DAZ

Carrara SDK™

Overview

Extension Architecture

Version 6.0
September 2007

Licensee acknowledges that the CarraraSDK Development Toolkit may contain bugs, errors and other problems that could cause system failures. Consequently, the CarraraSDK is provided to Licensee "AS IS," and DAZ disclaims any warranty or liability obligations to Licensee of any kind. Accordingly, Licensee acknowledges that any research or development that it performs regarding the CarraraSDK or any product associated with the CarraraSDK is done entirely at Licensee's own risk.

LICENSEE ACKNOWLEDGES THAT DAZ MAKES NO EXPRESS, IMPLIED, OR STATUTORY WARRANTY OF ANY KIND FOR THE PRODUCT INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY WITH REGARD TO PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE.

DAZ SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF REVENUE, LOSS OF PROFITS, BUSINESS INTERRUPTION, LOSS OF INFORMATION OR DATA AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE THE PROTOTYPE EVEN IF DAZ HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Content

Chapter 1 - Introduction 1

What is the Carrara SDK ? 1

An open architecture for 3D. 1

Headers and library. 1

Samples. 1

Documentation. 1

MCSketch. 1

Supported Platforms and Compilers. 1

What’s New in Carrara 6 ? 2

Unicode support 2

Non Linear Animation 2

Large scenes support 2

Multiple selection 2

What’s New in Carrara 5 ? 2

Change in resource files 2

Scene commands 2

Ghost menu 3

Tree chooser dialog 3

Class Id definition 4

Lighting Models 4

Change in interfaces 4

How the whole thing works 4

Shell and Extensions 4

COM dynamic linking 4

Identifying Components at startup: Auto Plug-And-Play 5

Family ID, Class ID... 5

...And Instances 5

Communicating between the 3D Shell and the 3D Component 6

Component User Interface and Parameter Maps (PMap) 6

Chapter 2 - Creating an Extension 9

The resource files (.dat and .txt) 9

Building the resource file 9

On Windows: 9

On MacOS 10

Extracting the localized strings from .dta 10

The ’COMP’ resource 11

The ’GUID’ resource 12

The ’PMap’ resource 13

Shaders and PMaps 14

The executable file (.mcx) 14

Build the executable 14

    Entry Points 14
    The Component Class 15
    TBasicUnknown 16
User Interface 17
    The 'Node' resource. 17
    The Pmap buffer 17

Chapter 3 - Using DAZ's COM Dynamic Linking 19

About COM 19
Component Registration Process 19
How the Component Server Works 19
How the Application Calls your Extension 19
    Extension Entry Points 19
    Minimal resources 20
    How your Extension Calls into the Application 20
        Using Shell objects received as parameters 21
        Using preset Shell objects stored in the library global variables 21
        Instanciating a Shell object yourself 21
    The "PMap" resource 22
IMCUnknown class 22
        What is a COM object? 22
    QueryInterface 23
    AddRef 23
    Release 23
    More on QueryInterface, AddRef and Release 24
Registering your Class ID with DAZ 24
    How to register 24
    Contacting DAZ - Developers Mailing list 25

Chapter 4 - Platforms and Compilers issues 27

Compiling for Windows 27
    Creating a new extension project for Visual C++ 2005: 27
Compiling for MacOS 28
    Creating a new project for XCode 28
    Debugging on MacOS 28

Chapter 5 - User Interface 29

User Interface Options 29
    No User Interface 29
    Auto PMap 29
    PMap and a Part 30
        Samples using a PMap and .rsr file 31
    Part Extensions 31
        To create a part extension: 31

Manually Controlling Your Own User Interface 32
  Creating your User Interface Containers 32
  Using a Dialog box 32
  Using a Window 32
  Setting a Child Part's Values 33
Examples of Specific user interface elements 34
  Registration Dialog box 34
  Tab Part 35
  Creating a Component Chooser 36
MCSketch 36
Extra Tokens for Each Part 36
List of all UI elements (parts) 36
  TMFEditTextPart - Edit Text part 37
  TMFIconButtonPart 37
  TMFTextButtonPart 37
  TMFDialPart 38
  TMFHierarchicalListPart 38
  TMFImagePart 38
  TMFCheckboxPart 39
  TMFColorPart 39
  TMFOverlayImageControlPart 39
  TMFRadioPart 40
  TMFToolbarPart 40
  TMFNodePart 40
  TMFAutoCenteringPart 41
  TMFBalloonPart 41
  TMFWindoidPart 41
  TMFDialogoidPart 42
  TMFCollapsiblePart, TMFSelectableCollapsiblePart 42
  TSingleComponentChooser 42
  TMultipleComponentChooser 43
  TMFFramePart 43
  TMFGradientPart 43
  TMFNodeReferencePart 44
  TMFRadioCluster 44
  TMFScrollPart 44
  TMFSlider, TMFCircularSliderPart, TMFLinearSliderPart 45
  TMFSolidColorPart 45
  TMFSplit3Part 45
  TMFCollapsibleListPart 46
  TMFVectorPart 46
  TMFListPart, TMFRuledListPart, TMFStringListPart 46
  TMFOffscreenPart 47
  TMFPoppedPart 47
  TMFRectPart 47
  TMFStaticTextPart 48
  TMFTextPopupPart 48
Interfaces supported by each UI element (part) 48

Chapter 6 - Counted Pointers 51

Defining Classes that May Need to be Counted 51
- TMCCountedObject/TMCSMPCountedObject 51
- Using AddRef() and Release() 51

Implementing Create 53
Implementing a Counted Pointer Access Method 53
Implementing QueryInterface() 54
Helper Classes for Counted Pointers 54
- Example of TMCCountedGetHelper 54
- Example of TMCCountedCreateHelper 55

TMCCountedPtr vs Interface Pointer 55
- Two functions returning InterfacePtr 56
- Never return a CountedPtr 56
- Creating objects 56
- Aggregation 56

Chapter 7 - Utilities 59

Various Shell Utilities (IShUtilities) 59
Thread Utilities (IShTreadUtilities) 59
Component Utilities (IShComponentUtilities) 59
SMP Utilities (IShSMPUtilities) 59
Action Utilities (IShActionManager) 59
Menu Utilities (IShMenuUtilities) 60
Part Utilities (IShPartUtilities) 60
Resource Utilities (IShResourceUtilities) 60
File Handling (IShFileUtilities) 60
- Creating Files 60
- Opening Files 60
- Working with Streams 61

Personality Utilities (IShPersonalityUtilities) 61
Drag & Drop Utilities (IShDragAndDropUtilities) 61
Change Management (IChangeManager) 61
Clipboard Utilities (IMFClipboard) 62
3D Utilities (I3DShUtilities) 62

Chapter 8 - MCCommon References 63

MCCore Interfaces 63
MCF Interfaces 63
- User Interface Flexibility 63

MCImage Interfaces 64

Chapter 9 - Failure Handling 65

Using the Failure Handling 65
General rules 65

A first simple example 65
Initializing your local variables properly 66
The Failure Handling Chain 67
Triggering a Failure yourself by using throw, ThrowIfNil(), ThrowIfNoMem(), and ThrowIfError() 67
Error codes 68
Message parameter coding 69
Failing Silently 69
Recovering from a Failure 70

Chapter 10 - Multi-Threading 71

Two types of threads. 71
Extensions and multi-threading 71

Chapter 11 - DataBase Overview 73

Scene 73
- Objects 73
- Scene Tree and Tree Elements 73
- Object Instances 73
- Lights 73
- Cameras 74
- Groups 74
- Master Groups vs Primitives 74

Coordinate Systems 74
- Global Coordinate System 74
- Working Box Coordinate System 75
- Local Coordinate System (or Object Coordinate System) 75
- Screen Coordinate System 76
- The screen pixels space 78

Geometry 79
- Geometric data type: 32-bit floating point (float) 79
- Units System 79
- Tree Elements Transformation 79
  - More on the rotation matrix: 80
- Geometry basics 80
  - Surface Point 80
  - Surface Normal 81
- UV Space (Texture Coordinate System) 81
  - Parametric mapping - a better solution 82
- Shading 82
- Facets 83
  - Interpolating in a facet 83
- Bicubic Patches 84
  - Patch normals 84
  - Patch u,v Space 85
  - To learn more about Bézier bicubic Patches 85

Color 85

Introduction

This chapter introduces the main concepts of the Carrara SDK. It will help you in making the main technical choices so you can have a smooth and exciting experience developing your 3D Components.

What is the Carrara SDK ?

An open architecture for 3D.

The SDK provides a way to extend the functionality of the application and add new features. It provides the tools and the documentation to create plug-ins for Carrara.

Carrara is entirely based on a plug-in model. Each of the components of the application (Modeler, Primitives, Lights, Renderers...) is implemented as a plug-in. This component architecture allow external developers to quickly add functionality to the application.

Headers and library.

The SDK provides a set of the C++ headers as well as a library that allow you to build plug-ins that are compatible with the API of Carrara. They also give you access to a wide range of utilities that makes it easier to write features for a 3D application.

Samples.

The SDK provides many samples of code that you can use as a starting point for your plug-ins. Those samples are detailed in the **Cookbook**. They provide good examples to understand how you can create your own extensions.

Documentation.

There are three different parts in the documentation of the SDK, each type answering a specific need of the 3D Components developer:

- The **Overview**: This pdf provides an overview to the SDK and explains the major concepts you need to understand to extend the functionality of Carrara. This is were you should start.
- The **Cookbook**: This pdf provides step-by-step descriptions of the various types of 3D Extensions that can be built with the SDK.
- The **Reference guide**: The reference guide provides a documentation of all the classes that constitute the API of Carrara.

MCSketch.

A resource Editor called MCSketch is provided with the SDK to help you create sophisticated user interfaces as quickly as possible.

Supported Platforms and Compilers.

The supported platforms are Mac OS X (10.3.9 or later), Windows 98, Windows 2000, Windows XP and Windows Vista. The supported compilers are Xcode 2.4 or later on the Macintosh, and Microsoft Visual C++ 2005 or later on Windows.

What’s New in Carrara 6 ?

This section describes the changes that were made to the SDK from version 5 to version 6. If you are porting plug-ins from a previous version, it is important that you read carefully this chapter. It might also give you ideas for new plug-ins... If you are new to the SDK, you can skip this section for now and come back to it later.

Unicode support

The strings stored in classes derived from TMCString are all using the UTF8 encoding. Use TMCWideString if you need to handle strings with 16 bits per character.

Non Linear Animation

The main interfaces dealing with NLA can be found in the files : I3DShAttributesSet.h, I3DShAttributesSetControler.h, I3DExAttributesSetControler.h, I3DExClip.h, NLAInterfaces.h. You can even create new external controlers.

Large scenes support

The magnitude of the scene (I3DShScene), can be accessed by using GetMagnitude and SetMagnitude. Cameras, lights and primitives can implement GetScalingFactor and SetScalingFactor when they need to change their behavior or default values depending on the current magnitude.

Multiple selection

In IExDataExchanger, HandleEvent has been deprecated and should be replaced by SimpleHandleEvent or GetUIHandler in components that can be displayed as a multiple selection in the properties (like light or camera components).

In IMFExPart, SetValue is now called SetValueLowLevel. The changes in IMFPart are unlikely to affect you.

What’s New in Carrara 5 ?

This section describes the changes that were made to the SDK from version 4 to version 5. If you are porting plug-ins from a previous version, it is important that you read carefully this chapter. It might also give you ideas for new plug-ins... If you are new to the SDK, you can skip this section for now and come back to it later.

Change in resource files

The use of resource files has been changed to be able to make localization much easier. The DTA files that were used before are still generated but have to be separated between a DAT file containing the resources common to all languages and a TXT file containing the localized strings. To separate your DTA files use the Resextracter in the Tools folder.

Once you have generated the DAT and TXT files you can create other TXT files to localize your plug-in. For instance your **myplugin.dta** file must be first separated in **myplugin.dat** and **myplugin.txt**, then you can translate the latter in french to **mypluginFR.txt**.

Scene commands

The scene commands resources have been modified to enable changement of keyboard shortcuts. For each scene command you need to add a **scmd** resource in your .R file:

```
resource 'scmd' (ResourceIdOfYourSceneCommandComponent)
{
```

```
          YourActionNumber,
          defaultMenu,
          {
             '3Dvw',// 3D View room
             'Stry',// Storyboard room
             //... and other rooms where you want the scene commands enable
          },
          kNoStrings,
          kDefaultName,
          kEditBindingGroupID,
          {
             kDefaultID, kDefaultName, 's', kShift, kCtrl, kNoAlt, kAnyPlatform;
             // the default keybord shortcut will be Shift + Ctrl + S
          },
          k3DViewBindingContextID
   };
```

Ghost menu

The ghost menu resource has changed and is now located in the .R file and not in the .RSR Here is how you should translate your old resources:

```
{
    TblE
    {
        irow 0
        icol 0
        ires 16100
        tool 9500
        Titl "Pen"
        Actn 9500
    }
    TblE
    {
        irow 0
        icol 1
        ires 16100
        tool 9503
        Titl "Add"
        Actn 9503
    }
}
```

is now in the .r file as:

```
resource 'ghmn' (128, "My Modeler Ghost Menu")
{
    {
        0, 0, 16100, 9500, "Pen", 9500,
        0, 1, 16100, 9503, "Add", 9503,
    }
};
```

Tree chooser dialog

To reference another object, light or camera you can still use its name but you can also use the new tree chooser dialog that will let the user pick a tree or a list of trees. These trees are then referenced by their Id and not by their name. This can be very helpful if users change their objects names quite often.

To see how to use the tree chooser dialog, take a look to the **Behavior** sample.

Class Id definition

We have made a slight change of the way you can define a class id. Use:

```
const MCGUID CLSID_TreePrim (R_CLSID_TreePrim);
```

instead of the old:

```
const MCGUID CLSID_TreePrim = {R_CLSID_TreePrim};
```

Lighting Models

The default lighting implementation used to be located in TBasicShader. While this was convenient, it was also causing some problems. For instance, if we fixed a bug in its implementation, all the shaders needed to be recompiled. For this reason, the default lighting model implementation is now in the application. This implies a few changes to your code:

1 If you implemented ShadeAndLight before, you need to make sure that you return MC_S_OK to let the shell know that you are overriding completely the lighting model. You will note that shadeAndLight takes a pointer to a I3DShLightingModel as a parameter. You can use the methods of this interface to call default implementations of the reflections, transparency, global illumination.

2 If you were overriding GetDirectLighting, GetReflections,... you can still do that but you have to let the application know by returning the proper combination of constant in GetImplementedOutput (for instance kUseCalculateDirectLighting).

If you have any problem, feel free to contact us directly.

Change in interfaces

We have not report here all the change that have been done in the various interfaces. Usually if you have not seen the change you will get a compilation error. However, when you are using basic implementations the error can be hidden and your function will not be called. Do not forget to check the signature of your functions if something is not working.

How the whole thing works

Shell and Extensions

The Application is the client of the different services provided by the **3D Components**. Because the 3D Components add features, they are often called **Extensions**, or **plug-ins**.

The Application is in charge of organizing the data flow between the Components, and it also offers a set of services to the 3D Components to allow then to interact with its own internal data (like the 3D Database). The Application is called the Shell, which is a better and more generic term.

COM dynamic linking

The communication between the 3D Shell and the 3D Components is done by using the Component Object Model (**COM**).

The choice was made to use the COM because COM is a very widely spread industry standard (COM is the low-level layer on which OLE is built), and is actively promoted by major industry players. So if you are already familiar with OLE, you will find it easier to get started with this documentation. COM offers a nice and clean C++-like interfacing, and it is highly recommended that you read the excellent book “Inside OLE 2” by Kraig Brockschmidt from Microsoft Press to learn everything about it.

COM users should read , “Using DAZ’s COM Dynamic Linking” chapter to learn all details about each technique and how to implement them.

Identifying Components at startup: Auto Plug-And-Play

When the 3D Shell is launched, it first identifies which 3D Components are available. To do this, it looks in its Extensions directory and in all sub-directories of the Extensions directory.

All files with the “.mcx” extension are considered as Components.

Please note that there is nothing to register anywhere in the operating system to allow the identification of the Component. It is completely automatic. **Even if you develop a Windows extension, it is not necessary to register your Component in the Windows Registry Database.** This allows a true Plug-And-Play installation of Components. No problem for uninstalling, no full path names issues, no conflicts between different versions or languages, etc.

For each Component file found, the 3D Shell tries to find the corresponding resources. The resources are located in a “.dta” file next to the Component file.

Now that the resources are found, the 3D Shell looks for all ‘**COMP’** resources in the file (you can put several Components in the same file). The ‘COMP’ resource is the key to each Component. It identifies the name of the Component, the API version number it is based on, his own version number, and most important of all, its **Family ID** and its **Class ID**.

Family ID, Class ID...

Each Component belongs to a **Family**. The Component Family defines the kind of Component the 3D Shell is dealing with: a Shader, a Camera, an Export Filter, etc. All the items in bold in the “3D Pipeline” chart in the Cookbook are the available Families.

Each Family has a 4 letter code, its **Family ID**. For example, the Family ID for Shaders is ‘shdr’. Your Component must belong to an existing Family, otherwise the 3D Shell will not know what to do with it. Family IDs are described for each Component in the Cookbook and Reference chapters, and there is a general table in “The ’COMP’ resource” on page 11.

In the COM terminology, a Family is strictly equivalent to an **Interface**. Interfaces are pure virtual classes inheriting from IMCUnknown (basically a COM definition). The Interface files are the I*.h headers.

Then Each Component has a **Class ID**. Its Class ID describes uniquely your Component. For example, in the Shader Family, there are different classes of shaders: the Checker Shader, the Marble Shader, the Wood Shader, etc. Each of this shader has a unique Class ID. Its Class ID is the Component’s key to its data and instantiation process. This is how your Component will be able to store its private data in a Carrara file and retrieve it later.

For this reason, if you intend to distribute your Component outside, even as a Freeware or Shareware, **it is vital that you register your Class ID with DAZ to ensure that it will be unique**. This Class ID will then be yours forever. This registration process is done by e-mail or fax, and it is free. See “Registering your Class ID with DAZ” on page 24 to learn how to register Class IDs.

In the COM terminology, a Class is strictly equivalent to a **Class**.

