This file contains chapter one of the more than three hundred pages long manual.

Chapter 1 **Introduction**

1.1 Preface

3D RenderLib for Windows is a 3D graphics dynamic link library (DLL) designed especially for use in the Microsoft Windows environment.

3D RenderLib supplies over three hundred functions, which can be used to program 3D graphics applications or to visualize 3D geometric data. 3D RenderLib contains, for instance functions to render geometric primitives, change their color or surface characteristics and manage hierarchical data storage.

Where needed this manual will explain the general principle of the computer graphics algorithms and techniques used. This manual is not intended to cover all aspects or algorithms of todays or even of the used computer graphics techniques. A list of computer graphics literature is included elsewhere in this manual.

1.1.1 What is a DLL?

A DLL is an executable module containing functions that Windows applications can call in order to perform useful tasks. DLLs are similar to run-time libraries. The main difference is that DLLs are linked with the application at run-time, not when you link the application files using the linker. DLLs allow several applications to share a single copy of a routine. If two Windows applications are running at the same time, and both use a particular DLL function, both share a single copy of the code for that function.

DLLs are language independent, i.e. every language capable of calling DLL functions can call 3D RenderLib functions. All that is necessary is an include file describing the used data-types and an import library.

1.1.2 A General Overview

3D RenderLib currently supports triangle, square and polygon primitives with vertex normals, surface normals or no surface information. Polylines are also supported. Furthermore, polygon sets can be specified to create more complex polygons, e.g. a polygon containing a hole.

Primitives can be rendered in wire-frame, phong-, gouraud-, flat- and non-shading mode. The actual rendering does not block the system. So the user can continue with other work while the rendering continues.

The way these primitives will look is changed by attributes. Attributes are for instance the current fill-color, edge-color or surface-characteristics.

3D RenderLib supports an unlimited number of directional-, point- and spot-ligth sources. These lamps can be colored.

3D RenderLib creates its own windows, and can do its own window handling. Each window can contain an unlimited number of viewports. A viewport can be thought of as a window in a virtual 3D world. What that viewport 'sees' in that world is projected and visible in the viewport's screen, which is an area in a window.

3D RenderLib uses abstract data-types to deal in an object-oriented way with its windows, viewports, lamps, geometric objects and data storage files.

Internally 3D RenderLib works with 24-bit color. Because 3D RenderLib is implemented as a Dynamic-link library (DLL), applications can utilize a 24-bit or a 8-bit graphic-display simply by installing the appropriate version of the 3D RenderLib DLL on the system, without any need to change or recompile the application.

1.2 Using 3D RenderLib

Although there are more ways to use functions from a DLL, using an import library is the most commonly used method. This is called implicit link-time import. An implicit import is performed by listing the import library for the DLL on the linker command line for an application.

Because of the high memory requirements of 3D graphics in general, we recommend to use 3D RenderLib in enhanced mode only.

More information on how to call functions from a DLL should be in the information supplied by the manufacturer of your programming environment. The examples on the installation disk also show the general principle.

1.2.1 The 3D RenderLib Libraries

The 3D RenderLib import library, called **3drender.lib** and the actual DLL, called **3drender.dll**, are supplied on the installation disk.

1.2.2 How Windows Locates DLLs

Windows locates a DLL by searching the same directories it searches to find an application. To be found by Windows, the 3D RenderLib DLL must be in one of the following directories:

1. The current directory.
2. The Windows directory (the directory containing 'win.com').
3. The Windows system directory (the directory containing 'kernel.exe).
4. Any of the directories listed in the PATH environment variable.
5. Any directory in the list of directories mapped in a network.

Windows searches the directories in the listed order.

1.2.3 Running Multiple Applications

3D RenderLib functions can be called from any Windows application that imports the 3D RenderLib import library. There is only one copy of the actual DLL code present in the system, regardless of the number of applications using 3D RenderLib. All these applications can render, edit etc. at the same time, all sharing the same DLL.

1.2.4 Window Handling and Behaviour

Since 3D RenderLib creates its own windows and can do its own window handling, it is also possible to write a Windows program by calling 3D RenderLib functions only.
The **RLib_Wait** function was especially created for this purpose. It waits until the user closes the specified window by selecting 'Close' from its system menu.

Information and functions are provided to alter or enhance the standard appearance or behaviour of 3D RenderLib windows.

