write components in any language. This implies a binary format for components.

   Another solution is a standard "header file" format and mapping functions to every programming language. Corba's IDL is an example of this. See 2.4.2.
3. Components need to be upgradable without breaking existing applications. A new and updated version of a component can replace a previous one without requiring action from the user of the component. This requires downward compatibility of components. Later we will see what this means for the interfaces of a component.
4. Components need to be relocatable (on a network). This requirement hides a specific implementation detail and offers an additional advantage. SW applications and their constituent components can be split in a client and a server part and thus hide their actual location for the user. It also allows components to work across the Internet or a local network.
5. Components should only have explicit dependencies. Using a component implies using other software components that implement interfaces it depends on. We want to know on forehand if a component will work in our context. This programming by contract is not supported by COM, but is supported by KOALA. See 2.4.3.

2.2 What is COM?

COM stands for Common Object Model. It is a Microsoft standard for components and how an application can use them. In more technical terms COM components are:

- WIN32 DLLs or executables (binaries) and hence programming language independent
- upgradable with backward compatibility, by versioning of interfaces
- relocatable on a network, through a so-called proxy mechanism

COM provides the component management functions to find, create and use components and also provides the network communication code to implement the proxy mechanism. COM is a way of writing componentised programs.

In the sequel we will cover the most important conceptual aspects of COM and show how COM meets the requirements we posed on SW components. We will not give a detailed description of how to build COM components. A good text for learning that is [4].

2.2.1 Encapsulation: COM Interfaces

In COM a component is defined by its interfaces. An interface is a set of (related) functions. These interfaces are the only way to get access to the functionality provided by the component. The COM notation for a component A with two interfaces is shown in Figure 4.

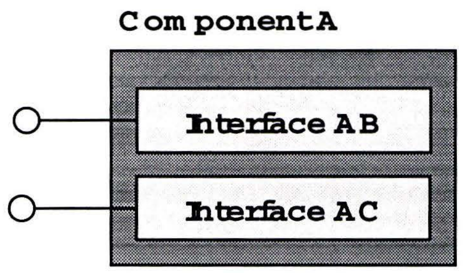


Figure 4 COMponent notation

Writing the code for such a component in C++ could be done as follows:

```
#define interface struct
interface IAB {
  virtual void __stdcall Fx1()=0;
  virtual void __stdcall Fx2()=0;
};
interface IAC {
  virtual void __stdcall Fy1()=0;
  virtual void __stdcall Fy2()=0;
};
```

This shows one way to code a COM component in a programming language. However this coding of components in a programming language is NOT standardized. Only the resulting binary format is standardized. Later on we will see that this format matches easily with the code a C++ compiler generates for a "struct" or a "class".

In COM interfaces support in fact both the access to a component and its encapsulation. In COM interfaces are first class citizens. See Figure 5.

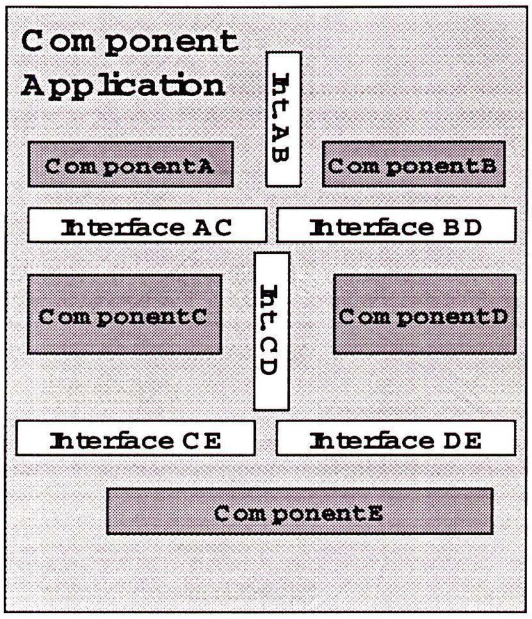


Figure 5 Components and Interfaces

2.2.2 Getting to know COM: IUnknown

In COM one interface is mandatory for every component. This interface is called IUnknown (see Figure 6) and offers functions to get to the other interfaces of the component and for lifetime management of the component. The functions of the IUnknown interfaces are:

- QueryInterface(IID &iid) – this function is a gateway to get to know (the interfaces of) the component. QueryInterface asks the component if it supports another interface identified by a unique identifier 'iid'. It returns a handle to the wanted interface. This handle can be used to call the functions of that interface.
- AddRef(), Release() – these functions offer lifetime management of the component. AddRef keeps track of the users of the component and Release() removes the caller as a user of the component. With zero users a component can release its resources (memory, network links, file-locks, etc.).

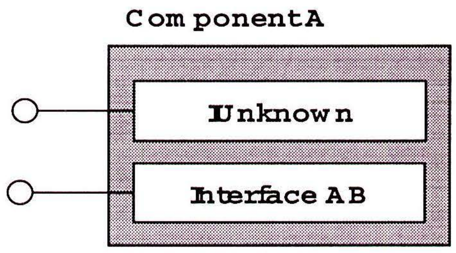


Figure 6 a COM Component

One other requirement of COM is that each interface implement these functions and hence offers a gateway to all other interfaces. In C++ this can be achieved by inheriting from IUnknown.

2.2.3 Under the hood

Implementing the binary format required by COM in C++ is straightforward. We illustrate this with a usage scenario. The user of a component A (which is called the client in COM) creates an instance of the interface (IAB). In Figure 7 the client uses a variable pA to store this instance. This variable in fact points to the class instance data. The class instance structure contains a pointer to a virtual function table (vtbl) as it's first member. This table contains pointers to the functions wherever they are in memory.

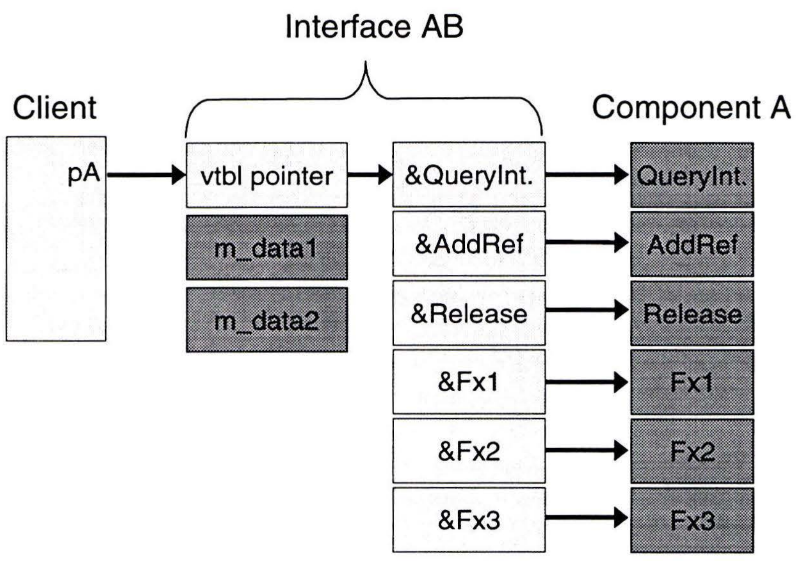


Figure 7 Under the hood

In Figure 8 we show how the implementation of the IUnknown functions in an arbitrary interface can be realized. COM standardizes the way in which the datastructures for an interface are formatted and how they are stored in memory. Also the way parameters are passed is standardized by COM.

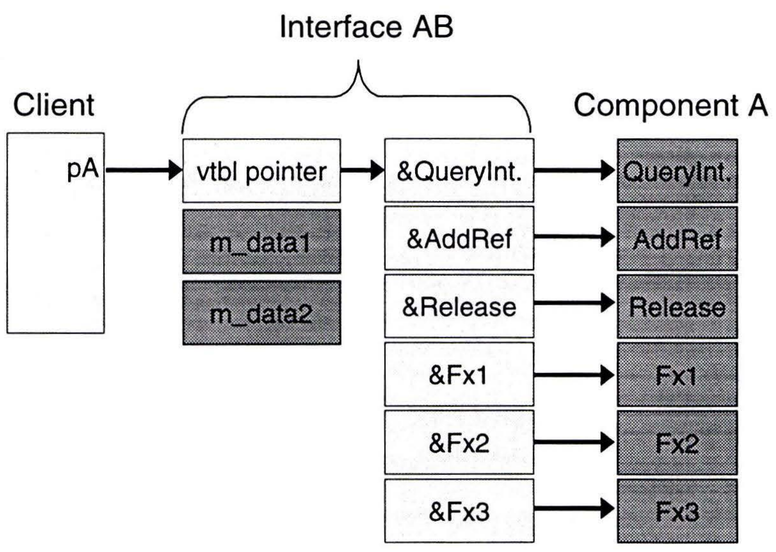


Figure 8 IUnknown implementation

2.2.4 Further decoupling - GUIDs, Idispatch and Type libraries

The next issue we consider is how a client creates a component in a way that fulfills the requirements of 2.1.1.

- The first solution that can be considered is to use a CreateInstance() function from a library. This however does imply static linking in some form and hence is not a good solution with respect to our requirements.
- A second possible solution is to call CreateInstance() in a DLL. This is better, but has the disadvantage that the name and location of the DLL has to be known in the code, which hampers customization and upgrading of components and hence is not good enough.
- The COM solution is to supply a parameterized function in the COM library. This function is CoCreateInstance (prototype: CoCreateInstance (CLSID &clsid, IID &iid)) and takes a component and an interface identifier as parameters. These identifiers are so-called GUIDs, globally unique identifiers. GUIDs define interfaces (IIDs) and components (CLSIDs) uniquely. They are 128 bits long, contain a 60 bit time-stamp, and a 48 bit system identification. They are stored in the PCs Registry. The Registry keeps track of the server (DLL or EXE) that implements the component interface.
- If even more flexibility is needed COM supplies a mechanism to call component interfaces by name (strings). This is implemented in the IDispatch interface and is the basis for Automation. Automation supports the use of components in scripting languages (Visual Basic, VB Script, Java Script). Together with IDispatch we mention "type libraries", a language independent binary equivalent of header files, that allows direct interface calls in a safe and fast (compared to IDispatch) way. Automation is covered in a bit more detail in the next chapter.

2.2.5 COM in summary

With this we conclude the overview of COM. Further details can be found in many books and on-line manuals.

In summary, COM provides a standard way to build binary software components and it provides the framework and management functions to use them in a way that hides implementation details, upgradability and location transparency.

2.3 ActiveX COMponents

ActiveX is a set of technologies that uses the COM to enable software components to interact with one another in a networked environment, regardless of the language in which they were created. ActiveX technologies can be used to create applications to run on the desktop or the Internet.

ActiveX includes both client and server technologies, including the following:

- ActiveX Controls are interactive objects, which can be used in containers such as a Web site.
- ActiveX Documents enable users to view documents, such as Microsoft Excel or

Word files, in the entire client area of a Web browser or other ActiveX container.

- Active Scripting controls the integrated behavior of several ActiveX Controls and/or Java programs from a browser or server.

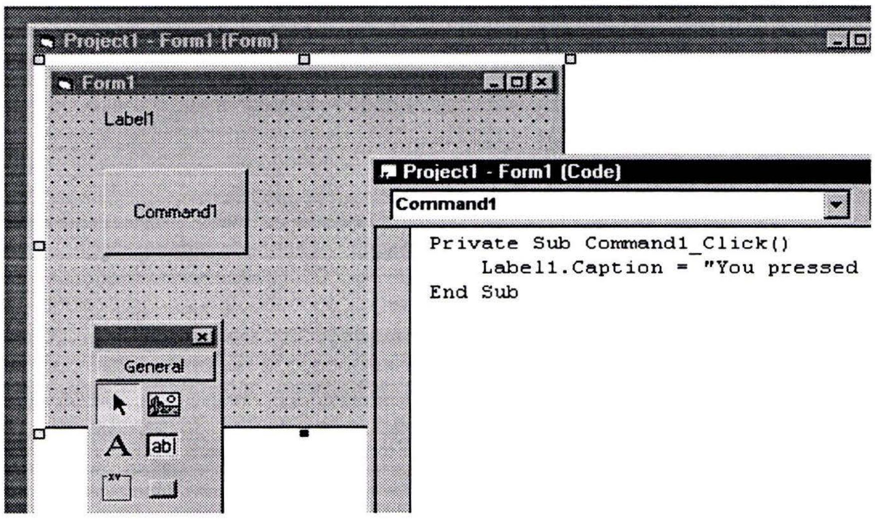
A bit of history:

In the past everything based on COM was called OLE. Now OLE is restricted to technologies (interfaces) that allow users to create and edit documents containing items or "objects" created by multiple applications. OLE was originally an acronym for Object Linking and Embedding. However, it is now referred to simply as OLE. Parts of the old OLE that are not uniquely related to linking and embedding are now part of ActiveX. This includes Automation. Automation (formerly known as OLE Automation) makes it possible for one application to manipulate objects implemented in another application, or to "expose" objects so they can be manipulated. Automation is used in OLE as well as in ActiveX.

In this part we cover ActiveX Controls and their containers, the part of ActiveX that covers UI components.