...And Instances

So far, this is all the 3D Shell does at startup. It simply identifies all available Components for later use. No Component code has been loaded or executed.

Then comes the time of instantiation. For example, a file is read, and it contains a 3D Object with a Marble Shader. The 3D Shell does not know anything about Marble Shaders. It just gets the Family ID and the Class ID. Looking up in its Component Class directory, it **instantiated** a Marble shader by asking the dll / shared library implementing it to create one. The Shell just keeps an anonymous pointer on the Component **Instance**, and this will be good enough to communicate with it: reading/writing its data, handling the user interface, executing it, etc. This will be explained in detail in the sections that follow.

Note that each instance has its own data values. For example, if another object in the file has another Marble texture on it, then a new instance will be created in order to store the parameters of this second Marble texture. This is how you can have several Marble textures with different vein spacing on different objects: they are different instances of the same Marble Class.

In the COM terminology, an Instance is strictly equivalent to an **Instance** (sorry COM-savvy readers, you got the point already...).

Communicating between the 3D Shell and the 3D Component

Once the Component has been instanciated, the 3D Shell will need to access the different Component services (i.e. routines). Think of the 3D Shell as the orchestra director. The 3D Shell will call your Component routines when needed. “Do not call us, you will be called” could be the 3D Shell motto.

However, there are times when your Component needs additional services from the 3D Shell. For example, a 3D Import Filter will have to be able to create and manipulate data structures in the 3D Shell Database. Therefore, the 3D Shell offers a complete API to do all kind of things. When a routine of your Extension is called, you can then call back the Shell to complete your job.

Component User Interface and Parameter Maps (PMap)

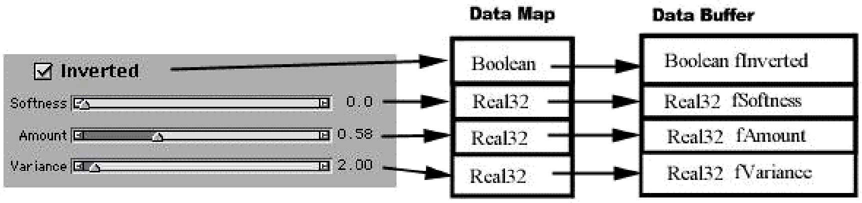
Often, the 3D Shell will want to show to the user the Component’s User Interface so its parameters can be set before executing. The 3D Shell has unique way to do this so the user interface of your Component is seamlessly integrated in the Shell user interface, providing a consistent and integrated experience to the user.

The key to this UI integration is the ‘**PMap’** resource. This resource is also the key to saving and reloading the Component’s data.

The ‘PMap’ resource describes a table called the **Data Extension Map**. During the initialization process, the Component told the Shell where its “public data” was stored: this is called the **Extension Data Buffer**, and it is located in the extension’s own RAM space. It is a record of the Component parameters (like the vein spacing values and the veins colors of the Marble Shader). The 3D Shell merely has an anonymous pointer on the Extension Data Buffer. The Data Extension Map will help the 3D Shell identifying the types and addresses of each data in the Extension Data Buffer, and build the relationship with the user interface.

Put simply, the Data Extension Map has two values for each entry: an ID to identify the user interface element (button, slider, etc.), and a type to identify the kind of data store in the Extension Data Buffer (Boolean, fixed, long, etc.).

This way, when the user changes something like the state of a check box, the Shell is able to compute the address of the corresponding item in the Data Buffer, and change it directly. Then it calls the Component and tells it that its data have been changed, so that the Component can react and update any internal pre-processed data.



Note that you do not have to write any code to handle your user interface. The Shell reads the resources describing your user interface from your Component file, and takes care of everything: it reacts when the user clicks on buttons, sliders, etc. because it knows about all these user interface elements.

The other advantage of this mapping system of the 3D Component public data is that it provides the 3D Shell with a simple way to save the Component data in a file and retrieve it later without any knowledge of the Component’s purpose in life. To store the Component data in the file, the Shell just writes the data ID and the value, and that’s it. Later, after the Component is instanciated, the 3D Shell will write back this data in the Extension Data Buffer.

You will find details on how to build the user interface itself (using the MCSketch, the user interface building tool) and how to compile a .dta file in the chapters that follow.

Creating an Extension

An extension is a shared library that defines components (COM Objects) that implement new functionality. In Carrara, every plug-in is composed of at least three files:

- The shared library (or dll): This is the executable of your extension. It has a "**.mcx**" extension.
- The resource file: This file contains the definition of the components contained in your plug-in. It has a "**.dat**" extension.
- The localized strings files: These files contains the strings of your component that should be localized. For each language you can create a file. There should be at least on file with a "**.txt**" extension but you can add a french file ending with "**FR.txt**" or other languages. Note that if you do not had a file for the current language then Carrara will automatically default to the English file.

All those files should have the same name (except for the language suffix). So for instance, if you create a plug-in called MyPlugin, the three files should be named respectively: MyPlugin.mcx and MyPlugin.dat and MyPlugin.txt.

Those files should be copied in the Extension subfolder of the Carrara folder so that Carrara recognizes them as plug-ins.

The following text explains in more details how you can create those files:

The resource files (.dat and .txt)

The resource files contains the definition of the components that are implemented in your plug-in as well as all the resources that are used for the user interface of your plug-in (dialogs, strings...)

For each component implemented in your plug-in, you have to create a ’COMP’ resource. The ’COMP’ specifies the type of extension (Family) and its name. In addition to the COMP resource, you have to create the “GUID” resource to tell the Shell that the extension is COM and to give the interface ID (IID) of the object and its class ID (CLSID).

Additionally you can declare a ’PMAP’ resource to define the parameters of your component. You can also create a ’Node’ resource with MCSketch which will be used as the user interface of your component.

All this resources should use the same Resource ID (the number that identifies the resource) so that the shell knows they belong to the same component.

Building the resource file

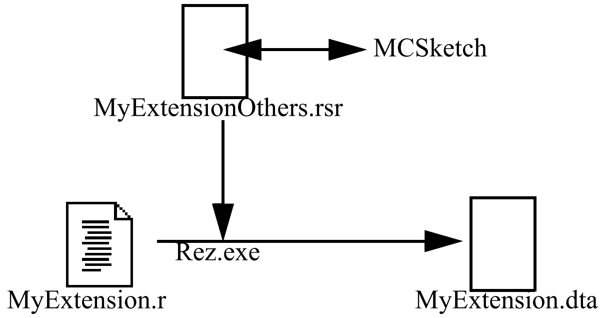
The resource file is built using a resource compiler. It takes as input two types of files:

- The **.r files** are text files that contain C like declaration of resources. You can edit them with an text editor.
- The **.rsr files** are actually compiled resource file that the resource compiler will merge into the resource file. You can create them with MCSketch.

On Windows:

Resource files (also called data files or .dta files) are built by compiling a .r file that contains the list of resources described in a text format. This format is the same as the Rez format defined by Apple Computer and used on the Macintosh. The result is a binary file that is called a resource file.

Text .r files can also include other (already compiled) resource files to build a more complex resource file. Some tools, like MCSketch, the UI builder provided with Carrara, builds a resource file that can be used with a resource compiler.



The Rez compiler can be found in the QuickTime toolkit on Apple Web site.

Steps to set-up the Rez.exe Compiler on Windows:

1. We suggest you use Rez, an Apple version of the Macintosh resource compiler, distributed as part of Apple's QuickTime SDK.
2. So go to http://developer.apple.com/quicktime/ and download the QuickTime SDK for Windows. (After downloading Rez you should get a folder (~8Meg) which has 5 sub-folders: CIncludes, ComponentIncludes, Libraries, RIncludes, and Tools.) We suggest you make this folder SDK/Tools/Rez, then you do not need to modify the BuildDTA.bat file.
3. Create an empty SysTypes.r file in QT RIncludes directory that was part of the Rez download.
4. Go into the SDK/Build/BuildDTA.bat file and make sure it uses the correct path to rez for your machine. If you put Rez where we suggest, this step is unnecessary.

An alternative solution is to compile the *.dta files on a Macintosh and copy them on your PC.

When you have created your .dta files, you have to separate them in .dat and .txt files. You can use for that the ResExtracter located in the folder SDK/Tools/ResExtracter. If you try to build the samples with MS Visual Studio, a script extracts automatically the resources for you.

On MacOS

The process on MacOS is very similar to Windows, and Rez is part of the developer tools.

When you have created your .dta files, you have to separate them in .dat and .txt files. You can use for that the ResExtracter located in the folder SDK/Tools/ResExtracter. If you try to build the samples with XCode, a script extracts automatically the resources for you.

Extracting the localized strings from .dta

To separate the **.dta** file in a **.dat** file and a **.txt** file use the ResExtracter in the folder SDK/Tools/ResExtracter.

For instance, to extract the strings from a file named myExtension.dta, you type:

ResExtracter myExtension.dta

This will generate two files: myExtension.dat and myExtension.txt

Those two files are the ones that should be placed into the extension folder next to the executable (**.mcx**). The file myExtension.txt contains all the strings that needs to be translated for a foreign version (for example in French). To create a localized version of your plugin in French, simply make a copy of this file named myExtensionFR.txt and translate the strings in French.

FR is the language code for French. Other languages will use other language code (for instance GE for german, JP for Japanese).

The ’COMP’ resource

The resource “COMP” describes the type of extension and some additional information.

Here is an example of a ’COMP’ declaration:

```
resource ‘COMP’ (128)     // resource ID = 128
{
        ‘shdr’,            // type of extension, the family name
        ‘COch’,            // Class ID, to identify your extension in the family
        "Checker",         // Name of your extension, it will appears in the Shell to
                            // describe and select your object
        "COM Examples",    // Sub Family Name, for a shader,
                            // it will create a new submenu
                            // called "COM Examples" in the menu of the Shader Editor
        1,                 // Version of your component
        "1.0",             // Version String
        "My Comments",     // Comments
        kRDAPIVersion      // API Version the Extension is based on
}
```

The fields in a COMP resource are:

- **Family ID** : It is a four characters identifier that specifies the kind (family) of component (for instance it could be a shader or a final renderer)
- **Class ID** : This four characters identifier identifies the component in the family. Note that it should be unique within its family (class IDs all in lower case are reserved for DAZ use). You should register your components with DAZ to make sure the Class ID you choose does not conflict with another one.
- **Component name:** This is the name of your component in the user interface of Carrara. It will appear in the menu where the user can select your component.
- **SubFamily name**: This is the name of the subfamily. It is sometimes used to create submenus (for instance for shaders). If the subfamily is “Hidden” then the component is not displayed in the User Interface.
- **Version of the Component**: This number is used for component versioning. This way you can still read previous version of component by providing a different component for each version. If there is only one version of your component, the component version should be 1.
- **Version String:** This string is not used at this time.
- **Comments:** This string is not used at the time
- **API Version**: use kRDAPIVersion to be compatible with the current version of the SDK

Attention: the version information of the ’COMP’ resource has changed in version 4

Here is the list of Extension Family ID and the corresponding COM IID for each type of Extension.

Table 1: Family ID and Interfaces of Carrara

Family	Family ID	COM IID
Ambient Light	ambi	I3DExAmbientLight
Animation Method	amet	I3DExAnimationMethod
Atmospheric shader	atmo	I3DExAtmosphericShader
Attached generic data	data	I3DExDataComponent
Backdrop	drop	I3DExBackdrop
Background shader	back	I3DExBackground
Camera transformation	came	I3dExCamera
ColorPickers	pick	I2DExColorPicker

Table 1: Family ID and Interfaces of Carrara

Family	Family ID	COM IID
Constraints	link	I3DExConstraint
DropAreas	drpA	IMFExDropArea
DropCandidates	drpC	IMFExDropCandidates
Export	3Dou	Ì3DExExportFilter
External Module	modu	I3DExModule
External Part	part	IMFExPart
External List View	part	IMFExListPart
Final Renderer	frnd	I3DExFinalRenderer
Gel	'gel '	I3DExLightsourceGel
ICC Processor	iccp	I2DExICCProcessor
Import	3Din	Ì3DExImportFilter
Interactive Renderer	RndB	I3DExRendererBox
Light	lite	I3DExLightsource
Modifiers	modi	I3DExModifier
Post Render Filter	post	I3DExPostRenderer
Primitive	prim	I3DExGeometricPrimitive
Reflection Render Features	fleF	I3DExReflectionFeature
Refraction/Transparency Render Features	fraF	I3DExRefractionFeature
Registerers	regi	IExRegisterer
Scene Modifiers	smod	I3DExModifier
Scene Commands (Scene Operations)	scmd	I3DExSceneCommand
Shader	shdr	I3DExShader
Shadow Render Features	shdF	I3DExShadowFeature
Terrain Filter	tfil	I3DExTerrainFilter
Tweener	twee	I3DExTweener
Volumetric Effects	Volu	I3DExVolumetricEffect

The 'GUID' resource

GUID stands for Globally Unique IDentifier, and is a 128-bit (16-byte) structure that uniquely identifies an interface.

When you create a COM object, you need to also create an unique GUID. To tell the Shell that your extension can be called with the COM mechanism, you have to create a GUID resource that contains the IID (Interface ID) and the CLSID (Class ID) of the COM object.

Microsoft provides two utilities: UUIDGEN.EXE (command-line) and GUIDGEN.EXE (a UI-based version of the same application), that generate unique GUIDs. MacOS users can use "Create GUID.ppc", a similar Microsoft utility.

You create the GUID declaration resource in the .r file associated with your component.

Here is an example of a ’GUID’ declaration:

```
resource 'GUID' (128)
{
        {
            R_IID_I3DExModifier,
            R_CLSID_ExBarycenter
        }
};
```

The definitions of `R_IID_I3DExModifier` and `R_CLSID_ExBarycenter` are usually stored in a .h file that is shared by the .r file and the .cpp files (it is just a convenience to avoid duplicating definitions). See the toolkit samples for an illustration of this little trick.

The ’PMap’ resource

A resource “*PMap*” is used to define all parameters your extension needs to save and restore.

Usually, they are user interface parameters, but they can be purely internal data which your extension needs to have saved so that it can restore its state from a file. For more information on PMaps and how they relate to your user interface, refer to “Auto PMap” on page 29.

Here is an example of a ’PMap’ declaration:

```
resource 'PMap' (128)
{
        {
        'axis','in32',noFlags,"Axis","",
        'prm2','re32',interpolate,"Parameter 2","",
        'prm3','re32',interpolate,"Parameter 3","",
        }
};
```

In the parameter map, you must provide the following information for each parameter:

- A four character identifier (for example ’axis’). This ID is used to identify the parameter. It is also used to establish the relationship between a widget in the User Interface and the parameter (the ID of the part that edits this parameter should be the same as the ID of the parameter).
- A four character identifier (for example ’in32’) that identifies the type of the parameter.
- A flag (interpolate means that the parameter is animated)
- The name of the parameter (this name is displayed in the hierarchy if the parameter is animated)
- An optional string that contains that describes the parameter.

Here is a list of the types that can be used in ’PMap’ resources:

Table 2: Types used in PMap

C++ Type	Keyword	Comments
Boolean	bool	= unsigned char.
short, uint16, int16	in16	
long, uint32, int32	in32	
float, real, real32	re32	Single precision floating point

Table 2: Types used in PMap

C++ Type	Keyword	Comments
double, real64	re64	Double precision floating point
TMCColorRGBA	colo	color
TColorRGB214	colF	Fixed color 2.14
TMCGradient	grad	Color grandient
TMCRect	rect	2D Rectangle
TMCPoint	poit	2D Point
TMCString255	s255	255 characters string
TMCDynamicString	Dstr	Dynamic string (unlimited length)
TVector2	vec2	2D vector
TVector3	vec3	3D vector
TVector3	dire	Direction parameter
TMatrix33	mx33	3 x 3 matrix
TSingleComponentChooser	comp	Component Chooser (allows the selection/creation of a component of one or more Families)
TMultipleComponentChooser	cmp#	Multiple Component Chooser (allows the selection/creation of one or more Families)
ActionNumber	actn	Same as an int32
IMCUnknown	iunk	Data is an anonymous IMCUnknown pointer

For a list of general PMap types, go to Common/PMapTypes.h.

Shaders and PMaps

PMaps for shaders have a specific feature that allows reusing the previous values of a shader when switching to another one. If some entries of the new PMap have the same IDs as some entries of the previous PMap, then the values are copied. This is just a convenience for the user, and is implemented only in the Shader Editor.

The executable file (.mcx)

Build the executable

For more information on how to build the shared library, see , “Platforms and Compilers issues”

Entry Points

Every plug-in for Carrara should have the following entry points:

- **MCDllGetClassObject()**: standard COM entry, used by the Shell to ask your code to instantiate your Extension through a Class Factory

- **MCDllCanUnloadNow()**: standard COM entry, not used in Carrara
- **MCDllInit()**: low-level initialization entry, used for memory allocation initialization, and other utilities initialization of the libraries.
- **MCDllCleanUp()**: counterpart of MCDllInit(), used for deallocation when unloading the DLL
- **MCDllInitExceptionTranslator()**: Failure Handling initialization
- **MCDllCleanUpExceptionTranslator()**: Failure Handling clean-up

The library provided in the Carrara SDK already implements these calls, so you don't have to implement any of those. However **it is important that your project properly exports them** (on Windows this is done by including a .def file in your project) otherwise your plug-in will not work. The actual routines that you will have to implement are the following:

- **Extension3DInit** is called when your plug-in is first loaded in memory (usually the first time one of the component defined in your plug-in is used). You can use this call to initialize global structures.

```
void Extension3DInit(IMCUnknown* utilities)
{
       // Perform your dll initialization here
}
```

- **Extension3DCleanup** is called when your plug-in is unloaded from memory (usually when the application is closed). You can use it to delete global structures for your extension.

```
void Extension3DCleanup()
{
       // Perform any necessary clean-up here
}
```

- **MakeCOMObject** is called to create the components that your plug-in defines.

To instantiate your extension in `MakeCOMObject`, using a simple `new` command:

```
TBasicUnknown* MakeCOMObject(const MCCLSID& classID)
{
       if (classID == CLSID_MyExtension)
          return new TMyExtension;

       return NULL;
}
```

Your extension should always inherit more or less indirectly from TBasicUnknown.