1.3 Abstract Data-types

3D RenderLib makes use of several abstract data-types. To avoid confusion we will not call these abstract data-types objects. When the term 'object' is used we mean an geometric object consisting of one or more primitives and attributes.

Every abstract data-type is visible to the programmer only through its identifier. An identifier can be thought of as a handle, referring to a collection of variables and functions, the actual abstract data-type. The characteristics of an abstract data-type can only be changed by using, the especially for this purpose designed, functions.

For instance a display shows it self as a window. When the programmer wants to alter the window's background color, it should call **RLib_SetDisplayBackGroundColor**, specifying the **RLib_DISPLAY** data-type, which identifies the meant display.

In this manual we often refer to an identifier as being the abstract data-type itself. It should be emphasized, that strictly speaking this is not the

case. The programmer has access to an identifier, pin-pointing the actual abstract data-type, which is allocated in memory when the data-type is created.

The abstract data-types used by 3D RenderLib are:

1.3.1 The RLib_BASE data-type

The **RLib_BASE** data-type, which identifies a base, contains information and resources which are used by all other abstract data-types.
The base is at the top of the hierarchy of abstract data-types. Every application contains one **RLib_BASE** data-type. When other abstract data-types are created it is usually necessary to specify the, already opened, base. (Except for a viewport which is created by specifying the display it is created in.)

The first 3D RenderLib function to call is **RLib_OpenBase**. This will initialize some buffers and resources used by all 3D RenderLib functions.

The last 3D RenderLib function to call is **RLib_CloseBase**. This will free the allocated buffers and resources

1.3.2 The RLib_DISPLAY data-type

The **RLib_DISPLAY** data-type, which identifies a display, contains information and resources which are used by **RLib_VIEWPORT** data-types. A display will show itself to the user as a window.

A **RLib_DISPLAY** data-type manages for instance a window and the window handling routines.

1.3.3 The RLib_VIEWPORT data-type

The **RLib_VIEWPORT** data-type, which identifies a viewport, contains information about the virtual 3D world and how and where it should be seen. It contains a camera through which it looks into the virtual world. A **RLib_VIEWPORT** data-type also contains attributes, which determine how the primitives will look when they are rendered.

Each viewport has a screen. A viewport's screen is that part of a display's window that was assigned to this viewport.

1.3.4 The RLib_LAMP data-type

The **RLib_LAMP** data-type, which identifies a lamp, contains information about a light source. The same data-type is used for all types of lamps.

A **RLib_LAMP** data-type contains a lamp's color and, depending on the type of lamp, the lamp's position and direction.

1.3.5 The RLib_SESSION data-type

The **RLib_SESSION** data-type, which identifies a session, contains information and resources which are used by **RLib_STRUCTURE** data-types.

A **RLib_SESSION** data-type manages an archive file (session). This session is used for hierarchical object-oriented data-storage.

1.3.6 The RLib_STRUCTURE data-type

The **RLib_STRUCTURE** data-type, which identifies a structure, is designed for hierarchical object-oriented data-storage. A structure can contain elements. An element can describe a primitive, an attribute or a call to another structure.

1.3.7 Identifying the Abstract data-types

All abstract data-type can have a unique name. A **RLib_SESSION** data-type, which contains a file, has a name by default. It should be specified when the session is created.

In contrast to all other abstract data-types, a **RLib_SESSION** has no ID. An ID can be used in the same way as an abstract data-type's name.

These names and IDs are unique identifiers. It is not possible to have more then one display, viewport or lamp with the same name or ID. Furthermore it is not possible to have more then one structure in a session with the same name or ID.

An abstract data-type's name and ID can be used to identify a specific abstract data-type.

1.3.8 Live Cycle of the Abstract data-types

Each of the data-types must be created before they can be used, and all but the **RLib_SESSION** data-type, can be deleted.

The **RLib_DISPLAY**, **RLib_VIEWPORT** and **RLib_LAMP** data-types are fully memory based and can only exist during the execution of the application. Once these data-types are created they can be used. Each of these data-types can have a unique name and ID. This name and ID can be used to gain access to a particular data-type, for instance in another part of the program than where the data-type originally was created.

The **RLib_SESSION** and **RLib_STRUCTURE** data-type can 'live' beyond the application. A **RLib_SESSION** only has a name, a **RLib_STRUCTURE** data-type can have a unique name and ID. This name and ID can be used to gain access to a particular structure, for instance in another part of the program, or even in another program than where the structure originally was created.