2.3.1 ActiveX Controls Overview

ActiveX controls are COM components that can be used as building blocks for user interfaces. They can be used with drag and drop in many programming environments for easy, fast programming or prototyping. They support event driven programming, support scripting and can be published on the Web. An example of a Button control in the Visual Basic programming environment is shown in Figure 9.



```
Private Sub Command1_Click()
    Label1.Caption = "You pressed
End Sub
```

Figure 9 Button Control

Let's look at ActiveX controls in more detail. ActiveX is a name for a set of technologies and services, all based on COM. ActiveX controls (formerly called OLE controls) are COM objects with a set of interfaces that lets them behave like controls for Windows. They can be used in visual development environments like Visual Basic and Delphi and can even be included in HTML pages. From the programmers' point of view an ActiveX control consists of a visual representation and has properties, methods and events.

- *Properties* can be set to certain values and/or their values can be read.
- Via the *methods* parts of the actual functionality of the control can be activated.
- The control can send *events* to the outside world to indicate that certain occurrences happened.

An application that uses ActiveX controls is called a container. The container application embeds the visual representation and contains event handlers to properly act upon the events fired by the control. See Figure 10.

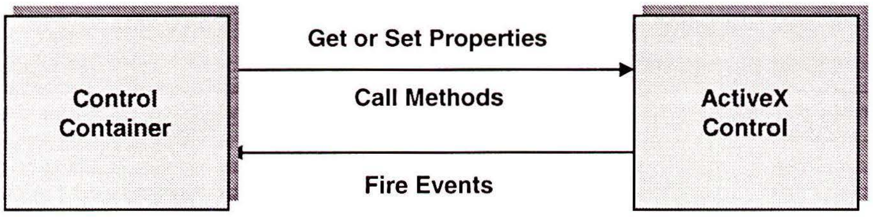


Figure 10 control container communication

2.3.2 Building ActiveX Components

ActiveX controls can be created with Microsoft Visual C++, with Microsoft Visual Basic, or with another programming environment. For this project we used Visual Basic and Visual C/C++. Visual C++ has a ControlWizard that sets up the basic framework for the control by creating a *Control Class*. Via the AppWizard properties, events and methods can be added to the control. It sets up a framework for the get and/or set functions, the event handlers and the methods that define the behavior of the control. These frameworks just need to be filled-in by the programmer. In Figure 11 we show how this works. The wizard generates the skeleton for your control class with the proper calls of the generic COleControl parent class. COleControl implements all the required COM interfaces for an ActiveX control.

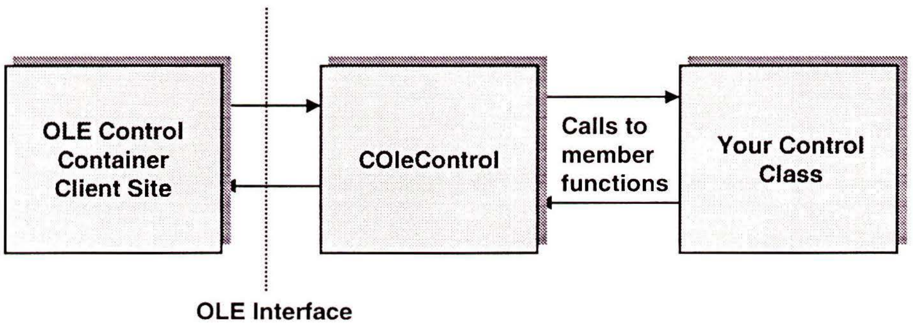


Figure 11 MFC Ole Control Framework

Without a wizard a lot of work would have to be done manually, including implementing all COM interfaces required by a control. The ControlWizard generates code that is based on MFC, the Microsoft Foundation Classes, a set of C++ classes that implements generic Windows programming constructs. Besides MFC Visual C/C++ offers ATL, the Active Template Library, a set of C++ templates for Windows programming. ATL has a template for building controls. Other environments that support the construction of controls include Visual Basic, Borland Delphi, and Borland C Builder. However it is still a far from perfect world. When the environment does not support something directly still a lot of work needs to be done manually and the environment is often more a handicap than a help.

2.3.3 Control Containers

The counter part of the control component is the control container. An ActiveX control container is a container that fully supports ActiveX controls and can incorporate them into its own windows or dialogs.

The ActiveX control container interacts with the control via exposed methods and properties. The embedded ActiveX control can also interact with the container by firing (sending) events to notify the container that an action has occurred. The control container can choose to act upon these notifications or not.

The control container uses (OLE) Automation to access the objects exposed by the server (of the control) and their (dispatch) interfaces, see 2.2.4. Examples of control containers are Internet Explorer (3.0 and higher), Visual Basic forms, Microsoft Word, Visual C/C++ dialogues, Visual Studio's test container, Borland Delphi and Builder, etc. These containers all support the basics of embedding controls. However they differ in the way they invoke exposed interfaces, both in the order and the exact usage of methods. To make sure that a component works in all a container, it has to be tested there.

One of the controls we created, the 3D *Scene*, is also a container. However, implementing a control container from scratch is a lot of hard work and we were happy to use the support offered by the Visual Basic environment, which provides a framework to implement a container too.

2.3.4 Under the hood of OLE, Automation, Controls and Containers

An ActiveX control is an automation server, a COM component that implements the IDispatch interface. IDispatch is one COM interface that allows a component to offer many services. For scripting languages IDispatch offers a way to use COM components without parsing header files. Functions of the component can be called by name. See the Figure 12.

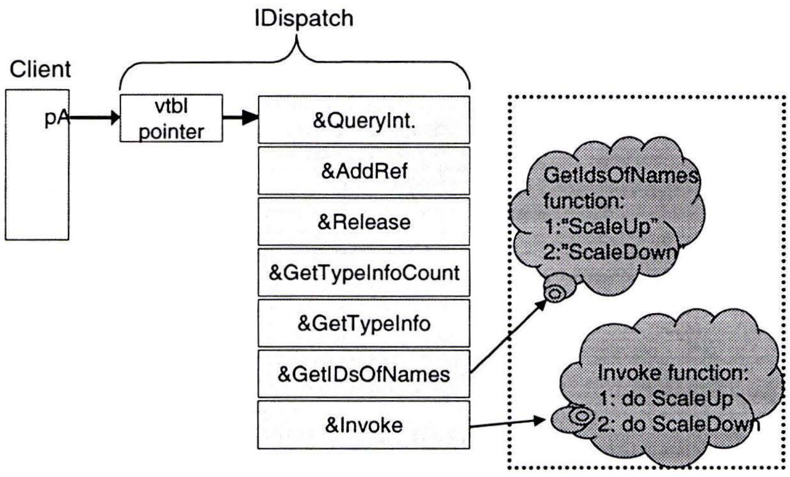


Figure 12 IDispatch

IDispatch offers a function GetIdsOfNames to translate strings (function names) into dispatch identifiers (DISPIDS) that can be used to call a generic Invoke functions, with a generic parameter array. This is a very flexible way of offering all functionality via a single standard interface, but because of this genericity it is expensive (slow). The modern way of working is to provide a dual interface, i.e. support for both IDispatch and direct "vtbl" access of functions. In figure 12 this would mean that &ScaleUp and &ScaleDown are added to the table.

Besides IDispatch a control has to support much more. In this section we will give a quick overview of the most important Automation and OLE interfaces that are relevant for controls and containers.

For programmability (automation) of a control we have the already discussed IDispatch interface. This allows a container to call methods and to get and set properties in a control. But events go the other way. For this purpose we can use connection points. Connection points offer a generic mechanism for outgoing interfaces and are implemented in the COM IConnectionPoint and IConnectionPointContainer interfaces. These interfaces allow a container to connect event handlers to events fired by the control. In addition to connection points for events a control can communicate changes of (data) properties to its container via the IAdviseSink interface.

Handling of keyboard events, ambient properties and coordinate transforms between control and container is done via IOleControl and IOleControlSite functions.

Storage of controls in files is handled via IPersistStreamInit.

Drag and drop is part of OLE and is done via IDropSource and IDropTarget interfaces.

Displaying a control, data transfer for cut and paste, support for storage of a control in a file is done with IDataObject, IViewObject, and IPersistStorage.

Controls use a lot of the interfaces for compound document management (OLE), such as IOleObject, IOleClientSite, IOleInPlaceActiveObject, IOleInPlaceUIWindow, and IOleInPlaceFrame and IOleInPlaceSite. All these interfaces handle aspects of the interaction between a component (document or control) and it's container (document or ActiveX control container).

Using a framework such as that provided by MFC a lot of interfaces need not be implemented again for each new control. In MFC a control you create is derived from the COleControl class which provides a default implementation for all control interfaces.

For more information on controls, COM and OLE we refer to [4] and [5].

2.4 Other SW Component Technology

There are alternatives for COM. However we think that on the PC platform Microsoft's COM is here to stay and will broaden its scope (COM, DCOM, COM+). Disadvantages of COM are the fact that it is not an open standard and only well supported on the Windows PC platform. Clear advantage is its language independence. Support for COM on other platforms is slowly becoming available. See [6]. Below we will cover two alternatives superficially.

2.4.1 Java and Java Beans

One important alternative comes from Sun, with the Java programming language. Java Beans are Java components that are similar to ActiveX controls in many ways. They can also be distributed over the Web and they run on any Java virtual machine. This makes them programming language specific, but platform independent. It remains to be seen what technology will win on the Internet. Java still has a small disadvantage with respect to performance, but offers a cleaner programming model. However to access the specific media processing capabilities of a computer platform, so-called native modules have to be written in another language and be wrapped in a Java class.

2.4.2 CORBA

Another alternative we mention here is CORBA, the Common Object Request Broker Architecture, which is similar to COM in many ways, but intended for networked environments from the start and more targeted towards business applications. The ORB, Object Request Broker, takes the place of the COM infrastructure. Main advantages of CORBA are the fact that it is an open standard (from OMG, the Object Management Group) and that it is supported on many platforms for enterprise applications. Various parties have announced that CORBA will be interoperable with COM in the future. CORBA is language independent and is in many aspects similar to COM. Interworking between COM and CORBA is possible. See [7].

2.4.3 KOALA

Koala is a Philips Research method to model SW components. It exceeds COM in the sense that it models the interfaces provided by a component as well as the interfaces required by a component. This allows design time determination of what is needed to use a component.

With a *requires* interface a component can also provide an interface to support diversity. An example of this is a component that implements the sound features in a TV set with a *requires* interface that selects the type of sound (stereo, Dolby surround, mono, incredible-surround etc.). This selection determines which internal sub-components of the sound component will be linked in the resulting executable.

Koala comes with a graphical notation that allows software systems to be constructed from software components like a 2-dimensional connection puzzle. It gives the user a nice overview of all components in a family of related software systems.

A difference with COM is that KOALA is aimed at embedded software and that dynamic linking and upgrading is not required. Also location transparency is not an issue. This allows the use of a preprocessor (currently for C++) to implement KOALA in a compile-time toolset. One of the main tasks of this processing is the elimination of all dead code, components that are not used in a specific instance of the system.

A good starting point for KOALA is [8] (intern Philips).

3 DirectX

3.1 Introduction – What is DirectX?

DirectX is a set of APIs for high-performance multimedia and game programming. This set was developed by Microsoft to rival MS-DOS and game-console performance in media programming. A second goal was to create a HW independent game and multimedia software platform. Thirdly it helped Microsoft to steer HW development for PCs. It has been a success from the start. Already in 1997 9 of 10 PC games were DirectX based (source: WinHEC'98 conference).

One problem we encountered with DirectX: Microsoft does come with new versions regularly that extend the functionality and does not come with the newest versions on all platforms. E.g. DirectX 5 is not supported on Windows NT 4.0. Future versions are likely to be supported on Windows 98 or NT 5.0 only. The latest version of the DirectX SDK (Software Development Kit) can be obtained from Microsoft for a nominal fee. The device driver libraries can be downloaded for free. In this project we used DirectX 5.

DirectX is COM based, but in a minimalistic way. In principle a programmer creates a single COM object for each HW resource you use in your application (display adapter, sound board, joystick, etc.). After that you call this object's methods to create derived objects (sound buffers, display surfaces, force effects, etc.). DirectX even comes with a special implementation of CoCreateInstance to make this easier.

DirectX (version 5) is split in two sets of APIs: DirectX Foundation and DirectX Media (Figure 13).

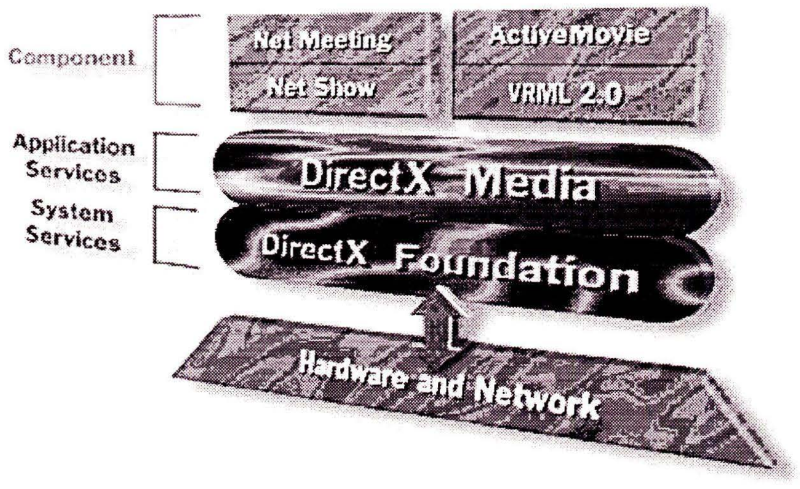


Figure 13 DirectX Overview

DirectX Foundation (Figure 14) gives access to basic media hardware. It includes the following interfaces:

- DirectDraw, gives direct access to display devices, offers fast blitting, overlay surfaces, z-buffer support, stretching, transparency, palette mapping

- DirectSound, supports low-latency mixing of (3D) sounds, hardware acceleration and direct access to the sound device
- Direct3D (immediate mode), gives access to a complete 3D rendering pipeline
- DirectInput, gives fast access to user input devices, including force feedback.

