Carrara SDK

Cookbook

How to write extensions™

September 2007

The software described in this manual is furnished under a licensing agreement contained in the license.txt file in the SDK folder. It may only be used or copied in accordance with the terms of this license. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical or otherwise, without the prior express written permission of DAZ 3D, Inc. The information in this user guide is provided for informational use only, is subject to change without notice, and should not be construed as a commitment by DAZ 3D, Inc. DAZ 3D, Inc assumes no responsibility or liability for any errors or inaccuracies that may appear in this user guide.

Licensee acknowledges that the Carrara SDK may contain bugs, errors and other problems that could cause system failures. Consequently, the Carrara SDK 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 Carrara SDK or any product associated with the Carrara SDK 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

Introduction

Purpose of Cookbook

This cookbook will step through the creation of an extension for each family. It details the extension, and aids in the creation of your own extensions.

How to use this Cookbook

The first section, *Creating Your Own Component*, describes the part of an extension that is common to all and the general steps to take to get started. The Database Overview chapter at the end of the book, gives you the mathematical background and the definition of the different 3D terms used throughout the documentation. If you find anywhere an unknown 3D concept or term, chances are that it is explained in the Database Overview chapter. The remaining chapters give an overview of each extension family, and describe the included sample code.

Family Chapters

Each chapter which describes an extension family begins with a section describing the Interface ID and header files used by the family. An overview of the methods used by the family follows. Finally, each example extension included for each family is detailed.

Where to Find Specific User Interface Examples

The following examples demonstrate setting up functionality without using a user interface:

- Exporter\DXF
- Importer\Imp
- PostRenderer\Sand
- Modeler\Step2 and \Step3
- FinalRenderers\WireRenderer

The following set up an auto PMap (simplistic UI):

- Modifiers\Barycenter
- LightSourceGels\Gel
- FinalRenderers\MixRenderer
- SceneCommands\StairCommand
- Modelers\Step1

For examples which use a PMap and an rsr file (created using MCSketch), to define the following user interface elements, refer to the these samples:

Table 1:

UI Element	Samples
Checkbox	PostRenderer sample: ColorBalance
Color Chooser	Backgrounds sample: Background, LightSources sample: Light, VolumetricEffects sample : Simple Volumetric Effect
Icon	Primitive samples: Flat, Sphere, Star, and Teapot

Table 1:

UI Element	Samples
Radio Button group	Modifiers sample: NonUniformScale, Constraint sample: ScrewConstraint, Lighting-Models sample : Simple Lighting Model
Scrollable Edit Text	Backgrounds\Background, Shader\Checker and Rainbow, LightSource\Light
Sliders	Camera\Camera, PostRenderer\ColorBalance, Modifier\Explode and NonUniformScale, Constraint\ScrewConstraint, Tweener\Tweener, TerrainFilters\Simple Terrain Filter
Dialog box	SceneCommands\Staircase
Create windows	Modelers\Step4
Set up menus	Modelers\Step4
Set up/update Properties	Modelers\Step4
Set up Sequencer hierarchy	Modelers\Step4

Before you begin

Chapter 2 -, "Creating Your Own Component" will guide you through creating a MSVC++ project in order to create an extension. Before you develop an extension, make sure you have done the following:

1. Read the section in this cookbook describing the family you are interested in, and make the appropriate technical/algorithmic decisions related to your own application.

2. Review the examples in the SDK that relate to the type of extension you want to create. The cookbook chapters will give you details on what samples are available.

3. Use the example in the SDK as a framework to do your own extension. Each family has an example. Read the section below to start your new extension project.

Creating Your Own Component

This chapter contains step-by-step information that guides you through the process of creating a new component project of your own for any family type.

Your project

Your extension project should typically have at least two folders: Source Files and SDK Files. Source Files contains the files specific to your extension and SDK Files contains the files needed from the SDK to compile your extension.

SDK Files:

The standard files generally needed in your SDK Files folder will include the following:

SDK\Include\Common\BasicCOMExtImplementations.h, .cpp
SDK\Include\Common\BasicCOMImplementations.h
SDK\Include\Common\COMUtilities.h
SDK\Source\COMUtilities.cpp
or
CarraraLib
You will also need:
SDK\Include\Carrara\Basic*MyExtensionType*.h

This file is the interface of the basic implementation of your extension's family (shaders, light sources, etc.). You may need to include different Basic***.h and Basic***.cpp files depending on the family you use.

Source Files:

The standard files in your Source Files folder will include the following:

MyExtension.h
MyExtension.cpp
MyExtensionDef.h
MyExtension.def

Each of these files will be described in detail below.

Architecture of your extension

As you already know, your extension works like a COM object that needs to communicate with the shell application. In order to perform this communication and to define a User Interface (UI), you will have to create **resources** and link them to your extension.

The whole system of communication is based on IDs. So, a class that needs to be instantiated by the Shell must have an ID, defined in the resources of the extension. This ID, called a **GUID**, must be unique. Therefore, when you select your extension in the UI of Carrara (in a menu for instance), a GUID is found in the corresponding resource and a request for the creation of an object matching this GUID is sent. At this point, you simply have to respond to this request by the instantiation of your class.

As a result, you will have at least to create these files:

- MyExtension.h and MyExtension.cpp (the interface and implementation files of your main class)
- MyExtension.r (the resource file which is read by the Shell)
- MyExtension.def (defines the extension's name)

Copying a Sample

The easiest way to create a component project is to start from an existing sample. What you may want to do

is copy the entire folder of one of the samples and then adapt the files in it. In the following example, we use the Terrain Filter sample.

MyExtension.r, MyExtension.rsr

These files should be added to the resource project, not the main project. The MyExtension.r file contains a description of the resource. You can also add a .rsr file generated by MCSketch to add your user interface elements. If you have a .rsr file, you need to include (not #include) the file in the .r file. With a .rsr file, the PMap resource will be used to automatically generate a simple UI. Here is an example of .r file taken from the Terrain Filter sample:

```
#include "External3DAPI.r"
#include "Copyright.h"
#include "simpleterrainfilterDef.h" /* contains the IDs we created */
#include "interfaceids.h"

#ifndef qUsingResourceLinker
include "simpleterrainfilter.rsr";
#endif
```

You can define several components in a resource file. The resources of the same component are identified by the number following the '(' of each resource definition. Here we have only one component whose number ID is defined by R_SimpleTerrainFilterID (whose value is defined in SimpleTerrainFilterDef.h).
This number also matches a potential node part created with MCSketch.

```
// Simple Terrain Filter

resource 'COMP' (R_SimpleTerrainFilterID, "Simple Filter", purgeable)
{
   'tfil', /* terrain filter family ID */
   'simp', /* our component ID */
   "SDK Simple Filter", /* name of our component displayed in the UI */
   "Filter", /* name of the family displayed in the UI */
   FIRSTVERSION,
   VERSIONSTRING,
   COPYRIGHT,
   kRDAPIVersion
};

resource 'GUID' (R_SimpleTerrainFilterID, "Simple Filter", purgeable)
{
   {
      R_IID_I3DExTerrainFilter, /* GUID of the family */
      R_CLSID_SimpleTerrainFilter /* GUID of our component */
   }
};

resource 'PMap' (R_SimpleTerrainFilterID, "Simple Filter", purgeable)
{
   {
      'stre','re32',   0, "Strength","",
   }
};
```

MyExtension.h, MyExtension.cpp

Your main class will derive from a basic implementation of a family. In this example, we derive from TBasicTerrainFilter. Three points must be addressed regarding the interface of your main class:

- add the **STANDARD_RELEASE** macro directly after the declaration of your constructor and destructor (for more informartion, see the Overview book)
- add a static data field to store your component **GUID** (here: *sClassId*)
- override the **`GetParamsBufferSize`**, **GetExtensionDataBuffer** and **ExtensionDataChanged**

methods if needed (for more information, see the Overview books)

```
class TSimpleTerrainFilter : public TBasicTerrainFilter
{
public:
   struct TParameterMap // this is the set of UI parameters
   {
      TParameterMap();
      real fStrength;
   };

public:
   static const MCGUID sClassId; /* stores the class ID (GUID) */

public:
   TSimpleTerrainFilter();
   ~TSimpleTerrainFilter();
   STANDARD_RELEASE;

   virtual void* MCCOMAPI GetExtensionDataBuffer();
   virtual int32 MCCOMAPI GetParamsBufferSize() const
   { return sizeof(fParameterMap);}

   virtual void MCCOMAPI Shuffle() {}
   virtual void MCCOMAPI Filter(TMCArray<real>& heightField, TVector2& height-
      Bound,const TIndex2& size, const TVector2& cellSize);
   virtual boolean MCCOMAPI CanBuildPreview() { return true; }

protected:
   real         GetValue(real x, real y);

protected:
   TParameterMap fParameterMap;

};
```

In the implementation file, besides the definition of your own methods, you need to:

- define your static field and assign it to the constant value used in the resource file:

```
const MCGUID TSimpleTerrainFilter::sClassId = { R_CLSID_SimpleTerrainFilter };
```

- add and adapt the following methods called by the Shell:

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

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

This method instantiates our main class if the matching *classId* has been passed in.

```
TBasicUnknown* MakeCOMObject(const MCCLSID& classId)
{
   TBasicUnknown* res = NULL;
   if (classId == TSimpleTerrainFilter::sClassId)
       res = new TSimpleTerrainFilter;
   return res;
}
```

MyExtensionDef.h

It's a good idea to gather your ID definitions in a separate file. In the case of the Terrain Filter sample, we only have two constants to define:

- our component GUID (used in the .r file and the .cpp file):

```
#define R_CLSID_SimpleTerrainFilter 0x31fbe7d9, 0xd041, 0x4941, 0xae, 0xa3,
       0xb9, 0x36, 0xd3, 0x1d, 0x70, 0x27
```

- and the component number ID (used in the .r file):

```
#define R_SimpleTerrainFilterID 128
```

MyExtension.def

This file aims at giving your extension a name and only has one line to be changed:

```
LIBRARY   SimpleTerrainFilter
EXPORTS
           MCDllInit
           MCDllGetClassObject
           MCDllCanUnloadNow
           MCDllCleanUp
           MCDllInitExceptionTranslator
           MCDllCleanUpExceptionTranslator
```

Family-Specific Resources

This chapter describes the format for family-specific resources.

‘Modu’ resource

The Modu resource is used to give references to the components of an External Module.

```
type 'Modu' {
    literal longint;/* Family */
    literal longint;/* Class */
    pstring[63];         /* Name */
    pstring[63];         /* SubFamily Name */
    literal int;         /* WindowID */
    literal int;         /* MBarID */
    literal int;         /* TBarID */
    literal int;         /* PrefsMappingResID */
    literal int;         /* PrefsViewResID */
    literal int;         /* DefaultPrefsResID */
    literal longint;/* Workspace class signature */
    literal longint;/* version */
    pstring[31];         /* Version string */
    pstring[255];        /* Comment */
    literal int;         /* minimized iconID */
    literal longint;/* window context menuID or -1 */
    literal int;         /* boolean: is document module, used for external modules
        */
    };
```

'CMNU' resource

The CMNU resource is used in an External Module to build menus.

```
type 'CMNU' {
    integer;
    fill word[2];
    integer textMenuProc = 0;
    fill word;
    unsigned hex bitstring[31] allEnabled = 0x7FFFFFFF;
    boolean disabled, enabled;
    pstring;
    wide array {
        pstring;
        byte noIcon;
        char noKey = "\0x00", hierarchicalMenu = "\0x1B";
        char noMark = "\0x00", check = "\0x12";
        fill bit;
        unsigned bitstring[7]plain;
        align word;
        longint nocommand = 0;
        };
    byte = 0;
    };
```

'MBAR' resource

The MBAR resource is used in an External Module to build a Menu Bar.
It contains an array of 'CMNU' ID resources.

Example :

```
resource 'MBAR' (125,"My External Modeler",purgeable) {
    { 118 , 110 }
};

resource 'CMNU' (118,"Item 1",purgeable) {
    ...
    };
resource 'CMNU' (125,"Item 2",purgeable) {
    ...
    };
```

'TBAR' resource

The TBAR resource is used in an External Module to build a ToolBar. It describes a list of icons with keyboard shortcuts, an action number and a tooltip name.

```
type 'TBAR' {
    pstring;
    integer = $$CountOf(ToolArray);/* Number of tools*/
    wide array ToolArray{
        integer;  /* tool resource ID */
        fill BYTE;
        char noKey = "\0x00";/* tool char equiv*/
        integer noAction = -1, noDisabling = -2;/* action number*/
        pstring;  /* tool name*/
        };
    };
```

Example:

```
resource 'TBAR' (kPreviewZoomTBAR, "Preview Zoom Tools", purgeable)
{
    "Preview Zoom Tools", straightLayout,
    {
        k2DPanToolID,"2D Pan"," ", noAction, alwaysEnabled, style_inset {120};
        k2DZoomToolID,"2D Zoom","z", noAction, alwaysEnabled, style_inset {121};
    }
};
```

If no action number is specified, then the icon will behave like a tool icon. If an action number is specified (just like in a CMNU resource), then the icon will behave like a button.

'prfs' resource

The 'prfs' resource gives the default preferences values of an extension.

```
type 'prfs'{
    cstring;
};
```

The string contains data in the rdd.ini format.
Example with 2 variables: BCOL and FCOL.

```
resource 'prfs' (144) {
    "{\n"
    "   BCOL 2\n"
     "FCOL 5\n"
    "}\n"
}
```

The Component Private resources (“Cmpp”)

Some external components like the exporter, the importer, and the geometric primitives have an extra resource called “Cmpp”. This resource has a different structure for each family.

Exporter and Importer

The “Cmpp” resource has the following structure for these two families:

```
type 'Cmpp'{     /* 'RDin' private data */
    longint;     /* Family Signature */
    longint;     /* Class Signature */
    longint;     /* flags */
    literal longint;/* creator (for Macintosh exporters only) */
// The following is the 'FTYP' format of RFrame, with only 1 entry
    FTYPDATA
};
```

Where FTYPDATA is:

```
#define FTYPDATA\
    integer = $$CountOf(TypeArray); /* Number of types*/ \
    wide array TypeArray{\
        literal longint; /* Reference */\
        cstring;/* Name */ \
        integer = $$CountOf(MacTypeArray); \
        wide array MacTypeArray{ \
            literal longint; /* Mac type  */ \
            }; \
        integer = $$CountOf(ExtTypeArray); \
        wide array ExtTypeArray{ \
            literal longint;/* Extension (last char ignored)  */ \
            }; \
        };
```

And the different flag values are :

```
#define isAlienType 0
#define isMainType 1
#define hasExportOptions 2
```

For example, the DXF Importer and Exporter “Cmpp” resources are:

```
resource 'Cmpp' (128) {
    '3Din',
    'dxf ',
    isAlienType,
    'RD3A',          // creator
    {
        'DXF ',    // Reference
        "DXF",         // Name
        {'TEXT' },// List of MacOS types
```

```
        {'DXF ' },// List of extensions
    };
};

resource 'Cmpp' (129) {
    '3Dou',
    'dxf ',
    isAlienType,
    'ttxt',          // creator
    {
        'DXF ',   // Reference
        "DXF",        // Name
        {'TEXT' },// List of MacOS types
        {'DXF ' },// List of extensions
    };
};
```

Primitives

This resource contains the ID (a short) of the main icon of the primitive (it will be shown in the hierarchy window).

```
type 'Cmpp' {
    integer; /*Icon ID*/
};
```

The two icons in the toolbar (not selected and selected) have their ID calculated like this: not selected ID = main ID + 50, and selected ID = main ID + 100.
Because the Shell does not support Windows Icon, you must not define a “Cmpp” resource in a Windows resource file.

Module

The editing capabilities of a modeler are defined in this resource.

```
type 'Cmpp' {// Private data of external modeler
    longint;// Can edit patches ?
    longint;// Can edit facets ?
    longint = $$Countof(primitiveIDList);
    array primitiveIDList {
        literal longint;// primitive ID
        longint;         // Native flag
        };
    };
```

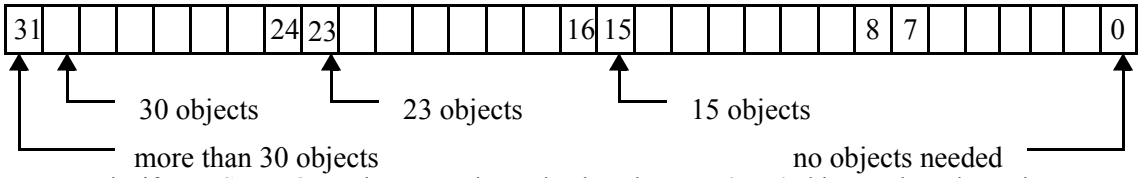
In file I3DExMdr.h, you will find the C data structure corresponding to the ‘Cmpp’ resource:

```
typedef struct{
    longfCanEditPatches;
    longfCanEditFacets;
    longfPrimitiveIDNb;
    SExModelerPrimitiveDatafPrimitiveList[1];
    } SExModelerData;
```

The first field of ‘Cmpp’ (fCanEditPatches) must be TRUE if modeler can edit any patch based object, FALSE is not.
The second field of ‘Cmpp’ (fCanEditFacets) must be TRUE if modeler can edit any facet based object, FALSE is not.
The last field (fPrimitiveList) is simply a list of primitive Class IDs. All those primitives are editable in the modeler. The field of ‘Cmpp’ just before (fPrimitiveIDNb) is the number of primitive Class IDs in the list (it is the size of the list).

Scene Operations

This resource is used to know in which case the Scene Operation can be called by the Shell. This resource only contains a long (4 bytes) in which each bit has a different signification:



For example, if your Scene Operation can only work when there are 2 or 4 objects selected, you have to put this value in the Cmpp resource:

```
00000000 00000000 00000000 00010100 = 20 = 0x14
```

Or, if your Scene Operation can be used when there is 1 or more object selected, you will have:

```
11111111 11111111 11111111 11111110 = 0xFFFFFFFE
```

And if your Scene Operation does not need any selection, it will be:

```
00000000 00000000 00000000 00000001 = 1
```

If you do not define the Cmpp resource, your Scene Operation can be used in every case (by default the value is 0xFFFFFFFF).
This resource can be created on both cross-Platform and Windows-only sides, but for the Windows resource do not forget to put the long in the Motorola format (Highest byte first).

Writing an AtmosphericShader

Family ID : 'atmo'
Interface ID : IID_I3DExAtmosphericShader
Interface file : I3DExEnvironment.h
Basic Implementation file: BasicEnvironment.h, BasicEnvironment.cpp

Overview

An atmospheric shader creates the illusion of a volumetric atmosphere by passing the light rays through a filter. The filter can alter the color of the rays according to the atmospheric effect being created.
There are two methods which must be implemented:

- **SegmentFilter(TVector3& beg, TVector3& end, TMCColorRGB& filterOut, boolean indirecLight)**
- **DirectionFilter(TVector3& origin, TVector3& direction, TMCColorRGB& filterOut, boolean indirectLight)**

SegmentFilter() is called for a finite ray (of which we know the beginning and the end). **DirectionFilter()** is called for an infinite ray (of which we know the beginning and the direction). See the reference manual for more specific information on the interface methods.

Location in the User Interface

An Atmospheric shader is accessed from the Properties tray after selecting the scene from the objects list.

Example: Fog

Description

This example simulates fog within the scene. A minimum altitude and a maximum altitude along the z-axis define the area of the fog. It is infinite along the x-axis and y-axis.

Parameters

The following variables are used for the user-input parameters:

```
fcolor:        This is used as the filter color for light rays found
   within the fog area.
fZmax:         The maximum value of the fog area along the z-axis.
fZmin:         The minimum value of the fog area along the z-axis.
fVisibility:   The distance of visibility within the fog area.
```

These are defined in Fog.h as follows:

```
// Data storage of our extension :
struct FogData
{
  TColorRGBA fColor;      // "Color" of the fog
  real       fZmax;       // Minimum altitude of the fog
  real       fZmin;       // Maximum altitude of the fog
  real       fVisibility; // Distance of Visibility
};
```

These are then mapped to the user interface parameters using the PMap resource in Fog.r.
Note that the parameters within the *FogData* struct must be in the same order as in the PMap.
The source color is filtered according to the color of the fog and the distance of visibility within the fog. The resulting filtered color is found by using this formula:

$$FilteredColor = SourceColor \times \text{Attenuation Factor} + FogColor \times (1 - \text{Attenuation Factor})$$

The Attenuation factor, also known as the filter coefficient, is a ratio of the distance within the fog of the source and of the distance of visibility. It is expressed using this formula:

$$\text{Attenuation Factor} = 1 - \frac{\text{distance in the fog}}{\text{distance of Visibility}}$$

If the distance within the fog is greater than the visibility, the filtered color is returned as the fog color.

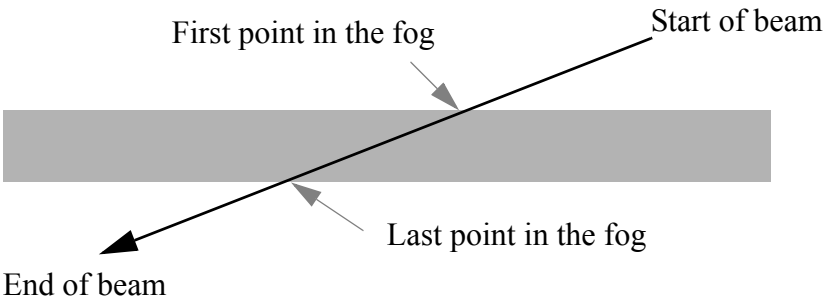
Functions

Two main functions must be implemented. The first, **SegmentFilter()**, is for finite light rays (for instance, the ray between a spot light and an object). The second, **DirectionFilter()**, is for infinite light rays (for instance, a light ray which doesn't intersect any of the objects in the scene).

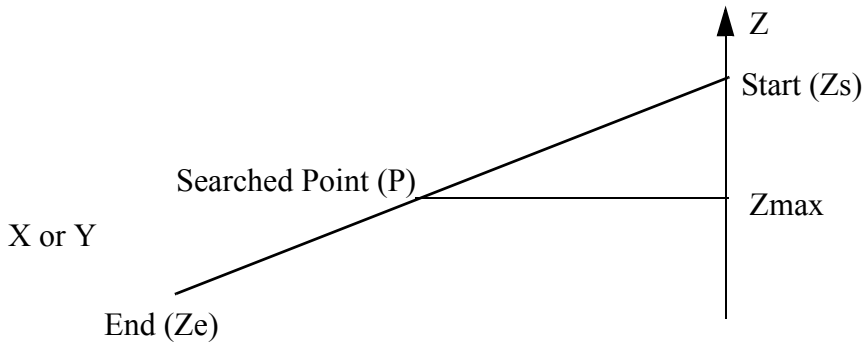
Fog::SegmentFilter

```
MCCOMErr Fog::SegmentFilter(TVector3* beg, TVector3* end, TColorRGB* filterOut)
```

The **SegmentFilter()** function must find the first and last point of a vector within the fog area.



To find the coordinates of the First or Last point in the fog, you can use the following expressions:



$$X_P = X_S + (X_E - X_S) \times \frac{(Z\max - Z_S)}{(Z_E - Z_S)}$$

$$Y_P = Y_S + (Y_E - Y_S) \times \frac{(Z\max - Z_S)}{(Z_E - Z_S)}$$

These expressions are implemented in **SegmentFilter()**:

```
MCCOMErr Fog::SegmentFilter(TVector3* beg, TVector3* end, TColorRGB* filterOut)
{
  real distanceInTheFog;
  TVector3beamVector;
  TVector3beaminfogbeg, beaminfogend;
  real filtercoef, colorcoef;

  if ((beg->z > fData.fZmax) && (end->z > fData.fZmax))
  {
     // the light beam is above the fog don't do anything
  }
  else if ((beg->z < fData.fZmin) && (end->z < fData.fZmin))
  {
     // the light beam is under the fog don't do anything
  }
  else
  {
   // The Light beam crosses the fog
   if (beg->z > fData.fZmax)
   {
      beaminfogbeg.z = fData.fZmax;
      beaminfogbeg.x = beg->x + (end->x - beg->x) / (end->z - beg->z) *
       (fData.fZmax - beg->z);
      beaminfogbeg.y = beg->y + (end->y - beg->y) / (end->z - beg->z) *
       (fData.fZmax - beg->z);
    }
    else if (beg->z < fData.fZmin)
   {
      beaminfogbeg.z = fData.fZmin;
      beaminfogbeg.x = beg->x + (end->x - beg->x) / (end->z - beg->z) *
       (fData.fZmin - beg->z);
      beaminfogbeg.y = beg->y + (end->y - beg->y) / (end->z - beg->z) *
       (fData.fZmin - beg->z);
    }
    else
   {
```

```
    beaminfogbeg.x = beg->x;
    beaminfogbeg.y = beg->y;
    beaminfogbeg.z = beg->z; // the begining point of the light beam is in the
     fog
   }
   if (end->z > fData.fZmax)
  {
    beaminfogend.z = fData.fZmax;
    beaminfogbeg.x = beg->x + (end->x - beg->x)/(end->z - beg->z) *
     (fData.fZmax - beg->z);
    beaminfogbeg.y = beg->y + (end->y - beg->y)/(end->z - beg->z) *
     (fData.fZmax - beg->z);
   }
   else if (end->z < fData.fZmin)
  {
    beaminfogend.z = fData.fZmin;
    beaminfogend.x = beg->x + (end->x - beg->x) / (end->z - beg->z) *
     (fData.fZmin - beg->z);
    beaminfogend.y = beg->y + (end->y - beg->y) / (end->z - beg->z) *
     (fData.fZmin - beg->z);
   }
   else
  {
    beaminfogend.x = end->x;
    beaminfogend.y = end->y;
   beaminfogend.z = end->z; // the ending point of the light beam is in the fog
   }
```

You can then find the distance within the fog by calculating the norm of the last point minus the first point.

```
   beamVector.x = beaminfogend.x - beaminfogbeg.x;
   beamVector.y = beaminfogend.y - beaminfogbeg.y;
   beamVector.z = beaminfogend.z - beaminfogbeg.z;
   distanceInTheFog = sqrt(beamVector.x*beamVector.x + beamVector.y*beamVec-
      tor.y + beamVector.z*beamVector.z);
   filtercoef = 1.0f - distanceInTheFog / fData.fVisibility;

   if (filtercoef < 0.0f)
  {
    filtercoef = 0.0f;
   }
   colorcoef = 1.0f - filtercoef;
   filterOut->R = filterOut->R*filtercoef + fData.fColor.R*colorcoef;
   filterOut->G = filterOut->G*filtercoef + fData.fColor.G*colorcoef;
   filterOut->B = filterOut->B*filtercoef + fData.fColor.B*colorcoef;
 }
```

Fog::DirectionFilter

```
MCCOMErr Fog::DirectionFilter(TVector3* origin, TVector3* direction, TColorRGB*
      filterOut)
```

DirectionFilter() is similar to the **SegmentFilter()**, but you are given only the beginning point of the light beam and its direction. The light beam is infinite in this direction.