The Component Class

Each component in your extension is implemented by a C++ class. This class implements one or several COM interfaces. For example, a shader will always implement I3DExShader which is the interface for the shader family. Additionally it could implement IExStreamIO if it needed to save extra data that are not stored in its parameter map.

To make it easier to get started, a number of "basic" classes that provide a default implementation of each family of component have been provided. These classes have a name that starts with **TBasic** (for example TBasicShader or TBasicFinalRenderer). You should derive your object from one of those classes if there is one available for the type of component you are implementing. In any case, your class should always derive from **TBasicDataExchanger** (either directly or through one of the Basic class) that provides the implementation for a basic component class.

So the declaration of your class should look this:

```
class MyComponent : public TBasicShader
{
public:
       static const MCGUID sClassID;

       MyComponent();
       ~MyComponent();

       STANDARD_RELEASE;
```

```
        virtual void* MCCOMAPI GetExtensionDataBuffer();
};
```

If you do not want to use the basic class (or if there is not one available), you could declare your class as follows:

```
class MyComponent : public TBasicDataExchanger,public I3DExShader
{
public:
        static const MCGUID sClassID;

        MyComponent();
        ~MyComponent();

        MCCOMErr MCCOMAPI QueryInterface(const MCIID& riid, void** ppvObj);
        uint32  MCCOMAPI AddRef();
        STANDARD_RELEASE;

        virtual void* MCCOMAPI GetExtensionDataBuffer();
};
```

A few remarks about the declaration of the class:

The field **sClassID** is used to store the ClassID of your extension so that you can use it in **MakeCOMObject**.

Even if you derive from the basic class, you still need to implement STANDARD_RELEASE. The reason for this are a little bit complicated so we will ignore them for now. The thing to remember is that you should always implement STANDARD_RELEASE at the bottom for your inheritance tree (in this case, your component).

TBasicUnknown

When you create a component you should implement the 3 methods of IMCUnknown: AddRef, Release and QueryInterface since your object derives from TBasicUnknown

Usually those methods are implemented by the basic class (except for Release that you should always implement in your class). However it is important to understand how they work so here is how to implement each of them:

AddRef()

You should simply call the parent of your class. TBasicUnknown provides the proper implementation for AddRef.

```
uint32 MyComponent::AddRef()
{
        return TBasicDataExchanger::AddRef();
}
```

Release()

You always use the STANDARD_RELEASE macro.

QueryInterface()

QueryInterface needs to return the proper interface if it is implemented by your component. The basic class provides a default implementation for the interfaces it derives from.

```
MCComErr MyComponent::QueryInterface(const MCIID& riid, void** ppvObj)
{
        if (MCIsEqualIID(riid, IID_I3DExShader))
        {
           TMCCountedGetHelper<I3DExShader> result(ppvObj);
           result = static_cast<I3DExShader*>(this);
           return MC_S_OK;
        }
        return TBasicDataExchanger::QueryInterface(riid, ppvObj);
}
```

User Interface

Carrara makes it quick and easy to provide a simple user interface for your component.

The ’Node’ resource.

For most components Carrara uses a ’Node’ resource for the user interface. A ’Node’ resource contains a Node Part.

You can create such a resource with MCSketch which is provided with the SDK. The resource ID of the resource should be the same as the resource ID of the ’COMP’ resource.

To change the resource ID in MCSketch:

- Select the resource.
- Select **Resource Info** from the **Edit** menu (or type Ctrl+I)
- Change the resource ID in the dialog.

If you want to use a different Node Part (for instance to share a node part between several component), you can implement **TBasicDataExchanger::GetResID()** and return the ID of the resource ID that you can to use.

In the ’NODE’ resource you can add various controls (sliders, buttons, check boxes...). Each control has a Part ID (the Part ID can be edited in the properties drawer on the right). Set the Part ID of each control to the four character ID of the parameter in the parameter map (see “The ’PMap’ resource” on page 13). The Shell will automatically map the value of the control with the value of the parameter and create key frames if necessary.

The Pmap buffer

Once you have set up the node part and the ’Pmap’ resource, there is only one last thing that is missing: your class needs to access the value of the parameter. For this purpose, your component must provide a buffer to store the value of its parameters at the current time.

This buffer must match exactly the ’Pmap’ resource. It is important to be careful that the fields have exactly the same type and are exactly in the same order as in the ’Pmap’ resource.

Here is an example:

```
struct myPMAP
{
        int32 fAxis;
        real  fParameter2;
        real  fParameter3;
};
```

corresponds to the following ’PMap’ resource:

```
resource 'PMap' (128)
{
        {
        'axis','in32',noFlags,"Axis","",
        'prm2','re32',interpolate,"Parameter 2","",
        'prm3','re32',interpolate,"Parameter 3","",
        }
};
```

To let the Shell know, where the buffer is located, you need to implement `GetExtensionDataBuffer()` in your class:

```
virtual void*    MCCOMAPI GetExtensionDataBuffer()
{
        return &fMyPmap;
}
```

That is all you need to know to create a simple user interface for your component.

For more information on parameter maps, see the documentation of TBasicDataExchanger.

Using DAZ’s COM Dynamic Linking

This chapter describes in more detail COM Dynamic Linking supported in Carrara.

Before reading this chapter, it is highly recommended to read the Introduction chapter, especially the section “How the whole thing works” on page 4.

About COM

COM (Component Objects Model) is the low-level dynamic linking system on which OLE is built. It is a very widely spread industry standard, and is actively promoted by major industry players. So if you are already familiar with OLE, you will find it easy to be started with this documentation. COM offers a nice and clean C++ like interfacing.

The best reference for COM is “Inside COM” by Dale Rogerson from Microsoft Press (ISBN 1-57231-349-8). The first chapters of “Inside OLE 2” by Kraig Brockschmidt from Microsoft Press are also a good introduction (ISBN 1-55615-618-9). You can also find all the necessary information in any edition of the MSDN (Microsoft Developers Network) CDs. On the Web, look at http://www.microsoft.com/com/.

Component Registration Process

Component registration provides true auto-plug and play installation of components. Simply drop component files in the correct folder and run. You don’t need to use the Windows registry or full-path names. There is no conflict between versions or languages (due to being based on COM).

Each component has one COMP and one GUID resource associated with it. Each extension file (.mcx file) can have more than one component by defining multiple COMP and GUID resources. To learn more about these resources refer to “The resource files (.dat and .txt)” on page 9.

How the Component Server Works

At start-up the Component Server identifies components in the application directory and below. It looks for .mcx extension. For each extension file found, it looks for a resource file with the .dta suffix. For each of these files, it looks for more components; it identifies components by a ‘COMP’ resource and adds the component to a list. There is one list of components for each family.

In additional to the general component registration resources, each type of component may have its own specific resources. For example, a module has a ‘Modu’ resource that adds on functionality that modules can do. To learn more, refer to the chapter on family-specific resources in the Cookbook.

How the Application Calls your Extension

“Simple” Components are extensions that do not call back into the Shell or application to perform their duties. Such extensions are typically Primitives, Shaders, Light sources, Ambient lights, Gels, Atmospheric Shaders, Background and Backdrop Shaders, and Cameras.

Extension Entry Points

Your COM Extension has these entry points:

- **MCDllGetClassObject()**: standard COM entry, used by the Shell to ask your code to instanciate your Extension through a Class Factory
- **MCDllCanUnloadNow()**: standard COM entry, not used in Carrara
- **MCDllInit()**: low-level intialization entry, used for memory allocation inits, and other utilities intialization of the libraries.

- **MCDllCleanUp()**: counterpart of MCDllInit(), used for deallocation when unloading the DLL
- **MCDllInitExceptionTranslator()**: Failure Handling initialization
- **MCDllCleanUpExceptionTranslator()**: Failure Handling clean-up

The library provided in the Carrara SDK already implements these calls, so you won’t have to worry too much about them (just make sure that they are exported properly by your DLL or Shared Library). The library also already implements the Class Factory scheme and everything, so the actual routines that you will have to implement are these:

```
// Initialization routine:
void Extension3DInit(IMCUnknown* utilities);

// Clean-up routine:
void Extension3DCleanup();

// Instantiation routine:
TBasicUnknown* MakeCOMObject(const MCCLSID& classID);
```

That’s it. The library takes care of the rest for you.

Extension3DInit is called when your plug-in is first loaded in memory (usually the first time one of the component defined in your plug-in is used). You can use this call to initialize global structures.

Extension3DCleanup is called when your plug-in is unloaded from memory (usually when the application is closed). You can use it to delete global structures for your extension.

MakeCOMObject is called to create the components that your plug-in defines.

Instantiate your extension in `MakeCOMObject`, using a simple `new` command:

```
TBasicUnknown* MakeCOMObject(const MCCLSID& classID)
{
        if (classID == CLSID_MyExtension)
           return new TMyExtension;
        else
           return NULL;
}
```

Your extension should always inherit more or less indirectly from TBasicUnknown (see the discussion about the TBasic* types later on).

Minimal resources

Make sure you choose a CLSID and that you copy its value in the GUID resource (see “The ’GUID’ resource” on page 12) for more details. Make sure that all your COMP, PMap, GUID, Node, etc. resources have the same ID number and that this number is the one returned by **GetResID**. You still need to choose a 4-letters Class ID and put it in your COMP resource.

And, once again, Windows programmers do not need to enter their IID and CLSID in the Windows Registry Database. Simply give your Component the .MCX suffix, and put it in the Extensions directory (or in any sub-directory of the Extensions directory). Carrara will find it automatically.

Use the samples of the Carrara SDK as a framework to start your own extensions.

How your Extension Calls into the Application

Some extensions need to call back the Shell. For example, an Import Filter may need to create 3D objects and put them in the scene.

There can be three types of situations:

- A Shell object pointer is passed to you as a parameter of one of your Extension procedures.
- A Shell object pointer is provided to you as a global variable.
- You need to instanciate a new Shell object and work on it.

Using Shell objects received as parameters

This is a very simple case. The Shell already allocated an object and gives you its pointer as the parameter of one of your Component procedures.

Example:

```
IMFExResponder::Receive(int32 message, IMFResponder* source, void* data);
```

The *source* parameter is a Shell object of the **IMFResponder** Interface. You simply call its methods as you wish. There is no need to call **AddRef**() on it. Do not call **Release**() either, the Shell will take care of its own children.

Example:

```
int32 instanceID;
instanceID = source->GetInstanceID();
```

Using preset Shell objects stored in the library global variables

Many useful methods can be accessed via global pointers pointing to objects of the Shell. The global utilities pointers are grouped by functionality into different COM interfaces (gShellUtilities, gResourceUtilities, ...). These interfaces can all be found in the header files called *something*Utilities.h. All of the global variables are initialized by the extension’s entry points MCDllInit and MCDllCleanUp, so you do not need to worry about that.

Example:

```
uint32* myArray; // array of 640*480 unsigned long
myArray = gMemoryUtilities->Calloc(sizeof(uint32), 640*480);
...
MCCOMErr result = gMemoryUtilities->Free(myArray);
```

Instanciating a Shell object yourself

There are times when you need to create an object, like creating a 3D Primitive to put it in the Scene.

The procedure is a little bit unusual for COM programmers. Instead of calling the global procedure **CoCreateInstance**(), you will call **IShComponentUtilities::CoCreateInstance**() on the *gComponentUtilities* or *gSh3DComponentUtilities* global variable (depending on your needs).

If you need to create a component, you should use **IShComponentUtilities::CreateComponent** so that the internal component is also created.

Example:

We create a Group and put it in the Scene:

```
HRESULT TDXFImporter::DoImport(char* fullPathName, I3DShScene* scene,
I3DShTreeElement* fatherTree)
{
  TMCCountedPtr<I3DShGroup> group;
  gSh3DComponentUtilities->CoCreateInstance(CLSID_StandardGroup, NULL,
CLSCTX_INPROC_SERVER, IID_I3DShGroup, (void**) &group);
  // Do whatever we want on the group
  ......

  // Use the group as a Tree Element to put it in the scene:
```

```
  TMCCountedPtr<I3DShTreeElement> groupTree; // The same as group, but as a Tree Element
  group->QueryInterface(I3DShTreeElement, (void**) &groupTree);
  groupTree->SetScene(scene);
  fatherTree->InsertLast(groupTree);

  //Note: you do not need to call Release() as TMCCountedPtr will do it automatically.
}
```

The “PMap” resource

A resource “*PMap*” describes the different parameters that the Shell can access. The PMap is key to UI integration as all components with UI have a PMap resource.

The PMap describes the UI data for the shell. There are 4 values in the PMap for each UI element:

- a four letters ID (to identify the element)
- a four letter type code (boolean, long, etc.)
- a flags field (animate this element, etc.)
- the name of the element (appears in the Time Line)
- an extra string containing optional additional data

Whether you create your resource on the Mac or PC, your “PMap” should look like this:

```
resource ‘PMap’ (128)
{ /* First Array Element */
  ’SIZH’,        /* ID of the UI element in the view */
  ’in32’,        /* type of the value */
  interpolate,   /* value can be animated */
  "Hori. Size",  /* name */
  "",            /* Optional data */
  /* Second Array Element */
  ’SIZV’,        /* ID of the UI element in the view */
  ’in32’,        /* type of the value */
  interpolate,   /* value can be animated */
  "Vert. Size",  /* name */
  "",            /* Optional data */
}
```

To learn more about PMaps and PMap types, refer to “The ’PMap’ resource” on page 13, “Auto PMap” on page 29, and “PMap and a Part” on page 30.

IMCUnknown class

What is a COM object?

A COM object is any object derived from IMCUnknown (simple inheritance only).

```
class IMCUnknown
{
public:
       virtual MCErr MCCOMAPI QueryInterface(const MCIID& riid, void** ppvObj)=0;
       virtual uin32 MCCOMAPI AddRef()=0;
       virtual uin32 MCCOMAPI Release()=0;
};
```

It is a key element of the Carrara COM implementation as it provides a way for a client to communicate with a component. These functions are explained in greater details in the sections that follow.

When you are creating components to be used with Carrara, you always inherit more or less indirectly from IMCUnknown.

QueryInterface

QueryInterface is the cornerstone of COM. In COM, a client always communicates with a component through an interface. QueryInterface defines the component; the interfaces that a component supports are the interfaces for which QueryInterface returns an interface pointer. You use QueryInterface to discover whether a component supports a particular interface. If the component supports the interface, QueryInterface returns a pointer to that interface.

```
virtual MCErr MCCOMAPI QueryInterface(const MCIID& riid, void** ppvObj)=0;
```

The first parameter is the Interface identifier structure or the **IID**. The second parameter is the address where QueryInterface places the requested interface pointer.

If you use the following notation:

if "ObjB = A->QI('B')" means: ObjA->QueryInterface(REFIID_B, &ObjB)

QueryInterface should follow the following rules:

```
(1) A->QI('A') == A
(2) (A->QI('B'))->QI('A') == A
(3) ((A->QI('B'))->QI('C'))->QI('A') == A
```

AddRef

AddRef and Release use reference-counting to manage memory. Reference counting enables components to delete themselves (once they are no longer being used). AddRef increases the reference count and Release decrements the reference count.

When do you use AddRef?

- If you are asked for an object by someone, call AddRef before returning a pointer to it.
- If you plan to keep a reference on an object passed as a parameter in a function, the function needs to call AddRef and then Release when done.

To learn methods that allow you to optimize reference counting and avoid using AddRef, see the MCCountedPtr template. Refer to , "Counted Pointers" to learn more.

Release

As mentioned earlier, Release decrements the reference count. When you are finished with an interface, you should call Release on that interface.

When do you use Release?

- QueryInterface increments the count; when done, call Release.
- A function you call to get an object should have incremented the count, so you only need to call Release.
- If you plan to keep a reference on an object passed as a parameter in a function, the function needs to call AddRef and then Release when done.

To learn methods that allow you to optimize reference counting and avoid using Release, see the MCCountedPtr template in , "Counted Pointers"

More on QueryInterface, AddRef and Release

It is highly recommended to read carefully , "Counted Pointers" for more details about using QueryInterface, AddRef and Release.

Registering your Class ID with DAZ

As explained in the section "How the whole thing works" on page 4 in the Introduction chapter, it is vital that you make sure that your Class ID is unique before distributing your software outside, even as a Freeware or Shareware. The reason is that if someone else uses the same Class ID in his own extension, the 3D Shell will be confused when trying to instantiate and initialize your 3D Component, causing a failure or a crash.

Registering your Class ID is free. There are no obligations. After acceptation, your Class ID will be yours forever and should never be assigned to someone else.

- A Class ID is a set of 4 ASCII characters.
- Ranges of Class ID will **not** be registered by DAZ (like "all IDs from 'AAAA' to 'AAAZ'").
- Class IDs are case-sensitive.
- Class IDs with all lowercase letters are reserved to DAZ (like 'abcd'). Combos are fine, though (like 'Abcd').
- Try avoiding special characters (ASCII code > 128), because these characters are different between MacOS and Windows.

How to register

Simply email the following information to **sdk@daz3d.com**. Make sure you include the word "registration" in your e-mail title to allow faster processing.

Name:
Company:
Address: (if sending via mail)
Phone:
Fax: (if sending via Fax)
E-Mail:

Component Name	Family ID	Class ID

Please include a brief description of your Component(s):

Send this information to:

E-Mail: sdk@daz3d.com

You will then receive a confirmation message from DAZ by email. If a conflict occurs, DAZ will offer an alternate ID.

Contacting DAZ - Developers Mailing list

If you have any questions regarding the SDK, you can join the SDK user list at:

http://groups.yahoo.com/group/carraraSDK

where the engineering team of Carrara will answer your questions.

Platforms and Compilers issues

This chapter describes details about specific issues related to compilers and platforms.

Compiling for Windows

The supported compiler is Visual C++ 2005 or better from Microsoft. You should install the Service Pack 1.

You can use the Express edition, but you'll also have to download the platform SDK. This page explains how to setup everything : http://msdn2.microsoft.com/en-us/express/aa700755.aspx

Note that COM extensions are simply standard 32-bit DLLs with a **.mcx** suffix. Take a look at the sample make files in the Carrara SDK and you'll see that there are no special tricks there. Data files (files containing resources) have the .dat suffix.