All DirectX Foundation interfaces offer a so-called hardware abstraction later (HAL). This HAL hides the details of the specific hardware for the user and offers generic standardized functionality. If the actual hardware cannot support these functions, they can be emulated. This emulation is done in a so-called hardware emulation layer (HEL). An example of this architecture for DirectSound is shown in Figure 16.

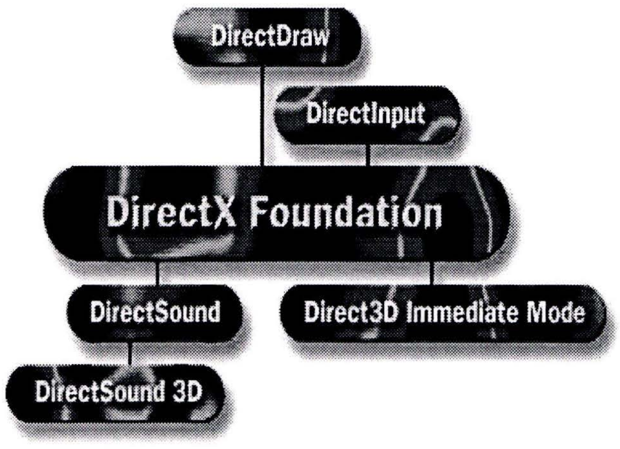


Figure 14 DirectX Foundation

DirectX Media (Figure 15) offers high-level services based on DirectX Foundation or other basic functionality. It covers:

- DirectPlay – a standard way to support multi-player games over a communication infrastructure
- Direct3D Retained Mode – an easy to use 3D drawing operations (model is kept in buffer)
- DirectAnimation - unified animation API for SW controlled animations on the Internet
- DirectShow - media-streaming architecture for video in windows
- DirectModel – support for large 3D models including level of detail
- VRML - support for VRML on top of DirectX

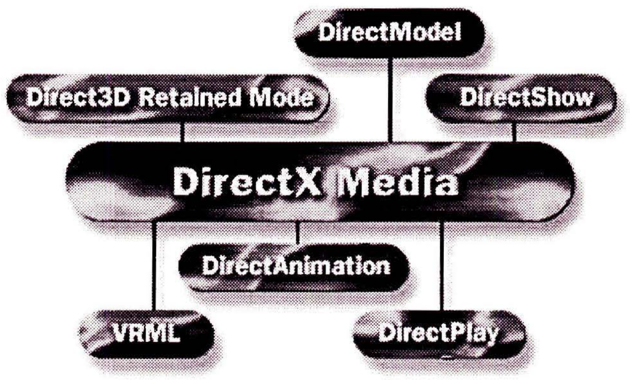


Figure 15 DirectX Media

In the sequel we cover the DirectX Foundation interfaces in more detail. We did not explore DirectX Media, since we wanted to understand the basis first.

3.2 DirectSound

3.2.1 Introduction

The Microsoft DirectSound application programming interface (API) is the audio component of DirectX. DirectSound provides low-latency mixing (i.e., very small delay times), hardware acceleration, and direct access to the sound device. It provides this functionality while maintaining compatibility with existing device drivers.

Like other components of DirectX, DirectSound allows you to use the hardware in the most efficient way possible while insulating you from the specific details of that hardware with a device-independent interface. Applications will work well with the simplest audio hardware but will also take advantage of the special features of cards and drivers that have been enhanced for use with DirectSound.

Here are some other things that DirectSound makes easy:

- Querying hardware capabilities at run time to determine the best solution for any given personal computer configuration
- Using property sets so that new hardware capabilities can be exploited even when they are not directly supported by DirectSound. Note that this goes against the philosophy of using a hardware abstraction layer to use any type of sound hardware. Apparently programmers did not learn enough from the DOS past.
- Low-latency mixing of audio streams for rapid response
- Generating 3-D sound
- Capturing sound

DirectSound implements a new model for playing and capturing digital sound samples and mixing sample sources. As with other object classes in the DirectX API, DirectSound uses the hardware to its greatest advantage whenever possible, and it emulates hardware features in software when the feature is not present in the hardware.

DirectSound accesses the sound hardware through the DirectSound hardware-abstraction layer (HAL), an interface that is implemented by the audio-device driver.

The DirectSound HAL provides the following functionality:

- Acquires and releases control of the audio hardware
- Describes the capabilities of the audio hardware
- Performs the specified operation when hardware is available
- Causes the operation request to report failure when hardware is unavailable

The device driver does not perform any software emulation; it simply reports the capabilities of the hardware to DirectSound and passes requests from DirectSound to the hardware. If the hardware cannot perform a requested operation, the device driver reports failure of the request and DirectSound emulates the operation.

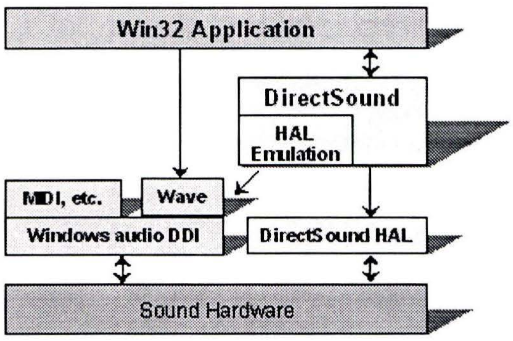


Figure 16: The relationship between DirectSound and Windows sound functions

An application can use DirectSound as long as the DirectX run-time files are present on the user's system. If the sound hardware does not have an installed DirectSound driver, DirectSound uses its hardware-emulation layer (HEL), which employs the Windows multimedia waveform-audio (**waveIn** and **waveOut**) functions. Most DirectSound features are still available through the HEL, but of course hardware acceleration is not possible. DirectSound automatically takes advantage of accelerated sound hardware, including hardware mixing and hardware sound-buffer memory.

DirectSound playback is built on the **IDirectSound** component object model (COM) interface and on other interfaces for manipulating sound buffers and 3-D effects. These interfaces are **IDirectSoundBuffer**, **IDirectSound3DBuffer**, and **IDirectSound3DListener**. DirectSound capture is based on the **IDirectSoundCapture** and **IDirectSoundCaptureBuffer** COM interfaces.

Another COM interface, **IKsPropertySet**, provides methods that allow applications to take advantage of extended capabilities of sound cards.

Finally, the **IDirectSoundNotify** interface is used to signal events when playback or capture has reached a certain point in the buffer.

3.2.2 Playback Overview

The jrSoundControl ActiveX control developed during this project (see 6.1) encapsulates only components of DirectSound relating to playback. Therefore, this section will go into DirectSound playback in a little more detail.

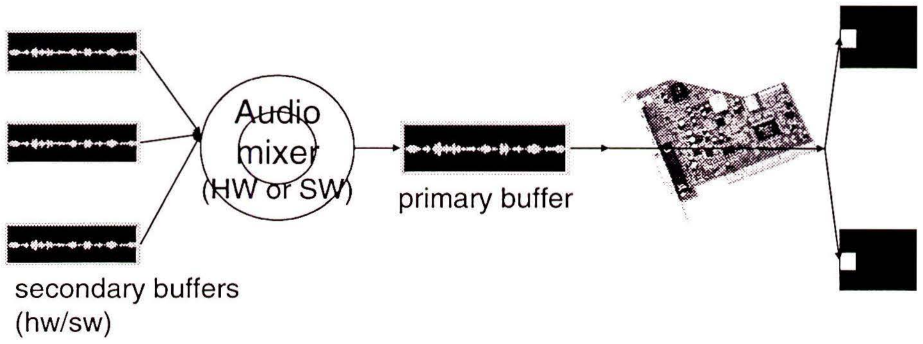


Figure 17 DirectSound playback overview

To use DirectSound, first a so-called DirectSound object (derived from the **IDirectSound** interface) has to be created. This object represents the sound card itself.

Sound samples in the form of pulse code modulation (PCM) data are contained in so-called *sound buffers* (objects derived from **IDirectSoundBuffer** or **IDirectSound3DBuffer**). Individual sounds are kept in *secondary buffers*, which are used to start, stop, and pause sound playback, as well as to set attributes such as frequency and format. Secondary buffers may be either *static* or *streaming*. In the case of static buffers the entire sound is kept in memory and this is good for short sounds. With streaming buffers the sound data is transferred into the buffer a block at a time, a good strategy for longer sounds.

When secondary buffers are played, DirectSound takes the data from each buffer and mixes it into the *primary buffer* (of which there is only one in an application). If the sound format (sampling rate, bits per sample, mono or stereo) of a secondary sound buffer differs from the format set for the primary buffer, the necessary conversion is performed automatically. From the primary buffer, sounds are sent to the output device (sound card) to be made audible.

DirectSound does not include functions for parsing a sound file. It is the programmer's responsibility to stream data in the correct format into the secondary sound buffers.

Depending on the card type, DirectSound buffers can exist in hardware as on-board RAM, wave-table memory, a direct memory access (DMA) channel, or a virtual buffer (for an I/O port based audio card). Where there is no hardware implementation of a DirectSound buffer, it is emulated in system memory.

Multiple applications can create DirectSound objects for the same sound device. When the input focus changes between applications, the audio output automatically switches from one application's streams to another's. As a result, applications do not have to repeatedly play and stop their buffers when the input focus changes.

Through the **IDirectSoundNotify** interface, DirectSound provides a mechanism to notify the client when the play cursor reaches positions within a buffer that have been specified by the client, or when playback has stopped. This mechanism can be especially useful to determine when fresh sound data can be safely transferred into a streaming secondary sound buffer.

3.3 DirectDraw & Direct3D

3.3.1 DirectDraw

Direct draw is the part of DirectX to manage 2D graphics. For this it usually uses two or more buffers. Buffer one holds the graphics on the screen itself and is called the front buffer. The second buffer that is used to build up the graphical image is called the back buffer. When a frame is completely build the front buffer is flipped with the back buffer to display the new image. You can also have additional buffers with graphics data, if these are needed. These off screen surfaces can be used for triple buffering, masks, sprites, textures etc.

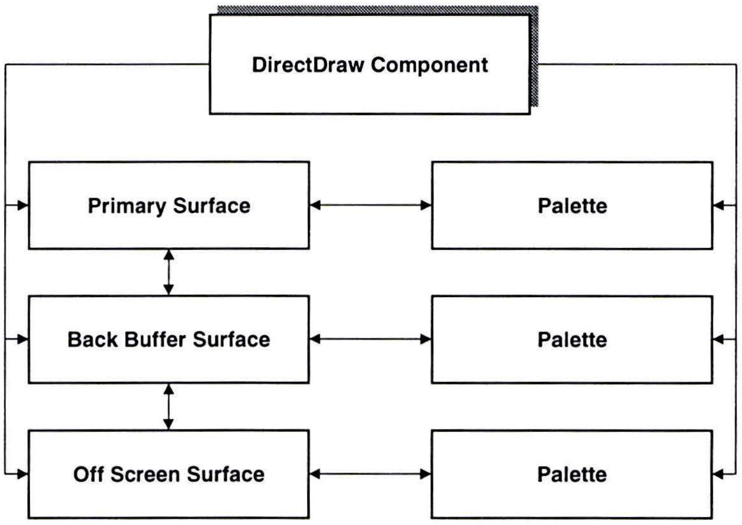


Figure 18 Layout of the buffers

DirectDraw is a DirectX component that allows you to directly manipulate display memory, the hardware blitter, hardware overlay support, and flipping surface support. DirectDraw provides this functionality while maintaining compatibility with existing Windows-based applications and device drivers.

DirectDraw is a software interface that provides direct access to display devices while maintaining compatibility with the Windows graphics device interface (GDI). It is not a high-level application programming interface (API) for graphics. Because it maintains a direct contact with the hardware, the user is more responsible for the good execution of a program.

The interface allows an application to enumerate the capabilities of the underlying hard-ware and then use any supported hardware-accelerated features. Features that are not implemented in hardware are emulated by DirectX (HEL).

3.3.2 Direct3D

Direct3D is divided into 2 main parts namely:

- Immediate Mode, which gives complete access to the rendering pipeline
- Retained Mode, a high level API on top of immediate mode to make programming easier

Direct3D provides the user with a way to determine the type of 3D hardware that is present in a system and see what functionality it has. During the initialization a user must also create a device so DirectX will know what drivers to use. After this initialization the user can create 3D objects and add them to the 3D world. These objects are placed in a tree so each object can have a parent and one or more children.

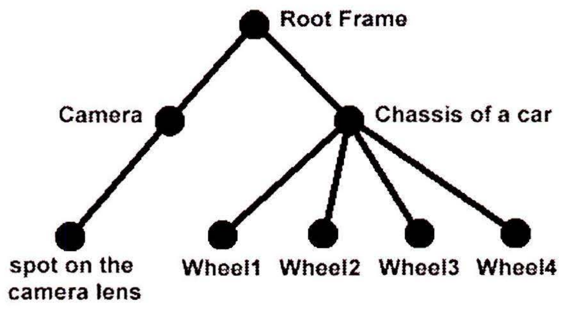


Figure 19 The Direct3D object tree

In the case above you can move the car and it's four wheels forward by simply moving the chassis. This is because the wheels are children of the chassis.