To determine the distance of the beam in the fog, you calculate the entry point and the exit point of the light beam.

```
MCCOMErr Fog::DirectionFilter(TVector3* origin, TVector3* direction, TColorRGB*
      filterOut)
{
  real distanceInTheFog;
  TVector3beamVector;
  TVector3beaminfogbeg, beaminfogend;
  real filtercoef, colorcoef;
```

```
if ((origin->z > fData.fZmax) && (direction->z >= 0.0f))
{
  // Don't do anything
}
else if ((origin->z < fData.fZmin) && (direction->z <= 0.0f))
{
  // Don't do anything
}

else
{
  if ((origin->z > fData.fZmax) || (origin->z < fData.fZmin))
  {
     beaminfogbeg.z = fData.fZmax;
     beaminfogbeg.x = origin->x + direction->x*(fData.fZmax - origin->z) /
      direction->z;
     beaminfogbeg.y = origin->y + direction->y*(fData.fZmax - origin->z) /
      direction->z;
     beaminfogend.z = fData.fZmin;
     beaminfogend.x = origin->x + direction->x*(fData.fZmin - origin->z) /
      direction->z;
     beaminfogend.y = origin->y + direction->y*(fData.fZmin - origin->z) /
      direction->z;
  }
  else if (direction->z > 0.0f)
  {
     beaminfogbeg.x = origin->x;
     beaminfogbeg.y = origin->y;
     beaminfogbeg.z = origin->z;
     beaminfogend.z = fData.fZmax;
     beaminfogend.x = origin->x + direction->x*(fData.fZmax - origin->z) /
      direction->z;
     beaminfogend.y = origin->y + direction->y*(fData.fZmax - origin->z) /
      direction->z;
  }
  else if (direction->z < 0.0f)
  {
     beaminfogbeg.x = origin->x;
     beaminfogbeg.y = origin->y;
     beaminfogbeg.z = origin->z;
     beaminfogend.z = fData.fZmin;
     beaminfogend.x = origin->x + direction->x*(fData.fZmin - origin->z) /
      direction->z;
     beaminfogend.y = origin->y + direction->y*(fData.fZmin - origin->z) /
      direction->z;
  }
  else
  {
    // direction->z=0.0
    *filterOut = fData.fColor;
    return MC_S_OK;
  }
```

The same formula as in **SegmentFilter()** is then used.

Writing a Background

Family ID : 'back'
Interface ID : IID_I3DExBackground
Interface file : I3DExEnvironment.h
Basic Implementation files: BasicEnvironment.h, BasicEnvironment.cpp

Overview

A background is used to simulate an environment around all objects. The image data of the background is wrapped to an imaginary sphere surrounding the scene, and can be reflected and refracted by the objects within.
The *I3DExBackground* interface has the following methods:

- **GetBackgroundColor(TMCColorRGB &resultColor,const TVector3 &direction)**
- **GetBackgroundColor(TMCColorRGB &resultColor,real &confidence,const BackGroundArea &area)**

These two implementations are similar to shaders. The first implementation of **GetBackgroundColor()** defines the color at each point in the image. The variable *direction* is a vector from the camera to the point being rendered.
The second implementation uses a *BackGoundArea*. This is similar to a *ShadedArea* (see , "Writing a Shader"). It is used to calculate an average for each pixel.
You must implement the point version of **GetBackgroundColor()**. You can optionally implement the *BackGroundArea* version.

Location in the User Interface

Backgrounds and Backdrops are accessible in the Properties tray of the Assemble room, after selecting the Scene from the Instances list. It is important to note that a background won't show up in the final image, if a backdrop has been selected too.

Example: Back

Description

This example of background will simulate a sunset above the horizon line, and a flat ground color below the horizon line.

Parameters

The following variables for the user-input parameters are used:

```
fSunColor:     The color returned for the sun.
fSunDiameter:  The diameter of the sun.
fWestDirection:The direction of the sun.
fWestColor:    The color of the sky around the sun.
fEastColor:    The color of the sky opposite the sun.
fEarthColor:   The color returned below the horizon line.
```

These are defined in Back.h as follows:

```
struct BackData
```

```
{
    TMCColorRGBA fSunColor;
    real          fSunDiameter;
    real          fWestDirection;
    TMCColorRGBA fZenithColor;
    TMCColorRGBA fWestColor;
    TMCColorRGBA fEastColor;
    TMCColorRGBA fEarthColor;
};
```

These are then mapped to the user interface parameters using the PMap resource in Back.r. Note that the parameters within the *BackData* struct must be in the same order as in the PMap.

Functions

Back::GetBackgroundColor

```
MCCOMErr Back::GetBackgroundColor(TMCColorRGB &color,const TVector3 &direction)
```

The *BackGroundArea* version of **GetBackgroundColor()** will not be implemented in this example. This implementation uses *direction* to determine where in the sky or ground the pixel is located, and then returns the appropriate color in the variable *color* based on the UI parameters.

```
MCCOMErr Back::GetBackgroundColor(TMCColorRGB &color,const TVector3 &direction)
{
    real in_sun;

    in_sun=(direction)[0]*fWestVector[0] + (direction)[1]*fWestVector[1] +
        (direction)[2]*fWestVector[2];
    if ((direction)[2]<0.0)
    { // You look the earth and not the sky
        color=fData.fEarthColor;
    }
    else if (in_sun>fSunLimit)
    { // You look directly at the sun
        color=fData.fSunColor;
    }
    else if (in_sun>0.0)
    { // You look in the West Direction
//      color->Mode=0;
        color.R   =fData.fWestColor.red*in_sun+fData.fZenithColor.red*(1.0-
        in_sun);
        color.G   =fData.fWestColor.green*in_sun+fData.fZenithColor.green*(1.0-
        in_sun);
        color.B   =fData.fWestColor.blue*in_sun+fData.fZenithColor.blue*(1.0-
        in_sun);
    }
    else
    { // You look in the East Direction
//      color->Mode=0;
        color.R   =-fData.fEastColor.red*in_sun+fData.fZenith-
        Color.red*(1.0+in_sun);
        color.G   =-fData.fEastColor.green*in_sun+fData.fZenith-
        Color.green*(1.0+in_sun);
        color.B   =-fData.fEastColor.blue*in_sun+fData.fZenith-
        Color.blue*(1.0+in_sun);
    }
    return MC_S_OK;
}
```

Writing a Camera

Family ID : ‘came’
Interface ID : IID_I3DExCamera
Interface file : I3DExCamera.h
Basic Implementation files: BasicCameraLightGel.h, BasicCameraLightGel.cpp

Overview

Cameras define the projection from the 3D scene to the 2D rendered image on the screen.
I3DExCamera has the following methods:

- **SetTransform(const TTransform3D& transform)**

SetTransform() is called prior to any of the other methods. It can be used for pre-processing, and for assigning the camera transforms to a private variable.

- **CreateRay(const TVector2&screenPosition, TVector3& resultOrigin, TVector3& result-Direction)**
- **CreateRay(const TVector2& screenPosition,const TVector2& screenDerivate, Ray3D &ray)**

CreateRay() calculates a ray from the *screenPosition* to the camera. It should return true if a ray could be calculated, and false otherwise. The ray tracer uses this method.

- **IsProjectionAffine(void)**

IsProjectionAffine() should return true only if the ’w’ component (the 4th one) from the projection is always 1.0.

- **PreProject3DTo2D(const TVector3& cameraPosition, TVector4& projectedPosition**

PreProject3DTo2D() can be used to quickly check if a point is behind the camera by using its projection.

- **Project3DTo2D(const TVector3& cameraPosition, TVector3& screenPosition)**

Project3DTo2D() must be implemented for your camera to be used by the Hybrid Ray Tracer renderer.

- **GetStandardCameraInfo(TStandardCameraInfo& cameraInfo)**

For your camera to be used by the interactive renderer, you must implement **GetStandardCameraInfo().** For more information on **TStandardCameraInfo**, see data structures chapter in the reference.

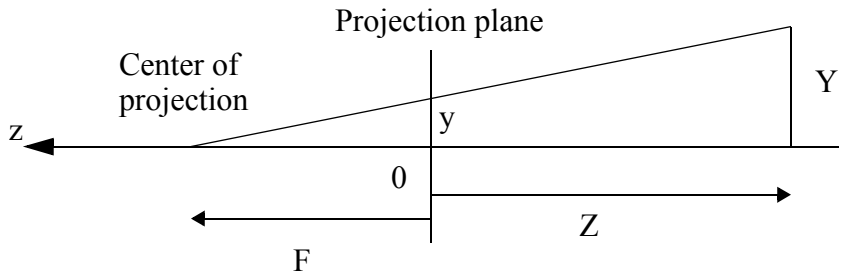
Location in the Interface

Cameras can be added from the Insert Menu in the Assemble room. If an Icon is provided in a UImg resource, a button will also be added to the tool set.

Example: Camera

Description

Camera creates a Conical Camera with a focal length of 25mm and a zoom lens. It is displayed by the Hybrid Ray Tracer and interactive renderers.
Below is the standard definition of the conical projection:



Parameters

The following variable is used for the user-input parameters:

`fZoomCoef:` **`The amount of zoom.`**

This is defined in Camera.h as follows:

```
struct CameraData
{
   int16 fZoomCoef;
};
```

This is then mapped to the user interface parameters using the PMap resource in Camera.r.
Note that the parameters within the *CameraData* struct must be in the same order as in the PMap.

Functions

Camera::SetTransform

```
MCCOMErr Camera::SetTransform( const TTransform3D& transform )
```

SetTransform() is used to copy the transformation data:

```
MCCOMErr Camera::SetTransform( const TTransform3D& transform )
{
   fTransform=transform; // Copy the data of transform and not the pointer.

   fCameraOrigin = TVector3::kZero;
   fXCameraDirection.SetValues( fHalfWidth, 0.0, 0.0 );
   fYCameraDirection.SetValues( 0.0, fHalfWidth, 0.0 );
   fZCameraDirection = TVector3::kNegativeZ;

   fCameraOrigin=fTransform.TransformPoint(fCameraOrigin);
   fXCameraDirection=fTransform.TransformVector(fXCameraDirection);
   fYCameraDirection=fTransform.TransformVector(fYCameraDirection);
   fZCameraDirection=fTransform.TransformVector(fZCameraDirection);

   return MC_S_OK;
}
```

Camera::Project3DTo2D

```
boolean Camera::Project3DTo2D(const TVector3& cameraPosition, TVector3& screen-
      Position ) const
```

This functions returns true if the point passed in *screenPosition* is visible by the camera, and false otherwise.

```
boolean Camera::Project3DTo2D(const TVector3& cameraPosition, TVector3& screen-
      Position ) const
{
      const real distToCOP = - cameraPosition.z;

   if ( distToCOP > 0 )
   {
      const real invW = 1.0/distToCOP;
```

```
            screenPosition.x = fInvHalfWidth * cameraPosition.x * invW;
            screenPosition.y = fInvHalfWidth * cameraPosition.y * invW;
            screenPosition.z = distToCOP;

            return true;
        }

        return false;
    }
```

Camera::CreateRay

```
boolean Camera::CreateRay( const TVector2&screenPosition, TVector3& resultOri-
      gin, TVector3& resultDirection)
```

CreateRay() is used by the ray tracer. In this example it is the opposite of the transformation used in **Project3DTo2D()**:

```
BOOLEAN Camera::CreateRay( const TVector2&screenPosition, TVector3& resultOri-
      gin, TVector3& resultDirection) const
{
    resultOrigin = fCameraOrigin; // origin is always at the centerpoint

    // direction changes since rays converge to the camera center
    resultDirection[0] = fZCameraDirection.x + ( screenPosition[0]*fXCameraDirec-
        tion[0]+screenPosition[1]*fYCameraDirection[0] );
    resultDirection[1] = fZCameraDirection.y + ( screenPosition[0]*fXCameraDirec-
        tion[1]+screenPosition[1]*fYCameraDirection[1] );
    resultDirection[2] = fZCameraDirection.z + ( screenPosition[0]*fXCameraDirec-
        tion[2]+screenPosition[1]*fYCameraDirection[2] );

    resultDirection.Normalize();

    return true;
}
```

Camera::CreateRay

```
boolean Camera::CreateRay(const TVector2& screenPosition,const TVector2& screen-
      Derivate, Ray3D &ray)  const
```

CreateRay() generates a ray through a given screen location [-1,1] with a given thickness (screenDerivate).

```
{
    TVector3 &direction=ray.fDirection;

    direction.x = fZCameraDirection.x + screenPosition[0] * fXCameraDirection.x +
        screenPosition[1] * fYCameraDirection.x;
    direction.y = fZCameraDirection.y + screenPosition[0] * fXCameraDirection.y +
        screenPosition[1] * fYCameraDirection.y;
    direction.z = fZCameraDirection.z + screenPosition[0] * fXCameraDirection.z +
        screenPosition[1] * fYCameraDirection.z;

    real invNorm = 1.0f / direction.GetMagnitude();

    direction *= invNorm;

    ray.fDx = ( fXCameraDirection - direction*(direction*fXCameraDirection) )
        *(screenDerivate.x*invNorm);
    ray.fDy = ( fYCameraDirection - direction*(direction*fYCameraDirection) )
        *(screenDerivate.y*invNorm);

    ray.fOrigin = fCameraOrigin;
```

```
        ray.fOx=TVector3::kZero;
        ray.fOy=TVector3::kZero;

        return true;
    }
```

Writing a Constraint

Family ID : 'link'
Interface ID : IID_I3DExConstraint
Interface file : I3DExConstraint.h
Basic Implementation file: Basic3DCOMImplementations.h, Basic3DCOMImplementations.cpp

Overview

A constraint, known as a link in RDS, defines motion links between Tree Elements. It allows you to define "mechanical" constrains (degrees of freedom) between one Tree Element and its father.
In order to animate the link, you have to put each freedom value (one for each degrees of freedom) in the "PMap", even if they do not appear in the User Interface.
This sample creates a link that acts like a screw. So you need to specify that it has only one degree of freedom, the number of turns, and the axis and the step of the screw.
The following methods must be implemented:

- **GetNbrFreedom()**
- GetFreedomRange(int16 index, real& min, real& max)
- **IncrementFreedomValue(int16 index, real& increment)**
- GetTransform(TTransform3D& transform)
- GetTransformPartialDerivate(int16 index, TTransform3D& transform)

GetNbrFreedom() lets the 3D Shell know the number of degrees of freedom. Because the freedom value can be limited, you have to give the range of values. This range is defined as a range of increment/decrement around the value.
In this sample, you don't want to limit the number of turns so you'll always return the maximum range of increment and decrement, with **GetFreedomRange()**. The Shell can't directly modify the freedom value, so you need to implement **IncrementFreedomValue()**. **GetTransform()** gives the rotation matrix and translation vector due to the freedom values. To allow the inverse kinematics mechanism to work with your constraint, you have to implement the **GetTransfromPartialDerivate()** function.
See the reference manual for more specific information on the interface methods.

Location in the User Interface

A constraint is accessed from the Properties tray: Motion/Transform button: constraint panel, after selecting the child object or group.

Example: ScrewConstraint

Description

The sample link acts like a screw. It has one degree of freedom, and you need to specify the number of turns, the axis, and the step of the screw.

Parameters

The following variables are used for the user-input parameters:

```
fAxis:          This is the id of the axis.
fStep:          The step of the screw, or 1 turn is a translation of
   fStep.
fFreedomValue: The range of values for the degrees of freedom.
```

These are defined in ScrewConstraint.h as follows:

```
// Data storage of our extension :
struct ScrewConstraintData
{
   int32fAxis;  // ID of the axis
   real32fStep; // Step of the screw (1 turn -> translation of fStep)
   real32fFreedomValue;
};
```

These are then mapped to the user interface parameters using the PMap resource in ScrewConstraint.r.

Functions

ScrewConstraint::GetNBRFreedom

```
virtual int16  MCCOMAPI GetNbrFreedom() const
```

GetNbrFreedom() lets the 3D Shell know the number of degrees of freedom. Because the freedom value can be limited, you have to give the range of values. This range is defined as a range of increment/decrement around the value.

```
int16 ScrewConstraint::GetNbrFreedom() const
{
   return 1;
}
```

ScrewConstraint::IncrementFreedomValue

```
MCCOMErr ScrewConstraint::IncrementFreedomValue(int16 index, real& increment)
```

The Shell can't directly modify the freedom value, so you need to implement **IncrementFreedomValue()**.

```
MCCOMErr ScrewConstraint::IncrementFreedomValue(int16 index, real& increment)
{
   if (index == 1)
      fData.fFreedomValue += increment;

   return MC_S_OK;
}
```

ScrewConstraint::GetFreedomRange

```
MCCOMErr ScrewConstraint::GetFreedomRange(int16 index, real& min, real& max)
      const
```

In this sample, you don't want to limit the number of turns so you'll always return the maximum range of increment and decrement with **GetFreedomRange()**.

```
MCCOMErr ScrewConstraint::GetFreedomRange(int16 index, real& min, real& max)
      const
{
   if (index == 1)
   {
      min = -32767.0f;
      max =  32767.0f;
   }
   return MC_S_OK;
}
```

ScrewConstraint::GetTransform

```
MCCOMErr ScrewConstraint::GetTransform(TTransform3D& transform) const
```

GetTransform() gives the rotation matrix and translation vector due to the freedom values.

Note: When the freedom values are initialized the function **GetTransform** must return the identity matrix for the rotation and a null vector for the translation.

```
MCCOMErr ScrewConstraint::GetTransform(TTransform3D& transform) const
{
```

```
    int16  u,v,w;
    real   alpha,cosa,sina;
    real   altitude;
    TMatrix33 m33;

    altitude = fData.fFreedomValue * fData.fStep;
    alpha = fData.fFreedomValue * kRealTwoPI;

    if (fData.fAxis == kRadioX_id)
    {
       u=1;
       v=2;
       w=0;
    }
    else if (fData.fAxis == kRadioY_id)
    {
       u=2;
       v=0;
       w=1;
    }
    else if (fData.fAxis == kRadioZ_id)
    {
       u=0;
       v=1;
       w=2;
    }

    transform.fTranslation[u] = 0.0;
    transform.fTranslation[v] = 0.0;
    transform.fTranslation[w] = altitude;
    sina = RealSin(alpha);
    cosa = RealCos(alpha); //QuickSinCos(alpha,sina,cosa);
    m33[u][u] = cosa; m33[u][v] = sina; m33[u][w]=0.0;
    m33[v][u] = -sina; m33[v][v] = cosa; m33[v][w] = 0.0;
    m33[w][u] = 0.0; m33[w][v] = 0.0; m33[w][w] = 1.0;
    transform.fRotationAndScale = m33;

    return MC_S_OK;
}
```

ScrewConstraint::GetTransformPartialDerivate

```
MCCOMErr ScrewConstraint::GetTransformPartialDerivate(int16 index, TTransform3D&
      transform) const
```

To allow the inverse kinematics mechanism to work with the constraint, implement the **GetTransfromPartialDerivate()** function. A partial derivative transformation is a transformation where each element is derived by a freedom degree. So you have as much Partial Derivative as freedom degrees:

$$\frac{\partial R}{\partial \alpha} = \begin{bmatrix} \frac{\partial i_x}{\partial \alpha} & \frac{\partial j_x}{\partial \alpha} & \frac{\partial k_x}{\partial \alpha} \\ \frac{\partial i_y}{\partial \alpha} & \frac{\partial j_y}{\partial \alpha} & \frac{\partial k_y}{\partial \alpha} \\ \frac{\partial i_z}{\partial \alpha} & \frac{\partial j_z}{\partial \alpha} & \frac{\partial k_z}{\partial \alpha} \end{bmatrix}$$

$$\frac{\partial T}{\partial \alpha} = \begin{bmatrix} \frac{\partial T_x}{\partial \alpha} \\ \frac{\partial T_y}{\partial \alpha} \\ \frac{\partial T_z}{\partial \alpha} \end{bmatrix}$$

```
MCCOMErr ScrewConstraint::GetTransformPartialDerivate(int16 index, TTransform3D&
      transform) const
{
   int16 u,v,w;
   real alpha,cosa,sina;
   TMatrix33 m33;
   if (fData.fAxis == kRadioX_id)
   {
      u = 1;
      v = 2;
      w = 0;
   }
   else if (fData.fAxis == kRadioY_id)
   {
      u = 2;
      v = 0;
      w = 1;
   }
   else if (fData.fAxis == kRadioZ_id)
   {
      u = 0;
      v = 1;
      w = 2;
   }
   if (index == 1)
   {
      transform.fTranslation[u] = 0.0;
      transform.fTranslation[v] = 0.0;
      transform.fTranslation[w] = fData.fStep;
      alpha = kRealTwoPI * fData.fFreedomValue;
      sina  = RealSin(alpha);
      cosa  = RealCos(alpha); //QuickSinCos(alpha,sina,cosa);
      m33[u][u] = -kRealTwoPI * sina;
      m33[u][v] = kRealTwoPI * cosa;
      m33[u][w] = 0.0;
      m33[v][u] = -kRealTwoPI * cosa;
      m33[v][v] = -kRealTwoPI * sina;
      m33[v][w] = 0.0;
      m33[w][u] = 0.0;
      m33[w][v] = 0.0;
      m33[w][w] = 0.0;
      transform.fRotationAndScale = m33;
   }
   return MC_S_OK;
}
```

Writing a Data Component

Family ID : 'data'
Interface ID : IID_I3DExDataComponent
Interface file : I3DExRenderFeature.h
Basic Implementation file: BasicDataComponent.h, BasicDataComponent.cpp

Overview

A data component is a generic object that contains additionnal data for a given type of object (and these data can be animated). Moreover, a data component generally delegates the actions associated with these data to another specific class. For instance, you can create a data component to add a specific effect to all the primitives or to some lights. For that purpose, besides writing the class that performs the effect, you will have to associate it with its data component class, which derives from *TBasicDataComponent* and implements these methods:

- **IsActive(I3DShTreeElement *tree)**
- **GetPostRenderList(TMCArray<IDType> &idArray)**

if your goal is to create a post renderer (see "Writing a Post Render Filter"),

- **GetVolumetricList(TMCArray<IDType>& idArray)**

if your goal is to create a volumetric effect (see "Writing a Volumetric Effect").

IsActive() returns *true* if the data component can be applied to the passed-in parameter.
GetPostRenderList() adds the class ID of your post renderer to the list of post render filters to be applied.
GetVolumetricList() adds the class ID of your volumetric effect to the list of volumetric effects to be applied.

Your data component must specify, in a resource file, which (one or more) types of object it'll be added to, amongst the following:

- Camera
- Group
- Helper Object
- Instance
- Light
- Primitive Instance

Location in the User Interface

If visible in the user interface, a data component is most often accessed through the Effects panel of the Properties tray, in the Assemble room. For instance, if the data component applies to lights, you will have to select a light to be able to see the data component in the panel.

Example: Light Cone (volumetric effect)

Description

In this section, we describe how a data component has been associated to a volumetric effect which can only apply to some lights (this is the Volumetric Effect sample, described in "Writing a Volumetric Effect"). Thus, the volumetric effect (Simple Light Cone) will be accessed through the Effects panel of the Properties tray in the Assemble room when a light is selected. The data component stores all the data needed by the volumetric effect and knows its class ID. Therefore, if you have three spot lights in your scene and you want to activate the simple light cone effect for two of them, then you will have three instantiations of your data

component class but only one volumetric effect instantiation, which will compute the effect for the light sources that asked for it.

Parameters

The following variables are used for the user-input parameters:

```
fIntensity:    Intensity (%)
fRadius:       Light source radius (in)
fFogColor:     Fog effect color
fFallOff:      quality of fall off
```

This is defined in *lcPMAP.h* as follows:

```
struct LightConeInfo
{
    LightConeInfo();

    real         fIntensity;  // Intensity (%).
    real         Radius;      // Light source radius (in).
    TMCColorRGBA fFogColor;   // Fog color
    real         fFallOff;    // quality of fall off
};
```

This is then mapped to the user interface parameters using the PMap resource in *SimpleVolumetricEffect.r*. Note that the parameters within the *LightConeInfo* structure must be in the same order as in the PMap. The PMap resource will be declared as part of the data component, to which we'll add a 'data' resource that specifies the type of object that can use our volumetric effect:

```
resource 'data' (R_SimpleLightConeControlID)
{
    {
    'li  '   // can be applied to lights
    }
};
```

Functions

In this example, we only have two methods to override: **IsActive** and **GetVolumetricList**. The data component class is called *TLightConeControl*.

TLightConeControl::IsActive

```
boolean TLightConeControl::IsActive(I3DShTreeElement *tree)
```

IsActive returns *false* if the component has no effects on a given type of light.

```
boolean TLightConeControl::IsActive(I3DShTreeElement *tree)
{
    return DataComponentUtils::HasLightCone(tree);
}
```

We use a function encapsulated in a structure dedicated to the data component.

```
boolean DataComponentUtils::HasLightCone(I3DShTreeElement *tree)
{
```

The passed-in tree element can be regarded as a light:

```
    TMCCountedPtr<I3DShLightsource> light;
    tree->QueryInterface(IID_I3DShLightsource,(void**)&light);
    MCAssertPointerCheck(light);
```

Then we can ask if this type of light supports the light cone effect:

```
    light->GetLightInfo(hasLightCone,...,...,...);
    return hasLightCone;
}
```

TLightConeControl::GetVolumetricList

```
void TLightConeControl::GetVolumetricList(TMCArray<IDType>& idArray)
```

GetVolumetricList adds the volumetric effect IDs to the array of volumetric effect IDs that we want to apply. Here, we only have to add the class ID of our Volumetric Effect.

```
void TLightConeControl::GetVolumetricList(TMCArray<IDType>& idArray)
{
   if (fPmap.fEnable) idArray.AddElem(SimpleLightConeVolID);
}
```

Writing a Modifier

Family ID : 'modi'
Interface ID : IID_IDExModifier
Interface file : I3dExModifiers.h
Basic Implementation file: BasicModifiers.h, BasicModifiers.h

Overview

Modifiers can be applied to any object or group of objects of the scene, that is any tree except the universe. They can affect the transform of the tree (position and scaling) and the geometry of the objects that belong to the tree.
Depending on what you are doing, you don't need to implement all the functions of the modifiers' API. With **GetModifierFlags**, you tell the shell what kind of modifier you implemented: *behavior* if you only affect the data stored on a tree (in most cases the transform), *deformer* if you only change the geometry. If needed, your modifier can be a behavior and a deformer.
For behaviors, the only required function is **Apply**(I3DShTreeElement* tree) which gives you full control of the tree. For example, changing the position and scaling is easy with the TTreeTransform methods of the tree.
For deformers, you need to implement at least **DeformPoint** (if your deformation depends only on the 3D coordinates of the vertices and is continuous) or **DeformFacetMesh** (if you want more information or need to separate the mesh in pieces). You should implement **SetBoundingBox** if you need to keep the bounding box of the region in which the deformer is applied.
See the reference manual for more information on the *I3DExModifier* methods.
The NonUniformScale sample shows how to create radio buttons and sliders using a PMap and a rsr file. The Explode sample also shows how to create a slider, and demonstrates how to create a checkbox. The Barycenter sample shows how to set up an automatic PMap. The Behavior sample shows how to create a slider and scrollable edit text boxes.