1.3.9 Multiple use of Abstract data-types

Since Windows is an event driven environment 3D RenderLib's abstract data-types can be used by several parts of a program at the same time.

To facilitate this concurrent use of data-types all abstract data-types contain a counter. This open-counter is used by 3D RenderLib to keep track of the number of times a data-type is in use, and to make sure no abstract data-type is closed or deleted by one part of an application while it is still use in another part.

Special care should be taken when more then one application make use of the same session or structure. 3D RenderLib currently can not facilitate that kind of 'global' use of its abstract data-types.

1.4 Function Groups

3D RenderLib contains over three hundred functions. All 3D RenderLib functions have the '**RLib_**' prefix. This prefix is followed by the actual function name.

The 3D RenderLib functions can be divided in several groups which can be distinguished by the first word of the actual function name. A few important groups are:

1.4.1 The 'RLib_Set' and 'RLib_Inq' Function Groups

The '**RLib_Set**' functions change run-time data. This data is lost when the application ends. For instance all data contained in displays, viewports or lamps are changed using functions from this group.
An example of a function from this group is **RLib_SetFillColor**, which alters a viewport's current fill-color.

The '**RLib_Inq**' functions retrieve run-time data.

1.4.2 The 'RLib_Put' and 'RLib_Get' Function Groups

The '**RLib_Put**' functions change or insert edit-time data. The data is not lost when the application ends. For instance all elements contained in a structure are inserted by calling functions from this group.
An example of a function from this group is **RLib_PutFillColor**, which inserts a fill-color element in a structure.

The '**RLib_Get**' functions retrieve edit-time data.

1.4.3 The 'RLib_Render' Function Group

The '**RLib_Render**' functions are used in immediate-rendering. For instance, **RLib_RenderPolygons**, renders the specified polygon data in a

viewport, using all current set data.

RLib_RenderStructure is a function from the '**RLib_Render**' group which is of course used in structure rendering.

1.4.4 The 'RLib_Open' and 'RLib_Close' Function Groups

The '**RLib_Open**' and '**RLib_Close**' functions are used to gain access to a particular abstract data-type, for instance in another part of the program than where the data-type originally was created, and to let 3D RenderLib know this data-type is currently used.

1.4.5 'RLib_Set' vs. 'RLib_Put' Functions

The functions **RLib_SetDisplayID** and **RLib_PutStructureID** illustrate the difference between the run- and edit-time routines.
Both these functions change the ID of an abstract data-type. A **RLib_DISPLAY** data-type is memory based, and can only exist during the execution of an application. A **RLib_STRUCTURE** data-type is partially file based and can 'live' beyond the application. The changed structure-ID is still the same the next time the structure is opened, while the display-ID is lost when the application ends.

A **RLib_STRUCTURE** data-type also contains run-time data. For instance the current element-index, which is changed by **RLib_SetElementIndex**. When an application ends, a structure's element-index is lost. The next time the structure is opened, its element-index is again set to default.

1.5 Structure vs. Immediate Rendering

3D RenderLib supports two types of rendering; structure rendering and immediate rendering.

1.5.1 Immediate Rendering

In immediate rendering a primitive is rendered by calling one of the '**RLib_Render**' functions. The primitive is rendered in the viewport that is specified. All the viewport's current attributes, camera position and direction, projection type and render-mode are used.

For instance, using immediate rendering this is how to render two polygons, the first red and the second green.
First set the viewport's current fill-color to red by calling **RLib_SetFillColor**, specifying red. Then the first polygon is rendered by calling **RLib_RenderPolygons**. Again, **RLib_SetFillColor** is called to change the viewport's current fill-color to green, and the second polygon is rendered.

1.5.2 Structure Rendering

In structure rendering a structure is rendered, for instance by calling **RLib_RenderStructure**. This structure contains a list of elements. An element can describe a primitive, an attribute or a call to another structure.

Rendering a structure will cause 3D RenderLib to traverse the specified structure. Every element encountered will be interpreted. When an attribute element is encountered, the specified attribute is changed in the viewport in which the structure is rendered. When a primitive element is encountered, that primitive is rendered. When another structure is called, this structure is opened and then rendered. This latter 'call elements' are responsible for the hierarchical nature of the data storage.