With Direct3D you can also manage your viewport, this is the visible area in your 3D world. This way you can also change the depth of your view. Direct3D is closely linked to DirectDraw, meaning you can use these 2 parts of DirectX together to have 2D and 3D graphics on one screen.

Direct3D will help you with the construction of a 3D world, by letting you:

- Position objects
- Scale objects
- Rotate objects
- Load objects from file
- Change the look of your objects (texture and color)

Once you have completely constructed your 3D world Direct3D can render it to the screen for you so you will see the 3D world on your 2D screen. To make *things* even more realistic you can also manipulate the camera in your 3D world so you can view it from all possible angles. There are a lot of effects you can use when rendering your 3D world so your worlds can range from ugly and fast to very beautiful and slow. Some of the rendering possibilities are:

- Rendering of the objects, do you only want to place the points or the lines of the objects or do you want to render the object completely solid.

- Effects of lights, do you want to give each polygon in your object a single color or do you want to calculate a shading and give each pixel a different color.

3.3.3 The Interfaces

DirectDraw is built on the component object model (COM). You create an object that links to the hardware and then call it's methods to control it. The main object you need to create is IdirectDraw (you always need this interface). After that you can use the other interfaces of DirectDraw. The most important interfaces are:

- IDirectDrawClipper. With this interface you can set the visible part of a surface, so the graphics is automatically clipped when it reaches the edges.
- IDirectDrawSurface. This interface is used to create the different surfaces to blit the graphics on. The front- and back buffer uses in DirectDraw are both surfaces.

Direct3D is also based on COM (all parts of DirectX are) but has a lot more interfaces to control it. The main interface you will need to use Direct3D is IDirect3D. When this interface is created you can use one of the following interfaces for the different aspects of 3D (only the most important ones are given):

- IDirect3DRM, the interface for retained mode. This is an extra interface on top of the standard 3D interface to make the 3D programming less complex.
- IDirect3Ddevice, the interface that holds the information about the driver you want to use to display the graphics.
- IDirect3Dviewport. This interface controls the dimensions of the visible part of the 3D world.
- IDirect3Dframe, which is part of the Direct3D tree object structure. This interface is used to hold information about the 3D objects in your world.
- IDirect3DTexture, the interface to work with texture that can be mapped onto the 3D objects.
- IDirect3DMaterial, to give the 3D object some kind of material (E.G. a shine)

3.4 DirectInput

This part of DirectX (see [9]) provides support for various kinds of input devices, including mice, keyboards, joysticks, game pads, flight yokes, VR headgear. It also provides support for force-feedback devices, like the Microsoft SideWinder FF Pro joystick, which actually are input/output devices.

The main advantage of DirectInput is, like other DirectX components, that it gives faster access to input data by communicating directly with hardware drivers. By doing so, a lot of Windows overhead is bypassed.

Like other DirectX components DirectInput is COM-based. The DirectInput architecture consists of a DirectInput object and for each input device a DirectInputDevice object. Each device has its own object instances representing buttons, axes, keys etc.. Force feedback effects are represented by DirectInputEffect objects. The COM objects and their interfaces are depicted in Figure 20

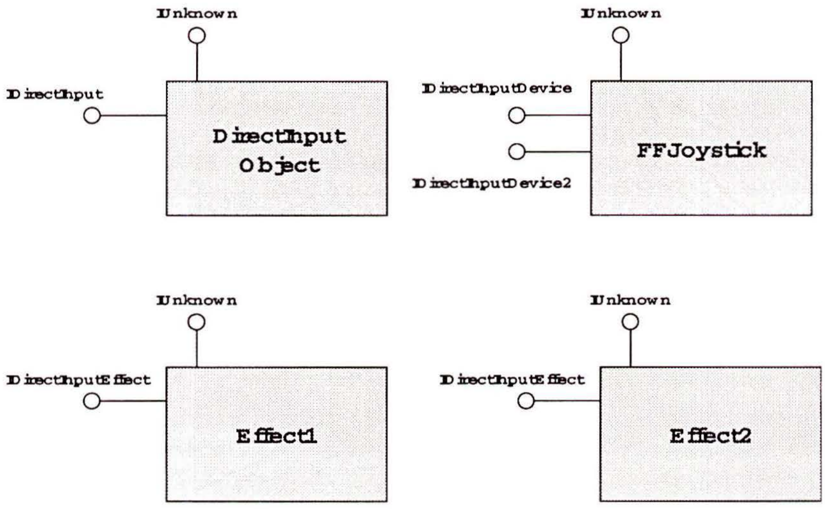


Figure 20 DirectInput COM objects and their interfaces

The DirectInput object has an IDirectInput COM interface that is obtained by calling the **DirectInputCreate** function. The input-devices that are available on your system can be enumerated by the method **IDirectInput::EnumDevices**. The wanted DirectInputDevice objects can be created by **IDirectInput::CreateDevice, which** returns a pointer to an IDirectInputDevice interface. The object instances of the device are enumerated with the **IDirectInputDevice::EnumObjects** method. This method calls a callback function from which a structure is made available for each enumerated object instance.

DirectInput supports two ways of getting data from an input device: polling and event notification. The IDirectInputDevice offers a **GetDeviceData** and a **SetEventNotification** function for this.

The force feedback functionality was not supported by the original IDirectInputDevice interface. A second COM interface to the DirectInputDevice object was added: IDirectInputDevice2. A pointer to this interface is obtained via the **IDirectInputDevice::QueryInterface** method. The force effect objects for force feedback devices are created with the **IDirectInputDevice2::CreateEffect** method. These effect objects themselves are also equipped with COM interfaces via which they can be controlled (**IDirectInputEffect::Start**, **IDirectInputEffect::Stop**, etc...).

4 DirectX + ActiveX = Multi-modal UI components

In this chapter we give a description of the functionality of each of the ActiveX controls we constructed to make use of the media processing functions offered by DirectX. Together these components allow the rapid construction of applications with multi-modal user interfaces.

4.1 JrSoundControl

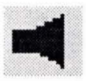


Figure 21 jrSoundControl icon

The jrSoundControl ActiveX control is designed to make it easy to play wave files in a Microsoft Windows program, or any environment capable of ActiveX integration. The control gives the programmer all the advantages of DirectSound, but completely shields him from the complexities of the DirectX architecture. Also, the control takes care of accessing and loading sound data from wave files, a nontrivial affair, which DirectSound provides no support for.

The advantages of using this control rather than the standard Windows multimedia functions are many. Any number of wave files can be played at the same time. These files may have completely different sound formats in terms of sampling frequency, bits per sample and number of channels (mono/stereo): Conversion and mixing is done automatically. Playback of any of the sounds can be stopped and resumed or restarted at any point, without any noticeable delays. Various aspects of each sound, notably volume, playback frequency and left-right balance can be varied at will, even during playback. Each sound can be made to play only once, or continuously, in a loop, and there is no limit to the lengths of the sounds to be played. All this is achieved with modest memory needs (longer files are "streamed" rather than read into memory as a whole) and low-latency playback: typically, playback will start within 10 ms of the command being given.

Using the control in a program is very simple. The programmer need not know anything about either wave files (other than that they contain sound) or DirectX, the underlying technology used. The whole process is completely transparent.

4.1.1 Using jrSoundControl

ActiveX controls can expose methods, properties and events to the programmer using the control. The jrSoundControl control exposes only methods, eleven to be exact. A complete list of the available methods, with a short description of each, is given in the Appendix. All method names start with the characters "jr". This is done so that in programming environments that employ the "intellisense" technique (e.g., Visual Basic and Visual C++ version 6), the methods are neatly grouped together in the popup box from which a method can be selected.

In its simplest form, only three statements are needed in a C++ program to play a wav file using jrSoundControl: one to declare a variable for the sound, one to initialize the variable and get the wav file ready for playing, and one to actually play the sound:

```
CjrSoundControl sndBeep;                // Declare variable sndBeep
sndBeep.jrInitSound ("beep.wav");       // Initialize
sndBeep.jrPlaySoundSingle();            // Play sndBeep once
```

After the first two statements -declaration and initialization- any of the available methods can be used. The various "set" methods may also be used during playback and will then take effect immediately.

When the sound is no longer needed, destroy it:

```
sndBeep.jrDestroySound();               // Give back memory
```

If more than sound is needed, simply declare CjrSoundControl variables for each sound to be played. Each sound can then be played, stopped or changed independently of the others.

That's all there is to it.

4.1.2 Remarks on jrSoundControl

- As explained earlier, static sound buffers are suitable for short sounds, while streaming buffers are typically used for longer sounds. jrSoundControl is programmed to create static buffers for wav files shorter than 2 seconds and streaming buffers otherwise. A next version of the control will leave this choice to the user.
- jrSoundControl does not support multiple sound cards: The preferred audio device on the system is always used for playback. This is the audio device selected by the user through the Multimedia applet in the Control Panel.
- jrSoundControl always makes the sound format of the primary buffer equal to the sound format of the last secondary buffer initialized. This format may thus change during the life of an application. Due to a bug in DirectSound, audible distortion may occur when sound data is converted to another format when it is mixed from its secondary buffer into the primary buffer. Best results are therefore obtained when all sounds used in an application have the same sound format to start with, in which case no conversion is necessary at all.
- jrSoundControl does not support 3D sound.

4.2 Direct3D Controls - Scene and Thing

The Direct3D controls consist of 2 parts. The first part is the Direct3D container. This part renders the 3D world for you, and is usually only needed once. This container can contain Direct3D *things* which represents the objects and lights in the world. You can have a number of these *things* inside your container. For each *thing* you can set some information about what it presents in your 3D world by manipulating the properties of the *thing*. The container then evaluates these properties and constructs the 3D world accordingly.

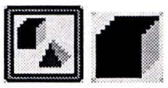


Figure 22 Scene and Thing icons

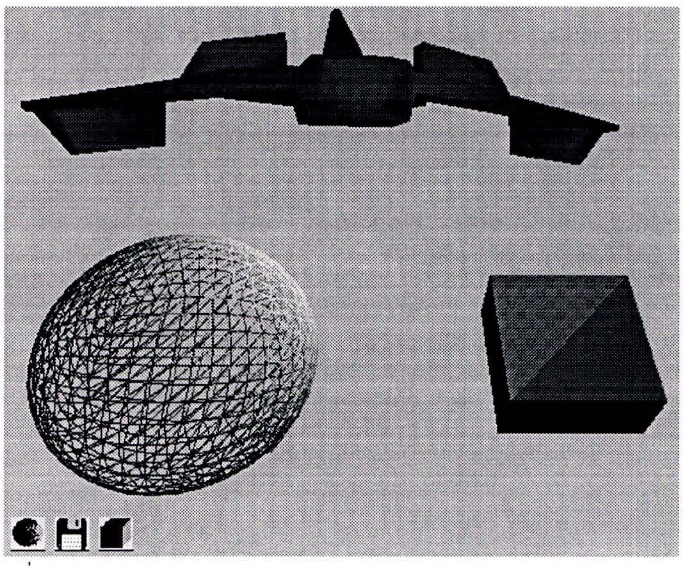


Figure 23 Container with things

In the picture above, the grey area represents the container. The 3 icons at the bottom left represent the *things* that can be placed inside this container. They all have a picture that identifies their object. The container reads the properties of these *things* to construct a 3D world, which you can also see displayed inside the container.

4.2.1 Direct3D Scene

The *scene* is the container part of the controls. It is responsible for the rendering of the 3D world. Some things you can manage with the *scene* are:

- The position and angle of the camera
- A fog effect
- A background color or picture
- Dithering effects
- Rendering speed
- Rendering types (Points, wireframe, solid)
- Light types (flat, gouraud, phong)
- Clipping planes

The container has another added feature, namely it is able to render your 3D world at design time. What this means is, that if you place a *thing* inside it a design time you will immediately see that *thing* appear inside your 3D world. Also whenever you manipulate a *things* properties, you will immediately see the changes in your 3D world. With this feature it is easy to construct simple 3D world because you get immediate feedback. It isn't necessary to run you program every time you changed some*thing*, to see if you made the correct changes. This type of interface is a lot more instinctive to a user.

4.2.2 Direct3D Thing

The name *thing* was give to this component because it can represent 2 *things* in a 3D world. First of all you can use a *thing* to create an object (like a cube or a car) inside your 3D world and you can also use a *thing* to represent a light source inside your 3D world. This light source is represented by a diamond shaped object at design time to give the user an idea of the place of the light source. At runtime this diamond is not visible.

Things you can manage with the *thing* controls are:

- Type of the *thing* (object or light source)
- Type of object, there are some standard objects like a cube, cone, sphere, etc. or you can load an object from file
- Type of rendering for the thing (points, wireframe, solid)
- Type of shading for the thing (flat, gouraud, phong)
- Color and texture of an object
- The position of the object
- The rotation angles of the object
- The scaling factor of the object

If you want more information about Direct3D or the *scene* and *thing* components, there are 2 manuals available. First of all there is a user manual that explains how to use the *scene* and *thing* components to create 3D worlds with them. And there also is a technical manual that explains Direct3D in depth and shows how you can use Direct3D to create a 3D component. See [10] and [11].