Location in the User Interface

Modifiers are found on the Modifier tab of the Properties Tray. They are applied to the selected object or group. You can have several modifiers at several levels on a tree (e.g. on an object and on the group that contains it). Behaviors are applied starting from the top most group and going down the tree. Deformers are evaluated upward.

Example: Non-Uniform Scale Deformer

Description

The Non-Uniform Scale Deformer scales objects disproportionately around either the X, Y, or Z axis. The two remaining axes are given scaling values, which are labeled the 'U' and 'V' axes. It illustrates the use of DeformPoint.

Parameters

The following variables are used for the user-input parameters:

```
fAxis:          The axis to scale (X, Y, or Z).
fUBegScale:     The Begining scale value for the U Axis.
fUEndScale:     The Ending scale value for the U Axis.
fVBegScale:     The Begining scale value for the V Axis.
```

fVEndScale: **The Ending scale value for the V Axis.**
fBoundingBox: **The object's bounding box.**

These are defined in NonUniformScale.h as follows:

```
struct NonUniformScaleData
{
    int32     fAxis;
    real      fUBegScale;
    real      fUEndScale;
    real      fVBegScale;
    real      fVEndScale;
    TBBox3D   fBoundingBox;
};
```

These are then mapped to the user interface parameters using the PMap resource in NonUniformScale.r. Note that the parameters within the *NonUniformScaleData* struct must be in the same order as in the PMap. Note for advanced users: The **fBoundingBox** field does not appear in the UI. It was added to the PMap to avoid having to write a clone function. Here is what would look like the clone function:

```
void NonUniformScale::Clone(IExDataExchanger** outExchanger,IMCUnknown* pUnk-
       Outer)
{
    TMCCountedCreateHelper<IExDataExchanger>result(outExchanger);
    NonUniformScale * theClone = new NonUniformScale ();
ThrowIfNil(theClone);
    theClone->SetControllingUnknown(pUnkOuter);
    theClone->fBoundingBox= fBoundingBox;
    result= (IExDataExchanger*)theClone;
}
```

Functions

The three *I3DExModifiers* methods that are implemented for this example include: **SetBoundingBox()**, **DeformPoint()**, and **DeformBBox()**. **DeformFacetMesh()** is not needed for this example as only the coordinates of each point are changed .

NonUniformScale::SetBoundingBox

```
MCCOMErr NonUniformScale::SetBoundingBox(TBBox3D* bbox)
```

SetBBox() assigns the variable **fBoundingBox** to the current bounding box from the shell:

```
MCCOMErr NonUniformScale::SetBountingBox(TBox3D* bbox)
{
    fData.fBoundingBox=*bbox;
}
```

NonUniformScale::DeformBBox

```
MCCOMErr NonUniformScale::DeformBBox(const TBBox3D &in,TBBox3D &out)
```

DeformBBox evaluates how the deformation will transform the bounding box *in*. It uses **DeformPoint** to compute the result and puts it in *out*.

```
MCCOMErr NonUniformScale::DeformBBox(const TBox3D &in,TBox3D &out)
{

    TVector3  TempPoint[2];

    TVector3  point, deformed;

    TempPoint[0] = in.fMin;
    TempPoint[1] = in.fMax;
```

```
    int XX, YY, ZZ, checkbound;  //Loop Controls

    out=in;

    for (XX=1; XX>=0; XX--)
    {
       point[0]=TempPoint[XX][0];
       for (YY=1; YY>=0; YY--)
       {
          point[1]=TempPoint[YY][1];
          for (ZZ=1; ZZ>=0; ZZ--)
          {
             point[2]=TempPoint[ZZ][2];
             DeformPoint(&point, &deformed);
             for (checkbound=2; checkbound>=0; checkbound--)
             {
                if (deformed[checkbound] < out.fMin[checkbound])
                   out.fMin[checkbound] = deformed[checkbound];
                if (deformed[checkbound] > out.fMax[checkbound])
                   out.fMax[checkbound] = deformed[checkbound];
             }
          }
       }
    }
    return MC_S_OK;
}
```

NonUniformScale::DeformPoint

```
MCCOMErr NonUniformScale::DeformPoint(TVector3& point, TVector3& result)
```

DeformPoint() is where the deformation takes place. First, *u* and *v* are assigned depending on the value of **`fAxis`**. *w* is assigned to the scale axis. Then a linear scale is calculated between the beginning and ending of the bounding box.
The scaling is performed for both the *u* and *v* axis using the following formula:

$$Result = P(B + wrelative(E - B))$$

Where P is the variable *Point*, B is the Beginning Scale, and E is the Ending Scale. The relative scale, *wrelative*, is calcuated by this formula:

$$wrelative = P_w - \left(\frac{Bmin}{Bmax - Bmin}\right)$$

Where *Bmin* is the bounding box minimum, and *Bmax* is the bounding box maximum.

```
MCCOMErr NonUniformScale::DeformPoint(TVector3& point, TVector3& result)
{
    int16     u, v, w;
    real      wrelative;

    switch (fData.fAxis)
    {
    case kRadioX_id:
       u=1; //Y Axis
       v=2; //Z Axis
       w=0; //X Axis
       break;
    case kRadioY_id:
       u=2; //Z Axis
       v=0; //X Axis
       w=1; //Y Axis
```

```
        break;
    case kRadioZ_id:
        u=0; //X Axis
        v=1; //Y Axis
        w=2; //Z Axis
        break;
    default:
        break;
    }
    if ((fData.fBoundingBox.fMax[w]-fData.fBoundingBox.fMin[w])==0.0)
    {
        *result = *point;
        return MC_S_OK;
    }
    wrelative=((*point)[w] - fData.fBoundingBox.fMin[w])/(fData.fBounding-
        Box.fMax[w]-fData.fBoundingBox.fMin[w]);
    if ((fData.fUEndScale+fData.fUBegScale)==0.0)
    {
        (*result)[u]=0.0;
    }
    else
    {
        (*result)[u] = (*point)[u] * (fData.fUBegScale + wrelative * (fData.fUEnd-
        Scale-fData.fUBegScale));
    }
    if ((fData.fVEndScale+fData.fVBegScale)==0.0)
    {
        (*result)[v] = 0.0;
    }
    else
    {
        (*result)[v] = (*point)[v] * (fData.fVBegScale + wrelative * (fData.fVEnd-
        Scale-fData.fVBegScale));
    }
    (*result)[w] = (*point)[w];

    return MC_S_OK;
```

In addition to inheriting from **TBasicDeformModifier**, **NonUniformScale** also inherites from **TBasicWireframe** in order to track scaling from within the 3D window. Multiple-inheritance requires a custom implementation of **AddRef** and **QueryInterface**. See the code sample for specific information.

Example: Explode

Description

The Explode modifier breaks an object into pieces which then fall influenced by gravity. You can specify the size of the pieces, the speed of the explosion, and gravity as a percentage.
Some of the functions are implemented similarly as in **NonUniformScale**. You may want to look at the source code to see how they are implemented.

Parameters

The following variables are used for the user-input parameters:

```
fMinSize:       The minimum size of a facet when an object explodes.
fExplosionFactor: The factor that sets the size of explosion.
fFloor:           The floor of the explosion area.
fGravityCoef:     The gravity coefficient.
```

```
bool              fApplyGravity;
fBoundingBox:     The object's bounding box.
```

These are defined in Explode.h as follows:

```
struct ExplodeData {

  real    fMinSize;
  real    fExplosionFactor;
  real    fFloor;
  real    fGravityCoef;
  bool    fApplyGravity;
  TBBox3D fBoundingBox;  // Bounding Box
};
```

These are then mapped to the user interface parameters using the PMap resource in Explode.r. Note that the parameters within the *ExplodeData* struct must be in the same order as in the PMap.

Functions

The *I3DExModifiers* methods that are implemented for this example include: **SetBoundingBox()**, **DeformPoint()**, **DeformBBox()** and **DeformFacetMesh()**. When functions are implemented similarly to the NonUniformScale sample, they are not documented here.

Explode::DeformPoint

```
MCCOMErr Explode::DeformPoint(TVector3& point, TVector3& result)
```

DeformPoint() is called by **DeformBBox()**. It specifies whether the gravity coefficient or the floor will affect the explosion.

```
MCCOMErr Explode::DeformPoint(TVector3& point, TVector3& result) {

  TVector3 Translate;
  int i;

  Translate[0] = (*point)[0] * fData.fExplosionFactor;
  Translate[1] = (*point)[1] * fData.fExplosionFactor;
  if (fData.fGravityCoef == 0)
    Translate[2] = (*point)[2] * fData.fExplosionFactor;
  else {
    real a = 9.8f;
    if ((*point)[2] <= fData.fFloor)
      Translate[2] = 0.0;
    else
      Translate[2] = (*point)[2] * fData.fExplosionFactor - fData.fGravityCoef *
       (fData.fExplosionFactor * fData.fExplosionFactor);
   if ((*point)[2] + Translate[2] < fData.fFloor)
     Translate[2] = fData.fFloor - (*point)[2];
  }
  for (i=0; i<3; i++) {
   (*result)[i] = (*point)[i] + Translate[i];
  }

  return MC_S_OK;
}
```

Explode::DeformBBox

```
MCCOMErr Explode::DeformBBox(const TBBox3D &in,TBBox3D &out)
```

DeformBBox evaluates how the deformation will transform the bounding box *in*. It uses **DeformPoint** to compute the result and puts it in *out*.

```
MCCOMErr Explode::DeformBBox(const TBBox3D &in, TBBox3D &out) {
```

```
    DeformPoint(in.fMin, out.fMin);
    DeformPoint(in.fMax, out.fMax);
    return MC_S_OK;
}
```

Explode::DeformFacetMesh

```
MCCOMErr Explode::DeformFacetMesh(real lod,FacetMesh* in, FacetMesh** outMesh) {
```

DeformFacetMesh() subdivides the facets into smaller pieces, deforms them and recreates a new mesh with the deformed facets.

```
MCCOMErr Explode::DeformFacetMesh(real lod,FacetMesh* in, FacetMesh** outMesh) {

    FacetMeshAccumulator accu;
    FacetMeshFacetIterator iter;
    iter.Initialize(in);
    for (iter.First(); iter.More(); iter.Next()) {
        explodeFacet(&iter.GetFacet(), &accu);
    }
    accu.MakeFacetMesh(outMesh);
  return MC_S_OK;
}
```

Example: Barycenter

Description

The Barycenter shows an example of what you can do in the **Apply** function. It also demonstrates the use of data components (see “Writing a Data Component”).

Parameters

The following shows how to set up a data component in a PMap.

```
resource 'PMap' (129)
{
    {  // TODO: Add your parameters by providing :
       //   four letters ID,
       //   four letters type ID,
       //   interpolate or zero,
       //   name
        'WIEG','re32', interpolate, "Weight!","",
    }
};
resource 'data' (129)
{
    {
    'prim'   // can be applied to primitives
    }
};
```

Functions

The following *I3DExTreeElement* method is implemented for this example:

ExBarycenter::Apply

```
    virtual MCCOMErrMCCOMAPI Apply(I3DShTreeElement* tree);
```

Apply() applies the modifier to the selected object according to the other objects that exist in the universe (also known as the tree). For example, if you were aligning objects, Apply() would get all of the objects to perform the alignment.

```
MCCOMErr ExBarycenter::Apply(I3DShTreeElement* tree)
{
   // check for all the instances to find a misc feature
   TMCCountedPtr<I3DShScene> ascene=NULL;
   TMCCountedPtr<I3DShGroup> agroup;
   TMCCountedPtr<I3DShTreeElement> atree=NULL;
   TMCCountedPtr<I3DShTreeElement> atreeBis=NULL;
   TMCCountedPtr<I3DShDataComponent> agendata=NULL;
   TMCCountedPtr<Weight> weightGenData=NULL;

   const TRenderableAndTfmArray* instances;
   real totalWeight=0;
   TVector3 sumPos; sumPos[0]=0; sumPos[1]=0; sumPos[2]=0;
   int jj;
   unsigned long nbMisc;

   tree->GetScene(&ascene);
   ascene->GetTreeRoot(&agroup);
   agroup->QueryInterface(IID_I3DShTreeElement,(void**)&atree);
   ThrowIfNil(atree);

   atree->BeginGetRenderables(instances);
   TRenderableAndTfmArray::const_iteratoriter = instances->Begin();

   for (const TRenderableAndTfm* curInstance = iter.First(); iter.More(); curIn-
      stance = iter.Next())
   {
      curInstance->fInstance->QueryInterface(IID_I3DShTreeElement,
      (void**)&atreeBis);
      /*if (atreeBis == tree)
      {
         TRenderableFlags renderableFlags = curInstance->fRenderable->GetRen-
      derableFlags();
         renderableFlags.SetMaskedValue(TRenderableFlags::kStaticMask,false);
      }
      */
      if (atreeBis != tree)
      {
         nbMisc=atreeBis->GetDataComponentsCount();
         for (jj=nbMisc; jj>0; jj--)
         {
            // loop on all the misc features
            atreeBis->GetDataComponentByIndex(&agendata,jj-1);
            if (agendata->QueryInterface(CLSID_ExWeight, (void**)&weightGen-
      Data)==MC_S_OK)
            {
               real weight=weightGenData->GetWeight();
               totalWeight += weight;
               sumPos[0]+=curInstance->fT.fTranslation[0] * weight;
               sumPos[1]+=curInstance->fT.fTranslation[1] * weight;
               sumPos[2]+=curInstance->fT.fTranslation[2] * weight;
            }
         }
      }
   }

   if (totalWeight!=0)
```

```
    {
        TTransform3D tr;
        // Set the new position of tree
        tree->GetGlobalTransform3D(tr);
        tr.fTranslation[0] = sumPos[0] / totalWeight;
        tr.fTranslation[1] = sumPos[1] / totalWeight;
        tr.fTranslation[2] = sumPos[2] / totalWeight;
        tree->SetGlobalTransform3D(tr);
    }

    atree->EndGetRenderables();

    return MC_S_OK;
}
```

Example: Behavior

Description

The Behavior modifier demonstrates how to align the selected object with a relative position to two other objects. You clone an object with the **Clone()** function, calculate the relative position between the objects, then apply the change to the tree structure with the **Apply()** function.
Some of the functions are implemented similarly as the other samples. You may want to look at the source code to see how they are implemented.

Parameters

The following variables are used for the user-input parameters:

fNameObject1: **The name of the first object to which to align.**
fNameObject2: **The name of the second object to which to align.**
fRelPos: **The relative position between the two objects.**

These are defined in Behavior.h as follows:

```
struct BehaviorData
{
  TMCString255fNameObject1;// name of the first object to align to
  TMCString255fNameObject2;// name of the second object to align to
  real32fRelPos;// relative position between the two objects (0 first, 1 second)
};
```

These are then mapped to the user interface parameters using the PMap resource in Explode.r. Note that the parameters within the *BehaviorData* struct must be in the same order as in the PMap.

Functions

The *I3DExModifier* methods that are implemented for this example include: **Clone()**, **HandleEvent()**, and **Apply()**. When functions are implemented similarly to the earlier samples, they are not documented here.

Behavior::Clone

```
void Behavior::Clone(IExDataExchanger** res,IMCUnknown* pUnkOuter)
```

Clone() clones the object in memory before the modifier is applied in case of Undo. This allows the modifier to easily be "undone".

```
void Behavior::Clone(IExDataExchanger** res,IMCUnknown* pUnkOuter)
{
    TMCCountedCreateHelper<IExDataExchanger> result(res);
```

```
        Behavior* clone = new Behavior();
        result = (IExDataExchanger*)clone;

        if (clone)
            clone->fData=fData; // copy the BehaviorData
        clone->SetControllingUnknown(pUnkOuter);
    }
```

Writing a 3DExporter

Family ID : '3Dou'
Interface ID : IID_I3DExExportFilter
Interface file : I3DExImportExport.h
Basic Implementation file: Basic3DImportExport.h, Basic3DImportExport.cpp

Overview

An exporter allows you to export files from Carrara into a different file format. Generally speaking, writing an exporter is fairly easy.
The example code contains details that apply to the DXF file format. Refer to the source code for more information. In this example, notice the heavy use of the QueryInterface() call. Make sure you are familiar with the I3DShObject, I3DShTreeElement and I3DShInstance interfaces.
Make sure you build the right 'Cmpp' (Component Private) resource. It contains the information necessary to the 3D Shell to display your file format extension and name in the Save As dialog. Please refer to , "The Component Private resources ("Cmpp")" for all details.
Among all the methods of the I3DExExportFilter interface, you must at least implement DoExport. It is used to export the scene data. See the reference manual for more specific information on the other methods.

Location in the User Interface

An exporter is accessed either from the File menu> Export command or from the File menu> Save As command. If you return true in WantsOptionDialog(), your exporter's options will be shown in a dialog before saving the file.

Example: DXF

Description

This example shows how to export Carrara scene data into the DXF file format. In this example, light and camera data are not exported.
In this exporter, the basic algorithm is:

```
Create the output file                          Instanciate a TMCfstream
For each object instance in the scene:          I3DShScene::GetInstanceByIndex()
  Get the 3D object referenced by the instance I3DShInstance::WriteInstance
    If it is a Primitive                        QueryInterface(IID_I3DShPrimitive)
      Get its global Transformation             I3DShTreeElement::GetGlobalTreeTrans-
                  form()
      Get the Primitive's geometry as facets
        Transform each facet in Global coordinates
        Write the facet to the output file
```

Parameters

No parameters are defined for user input.

TDXFExporter::DoExport

```
MCCOMErr TDXFExporter::DoExport(IMCFile * file, IMCUnknown* elem, IMCUnknown*
      subElem)
```

For 3D exporters, *elem* maps to I3DShScene and *subElem* maps to I3DShTreeElement through QueryInterface.
DoExport() gets the number of instances in a scene, making sure that each element is a type that can be exported into the target format. It then writes out the header information, the data for each object and the object's facet data when appropriate. Then DoExport() writes the data for the end of the DXF file.

```
MCCOMErr TDXFExporter::DoExport(IMCFile* file, IMCUnknown* elem, IMCUnknown*
      subElem)
{
   TMCfstream* stream = NULL;
   uint32 nbInst = 0;
   TMCCountedPtr<I3DShScene> scene;
   MCCOMErr error;

   error = elem->QueryInterface(IID_I3DShScene,(void**)&scene);
   if (error != MC_S_OK)
      return error;
   else if (scene == NULL)
      return MC_E_FAIL;

   TMCCountedPtr<IMCUnknown> progressKey;
   try
   {
      nbInst =scene->GetInstanceListCount();

      // Progress bar
      TMCString255 progressMsg;
      void* oldResources = NULL;
      gResourceUtilities->SetupComponentResources('3Din', 'COEA', &old-
      Resources);
      gResourceUtilities->GetIndString(progressMsg, 131, 1);// string  1 of
      STR# 131 (see DXF.r)
      gResourceUtilities->RestoreComponentResources(oldResources);

      gShellUtilities->BeginProgress(progressMsg, &progressKey, nbInst);

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

      stream->SetInfo(gShellUtilities->GetCreator(), IDTYPE('T', 'E', 'X',
      'T'));

      WriteDXFBegin(stream);

      TMCCountedPtr<I3DShInstance> instance;
      for (uint32 i=0; i<nbInst; ++i)
      {
         scene->GetInstanceByIndex(&instance,i);
         WriteInstance(stream, instance, i);
         gShellUtilities->IncrementProgress(1, progressKey);
      }

      WriteDXFEnd(stream);

      delete stream;
      gShellUtilities->EndProgress(progressKey);
   }
   catch (...)
```

```
        {
            delete stream;
            gShellUtilities->EndProgress(progressKey);
            throw;
        }
        return MC_S_OK;
    }
```

Writing a Final Renderer

Family ID : 'frnd'
Interface ID : IID_I3DExFinalRenderer
Interface file : I3DExFinalRenderer.h
Basic Implementation file: BasicFinalRenderer.h, BasicFinalRenderer.cpp

Overview

A renderer controls the way you output your scene data into an image or movie format. The renderer interface allows you to create a new external renderer for Carrara.
The renderer module needs interfaces to the following required elements from the Carrara shell : the camera, the tree top (usually the universe), the environment (atmosphere, backdrop & background), the ambient light, and the offscreen area (which is used as a zone in which to draw). Since it may have parameters defined by the user in the scene settings, the renderer is related to I3DExDataExchanger and may use a PMap.

There are two final renderer samples:

- A simple wireframe renderer with no special features, a minimum implementation
- A mix renderer that includes a ZBuffer, an environment, a Phong reflection model, an interface to G-buffers, and support for tiled offscreen. In that example, more of the renderer interface is used.

I3DExFinalRenderer provides following methods for both samples:

- **SetTreeTop(I3DShGroup* treeTop);**
- **SetCamera(I3DShCamera* camera);**
- **SetEnvironment(I3DShEnvironment* environment);**
- **SetAmbientLight(const TMCColorRGB &ambiantColor);**
- **SetFieldRenderingData(int32 fieldRenderingSettings,int16 firstFrame) ;**
- **GetRenderStatistics(I3DRenderStatistics** renderstats) ;**
- **PrepareDraw(const TMCPoint &size,const TBBox2D& uvBox,const TBBox2D& productionFrame, boolean (*callback) (int16 vv, void* priv),void* priv);**
- **FinishDraw();**
- **RenderVolumetrics(boolean renderEffects);**
- **RenderBackGround(boolean enable);**
- **OptimizeTileSize(TMCPoint& inOutTileSize);**
- **GetTileRenderer(I3DExTileRenderer** tileRenderer);**

WireRenderer implements the following methods:
- RenderScene(const TMCRect* area)
- BlocCopy(RTData* pixels)
- Rasterize(TFacet3D facet, TTransform3D L2C, I3DShInstance* CurrentInstance)
- PixelSet (int32 x, int32 y, TMCColorRGBA* color)
- DrawLine(int16 x1, int16 y1, int16 x2, int16 y2, TMCColorRGBA* Color)

MixRenderer implements the following additional methods:
- MixRenderer::InitZbuffer()
- AllocMem()
- ReleaseMem()
- GetLights()
- FreeLights()
- RenderScene(const TMCRect* area)
- ZBufferSet (int32 x, int32 y, real val)

- Interpolate(float A, float B, long y1, long y2, long dy, float *array)
- FillPoly(TFacet3D facet, I3DShInstance *CurrentInstance, ShadingIn *myShadingIn, ShadingOut *myShadingOut, int numVertices)
- BackgroundColor(TVector2 theScreenUV, TMCColorRGBA& resultColor)

See the reference manual for more specific information on the interface methods.

Location in the User Interface

A renderer is accessed from the Properties tray after clicking the Render button to switch to the Render room.

Example: WireRenderer

Description

This example shows how to create a renderer with a minimum of features. However, the following items are needed:

- a view resource for the user to choose a renderer
- an interface to the camera (absolutely necessary to change between the different coordinate systems)
- an interface to the image area (the full image to be rendered)
- an interface to the offscreen (an area into which the renderer draws, in this example it has the size of the image area)
- an interface to the top of the tree hierarchy (to get all the objects to be drawn).

To create this renderer, you'll need to do the following:

- Tell the shell to render the image in one single pass (no tiling)
- Transform the tree into a sequence of instances in which we get the objects (each instance has a transformation matrix from local coordinates to global coordinates associated)
- From the camera, get the transformation matrix from global coordinates system to the camera coordinates system
- Create a facet iterator to get each mesh facet in local coordinates
- Transform the vertices' local coordinates in screen coordinates then project them in screen space
- Draw lines between the points

Parameters

In this example, there are no user-input parameters, but the following structures are defined in WireRenderer.h:
This structure is used to draw the image.

```
struct FlatBuffer
{
   uint16* r;
   uint16* g;
   uint16* b;
};
```

This structure is used during rendering

```
struct InstanceAndTransform// This structure will be useful during rendering
{
   I3DShInstance*fInstance;
```

```
    TTransform3DfT;
};
```

If there were user-input parameters, they would need to be mapped to the user interface parameters using a PMap resource in WireRenderer.r.
Note that the parameters within the *WireRendererData* struct must be in the same order as in the PMap.