For instance, using structure rendering this is how to render two polygons, the first red and the second green.
First create a session and a structure in it. Then insert a fill-color element, specifying red, by calling **RLib_PutFillColor**. Next insert a primitive element, specifying the first polygon, by calling **RLib_PutPolygons**. Then the green color is inserted by calling **RLib_PutFillColor**. As last, the second polygon is inserted by calling **RLib_PutPolygons**.
This structure can be rendered by calling **RLib_RenderStructure**.

1.5.3 When To Use Which Rendering Method

When something simple, which is not worth the trouble of creating a session and structure for, should be rendered, immediate rendering is the fastest solution.
Immediate rendering offers a direct rendering interface. This can be very useful, for instance when an own data-base is used, or for rendering parametric surface representations like Bezier patches, by approximating the surface by a polygon mesh.

For applications in which there is significant change in geometric data between successive renderings, editing a structure does not pay.

The big advantage of structure rendering is the fact that the structure can be used several times, it can be edited and rendered again. A complex object, consisting of many primitives can be described by hierarchical structures and rendered by simply rendering the top-most structure. The session, containing the structures is available the next time the application is run.

Therefore, when a complex object is build or rendered, using structures is the method of choice.

1.5.4 Structure hierarchy

Here is an example of structure hierarchy.
A car can be described by three structures. The structure at the top of the hierarchy, for instance named 'Car', could call a structure describing the cars body. Next it could call the structure 'Wheel', describing a wheel, four times, each time using a different matrix, transforming each wheel to its

desired location. This car can be rendered by simply rendering the 'car' structure.

Notice that structure hierarchy can also be used to avoid data duplication. When an object contains a hundred screws, there is only need for one structure describing a screw. This structure can be called every time a screw is needed. Each time specifying the right matrix to transform the screw to its desired location.

1.5.5 Archive files and Display lists

Most graphics systems today offer hierarchical rendering. This is usually called a display list. Next to this display list an archive file is used for data storage. 3D RenderLib combines the two.

3D RenderLib's structures are file based optimized display lists. The big advantage is that data storage is natural. A created display list (structure) can be used the next time the application is run, or by other applications. The stored data can be directly rendered.

A disadvantage is of course, the fact that a file based display list can be slower then a memory based display list. In our experience there is usually only a minor difference.

The reason for this is disk caching and memory swapping.
A disk-cacher saves data read from the disk in memory. When that data is read again, the disk-cacher supplies the data directly from memory.
Window's SMARTDrive is such a disk-caching program.
When Windows gets low on memory it compensates by swapping information from memory to a swap file on the disk (enhanced mode only).

It is easy to see that both memory and file access are depending on the actual amount of available memory. When there is sufficient memory, file access will be mostly memory access. When there is insufficient memory both file and memory access will be mostly disk access.

1.5.6 Speed

Structure rendering is usually slightly slower then immediate rendering. The reason for this is that structure and immediate rendering use in fact the same functions but the former has the overhead of retrieving the data.

When a structure is rendered, all encountered elements are interpreted. For instance when a polygons element is encountered, the data that describes these polygons is retrieved by calling **RLib_GetPolygons**. Next the polygons are rendered by calling **RLib_RenderPolygons**.
It is easy to see that it is faster to directly call **RLib_RenderPolygons** as in immediate rendering.

The difference in rendering speed is usually very minor.
When very small and simple objects, that are easy manageable without the use of structures, are rendered, immediate rendering is usually faster. But when more complex objects are rendered using immediate rendering, the overhead of managing the data and the size of the application needed

to manage the data, undoes its speed advantage.

1.5.7 Function Coherence

The actual function names and arguments used in immediate and structure rendering are practically the same.

For example:
RLib_RenderPolygons(NumberPoly, NumberIndex, Index, NumberPoints, Points, Viewport)
RLib_PutPolygons(NumberPoly, NumberIndex, Index, NumberPoints, Points, Structure)

The first function renders the described polygons, while the second function inserts an element describing the polygons. When the structure is rendered, the inserted element will be encountered and the described polygons will be rendered.

Notice that the function's names only differ in their first word, and the function's arguments only differ in their last argument, the specified abstract data-type.

RLib_RenderPolygons is an 'immediate render' function, the viewport in which the rendering should be performed is specified.

RLib_PutPolygons is a 'structure render' function, the structure in which the polygons should be inserted is specified. A structure can be rendered by calling:

RLib_RenderStructure(Structure, Viewport, Matrix)

Now the viewport in which the rendering should be performed is specified.