For the standard information about Direct3D you can also check out the documentation that is delivered with the DirectX SDK.

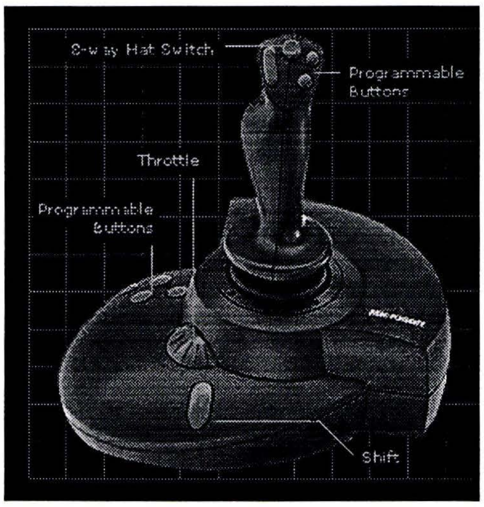
For more information on Direct3D, you can also read [12].

4.3 ActiveX Controls for Microsoft SideWinder FF Pro Joystick

In order to get familiar with the DirectInput API we have built some ActiveX controls and some demo applications for a force feedback input (/output) device. The most generic control, called *SWFF* will be described in this report. The force feedback device we used, e Microsoft SideWinder Force Feedback Pro Joystick [13], needs DirectInput-compliant device drivers.

4.3.1 The joystick

The Microsoft SideWinder FF Pro device (see Figure 24) is equipped with a stick with two axes (X, Y) with programmable force feedback facilities and 1 axis - the rudder or Z-axis - with a passive centering spring. It has 9 buttons, a throttle potentiometer and a point-of-view switch with 8 possible orientations.



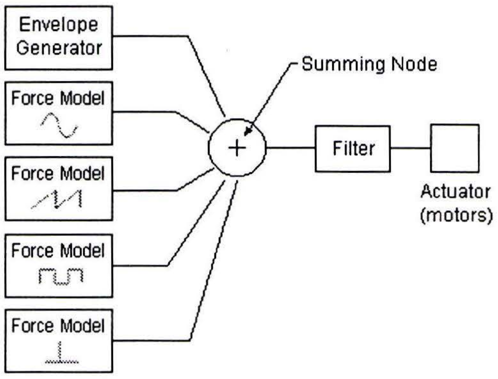


Figure 24 Sidewinder FF Pro Joystick

Figure 25 The force synthesizer

The joystick comes with device drivers that are DirectX 5.0 compliant. An application must create force effect objects prior to using them. Various force effects may be combined as indicated by the force synthesizer scheme in Figure 25. The filter in this picture cuts of amplitudes that are out of range for the device. There are two basic types of force effects: waveform and behavioral effects. A third type of effect, referred to as device-dependent effect, includes both waveform and behavioral effects. The scheme in Figure 26 gives an overview of this classification of effects.

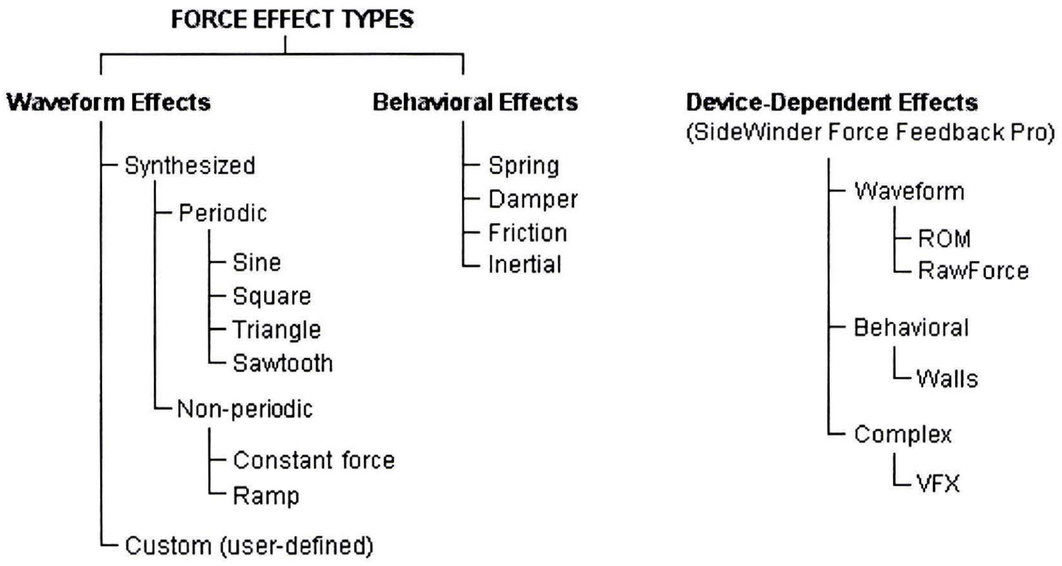


Figure 26 Classification of force effects

Another classification can be made if we distinguish position dependent from position independent effects (see Figure 27). The software developer kit (SDK) contains the Visual

Force Factory software to create force effect (VFX) files. These files are the equivalent of audio wave files and are the result of combining the various force effects into a complex effect.

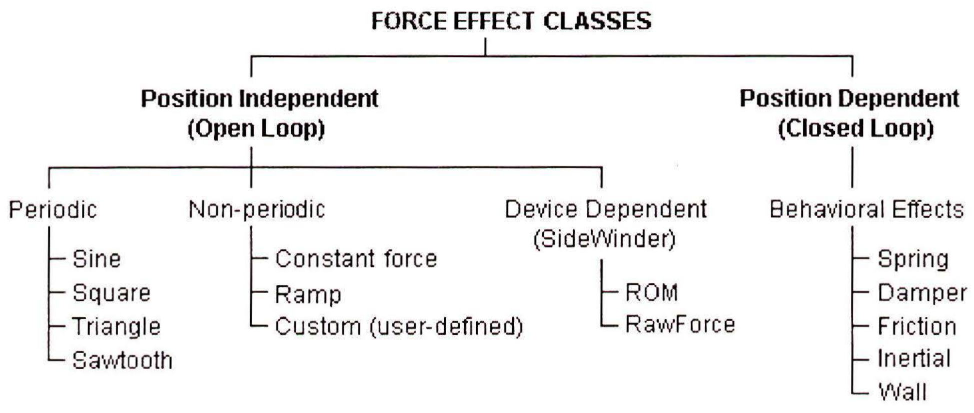


Figure 27 Functional classification of force effects

4.3.2 The SWFF control

This section will give a dense description of the *SWFF* Control, which is an ActiveX component (OCX) that was developed to gain access to most of the functionality of Microsoft's SideWinder Force Feedback Pro joystick. A detailed description will be given in a separate user manual [14].

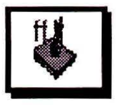


Figure 28 The SWFF control icon

4.3.2.1 How it is built

The *SWFF* control is created with Visual C++ by making use of the ControlWizard to set up the basic framework of the *Control* class and the AppWizard to add a framework for the properties, methods and events of the control. These frameworks are filled-in later to add the actual functionality to the control.

4.3.2.2 What's in it?

The control contains a number of *properties* related to the input part (e.g. the state of the buttons, throttle, rudder, calibration mode, axis mode, internal timer etc.) and the force feedback part (e.g. direction and gain of effects) of the joystick.

The *methods* can be divided into those related to force effects (creation, starting, stopping and destroying of effects), methods to create and destroy borders, a method to poll the XY- position of the stick and the **AboutBox** method to display version and copyright information in a messagebox.

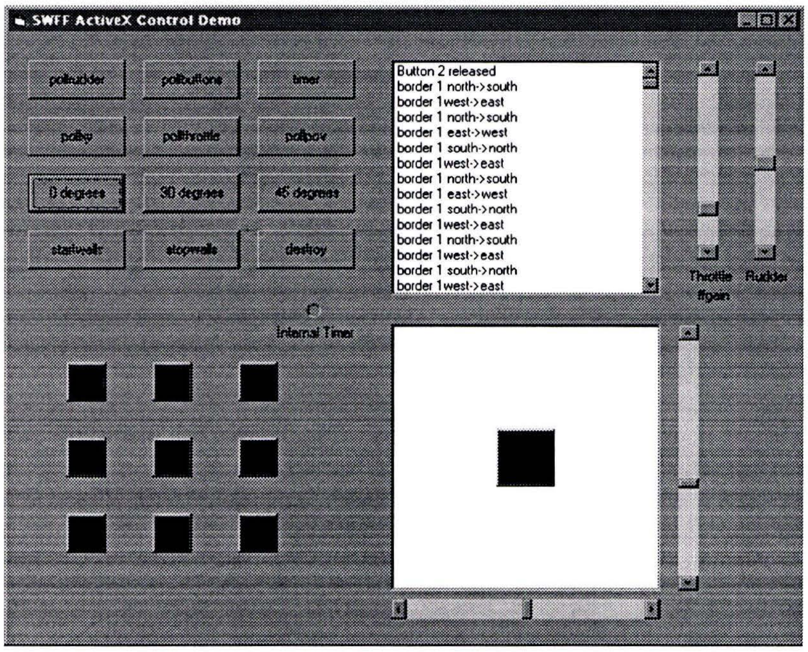
A number of *events* are added to the control. After the control is initialized, the **Initialized** event is fired. There is an event for each pressed (**Button0**, **Button1**...) and each released (**RelButton0**, **RelButton1**...) button. Whenever a border is crossed this is notified by sending events indicate the orientation of the crossing (see below). The control has its own internal timer that is used to poll the position of the stick and the state of the buttons. In order to make this timer available outside the control a **Timer** event is fired after each internal timer lapse.

Point-wise descriptions of all its properties, methods and events are listed in the Appendix.

4.3.2.3 What's a border?

Besides the normal polling option (see the GetPosition method below) we devised an event-based concept for the notification of the position of the stick in its XY-system by using *borders*. A *border* is a virtual straight line-piece in the XY-coordinate system of the joystick defined by its begin-point and end-point coordinates. Whenever the joystick's XY-position crosses one of these borders, event messages (**NS(border_index)**, **SN(border_index)**, **EW(border_index)**, **WE(border_index)**) will be fired to the application's window. The border_index argument identifies which border was crossed.

4.3.3 A demo application for the SWFF Control



```
Button 2 released
border 1 north->south
border 1west->east
border 1 north->south
border 1 east->west
border 1 south->north
border 1west->east
border 1 north->south
border 1 east->west
border 1 south->north
border 1west->east
border 1 north->south
border 1west->east
border 1 south->north
border 1west->east
```

Figure 29 The graphical interface of the SWFF demo application

In order to demonstrate the options of this control we built a Visual Basic application. A screen dump of the graphical interface is shown in Figure 29. It is provided with a number of button controls in order to activate certain methods, e.g. pollrudder reads out the rudder axis and prints it in the scrollable list region right from the buttons. Whenever one or more buttons are pressed the corresponding square indicators at the lower left are highlighted. The area at the lower right corresponds with the XY-system of the joystick. The position of the sliders indicate the actual *x,y* position of the stick. The square in the center of this area corresponds with 4 borders that are defined at the position of the 4 sides of the square. Whenever the joystick crosses these borders events are created which is notified in the scrollable list. The value of the rudder (Z) axis is indicated by the scrollbar at the top right . The Throttle (potentiometer) value is indicated by the other vertical scrollbar. This value also controls the gain of the force effects. All events that are fired from the control are notified in the scrollable list.

5 Using NUKE components

Creating components is one thing, using them in applications that make sense is what proves their usability. To demonstrate the possibilities of our components to create multi-modal user interfaces we created two simple applications ourselves. These will be illustrated in the first part of this chapter and aim to indicate that with the right software components sophisticated interfaces can be build rapidly. Our components were also used in four other research projects, either to add a new modality or to speed up the implementation. This is illustrated in the second part of this chapter.

5.1 Simple demo applications

5.1.1 Calendar-Wheels

The first demonstrator we created was the so-called Calendar-Wheels demo. This demonstration program used our DirectSound and DirectInput ActiveX controls in conjunction with a speech synthesis library and a first component used to explore DirectDraw, the Selection-Wheel. A screenshot is given in Figure 30.

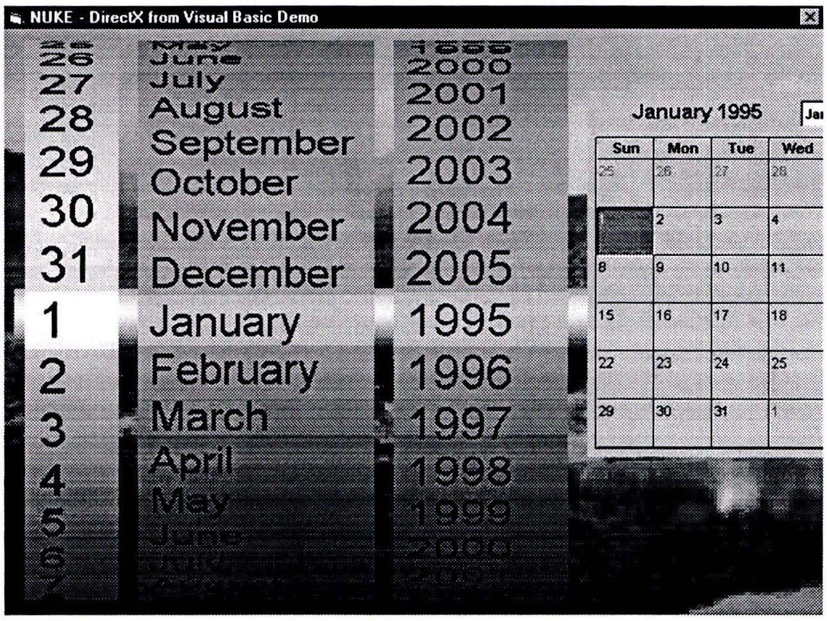


Figure 30 Calendar-Wheels demo

This demonstrator allowed turning the selection wheels for date, month and year by moving the joystick. Setting a wheel in motion by moving the joystick forward or backward caused both a sound and a force effect to be played by the application. Pressing the fire button triggered speech synthesis of the selected date. This demonstrator served the purpose of demonstrating DirectSound and DirectInput possibilities, in conjunction with rapid application development with ActiveX controls.