Functions

WireRenderer::SetTreeTop

```
MCCOMErr WireRenderer::SetTreeTop(I3DShGroup* treeTop)
```

The **SetTreeTop()** function specifies which tree needs to be rendered (generally the universe). A scene has two parts: a hierarchical tree and an objects list. The Scene Tree defines the positions and the relationship between Tree Elements. Basically, for each level in the tree, a transformation is defined, thus by composing transformations from the root to the considered object, you get its position in the Global System.

```
MCCOMErr WireRenderer::SetTreeTop(I3DShGroup* treeTop)
{
    fTreeTop = treeTop;
    return MC_S_OK;
}
```

WireRenderer::SetCamera

```
MCCOMErr WireRenderer::SetCamera (I3DShCamera* camera)
```

SetCamera() specifies the rendering camera. The camera is also used to change between the different coordinate systems (local and global).

```
MCCOMErr WireRenderer::SetCamera (I3DShCamera* camera)
{
    fCamera = camera;
    return MC_S_OK;
}
```

WireRenderer::SetEnvironment

```
MCCOMErr WireRenderer::SetEnvironment (I3DShEnvironment* environment)
```

SetEnvironment() sets up the background, backdrop, and atmospheric shader.

```
MCCOMErr WireRenderer::SetEnvironment (I3DShEnvironment* environment)
{
    if (!environment || environment->HasBackground()) fBackground=environment;
    if (!environment || environment->HasBackdrop()) fBackdrop=environment;
    if (!environment || environment->HasAtmosphere()) fAtmosphere=environment;
    return MC_S_OK;
}
```

WireRenderer::SetAmbientLight

```
MCCOMErr WireRenderer::SetAmbientLight (const TMCColorRGB &ambientColor)
```

SetAmbientLight() sets the ambient light color.

```
MCCOMErr WireRenderer::SetAmbientLight (const TMCColorRGB &ambientColor)
{
    fAmbientColor = ambientColor;
    return MC_S_OK;
}
```

WireRenderer::SetFieldRenderingData

```
MCCOMErr WireRenderer::SetFieldRenderingData (boolean useFieldRendering,int16
      firstFrame)
```

SetFieldRenderingData() specifies multiple frames for video compositing and updates an option found in Scene Settings under the Renderer panel in the Properties tray.

```
MCCOMErr WireRenderer::SetFieldRenderingData (boolean useFieldRendering,int16
      firstFrame)
{
   return MC_S_OK;
}


void WireRenderer::GetRenderStatistics(I3DRenderStatistics**renderstats)
{
}
```

WireRenderer::RenderScene

```
void WireRenderer::RenderScene(const TMCRect* area)
```

RenderScene() is called by **PrepareDraw()**. It allows us to draw the image in the buffer.

```
void WireRenderer::RenderScene(const TMCRect* area)
{
   TTransform3D G2C ;//Transformation from Global to Camera
   TTransform3D L2C ;//Transformation from Local to Camera
   TTransform3D L2G ;//Transformation from Local to Global

   TMCCountedPtr<FacetMesh>amesh ;// Facet Mesh data
   TMCColorRGBA color;

   fBuffer.r = new uint16[fImageHeight*fImageWidth];
   fBuffer.g = new uint16[fImageHeight*fImageWidth];
   fBuffer.b = new uint16[fImageHeight*fImageWidth];

   // clean the buffer
   for (int32 i=0; i<fImageHeight*fImageWidth; i++)
   {
      fBuffer.r[i]=0;
      fBuffer.g[i]=5000;
      fBuffer.b[i]=5000;
   }

   fCamera->GetGlobalToCameraTransform(&G2C) ; //We get transformation from Glo-
      bal to Camera,
                                          //actually in Screen Coord

   FacetMeshFacetIterator facetIterator;

   int32     theKind;//Used to stock the kind of an instance
   theKind = 0;

   TRenderableAndTfmArray::const_iteratoriter = fInstances->Begin();
   for (const TRenderableAndTfm* current = iter.First(); iter.More(); current =
      iter.Next())
   {
      theKind = current->fInstance->GetInstanceKind();
      if((theKind == 3) || (theKind == 4))
         continue;

      L2G = current->fT ;//In InstanceAndTransform fT is not a Translation as we
      may imagine
                          //but a AFFINETRANSFORM cad Translation + Rotation
```

```
                                    //we get transformation from Local to Global system

            //Then we get our complete transformation from local to Camera Coordinate
            L2C = G2C*L2G;

            current->fInstance->GetFMesh(0.0, &amesh) ;//get a pointer to IFacetMesh
            interface

            facetIterator.Initialize( amesh ) ;//Initialization for a using of a facet
            iterator

            TFacet3D facet;

            for ( facetIterator.First(); facetIterator.More(); facetIterator.Next())
            { // browsing into facets list
                facet = facetIterator.GetFacet();
                Rasterize( facet, L2C, current->fInstance );
            }
        }
    }
```

WireRenderer::PrepareDraw

```
MCCOMErr WireRenderer::PrepareDraw (const TMCPoint& size,const TBBox2D&
        uvBox,const TBBox2D& productionFrame, boolean (*callback) (int16 vv, void
        *priv),void* priv)
```

PrepareDraw() is called just at the beginning of the rendering of each frame. The argument *size* defines the size of the image to render, and *uvBox* specifies the screen coordinates of the image. (Note that it can be outside the production frame.) Note that productionFrame specifies the screen coordinates of the production frame.

```
MCCOMErr WireRenderer::PrepareDraw (const TMCPoint& size,const TBBox2D&
        uvBox,const TBBox2D& productionFrame, boolean (*callback) (int16 vv, void
        *priv),void* priv)
{
    fImageArea = TMCRect(0,0,size.x,size.y);

    fImageWidth = size.x;//We get the dimensions of
    fImageHeight = size.y;//the image to be drawn
    fDepth = 8;                          // For compatibility with Put32

    fOffscrOffset[0] = fImageWidth/2;
    fOffscrOffset[1] = fImageHeight/2;

    fZoom[0] = fImageWidth/2;
    fZoom[1] = fImageHeight/2;

//  fUVMinMax.fMin[0]=(totalarea.left-fOffscrOffset[0])/fZoom[0];
//  fUVMinMax.fMax[0]=(totalarea.right-fOffscrOffset[0])/fZoom[0];
//  fUVMinMax.fMin[1]=-(totalarea.bottom-fOffscrOffset[1])/fZoom[1];
//  fUVMinMax.fMax[1]=-(totalarea.top-fOffscrOffset[1])/fZoom[1];

    fTreeTop->QueryInterface(IID_I3DShTreeElement,(void**)&fTree);
    ThrowIfNil(fTree);

    fTree->BeginGetRenderables(fInstances);

    RenderScene(&fImageArea);

    return MC_S_OK;
}
```

WireRenderer::FinishDraw

```
MCCOMErr WireRenderer::FinishDraw ()
```

FinishDraw() is called when the drawing is finished; it deletes temporary components.

```
MCCOMErr WireRenderer::FinishDraw ()
{
    fTree->EndGetRenderables();
    fTree=NULL;

    delete fInstances;
    fInstances=NULL;

    delete fBuffer.r;
    delete fBuffer.g;
    delete fBuffer.b;

    return MC_S_OK;
}
```

WireRenderer::BlocCopy

```
void WireRenderer::BlocCopy(RTData* pixels)
```

BlocCopy() is called by the DrawRect call. It copies a tile of the buffer to the RTData element pixels.

```
void WireRenderer::BlocCopy(RTData* pixels)
{
    int i,j;

    int32 height=fDrawingArea.bottom-fDrawingArea.top;
    int32 width=fDrawingArea.right-fDrawingArea.left;

    for (j=0; j<height; j++)
    {
        for (i=0; i<width; i++)
        {
            pixels->r[i+j*width] = fBuffer.r[i+fDrawingArea.left + (j+fDrawin-
    gArea.top)*fImageWidth];
            pixels->g[i+j*width] = fBuffer.g[i+fDrawingArea.left + (j+fDrawin-
    gArea.top)*fImageWidth];
            pixels->b[i+j*width] = fBuffer.b[i+fDrawingArea.left + (j+fDrawin-
    gArea.top)*fImageWidth];
        }
    }
}
```

WireRenderer::GetTileRenderer

GetTileRenderer() is called once for each rendering thread if your renderer supports multiple rendering threads (ie returns true in **IsSMPAware**). In the case of the WireRenderer, we return the renderer itself (this) because we do not support multi-threading. Otherwise we would need to create a new object that would implement the I3DExTileRenderer interface each time this function is called.

```
void WireRenderer::GetTileRenderer(I3DExTileRenderer** tileRenderer);
{
    TMCCountedGetHelper<I3DExTileRenderer> result(tileRenderer);
    result = this;
}
```

WireRenderer::Init

Init is called at the beginning of the rendering of each tile. It is given the rectangle of the tile in pixels (rect) inside the image as well as the coordinates of the tile in screen coordinates (uvBox)

```
void WireRenderer::Init(const TMCRect& rect,const TBBox2D& uvBox)
{
    fCurrentTileRect = rect;
    fCurrentUVBox = uvBox;
}
```

WireRenderer::GetNextSubTile

GetNextSubTile is called by the shell to determine if there is another sub tile to render. Here we return false because we do not support sub tiles. (using sub tiles allows you to update the image as the rendering is being performed).

```
boolean WireRenderer::GetNextSubTile(TMCRect& rect)
{
    return false;
}
```

WireRenderer::RenderSubTile

RenderSubTile is called after **GetNextSubTile** if it did not return false. In this case, it never happens.

WireRenderer::FinishRender

FinishRender is called at the end of the rendering of a tile. You can use it to perform a final pass on the pixels of the tile. If you used sub tiles, you should have already filled the color buffer but you might want to use this call to fill the G-Buffers since they do not need to be updated for the UI.

In our example, we are not using the sub tiles so this is were we fill the color buffer. The color buffer for this tile is defined in the struct RTData. We simply call BlocCopy to copy the pixel color of the image into the final image.

```
void WireRenderer::FinishRender(const RTData& pixels)
{
    fDrawingArea = fCurrentTileRect;
    BlocCopy(&pixels);
}
```

WireRenderer::Rasterize

```
void WireRenderer::Rasterize(TFacet3D facet, TTransform3D L2C, I3DShInstance*
       CurrentInstance )
```

Rasterize() draws the facets in the buffer, transforming the coordinates from Local to Camera. To learn more about the toolbox mentioned below, refer to Toolbox.h in the WireRenderer directory.

```
void WireRenderer::Rasterize(TFacet3D facet, TTransform3D L2C, I3DShInstance*
       CurrentInstance )
{
    //We transform Vertex in Local to Vertex in Camera Coord using L2C
    TVector3 vertex1, vertex2, vertex3;

    WireTransformPoint(vertex1, L2C, facet.fVertices[0].fVertex);
// This is a call from the toolbox we have created
    WireTransformPoint(vertex2, L2C, facet.fVertices[1].fVertex);
    WireTransformPoint(vertex3, L2C, facet.fVertices[2].fVertex);

    TMCColorRGBAcolor, notused;
```

```
    CurrentInstance->GetMainColors(color, notused);

    TVector2 screenVertex1;
    TVector2screenVertex2;
    TVector2screenVertex3;

    real   zOut ;

    //We transform Vertex in Screen Coord in Vertex in Screen Space, from 3D to 2D
    boolean result1 = fCamera->Project3DTo2D(&vertex1, &screenVertex1, &zOut);
    boolean result2 = fCamera->Project3DTo2D(&vertex2, &screenVertex2, &zOut);
    boolean result3 = fCamera->Project3DTo2D(&vertex3, &screenVertex3, &zOut);

    //Before displaying each pixel, we have to apply Zoom transformation, and to
       center
    screenVertex1[0] = fOffscrOffset[0] + (screenVertex1[0] * fZoom[0]) ;
    screenVertex1[1] = fOffscrOffset[1] - (screenVertex1[1] * fZoom[1]) ;

    screenVertex2[0] = fOffscrOffset[0] + (screenVertex2[0] * fZoom[0]) ;
    screenVertex2[1] = fOffscrOffset[1] - (screenVertex2[1] * fZoom[1]) ;

    screenVertex3[0] = fOffscrOffset[0] + (screenVertex3[0] * fZoom[0]) ;
    screenVertex3[1] = fOffscrOffset[1] - (screenVertex3[1] * fZoom[1]) ;

    DrawLine((int16)screenVertex1[0], (int16)screenVertex1[1],
       (int16)screenVertex2[0], (int16)screenVertex2[1], &color ) ;
    DrawLine((int16)screenVertex2[0], (int16)screenVertex2[1],
       (int16)screenVertex3[0], (int16)screenVertex3[1], &color ) ;
    DrawLine((int16)screenVertex3[0], (int16)screenVertex3[1],
       (int16)screenVertex1[0], (int16)screenVertex1[1], &color ) ;

}
```

WireRenderer::PixelSet

```
void WireRenderer::PixelSet (int32 x, int32 y, TMCColorRGBA* color)
```

PixelSet() allows you to change the value of a pixel, and is called by **DrawLine()**.

```
void WireRenderer::PixelSet (int32 x, int32 y, TMCColorRGBA* color)
{
    if ((x>0)&&(x<fImageWidth)&&(y>0)&&(y<fImageHeight))
    {
        fBuffer.r[x + fImageWidth*y] = color->R;
        fBuffer.g[x + fImageWidth*y] = color->G;
        fBuffer.b[x + fImageWidth*y] = color->B;
    }
}
```

WireRenderer::DrawLine

```
void WireRenderer::DrawLine(int16 x1, int16 y1, int16 x2, int16 y2, TMCColorRGBA*
       Color )
```

DrawLine() draws a line using the Bresenham Drawliner algorithm.

```
void WireRenderer::DrawLine(int16 x1, int16 y1, int16 x2, int16 y2, TMCColorRGBA*
       Color )
{
{
    int16x,y,xi,yi,i,c,dx,dy;

    x=x1;
```

```
    y=y1;

    PixelSet(x, y, Color);

    if (x1 < x2)
       xi=1;
    else
       xi=-1;

    if (y1 < y2)
       yi=1;
    else
       yi=-1;

    dx=abs(x1-x2);
    dy=abs(y1-y2);

    if (dy > dx)
    {
       c=dy >> 1;

       for (i=1;i<dy;i++)
       {
          y=y+yi;
          c=c+dx;

          if (c >= dy)
          {
             c=c-dy;
             x=x+xi;
          }
          PixelSet(x, y, Color);
       }
    }

    else
    {
       c=dx >> 1;
       for (i=1;i<dx;i++)
       {
          x=x+xi;
          c=c+dy;
          if (c >= dx)
          {
             c=c-dx;
             y=y+yi;
          }
          PixelSet(x, y, Color);
       }
     }
}
```

Example: MixRenderer

Description

This example shows how to create a renderer with additional features.

These features include: a ZBuffer, an environment, a Phong reflection model, an interface to G-buffers, and support for tiled offscreen. In addition, more of the renderer interface is used.

Parameters

In this example, there are no user-input parameters, but the following structures are defined in MixRenderer.h:
This structure is used to draw the image:

```
struct FlatBuffer
{
   uint16* r;
   uint16* g;
   uint16* b;
};
```

This structure creates an automatic PMap:

```
struct RendParams
{  //Create a pMap-like structure
   long frontier;
};
```

This structure is used during rendering:

```
struct InstanceAndTransform // This structure will be useful during rendering
{
   I3DShInstance* fInstance;
   TTransform3D fT;
};
```

If there were user-input parameters, they would need to be mapped to the user interface parameters using a PMap resource in MixRenderer.r.
Note that the parameters within the *RendParams* struct must be in the same order as in the PMap.

Functions

Many of the functions in the MixRenderer sample need to be set up similar to the WireRenderer sample.

MixRenderer::InitZbuffer

```
void MixRenderer::InitZbuffer()
```

The **InitZBuffer()** function initializes the ZBuffer.

```
void MixRenderer::InitZbuffer()
{
   //Seems very clear, we initialize the Zbuffer
   fZbuffer = new float[fImageHeight*fImageWidth];

   for ( int i=0; i<fImageHeight*fImageWidth; i++)
      fZbuffer[i] = FPOSINF;
}
```

MixRenderer::AllocMem

```
void MixRenderer::AllocMem()
```

The **AllocMem()** function allocates memory for use as a buffer.

```
void MixRenderer::AllocMem()
{
   //According to the shading flags we instanciate only the requested items
      array
   if ( rightX == NULL)
   {
```

```
            rightX = new float[fImageHeight];
            leftX = new float[fImageHeight];

            rightZ = new float[fImageHeight];
            leftZ = new float[fImageHeight];

            rightGlobalPosX = new float[fImageHeight];
            leftGlobalPosX = new float[fImageHeight];
            rightGlobalPosY = new float[fImageHeight];
            leftGlobalPosY = new float[fImageHeight];
            rightGlobalPosZ = new float[fImageHeight];
            leftGlobalPosZ = new float[fImageHeight];

            rightGlobalNorX = new float[fImageHeight];
            leftGlobalNorX = new float[fImageHeight];
            rightGlobalNorY = new float[fImageHeight];
            leftGlobalNorY = new float[fImageHeight];
            rightGlobalNorZ = new float[fImageHeight];
            leftGlobalNorZ = new float[fImageHeight];
        }

        if (( fFlags.fNeedsUV == (uint)true ) && ( rightU == NULL))
        {
            rightU = new float[fImageHeight];
            leftU = new float[fImageHeight];
            rightV = new float[fImageHeight];
            leftV = new float[fImageHeight];
        }

        if (( fFlags.fNeedsPointLoc == (uint)true ) && ( rightLocPosX == NULL ))
        {
            rightLocPosX = new float[fImageHeight];
            leftLocPosX = new float[fImageHeight];
            rightLocPosY = new float[fImageHeight];
            leftLocPosY = new float[fImageHeight];
            rightLocPosZ = new float[fImageHeight];
            leftLocPosZ = new float[fImageHeight];
        }

        if (( fFlags.fNeedsNormalLoc == (uint)true ) && ( rightLocNorX == NULL ))
        {
            rightLocNorX = new float[fImageHeight];
            leftLocNorX = new float[fImageHeight];
            rightLocNorY = new float[fImageHeight];
            leftLocNorY = new float[fImageHeight];
            rightLocNorZ = new float[fImageHeight];
            leftLocNorZ = new float[fImageHeight];
        }
    }
```

MixRenderer::ReleaseMem

```
void MixRenderer::ReleaseMem()
```

The **ReleaseMem()** function releases memory once a buffer is no longer needed.

```
void MixRenderer::ReleaseMem()
{
    //According to the shading flags we instanciate only the requested items
        array
    if ( leftX != NULL) {delete leftX; leftX=NULL;}
    if ( leftZ != NULL) {delete leftZ; leftZ=NULL;}
```

```
    if ( leftU != NULL) {delete leftU; leftU=NULL;}
    if ( leftV != NULL) {delete leftV; leftV=NULL;}
    if ( leftGlobalPosX != NULL) {delete leftGlobalPosX; leftGlobalPosX=NULL;}
    if ( leftGlobalPosY != NULL) {delete leftGlobalPosY; leftGlobalPosY=NULL;}
    if ( leftGlobalPosZ != NULL) {delete leftGlobalPosZ; leftGlobalPosZ=NULL;}
    if ( leftGlobalNorX != NULL) {delete leftGlobalNorX; leftGlobalNorX=NULL;}
    if ( leftGlobalNorY != NULL) {delete leftGlobalNorY; leftGlobalNorY=NULL;}
    if ( leftGlobalNorZ != NULL) {delete leftGlobalNorZ; leftGlobalNorZ=NULL;}
    if ( leftLocPosX != NULL) {delete leftLocPosX; leftLocPosX=NULL;}
    if ( leftLocPosY != NULL) {delete leftLocPosY; leftLocPosY=NULL;}
    if ( leftLocPosZ != NULL) {delete leftLocPosZ; leftLocPosZ=NULL;}
    if ( leftLocNorX != NULL) {delete leftLocNorX; leftLocNorX=NULL;}
    if ( leftLocNorY != NULL) {delete leftLocNorY; leftLocNorY=NULL;}
    if ( leftLocNorZ != NULL) {delete leftLocNorZ; leftLocNorZ=NULL;}

    if ( rightX != NULL) {delete rightX; rightX=NULL;}
    if ( rightZ != NULL) {delete rightZ; rightZ=NULL;}
    if ( rightU != NULL) {delete rightU; rightU=NULL;}
    if ( rightV != NULL) {delete rightV; rightV=NULL;}
    if ( rightGlobalPosX != NULL) {delete rightGlobalPosX; rightGlobalPosX=NULL;}
    if ( rightGlobalPosY != NULL) {delete rightGlobalPosY; rightGlobalPosY=NULL;}
    if ( rightGlobalPosZ != NULL) {delete rightGlobalPosZ; rightGlobalPosZ=NULL;}
    if ( rightGlobalNorX != NULL) {delete rightGlobalNorX; rightGlobalNorX=NULL;}
    if ( rightGlobalNorY != NULL) {delete rightGlobalNorY; rightGlobalNorY=NULL;}
    if ( rightGlobalNorZ != NULL) {delete rightGlobalNorZ; rightGlobalNorZ=NULL;}
    if ( rightLocPosX != NULL) {delete rightLocPosX; rightLocPosX=NULL;}
    if ( rightLocPosY != NULL) {delete rightLocPosY; rightLocPosY=NULL;}
    if ( rightLocPosZ != NULL) {delete rightLocPosZ; rightLocPosZ=NULL;}
    if ( rightLocNorX != NULL) {delete rightLocNorX; rightLocNorX=NULL;}
    if ( rightLocNorY != NULL) {delete rightLocNorY; rightLocNorY=NULL;}
    if ( rightLocNorZ != NULL) {delete rightLocNorZ; rightLocNorZ=NULL;}
}
```

MixRenderer::GetLights

```
void MixRenderer::GetLights()
```

The **GetLights()** function gets a count of all of the light sources, then places them in an index. It is used by **PrepareDraw()**.

```
void MixRenderer::GetLights()
{

    TMCCountedPtr<I3DShTreeElement> aTree;
    TMCCountedPtr<I3DShScene> scene;
    fTreeTop->QueryInterface(IID_I3DShTreeElement,(void **)&aTree);
    ThrowIfNil(aTree);
    aTree->GetScene(&scene);

    //fLights = new TMCCountedPtrArray<I3DShLightsource>[fNbLights];
    //fLights = new LightTable;
    fNbLights = scene->GetLightsourcesCount();

    for (int32 index=0; index<fNbLights; index++)
    {
        TMCCountedPtr<I3DShLightsource> light;
        scene->GetLightsourceByIndex(&light,index);
        fLights[index] = light;
    }
}
```

MixRenderer::RenderScene

```
void MixRenderer::RenderScene(const TMCRect* area)
```

The **RenderScene()** function is called by the the PrepareDraw method. It allows you to draw the image in the buffer.

```
void MixRenderer::RenderScene(const TMCRect* area)
{
   //InstanceAndTransform *instances ;// Pointer to an array of tree element
   //long nbrInstances ;// Amount of elements in tree hierarchy
   TTransform3D G2C ;//Transformation from Global to Camera
   TTransform3D L2C ;//Transformation from Local to Camera
   TTransform3D L2G ;//Transformation from Local to Global

   TMCCountedPtr<FacetMesh>amesh ;// pointer to IFacetMesh interface

   TFacet3Dfacet;
   TMCColorRGBAcolor, notused;
   int32     theKind=0;//Used to stock the kind of an instance


   fTreeTop->QueryInterface(IID_I3DShTreeElement,(void**)&fTree);
   ThrowIfNil(fTree);
   fTree->BeginGetRenderables(fInstances);


   fCamera->GetGlobalToCameraTransform(&G2C) ; //We get transformation from Glo-
      bal to Camera,
   //actually in Screen Coord

   FacetMeshFacetIterator* facetIterator = new FacetMeshFacetIterator;
   TRenderableAndTfmArray::const_iteratoriter = fInstances->Begin();

   for (const TRenderableAndTfm* current = iter.First(); iter.More(); current =
      iter.Next())
   {
      theKind = current->fInstance->GetInstanceKind();
      if ((theKind == 3) || (theKind == 4))
         continue;

      L2G = current->fT ;//In InstanceAndTransform fT is not a Translation as we
      may imagine
                          //but a AFFINETRANSFORM cad Translation + Rotation
                          //we get transformation from Local to Global system

      //Then we get our complete transformation from local to Camera Coordinate
      L2C = G2C*L2G;

      current->fInstance->GetShadingFlags(fFlags); //We get flags from shaders
      current->fInstance->GetIndex(fIndex);

      //We need to know whether the mapping concerning the instance is paramet-
      rical or projectional
      /*if ( current->fInstance->GetMappingKind() == kParametricMapping )
         fParametricMapping = true;
      else
      {
         fParametricMapping = false;
         fFlags.fNeedsPointLoc = true;//DoShade will need Coord in Local system
      to compute uv coordinates.
         fFlags.fNeedsUV = false;//We don't need to compute uv coord, DoShade
      do it by using Coord in Local system.
```

```
            fFlags.fNeedsIsoUV = false;//idem for isoUV
        }*/
        fParametricMapping = true;

        AllocMem(); //We allocate the right buffers for the right items

        current->fInstance->GetMainColors(color,notused);
        current->fInstance->GetFMesh(0.0, &amesh) ;//get a pointer to IFacetMesh
        interface

        facetIterator->Initialize( amesh ) ;//Initialization for a using of a
        facet iterator

        for ( facetIterator->First(); facetIterator->More(); facetIterator-
        >Next())
        { // browsing into facets list
            facet = facetIterator->GetFacet();
            Rasterize( facet, L2G, L2C, current->fInstance );
        }

        //We're now filling the pixels which don't refer to any object and thus
        have been left aside
        for (int y = 0; y < fImageHeight ; y++)
        {
            TMCColorRGBA color;
            int line = y*fImageWidth;
            TVector2 screenXY;
            screenXY[1] = (real)y;
            for (int x=0; x<fImageWidth; x++)
            {
                if (fZbuffer[line+x] == FPOSINF)
                {
                    color.R = 0;
                    color.G = 0;
                    color.B = 0;
                    screenXY[0] = (real)x;
                    BackgroundColor(screenXY,color);
                    PixelSet(x,y,&color);
                }
            }
        }
        ReleaseMem();
    }
    delete facetIterator;
    fTree->EndGetRenderables();
    fTree->Release();
    fTreeTop->Release();
}
```

MixRenderer::ZBufferSet

```
void MixRenderer::ZBufferSet (int32 x, int32 y, real val)
```

The **ZBufferSet()** function sets the ZBuffer.

```
void MixRenderer::ZBufferSet (int32 x, int32 y, real val)
{
    if ((x>0)&&(x<fImageWidth)&&(y>0)&&(y<fImageHeight))
    {
        fZbuffer[x + fImageWidth*y] = val;
    }
}
```

MixRenderer::Interpolate

```
void MixRenderer::Interpolate(float A, float B, long y1, long y2, long dy, float
      *array)
```

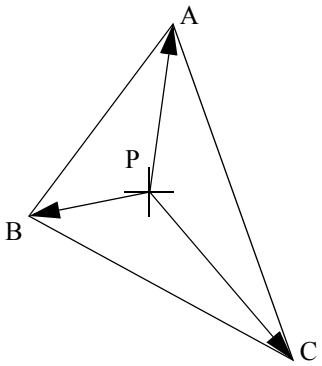
The **Interpolate()** function linearly interpolates the values between A and B and stores them in array.

```
void MixRenderer::Interpolate(float A, float B, long y1, long y2, long dy, float
      *array)
{
   float slope;
   float x;

   slope = (A - B)/dy;

   if ( dy < 0 ) x = A;
   else x = B;

   for (int j = y1 ; j <= y2 ; j++ )
   {
      if ( j >= 0 && j < fImageHeight) array[j] = x; //Clipping y axis in Screen-
      Pixel Space
      x += slope;
   }
}
```

MixRenderer::FillPoly

```
void MixRenderer::FillPoly(TFacet3D facet, I3DShInstance *CurrentInstance, Shad-
      ingIn *myShadingIn, ShadingOut *myShadingOut, int numVertices)
```

The **FillPoly()** function fills in the polygons, however, it needs counter-clockwise oriented triangles.