Creating a new extension project for Visual C++ 2005:

The first step is to create a new project. You should create this project within the SDK hierarchy. You can create it next to an existing sample like CarraraSDK\Samples\Shaders\MyShader. If you create the project outside the samples folder, you should create sub folders that follow the same structure (like CarraraSDK\MyProjects\MyShaders\MyChecker). This avoids any problems with the relative path names given below. Otherwise you'll need to adjust the number of dots in the given paths.

1 Create a new, empty project. Set the project name and location within the samples folder, then set the project type to Win 32 Dynamic-Link Library.

2 Open the project settings.

2.1) In the C/C++ tab:

Select the "Project Options" field and add "@..\..\..\Build\CPFlags.opt" as the 2nd to last item (before /c). (The number of dots depends on the position of your project relative to the file). This is an options file which sets the include paths. Make sure the file is updated to have the correct paths.

2.2) In the Link tab:

Change the output filename to myExtension.mcx (instead of myExtension.dll).

Add the following to the Object/library modules line of the Debug and Release builds respectively:

..\..\..\lib\win\debug\CommonLibrary.lib

..\..\..\lib\win\release\CommonLibrary.lib

You may also need to add other libs (mpr.lib version.lib vfw32.lib)

2.3) In the "Post-build step" tab

Add the following two commands (in that order):

```
call ..\..\..\Build\BuildDTA.bat Debug MyExtension
```

```
call ..\..\..\Build\CopyMCX.bat Debug MyExtension
```

Compiling for MacOS

The supported compiler is Xcode 2.4 on Mac OS 10.4.

COM extensions are standard **Shared Libraries** with a **.mcx** suffix. Data files (files containing resources) have the **.dat** suffix in their name.

Setting up Carrara :

The sample projects are setup to use Carrara 6 Pro for debugging after copying the sample extension files directly inside Carrara. If you installed Carrara in a different location or if you do not have the Pro version, see below how to use a different Carrara folder.

Compiling one sample

- With Xcode, open the project "Proj.xcodeproj" located in the sample's folder (eg Samples/Atmospheres/Fog).
- Choose your active build configuration in the popup (Debug or Release)
- Click on Build
- If successful, you can click on the Debug button to launch Carrara and test the plugin.

Compiling all the samples

- Open a terminal window and cd to the BuildMac folder of the sdk.
- Type "./copyProjects.sh fog" to use the fog project as a template for the other samples.
- Type "./compile.sh debug" to compile all the samples in debug.
- Type "./compile.sh debug clean" when you need to remove the obj files for all the samples.

Using a different Carrara folder

- Open one of the projects with Xcode (eg Samples/Atmospheres/Fog).
- Open the project's info panel and select the build tab.
- Scroll to the bottom and edit the CARRAPATHPRO variable.
- Close the project's info panel.

Creating a new project for XCode

All the project files for the sdk samples are exactly the same (a script properly renames the resources and the dll when copying them to the Carrara folder). Therefore, we strongly recommend that you use the same template for your projects.

Debugging on MacOS

You can debug your extension using the XCode build-in debugger:

1. Build your extension using XCode build , or using the ./compile.sh as explained earlier.

2. Click on the Debug button to launch Carrara and test the extension.

User Interface

MCSketch is a resource editor provided with the SDK. The resources used in Carrara are fully cross-platform, and can be created on either a Macintosh or a PC, since MCSketch exists on both platforms.

In this part, we will describe all the needed resources on both platform (Macintosh or Windows).

Make sure you are familiar with the main concepts of Components before reading this chapter. Specifically, be sure you read "How the whole thing works" on page 4.

User Interface Options

When you are creating an extension, you need to decide which strategy to use for creating the user interface. For example, if you are creating an extension such as an importer or exporter, you may not even need to create a user interface. Refer to "No User Interface" on page 29 to learn more. Once you've determined that your extension needs a mechanism for user interaction, you still have several different options for creating the user interface. If you only need simple controls, such as a slider or radio button, you can use an automatically generated PMap. Refer to "Auto PMap" on page 29 to learn more.

However, a PMap combined with a **Part** definition allows you the greatest flexibility in creating a user interface for your extension. Refer to "PMap and a Part" on page 30 to learn more.

To summarize, your extension can provide:

- No user interface
- An auto-generated user interface (Auto-PMap)
- A user interface specified with a static resource. In this case:
  - the data buffer is automatically maintained
  - many user interface elements are available: check box, edit text, color picker, component chooser, etc.
- A user interface generated from dta file, using standard MCFrame user interface elements
- A user interface generated from dta file, using extended MCFrame user interface elements (**Part Extensions**)

No User Interface

If you are creating an extension such as an importer or exporter, or if you have created a renderer, such as a postrenderer or final renderer, you may not need to create any user interface elements for your extension; it simply functions without user interaction.

Refer to the Cookbook and the Samples folder to see several examples of extensions created without a user interface.

Auto PMap

A resource "*PMap*" describes the different parameters that the Shell can access. All components with user interface have a PMap resource. In some cases, a PMap is all that you need to create the user interface for your component.

Here is an example of a PMap:

```
resource 'PMap' (129)
{
        {   // TODO: Add your parameters by providing:
            //   four letters ID,
            //   four letters type ID,
            //   interpolate or zero,
            //   name
            //   extra token string (usually "")
             'WIEG','re32', interpolate, "Weight!","", // this parameter is animated
```

```
                //'WIEG','re32',noFlags,"Weight","", // this parameter is NOT animated
            }
    };
```

PMap and a Part

If your extension needs a user interface (UI), and what is provided by the Auto-PMap is not enough, you can use a "part".

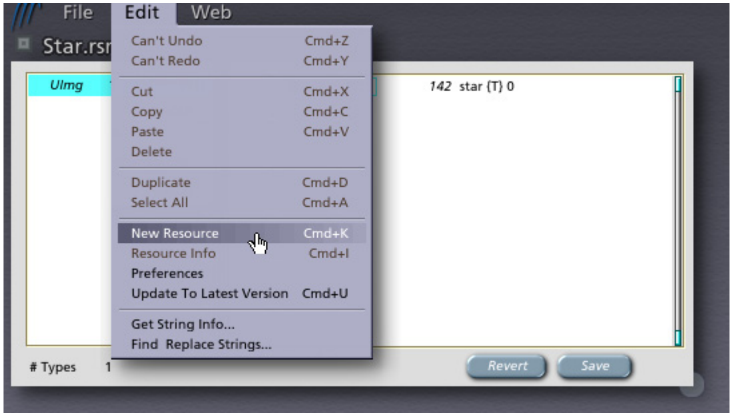
A part is a piece of user interface. You create and edit a part using MCSketch. The main benefit of using a part instead of an Auto-PMap is that you can personalize the UI of your extension with your icons, your names and copyrights, your URL, and so on. The ideal is to let the Auto-PMap do the job while you debug and test the algorithms of your extension, and in the end finalize the extension with the UI.

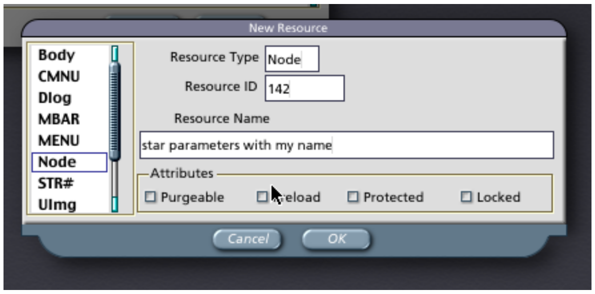
You can combine a PMap with a part definition contained in a .rsr file. If the PMap and the part have the same number, they automatically hook together. Note that the PMap is defined in a .r file, which must include the .rsr file.

For example, the Star sample has no part in its .rsr file, Star.rsr. The PMap defined in Star.r is:

```
resource 'PMap' (142)
{
        {
            'NBBR','in16',interpolate,"# Branche","",
        }
};
```

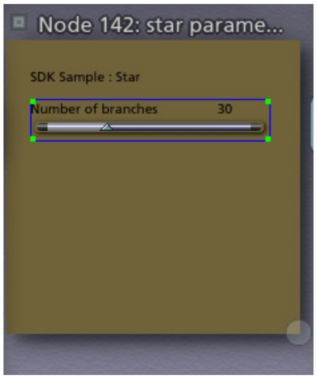
Since there is no part of ID 142, the Star sample uses the Auto-PMap mechanism (which is enough for testing). If you want to add your own UI, open Star.rsr in MCSketch, create a 'New Resource' via the menu 'Edit', and choose the type 'Node'. A node is a part that can contain other parts.





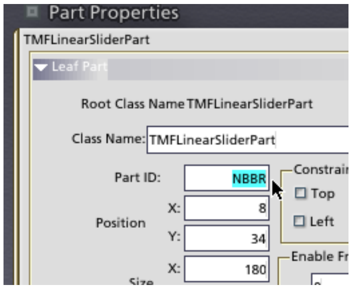
Resource Type: Node
Resource ID: 142
Resource Name: star parameters with my name

Insert in that Node part all the UI elements that you need. For example insert a 'Static Text part' to show your copyright, and a 'Linear Slider part' to display and edit the parameter of the PMap. Edit the ID of the slider part to make it fit the ID of the parameter in the PMap: NBBR. This allows the Shell to match the PMap fields and the UI parts as shown.



SDK Sample : Star

Number of branches 30



Root Class Name TMFLinearSliderPart

Class Name: TMFLinearSliderPart

Part ID: NBBR

Position X: 8

Y: 34

Size X: 180

☐ Top

☐ Left

Samples using a PMap and .rsr file

The introduction to the cookbook provides a list of samples that demonstrate creating different user interface elements. To see examples that use a PMap and an rsr file (created using MCSketch), to define the following user interface elements, refer to the samples.

- Checkbox
- Color chooser
- Icon
- Radio button group
- Scrollable edit text
- Sliders
- Dialog boxes
- Windows
- Menus
- Properties (set up and update)
- Sequencer hierarchy

Part Extensions

Another way to create a user interface, is to create **Part Extensions**. Once you create a part extension, you need to tie it to a part of your dialog or window to customize its behavior.

To create a part extension:

1. Create the resource with MCSketch. For example, you could create a node part. Then, add user interface elements such as a slider.
2. Make sure the ID of the part is the same as the resource ID of the PMap. (This is the number in the resource list in MCSketch, and the resource number in the .r file.)
3. Verify that the first field of the PMap in the .r file is set to the part ID (for the parts within the node).
4. Make sure that the object parameters (in the .h file) have a field for each element and that the size of the field matches the PMap element's second field (which is size).
5. Include the resource file in the .r file. (#include new.rsr).

For an example, refer to "Tab Part" on page 35.

Manually Controlling Your Own User Interface

In some cases, you'll need to manually control the user interface for your component. The following samples show you how to create and control specific user interface elements.

Creating your User Interface Containers

A dialog box and a window are both examples of user interface containers. The sections below, show how to create them.

Using a Dialog box

To manage dialogs, you must get used to the following interfaces: IShResourceUtilities, IShPartUtilities, and IMFDialogPart.

Here is some pseudocode that shows how to create a dialog box.

```
TMCCountedPtr<IMFPart> dialogPart;
TMCCountedPtr<IMFDialogPart> dialog;
void* oldResources = NULL;
boolean result = false;

//-- Load the resources from the DTA file
gResourceUtilities->SetupComponentResources(yourFamilyID,
yourClassID, &oldResources);
gPartUtilities->CreatePartByResource(&dialogPart,
kMFDialogResourceType, yourDialogID);
gResourceUtilities->RestoreComponentResources(oldResources);

//-- Now open the dialog
dialogPart->QueryInterface(IID_IMFDialogPart, (void**)&dialog);
result = dialog->Go();

//-- Get the value if we exited thru OK
if (result)
{
   ... do something ...
}

//-- We are done with the dialog
dialog->Finished();

return result;
```

To learn more about TMCCountedPtr, refer to "Implementing a Counted Pointer Access Method" on page 53.

Using a Window

A window can also be created from resources. That is mostly useful when you create a modeler and therefore need to have a window of your own. Here is sample code of the creation of a window. This piece of code comes from the Modeler sample.

```
TMCCountedPtr<IMFPart> window;
TMCCountedPtr<IMFPart> mainPart;

QueryInterface(IID_I3DShModule, (void**) &shModule);
ThrowIfNil(shModule);

void* oldResources = NULL;
```

```
        gResourceUtilities->SetupComponentResources(kRID_ModuleFamilyID, kMode-
lerID, &oldResources);

        shModule->CreateWindowByResource(&window, kModelerView, true);
        ThrowIfNil(window);

        // Retrieve a pointer on the main part, which ID is ‘MODv’ (see in MCSketch)
        window->FindChildPartByID(&mainPart, 'MODv');
        ThrowIfNil(mainPart);

        // Restore the resources chain.
        gResourceUtilities->RestoreComponentResources(oldResources);
```

Once your window exists, you can show it or hide it, activate or deactivate it. To do so, use the IMFWindow interface, and call Show() and Activate(). You should only need to do that in the I3DExModule’s methods Hydrate/Dehydrate, and Activate/Deactivate.

Check the Modeler sample for implementation.

Setting a Child Part’s Values

Before opening an dialog, you must set up the dialog to the current values, the default values, or the last values that the user entered. To do so, you must first find a pointer on the part’s interface. That is done with IMFPart::FindChildPartByID. Use this method on a part at the top of you parts hierarchy to find the item you want to modify, and then change the value of that item either by using its own interface or via the regular IMFPart interface. If we rework the example of the dialog box given above, here is what we can have:

```
TMCCountedPtr<IMFPart> dialogPart;
TMCCountedPtr<IMFDialogPart> dialog;
void* oldResources = NULL;
boolean result = false;

gResourceUtilities->SetupComponentResources(yourFamilyID,
yourClassID, &oldResources);
gPartUtilities->CreatePartByResource(&dialogPart,
kMFDialogResourceType, yourDialogID);
gResourceUtilities->RestoreComponentResources(oldResources);

//-- Now let's say you want to set a item value before opening the dialog:

TMCCountedPtr<IMFPart> editText;
real32 value;
dialogPart->FindChildPartByID(&editText, yourItemID);
if (!editText) return result;
value = 1.0;
editText->SetValue( (void*)&value, kReal32ValueType, true, false);


//-- Now open the dialog
dialogPart->QueryInterface(IID_IMFDialogPart, (void**)&dialog);
result = dialog->Go();

//-- Get the value if we exited thru OK
if (result)
{
   editText->GetValue( (void*)&value, kReal32ValueType);
   ... do something with it...
}
```

```
//-- We are done with the dialog
dialog->Finished();

return result;
```

Most of the parts can be accessed using the IMFPart interface. For example, the TMFStaticTextPart, the TMFEditTextPart, the TMFLinearSliderPart, and the TMFRadioClusterPart are handled by the IMFPart interface. Then of course, various IMF* child part examples tie into different IMF* interfaces, like tab parts tie into IMFTabPart.

Examples of Specific user interface elements

The following examples show how to implement specific user interface elements.

Registration Dialog box

Here is a code snipet that shows a registration dialog that collects a name and number. This illustrated how to use text boxes (EditTextPart parts). Before getting to the code, you need to create the appropriate items in MCSketch:

- Create a ’Dlog’ resource and choose an ID for it (say 1000)
- Put 2 TMFEditTextPart parts inside your dialog. If you drag and drop the Edit Text element of the Part Palette on the left, it comes inside a TMFScrollPart, which will allow your text to scroll. Be careful to select the TMFEditTextPart and not the TMFScrollPart (look at the Part Properties palette on the right)
- Assign a 4 letters "part ID" to each of your TMFEditTextPart parts (say ’Name’ and ’SeNb’).
- Add some Static text parts to have a nice looking dialog. Give a title to your dialog.

```
boolean RegistrationDialog(TMCString& name, TMCString& serialNumber)
{
TMCCountedPtr<IMFPart> dialogPart;
TMCCountedPtr<IMFDialogPart> dialog;
void* oldResources = NULL;
boolean result = false;

name = kNullString;
serialNumber = kNullString;

//-- Load dialog resources
gResourceUtilities->SetupComponentResources(yourFamilyID,
yourClassID, &oldResources);
gPartUtilities->CreatePartByResource(&dialogPart,
kMFDialogResourceType, 1000);
gResourceUtilities->RestoreComponentResources(oldResources);

//-- Find dialog items
TMCCountedPtr<IMFPart> nameTextPart;
TMCCountedPtr<IMFPart> serialTextPart;

dialogPart->FindChildPartByID(&nameTextPart, ’Name’);
if (!nameTextPart) return result;
dialogPart->FindChildPartByID(&serialTextPart, ’SeNb’);
if (!serialTextPart) return result;

//-- Now open the dialog
dialogPart->QueryInterface(IID_IMFDialogPart, (void**)&dialog);
result = dialog->Go();

//-- Get the value if we exited thru OK
if (result)
{
   nameTextPart->GetValue( (void*)&name, kStringValueType);
   serialTextPart->GetValue( (void*)&serialNumber, kStringValueType);
}

//-- We are done with the dialog
dialog->Finished();

return result;

}
```

Tab Part

To create a tab part you'll need to set up the .r and .rsr files as described below.

In the .r file:

- Add a new part in your .r file, that lists all the tabs you want to include in your user interface.
- Create a new resource, called TABS, and select an ID number for this new part. The ID number must be within the range of 127 and 5,000.
- Create a list of all your pages, respectively giving another ID for your page, its name and the ID of the icon with which you want to associate it.

You will obtain something like this:

```
resource 'TABS' ( id_of_tab )
{
      {
           id_of_first_page, name_of_the_page, id_of_the_assiociated_icon,
           ...
       }
}
```

In the .rsr file:

In this file, you first have to create a TabPart. To do so, create a simple node and change the class name into TMFTabPart. This part will contain two other nodes which will be the hosts for the icons and pages areas respectively. Then, put on some parameters in the Custom Tokens area. There actually are four possible tokens:

- TABS: it's the only required parameter. Enter it next to the id you have previously chosen for your tab (id_of_tab in the .r file).
- icoT: it's an optional integer. Set it equal to 1 to associate icons with your tab pages.
- alNI: it's optional boolean. By default, Carrara will put the name of your page above your icons. If you set this boolean to 1, the name of your pages will be then displayed beside your icons.
- Page: it's another optional integer. By default, Carrara won't open any tab page. If you set this integer to a value (between 1 and the number of your pages), Carrara will open the indexed page. This index is based on the order of the pages in the .r file (within the resource 'TABS' you previously created).