5.1.2 Ball-game - demo/tutorial on using the NUKE controls

The ball-game is our second demonstrator. It served the purpose of demonstrating during a presentation how easy advanced multi-modal applications can be made. The sequel of this part follows the steps taken to build the application yourself.

5.1.2.1 Step 1 - a simple Scene and Thing

The first thing to do is to start up VB. We assume you have installed the Nuke Controls, including VBD3D OCX, the Direct3D controls. This means you can select them in VB. You can do this by selecting the components option from the project menu. Find the VBD3D component in the list (you can speed this up by pressing the v) and select it.

Now you are ready to use the *Scene* and *Thing* components. Add a (one) *scene* container to your form, make it fairly big (e.g. almost the size of the form, you can even enlarge the form to make a really big 3D scene). Next place one *thing* control inside the *scene* container. You will see some things happening now:

- The picture inside the *thing* control is that of a cube, this means that this *thing* control currently represent a cube in your *scene.*
- A black square appears in your container. This is your *thing* control (which is now a cube as that is the standard form) that is rendered inside the *scene* container. The *scene* is rendered at design time so you can immediately see what you have created.

Your form should now look like this:

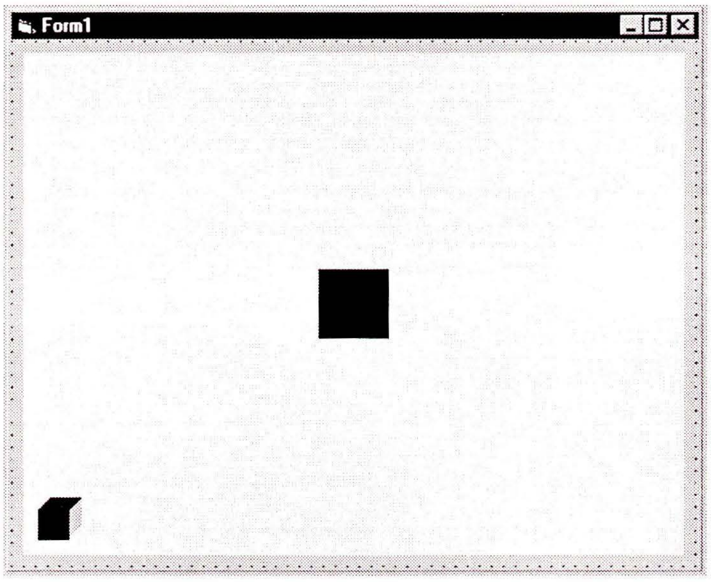


Figure 31 The first Thing control

While you have the *thing* control in your container selected browse through the properties window on the right. Here you can manipulate your *thing* by changing the properties of it. You might wonder why the square you see in the *scene* container is black, since the color property of the *thing* control is set to white. This is because there is no light in our *scene.* A white object with no light shining on it is displayed as black.

5.1.2.2 Step 2 - then there was light

The next thing to do will be adding a little light. Place another *thing* control in the *scene* container. Since the *thing* control will default to a cube you will not see anything change this time (except that there are now 2 cube things). Find the ThingType property of this second *thing* control. With this property you can determine if your *thing* is an object (ThingType is set to 0) or a light source (ThingType is set to 1). Set this property to 1 to make your second *thing* a light source. You will immediately see the square color change to light gray, this is because the light source is set to half intensity on default.

To change the cube into a ball you will need to know a bit more about the objects a *thing* can represent. A *thing* has a number of standard objects namely a cube, cone, cylinder, sphere and torus. By default a *thing* represent the standard cube object. To change the type of object each *thing* has a property named ObjectType. This property can have the following values:

Value of ObjectType	Object the *Thing* represents
0	A standard Cube
1	A standard Cone
2	A standard Cylinder
3	A standard Sphere
4	A standard Torus
5	An object loaded from file (DirectX x-file)

Since we want to make a bouncing ball we can use the standard sphere object for this. So change the ObjectType property of the first *thing* in the *scene* container (the one with the cube picture on it) to 3. After doing this the picture in the *thing* control will change to that of a sphere so you can see which object it represents and the container will now render a sphere. Your form should now look like this:

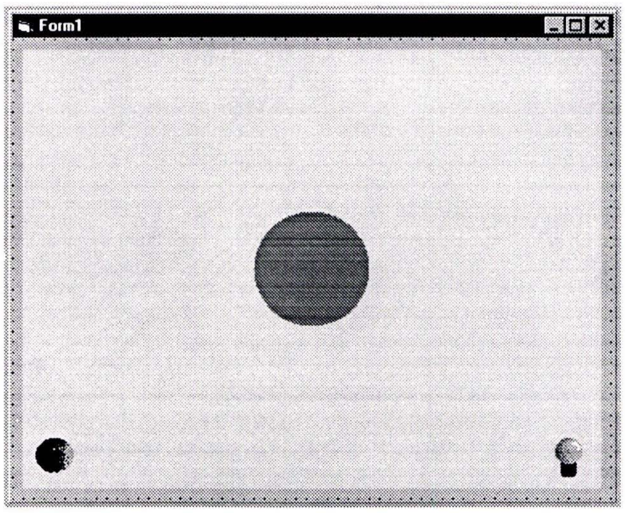


Figure 32 The scene with a light source

5.1.2.3 Step 3 - Enhancing the scene

To make a nice demo we need a few more things. First of all we will need something for the ball to bounce on. Add another *thing* to the *scene* container (that will default to a cube inside the sphere). Change the PosY property of the cube (thing3) to -5 and you will see the cube appear beneath the ball. Now we will have to stretch the cube a little. Change the ScaleX and ScaleZ properties to 5 to create a platform. Also change the ScaleY property to 0,5 so the complete platform will be visible.

Another thing that doesn't make the image very clear is that the objects only have 1 color. This is because we only have an ambient light present. An ambient light doesn't have a direction or position but simply lights everything in the scene with the same amount of light. Therefore add a point light source. A point light source has a position and emits light in all directions. The light gets less intensive when farther away from the light source. Add another thing to the container and make a light source of it by setting the ThingType property to 1. Now you're objects will be completely white. This is because you now have 2 ambient lights present both at half intensity making full intensity together. We need to change this light source to a point light source. You can do this by changing the LightType property.

The LightType property can have the following values:

LightType	Type of Light	Effect
0	Ambient	Complete area is lit with the same intensity
1	Point	Positional light in all directions (slow)
2	Spot	Light with orientation and position (slow)
3	Directional	Light that only has orientation (e.g. the sun) (fast)
4	Parallel	Light with orientation determined by position (fast)

Set the LightType property to 1. Now you only see the effect of the point light on the platform and not on the ball. This is because the point light source is positioned inside the ball. Change the PosX property to 5, the PosY property to -4 and the PosZ property to 15 (when you move the light you will see a red diamond shaped object at its current position). Now you will see the light source in effect. This looks much nicer, but it still has one problem. Unless you are running in True- Color mode you can see color lines on the objects (this effect is called banding). You can do something about banding by enabling dithering (blending the colors) while rendering the *scene*. Select the *scene* container and set the Dither property to True, to make things look nicer. Your form should now look like this:

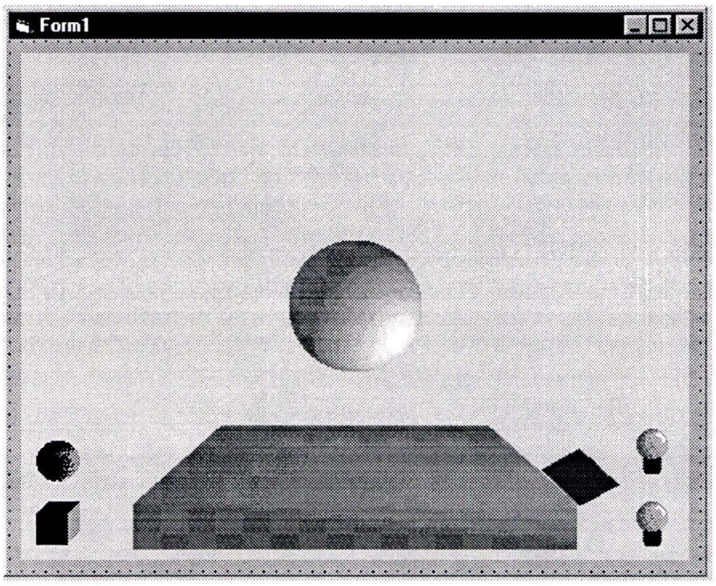


Figure 33 The finished scene

5.1.2.4 Step 4 - Time for some coding

Now it's time to make the ball bounce. First we need to place the ball a bit higher so it will bounce a bit longer. Set the PosY property of the ball (thing1) to 5. Add some code to move the ball. First declare the following variables:

```
Dim t As Single   ' simulated time
Dim v0 As Single  ' simulated speed
Dim h As Single   ' simulated height
```

We are going to use a simple quadratic formula to simulate falling. Select the Initialize event of the *scene* container and place the following code inside it:

```
Private Sub Scene_Initialize()
    t = 1.5: v0 = 15
End Sub
```

The Initialize event is called when de container is created, this happens when you start your program. You can use this event to initialize the variables you use in your code. Next is the PostRender event. This event is called after the *scene* is rendered to the screen. You can use this event to manipulate the objects in your 3D world each frame. The code you can place in the PostRender event is:

```
Private Sub Scene_PostRender()
    t = t + 0.1
    h = v0 * t - 5 * t * t 'Newton
    if h < 0 Then h = 0
    Ball.PosY = h
    If h < 0.1 Then ' bounce up
        t = 0: v0 = v0 * 0.8
    End If
End Sub
```

If you run the program after adding this code the ball will start bouncing. To speed things up a little bit you can set the frametime (minimum time needed for each frame) property of the *scene* container a bit lower (E.g. 10 for 10 milliseconds).

5.1.2.5 Step 5 - Controlling the platform with the joystick

To control the movement of the base with the joystick, we add the SWFF control to the set of available components in the same way we added the VBD3D controls. Next we place a SWFF control on our form and we name it joystick. We also declare some extra variables:

```
Dim Effect As Long
Dim EffectOn As Boolean
Dim X As Long, Y As Long
```

Last but not least we add a few lines of to initialize the joystick timing and make pressing the fire button throwing the ball up:

```
Private Sub Joystick_Button0()
    t = 1.5: v0 = 15
End Sub

Private Sub Joystick_Initialized()
    Joystick.TimerInterval = 10
    Joystick.AutoCenter = False
    Effect = Joystick.CreateConstantForce(-1, 0, 5000, -1)
    EffectOn = False
End Sub
```

We want to start and stop the constant force when the ball lands on or leaves the base to simulate weight. We also want to connect the position of the joystick to the position of the base. For this we change the PostRender routine:

```
Private Sub Scene_PostRender()
    t = t + 0.1
    h = v0 * t - 5 * t * t
    Call Joystick.GetPosition(X, Y)
    Platform.PosX = X / 2000
    Platform.PosY = -Y / 2000
    If h < 0 Then h = 0
    Ball.PosY = h + (Platform.PosY + 1.75)
    If h < 0.1 And Not EffectOn Then
        t = 0: v0 = v0 * 0.8
        Joystick.StartEffect Effect
        EffectOn = True
    End If
    If h >= 0.1 And EffectOn Then
        Joystick.StopEffect Effect
        EffectOn = False
    End If
End Sub
```

After this step we can move the platform up and down with the joystick and we feel the weight of the ball if it's on the platform. We can also fire the ball up by pressing the fire button.

5.1.2.6 Step 6 - Adding sounds on impact

To make things even more natural we add sound to the demonstrator. Similar to the previous steps, add the control for sound (jrSoundControl) to the set of available components and place one instantiation on our form. Call it "BangSound". We use an audio file called "bang.wav", with a "bang" sound to simulate an impact of a steel ball and start it with diminishing attenuation when the ball hits the platform. This completes our demonstrator with the following code:

```
Dim Att As Single
Dim vin As Single
Const AttMax = 100


Private Sub Scene_Initialize()
    t = 1.5: v0 = 15: vin = v0
    BangSound.jrInitSound ("bang.wav")
End Sub


Private Sub Scene_PostRender()
    t = t + 0.1
    h = v0 * t - 5 * t * t
    Call Joystick.GetPosition(X, Y)
    Platform.PosX = X / 2000
    Platform.PosY = -Y / 2000
    If h < 0 Then h = 0
    Ball.PosY = h + (Platform.PosY + 1.75)
    If h < 0.1 And Not EffectOn Then
        t = 0: v0 = v0 * 0.8
        Att = AttMax * (v0 / vin - 1) ^ 4 ' become softer
        BangSound.jrSetAttenuation att
        BangSound.jrPlaySoundSingle
        Joystick.StartEffect Effect
        EffectOn = True
    End If
    If h >= 0.1 And EffectOn Then
        Joystick.StopEffect Effect
        EffectOn = False
    End If
End Sub
```

This completes our second demonstrator. A screen shot is also shown on the cover page of this report. With a few lines of code we can animate a 3D world, we can feel it and we can hear it.