FillPoly is a major piece of code for the MixRenderer, since it is where rasterization takes place. The most common approach to rasterization is to use a barycentric algorithm. Considering a point on the facet you interpolate its coordinates, normal, and screen position from the 3 vertices.

Example : finding the depth of a point from depth of 3 vertices.

$$P_Z = \alpha A_Z + \beta B_Z + \Gamma C_Z;\ \ \alpha = |\mathbf{AP}|/\Delta, \beta = |\mathbf{BP}|/\Delta, \Gamma = |\mathbf{CP}|/\Delta;\ \ \Delta = |\mathbf{AP}| + |\mathbf{BP}| + |\mathbf{CP}|$$

The advantages of this method are:

- it is simple
- it requires no memory allocation, thus reducing memory leaks and suballocation errors

Its main drawback is obvious : you have to recalculate every coefficient for every point.

```
void MixRenderer::FillPoly(TFacet3D facet, I3DShInstance *CurrentInstance, Shad-
      ingIn *myShadingIn, ShadingOut *myShadingOut, int numVertices)
{

   long ymin , ymax;
```

```
    long y1,y2, dy;
    ymin = IPOSINF ;
    ymax = INEGINF ;

    //Be careful, facets are not oriented, and our polygon-filling needs counter-
       clockwise oriented triangles
    int PointOrder[3] = { 0, 1, 2};

    TVector3 res = MixNormalFacet(fScreenPoint[0], fScreenPoint[1], fScreen-
       Point[2]);

    if ( res[2] > 0 )
    {
       PointOrder[1] = 2;
       PointOrder[2] = 1;
    }
//We interpolate along triangle edges, ie we interpolate vertically
    //We just interpolate items wanted by the RayDream Shader
    //Basically, to interpolate, we use two arrays : one left and one right.
    //If the next vertex (counterclockwisely speaking), is below we use left
       array
    //else we use right array.
    //Roughly speaking, we use bilinear interpolation
    for ( int k = 0 ; k < numVertices ; k++ )
    {
       int i = PointOrder[k];
       y1 = (long)(fScreenPoint[i][1]+0.5);
       int t = PointOrder[(k+1)%numVertices];
       y2 = (long)(fScreenPoint[t][1]+0.5);

       if ( y1 > y2 )
       {
          if ( y2 < ymin ) ymin = y2;
          if ( y1 > ymax ) ymax = y1;

          dy = y2-y1;

          Interpolate(fScreenPoint[t][0], fScreenPoint[i][0], y2, y1, dy,
       rightX);
          Interpolate(fScreenPoint[t][2], fScreenPoint[i][2], y2, y1, dy,
       rightZ);

          Interpolate(fGlobalVertex[t].fVertex[0], fGlobalVertex[i].fVertex[0],
       y2, y1, dy, rightGlobalPosX);
          Interpolate(fGlobalVertex[t].fVertex[1], fGlobalVertex[i].fVertex[1],
       y2, y1, dy, rightGlobalPosY);
          Interpolate(fGlobalVertex[t].fVertex[2], fGlobalVertex[i].fVertex[2],
       y2, y1, dy, rightGlobalPosZ);

          Interpolate(fGlobalVertex[t].fNormal[0], fGlobalVertex[i].fNormal[0],
       y2, y1, dy, rightGlobalNorX);
          Interpolate(fGlobalVertex[t].fNormal[1], fGlobalVertex[i].fNormal[1],
       y2, y1, dy, rightGlobalNorY);
          Interpolate(fGlobalVertex[t].fNormal[2], fGlobalVertex[i].fNormal[2],
       y2, y1, dy, rightGlobalNorZ);

          if ( fFlags.fNeedsUV == (uint)true )
          {
             Interpolate(facet.fVertices[t].fUV[0], facet.fVertices[i].fUV[0],
       y2, y1, dy, rightU);
             Interpolate(facet.fVertices[t].fUV[1], facet.fVertices[i].fUV[1],
```

```
y2, y1, dy, rightV);
   }

   if ( fFlags.fNeedsPointLoc == (uint)true )
   {

      Interpolate(facet.fVertices[t].fVertex[0], facet.fVerti-
ces[i].fVertex[0], y2, y1, dy, rightLocPosX);
      Interpolate(facet.fVertices[t].fVertex[1], facet.fVerti-
ces[i].fVertex[1], y2, y1, dy, rightLocPosY);
      Interpolate(facet.fVertices[t].fVertex[2], facet.fVerti-
ces[i].fVertex[2], y2, y1, dy, rightLocPosZ);
   }

   if ( fFlags.fNeedsNormalLoc == (uint)true )
   {
      Interpolate(facet.fVertices[t].fNormal[0], facet.fVerti-
ces[i].fNormal[0], y2, y1, dy, rightLocNorX);
      Interpolate(facet.fVertices[t].fNormal[1], facet.fVerti-
ces[i].fNormal[1], y2, y1, dy, rightLocNorY);
      Interpolate(facet.fVertices[t].fNormal[2], facet.fVerti-
ces[i].fNormal[2], y2, y1, dy, rightLocNorZ);
   }
}

else if ( y1 < y2 )
{
   if ( y1 < ymin ) ymin = y1 ;
   if ( y2 > ymax ) ymax = y2 ;

   dy = y2 - y1;

   Interpolate(fScreenPoint[t][0], fScreenPoint[i][0], y1, y2, dy,
leftX);
   Interpolate(fScreenPoint[t][2], fScreenPoint[i][2], y1, y2, dy,
leftZ);

   Interpolate(fGlobalVertex[t].fVertex[0], fGlobalVertex[i].fVertex[0],
y1, y2, dy, leftGlobalPosX);
   Interpolate(fGlobalVertex[t].fVertex[1], fGlobalVertex[i].fVertex[1],
y1, y2, dy, leftGlobalPosY);
   Interpolate(fGlobalVertex[t].fVertex[2], fGlobalVertex[i].fVertex[2],
y1, y2, dy, leftGlobalPosZ);

   Interpolate(fGlobalVertex[t].fNormal[0], fGlobalVertex[i].fNormal[0],
y1, y2, dy, leftGlobalNorX);
   Interpolate(fGlobalVertex[t].fNormal[1], fGlobalVertex[i].fNormal[1],
y1, y2, dy, leftGlobalNorY);
   Interpolate(fGlobalVertex[t].fNormal[2], fGlobalVertex[i].fNormal[2],
y1, y2, dy, leftGlobalNorZ);


   if ( fFlags.fNeedsUV == (uint)true )
   {
      Interpolate(facet.fVertices[t].fUV[0], facet.fVertices[i].fUV[0],
y1, y2, dy, leftU);
      Interpolate(facet.fVertices[t].fUV[1], facet.fVertices[i].fUV[1],
y1, y2, dy, leftV);
   }

   if ( fFlags.fNeedsPointLoc == (uint)true )
```

```
            {

                Interpolate(facet.fVertices[t].fVertex[0], facet.fVerti-
        ces[i].fVertex[0], y1, y2, dy, leftLocPosX);
                Interpolate(facet.fVertices[t].fVertex[1], facet.fVerti-
        ces[i].fVertex[1], y1, y2, dy, leftLocPosY);
                Interpolate(facet.fVertices[t].fVertex[2], facet.fVerti-
        ces[i].fVertex[2], y1, y2, dy, leftLocPosZ);
            }

            if ( fFlags.fNeedsNormalLoc == (uint)true )
            {
                Interpolate(facet.fVertices[t].fNormal[0], facet.fVerti-
        ces[i].fNormal[0], y1, y2, dy, leftLocNorX);
                Interpolate(facet.fVertices[t].fNormal[1], facet.fVerti-
        ces[i].fNormal[1], y1, y2, dy, leftLocNorY);
                Interpolate(facet.fVertices[t].fNormal[2], facet.fVerti-
        ces[i].fNormal[2], y1, y2, dy, leftLocNorZ);
            }

        }
    }

    //n-gones filling

    floatdelta;
    floatz;
    floatdeltaZ, deltaGlobalPosX, deltaGlobalPosY, deltaGlobalPosZ,
        deltaGlobalNorX, deltaGlobalNorY, deltaGlobalNorZ,
        deltaLocPosX, deltaLocPosY, deltaLocPosZ,
        deltaLocNorX, deltaLocNorY, deltaLocNorZ,
        deltaU, deltaV;

    floatGlobalPosX, GlobalPosY, GlobalPosZ,
        GlobalNorX, GlobalNorY, GlobalNorZ,
        LocPosX, LocPosY, LocPosZ,
        LocNorX, LocNorY, LocNorZ,
        U,V;


    TVector2 uv;

    if ( ymin < 0 ) ymin = 0;
    if ( ymax > fImageHeight) ymax = fImageHeight-1;

    //We fill and interpolate horizontally from left array to right array
    for (long y = ymin  ; y <= ymax ; y++ )
    {

        //Slope initialization for each item
        if ( (delta = leftX[y] - rightX[y]) == 0 )
        {
            deltaZ = 0;
            deltaGlobalPosX = 0; deltaGlobalPosY = 0; deltaGlobalPosZ = 0;
            deltaGlobalNorX = 0; deltaGlobalNorY = 0; deltaGlobalNorZ = 0;
            deltaU = 0; deltaV = 0;
            deltaLocPosX = 0; deltaLocPosY = 0; deltaLocPosZ = 0;
            deltaLocNorX = 0; deltaLocNorY = 0; deltaLocNorZ = 0;
        }

        else
```

```
{
   deltaZ = (leftZ[y]-rightZ[y])/delta;
   deltaGlobalPosX = (leftGlobalPosX[y]-rightGlobalPosX[y])/delta;
   deltaGlobalPosY = (leftGlobalPosY[y]-rightGlobalPosY[y])/delta;
   deltaGlobalPosZ = (leftGlobalPosZ[y]-rightGlobalPosZ[y])/delta;

   deltaGlobalNorX = (leftGlobalNorX[y]-rightGlobalNorX[y])/delta;
   deltaGlobalNorY = (leftGlobalNorY[y]-rightGlobalNorY[y])/delta;
   deltaGlobalNorZ = (leftGlobalNorZ[y]-rightGlobalNorZ[y])/delta;

   if ( fFlags.fNeedsUV == (uint)true )
   {
      deltaU = (leftU[y]-rightU[y])/delta;
      deltaV = (leftV[y]-rightV[y])/delta;
   }

   if ( fFlags.fNeedsPointLoc == (uint)true )
   {
      deltaLocPosX = (leftLocPosX[y]-rightLocPosX[y])/delta;
      deltaLocPosY = (leftLocPosY[y]-rightLocPosY[y])/delta;
      deltaLocPosZ = (leftLocPosZ[y]-rightLocPosZ[y])/delta;
   }

   if ( fFlags.fNeedsNormalLoc == (uint)true )
   {
      deltaLocNorX = (leftLocNorX[y]-rightLocNorX[y])/delta;
      deltaLocNorY = (leftLocNorY[y]-rightLocNorY[y])/delta;
      deltaLocNorZ = (leftLocNorZ[y]-rightLocNorZ[y])/delta;
   }
}

//First value initialization
z = leftZ[y];

GlobalPosX = leftGlobalPosX[y];
GlobalPosY = leftGlobalPosY[y];
GlobalPosZ = leftGlobalPosZ[y];

GlobalNorX = leftGlobalNorX[y];
GlobalNorY = leftGlobalNorY[y];
GlobalNorZ = leftGlobalNorZ[y];

if ( fFlags.fNeedsPointLoc == (uint)true )
{
   LocPosX = leftLocPosX[y];
   LocPosY = leftLocPosY[y];
   LocPosZ = leftLocPosZ[y];
}
if ( fFlags.fNeedsNormalLoc == (uint)true )
{
   LocNorX = leftLocNorX[y];
   LocNorY = leftLocNorY[y];
   LocNorZ = leftLocNorZ[y];
}
if ( fFlags.fNeedsUV == (uint)true )
{
   U = leftU[y];
   V = leftV[y];
}

//From left array to right array
```

```
for ( int x = (int)(leftX[y]+0.5) ; x < (int)(rightX[y]+0.5) ; x++ )
{

   if ( x >= fFrontier && x < fImageWidth )
   {  //Clipping in x-axis

      int ind = y*fImageWidth + x;

      //Zbuffer test
      if ( z < fZbuffer[ind] )
      {
         fZbuffer[ind] = z;

         if ( fFlags.fNeedsPoint == (uint)true )
         {
            myShadingIn->fPoint[0] = GlobalPosX;
            myShadingIn->fPoint[1] = GlobalPosY;
            myShadingIn->fPoint[2] = GlobalPosZ;
         }

         if ( fFlags.fNeedsNormal == (uint)true )
         {
            myShadingIn->fGNormal[0] = GlobalNorX;
            myShadingIn->fGNormal[1] = GlobalNorY;
            myShadingIn->fGNormal[2] = GlobalNorZ;
         }

         if ( fFlags.fNeedsUV == (uint)true )
         {
            uv[0] = U;
            uv[1] = V;
            myShadingIn->fUV = uv;
         }

         if ( fFlags.fNeedsPointLoc == (uint)true )
         {
            myShadingIn->fPointLoc[0] = LocPosX;
            myShadingIn->fPointLoc[1] = LocPosY;
            myShadingIn->fPointLoc[2] = LocPosZ;
         }

         if ( fFlags.fNeedsNormalLoc == (uint)true )
         {
            myShadingIn->fNormalLoc[0] = LocNorX;
            myShadingIn->fNormalLoc[1] = LocNorY;
            myShadingIn->fNormalLoc[2] = LocNorZ;
         }

         //We fill ShadingIn structure and we get ShadingOut structure
         CurrentInstance->DoShade(*myShadingOut,*myShadingIn);

         TVertex3D thePoint;
         thePoint.fVertex[0] = GlobalPosX;
         thePoint.fVertex[1] = GlobalPosY;
         thePoint.fVertex[2] = GlobalPosZ;

         thePoint.fNormal[0] = GlobalNorX;
         thePoint.fNormal[1] = GlobalNorY;
         thePoint.fNormal[2] = GlobalNorZ;

         TVector2 pos;
```

```
                    pos[0] = x - fOffscrOffset[0];
                    pos[0] /= fZoom[0];
                    pos[1] = -y + fOffscrOffset[1];
                    pos[1] /= fZoom[1];

                    TVector3 resultOrigin, resultDirection;

                    //We get resultDirection giving the view direction for each
          pixel
                    fCamera->CreateRay(&pos, &resultOrigin, &resultDirection);

                    //We get color pixel
                    TMCColorRGBA color;
                    color.R = 10000;
                    color.G = 00;
                    color.B = 00;
                    PixelColor(myShadingOut, thePoint, fFlags.fChangesNormal,
                       resultDirection, resultOrigin, color);

                    //We display each pixel
                    PixelSet(x,y,&color);

                    //Fill the gBuffer
                    //if ( fHasGBuffer32 == true )
                    // StoreGBuffers(&thePoint , f32Data.channel, x, y, z, &myShad-
          ingOut, myShadingIn->fUV);
                  }
               }

               z += deltaZ;

               GlobalPosX += deltaGlobalPosX;
               GlobalPosY += deltaGlobalPosY;
               GlobalPosZ += deltaGlobalPosZ;

               GlobalNorX += deltaGlobalNorX;
               GlobalNorY += deltaGlobalNorY;
               GlobalNorZ += deltaGlobalNorZ;

               LocPosX += deltaLocPosX;
               LocPosY += deltaLocPosY;
               LocPosZ += deltaLocPosZ;

               LocNorX += deltaLocNorX;
               LocNorY += deltaLocNorY;
               LocNorZ += deltaLocNorZ;
               U += deltaU;
               V += deltaV;
            }
         }
      }
```

MixRenderer::BackgroundColor

```
void MixRenderer::BackgroundColor(TVector2 theScreenUV, TMCColorRGBA& result-
      Color)
```

The **BackgroundColor()** function is used to fill pixels of infinite depth in Zbuffer.

```
void MixRenderer::BackgroundColor(TVector2 theScreenUV, TMCColorRGBA& result-
      Color)
{
```

```
        TVector2UVPoint;
        TVector3rayOrigin, rayDirection;

        resultColor.R = (float)0.2;
        resultColor.G = (float)0.3;
        resultColor.B = (float)0.4;

        UVPoint[0] = theScreenUV[0]-fOffscrOffset[0];
        UVPoint[0] /= fZoom[0];
        UVPoint[1] = -theScreenUV[1]+fOffscrOffset[1];
        UVPoint[1] /= fZoom[1];

        fCamera->CreateRay(&UVPoint, &rayOrigin, &rayDirection);

        boolean fullAreaDone=false;
        //If there is no backdrop but a background, it appears behind the scene
        if (fBackground)
        {
            fBackground->GetEnvironmentColor(rayDirection,fullAreaDone,resultColor);
        }

        //Overrides the background
        if (fBackdrop)
        {
            fBackdrop->GetBackdropColor(UVPoint, fullAreaDone,fUVMinMax, result-
            Color);
        }

        if (fAtmosphere)
        {
            fAtmosphere->DirectionFilter(rayOrigin,rayDirection,resultColor);
        }
    }
```

Writing a Light Source Gel

Family ID : 'gel'
Interface ID : IID_I3DExLightsourceGel
Interface file : I3DExGel.h
Basic Implementation file: BasicCameraLightGel.h, BasicCameraLightGel.cpp

Overview

A Light Gel is a colored slide placed in front of a light source to modify the color and intensity of the light beam.
I3DExGel has the following method:

- **GetGelValues(TVector2* gelScreenPosition, TMCColorRGB &result)**

As with **GetColor()** from *I3DExLight* **GetGelValues()** returns true if the point from *gelScreenPosition* is affected by the Gel and false otherwise. The change is passed in the variable *result*. The point in *gelScreenPosition* is a 2D point on a virtual slide with values from -1.0 to 1.0 on each axis.

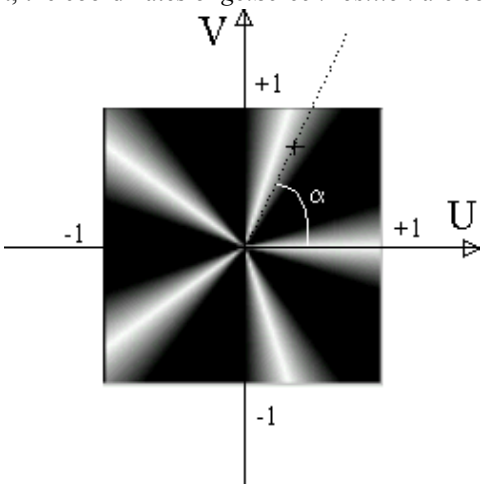
Location in the User Interface

Gels are added in the Assemble room from the Effects panel of the Properties tray while a light is selected.

Example: Gel

Description

Gel creates a star shaped slide. The user is able to set the number of points for the star from 3 to 30.
To create the star effect, the coordinates of *gelScreenPosition* are converted to polar coordinates:



Gel Screen

The angle α is used to find the branches of the star by taking the modulo:

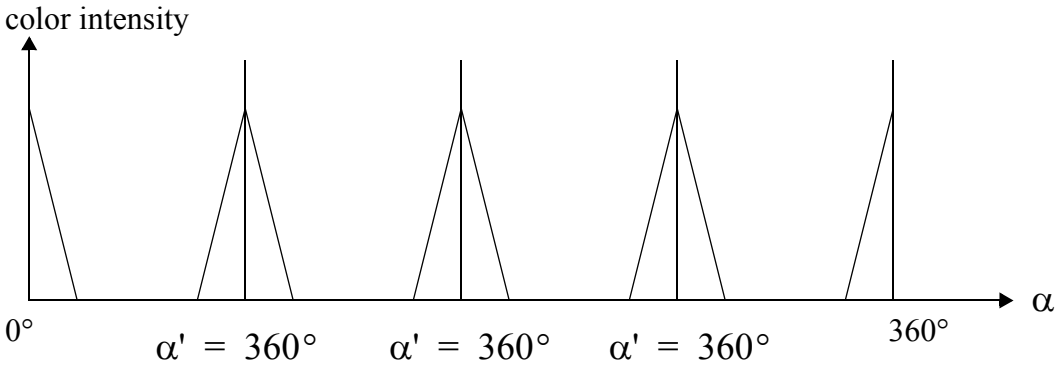
$$\alpha' = (\alpha \times nbBranches)\%360$$

- α' is an angle between 0 and 360°
- *nbBranches* is the number of branches of the star

α' is then converted to a color factor as shown below:

Transformation function

The peaks are periodically spaced in the α-Space:



Example of α-peaks with nbBranches=4

Parameters

The following variables are used for the user-input parameters:

fNbBranches: The number of points on the star.

These are defined in NonUniformScale.h as follows:

```
struct GelData
{
    int16 fNbBranches;  // Nb of branches of the star
};
```

These are then mapped to the user interface parameters using the PMap resource in Gel.r. Note that the parameters within the *GelData* struct must be in the same order as in the PMap.

Functions

Gel::GetGelValues

```
boolean Gel::GetGelValues(TVector2* gelScreenPosition, TMCColorRGB &result)
```

GetGelValues implements the formula described above to create the star gel. It will always returns true as the value in *result* is always changed by the gel.

```
boolean Gel::GetGelValues(TVector2* gelScreenPosition, TMCColorRGB &result)
{
    real alpha;
    real graylevel;
    alpha=atan2((*gelScreenPosition)[1],(*gelScreenPosition)[0]); //QuickArcSin-
        Cos((*gelScreenPosition)[1],(*gelScreenPosition)[0],alpha);
    alpha*=fData.fNbBranches;
```

```
    while (alpha<0.0) alpha+=kPI*2.0; // Modulo 2 pi
    while (alpha>kPI*2.0) alpha-=kPI*2.0; // Modulo 2 pi

    if (alpha<kPI/2.0)
    {
       graylevel=1.0-alpha/(kPI/2.0);
    }
    else if (alpha>kPI*1.5)
    {
       graylevel=(alpha-kPI*1.5)/(kPI/2.0);
    }
    else
    {
       graylevel=0.0;
    }

    result.R=graylevel;
    result.G=graylevel;
    result.B=graylevel;

    return true;
}
```

Writing a 3D Import Filter

Family ID : '3Din'
Interface ID : IID_I3DExImportFilter
Interface file : I3DExImportExport.h
Basic Implementation file: Basic3DImportExport.h, Basic3DImportExport.cpp

Overview

Writing an import filter is easier than trying to create a file with the Carrara file format.

There is one method which must be implemented:
DoImport(IMCFile* file, IMCUnknown* elem, IMCUnknown* subElem);

DoImport() is used to import the scene data.
Note the heavy use of the QueryInterface() call. Make sure you are familiar with the I3DShPrimitive, I3DShTreeElement and I3DShInstance interfaces.
Make sure you build the right 'Cmpp' (Component Private) resource. It contains the information necessary to the 3D Shell to display your file format extension and name in the Open and Import dialogs.
Please refer to the family-specific resources section in the Cookbook for all details.

Location in the User Interface

An importer is accessed either from the File menu> Open command or the File menu> Import command. If you return true in WantsOptionDialog(), your importer's options will be shown in a dialog before loading the file.

Example: Imp

Description

This example is a facets importer. The file format is text based and the extension on the PC is .eas for "Easy". This file can be created with any text editor.

An easy file is a set of surfaces displayed as follows :

number of points
first point
second point
...
last point

After the last surface, a zero must be added to specify the end of the file, followed by a space or a carriage return to prevent reading failure. For example, a square will be defined as follow:

```
4       number of points
0 0 0   first point (x, y, and z)
0 1 0   second point
1 1 0   third point
1 0 0   last point

0       no more surfaces, don't forget to add a space or a carriage return
```

Parameters

No parameters are defined for user input.

Functions

Imp::DoImport

```
MCCOMErr Imp::DoImport(IMCFile* file, IMCUnknown* firstPtr, IMCUnknown*
  secondPtr)
```

The **DoImport()** function creates a 3D object then inserts it in the scene. Next it creates a default rendering camera and light source.
The first part of this function determines where you need to insert your 3D data.

Start by getting a pointer on the scene and a pointer on the Tree Element under which to insert your data. Often, this Tree Element will be NULL. In this case, you need to insert under the Scene Tree Root. It is a sensible thing to call I3DShScene::CreateTreeRootIfNone() to make sure there is one, just in case.

In the following code example, we choose to create a group if *fatherTree* is provided. This is a choice we make. The variable *topTree* is where we shall insert our data in the end:

```
MCCOMErr Imp::DoImport(IMCFile* file, IMCUnknown* firstPtr, IMCUnknown* sec-
      ondPtr)
{
   TMCfstream*  stream = NULL;
   TMCCountedPtr<I3DShScene> scene;
   TMCCountedPtr<I3DShTreeElement> fatherTree;
   TMCCountedPtr<I3DShTreeElement> topTree;
   MCCOMErr error;

   error = firstPtr->QueryInterface(IID_I3DShScene,(void**)&scene);
   if (error != MC_S_OK)
      return error;
   else if (scene == NULL)
      return MC_E_FAIL;

   if (secondPtr != NULL)
   {
      error = secondPtr->QueryInterface(IID_I3DShTreeElement,(void**)&father-
      Tree);
      if (error != MC_S_OK)
         return error;
      else if (fatherTree == NULL)
         return MC_E_FAIL;
   }

   TMCCountedPtr<IMCUnknown> progressKey;
   try
   {
      //     *****     method to obtain stream:******
      TMCString1023 fullPathName;
      file->GetFileFullPathName(fullPathName);
      stream = new TMCfstream(fullPathName.StrGet(),TMCiostream::in);
      ThrowIfNil(stream);

      TMCString255 progressMsg;
      void* oldResources = NULL;
      gResourceUtilities->SetupComponentResources('3Din', 'COEA', &old-
      Resources);
      gResourceUtilities->GetIndString(progressMsg, 130, 1);// string  1 of
```

```
        STR# 130 (see Imp.r)
        gResourceUtilities->RestoreComponentResources(oldResources);

        gShellUtilities->BeginProgress(progressMsg, &progressKey);

        if (fatherTree == NULL)
        {
            TMCCountedPtr<I3DShGroup> topGroup;
            scene->CreateTreeRootIfNone();
            MCCOMErr err = scene->GetTreeRoot(&topGroup);
            topGroup->QueryInterface(IID_I3DShTreeElement,(void**)  &topTree);
        }
        else
        {
            gComponentUtilities->CoCreateInstance(CLSID_StandardGroup, NULL,
        MC_CLSCTX_INPROC_SERVER, IID_I3DShTreeElement, (void**) &topTree);
            ThrowIfNil(topTree);
            topTree->SetScene(scene);
            fatherTree->InsertLast(topTree);
        }



        scene->CreateRenderingCameraIfNone('coni', (fatherTree == NULL));// Cre-
        ate a conical rendering camera if none, and a Distant light if we not
        importing in an existing scene
        DoReadEasyFile(stream, scene, topTree);


        delete stream;
        gShellUtilities->EndProgress(progressKey);
    }

    catch (...)
    {
        delete stream;
        gShellUtilities->EndProgress(progressKey);
        throw;
    }
    return MC_S_OK;
}
```

Because the Easy format does not have the notion of cameras and light sources, we shall use a built-in API call designed just for that: CreateRenderingCameraIfNone creates a conical camera (we use the Conical Camera Class ID ‘coni’), and a default Distant Light if we are not importing in an existing scene (in this case, the second parameter is TRUE).
Of course, you can create any type of standard camera and light sources, and place them where you want in 3D.