To learn more about custom tokens, see "Extra Tokens for Each Part" on page 36.

Creating icons

Let's deal now with the icons area. Create a node inside the TMFTabPart, and change the class name into TMFTabArea. Also change the Part ID to TabA. This is the default name for the shell to recognize this node as the host for icons. To finish, put on the 'top', 'bottom' and 'left' constraints.

Then, for the pages area, create a simple node within the TMFTabPart. Change the Part ID into Host. This is also the default name for the shell to recognize this part as the host for user interface tab pages. To finish, set all the constraints to on.

Creating a Component Chooser

Quite often in the interface you need the user to choose an item. And quite often, the possible item is one component plugged into the Shell. For example, the Properties Palette of Carrara for any object allows to apply a Constraint to that object by selecting in the list of all the constraints currently plugged into Carrara's Shell. Once chosen, the UI's component shows up, and the user can set the parameters of the component. You may also notice that most of the time, the component chooser is a collapsible part.

That kind of UI item is always done the same way in Carrara. The whole operation is done by what we call the component chooser.

A component chooser is a very powerful tool, with only a few limitations. For example, although most of the time choosing the component is done for a given family, you are not restricted to only one family. Or you may want the part to be collapsible or not.

There are two major types of component choosers :

- the single component chooser; to be used when you need just ONE component at the time.
- the multiple component chooser; to be used when you want a dynamic list of components.

In this example, let's focus on the most common : the single component chooser. To create a single component chooser you'll need to set up the .r and .rsr files as described below.

In the .rsr file:

In this file, you first have to create a single component chooser. To do so, create a simple node and change the class name into TSingleComponentChooser. Then, put on some parameters in the Custom Tokens area.

To learn more about custom tokens, see "Extra Tokens for Each Part" on page 36.

MCSketch

MCSketch provides a cross-platform tool for user interface data creation. It adds the ability to edit your own resource template.

Extra Tokens for Each Part

Many parts have features that do not show up in MCSketch as buttons in the Properties palette. You set the values for these extra features in the "Custom Tokens" box in the Part Properties palette of MCSketch. Warning : there is no control from MCSketch about the content of this text box. Therefore, you need to follow strictly the Token Manager syntax rules (which are the same as the syntax rules of the file format : pairs of token and value).

For an example of how to create tokens, see the SDK\Samples\LightSources\Light\Light.r and Light.rsr files.

List of all UI elements (parts)

When you want to create a UI element in MCSketch (for example, a button in a dialog), you can either drag and drop a preset one from the Part Palette on the left, or select the right menu item from the MFWindow/Plugin Group#xxx sub menus.

The Part Properties palette then shows several important pieces of information about the part you just created:

- The **Root Class Name** : this is the name of the C++ object behind the element.
- The **Class Name** : often, you want in fact to instantiate a more sophisticated version of this C++ object. For example, you can create a TMFEditTextPart (a text box), but in fact you want a TMF3DUnitEditTextPart (to have units in your text box). Simply type "TMF3DUnitEditTextPart" in the Class Name field.

- The **Custom Tokens** field : allow you to set up additional parameters to your part if they cannot be set through an editor (see “Extra Tokens for Each Part” on page 36).
- The Editor specific to the part : for example, a TMFEditTextPart editor allows you to change the default text, set the text type, justification, etc.

TMFEditTextPart - Edit Text part

Purpose : Free text entry or numerical values entry
Root Class : TMFEditTextPart
Default Class Name : TMFEditTextPart
Other possible Class Names : TMF3DUnitEditTextPart
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types : kInt16ValueType, kInt32ValueType, kReal32ValueType, kInt32MinMaxOnlyValueType, kReal32MinMaxOnlyValueType, kStringValueType
Editor in MCSketch : yes
Custom Tokens :

Token	Type	Default value	Purpose
NtyA	boolean	false	Notify on all changes (not just the end of the editing)
SpnB	boolean	true	Display a "spin box"
Ntfy	boolean	true	Notify if value is out of range

Comments : None

TMFIconButtonPart

Purpose : **zot**
Root Class : TMFIconButtonPart
Default Class Name : TMFIconButtonPart
Other possible Class Names : none
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch : yes
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFTextButtonPart

Purpose :
Root Class : TMFButtonPart
Default Class Name : TMFButtonPart
Other possible Class Names : TMFIconButtonPart,
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch : yes
Custom Tokens :

Token	Type	Default value	Purpose

Token	Type	Default value	Purpose

Comments :

TMFDialPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFHierarchicalListPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFImagePart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFCheckboxPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFColorPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFOverlayImageControlPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFRadioPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFToolbarPart

Purpose : Selection of tools
Root Class : TMFNodePart
Default Class Name : TMFToolbarPart
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder, IMFToolbarPart
Supported value types : kInt16ValueType, kInt32ValueType
Editor in MCSketch : no
Custom Tokens :

Token	Type	Default value	Purpose
HorL	boolean	true	Defines whether the toolbar is in a horizontal (true) or vertical (false) orientation
TBAR	int16	0	Reference to a TBAR resource whose tools will be loaded into the toolbar
Glob	boolean	true	Defines whether the toolbar is global (true) or local (false). Global toolbars enforce mutual-exclusivity of tool selection across all global toolbars, while local toolbars only enforce it within itself.
TlRc	IDType		Defines the part ID of the part that should receive tool selections (SelfMenuAction or SelfToolAction). If this token is not defined, tool selections will be passed to the first responder.
SpSz	int32	5	Defines the space between tools
RAct	boolean	false	If true, tools will respect the activeness of their enclosing window (i.e., tools will be disabled if the window is disabled). If false, tools will ignore the window activeness.

Comments :

TMFNodePart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :

Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFAutoCenteringPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFBalloonPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFWindoidPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :

Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFDialogoidPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFCollapsiblePart, TMFSelectableCollapsiblePart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, and IMFCollapsiblePart
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TSingleComponentChooser

Purpose : **Let the user choose a component from a pop-up menu. Many features included automagicaly, like selection/drag’n’drop, mini-part display, etc.**
Root Class : TMFLeafPart
Default Class Name : TSingleComponentChooser
Other possible Class Names : TSingleCompChooserWithWireFrame, TSingleCompChooserWithPreview
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, IMFCollapsiblePart, and ISingleComponentChooser
Supported value types : ’comp’ (AKA kComponentValueType
Editor in MCSketch : no

Custom Tokens :

Token	Type	Default value	Purpose
fmly	IDType		Family ID of the components to fit into the menu.
bnon	boolean	true	Is 'none' a valid option ?
shof	boolean	true	Show the family name.
MskI	uint32	0	Include Mask for the pop-up menu
MskE	uint32	0	Exclude Mask for the pop-up menu
Sort	boolean	false	Sort the pop-up menu by family name.
Subm	boolean	false	Sub-menu mode : true will create sub-menus of the items by sub-family. The title of the sub-menu is the sub-family string (in the COMP resource).
Mini	boolean	false	Use a mini part ? If true, the collapsible state displays the part that is specified in the COMP resource of the component, and linked to the PMap.
Name	string		NAme of the component chooser.
shpp	boolean	true	Show the pop-up.
SHos	IDType		ID of the selectable host.; that must be a parent of the chooser.

Comments :

TMultipleComponentChooser

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFFramePart

Purpose : Draws a frame in the bounds of the part
Root Class : TMFNodePart
Default Class Name : TMFFramePart
Other possible Class Names : none
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, and IMFFramePart
Supported value types : none
Editor in MCSketch : no
Custom Tokens : none
Comments :

TMFGradientPart

Purpose : Draws a horizontal or vertical gradient between arbitrary colors and transparencies
Root Class : TMFNodePart

Default Class Name :	TMFGradientPart
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	none
Editor in MCSketch :	no
Custom Tokens :	

Token	Type	Default value	Purpose
SrtF	Fill Attr.	black	Start Fill
EndF	Fill Attr.	white	End Fill
SrtT	real32	0	Start Transparency (0.0 to 1.0)
EndT	real32	0	End Transparency (0.0 to 1.0)
Vert	boolean	false	Defines whether the gradient runs left to right (false) or top to bottom (true)

Comments :

TMFNodeReferencePart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFRadioCluster

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFScrollPart

Purpose :	**zot**

Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFSlider, TMFCircularSliderPart, TMFLinearSliderPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFSolidColorPart

Purpose : **Fills the part bounds with a color**
Root Class : TMFNodePart
Default Class Name : TMFSolidColorPart
Other possible Class Names : none
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, and IMFSolidColorPart
Supported value types : none
Editor in MCSketch : no
Custom Tokens :

Token	Type	Default value	Purpose
Fill	Fill Attr.	white	Fill (color)
Trns	real32	0	Transparency (0.0 - 1.0)

Comments :

TMFSplit3Part

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :

Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFCollapsibleListPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFVectorPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFListPart, TMFRuledListPart, TMFStringListPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	

Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFOffscreenPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFPoppedPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	
Custom Tokens :	

Token	Type	Default value	Purpose

Comments :

TMFRectPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	

Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFStaticTextPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	

Custom Tokens :

Token	Type	Default value	Purpose

Comments :

TMFTextPopupPart

Purpose :	**zot**
Root Class :	
Default Class Name :	
Other possible Class Names :	
Supported interfaces :	IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :	
Editor in MCSketch :	

Custom Tokens :

Token	Type	Default value	Purpose

Comments :

Interfaces supported by each UI element (part)

The following table summarizes the list of implemented interfaces for each User Interface element that you can create in MCSketch. This information can also be found in the invidual decription of each part (see “List of all UI elements (parts)” on page 36). Very often, you need to talk to a part using several interfaces, using QueryInterface to get pointers on these interface. In the dialog example above, we talk to the dialog as a IMFPart and as a IMFDialogPart.

All parts implement :

IMCUnknown, IMFPart, IShComponent, and IMFResponder.

Most of the time, using IMFPart::SetValue, IMFPart::GetValue, and other IMFPart calls is good enough to set up your UI elements. However, some parts implement additionnal interfaces for special purposes.

Table 3: Additionnal interfaces supported by some UI elements

Class Name	Additionnal supported interfaces
TMFCheckboxPart TMFColorPart TMFOverlayImageControlPart TMFRadioPart	IMFImageControlPart
TMFToolbarPart	IMFImageControlPart, IMFToolbarPart
TMFWindoidPart	IMFWindow
TMFDialogoidPart	IMFWindow, IMFDialogPart
TMFFramePart	IMFFramePart
TMFCircularSliderPart TCircularConstraintSliderPart TMFLinearSliderPart TLinearContraintSliderPart	IMFSliderPart
TBitmapSliderPart	IMFSliderPart, IMFBitmapSliderPart
TMFSolidColorPart TColorPickerSwatch	IMFSolidColorPart
TMFTabPart	IMFTabPart
TComponentUIPart	IMFParameterComponentPart
TParameterPart	IMFParameterPart
TTextureMapPart	ITextureMapPart
TMFListPart TMFRuledListPart	IMFListPart
TMFTextPopupPart TMF3DUnitsPopupPart TMFFontListPopupPart TMFFontStylePopupPart TMFIconPopupPart	IMFTextPopupPart

Counted Pointers

Using counted pointers allows you to easily deal with reference counting by hiding the calls to AddRef() or Release() directly.

Note: After construction of an object, its reference count should be 1.

Before you begin, it is important to outline why reference counting is important:

1. It is much safer. You can much more easily avoid double-deletes, leaks and other memory problems using reference counting. Sometimes you need to keep references to things that normally would be deleted. For instance, in TMFAction and subclasses, it is convenient to keep references to items that are removed, reordered, deleted, etc. Without reference counting, it may be impossible to keep the pointer valid without leaking.
2. It is actually easier in some cases to keep reference counts. Keeping reference counts alleviates the need for ownership of pointers. To use a metaphor: last one out turns off the lights.
3. You have no choice. In COM-based applications, reference counting is required. For instance, in order to pass an ISceneDocument interface to an external component, you need to implement a reference count for that object. If you simply reference count TSceneDocument, and not its parent classes (including TMFResponder), then half of the object is reference counted and half is not. And once the reference count becomes zero, the object is destructed, even though the pointers that were not reference counted still think that the object is valid.

Defining Classes that May Need to be Counted

Here are some rules for defining classes that may need to be counted:

- Derive your class from TMCSMPCountedObject.
- Make the destructor protected; it can be private if there is no sub-class.
- Make the constructor protected; it can be private if there is no sub-class.
- Provide a static “Create::(T**)” method. (“Create::(T** P1, ...)” if other parameters are needed for constructor.)

TMCCountedObject/TMCSMPCountedObject

These objects implement simple reference counting. In Carrara, only **TMCSMPCountedObject** should be used because it uses an atomic counter for reference counting and is thus compatible with multi-threading.

Note: AddRef() and Release() need to be virtual MCCOMAPI's so that real COM objects can subclass from TMCSMPCountedObject and override the functions. However, this should not add any size overhead, since subclasses of TMCSMPCountedObject should already have virtual destructors.

Destructors

Destructors for all subclasses of TMCSMPCountedObject should be protected and virtual. You should never call delete on any such object — use Release() instead.

Be careful: C++ lets you make members more public without warning (so check your class to make sure what access level the destructor should be).

Also, be sure you do not rely on destructors to remove their objects from global lists.

This clearly cannot work, since if an object is on a global list (that is presumably reference counted), the destructor will never be called, since there will always be a reference to the object.

Using AddRef() and Release()

Basically, there are only two rules for proper reference counting:

1. Call AddRef() whenever you assign a reference counted pointer to a variable.
2. Call Release() whenever you are done.

Experienced COM programmers out there might realize that we are one rule shy of listing all of the COM rules for reference counting. The rule that we are missing is: Call AddRef() before returning. That's true, it's gone, but because of the way TMCCountedPtr works, we need to adjust for it. TMCCountedPtr AddRef()s on assignment, which includes function return values. So technically, the AddRef() occurs after the function return, so rule (1) applies.

However, if you don't use a TMCCountedPtr, you need to explicitly call AddRef() on function returns when they return a reference counted value.

Note: We're still OK on QueryInterface() though, since the assignment happens inside the function, so you won't need to call AddRef() after returning from QueryInterface().

TMCCountedPtr

The above rules imply a lot of calls to AddRef() and Release(), which can clutter the code to unreadability. TMCCountedPtr's make things much easier since they call AddRef and Release to implement the rules above.

TMCCountedPtr is a behavior-wrapper for a pointer to any class that implements AddRef() and Release() (including TMCCountedObject). TMCCountedPtr overrides many operators (including *, ->, &, =, and cast) in order to call AddRef() and Release() at appropriate times and check pointers when necessary. See TMCCountedPtr.h for more information.

The only data in TMCCountedPtr is the pointer itself and there are no virtueless, so the class has no size overhead.

It is a template class and should be used thus:

```
TMFResponder*                fMyResponder;  //old becomes
TMCCountedPtr<TMFResponder> fMyResponder;  //new
```

Since the destructor calls Release(), TMCCountedPtrs are best used as stack-based objects or class data members only. Do not use TMCCountedPtr* or use TMCPtrArray<TMCCountedPtr<T>*>, since you want their destructor to be called automatically.

Do not pass TMCCountedPtr's as arguments to functions. It is not necessary, since passing parameters does not keep a reference (so no AddRef() needs to be called).

Remember, only if you assign to a variable do you need to call AddRef().

TMCCountedPtrArray

TMCCountedPtrArray is to TMCPtrArray what TMCCountedPtr is to T*.

TMCCountedPtrArray<T> is a template class that is a subclass of TMCClassArray< TMCCountedPtr< T > >. It should be used thus:

```
TMCPtrArray<TMFResponder*>  fMyResponders;  //old becomes
TMCCountedPtrArray<TMFResponder>fMyResponders;  //new
```

These arrays are a cross between TMCPtrArray and TMCClassArray, since they have properties of both. For instance, TMCCountedPtrArray<T>::iterator::First() returns a TMCCountedPtr<T> (like a TMCPtrArray), but it is a class array, so destructors are automatically called (and thus Release()) on removal from the list.

Miscellaneous

Note: You also cannot create counted objects on the stack, since that involves an implicit destructor at the end of the scope. For instance, you cannot do this:

```
{
            TMFWindow myWindow(...);
}
```

Instead, use a TMCCountedPtr and call the static Create() fuction. The release will be called by the implicit destructor on the TMCCountedPtr.

```
{
        TMCCountedPtr<TMFWindow> myWindow;
        TMFWindow::Create(&myWindow);
}
```

Note: You need to be careful to handle cycles correctly. A cycle occurs when object A has a reference to B and B has a reference to A. If a cycle is created, the objects will never be destroyed even if there is no reference to the cycle outside of itself ! It is best to avoid cycles in counted reference. To avoid cycles you should defines which object is the owner of the other object: for instance, if A owns B, then A keeps a counted reference to B but B keep an uncounted reference to A.

Reference Count Optimizations

TMCCountedPtr's are indiscriminate. They reference count everything, even when you don't really need to. For instance,

```
{
        TMCCountedPtr<TMFResponder> r = GetFirstResponder();
        r->DoSomethingThatWillNotRelease();
        r->DoSomethingElse();
}
```

You do not really need a counted pointer here, since you are not keeping a reference beyond the scope of the function, and the function DoSomethingThatWillNotRelease() will not decrement the reference count on r, so it won't go away out from under you.