5.2 Applications using NUKE components

In this section we give a short overview of other applications using NUKE components. For more details on these applications we refer to the projects in which these applications are developed.

5.2.1 A20 - Multi-modal TV control prototype

In the Screen Management project an authoring and simulation environment for TV user interfaces has been created. This environment is intended to speed up the development process of families of TV user interfaces. To one of the simulations we added sound effects when moving the "puck" through the menu tree. These effects use the possibilities of DirectSound: fading left and right, multiple simultaneous sounds, pitch changes.

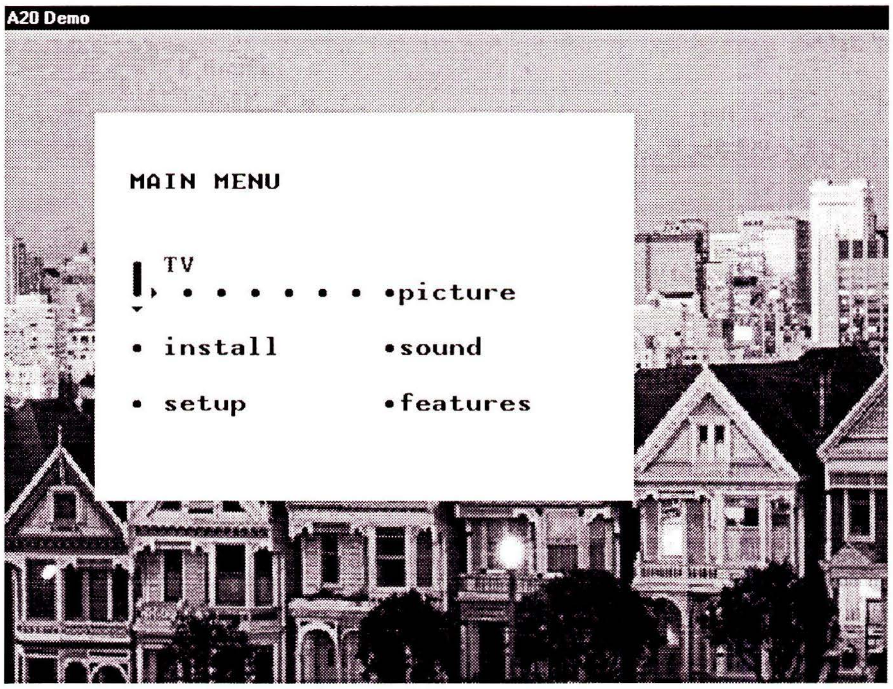


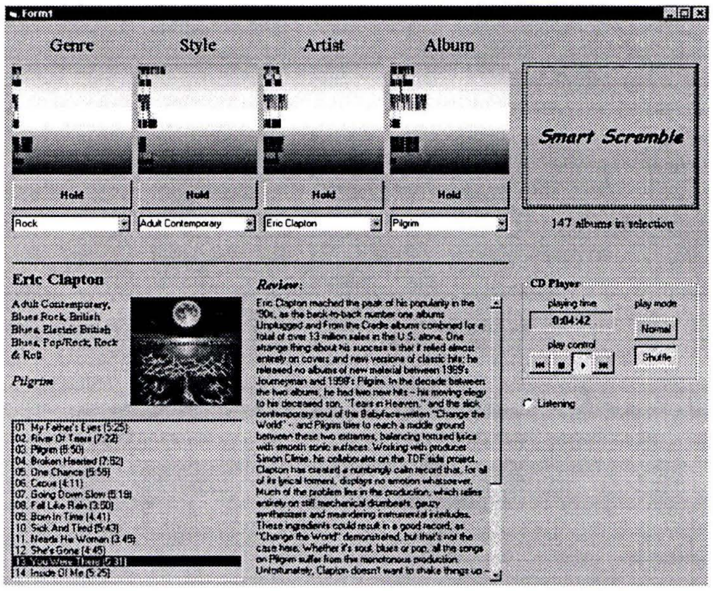
Figure 34 A20 Main Menu

We also changed the use of remote control cursor keys to use of the joystick with force effects for the different types of nodes in the menu tree. We made these additions to the simulation (the menu-interpreter) in a few days. This demonstrator served to show mainly the use of digital sound in a TV menu. The use of force feedback on a remote control is not foreseen in the near future (although in Philips Research Aachen can create very small, low power force feedback devices). For us it was another exercise to find out how useful our components really are and how easy to use from a programmer's perspective.

5.2.2 MacDice – Multi-Modal Music Browser

The second project that used our work was the MacDice project. This project evaluated the added value of multi-modal interfaces in a scientific way.

The modalities used included graphics and sound output, both speech and non-speech, speech and remote control input. One of the MacDice demonstrators uses both the Direct-Sound control and the SelectionWheel. A screenshot is given in Figure 35.



Rock | Adult Contemporary | Eric Clapton | Pilgrim

147 albums in selection

0:04:42

Eric Clapton

Adult Contemporary, Blues Rock, British Blues, Electric British Blues, Pop/Rock, Rock & Roll

Pilgrim

01. My Father's Eyes (5:25)
02. River Of Tears (7:22)
03. Pilgrim (5:50)
04. Broken Hearted (7:52)
05. One Chance (5:55)
06. Circus (4:11)
07. Going Down Slow (5:19)
08. Fall Like Rain (3:50)
09. Born In Time (4:41)
10. Sick And Tired (5:43)
11. Needs His Woman (3:45)
12. She's Gone (4:45)
13. You Were There (5:31)
14. Inside Of Me (5:25)

Review:

Eric Clapton reached the peak of his popularity in the '90s, as the back-to-back number one albums Unplugged and From the Cradle albums combined for a total of over 13 million sales in the U.S. alone. One strange thing about his success is that it relied almost entirely on covers and new versions of classic hits; he released no albums of new material between 1989's Journeyman and 1998's Pilgrim. In the decade between the two albums, he had two new hits -- his moving elegy to his deceased son, "Tears in Heaven," and the slick contemporary soul of the Babyface-written "Change the World" -- and Pilgrim tries to reach a middle ground between these two extremes, balancing tortured lyrics with smooth sonic surfaces. Working with producer Simon Climie, his collaborator on the TDF side project, Clapton has created a numbingly calm record that, for all of its lyrical torment, displays no emotion whatsoever. Much of the problem lies in the production, which relies entirely on stiff mechanical drumbeats, gauzy synthesizers and meandering instrumental interludes. These ingredients could result in a good record, as "Change the World" demonstrated, but that's not the case here. Whether it's soul, blues or pop, all the songs on Pilgrim suffer from this monotonous production. Unfortunately, Clapton doesn't want to shake things up --

Figure 35 MacDice JukeBox

5.2.3 WWICE

The WWICE project prototypes an in-home digital network on which multiple applications can be active on multiple places. To support multi-stream audio the base class of the jrSoundControl has been used and put in a Java wrapper for usage by the WWICE application.

5.2.4 Grapefruit

The Grapefruit project explored the use of advanced graphics in user interfaces for consumer electronic products. Advanced includes 3D, animation and translucency. The demonstrators of Grapefruit were build directly in Visual C/C++ on top of DirectX and OpenGL. We now explore usage of the VBD3D controls (Scene and Thing), that seems to give a easier to use generic framework.

6 Conclusions and future

From our work and from the scientific literature we strongly believe that SW components have the future. This also holds for UI SW components. We have shown that combining two Microsoft technologies to create a set of UI components allows the rapid construction of multi-modal user interfaces on the PC. For the project team this was a useful learning expedition, with a set of (re-)usable components as the most tangible result.

We also believe that "beyond the GUI" interfaces are coming. On the PC the game industry takes the lead, but other consumer devices will follow. Our components can be and are used to develop options in this direction. Packaging DirectX functionality in ActiveX components allows rapid prototyping of such multi-modal interfaces.

On the PC the development is ongoing. Highly quality animated 3D interfaces with speech input and output will become common and not just for games. Next steps will include the use of computer vision based gesture recognition. We expect Microsoft to integrate DirectX with the Windows operating system more and more in the near future. In Windows'98 DirectSound is already part of the OS. We also expect Microsoft to put more and more generic user input- and output functionality in the operating system.

6.1 Future work

We have the following suggestions for continued explorative work in UI SW components:

- Support speech recognition and speech synthesis, using also standardized APIs in easy to use components
- Support gesture and handwriting recognition for touch screens or pen-based interaction.
- Explore UI components on the Web - both Java and ActiveX based.

7 Glossary

API	Application Programmers Interfaces
ATL	Active Template Library, a set of C++ templates for Windows programming
CLSID	CLass IDentifier
DISPID	DISPatch IDentifier
COM	Microsofts Common Object Model
CORBA	Common Object Request Broker Architecture
DCOM	Distributed COM, COM over a network
DLL	Dynamic Link Library
DMA	Direct Memory Access
EPG	Electronic Program Guide, on-screen alternative for paper TV guide
EXE	Executable file
GDI	Graphics Device Interface
GUI	Graphical User Interface
GUID	Globally Unique IDentifier
HAL	Hardware Abstraction Layer
HEL	Hardware Emulation Layer
HTML	Hyper Text Mark-up Language, format of documents on the Web
IDL	Interface Definition Language
IID	Interface IDentifier
IPO	centre for research on User-System Interaction on the TUE
MFC	Microsoft Foundation Classes, a framework for Windows programming
NUKE	New User Interface Knowledge Exploration, project name
OCX	OLE Control eXtension, old name for ActiveX Control
OLE	Object Linking and Embedding, now a Microsoft term in itself
ORB	Object Request Broker, CORBA equivalent for the component registry
PCM	Pulse Code Modulation
PDA	Personal Digital Assistant
SDK	Software Development Kit
USIT	User System Interaction Technology, Nat.lab. research group
VRML	Virtual Reality Modeling Language (3D file format)

8 References

[1] Moll, H.F., *NUKE Project Agreements*, Nat.Lab. memo, 1998

[2] Reeves, B. and Nass, C., *The Media Equation; How People Treat Computers, Television, and New Media Like Real People and Places*, Cambridge Univ Pr (Trd); ISBN: 157586052X , 1997

[3] Vet ed., J. de, *A Vision of the Future: User Interfaces for Home and Away*, Nat.Lab. memo, 1998

[4] Rogerson, D., *Inside COM* ,Microsoft Press, ISBN: 1-57231-349-8, 1997

[5] Denning, A., *Ole Controls Inside Out*, Microsoft Press, ISBN 1-55615-824-6, 1995

[6] SoftwareAg, *EntireX, DCOM implementation for Unix/Linux*, http://www.softwareag.com/corporat/solutions/entirex/entirex.htm

[7] Ronan Geraghty (Editor), *COM-CORBA Interoperability (Microsoft Technologies Series)*, Prentice Hall Computer Books; ISBN: 0130962775

[8] Pennings, M., *KOALA Introduction course*, IST/IPA, 1998

[9] Bargen, B. and Donelly, P., *Inside DirectX*, Redmond, WA, Microsoft Press, 1998

[10] Voort, M. v.d., *Nuke Direct3D Components User Manual*, IPO Manual in preparation. IPO, Center for Research on User-System Interaction, 1999

[11] Voort, M. v.d., *Nuke Direct3D Components Reference Manual*, IPO Manual in preparation, 1999

[12] Peter J. Kovach, *The Awesome Power of Direct3D/DirectX*, 1997, Manning Publications Company, ISBN 1884777473

[13] Microsoft, *Sidewinder Force Feedback SDK Programmer's Reference Version 1.2*, Redmond, WA, Microsoft Cooperation, 1997

[14] Boschman, M.C., *SWFF: an ActiveX control for the Microsoft Sidewinder FF Pro Joystick*. IPO Manual in preparation, 1999

[15] Pijper, J.R. de, *Using Direct Sound*, IPO Manual in preparation, 1999

9 Appendices

9.1 Appendix A - SWFF Properties, methods and events

9.1.1 SWFF Properties

Property name	**Type**	**Arguments**	**Values**	**Description**

AutoCenter	boo-lean	none	True=on
			False=off
				get/set property indicating whether the X&Y axes of the joystick will be auto-centered by its motors.