Writing a Modeler

Family ID : 'modu'
Interface ID : IID_I3DExModule
Interface file : I3DExModule.h
Basic Implementation file: BasicModule.h, BasicModule.cpp

Overview

A modeler allows for the creation of geometry. The geometry is stored in a geometric primitive of a type supported by the modeler, such as a spline primitive for the Spline modeler. You can create any type of module such as another renderer. In this example, you'll learn how to create a modeler.

I3DExModule has the following methods:

- **Initialize(IMCUnknown* inElement)**
- **Destroy()**
- **Hydrate()**
- **Dehydrate()**
- **Activate()**
- **Deactivate()**
- **Import(IMCUnknown* inElement, void* inImportData)**
- **BuildMenuBar()**
- **GetPreferredRoom(IDType& roomId)**
- **WantsToBeInCurrentRoom()**
- **SubModuleClosedMainWindow(I3DShModule* subModule)**
- **SubModuleAboutToBeDestroyed(I3DShModule* subModule)**
- **SubModuleAboutToBeHydrated(I3DShModule* subModule)**
- **AboutToCloseMainWindow()**

Location in the User Interface

Switching to the modeling room will open the default modeler for the currently selected object.

Example 1: Modeler Step 1

Description

The Tree primitive is used by the modeler. It contains the geometry created by your external modeler. It is very similar to the other examples from the "Writing a Geometric Primitive".

Parameters

The following variables are used for the user-input parameters:

```
fNbBranches:    Number of branches for the tree.
fAngle:         Angle between branches.
fRecursiv:      Number of times to recursively parse each branch.
```

These are defined in TreePrim.h as follows:

```
struct TreeData
{
```

```
    int16 fNbBranches;
    float  fAngle;
    int16 fRecursiv;
};
```

These are then mapped to the user interface parameters using the PMap resource in TreePrim.r. Note that the parameters within the *TreeData* struct must be in the same order as in the PMap. To understand this example, it is divided into basic steps.

What you need for a modeler

First, you need a primitive to model. In this case you'll model a tree. Start by coding the primitive first. Once you have your primitive you'll go through the modeler. This one will be able to change parameter of your tree. But it also needs to send messages to the rest of Carrara. For example the modeler needs to make sure that the Properties tray gets updated, anytime you change the angle between the tree branches.

The first step

In the sample folder you'll find a Step1 folder. In this step you'll make an extension that allow you to create a tree primitive. The way to code this step of the modeler is the same as the other primitive's samples. Refer to the primitive samples ("Writing a Geometric Primitive") to learn more.

```
struct TreeData
{
    int16 fNbBranches;// Number of branches of the 3D Tree
    float  fAngle;// Angle for branches
    int16 fRecursiv;// Recursivity number for tree construction
};

// Tree Object definition :
// TreePrimitive Object :

class COMTreePrim : public TBasicPrimitive
{
    public :
        COMTreePrim();
        STANDARD_RELEASE;

        // IExDataExchanger methods :
        virtual void* MCCOMAPIGetExtensionDataBuffer();
        virtual MCCOMErr MCCOMAPIExtensionDataChanged();
        virtual int16 MCCOMAPIGetResID();

        // I3DExGeometricPrimitiv methods
        // Geometric Calls
        virtual MCCOMErr MCCOMAPI        GetBBox( TBox3D* bbox);
        virtual MCCOMErr MCCOMAPI       EnumFacets( EnumFacetsCallback callback,
 void* privData,real fidelity);


        // Shading Calls
        virtual uint32MCCOMAPIGetUVSpaceCount();
        virtual MCCOMErr MCCOMAPI        GetUVSpace( uint32 uvSpaceID, UVSpace-
        Info* uvSpaceInfo);

        virtual booleanMCCOMAPISetData(const TreeData& data);

        virtual TreeDataMCCOMAPIGetData()
        {
            return fData;
        };
```

```
}
```

Example 2: Modeler Step 2

Description

This step creates a 'blank' modeler for the primitive. There are no additional user input parameters as the parameters are all stored in the primitive.

Functions

The second step:
In this second step, we begin the implementation of the modeler. So now you can create a tree primitive, and if you go to the model room of Carrara you'll have a modeler for this primitive. For the moment it won't do anything. The goal of this step is to learn how to instantiate a modeler.
What to do ?

First of all we need a MyModelerDef.h files. In this file you must have all the R_CLSID. Here you need one for the primitive and another for the modeler. You can also define useful IDs.

```
#ifndef __ModelerDEF__
#define __ModelerDEF__

// Define the External Modeler CLSID
// {99E51BC1-707B-11d1-B20A-004095455A27}
#define R_CLSID_Modeler 0x99e51bc1, 0x707b, 0x11d1, 0xb2, 0xa, 0x0, 0x40, 0x95,
       0x45, 0x5a, 0x27

#define kModelerID'MODp'// Class ID
#define kModeler 134// COMP ID
#define kModelerPrefs144
#define kModelerView135// 'MODv'

// Define the COMTreePrim CLSID
// B63DF131-707B-11d1-B20A-004095455A27
#define R_CLSID_COMTreePrim 0xb63df131, 0x707b, 0x11d1, 0xb2, 0x0a, 0x00, 0x40,
       0x95, 0x45, 0x5a, 0x27

#endif
```

The two files of the primitive don't change much. We just have to create a modeler with the MakeCOMobject method in the MyPrimitive.cpp file. Just add a line to test the class ID. If it is a MyModeler ID then create a modeler.

```
TBasicUnknown* MakeCOMObject(const MCCLSID& classId)// This method instanciate
{                                                   // the object COMTreePrim
   TBasicUnknown* res = NULL;

   if (classId == CLSID_COMTreePrim) res = new COMTreePrim;
   if (classId == CLSID_Modeler) res = new Modeler;

   return res;
}
```

The MyModeler.h file:

This file contains the description of the modeler class. This class inherits from TBasicModule (we need to include BasicModule.h).The most important part in the description of the class is the declaration of the protected menbers. Most of them are needed to make the modeler communicate with the shell and other parts. All the TMCCountedPtr will be usefull in order to achieve this goal. We also have a few methods that are used during the initialization.

```
class Modeler : public TBasicModule
{
    public :

        Modeler();

        STANDARD_RELEASE;

        // IMCUnknown methods
        virtual MCCOMErrMCCOMAPI QueryInterface(const MCIID& riid, void**
        ppvObj);
        virtual uint32MCCOMAPI AddRef();


        virtual MCCOMErr MCCOMAPI Initialize(IMCUnknown* inElement);
        virtual MCCOMErr MCCOMAPI Destroy();
        virtual MCCOMErr MCCOMAPI Hydrate();
        virtual MCCOMErr MCCOMAPI Dehydrate();

        virtual MCCOMErrMCCOMAPI Activate();
        virtual MCCOMErrMCCOMAPI Deactivate();

    protected :

        TMCCountedPtr<I3DShModule>fModule;
        TMCCountedPtr<IMFDocument>fDoc;
        TMCCountedPtr<ISceneDocument>fSceneDoc;
        TMCCountedPtr<I3DShScene>fScene;
        TMCCountedPtr<I3DShGroup>fUniverse;
        TMCCountedPtr<IMFPart>fWindow;
        TMCCountedPtr<IMFPart>fMainPart;
        TMCCountedPtr<I3DShModule>fHierarchyModule;

        //-- Dependencies data
        TMCCountedPtr<IChangeChannel>fTimeChangeChannel;

        //-- Primitive Data
        TMCCountedPtr<I3DShPrimitive>fPrimitiveData;
        TMCCountedPtr<IComponentAnim>fClonedComponent;

        virtual MCCOMErrMCCOMAPI Initialize1(I3DShGroup* universe);
        virtual MCCOMErrMCCOMAPI Initialize2(I3DShPrimitive* primitive);
        virtual voidMCCOMAPI CreateWindows();
        virtual voidMCCOMAPI DestroyWindows();

        void GetActionResponder(IMFResponder** responder);
};
```

The MyModeler.cpp file:
We'll describe each method of the Modeler:
QueryInterface and AddRef: We just have to make sure that the TBasicModule methods are called.

Functions

Modeler::QueryInterface

```
MCCOMErr Modeler::QueryInterface(const MCIID& riid, void** ppvObj)
```

We just have to make sure that the TBasicModule method is called.

```
MCCOMErr Modeler::QueryInterface(const MCIID& riid, void** ppvObj)
{
    return TBasicModule::QueryInterface(riid, ppvObj);
}
```

Modeler::AddRef

```
uint32 Modeler::AddRef()
```

We just have to make sure that the TBasicModule method is called.

```
uint32 Modeler::AddRef()
{
    return TBasicModule::AddRef();
}
```

Modeler::Initialize

```
MCCOMErr Modeler::Initialize(IMCUnknown* inElement)
```

Initialize: We test the inElement argument. If it exists it has to be an I3DShGroup or an I3DShPrimitive. According to the type, we just initialize the universe or the primitive. This is done with the two protected methods: Initialize1 (for the universe) and Initialize2 (for the primitive).

```
MCCOMErr Modeler::Initialize(IMCUnknown* inElement)
{
    // Initialize is going to be called twice : once with the Universe, once with
       the Master Object
    if(!inElement) return MC_E_INVALIDARG;

    TMCCountedPtr<I3DShGroup> universe;
    if(inElement->QueryInterface(IID_I3DShGroup, (void**) &universe) == MC_S_OK)
    {
    return Initialize1(universe);
    }

    if(!fUniverse) return MC_E_INVALIDARG;

    TMCCountedPtr<I3DShPrimitive> primitive;
    if(inElement->QueryInterface(IID_I3DShPrimitive, (void**)&primitive) ==
       MC_S_OK)
    {
       return Initialize2(primitive);
    }

    return MC_E_INVALIDARG;
}
```

Modeler::Initialize1

```
MCCOMErr Modeler::Initialize1(I3DShGroup* universe)
```

Initialize1: If the universe exists then we equalize the fUniverse member to the universe argument

---

```
MCCOMErr Modeler::Initialize1(I3DShGroup* universe)
{
    // Initialization Call #1 : the Universe is passed

    if(!universe) return MC_E_INVALIDARG;
    fUniverse = universe;

    return MC_S_OK;
}
```

Modeler::Initialize2

```
MCCOMErr Modeler::Initialize2(I3DShPrimitive* primitive)
```

Initialize2: First of all we have to create a Window. This is the goal of the protected method CreateWindow (described below). Then we register ourselves as listeners of some channels. For this step we just need the TimeChangeChannel. Finally we keep a backup.

```
MCCOMErr Modeler::Initialize2(I3DShPrimitive* primitive)
{
    // Initialization Call #2 : the Master Object (primitive) is passed

    //-- Load windows and set up UI elements
    CreateWindows();

    //-- Register ourselves as listeners to some channels
    fScene->GetTimeChangeChannel(&fTimeChangeChannel);
    ThrowIfNil(fTimeChangeChannel);
    fTimeChangeChannel->RegisterListener(this);

    //-- Get target data
    fPrimitiveData = primitive;
    ThrowIfNil(fPrimitiveData);

    return MC_S_OK;
}
```

Modeler::Destroy

```
MCCOMErr Modeler::Destroy()
```

Destroy: As we have created a lot of things, we have to do the clean up when leaving. We destroy the window. If the hierarchy module exists we delete it. We delete all the protected menbers. We also need to unregister ourselves from the channel (here we just have to unregister from the TimeChangeChannel). Finally we call the Destroy method of the TBasicModule.

```
MCCOMErr Modeler::Destroy()
{
    DestroyWindows();

    if (fHierarchyModule)
        fHierarchyModule->Destroy();
    fHierarchyModule = NULL;

    //-- Clear everything
    fUniverse = NULL;
    fWindow = NULL;
```

```
    fPrimitiveData = NULL;
    fClonedComponent = NULL;

    //-- Deregister ourselves as listeners
    fTimeChangeChannel->UnregisterListener(this);
    fTimeChangeChannel = NULL;

    fPrimitiveData = NULL;// Will trigger a release

    return TBasicModule::Destroy();
}
```

Modeler::Hydrate

```
MCCOMErr Modeler::Hydrate()
```

Hydrate: This method allows us to prepare the data before an activation. This method is called when we enter the modeling room. Nonetheless, we hydrate ourselves but we also hydrate other modules (here we have to hydrate the hierarchy module).

```
MCCOMErr Modeler::Hydrate()
{
    TMCCountedPtr<IMFWindow> win = NULL;
    fWindow->QueryInterface(IID_IMFWindow, (void **)&win); ThrowIfNil(win);
    win->Show(true, true);
    gMenuUtilities->SetCurrentGlobalTool(kMoveToolID, true);

    //-- Hydrate other Modules
    if (fHierarchyModule)
        fHierarchyModule->Hydrate();
    return TBasicModule::Hydrate();
}
```

Modeler::Dehydrate

```
MCCOMErr Modeler::Dehydrate()
```

Dehydrate: This method is called when we leave the modeling room. We just free as many data as we can but the modeler still exists.

```
MCCOMErr Modeler::Dehydrate()
{
    //-- Dehydrate other Modules
    if (fHierarchyModule)
        fHierarchyModule->Dehydrate();

    //-- Hide window
    TMCCountedPtr<IMFWindow> win = NULL;
    fWindow->QueryInterface(IID_IMFWindow, (void **)&win); ThrowIfNil(win);
    win->Show(false, false);

    return TBasicModule::Dehydrate();
}
```

Modeler::Activate

```
MCCOMErr Modeler::Activate()
```

Activate: After hydrating the module we activate it. It allows us to make it appear on the screen. Once again

if other Modules are needed we activate them during this call.

```
MCCOMErr Modeler::Activate()
{
    //-- Select window
    TMCCountedPtr<IMFWindow> win;
    fWindow->QueryInterface(IID_IMFWindow, (void**) &win);ThrowIfNil(win);
    win->SelectWindow();

    //-- Activate other Modules
    if (fHierarchyModule)
        fHierarchyModule->Activate();
    return MC_S_OK;
}
```

Modeler::Deactivate

```
MCCOMErr Modeler::Deactivate()
```

Deactivate: Allows us to deactivate other modules. Then all modules disappear from the screen.

```
MCCOMErr Modeler::Deactivate()
{
    //-- Deactivate other Modules
    if (fHierarchyModule)
        fHierarchyModule->Deactivate();

       return MC_S_OK;
}
```

Modeler::CreateWindows

```
void  Modeler::CreateWindows()
```

CreateWindow: Called during Initialize2. We initialize protected menbers needed by the modeler. Then we create the window. Here we create it from the resources. Then we can initialize the fMainPart member.

```
void  Modeler::CreateWindows()
{
    QueryInterface(IID_I3DShModule, (void**) &fModule);
    ThrowIfNil(fModule);

    fDoc = fModule->GetDocumentNoAddRef();
    ThrowIfNil(fDoc);

    fDoc->QueryInterface(IID_ISceneDocument, (void**) &fSceneDoc);
    ThrowIfNil(fSceneDoc);

    fSceneDoc->GetScene(&fScene);
    ThrowIfNil(fScene);

    void* oldResources = NULL;
    gResourceUtilities->SetupComponentResources('modu', kModelerID, &old-
        Resources);

    fModule->CreateWindowByResource(&fWindow, kModelerView, true);
    ThrowIfNil(fWindow);
```

```
    fWindow->FindChildPartByID(&fMainPart, 'main');
    ThrowIfNil(fMainPart);

    gResourceUtilities->RestoreComponentResources(oldResources);
}
```

Modeler::DestroyWindows

```
void Modeler::DestroyWindows()
```

DestroyWindow: We need to free memory.

```
void Modeler::DestroyWindows()
{
    fModule = NULL;
    fDoc = NULL;
    fSceneDoc = NULL;
    fScene = NULL;
    fWindow = NULL;
    fMainPart = NULL;
}
```

Example 3: Modeler Step 3

Description

Functions

The third step:
In this step the modeler contains a view. In this view we represent the Tree primitive in 2D. We also need to handle the TreePrimitive in a different manner. In fact we need the modeler to be able to use the primitive data. That's why we create a new interface *ITreePrim*. Then our tree primitive inherits from it. We do the same for the view with an IModelerview. With these two interfaces, our modeler will be able to get the data it needs. As our primitive and our view support multiple inheritance, we need to code the QueryInterface and the AddRef method. We need a structure that contains data for drawing.
In order to have a view we need to create it as the primitive and the modeler. So we have to modify the MakeCOMObject method (in the TreePrim.cpp file).

```
TBasicUnknown* MakeCOMObject(const MCCLSID& classId)// This method instanciate
{                                                   // the objects for Modeler
    TBasicUnknown* res = NULL;

    if (classId == CLSID_COMTreePrim) res = new COMTreePrim;
    if (classId == CLSID_Modeler) res = new Modeler;
    if (classId == CLSID_ModelerView) res = new ModelerView;

    return res;
}
```

ModelerView.h

```
struct DrawTreePrivData
{
    IMCGraphicContext*fGC;
```

```
    int32        fNumView;
    real         fZoom;
};

TMCPoint TreePoint3Dto2D( TVector3 *v , int numView , float zoom );

// VIEW Object definition :
// VIEW Object :

class ModelerView : public TBasicPart,
   public IModelerView
{

    public :
        ModelerView();
        virtual MCCOMErr MCCOMAPI QueryInterface(const MCIID &riid, void**
        ppvObj);
        virtual uint32MCCOMAPI AddRef();
        STANDARD_RELEASE;

        virtual void MCCOMAPISelfDraw(IMCGraphicContext* graphicContext, const
        TMCRect& inZone);

        virtualModeler*MCCOMAPIGetModelerNoAddRef();
        virtualvoidMCCOMAPISetModeler(Modeler* TheModeler);
        virtualvoidMCCOMAPISetBackgroundColor(const TMCRGBColor& rgbcolor);

      protected:
        Modeler*  fTreeModeler;
        int              ViewSide;
        float        fZoom;
        TMCRGBColorfbackgroundColor;
};
```

Void ModelerView::SetBackgroundColor

```
void ModelerView::SetBackgroundColor( const TMCRGBColor &rgbcolor )
```

Description of the view methods
The view will draw the 2D presentation of the primitive.
First we set the background color.

```
void ModelerView::SetBackgroundColor( const TMCRGBColor &rgbcolor )
{
    if ( (fbackgroundColor.rr != rgbcolor.rr) || (fbackgroundColor.gg != rgb-
        color.gg) || (fbackgroundColor.bb != rgbcolor.bb))
    {
        fbackgroundColor = rgbcolor;
    }
}
```

Void ModelerView::SetModeler(Modeler* TheModeler)

```
voidModelerView::SetModeler( Modeler* TheModeler )
```

We set the modeler which uses the view.

```
voidModelerView::SetModeler( Modeler* TheModeler )
{
```

```
    fTreeModeler = TheModeler;
}
```

Modeler* ModelerView::GetModelerNoAddRef()

```
Modeler* ModelerView::GetModelerNoAddRef()
```

We can access the modeler.

```
Modeler* ModelerView::GetModelerNoAddRef()
{
    return fTreeModeler;
}
```

Void ModelerView::SelfDraw

```
void ModelerView::SelfDraw(IMCGraphicContext* graphicContext, const TMCRect&
       inZone)
```

The SelfDraw method will be called each time we need to redraw the view.

```
void ModelerView::SelfDraw(IMCGraphicContext* graphicContext, const TMCRect&
       inZone)
{
    PenColor.rr = 0;
    PenColor.gg = 0;
    PenColor.bb = 65535;

    graphicContext->FillRect(*(TMCRect*)&inZone , *(TMCRGBColor*)&fbackground-
       Color);
    graphicContext->SetPen( kCopyPen , kSolidPen , 1 , PenColor);

    if ( GetModelerNoAddRef() )
    {
        DrawTreePrivData PrivData;

        // Nbr branches total of the tree : ----------------------------
        TreeData treedata = GetModelerNoAddRef()->GetTreePrimNoAddRef()->Get-
        Data();
        long totalBranches = 1;
        int i;

        for(i=1;i<=treedata.fRecursiv;i++)
            totalBranches += pow((float)treedata.fNbBranches,i);

        PrivData.fGC = graphicContext;
        PrivData.fNumView = ViewSide;
        PrivData.fZoom    = fZoom;

        GetModelerNoAddRef()->GetTreePrimNoAddRef()->EnumFacets(DrawTree,(void *)
        &PrivData,0.0);
    }
}
```

Void DrawTree

```
void DrawTree( TFacet3D *facets, void *privData)
```

The DrawTree method is the following

```
---------------------------------------------------------------------------------------------------------------
void DrawTree( TFacet3D *facets, void *privData)
{
    IMCGraphicContext* fGC = (IMCGraphicContext*) ((DrawTreePrivData *)privData)-
        >fGC;
    int numView = ((DrawTreePrivData *)privData)->fNumView;
    float zoom  = ((DrawTreePrivData *)privData)->fZoom;
    TMCPoint pt0,pt1,pt2;

    pt0 = TreePoint3Dto2D( &(facets->fVertices[0].fVertex) , numView , zoom );
    pt1 = TreePoint3Dto2D( &(facets->fVertices[1].fVertex) , numView , zoom );
    pt2 = TreePoint3Dto2D( &(facets->fVertices[2].fVertex) , numView , zoom );

    if ( fGC )
    {
        fGC->DrawLine( pt0.x, pt0.y , pt1.x, pt1.y  );
        fGC->DrawLine( pt0.x, pt0.y , pt2.x, pt2.y  );
        fGC->DrawLine( pt2.x, pt2.y , pt1.x, pt1.y  );
    }
}
```

TMCPoint TreePoint3Dto2D

```
TMCPoint TreePoint3Dto2D( TVector3 *v , int numView , float zoom )
```

And here is the TreePoit3Dto2D method that allow us to specify the view number

---

```
TMCPoint TreePoint3Dto2D( TVector3 *v , int numView , float zoom )
{
    TMCPoint pt;
    int v1,v2;
    short z,xsign;

    switch ( numView )
    {
        case  1 :
            v1 = 0; v2 = 2; z=0;xsign = 1;
            break;
        case  3 :
            v1 = 0; v2 = 2; z=0;xsign = -1;
            break;
        case  6 :
            v1 = 1; v2 = 2; z=0;xsign = -1;
            break;
        case  4 :
            v1 = 1; v2 = 2; z=0;xsign = 1;
            break;
        case  8 :
            v1 = 0; v2 = 1; z=-100;xsign = 1;
            break;
        case  2 :
            v1 = 0; v2 = 1; z=-100;xsign = -1;
            break;
        default :
            v1 = 1; v2 = 2; z=0;xsign = -1;
            break;
    }

    pt.x = xsign*(short)((*v)[v1] * zoom  ) + 300   ;
    pt.y =      -(short)((*v)[v2] * zoom  ) + 300 +z;
```

```
    return pt;
}
```

Void Modeler::CreateWindows

```
void  Modeler::CreateWindows()
```

In the modeler we now have to create the view during the window creation. We also have to set the our modeler as the view modeler. We need to modify the Modeler.cpp file:

```
void  Modeler::CreateWindows()
{
    QueryInterface(IID_I3DShModule, (void**) &fModule);
    ThrowIfNil(fModule);

    fDoc = fModule->GetDocumentNoAddRef();
    ThrowIfNil(fDoc);

    fDoc->QueryInterface(IID_ISceneDocument, (void**) &fSceneDoc);
    ThrowIfNil(fSceneDoc);

    fSceneDoc->GetScene(&fScene);
    ThrowIfNil(fScene);

    void* oldResources = NULL;
    gResourceUtilities->SetupComponentResources('modu', kModelerID, &old-
       Resources);

    fModule->CreateWindowByResource(&fWindow, kModelerView, true);
    ThrowIfNil(fWindow);

    fWindow->FindChildPartByID(&fMainPart, 'MODv');

    fMainPart->QueryInterface(IID_IModelerView, (void**) &fModelerPart);
    ThrowIfNil(fModelerPart);

    fModelerPart->SetModeler(this);

    gResourceUtilities->RestoreComponentResources(oldResources);
}
```