Implementing Create

Implement Create as follows:

```
void TMyClass::Create(TMyClass** a)
{
        TMCCountedCreateHelper<TMyClass> result(a);
        result = new TMyClass;
}
```

Implementing a Counted Pointer Access Method

The accessor to a Counted Pointer can be implemented as follows:

```
TObjectClass* TMyClass::GetNoAddRef()
{
        return fObject
}
```

Or

```
void TMyClass::Get(TObjectClass** a)
{
        TMCCountedGetHelper<TObjectClass> result(a);

        result = fObject;  // fObject is a counted pointer
}
```

Implementing QueryInterface()

QueryInterface should always return something for IUnknown. This is usually done by calling QueryInterface on the parent class. Here is an example of implementation of QueryInterface:

```
MCCOMErr TMyClass::QueryInterface(const MCIID& riid, void** ppvObj)
{
        if (riid == IID_I3DExInstancePrimitive)
        {
           TMCCountedGetHelper<I3DExInstancePrimitive> result(ppvObj);
           result = static_cast<I3DExInstancePrimitive>(this);
           return MC_S_OK;
        }
        return TParentClass::QueryInterface(riid, ppvObj);
}
```

Helper Classes for Counted Pointers

Carrara provides two helper classes for Counted Pointers: TMCCountedGetHelper and TMCCountedCreateHelper.

You use TMCCountedGetHelper to implement a “Get” operation, or a similar interface. This will Release the previous object if there was any, clear the handle, and help you affect a new value, and do the AddRef. It will check as many things are OK as possible.

You use TMCCountedCreateHelper in a “Create” method (Typically when you create your counted object with the new operator (which initializes the RefCount to 1)). It also does plenty of validity checks.

It is *imperative* to use these new classes if you want to be able to track AddRefs/Release calls. Without that, the debugging code can’t match AddRef and Release. It is *imperative* to *not* call AddRef and Release explicitly, as these calls can't be tracked either.

All COM objects should implement the proper QueryInterface for IID_IMCUnknown.

Here are a few examples of uses for these two new classes (please study them carefully.)

Example of TMCCountedGetHelper

```
//Use with void**
//-----------------------------------------------------------------------------
MCCOMErr TMFBaseWindow::QueryInterface(const MCIID& riid, void** ppvObj)
//-----------------------------------------------------------------------------
{
        if (MCIsEqualIID(riid, IID_IMFWindow))
        {
           TMCCountedGetHelper<IMFWindow> result(ppvObj);
           result = (IMFWindow*)this;
           return MC_S_OK;
        }
        return TMFNodePart::QueryInterface(riid, ppvObj);
}

//Use with type, no conditional affectation
void TMFBaseWindow::GetWindowFirstResponder(TMFResponder** outResponder)
const
{
        TMCCountedGetHelper<TMFResponder> result(outResponder);
        result = fFirstResponder;
}


//Use with type, conditional affectation *outDocument can be NULL at the
```

```
void TMFDocument::FindDocForFile(TMCFile* afile, TMFDocument**
outDocument)
{
        TMCCountedGetHelper<TMFDocument> result(outDocument);
        TMCCountedPtrArray<TMFDocument>::iterator
iter(TMFDocument::sDocumentList);
        for (TMFDocument* aDocument = iter.First(); iter.More(); aDocument =
iter.Next())
        {
           if (aDocument->FindDoc(afile))
           {
              result = aDocument;
           }
        }
}

//a more complex use of the GetHelper, you need to use both a TMCCountedPtr and
//a TMCCountedGetHelper because the type are different. If, like before, you were
//not using a TMCCountedPtr for 'component', the reference tracking would be
//wrong.
void TComponentUI::CloneComponent(IShParameterComponent **res)
{
        TMCCountedGetHelper<IShParameterComponent> result(res);
        TMCCountedPtr<TComponent> component;
        Clone(&component);
        result=static_cast<TComponentUI*>(*&component);
}
```

Example of TMCCountedCreateHelper

```
//example of TMCCountedCreateHelper
void TMFWindoidPart::Create(
        const TMCWindowInfo& inWindowInfo,
        const EWindoidTypeinWindoidType,
        TMFWindoidPart**outObject)
{
        TMCCountedCreateHelper<TMFWindoidPart> result(outObject);
        result = new TMFWindoidPart(inWindowInfo, inWindoidType); //Init count
to 1, no AddRef
}

//an other example of TMCCountedCreateHelper
void TBrushMask::Clone(TComponent **res)
{
        TMCCountedCreateHelper<TComponent> result(res);
        TComponentUI* newComp = new TBrushMask(); //Init count to 1, no AddRef
        CloneComponentUIData(newComp);
        result=newComp;
}
```

TMCCountedPtr vs Interface Pointer

Always use TMCCountedPtr in function cores but pay attention to cycle references when the pointers are contained in an object.

Here is an example of the correct way to use a TMCCountedPtr:

```
        {
            TMCCountedPtr<interfaceType> ptr;
            QueryInterface(IID_interfaceType,(void**)&ptr);
            .....
            ptr->function(...);
            ....
        }
```

The same code with an interface pointer:

```
    {
        interfaceType *ptr=NULL;   // don't forget to initialise to NULL
                                  // !! (not necessary with counted ptr)
        QueryInterface(IID_interfaceType,(void**)&ptr);
        .....
        ptr->function(...);
        ....
        ptr->Release();   // do not forget this or will leak
    }
```

Two functions returning InterfacePtr

There are two ways to get an InterfacePtr:

```
interfaceType *GetInterface();              // does no AddRef
GetInterfaceType(interfaceType **);    // does a AddRef
```

For example:

```
TMCCountedPtr<interfaceType> ptr;    // or interfaceType *ptr;
obj->GetInterface(&ptr);
```

Never return a CountedPtr

For example:

```
interfaceType *Getfdfkdsjfk()
{
     TMCCountedPtr<..> ptr;
         ....
    return ptr;
}
```

What happens in this case is that the ptr will be released at the end of the function. So you will pass a reference to something that has just been released. This may cause crashes, specially in optimize compiled code, since the debugger often modifies the scope of variable for debugging purposes. Therefore those crashes are very difficult to find. If you happen to have crashes in optimize compiled code but not in debug compiled code, check the returned arguments for countedPtrs and local variables.

Creating objects

Objects should have a reference count of one when you just called their constructor. This is necessary for aggregation purposes. So if you write a CoCreateInstance function you should be careful to do a Release() after the QueryInterface or your reference count will be 2 when you get out of CoCreateInstance.

Aggregation

If an object keeps a reference to an interface to an other object with which it is aggregated, this pointer should not be counted (as it would make it impossible to delete the object).

However you usually need to call QueryInterface to get this internal reference which will do a AddRef on the aggregated object. So be sure to call Release just after the QueryInterface. This is possible as the reference count is initialized to 1.

Not all of the points concerning the aggregration are discussed here. You should at least be aware that QueryInterface and Release cannot be done in an aggregation as in a simple object.

Utilities

This chapter describes details about the utilities included with Carrara.

Many global utilities are available to extensions. They are initialized automatically by the SDK library when the Extension is loaded. You access these utilities through global variables.

Examples:

- File and streaming
- Memory allocation, off screen buffers, pixel iterators
- Menus, mouse action (for tracking), drag/drop
- Component utilities
- Etc.

Various Shell Utilities (IShUtilities)

Use the **IShUtilities** interface with the **gShellUtilities** pointer to access to various services from the Shell. These services include :

- Progress bar management
- Time count (TickCount)
- Random number generation
- Low level alerts
- Image "pixel buckets"
- Open documents browsing
- MMX utilities (Windows only)
- Text metrics, cursor, fonts list, keyboard low level tests, etc.

Thread Utilities (IShTreadUtilities)

IShTreadUtilities and **gShellTreadUtilities** are used to launch and kill cooperative threads. Note that those threads are not preemptive threads. If you want to launch preemptive threads, you should use **IShSMPUtilities**.

Component Utilities (IShComponentUtilities)

Use **IShComponentUtilities** and **gComponentUtilities** to manage components from your extension. The main purpose is to create components. You can then use QueryInterface to initiate a more personalized discussion with the component.

SMP Utilities (IShSMPUtilities)

Use **IShSMPUtilities** and **gShellSMPUtilities** for multi-tasking management (launching a new thread, killing one, etc.). See “Multi-Threading” on page 71 for details about multi-tasking.

Action Utilities (IShActionManager)

IShActionManager and **gActionManager** are used to post Actions and Mouse Actions. You can also test the status of modifier keys of the keyboard (Control, Alt, etc.).

Menu Utilities (IShMenuUtilities)

Use **IShMenuUtilities** and **gMenuUtilities** to manipulate menus: you can dynamically build menus, enable and disable menu items, etc.

Part Utilities (IShPartUtilities)

IShPartUtilities and **gPartUtilities** are used to create Parts (usually from a resource built in MCSketch). The most useful call of this interface is CreatePartByResource.

Resource Utilities (IShResourceUtilities)

Use **IShResourceUtilities** and **gResourceUtilities** to load any kind of resources stored in your DTA file by using GetResource or GetIndResource.

Use GetIndString to load string resources (STR#) with automatic Windows-to-MacOS or MacOS-to-Windows ASCII translation.

Before any resource loading operation, make sure that the current resource file is properly set on your DTA file by encapsulating your resource calls between SetupComponentResources and RestoreComponentResource.

File Handling (IShFileUtilities)

The utilities used to access and manipulate file streams are **IShFileUtilities**, **IShFileFormatUtilities** and **IShFileStream**. Another interface that you should become familiar with is IMCFile.

The global variables to use are gFileUtilities and gFileFormatUtilities.

Creating Files

When creating files, you never inherit from IMCFile. When you need to create a file, create it through IShFileUtilities::CreateIMCFile by doing the following:

```
TMCCountedPtr<IMCFile> myFile;

gFileUtilities->CreateIMCFile(&myFile); // Note the '&'

//Then you can use myFile at your leisure (don't forget it's a COM object)
myFile->....
```

Remember that when you see a Shell procedure with a double deferencing in its parameters list, it means a returned COM parameter, very much like in QueryInterface.

Examples:

```
IShUtilities::GetFontList(IMCFontList** outFontList);
IShMenuUtilities::CreateMenu(TMFMenuInfo& inInfo, IFMMenu** outMenu);
```

Opening Files

Once you created a IMCFile object, you can use it to manipulate a file. Link the interface to the physical file using SetWithPathname or SetWithFullPathname.

Example:

```
TMCCountedPtr<IMCFile> aFile;
```

```
        gFileUtilities->CreateIMCFile (&aFile);

        TMCDynamicString path;
        gShell3DUtilities->GetLastImageOpenPath(path);
        aFile->SetWithPathName(path);

        gFileFormatUtilities->OpenFileDialog(DialogName, aFile,
inTypes,Format1,Format2);
        aFile->GetFileName(pathName);
```

Working with Streams

This example shows how to write out to a stream.

```
TMCfstream* stream = NULL;
uint32 nbInst = 0;
TMCCountedPtr<I3DShScene> scene;

    TMCString1023 fullPathName;
    file->GetFileFullPathName(fullPathName);
    stream = new TMCfstream(fullPathName.StrGet(),TMCiostream::out);
    ThrowIfNil(stream);



    //      FailNIL(stream);
    //      FailOSErr((int16)stream->InitFileStream(fullPathName, kSh-
StreamOut));

    //stream->SetMacOSInfo('ttxt','TEXT'); // Needed on the Mac, Will not do
anything on the PC

    WriteDXFBegin(stream);
```

Personality Utilities (IShPersonalityUtilities)

IShPersonalityUtilities and **gPersonalityUtilities** can be used to get the color settings of the UI. You will probably never need it.

Drag & Drop Utilities (IShDragAndDropUtilities)

IShDragAndDropUtilities and **gDragAndDropUtilities** and used for 2 things: creation of drop candidates and drop areas, and launching a drag-and-drop action.

Change Management (IChangeManager)

IChangeManager and **gChangeManager** are used to Change Management. Change Management is a powerful messaging feature in Carrara. It works by creating “channels” (see **IChangeChannel** interface), to which “listeners” (**IChangeListener**) can subscribe. Anyone can then “post” messages on this channel, all listeners will hear about them.

Clipboard Utilities (IMFClipboard)

IMFClipboard and **gClipboardUtilities** are used to implement copy/paste across extensions and across the Shell. A **IMFClipping** interface is used to encapsulate the copied data.

3D Utilities (I3DShUtilities)

I3DShUtilities and **gShell3DUtilities** give a lot of utility routines specific to Carrara (the previous utilities were more focused on the framework itself, not the 3D stuff). There are lots and lots of goodies in there, from documents, rooms and modules management, to animation utilities and shader utilities.

MCCommon References

This chapter introduces the MCCommon references included with Carrara. It provides an overview of the application framework. Some of the MCCommon classes are documented in the Reference.

The MCFrame application framework provides an open architecture that allows you to create a seamless integration of externally developed components. It includes a single user interface format (similar to the dta format on the Macintosh) for cross platform use. There is also a cross-platform tool (MCSketch) provided for user interface data creation.

The frameworks provides basic classes with default behavior for most application-independent functionality such as windows, controls, dialogs, files, etc.

The framework is composed 3 layers:

- MCCore
- MCFrame (MCF)
- MCImage

Resources are also stored in a platform-independent way. The resource files (.dta) are the same on both platforms and contain a dump of the Macintosh resource fork. The Macintosh Resource Manager is not used by MCCore; rather, MCCore reads resources itself. The tools “Data Resource Compiler” and “Res2Data Postlinker” translate data files to their Mac resource equivalents and vice versa.

Each class in the platform independent part generally begins with TMF. Each MCCore class begins with TMC. Each class in the platform dependent part generally begins with either TMAC_MC or TWIN_MC.

MCCore Interfaces

MCCore provides low-level I/O functionality such as files, streams, and memory. The associated files are the IMC* and MC* headers. MCCore provides a level of abstraction to hide platform-specific code.

The main interfaces of MCCore are:

- IMCGraphicContext: provides Draw, Fill, Blit, GetSystemGCPtr, etc.
- IMCFile: provides Get/Read/Write files
- IMCFont and IMCFontList: provide accessors to the fonts, individually or as a list.

MCCore also provides a number of low level classes such as dynamic arrays (TMCArray, TMCPtrArray, TMCClassArray...), smart pointers (TMCCountedPtr), color classes (TMCColorRGB, TMCColorRGBA8...), rectangles (TMCRect), networking (TMCSocket)....

MCF Interfaces

MCF provides user interface level functionality such as view system, menus, and widgets. The associated files are the IMFEx*, IMF*, and MF* headers. MCF provides a hierarchical view system at the user interface level of the framework.

There are two main interfaces:

- IMFExPart: provides SelfDraw, SelfMouseDown, SelfMouseMoved, etc.
- IMFExResponder: provides Receive, SelfMenuAction, SelfPrepareMenu, SelfKeyDown/Up, etc.

User Interface Flexibility

MCF interfaces provide complete user interface flexibility to extensions.

Extension user interface is laid out using MCSketch. Refer to “MCSketch” on page 36 to learn more.

You can add a part extension for any visible user interface part component that you want to extend. When the shell loads that user interface element from a file, it sees the part extension, and connects the internal user interface part to the extension part. An extension can over-ride any method in the part to put in specialized behavior. An external part can also call back to internal part for standard behavior from its method (simulating inheritance!)

MCImage Interfaces

MCImage provides data storage and shared code area.

MCImage provides interfaces for:

- Color models and color pickers
- Application independent Import/Export (2D/3D specific Import/Export APIs too)
- RasterLayer Utilities (basic layer/channel/tile/pixel interface)
- MC Photoshop Host (ImageIO based component)

Failure Handling

Using the Failure Handling

Correctly handling the Failure Handling of Carrara will ensure that your extension will perform in the best conditions and protect your user for unpleasant situations...

The failure handling in Carrara is based on the standard C++ catch-and-throw scheme. Ray Dream Studio developers will have to replace their TRY, SUCCESS and CATCH macros by the standard C++ 'try' and 'catch' keywords.

General rules

- If anything goes wrong in your code, you are allowed to fail (by calling **throw**), Carrara will catch the failure event and displays the infamous "Program Error" dialog box. After that, normal life starts again, so your user can still save his/her work.
- Every time you call a Shell routine, it can fail (especially when allocating memory). So your extension may want to be prepared for it to perform some clean-up during the Failure Handling phase, or even analyze the problem, treat it, and resume execution. You do this by implementing a **Failure Handler**, using the **catch** keyword.
- Since the scheme is not based on returning error codes, your code stays clean and readable. You implement Failure Handlers only where necessary.

A first simple example

Consider the following code. We are going to allocate 2 buffers, and we want to make sure that the first buffer is deleted if the allocation of the second one failed. We also want to delete them if anything goes wrong in the other procedures we call when using these buffers.

```
void FailureSample1()
{
       char* buffer1 = NULL;  // Always initialize to NULL
       char* buffer2 = NULL;  // Always initialize to NULL

       try              // This encapsulates the "protected" code
       {  //-- "Protected" code starts here --
          buffer1 = (char*) malloc(128000);  // malloc will fail if unsuccessful
          buffer2 = (char*) malloc(256000);  // Still here ? Go for the 2nd malloc!

       ..... do whatever you like here (it can trigger failures just fine)....

          free(buffer1); buffer1 = NULL;     // We are done. Note that the pointers
          free(buffer2); buffer2 = NULL;     // are set to NULL

       }    //-- End of "protected" code
       catch(...)
       {  //-- Failure Handler starts here
          // Something wrong happened either during the calls to malloc or during
          // the code in the middle or in sub procedures. Perform clean-up.
          if (buffer1) free(buffer1);
          if (buffer2) free(buffer2);
          throw;  // IMPORTANT ! This will pass the failure over
       }  //-- End of Failure Handler
}
```

Some comments on this code. As you can see, we do no test if *buffer1* is NULL after the malloc, because we know that if the allocation failed, we shall jump **directly** to the Failure Handler and not go any further. Likewise, if any subprocedure fails for one reason or another, it will jump directly to your Failure Handler, bypassing all the stack levels.

Overall, the code style is much more compact and readable than a classic error handling code, which would look like this:

```
long ClassicFailureSample()  // We need to return an error code :-(
{
        char*    buffer1 = NULL;
        char*    buffer2 = NULL;
        long error;

        buffer1 = (char*) malloc(128000);
        if (buffer1 == NULL) return -1;

        buffer2 = (char*) malloc(256000);
        if (buffer2 == NULL)
        {
           free(buffer1);
           return -1;
        }


        error = SubProc(...);
        if (error)
        {
           free(buffer1);
           free(buffer2);
           return error;
        }
        ...

        //-- Done at last
        free(buffer1);
        free(buffer2);

        return 0;
}
```

Typically you end up writing more error handling code than useful code, and your code becomes unreadable. Or you have to use the infamous `goto` to group your error handling at the end of the procedure. Moreover, you constantly have to pass over the error codes returned by sub procedures and test them. Most of the time, you end up not writing any error handling code at all...