AxisMode	long	none	0=absolute 1=relative	get/set property indicating whether the axes of the joystick should be interpreted as *absolute* or *relative* values.
BufferSize	long	none	size of input-buffer	get/set property indicating the size of the joystick's input buffer.
Buttons	short	none	Bitwise indi-cation of button status	get property indicating which buttons are pressed at a time. Each button is represented by one bit in the Buttons field. Bit 0 corresponding to button 0 etc.
CalibrationMode	long	short *axis* 0=X-axis, 1=Y-axis, 2=Throttle, 3=Rudder	0: cooked 1: raw	get/set property determines whether calibrated (cooked) or uncalibrated (raw) data should be retrieved from the axis indicated by the argument.
DeadZone	long	short *axis* *axis*: see above	[0,10000]	get/set property indicates the size of the dead zone, i.e. a range around the center of the axis where the reported value is as being at the center of the range.
Direction	long	long *ef-fect_index*	[0,36000]	get/set property indicating the direction (in centigrades) of an effect indexed by the argument. (the index value is returned when the effect is created).
FFGain	long	None	[0,10000]	get/set property indicating the gain to be applied to all joystick motors.

FFLoad	long	None	[0,100]	get property indicating the memory load in percent points of the device.
Granularity	long	short *axis* *axis*: see above	[0,10000]	get property indicating the size of the granularity of the axis indexed by the argument. Granularity represents the smallest distance an axis will report movement. Hence, it determines the possible values of an axis. A value of 1 means that all values are possible.
NBorders	short	None	[0,99]	get property indicating the number of virtual borders defined in the XY coordinate system.
NEffects	short	None	[0,maxshort]	get property indicating the number of force effects created for the joystick.
POV	long	None	(0, 4500, 9000, 13500,18000, 22500, 27000, 31500): the 8 angles, or 65535: if POV is in its center-position.	get property indicating the last polled value of the POV (point of view) button. It can be in 9 states of which 8 express the angle of the button position in centigrades and one indicating that the POV button was in the center-position.
Rudder	long	None	[0,65535]	get property indicating the polled value of the Z-axis (also known as rudder) of the joystick.
Saturation	long	short *axis* *axis*: see above	[0,10000]	get/set property indicates the saturation level of an axis indexed by the argument. It determines the point at which the axis is at its most extreme position.
Throttle	long	None	[0,65535]	get property indicating the last polled value of the joystick's potentiometer (also known as throttle).
TimerInterval	long	None	[0,maxlong]	get/set property indicating the

			interval in microseconds of the internal timer. After each Timer-Interval the control will poll the joystick's coordinates and button states and checks whether borders are crossed. If necessary event messages will be sent to the application window.

9.1.2 SWFF Methods

Method name	**Arguments**	**Return value**	**Description**
AboutBox	None	void	Method displays a MessageBox containing general information about the control: copyright, version etc.
CreateBorder	long *x0,y0,x1,y1* *x0,y0,x1,y1*: coordinates of begin and end point of the border ([-10000, 10000])	short: index of the defined border	Method defines a border in the XY-plane by its begin and ending points. Whenever the joystick's XY-position crosses one of these borders, event messages will be fired to the containers window. Up to 100 (index [0,99]) borders can be defined.
CreateCon-stantForce	long *duration, direction, magnitude, trig-ger_button* *duration*: in miliseconds of the created effect. [0,maxlong]: the duration -1: infinite duration. *direction*: direction of the axis of the effect in centigrades [0,36000]. *magnitude*: strength of the force [0,10000]. *trigger_button*: [0,8]: effect starts after pressing button 0 - 8 -1: effect is not triggered by any button	long: index of the defined effect.	Method defines a constant force effect along an axis defined by the direction argument.

CreateDamper	long *duration, b_x, b_y, v0_x, v0_y, trigger_button*
	
	duration: see above.
	
	b_x, b_y: damping coefficients for the X and Y axis resp.
	[0,10000].
	
	v0_x, v0_y: velocity offset for both axes [-10000,10000].
	
	trigger_button: see above.
		long: index of the defined effect.	Method defines a damper, which is a velocity dependent force effect with the following force-velocity relations:
			Fx=b_x(v_x - v0_x)
			Fy=b_y(v_y - v0_y)
			Fx and Fy are the X and Y components of the Force and v_x and v_y are the velocity components along X and Y. The velocity is determined by the change in XY position caused by movement of the stick.

CreateFriction	long *duration, f_x, f_y, trigger_button* *duration:* see above *f_x,f_y*: friction forces for the X and Y axis resp. (-10000,10000) *trigger_button*: see above	long: index of the defined effect	Method defines friction effect, which is a force effect which occurs whenever the stick is moved: v_x != 0: Fx=f_x v_y != 0: Fy=f_y Fx and Fy are the X and Y components of the Force.

CreateInertia	long *duration, m_x, m_y, a0_x, a0_y, trigger_button*
	
	duration: see above.
	
	m_x,m_y: inertia coefficients (mass) for the two axes
	[-10000,10000].
	
	a0_x,a0_y: acceleration offset for both axes [-10000,10000].
	
	trigger_button: see above.
		long: index of the defined effect	Method defines Inertia, which is a acceleration dependent force effect with the following force-acceleration relation:
			Fx=m_x(a_x - a0_x)
			Fy=m_y(a_y - a0_y)
			
			Fx and Fy are the X and Y components of the force and a_x and a_y are the acceleration components along X and Y. The acceleration is determined by the change in XY position caused by movement of the stick.

CreatePeriodic	short *waveform*, long *duration, direction, magnitude, offset, phase, period, trig-*	long: index of the defined effect	Method defines periodic waveform effect, which is a force effect that occurs whenever the stick is moved from its center position. The

	ger_button
	
	waveform: indicates which wave function is used
	0: Square wave
	1: Sine wave
	2: Triangle wave
	3: Sawtooth up wave
	4: Sawtooth down wave
	
	duration: see above.
	
	magnitude: the amplitude of the wave effect [0,10000].
	
	offset: the offset of the wave function [-10000, 10000].
	
	phase: phase of the wave form. (0,9000,18000) centigrades.
	
	Period: period of the wave function in ms [0,maxlong].
	
	trigger_button: see above.
			center position for the spring is defined by the corresponding arguments. The force is defined by:
			
			Fx=k_x (x - center_x)
			
			Fy=k_y (y - center_y)
			
			Fx and Fy are the X and Y components of the Force, x and y are the coordinates of the joystick's position in the XY system.

CreateRampForce	long *duration, direction, start_magnitude, end_magnitude, trigger_button* *duration:* see above *direction*: orientation of the effect in centigrades [0,36000] *start_magnitude, end_magnitude*: the strengths of the two extremes of the ramp function [-10000,10000] *trigger_button*: see above.	long: index of the defined effect	Method defines ramp function effect. The force - time relation is defined by: F = start_magnitude + (end_magnitude - start_magnitude) /duration*t
CreateROMEffect	short *effect,*	long: index	Method creates a predefined

	long *duration, direction, trigger_button* *effect:* indicate which of the following ROM effects will be created: 0 =RandomNoise 1 =AircraftCarrierTakeOf f .... 31 = Cannon *duration*: see above. *direction*: see above. *trigger_button*: see above.	of the defined effect	ROM waveform effect identified by the effect argument.

CreateSpring	long *duration, k_x, k_y, center_x, center_y, trigger_button*
	duration: see above.
	k_x, k_y: friction forces for both axes [-10000 ,10000].
	center_x, center_y: offset of spring center position [-10000,10000].
	trigger_button: see above.
		long: index of the defined effect	Method defines spring effect, which is a force effect that occurs whenever the stick is moved from its center position. The center position for the spring is defined by the corresponding arguments. The force is defined by:
			Fx=k_x (x - center_x)
			Fy=k_y (y - center_y)
			Fx and Fy are the X and Y components of the Force, x and y are the coordinates of the joystick's position in the XY system.

CreateVFXEffect-FromFile	string *filename*, long *trigger_button* *filename:* pathname of the effect file *trigger_button*: see above	long: index of the defined effect	Method creates an effect object from a VFX or a FRC file that is previously created with the Visual Force Factory, which is included in the Side-Winder FF SDK.
CreateWall	long *duration, direction, distance*, boolean *inner*, long *coefficient, trigger_button* *duration:* see above.	long: index of the defined effect	Method defines a wall object, which occurs when the joystick position is passing the position of the wall. The wall offers a resistance when the joystick's position is moving away from its origin if the

	direction: orientation of the wall in centigrades. Only 4 directions are possible: 0: North 9000: East 18000: South 27000: West *distance*: distance of the wall from the origin of the XY-system [0,10000]. Inner: flag indicating whether resistance is directed towards (true) or away from (false) the center position. *coefficient*: the amount of resistance offered by the wall [-10000,10000]. *trigger_button*: see above.		inner argument is true. In the other case (inner=false) the wall generates resistance if the joystick is moving from outside the wall towards the origin. Up to 4 wall effects can be created at a time.
DestroyBorder	short *border_index* *index*:indicates the border object that should be destroyed. This value is previously returned by the Create-Border method.	void	Method destroys a previously created border object.
DestroyEffect	long *effect_index* *effect_index:* the ef-fect that should be destroyed. This value is previously returned by one of the Cre-ate<effect> methods.	void	Method destroys a previously created effect object.
GetPosition	long *x, y* *x,y:* output arguments, the coordinates of the position of the stick in the XY-system during the last poll.	void	Method reads the last polled position coordinates of the joystick and returns them into the x and y arguments.

StartEffect	long *effect_index* *effect_index:* indicates the effect that should be started. This value is returned by one of the Create<effect> methods.	void	Method starts a previously created effect object.
StopEffect	long *effect_index* *effect_index:* see above	void	Method stops a previously created and running effect object.

9.1.3 SWFF Events

Event name	**Description**
Button0, Button1, ..., Button8	These events are fired whenever one or more of the corresponding buttons are being pressed.
Initialized	This event is fired to indicate that the force-feedback device is initialized and acquired. The application shall wait until this event occurs before creating force effect objects.
NS(border_index), SN(border_index), WE(border_index), EW(border_index)	These events are fired whenever a border is crossed along one of the four directions North-to-South, South-to-North, West-to-East, or East-to-West. If the border is oblique then two events should occur at a time.
RelButton0, RelButton1, ..., RelButton8	These events are fired whenever one or more of the corresponding buttons are being released.
Timer	This event is fired after elapsing the interval of the timer that is running inside the control. The same timer is used to poll the position and check whether borders are crossed.

9.2 Appendix B - jrSoundControl

The following is a complete list of methods available through jrSoundControl, with a short description of each. Note that the control does not have properties or events.

9.2.1 jrSoundControl Methods

Method name	**Arguments**	**Return value**	**Description**
jrInitSound	LPCTSTR szWaveFile-Name	void	Each jrSoundControl variable must be initialized with this method. It constructs all the necessary DirectSound

			objects, opens the wave file specified as a parameter and either transfers all sound data into memory if the DirectSound buffer created for the file is a static one, or transfers a small portion of the sound data into memory if the buffer created is a dynamic one. In either case, the buffer is ready for playing after the call completes. If the wav file specified is not found or cannot be opened for any other reason, a runtime error results and the application is terminated.
jrDestroySound	none	void	When a jrSoundControl variable is no longer needed, it should be destroyed using this method.
jrPlaySound	none	void	Call this method to start playback in looping mode, i.e., when the end of the sound buffer is reached, it starts again at the beginning. Playback will continue until explicitly stopped with a call to jrStopSound().The call has no effect if the buffer is already play-ing.
jrPlaySoundSingle	none	void	Call this method to start playback in single mode. The sound is played through to the end of the buffer and then stops. The call has no effect if the buffer is already playing.
jrStopSound	none	void	Call this method to stop a sound during playback. The buffer is reset, so that a subsequent call to jrPlaySound() or jrPlaySoundSingle() will start at the beginning of the buffer. The call has no effect if the buffer is not playing.
JrGetPlaybackFre-quency	none	long: cur-rent play-back fre-quency in Hz	The current playback frequency in Hertz (Hz) that a buffer is set to can be obtained with this method. Unless it has been changed through a call to jrSetPlaybackFrequency, it equals the original sampling frequency of the wav file.

jrSetPlaybackFre-quency	long: playback frequency in Hz	void	Use this method to set the playback frequency to a different value in Hertz. Note that this will affect both the speed and the frequency of the sound: a doubling of the playback frequency makes the sound play back at twice the original speed and with double the original frequency. The playback frequency value must be between 10 and 100 000 Hz. An attempt to set to a value outside this range will result in the closest legitimate value.
jrGetAttenuation	none	long: attenuation in decibels (dB)	This method returns the attenuation to which a buffer is set, in decibels (dB). Originally, this value is 0, meaning that there is no attenuation and the sound will play back at full volume.
jrSetAttenuation	long: current attenuation, in decibels (dB)		Use this method to set the attenuation of the buffer to a different value, expressed in decibels (dB). This value must be between 0 (no attenuation, full volume) and 100 (maximum attenuation, silence). An attempt to set to a value outside this range will result in the closest legitimate value.
jrGetPan	void	long: current pan setting	This method returns the current so-called pan setting of the buffer, better known as the balance. This is a value expressed in decibels indicating the degree of attenuation of either the left (when the value is negative) or the right channel (when the value is positive). The pan value ranges from -100 (the left channel is completely silent) to +100 (the right channel is completely silent). Its initial value is 0. Note that the attenuation and pan settings are additive: If attenuation is set to 5 and pan to -5, the left channel will be attenuated by 10 dB and the right channel by 5 dB.
jrSetPan	long: pan value	void	Use this method to change the pan

	to be set	setting. The value must be between -100 (full attenuation of left channel) and +100 (full attenuation of right channel). The neutral setting is 0, where neither channel is attenuated. An attempt to set to a value outside this range will result in the closest legitimate value.