Example 4: Modeler Step 4

Description

Functions

The fouth step:
The last step is a nice one. We now allow the user to interact with the primitive trough the modeler. A lot of things must be done before that. But if we start from the step 3 it will not not such a great deal.
Let's begin with the primitive.
We want it to be animatable and savable. We need to change a few things in the data structure.

```
class TreeData : public TTimeBased
{
```

```
    public :
        TreeData();
        ~TreeData();

        virtual uint32 MCCOMAPI AddRef();
        virtual uint32 MCCOMAPI Release();

        int16fNbBranches;// Number of branches of the 3D Tree
        floatfAngle;  // Angle for branches
        int16fRecursiv;// Recursivity number for tree construction

    protected :
        virtual void RegisterParams();

    private :
        uint32    fRefCount;
};


// Tree Object definition :
// TreePrimitive Object :

class COMTreePrim : public TBasicPrimitive,
                    public I3DExAnimated,
                    public IExStreamIO,
                    public ICOMTreePrim
{
    public :
        COMTreePrim();

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

        // IExtension
        virtual void MCCOMAPI Clone(IExtension** res, IMCUnknown* pUnkOuter);

        // IExDataExchanger methods :
        virtual void* MCCOMAPIGetExtensionDataBuffer();
        virtual MCCOMErr MCCOMAPIExtensionDataChanged();
        virtual int16 MCCOMAPIGetResID();

        // I3DExGeometricPrimitiv methods
        // Geometric Calls
        virtual MCCOMErr MCCOMAPIGetBBox( TBox3D* bbox);
        virtual MCCOMErr MCCOMAPI    EnumFacets( EnumFacetsCallback callback,
        void* privData,real fidelity);
        virtual MCCOMErrMCCOMAPIEnumPatches( EnumPatchesCallback callback, void*
        privData);
        virtual MCCOMErr    MCCOMAPIGetNbrLOD(int16 &nbrLod);
        virtual MCCOMErr    MCCOMAPIGetLOD(int16 lodIndex,real &lod);
        virtual FacetMesh*  MCCOMAPIGetFacetMesh(uint32 index); // index is the
        lod number
        virtual FacetMesh*  MCCOMAPIGetFacetMesh(real  lod);   // lod = level of
        detail
        virtual boolean     MCCOMAPI    CanBeSplit();
        virtual MCCOMErr    MCCOMAPI    SplitPrimitive(TMCCountedPtrAr-
        ray<I3DExGeometricPrimitive>& subParts, TMCArray<TTransform3D>& subPart-
        Positions);
```

```
        // Shading Calls
        virtual uint32MCCOMAPIGetUVSpaceCount();
        virtual MCCOMErr MCCOMAPI    GetUVSpace( uint32 uvSpaceID, UVSpaceInfo*
        uvSpaceInfo);
        virtual MCCOMErr    MCCOMAPIUV2XYZ(TVector2* uv, uint32 uvSpaceID,
        TVector3* resultPosition, boolean* inUVSpace); // Optional - Return
        E_NOTIMPL if not implemented
        virtual MCCOMErr    MCCOMAPIGetUVSpaceRDS5(uint32 uvSpaceID,
        UVSpaceInfoRDS5* uvSpaceInfoRDS5); // To be implemented only if your prim-
        itive existed in RDS 5 or earlier, so that UV spaces can be mapped prop-
        erly to the new 0..1 boundaries
        virtual MCCOMErrMCCOMAPIAppendToRenderables(
                                              const TTransform3D& worldFromMod-
        elTfm,
                                              TRenderableAndTfmArray& render-
        ableAndTfm );


        virtual booleanMCCOMAPISetData(const TreeData& data);
        virtual TreeDataMCCOMAPIGetData();

        virtual floatMCCOMAPIGetAngleAndPoint( TMCPoint *pt, TMCPoint *pt0, int
        numView, float zoom );
        virtualbooleanMCCOMAPIGetIfDataChanged();

        // I3DExAnimated calls
        virtual MCCOMErr MCCOMAPI RegisterParams();
        virtual MCCOMErr MCCOMAPI InvalidateCaches(int32 itsID);
        virtual MCCOMErr MCCOMAPI CopyTimeData(IMCUnknown *dest);

        // IExStreamIO calls
        virtual MCCOMErr MCCOMAPI Read(IShTokenStream* stream, ReadAttributeProc
        readUnknown, void* privData); // readUnknown should be called by extention
        if unknown keyword is encountered (instead of calling SkipTokenData)
        virtual MCCOMErr MCCOMAPI Write(IShTokenStream* stream);

        void CopyData(COMTreePrim* dest) const;

    protected:
        TreeDatafData;       // Tree Data
        MCCOMErrUpdateBoundingBox( TVector3 *Coordinate );
        TVector3Rotate( TVector3 &Vertex , TVector3 &angle );
        TVector3Normal( TVertex3D*fVertices );

        TVector3Cube(EnumFacetsCallback callback, void* privData  , float SizeZ ,
        float SizeXY , TVector3 &angle , TVector3 &translate);
        MCCOMErrGetFacetsTree(EnumFacetsCallback callback, void* privData, int
        level , TVector3 angle , TVector3 translate );

        TBox3D fTreeBox;   // BoundingBox
        booleanfDataChanged;// true if Datas changed

    private :
        friend class TreeData;
        TMCCountedPtr<I3DShTimeBasedData> fTimeBasedData;
};
```

We also need to modify the modeler in order to establish communication. In fact we want it to exchange data

with the properties palette and the tree hierarchy.

```
---------------------------------------------------------------------------------------------------------------
class Modeler : public TBasicModule
{
    public :
        Modeler();
        STANDARD_RELEASE;

        // IMCUnknown methods
        virtual MCCOMErrMCCOMAPI QueryInterface(const MCIID& riid, void**
        ppvObj);
        virtual uint32MCCOMAPI AddRef();

        virtual MCCOMErr MCCOMAPI Initialize(IMCUnknown* inElement);
        virtual MCCOMErr MCCOMAPI Destroy();
        virtual MCCOMErr MCCOMAPI Hydrate();
        virtual MCCOMErr MCCOMAPI Dehydrate();

        virtual MCCOMErrMCCOMAPI Activate();
        virtual MCCOMErrMCCOMAPI Deactivate();

        virtual voidMCCOMAPI DataChanged(IChangeChannel*channel,
                                         IDType      changeKind,
                                         IMCUnknown*changedData);

        ICOMTreePrim*MCCOMAPI GetTreePrimNoAddRef(){return fCOMTreePrim;};
        I3DShPrimitive*MCCOMAPI GetI3DShPrimNoAddRef(){return fPrimitiveData;};
        IModelerView*MCCOMAPI GetPartNoAddRef(){return fModelerPart;};
        IMFPart*  MCCOMAPI GetIMFPartNoAddRef(){return fMainPart;};
        IMFDocument*MCCOMAPI GetDocNoAddRef(){return fDoc;}

        virtual voidMCCOMAPI SelfPrepareMenus();
        virtual booleanMCCOMAPI SelfMenuAction(ActionNumber actionNumber);

        virtual voidMCCOMAPI GetSelection(ISceneSelection**outSelection);
        virtual voidMCCOMAPI GetSelectionChannel(IChangeChannel**outChannel);

        void GetActionResponder(IMFResponder** responder);

        //-- Update Stuff
        void BeginImmediateUpdate();
        void PostImmediateUpdate();
        void EndImmediateUpdate();
        void ImmediateUpdate();

    protected :

        TMCCountedPtr<I3DShModule>fModule;
        TMCCountedPtr<IMFDocument>fDoc;
        TMCCountedPtr<ISceneDocument>fSceneDoc;
        TMCCountedPtr<I3DShScene>fScene;
        TMCCountedPtr<I3DShGroup>fUniverse;
        TMCCountedPtr<IMFPart>fWindow;
        TMCCountedPtr<IMFPart>fMainPart;
        TMCCountedPtr<I3DShModule>fHierarchyModule;

        //-- Dependencies data
        TMCCountedPtr<IChangeChannel>fSelectionChannel;
        TMCCountedPtr<IChangeChannel>fTimeChangeChannel;
        TMCCountedPtr<IChangeChannel>fTreePropertyChangeChannel;
        TMCCountedPtr<ISceneSelection>fSelection;
```

```
        TMCCountedPtr<IChangeChannel>fSelectionChangeChannel;
        TMCCountedPtr<IChangeChannel>fTreeHierarchyChannel;
        boolean                          fImmediateUpdate;


        //-- Primitive Data
        TMCCountedPtr<I3DShPrimitive>fPrimitiveData;
        TMCCountedPtr<ICOMTreePrim>fCOMTreePrim;
        TMCCountedPtr<IModelerView>fModelerPart;
        TMCCountedPtr<IComponentAnim>fClonedComponent;

        TMCCountedPtr<I3DShModule>fPropertiesModule;
        TMCCountedPtr<IChangeChannel>fImmediateUpdateChannel;

        virtual MCCOMErrMCCOMAPI Initialize1(I3DShGroup* universe);
        virtual MCCOMErrMCCOMAPI Initialize2(I3DShPrimitive* primitive);
        virtual voidMCCOMAPI CreateWindows();
        virtual voidMCCOMAPI DestroyWindows();
};
```

When we want interaction with our modeler we have to handle actions. So we need a standard action (ModelerAction) and a mouse action (ModelerMouseAction).

```
class ModelerAction : public TBasicAction
{

   protected:

        ModelerAction(TreeData& treedata, Modeler* modeler);

   public:

        static void Create(IShAction**, TreeData& treedata, Modeler* modeler);
        STANDARD_RELEASE;

        virtual MCCOMErr MCCOMAPI Do();
        virtual MCCOMErr MCCOMAPI Undo();
        virtual MCCOMErr MCCOMAPI Redo();
        virtual boolean  MCCOMAPI WillCauseChange();
        virtual boolean  MCCOMAPI CanUndo();
        virtual MCCOMErr MCCOMAPI GetName(TMCString& name);
        virtual void MCCOMAPI GetPartToRedraw(IMFPart** outPart, int32 stage);

   protected:
        TreeDatafOldTreeData,fNewTreeData;
        Modeler*fModeler;
};

class ModelerMouseAction : public TBasicMouseAction
{
   protected:

        ModelerMouseAction(TreeData& treedata ,Modeler* modeler );

   public:

        static void Create(IShMouseAction** outAction, TreeData& treedata, Mod-
        eler* modeler);
        STANDARD_RELEASE;
```

```
        virtual MCCOMErr MCCOMAPI Do();
        virtual MCCOMErr MCCOMAPI Undo();
        virtual MCCOMErr MCCOMAPI Redo();
        virtual boolean  MCCOMAPI WillCauseChange();

        virtual boolean  MCCOMAPI CanUndo();
        virtual void MCCOMAPI GetPartToRedraw(IMFPart** outPart, int32 stage);

        virtual voidMCCOMAPI Track(IMCGraphicContext* gc, int16 stage, TMCPoint&
        first, TMCPoint &prev, TMCPoint &cur,boolean moved, IShMouseAction**nex-
        tAction);
        virtual MCCOMErrMCCOMAPI Feedback(IMCGraphicContext* gc, int16
        stage,const TMCPoint& first,const TMCPoint &prev,const TMCPoint &cur,bool-
        ean moved,boolean show);
        virtual MCCOMErrMCCOMAPI Constrain(IMCGraphicContext* gc, int16
        stage,const TMCPoint& first,const TMCPoint &prev,TMCPoint &cur,boolean
        moved);
        virtualModeler*MCCOMAPI GetModelerNoAddRef(){ return fModeler;};

    protected:
        TreeDatafOldTreeData,fNewTreeData;
        Modeler*fModeler;

        int          fActionNumber,fNumView;
        float     fZoom,fSignAngle;
        TMCPointfContactPoint,fCurrentPoint;
        boolean   fDataChanged;
};
```

We also create a properties palette for our modeler

```
classModelerProp :public TBasicUnknown,
                        public IChangeListener,
                        public IPropertiesClient
{
    public:
        ModelerProp(
                        IPropertiesModule* inPropertiesModule,
                        Modeler* modeler);
        ~ModelerProp();

        // IMCUnknown methods
        MCCOMErrMCCOMAPI QueryInterface(const MCIID& riid, void** ppvObj);
        uint32 MCCOMAPI AddRef ();
        STANDARD_RELEASE;

        // IChangeListener methods
        void   MCCOMAPI DataChanged(IChangeChannel* channel,
                                            IDType   changeKind,
                                            IMCUnknown* changedData);
        // IPropertiesClient methods
        void   MCCOMAPIGetSelection(ISceneSelection**outSelection);
        void   MCCOMAPIGetSelectionChannel(IChangeChannel**outChannel);
        IDType MCCOMAPIGetControllingModuleID();
        ResourceIDMCCOMAPIGetPropResID(ISceneSelection* inSelection);

        void   MCCOMAPILoadPageData(IMFPart* inTopPart,
                                            ISceneSelection*inSelection);

        booleanMCCOMAPIHandlePageHit(IMFPart*inTopPart,
                                            ISceneSelection*inSelection,
```

```
                                                    int32       inMessage,
                                                    IMFResponder*inResponder,
                                                    void*       inData);

            void   MCCOMAPIGetExtraPart(IMFPart**outExtraPart);

        protected:

            TMCCountedPtr<IPropertiesModule>fPropertiesModule;
            TMCCountedPtr<Modeler>fModeler;

            // channels
            TMCCountedPtr<IChangeChannel>fImmediateUpdateChannel;
    };
```

MCCOMErr COMTreePrim::RegisterParams

```
MCCOMErr COMTreePrim::RegisterParams()
```

The tree primitive:
Detail of the new methods. These methods support animation and load/save ability. For the animation we need the following methods:

```
MCCOMErr COMTreePrim::RegisterParams()
{
    TMCCountedPtr<ITimeBased> timeBased;
    QueryInterface(IID_ITimeBased, (void**)&timeBased);
    ThrowIfNil(timeBased);
    fData.SetAnimated(true);
    timeBased->GetTimeBasedData(&fTimeBasedData);
    gTreeIndex = fTimeBasedData->SetParamGroup(&fData, "Tree***", 'Tree');
    return MC_S_OK;
}
```

MCCOMErr COMTreePrim::InvalidateCaches

```
MCCOMErr COMTreePrim::InvalidateCaches(int32 itsID)

MCCOMErr COMTreePrim::InvalidateCaches(int32 itsID)
{
    return 0;
}
```

void COMTreePrim::Clone

```
void COMTreePrim::Clone(IExtension** res, IMCUnknown* pUnkOuter)

void COMTreePrim::Clone(IExtension** res, IMCUnknown* pUnkOuter)
{
    TMCCountedCreateHelper<IExtension>result(res);

    COMTreePrim* theClone = new COMTreePrim();
    ThrowIfNil(theClone);
    theClone->SetControllingUnknown(pUnkOuter);
    CopyData(theClone);

    result = (IExtension*) theClone;
}
```

void COMTreePrim::CopyData

```
void COMTreePrim::CopyData(COMTreePrim* dest) const

void COMTreePrim::CopyData(COMTreePrim* dest) const
{
    dest->fData.fNbBranches = fData.fNbBranches;
    dest->fData.fAngle = fData.fAngle;
    dest->fData.fRecursiv = fData.fRecursiv;
}
```

MCCOMErr COMTreePrim::CopyTimeData

```
MCCOMErr COMTreePrim::CopyTimeData(IMCUnknown* dest)

MCCOMErr COMTreePrim::CopyTimeData(IMCUnknown* dest)
{
    // Needs the IExDataExchanger to get the internal data.
    TMCCountedPtr<IExDataExchanger> i3DDataExchanger;
    dest->QueryInterface(IID_IExDataExchanger, (void**) &i3DDataExchanger);
    ThrowIfNil(i3DDataExchanger);

    // 3. Copy the data
    fData.CopyTimeData(kWithAnim, (TreeData*) i3DDataExchanger->GetExtension-
        DataBuffer() );

    return MC_S_OK;
}
```

MCCOMErr COMTreePrim::Write

```
MCCOMErr COMTreePrim::Write(IShTokenStream* stream)
```

To be able to load and save our primitive we need two other methods

---

```
MCCOMErr COMTreePrim::Write(IShTokenStream* stream)
{
    // We write our attributes in this order Angle, Branche nbr and Step

    stream->Indent();
    stream->PutKeyword('Angl');
    stream->PutQuickFix(fData.fAngle);

    stream->Indent();
    stream->PutKeyword('Bran');
    stream->PutLong(fData.fNbBranches);

    stream->Indent();
    stream->PutKeyword('Step');
    stream->PutLong(fData.fRecursiv);

    gShell3DUtilities->WriteTimeBased(stream, IDTYPE('T','r','e','e'), &fData);

    return MC_S_OK;
}
```

MCCOMErr COMTreePrim::Read

```
MCCOMErr COMTreePrim::Read(IShTokenStream* stream, ReadAttributeProc readUn-
       known, void* privData)
```

```
MCCOMErr COMTreePrim::Read(IShTokenStream* stream, ReadAttributeProc readUn-
       known, void* privData)
{
   // the keyword has been read
   int8 token[256];
   real* Angle = 0;
   int32* Branches = 0;
   int32* Recurs = 0;

   stream->GetBegin();

   try
   {
      do
      {
         int16 err=stream->GetNextToken(token);
         if(err!=0) throw TMCException(err,0);

         if (!stream->IsEndToken(token))
         {
            int32 keyword;

            stream->CompactAttribute(token, &keyword);

            switch (keyword)
            {
               case IDTYPE('T','r','e','e'):
                  gShell3DUtilities->ReadTimeBased(stream, &fData);
                  break;
               case IDTYPE('A','n','g','l'):
                  stream->GetQuickFix(&fData.fAngle);
                  break;
               case IDTYPE('B', 'r', 'a', 'n'):
                   stream->GetLong((int32*)&fData.fNbBranches);
                  break;
               case IDTYPE('S', 't', 'e', 'p'):
                  stream->GetLong((int32*)&fData.fRecursiv);
                  break;
               default:
                  readUnknown(keyword, stream, privData);
            }

         }
      }
      while (!stream->IsEndToken(token));
   }
   catch(...)
   {
      throw;
   }

   return MC_S_OK;
}
```

Writing a Light Source

Family ID : 'lite'
Interface ID : IID_I3DExLightSource
Interface file : I3dExLight.h
Basic Implementation files: BasicCameraLightGel.h, BasicCameraLightGel.cpp

Overview

A light source creates light within a 3D scene. A light source extension has complete control of the light intensity returned to the 3D Shell. A light can be finite, as in a spot light, or infinite, as in a distance light source.
I3dExLight has the following methods:

- **SetTransform(TTransform3D* transform)**

SetTransform() is called prior to any of the other methods. It can be used for pre-processing, and for assigning the light transforms to a private variable.

- **GetDirection(const TVector3 &point,TVector3 &resultDirection,real &resultDistance)**

GetDirection() receives a 3D point, called *point*, on the surface of an object. It is expected to return the distance between the point and the center of the light source in *resultDistance,* and the direction vector between the point and the light source in *resultDirection*.

- **GetColor(const TVector3 &point,const TVector3 &direction,const real distance,TMCColorRGB &result,boolean &callForShadowEffect)**

In **GetColor()** the specifics of the light extension are implemented. A 3D point, called *point*, contains a point on the object's surface. **GetColor()** should return true if this point is illuminated by the light, and false otherwise. The variable *direction* and *distance* contain a direction vector between the point and the light source, and the distance between the point and the light source. The variable *result* is used to return the final color for the point as calculated by the light extension. The boolean variable *callForShadowEffect* should return true if the light casts shadows, so that **ShadowEffect()** will be called during the calculation of shadows.

- **IsVisibleInPerspective()**

IsVisibleInPerspective() returns true for lights which are visible in the perspective window. Lights which are not visible in the perspective window, such as infinite lights, return false.

- **ShadowEffect(real distance,TMCColorRGB &result)**

ShadowEffect() is used to return the color of shadows in *result* based on the distance between the point on the object's surface and the light source, as given in the variable *distance*.

- **ForEachShadowBuffer(ForEachShadowBufferCallback proc,void* priv)**

ForEachShadowBuffer() is implemented for lights which support Shadow Buffering, as opposed to Ray-Traced shadows.

- **GetLightInfo(boolean &hasLightCone,boolean &hasLightSphere,real &halfAngle,TTransform3D &transform)**

This is used to define several flags for the light source, particularly for lights similar to the spot light.

- **GetStandardLight(TStandardLight& light)**

GetStandardLight() is an optional method used by the interactive renderer for a quick implementation of the light.

Location in the User Interface

Lights are added to a scene in the Assemble room from the Insert menu. If an Icon is provided as a UImg resource, a button will be added to the toolbar as well. A light's properties are edited in the Properties Tray.

Example: Beams Light

Description

This example creates a multi-spot or beams light. An example of the output of this light is below.

This picture was made with only one BeamsLight at the center of the scene.

Parameters

The following variables are used for the user-input parameters:

```
fHorApertureAngle:    The Horizontal Angular Limit (-45 to 45).
fVerApertureAngle:    The Vertical Angular Limit (-45 to 45).
fIntensity:           Light Source Intensity.
fNbBeamsHorizontally:Number of Beams Horizontally.
fNbBeamsVertically:   Number of Beams Vertically.
fLightColor:          The color of the light.
fBeamAperture:        Angular limit of each individual beam.
```

These are defined in Light.h as follows:

```
// Data storage of our extension :
struct LightData
{
    int16        fHorApertureAngle;    // Angular Limits of the light source
    int16        fVerApertureAngle;    //
    real         fIntensity;          // Light source intensity
    int16        fNbBeamsHorizontally; // # of Beams Horiz. and Vert.
    int16        fNbBeamsVertically;   //
    TMCColorRGBA fLightColor;         // Default color
    int16        fBeamAperture;        // Angular Limit of the singular Beam
};
```

These are then mapped to the user interface parameters using the PMap resource in Light.r.
Note that the parameters within the *LightData* struct must be in the same order as in the PMap.
There are four functions in this example which are not methods of *I3DExLightSource:* **LocalToGlobal()**, **GlobalToLocal()**, **LocalToGlobalVector()**, and **GlobalToLocalVector()**. They are used by **GetColor()** to convert *direction* from global to local coordinates.

Functions

Light::SetTransform

```
MCCOMErr Light::SetTransform(TTransform3D* transform)
```

SetTransform() is called prior to any other methods. In this example it is implemented to copy the transformations into the private variable *fTransform*:

```
MCCOMErr Light::SetTransform(TTransform3D* transform)
{
   fTransform=*transform;
   return MC_S_OK;
}
```

Light::IsVisibleInPerspective

```
boolean Light::IsVisibleInPerspective()
```

IsVisibleInPerspective() is used to indicate if a light is visible in the 3D view. Infinite light sources, such as the distant light, would return false. Lights such as bulb lights or spot lights return true.

```
boolean Light::IsVisibleInPerspective()
{
  return true; // the source is not a distant light (like the sun)
               // so it can be in the 3D perspective display.
}
```

Light::GetDirection

```
MCCOMErr Light::GetDirection(const TVector3 &point,TVector3 &resultDirec-
      tion,real &resultDistance)
```

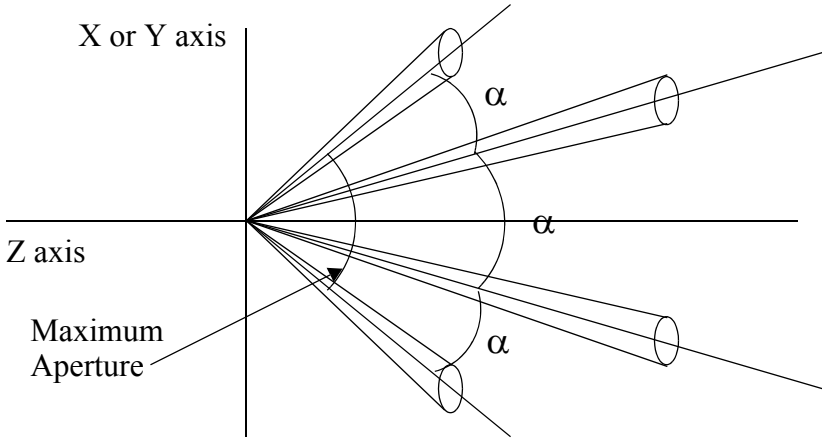
GetDirection() must calculate a vector from the lightsource center to the 3D position on the object's surface passed to the variable *point*. The vector is returned in *resultDirection*, and the length is returned in *resultDistance*.

```
MCCOMErr Light::GetDirection(const TVector3 &point,TVector3 &resultDirec-
      tion,real &resultDistance)
{
   (resultDirection)[0]=fTransform.fTranslation[0]-(point)[0];
   (resultDirection)[1]=fTransform.fTranslation[1]-(point)[1];
   (resultDirection)[2]=fTransform.fTranslation[2]-(point)[2]; // Vector from
      surface point to the Light
   resultDistance=sqrt((resultDirection)[0]*(resultDirection)[0] + (resultDi-
      rection)[1]*(resultDirection)[1]
               +(resultDirection)[2]*(resultDirection)[2]);
   (resultDirection)[0] /= resultDistance;
   (resultDirection)[1] /= resultDistance;
   (resultDirection)[2] /= resultDistance;
   return MC_S_OK;
}
```

Light::GetColor

```
boolean Light::GetColor(const TVector3 &point,const TVector3 &direction,const
      real distance,TMCColorRGB &result,boolean &callForShadowEffect)
```

GetColor() implements the specifics of the light extension.
For the BeamsLight, regularly spaced beams are created as illustrated below.



This figure shows how the BeamsLight will illuminate the scene. Each beam will be separated by the angle:

$$\alpha = \frac{MaxAperture}{NbBeams - 1}$$

If the number of beams is equal to one, the beam will be on the Z-axis as theta or phi will be equal to zero.