Let go back to sample 1, and highlight a few things.

Initializing your local variables properly

It is very important that all local variables that are used in your Failure Handler are always in a “known state”, especially pointers. Remember that the processor can “jump” in your Failure Handler at any time until you are done in your protected code block.

- Every time you declare a pointer, set it to NULL immediately.
- Every time you free an object, set its pointer to NULL immediately after.

This way your Failure Handler can always safely test your local variables pointers and know if it has to do anything of not.

Local variables that are not used in the Failure Handler do not require anything special.

The Failure Handling Chain

Every time you create a Failure Handler using `try`, it adds your Failure Handler at the top a global chain. When the `try` block is completed, it removes the Failure Handler from the chain.

This way, if each Failure Handler calls `throw` properly, each clean-up will be performed one after the other until the top level, where the Shell posts the cursed Program Error dialog.

Triggering a Failure yourself by using throw, ThrowIfNil(), ThrowIfNoMem(), and ThrowIfError()

It may be useful in some situations for your code to launch a Failure to abort in despair. To do this, you use `throw`, `ThrowIfNil()`, `ThrowIfNoMem()`, or `ThrowIfError`() :

Throwing an error will force the processor to jump to the first Failure Handler in the Failure Handling Chain. Any code located after `throw` will **never** be executed.

In addition, you can use ThrowIfNil(), ThrowIfNoMem(), and ThrowIfError() and other conditional throw routines.

Use ThrowIfNil to throw if the pointer passed as a parameter is NULL. A NIL pointer error message will appear.

Use ThrowIfNoMem to throw a NULL pointer as the result of a memory allocation failure. A "Not Enough Memory" error message will appear.

Use ThrowIfError() with an error code parameter, to throw if this error code is not zero. An appriopriate message (depending on the error code value) will appear.

Error codes

Table 4: Standard error codes of Carrara

Value	Meaning	Message
0	No error or silent error	*None*
-1	Generic Program Error	“Program Error”
-2	Memory Manager error. Heap has been corrupted	“Memory suballocation error(-2)”
-3	Bus error	"Program Error(-3)"
-4	Bus error or OS exception	"Program Error(-4)"
-5	A NULL pointer has been found	"Null pointer error(-5)"
-6		"Program Error"
-7	Problem when reading file	"Wrong file format"
-8	Going off a list	"List out of bounds"
-9	Memory Manager error. Heap has been corrupted	"Memory suballocation error(-9)"
-34	Disk is full	“Disk full”
-36	File Open error	"File already in use or left open"
-39	End of file error	"End of file"
-42	Too many files open	"Too many files open"
-43	File not found	"File not found"
-45	File is locked	"File locked"
-46	Volume is locked	"Disk locked"
-49	File is already open with write permission	"File already in use or left open"
-50	Parameter error	
-53	Volume is offline	"Disk not available"
-102	No object of that type in scrapbook	"the specified format is not in the edition"
-108	Out of memory	"Not enough memory"
-128	User has cancelled	*None*
-490	User Break	
-30001	Unsupported file format	"Problem opening this file. Unsupported file format"
-30002	Generic file i/o problem	"Problem reading this file"
-30003	Memory startup problem	"Not enough Memory to launch"
-30004	Windows Graphics space error	"Not enough Graphic memory space (GDI) to launch"

Table 4: Standard error codes of Carrara

Value	Meaning	Message
-30005	Windows User space error	"Not enough User memory space (windows, menus...) to launch"
-30006	File not found when reverting	"Disk copy deleted"
-30008	Error in Save Again	"Document currently open"
-30009	Wrong File Type	"Not the right kind of document"
-30010	Unimplemented feature	"Feature not yet implemented"
-30011	The C++ class could not be found	"\"^3\": Missing Sofware component";
-30012	The Class signature could not be found	"Missing Sofware component"
-30013	Drive mapping error	"Could not map network to local drive. All drives A: through Z: are in use"
-30014	External texture error	"External Texture Map or Movie not found"
-30015	Corrupted data error	"File had corrupted data"
-30016	DirectDraw init error	“Direct Draw driver needs to be installed"
-30017	MacOS system memory error	"Not enough memory left for the system (Finder)"
-30018	Help memory error	"Not enough memory left to open the Help file"

Message parameter coding

You can use the *message* filed of the TMCException class to have more control on the alert that will be posted.

Let *hiMessage* be the high 16 bits and *loMessage* the lower 16 bits of *message*.

Table 5: Message values

hiMessage	Meaning
0	*loMessage* is an Action Number (like in CMNU or TBAR resources). The Shell will try to find the corresponding name and display it in the alert. Example : Program Error in “Save As...”
0xFFFF	*loMessage* is the ID of a ALRT/DITL resource. If a DITL item contains a ^0 substituion tag, it will be replaced by the name of the error as found in the Errors table.
0xFFFE	*loMessage* is an error code used by Carrara to parse his errors string table.
other values	Carrara will post an alert containing the string located at the *loMessage* index in STR# (*hiMessage).*

Failing Silently

Sometimes, you will want to fail silently, i.e. abort without posting the Program Error dialog. To do this, pass 0 as the error code and message:

```
            throw TMCException(0, 0);
```

All failure handlers will be processed, but no dialog will appear.

Recovering from a Failure

In some situation, your code can decide that it treated the Failure to its full extent and that normal execution of the code should resume. For example, you may have tried to allocate a very large buffer in RAM and it failed, but your code has an alternate solution (using a smaller buffer for example).

To do this, simply do **not** call `throw` at the end of your Failure Handler. The processor will exit the `catch` block normally and keep going. It is perfectly safe and legal to do so.

Some (lazy) programmers use this in combination with `throw TMCException(.,.)` to exit quickly from a deep part of their code and go back at the top level, using the “message” parameter of the `TMCException` to pass over some information of what happened.

Multi-Threading

This chapter explains the use of Multi-Threading with Carrara and the various issues that arise when an extension is called from threads (for instance during the rendering)

Two types of threads.

Carrara has two different types of threads:

- **Cooperative threads** do not run concurrently. The cooperative threads have to call IShThreadUtilities::YieldProcesses on a regular basis so that the other threads can be executed.
- **Preemptive threads** run concurrently. If your system has more than one processor, each thread can run on a different processor.

You should use **cooperatives threads** when you are calling functions that are not reentrant. Cooperative threads tend to be easier to manage because there is no synchronization issues.

You should use **preemptive threads** when you are want to take advantage of the multiprocessor or when you want to make sure your threads does not block the application. You should note that the rendering is using preemptive threads so you have to make sure that the calls to your extensions that are called during the rendering are reentrant.

Extensions and multi-threading

Using **preemptive threads** means that your extension might be simultaneously running on many (two or more) processors. So if your extension is using global variables, static members, or those types of features, each processor will share and modify that information at the same time. That means they will influence each other. In other words, each processor will introduce bugs on the other processor's computations.

Do not use static members or global variables if you are using SMP.

To make your life easier certain type of extensions are duplicated (cloned) by the Shell so that you do not have to worry about the fields of your extensions being accessed by two processors. Here is the list of the extensions that use a different copy for each thread:

- Ambient Light
- Atmosphere
- Backdrop
- Background
- Shaders

All other type of extensions are shared between the various threads. If a method of your component is called during the rendering, you must make sure that it is reentrant because on a multiprocessor system, it will probably be called by both processors. Note that a number of calls are protected by critical sections so that only one processor can call it at the same time (for instance GetFacetMesh). However if you are not sure, you should make your calls reentrant.

Now, if you are familiar with Multi-threading issues and want to write an extension that uses the full power of the platform in a specific way, Carrara's SDK provides interfaces to do so. If you are interested, look at the files /Include/MCCommon/IShSMP.h

DataBase Overview

Carrara stores all the data of currently open documents in a global database. Third party extensions can access this database to access or modify these data. To each open document corresponds one scene.

Scene

A *Scene* is mostly comprised of 2 structures:

- An *Object List*, that contains all the *Objects* used in the scene.
- A *Scene Tree* that contains the hierarchical structure of the scene. This defines the positions and relationships between *Tree Elements*. These tree elements can be object instances, light sources, cameras and groups.

Objects

Objects (also called Master Objects) contain the geometric information defining its surface regardless of its position. They also contain the texture coordinates of the object. The Master Objects used in a Scene are located in the Objects tab of the Browser window in Carrara.

There are two types of Master Object (I3DShObject): *Primitive*s (I3DShPrimitive) and Master Groups (I3DShMasterGroup).

Primitives are the basic objects that are located in the toolbar as icons. There are different classes of primitives. Carrara defines a certain number of default internal primitives (the Polygon Array, the Patch Array, the Polygon List...). However most of the primitives are external primitives (coded in regular extensions) like the Cube, the Sphere, the Isocaedra, etc. The Sdk allows you to create your own type of primitive.

Scene Tree and Tree Elements

The scene tree is a hierarchical structure containing *Tree Elements* (they implement the interface I3DShTreeElement).

There are four types of tree elements:

- **Object Instances**: they implement I3DShInstance, I3DShShadableTree
- **Light Sources:** they implement I3DShLightsource, I3DShInstance
- **Cameras:** they implement I3DShCamera, I3DShInstance
- **Groups:** they implement I3DShGroup, I3DShShadableTree

Those elements are detailed below. All these elements have 2 things in common:

- a position in 3D space (orientation, location, scaling, etc...)
- a name

Object Instances

An object instance is a tree element that has a reference to a Master Object in the object list. Therefore there can be multiple instances in a scene of a same object. This object can be either a Primitive (I3DShPrimitive) or a Master Group (I3DShMasterGroup).

The instance also defines the shading of the object and implements I3DShShadableTree and points to a one of the Master Shader of the scene.

Lights

A *Light Source* is a tree element. There can be as many lights as needed in a scene. Light Sources are implemented as

Extensions.

Cameras

A *Camera* is another type of tree element. Like lights, there can be many cameras in a scene. One of them will be the *Rendering Camera*, i.e. the camera the final image will be rendered from. Cameras are implemented as Extensions.

Groups

A *Group* is a tree element used to gather other tree elements together. It can be open or closed.

Master Groups vs Primitives

In order to make clearer these two notions, let's consider a basic example. We are supposed to model a scene with two columns (see fig. 1).

First using regular instances, we can make the parallel with a class in C++. Basically, if we change the color or the shape of the Column Master Object, these two instances will change as well.

However, we are not able to "factorize" the position or orientation. If we want to move the base of the column, it will not move the base of the other column.

As opposed to the previous situation, by using Master Groups, moving the base of one column results in moving the other one. Indeed, in that case, the transformations are automatically updated.

Coordinate Systems

Global Coordinate System

When you look at the Assemble room in Carrara, the axis of the Global Coordinate System are organized like this:

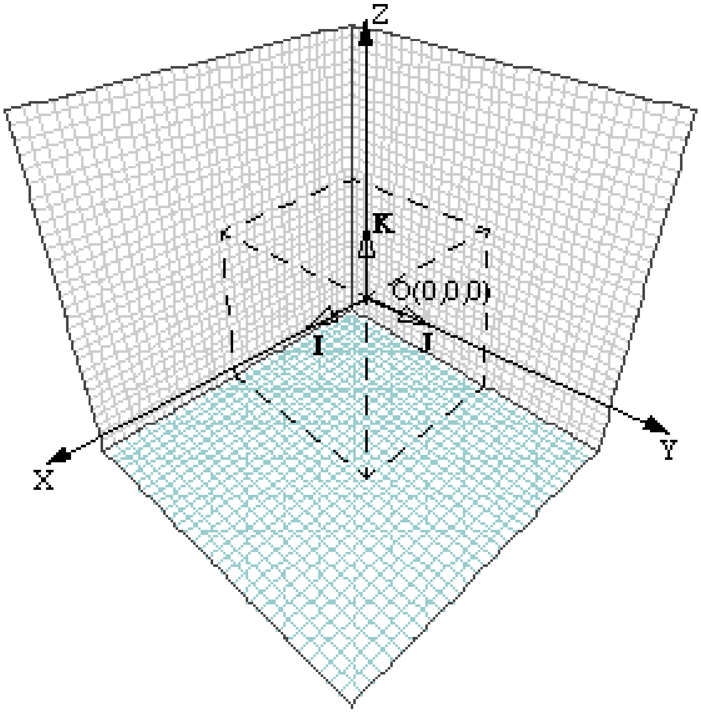


Figure 1. The Global Coordinate System

The projection of the (0, 0, 0) origin falls in the middle of each plane of the Working Box (the origin is *not* the far corner of the box). The **I**, **J**, **K** vectors are the unit vectors of the X, Y and Z axis.

Working Box Coordinate System

The Working Box Coordinate System is defined by the position of the Working Box. As the Working Box can be moved and rotated in Carrara, this coordinate system can be useful in some complex scenes. However, this system is never used when dealing with Extensions, so we will just mention it here.

Local Coordinate System (or Object Coordinate System)

The Local Coordinate System (also sometimes called the Object Coordinate System) define the system of an object or a group. Its axis, origin and scale depends on the positioning of the object in space.

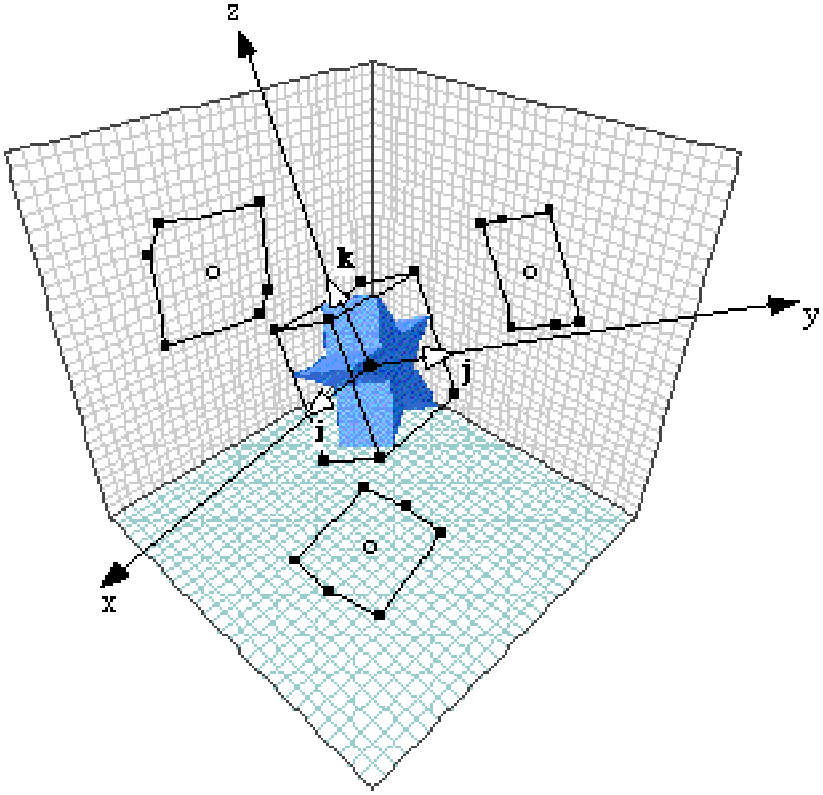


Figure 2. The Local Coordinate System of an object in a scene

The **i**, **j** and **k** vectors are the unit vectors of the x, y and z axis.

Screen Coordinate System

When a rendering occurs, the 3D data is projected onto the screen through the rendering camera. The axis of the 3D coordinate system attached to the screen is like this:

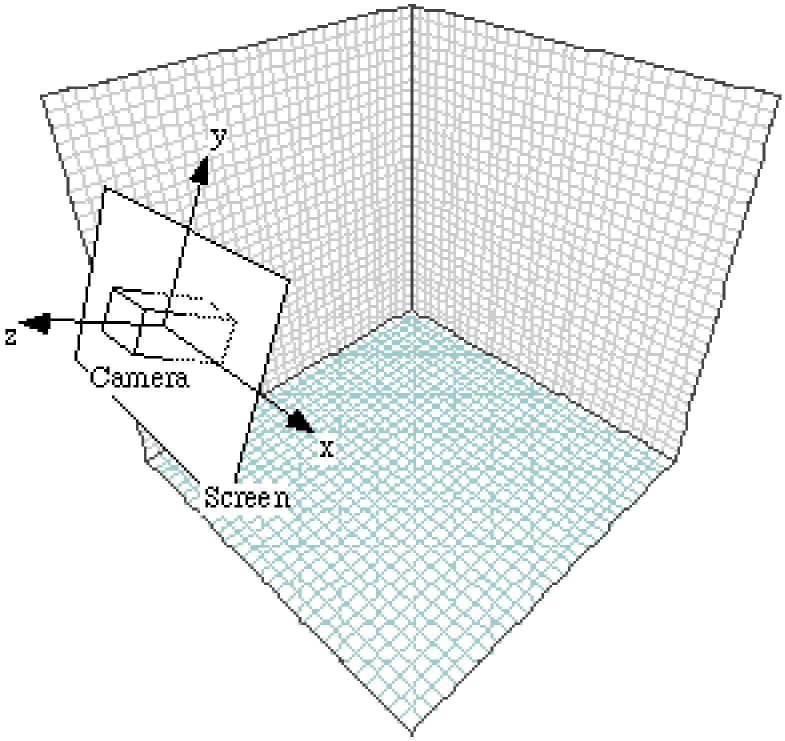


Figure 3. The Screen Coordinate System

As one can see, the objects seen by the camera have a negative z coordinate. The screen and the Screen Coordinate System are centered on the camera's center.

The Screen Coordinate System versus the Camera's Coordinate System

There is a slight difficulty here. The camera's aim is along the y axis of its transformation. As a result, here is the Local Coordinate System of the camera in the previous figure:

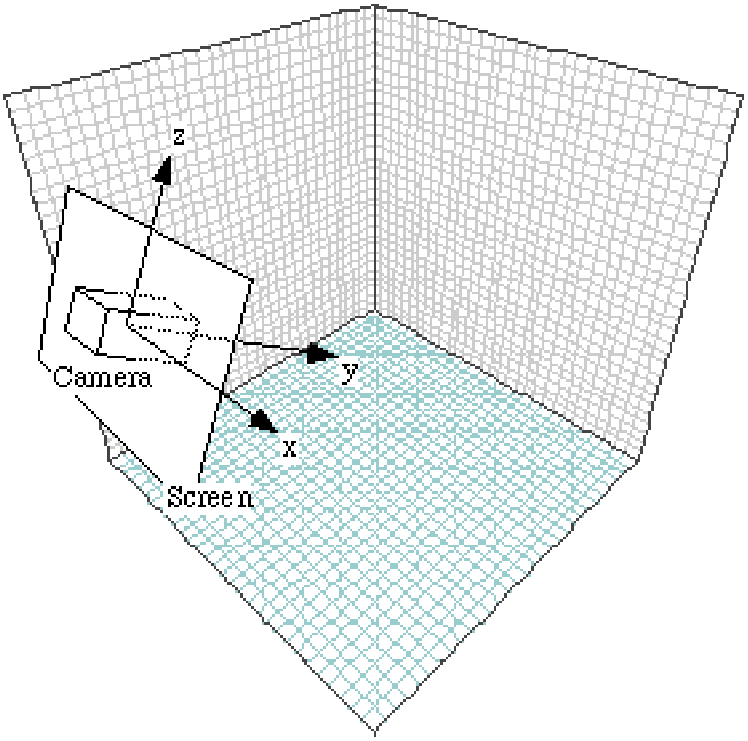


Figure 4. The camera's Local Coordinate System

You do not need to worry too much about this difference, because transformations calculated by the 3D Shell takes this into account. It is important only if you intend to position a camera in a scene.

The screen pixels space

The Screen Pixels Space is the actual Pixels coordinate system used to render the final image. Its unit system is in Points, as opposed to all other coordinate systems which are in 3D units. As a result:

1 3D unit = 288 points

because 1 inch = 72 points. More on the units business is covered later.

It is oriented with its vertical axis going down, and its (0, 0) origin in the top left corner of the image:

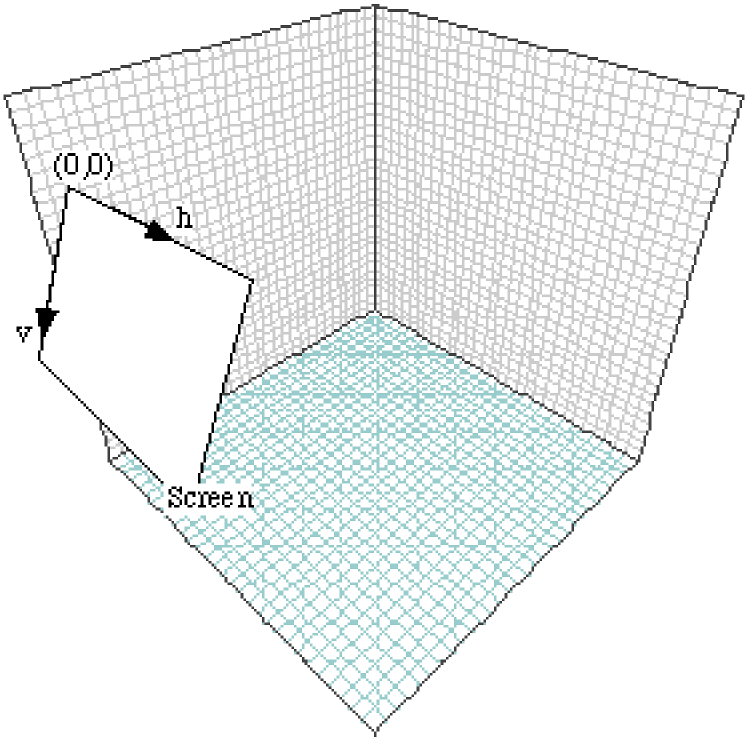


Figure 5. The Screen Pixels Space

Geometry

Geometric data type: 32-bit floating point (float)

All geometric data is in 32-bit floating point format:

```
typedef float Real;
```

Units System

The units system used in Carrara is defined as follows:

```
1 3D unit = 1 inch
```

All geometric data uses the 3D units. The various units shown by Carrara in the Properties tray are just handled at the user interface level.

The advantage of using a fixed system like this is that there is no problem of data conversion into different units.

Tree Elements Transformation

Each tree element (object instance, light source, group or camera) is defined relatively to its parent in the tree. Depending of the API procedure you call, you will get the transformation parameters that define the attitude of the object in space in one format or another. Whatever callback you use, you will get the following data:

- A 3x3 rotation matrix **R** (or the 3 vectors that define it)
- A translation vector **T**

- A uniform scaling factor **s**

To transform a point from the Local Coordinate System of this tree element into the Coordinate System of its parent, the following formula is used:

$$M = s[R]m + T$$

with m the point in local coordinates, and M the same point in the parent's coordinates.

To make it easier to read, here is the same equation in expanded format for each x, y and z coordinate:

$$M_x = s(R_{ix} \times m_x + R_{jx} \times m_y + R_{kx} \times m_z) + T_x$$

$$M_y = s(R_{iy} \times m_x + R_{jy} \times m_y + R_{ky} \times m_z) + T_y$$

$$\mathrm{M_z = s(R_{iz} \times m_x + R_{jz} \times m_y + R_{kz} \times m_z) + T_z}$$

With :

- m(m_x, m_y, m_z)point in local coordinates
- M(M_x, M_y, M_z)point in parent's coordinates
- T(T_x, T_y, T_z)position of the tree element origin in parent's coordinates
- s uniform scaling factor
- R rotation matrix (see below)

More on the rotation matrix:

If you consider the 3 **i**(i_x, i_y, i_z), **j**(j_x, j_y, j_z) and **k**(k_x, k_y, k_z) vectors of the Local Coordinates System, you can write easily the **R** matrix by putting each vector in each column like this:

$$[R] = \begin{bmatrix} i_x & j_x & k_x \\ i_y & j_y & k_y \\ i_z & j_z & k_z \end{bmatrix}$$

For additional information on transformation matrices and Tree Transformations, see the Data Structure chapters in the Reference.

Geometry basics

When you deal with objects, there are several concepts to define that are related to the object's shape.

Surface Point

Carrara deals with surfaces, not volumes. An object is made of surfaces that can be arbitrary complex. A point P on a surface is made of its x, y and z coordinates in the Object Coordinate System.

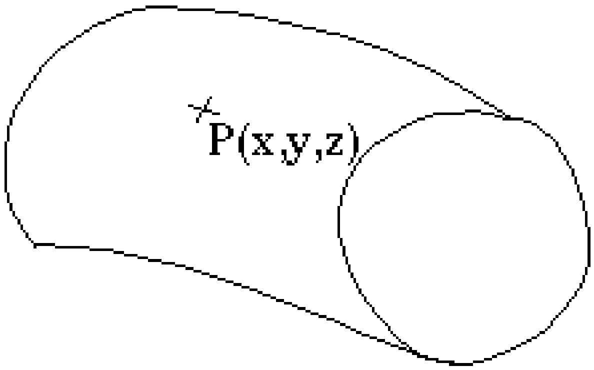


Figure 6. A point on a surface

Surface Normal

The Surface Normal at a point is the vector perpendicular to a plan that would be tangent to the surface at that point. The Normal is very important in 3D computer graphics, because it is heavily used for shading (especially in Phong and Ray Tracing shadings). Normals are usually normalized (their length is equal to 1), and always point outward.

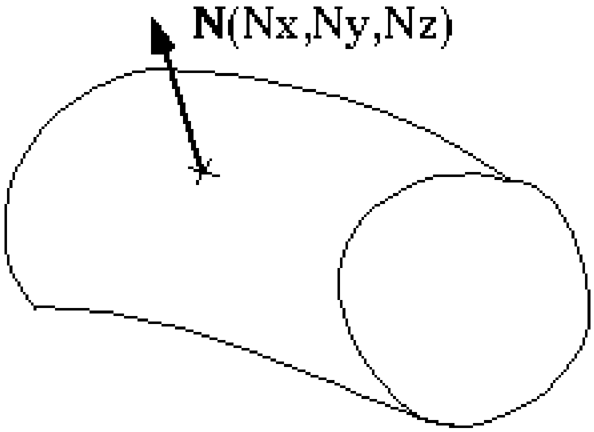


Figure 7. A Normal to the surface

Note that :

$$N_x^2 + N_y^2 + N_z^2 = 1$$

UV Space (Texture Coordinate System)

Wrapping a texture (such as a texture map) on an object surface involves finding the correspondence (the "mapping") between a 2D space (the image) and a 3D space (the object surface). The typical rendering question is: "I have a point P(x, y, z) on my surface, where should I look up in the texture map ?".

When one deals with simple shapes (sphere, cube, cylinders, etc.), it is easy to find a mapping. When one deals with complex objects (list of facets or patches), the matter becomes much more difficult. To solve this, most packages use a technique called "projection mapping": an intermediate imaginary surface surrounding the object is used, and the texture is placed on it. Then the surface color at a point is calculated by using the part of the texture that the point is facing.

The problem with this is that you can get awkward results when the object is very unlike the intermediate surface. Unwanted deformation is a typical problem: you want the texture to shrink or enlarge only where the object surface does.

Parametric mapping - a better solution

The Carrara Spline modeler is in fact capable of generating what we call "UV spaces", and the architecture of Carrara can keep this information down to the facets or bicubic patches level, in order to allow direct mapping (also called "Parametric Mapping").

The idea is this: because objects are built by combining 2D curves, it is possible to generate a 2D space that will behave topologically like the object surface.

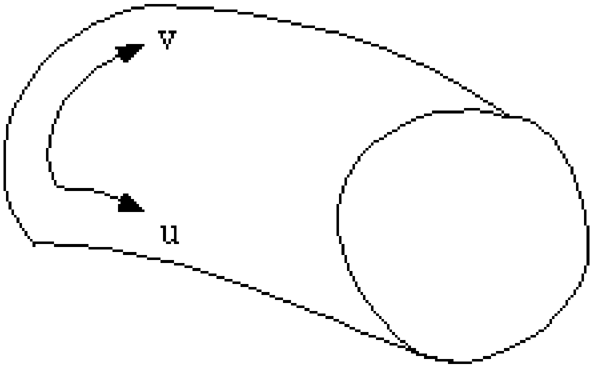


Figure 8. A typical u,v space on the side surface of an object

Every time you deal with a 3D point on the surface, you will be able to calculate the u,v coordinates for this point (provided that the object supports UV spacing). Say you need to know if the point is covered by a layer. The location of the layer is known by the u,v coordinates of its corners. The test becomes a simple 2D test: "are the u,v coordinates of the point inside a 2D rectangle ?".

Most of the time you get the u,v values from the 3D Shell. The only case where you have to generate them is when you develop a geometric primitive extension.

If the object has some surface discontinuities, then several u,v spaces are generated for the object. For example if your extrude a closed 2D curve, you will get 3 u,v spaces: one for the side surface, one for the back face and one for the front face. This is because it is not possible to have a u,v continuity across these different parts of the object (think of the problem of a napkin on a table). Each UV space has a number (0, 1, 2...), called a **UV Space ID**.

Shading

The shading can be applied on an object by giving general properties (color, reflection...) to the object or by mapping a texture on it. There is 4 ways of mapping a texture on an object.

- the parametric mapping,
- the box mapping,
- the cylindrical mapping,
- the spherical mapping.

The parametric mapping is the mapping using the UV Space information as seen in the last section. This is the better way of mapping, but because all objects don’t have necessarily a UV Space, four other ways of mapping are provided.

The box mapping, the cylindrical mapping and the sphere mapping are obtained by projecting a texture on a cube, cylinder or a cube on the object.

Facets

There are two kinds of low level geometric data used for rendering, exporting, etc.: **facets** and **bicubic patches**.

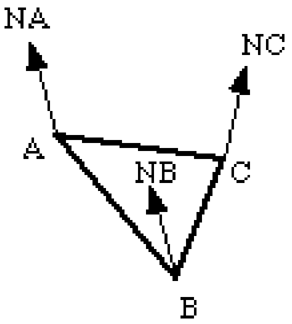


Figure 9. A 3D facet

Facets are triangles. Each vertex contains the (x, y, z) coordinates of the point, the Normal at this point, and the u,v coordinates at this point. The facet also stores the u,v Space ID to which it belongs.

```
typedef struct VERTEX3D {
  VECTOR3D    fVertex;    // x, y, z vertex coordinates
  VECTOR3D    fNormal;    // Nx, Ny, Nz normal values at that vertex
  NUM3D       fu,fv;      // Texture u,v values at that vertex
} VERTEX3D;

typedef struct FACET3D{
  VERTEX3D  fVertices[3];    // The facet three vertices
  short     fUVSpace;        // UV Space ID this facet belongs to
  short     fReserved;       // Reserved - 0
  } FACET3D;
```

Interpolating in a facet

Interpolating a point in a facet is a common exercise in 3D. If you call the vertices A, B and C, their Normals NA, NB, NC, and their u,v values uA, vA, uB, vB, and uC, vC, then:

from:

$$P = \alpha A + \beta B + \gamma C$$

you can interpolate:

$$N = \alpha NA + \beta NB + \gamma NC$$

Don’t forget to renormalize the normal:

$$\mathrm{N}_{\mathrm{nnit}} = \frac{\mathrm{N}}{\sqrt{\mathrm{N}^2}}$$

Likewise:

$$u = \alpha uA + \beta uB + \gamma uC$$
$$v = \alpha vA + \beta vB + \gamma vC$$

Of course, this is an approximation of the real values, but facets are already approximations of the real surface. As long as facets are small enough, this works just fine.

Bicubic Patches

Bicubic patches are more interesting geometric entities. They can describe more complex surfaces than facets, and are more resolution independent. For example Extrusions created by Carrara's Spline modeler generate bicubic patches. Turn on the Wireframe display; what you see are the boundaries of the patches.

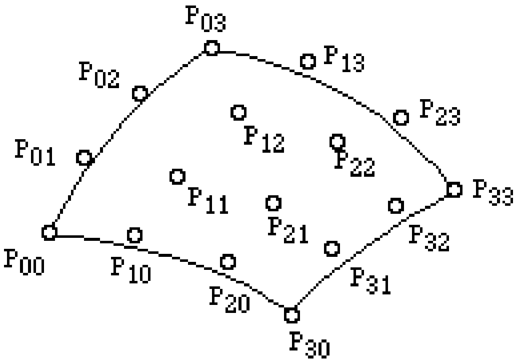


Figure 10. A bicubic patch

A patch is made of 16 3D points. You can think of a bicubic patch as a cubic Bézier curve that has its 4 vertices moving along 4 other Bézier curves.

```
typedef struct PATCH3D{
  VECTOR3D  fVertices[4][4];    // The patch 16 vertices
  NUM3D     fu[2];              // u values at the patch boundaries
  NUM3D     fv[2];              // v values at the patch boundaries
  short     fUVSpace;           // UV Space ID this patch belongs to
  short     fReserved;          // Reserved - 0
  } PATCH3D;
```

One can think of a bicubic Bézier patch as a Bézier curve moving on 4 other perpendicular Bézier curves. Let's apply this concept to calculate a point on the surface.

The patch can be defined as a parametric surface of two normalized parameters, tu and tv.

$$\text{Bicubic Patch} = S(t_u, t_v)$$

$$0.0 \leq t_u \leq 1.0$$

$$0.0 \leq t_v \leq 1.0$$

First consider the 4 Bézier curves defined by C0=(P00, P10, P20, P30), C1=(P01, P11, P21, P31), C2=(P02, P12, P22, P32), and C3=(P03, P13, P23, P33), and calculate on each a point at t_u:

$$P_n(t_u) = P_{0n}(1-t_u)^3 + 3\,P_{1n}(1-t_u)^2 t_u + 3\,P_{2n}(1-t_u)\,t_u^2 + P_{3n}t_u^3$$

Then calculate the point at tv on the Bézier curve ($P0(t_u)$, $P1(t_u)$, $P2(t_u)$, $P3(t_u)$) by re-using the same formulae as above. This is the result.

Patch normals

A nice thing about patches is that you do not have to worry about storing normals: they are automatically defined by the patch geometry. So this means that you do not have to calculate them for the 3D Shell. For the Mathematics savvy reader, remember, normals are defined as:

$$N(t_u, t_v) = \frac{\partial S}{\partial t_u} \times \frac{\partial S}{\partial t_v}$$

Patch u,v Space

The u and v values are constant along the patch boundaries.

P00, P01, P02, P03: u[0]   P00, P10, P20, P30: v[0]

P30, P31, P32, P33: u[1]   P03, P13, P23, P33: v[1]

(u, v) values at any point on the patch is calculated from these boundaries values by doing a Bézier interpolation.

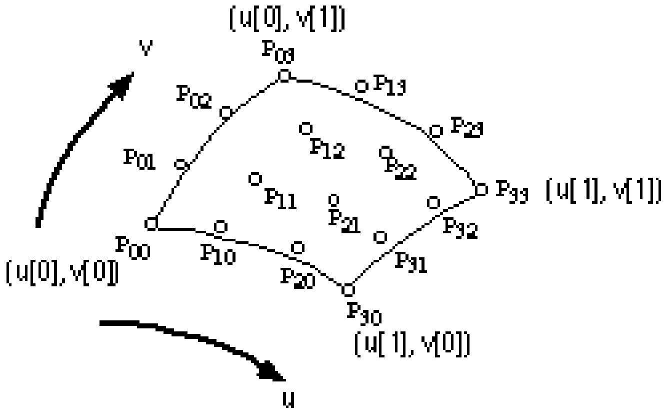


Figure 11. A patch (u,v) Space

To learn more about Bézier bicubic Patches

Read the excellent book « Introduction to Computer Graphics » by Folley - Van Dame - , published by Addison-Weisley.

Also take a look at source of the examples of the Carrara SDK, like the Teapot primitive or the patch deformer.

Color

In Carrara, two ways are used to stock a color : first a COLOR3D (See the Data Structures chapter in Reference Manual), then a long. <zot> TColor3D??

In a long (32 bits), we use this order ARGB, with the higher significant byte as A and the lower significante byte B.

A	R	G	B	
32	24	16	8	0

Color in long (32-bits) format.