The local coordinate of the direction vector is calculated. After using **GlobalToLocalVector()** for conversion, the beam index for the Horizontal and Vertical beams can be calculated:

$$Index_{Hor} = NearestIntegerOf\left(\left(\frac{\theta}{MaxAperture} + \frac{1}{2}\right) \times (NbBeams + 1)\right)$$

With both the Horizontal and Vertical index, the nearest direction beam is found with these formulae:

$$\theta = MaxApertureHor \times \left(\frac{Index_{Hor}}{NbBeamsHor - 1} - \frac{1}{2}\right)$$

$$\varphi = MaxApertureVer \times \left(\frac{Index_{Ver}}{NbBeamsVer - 1} - \frac{1}{2}\right)$$

$$V = \begin{cases} x = \sin\theta \cdot \cos\varphi \\ y = \sin\varphi \\ z = \cos\theta \cdot \cos\varphi \end{cases}$$

Once the two vectors are in the same coordinates system, the dot product is used to calculate the angle between the main direction of the beam and the given direction:

$$\cos(AngleBeamDirection) = \vec{D} \cdot \vec{V}$$

To determine if the given direction is in the light beam, a comparison is made between half of the cosine of the beam aperture and the cosine of *AngleBeamDirection*. If the latter is larger, then the point is in the beam and the light illuminates the point.
The *callForShadowEffect* value is returned as true to tell the renderer that the light source creates shadows, and that the renderer must call **ShadowEffect()**.
GetColor() implements the equations above to create the light. It also returns TRUE if the light illuminates the point, and FALSE otherwise:

```
boolean Light::GetColor(const TVector3 &point,const TVector3 &direction,const
       real distance,TMCColorRGB &result,boolean &callForShadowEffect)
{
    TVector3 localVector,refDir;
    real    angle,nearest_dir,hor,ver,theta,phi,r;
    real    costheta,sintheta,cosphi,sinphi;
    real    cosdifferentialAngle;
    real    angleLimit=360.0;

    callForShadowEffect=true; // We always want shadows for this lightsource

    GlobalToLocalVector(&fTransform,direction,&localVector);

    // initialize result to default color (intensity included)
    result=fData.fLightColor;
    result.R*=fData.fIntensity;
    result.G*=fData.fIntensity;
    result.B*=fData.fIntensity;

    // Nearest direction vector determination

    if (fData.fNbBeamsHorizontally!=1)
    {
        // direction is calculated in the Spherical Coordinates (see Spheric Cam-
        era)
        // -- Horizontal Determination
        angle=atan2(localVector[0],localVector[2])*180.0/3.14159265358979323846;
        //QuickArcSinCos(localVector[0],localVector[2],angle);
        if (angle>angleLimit/2.0)
        {
            angle-=angleLimit;
        }
        if ((angle>fHorAng+fBeamAngle)||(angle<-fHorAng-fBeamAngle))
            return false; // the point is outside the maximum horizontal aperture

        nearest_dir=(angle/(fHorAng*2.0)+0.5)*((real)fData.fNbBeamsHorizontally-
        1.0);
        hor=floor(nearest_dir);
        if (nearest_dir-hor>0.5)
            hor+=1.0;
        if (hor>((real)fData.fNbBeamsHorizontally-1.0))
            hor=((real)fData.fNbBeamsHorizontally-1.0);
        if (hor<0.0)
            hor=0.0;
        // hor must be between 0 and fNbBeamsHorizontally-1
        theta=fHorAng*2.0*(hor/((real)fData.fNbBeamsHorizontally-1)-0.5);
    }
    else
    {
        theta=0.0;
    }
    if (fData.fNbBeamsVertically!=1)
    {
```

```
            // -- Vertical Determination
            r=localVector[0]*localVector[0]+localVector[2]*localVector[2];
            r=sqrt(r);
            angle=atan2(localVector[1],r)*180.0/3.14159265358979323846; //QuickArcS-
            inCos(localVector[1],r,angle);
            if (angle>(angleLimit/2.0))
            {
               angle-=angleLimit;
            }
            if ((angle>fVerAng+fBeamAngle)||(angle<-fVerAng-fBeamAngle))
               return false; // the point is outside the maximum vertical aperture

            nearest_dir=(angle/(fVerAng*2.0)+0.5)*((real)fData.fNbBeamsVertically-
            1.0);
            ver=floor(nearest_dir);
            if (nearest_dir-ver>0.5)
               ver+=1.0;
            if (ver>((real)fData.fNbBeamsVertically-1.0))
               ver=((real)fData.fNbBeamsVertically-1.0);
            if (ver<0.0)
               ver=0.0;
            // ver must be between 0 and fNbBeamsVertically-1
            phi=(fVerAng*2.0)*(ver/((real)fData.fNbBeamsVertically-1.0)-0.5);
         }
         else
         {
            phi=0.0;
         }
         // nearest direction vector

         // Spherical Coordinates of the direction vector
         sintheta=sin(theta/180.0*3.14159265358979323846);
         costheta=cos(theta/180.0*3.14159265358979323846); //QuickSinCos(theta,sin-
            theta,costheta);
         sinphi=sin(phi/180.0*3.14159265358979323846);
         cosphi=cos(phi/180.0*3.14159265358979323846); //QuickSinCos(phi,sin-
            phi,cosphi);
         refDir[0]=sintheta*cosphi;
         refDir[1]=sinphi;
         refDir[2]=costheta*cosphi;
         // Direction comparaison
         cosdifferentialAngle=refDir[0]*localVector[0] +    // dot product to get the
            cosinus of the angle
                              refDir[1]*localVector[1] +
                                          refDir[2]*localVector[2];
         if (cosdifferentialAngle<fBeamLimit)
            return false; // the point is outside the Beam
         return true;
      }
```

Light::ShadowEffect

```
MCCOMErr Light::ShadowEffect(real distance,TMCColorRGB &result)
```

The Beams Light creates pure black shadows, so 0 is returned for the RGB channels in the variable *result*.

```
MCCOMErr Light::ShadowEffect(real distance,TMCColorRGB &result)
{
   result.R=0.0;
   result.G=0.0;
   result.B=0.0;
   return MC_S_OK;
}
```

Writing a Post Render Filter

Family ID : 'post'
Interface ID : IID_ I3DExPostRenderer
Interface file : I3DExPostRenderer.h
Basic Implementation file: BasicPostRenderer.h, BasicPostRenderer.cpp

Overview

A Post Render Filter operates on the pixels of an image after rendering. In addition to the pixel color data, G-Buffers are provided with information about the distance, XYZ position, XYZ normals, Surface UV, Alpha Channel, Diffuse color values, and more. See the Reference manual section on the *I3DExPostRenderer* interface for a complete list.
With the G-Buffers, a wide variety of effects can be achieved. For example, the distance channel is used by the Depth of Field settings to simulate camera focus as a post render effect.
I3DExPostRenderer has the following methods:

- **GetBufferNeeds(uint32 &needs, void* renderer)**

This method indicates which G-Buffers will be needed by the extension.

- **PrepareDraw(IShRasterLayer *input,const TMCRect &bounds,IShChannel *buffers[],I3DShScene *scene,TBox2D &uvBox)**

PrepareDraw() is called prior to post-rendering.

- **DrawRect(const TMCRect &outputRect,const TChannelDataBucket* output[])**

DrawRect() is where the desired effect is implemented.

- **FinishDraw()**

FinishDraw() is used for any needed cleanup.

Location in the Interface

Post Render Filters are found on the Properties tray, under the filters tab, when the scene is selected from the object list.

Example 1: Sand

Description

Sand is a simple example which manipulates the RGB data of each pixel. The value of each channel is passed to a function, which generates a random value between 0 and the maximum value of each channel - 1. If the random value is greater than or equal to the original, then the channel is turned off. Otherwise, it is set to its maximum value of 16,384. The result is a 'sandy' effect.
No G-Buffers are needed, and there are no UI parameters.

Functions

Sand::GetBufferNeeds

```
void Sand::GetBufferNeeds(uint32 &needs, void *renderer)
```

GetBufferNeeds() is used to indicate which G-Buffers are needed. None are used in this example:

```
void Sand::GetBufferNeeds(uint32 &needs, void *renderer)
{
   //No G-Buffers are needed for this example
   needs = 0;

}
```

Note: the second example needs G-Buffers.

Sand::PrepareDraw

```
MCCOMErr Sand::PrepareDraw(IShRasterLayer* input, const TMCRect& bounds,
      IShChannel* buffers[], I3DShScene* scene, I3DShCamera* renderingCamera,
      const TBBox2D& productionFrame)
```

The next function is **PrepareDraw()**. This function is called prior to **DrawRect()**. In this example, the private variable *fColorOffscreen* is assigned to the input:

```
MCCOMErr Sand::PrepareDraw(IShRasterLayer* input,
                       const TMCRect& bounds,
                       IShChannel* buffers[],
                       I3DShScene* scene,
                       I3DShCamera* renderingCamera,
                       const TBBox2D& productionFrame)
{
   //fColorOffScreen is assigned to the input buffer
   fColorOffscreen=input;

   return MC_S_OK;
}
```

Sand::DrawRect

```
MMCCOMErr Sand::DrawRect(const TMCRect& outputRect, const TChannelDataBucket*
      output[], const TBBox2D &uvBox)
```

The third function is where the output takes place. In **DrawRect()**, each RGB channel is passed through the **RandomColor()** function to turn the channel on or off:

```
MCCOMErr Sand::DrawRect(const TMCRect& outputRect, const TChannelDataBucket*
      output[], const TBBox2D &uvBox)
{
   const TChannelDataBucket *color[3];
   uint16 index[3]={0,1,2};
   fColorOffscreen->GetTile(outputRect,color,index,3,eTileRead);

   int32 x,y;
   int32 width=outputRect.GetWidth();
   int32 height=outputRect.GetHeight();

   for (y=0;y<height;y++)
   {
      uint16 *newRed=(uint16*)output[0]->RowPtr16(y);
      uint16 *newGreen=(uint16*)output[1]->RowPtr16(y);
      uint16 *newBlue=(uint16*)output[2]->RowPtr16(y);

      uint16 *oldRed   =(uint16*)color[0]->RowPtr16(y);
      uint16 *oldGreen =(uint16*)color[1]->RowPtr16(y);
      uint16 *oldBlue  =(uint16*)color[2]->RowPtr16(y);
      for (x=0;x<width;x++)
      {
         newRed[x]=RandomColor(oldRed[x]);
         newGreen[x]=RandomColor(oldGreen[x]);
```

```
            newBlue[x]=RandomColor(oldBlue[x]);
        }
    }
    fColorOffscreen->UnGetTile(color,index,3,false);

    return MC_S_OK;
}
```

Sand::FinishDraw

```
MCCOMErr Sand::FinishDraw()
```

The final method, **FinishDraw()**, is called after **DrawRect()**:

```
MCCOMErr Sand::FinishDraw()
{
    fColorOffscreen=NULL;
    return MC_S_OK;
}
```

Example 2: Color Balance

Description

In this example, a post render filter is implemented which adjusts the color balance. The Distance Channel of the G-Buffers is used to add a feature to ignore the background.

Parameters

The following variables are used for the user-input parameters:

```
fRIntensity:   Adjustment value to the Red channel, as a percentage.
fGIntensity:   Adjustment value to the Green channel, as a percent-
   age.
fBIntensity:   Adjustment value to the Blue channel, as a percentage.
fIgnoreBack:   A boolean value to ignore the background.
```

These are defined in ColorBalance.h as follows:

```
struct ColorBalanceData
{
      real   fRIntensity;            // Intensity in %
      real   fGIntensity;            // Intensity in %
      real   fBIntensity;            // Intensity in %
      bool   fIgnoreBack;             // Ignore Background;
};
```

These are then mapped to the user interface parameters using the PMap resource in ColorBalance.r. Note that the parameters within the *ColorBalanceData* struct must be in the same order as in the PMap.

Functions

ColorBalance::GetBufferNeeds

```
void ColorBalance::GetBufferNeeds(uint32 &needs, void *renderer)
```

GetBufferNeeds() is implemented to indicate that the Distance Channel is needed:

```
void ColorBalance::GetBufferNeeds(uint32 &needs, void *renderer)
{
   //Uses the Distance Channel
   needs=(1<<k32Distance);

}
```

ColorBalance:PrepareDraw

```
MCCOMErr ColorBalance::PrepareDraw(IShRasterLayer* input,const TMCRect& bounds,
      IShChannel* buffers[], I3DShScene* scene, I3DShCamera* renderingCamera,
      const TBBox2D& productionFrame)
```

In **PrepareDraw()**, the private variable *fColorOffScreen* is assigned to the input and *fZBuffer* to the Distance Channel:

```
MCCOMErr ColorBalance::PrepareDraw(IShRasterLayer *input,const TMCRect
      &bounds,IShChannel *buffers[],I3DShScene *scene,TBox2D &uvBox)
{
   //fColorOffScreen is assigned to the input buffer
   fColorOffscreen=input;

   //fZBuffer is set to the Distance Channel
   fZBuffer=buffers[k32Distance];

   fDistanceBuffer = new real[];

   return MC_S_OK;
}
```

ColorBalance::DrawRect

```
MCCOMErr ColorBalance::DrawRect(const TMCRect& outputRect, const TChannel-
      DataBucket* output[], const TBBox2D& uvBox)
```

In **DrawRect()**, the setup for the output is similar to Sand. In addition, a local variable, *dist*, is assigned to the corresponding Distance Buffer value for each pixel:

```
MCCOMErr ColorBalance::DrawRect(const TMCRect& outputRect, const TChannel-
      DataBucket* output[], const TBBox2D& uvBox)
{
   const TChannelDataBucket *color[3];
   uint16 index[3]={0,1,2};
   fColorOffscreen->GetTile(outputRect,color,index,3,eTileRead);

   real *dist=NULL;

   int32 x,y;
   int32 width=outputRect.GetWidth();
   int32 height=outputRect.GetHeight();

   void *baseAddress=(void*)fDistanceBuffer;
   fZBuffer->GetData(outputRect, &fDistance, baseAddress, eTileRead);

   for (y=0;y<height;y++)
   {
      uint16 *newRed=(uint16*)output[0]->RowPtr16(y);
      uint16 *newGreen=(uint16*)output[1]->RowPtr16(y);
      uint16 *newBlue=(uint16*)output[2]->RowPtr16(y);

      uint16 *oldRed   =(uint16*)color[0]->RowPtr16(y);
      uint16 *oldGreen =(uint16*)color[1]->RowPtr16(y);
      uint16 *oldBlue  =(uint16*)color[2]->RowPtr16(y);

      dist = (real*)fDistance.RowPtr32(y);
```

If the value of *dist* is equal to the background, and Ignore Background (*fIgnoreBack*) is true, then the original value of the pixel is returned. Otherwise, the value of each channel is multiplied by the intensity percentage set in the UI. Note that the result is checked to see if it is greater than 16,384 (0x4000h), which is the maximum value for each channel:

```
        for (x=0;x<width;x++)
        {
            // In the Distance Channel, the background color is 0x1e20
            if (dist[x] != 1e20f || fData.fIgnoreBack == false)
            {
                newRed[x] = oldRed[x]*(fData.fRIntensity);
                if ( newRed[x] > 0x4000) newRed[x] = 0x4000;

                newGreen[x] = oldGreen[x]*(fData.fGIntensity);
                if ( newGreen[x] > 0x4000) newGreen[x] = 0x4000;

                newBlue[x] = oldBlue[x]*(fData.fBIntensity);
                if ( newBlue[x] > 0x4000) newBlue[x] = 0x4000;
            }
            else //Pixel is part of the Background and Ignore Background is True
            {
                newRed[x]= oldRed[x];
                newGreen[x] = oldGreen[x];
                newBlue[x]= oldBlue[x];
            }
        }//For x
    }//For y
    fZBuffer->UnGetData(&fDistance,false);
    fColorOffscreen->UnGetTile(color,index,3,false);

    return MC_S_OK;
}
```

ColorBalance::FinishDraw

```
MCCOMErr ColorBalance::FinishDraw()
```

Finally, **FinishDraw()** is called. Any variables which were set in **PrepareDraw()** are cleared:

```
MCCOMErr ColorBalance::FinishDraw()
{
    fColorOffscreen=NULL;
    fZBuffer=NULL;

    delete [] fDistanceBuffer;
    fDistanceBuffer=NULL;

    return MC_S_OK;
}
```

Writing a Geometric Primitive

Family ID : 'prim'
Interface ID : IID_I3DExGeometricPrimitive
Interface file : I3DExPrimitive.h
Basic Implementation file: BasicPrimitive.h, BasicPrimitive.cpp

Overview

A geometric primitive describes the geometry of an object. It is possible to implement it in one of three ways: geometry only, geometry with a custom ray tracing method, or as a volumetric primitive.
To add a custom ray-tracing method you must inherit from the class **IRaytracablePrimitive** and **TBasicPrimitive**. To create a volumetric primitive you must inherit from the class **IVolumePrimitive** and **TBasicPrimitive**. See the reference guide for more information on these interfaces.
I3DExPrimitive uses the following methods:

- **GetBBox(TBox3D* bbox)**

GetBBox() defines the initial bounding box for the primitive.

- **EnumPatches(EnumPatchesCallback callback, void* privData)**

EnumPatches() implements the geometry of the object for spline patches.

- **GetNbrLOD(int16 &nbrLod)**
- **GetLOD(int16 lodIndex,real &lod)**

LOD stands for Level of Detail. LOD management allows the Shell to get multiple Facet Meshes from the primitive depending on the rendering needs. An LOD value can be thought of as an "error" level such that a low LOD value, indicating a low amount of error, produces a corresponding high number of polygons. An LOD index of 1 indicates the best possible quality. If your primitive has a fixed set of facets, GetNbrLOD() should return 1 to indicate a single level of detail. Otherwise, the *lodIndex* is used to calculate the *lod*. See the default implementation in BasicPrimitive.cpp as an example.

- **GetFacetMesh(uint32 index, FacetMesh** outMesh)**
- **GetFacetMesh(real lod, FacetMesh** outMesh)**

GetFacetMesh() is used to calculate the facets of the primitive based on the LOD or LOD Index. Only one of these functions should return a *FacetMesh*. If **GetFacetMesh(uint32 index)** is implemented, then **GetFacetMesh(real lod)** should return NULL.

- **GetUVSpaceCount()**
- **GetUVSpace(uint32 uvSpaceID, UVSpaceInfo* uvSpaceInfo)**
- **GetUVSpaceRDS5(uint32 uvSpaceID, UVSpaceInfoRDS5* uvSpaceInfo)**

GetUVSpaceCount() defines the number of UV spaces for the primitive. **GetUVSpace()** defines the UV values for each UV space.
GetUVSpaceRDS5() should be implemented if your primitive was ported from RDS 5 or earlier. It is used to convert the UV space from the RDS 5 version to the current version so that RDS files containing the primitive imported into the current application can be displayed and rendered correctly.

- **UV2XYZ(TVector2* uv, uint32 uvSpaceID, TVector3* resultPosition, boolean* inUVSpace)**

This is an optional method to convert UV coordinates to XYZ coordinates for your object.

- **CanBeSplit()**

This method should return true if the primitive implements **SplitPrimitive()**.

- **SplitPrimitive(TMCCountedPtrArray<I3DExGeometricPrimitive>& subParts,**

TMCArray<TTransform3D>& subPartPositions)

SplitPrimitive() builds a list of sub-parts of itself. The variable *subPartPositions* gives the relative position of each sub part in the object's local coordinates. Both arrays need to be of same size.

- **AppendToRenderables(const TTransform3D& worldFromModelTfm, TRenderableAndTfmArray& renderableAndTfm)**

Retrieve the renderables information through **AppendToRenderables()**.

- **AutoSwitchToModeler()**

AutoSwitchToModeler() returns true if the primitive should open in its modeler immediately after it is inserted into the scene.

Location in the User Interface

Geometric primitives are added to the scene from the Insert menu in the Assemble room. If an Icon is provided as a UImg resource, then the primitive will also add a button to the Toolbar.

Example 1: Flat

Description

The first example is simple geometry with two facets. There are no user parameters. This example provides an icon which looks like a flat plane as a UImg resource.

Functions

Flat::GetFacetMesh

```
MCCOMErr Flat::GetFacetMesh (uint32 lodIndex, FacetMesh** outMesh)
```

GetFacetMesh() is used to define the geometry. When you define the Vertices of your Facets, you must do it in the Trigonometric order. If you don't do that you'll get an error message saying that you have invalid normals.

```
MCCOMErr Flat::GetFacetMesh (uint32 lodIndex, FacetMesh** outMesh)
{
    FacetMeshAccumulator acc;

    real   size = 10.0f;
    TFacet3DaF;
    TVector3normal;

    normal[0]=0.0f;
    normal[1]=0.0f;
    normal[2]=1.0f;

    aF.fUVSpace=0;
    aF.fVertices[0].fVertex[0]=-size;
    aF.fVertices[0].fVertex[1]=-size;
    aF.fVertices[0].fVertex[2]=0.0f;
    aF.fVertices[0].fUV[0]=0.0f;
    aF.fVertices[0].fUV[1]=0.0f;
    aF.fVertices[0].fNormal=normal;

    aF.fVertices[1].fVertex[0]=size;
    aF.fVertices[1].fVertex[1]=-size;
```

```
    aF.fVertices[1].fVertex[2]=0.0f;
    aF.fVertices[1].fUV[0]=1.0f;
    aF.fVertices[1].fUV[1]=0.0f;
    aF.fVertices[1].fNormal=normal;

    aF.fVertices[2].fVertex[0]=-size;
    aF.fVertices[2].fVertex[1]=size;
    aF.fVertices[2].fVertex[2]=0.0f;
    aF.fVertices[2].fUV[0]=0.0f;
    aF.fVertices[2].fUV[1]=1.0f;
    aF.fVertices[2].fNormal=normal;

    acc.AccumulateFacet(&aF);

    aF.fVertices[0].fVertex[0]=size;
    aF.fVertices[0].fVertex[1]=size;
    aF.fVertices[0].fVertex[2]=0.0f;
    aF.fVertices[0].fUV[0]=1.0f;
    aF.fVertices[0].fUV[1]=1.0f;

    aF.fVertices[1].fVertex[0]=-size;
    aF.fVertices[1].fVertex[1]=size;
    aF.fVertices[1].fVertex[2]=0.0f;
    aF.fVertices[1].fUV[0]=0.0f;
    aF.fVertices[1].fUV[1]=1.0f;

    aF.fVertices[2].fVertex[0]=size;
    aF.fVertices[2].fVertex[1]=-size;
    aF.fVertices[2].fVertex[2]=0.0f;
    aF.fVertices[2].fUV[0]=1.0f;
    aF.fVertices[2].fUV[1]=0.0f;

    acc.AccumulateFacet(&aF);

    acc.MakeFacetMesh(outMesh);
    return MC_S_OK;
}
```

Flat::GetBoundingBox

```
void Flat::GetBoundingBox(TBBox3D* bbox)
```

The initial bounding box is defined by implementing **GetBoundingBox()**. As this is a flat horizontal plane, *fmin[2]* and *fmax[2]* are set to 0 to indicate no height along the Z-axis:

```
void Flat::GetBoundingBox(TBBox3D* bbox)
{
    bbox->fMin[0]=-10.0f; //X
    bbox->fMax[0]=10.0f;
    bbox->fMin[1]=-10.0f; //Y
    bbox->fMax[1]=10.0f;
    bbox->fMin[2]=0.0f; //Z
    bbox->fMax[2]=0.0f;
    return MC_S_OK;
}
```

Flat::GetUVSpace

```
MCCOMErr Flat::GetUVSpace(uint32 uvSpaceID, UVSpaceInfo* uvSpaceInfo)
```

The UV space is described in **GetUVSpace()**. In this example, we only need to define *fWraparound* as false because the object is already normalized such that the shell can calculate the UV values.

```
MCCOMErr Flat::GetUVSpace(uint32 uvSpaceID, UVSpaceInfo* uvSpaceInfo)
{
    if (uvSpaceID == 0)
    {
        uvSpaceInfo->fWraparound[0] = false;  // No Wrap around
        uvSpaceInfo->fWraparound[1] = false;
    }
    return MC_S_OK;
}
```

Flat::GetUVSpaceCount

```
uint32 Flat::GetUVSpaceCount()
```

The number of UV spaces is defined in **GetUVSpaceCount()**. There is a single UV space covering the entire surface.

```
uint32 Flat::GetUVSpaceCount()
{
    return 1; // the flat is describe with only 1 UV-Space
}
```

Example 2: Star

Description

The 2nd example is a 3D Star. This example provides an icon which looks like a star as a UImg resource.

A Star with 5 branches

Parameters

The star has the following UI parameters:

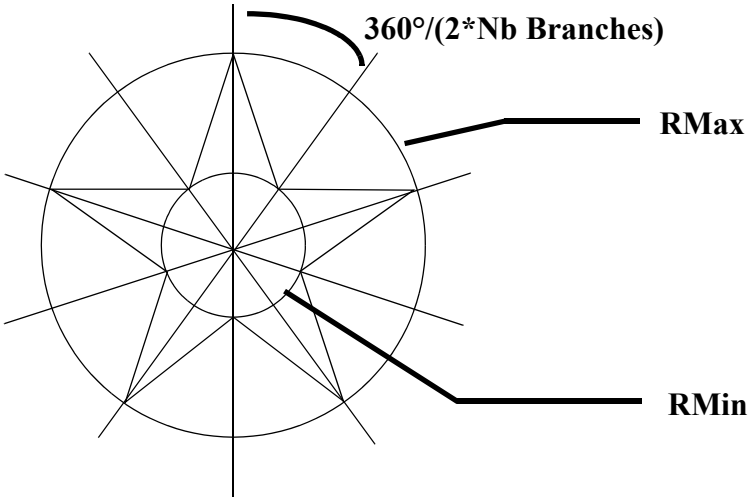
fnbBranches: The number of points on the star, ranging from 2 to 20.

This is defined in Star.h as follows:

```
struct StarData
{
    int16 fNbBranches;  // Number of branches of the 3D Star (from 2 to 20)
};
```

This is then mapped to the user interface parameters using the PMap resource in Star.r. Note that the parameters within the *StarData* struct must be in the same order as in the PMap.

The star is constructed by creating instances of the basic geometry rotated around a center point. The angle between instances is calculated as shown below:



Calculating the angle between instances

Functions

Star::GetFacetMesh

```
MCCOMErr Star::GetFacetMesh (uint32 lodIndex, FacetMesh** outMesh)
```

GetFacetMesh() is used to define the geometry. When you define the Vertices of your Facets, you must do it in the Trigonometric order. If you don't do that you'll get an error message saying that you have invalids normals.

```
MCCOMErr Star::GetFacetMesh (uint32 lodIndex, FacetMesh** outMesh)
{
    TFacet3D starFacet;
    TVector3 normal;
    FacetMeshAccumulator starMeshAcc;

    real angle,anglestep,radius1,radius2;//,radiusswap,sinus,cosinus;
    real k360=3.14159265358979323846264233f * 2.0f;

    angle=0.0f;
    anglestep=k360/((real)fData.fNbBranches) * 0.5f;
    radius1=10.0f;
    radius2=4.0f;

    // Inferior Facets
    // -- Common Vertex of each inferior facets
    starFacet.fVertices[0].fVertex[0]=0.0f;
    starFacet.fVertices[0].fVertex[1]=0.0f;
    starFacet.fVertices[0].fVertex[2]=-4.0f;

    starMeshAcc = MakeStarFacet( starMeshAcc, starFacet, 2, 1, fData.fNbBranches
        );

    // Superior Facets
    // -- We change the z value for the common Vertice of superiors Facets
    starFacet.fVertices[0].fVertex[2]=4.0;

    starMeshAcc = MakeStarFacet( starMeshAcc, starFacet, 1, 2, fData.fNbBranches
        );

    starMeshAcc.MakeFacetMesh(outMesh);
```

```
    return MC_S_OK;
}
```

The UV definitions for this object are identical to Flat.

Example 3: TeaPot

Description

The Teapot example uses bi-cubic patches. A list of patches, found in TeaPotData.h, is used. The **callback()** function is called once for each patch. This example does not include UV data for the patches. There are no user-input parameters. This example provides an icon which looks like a tea pot as a UImg resource.

Functions

TeaPot::EnumPatches

```
MCCOMErr TeaPot::EnumPatches( EnumPatchesCallback callback, void* privData)
```

EnumPatches() is implemented to create the TeaPot from the patch data:

```
MCCOMErr Teapot::EnumPatches( EnumPatchesCallback callback, void* privData)
{
   int16 indexPatch;
   TPatch3D TeapotPatch;
   int16 uPatchIndex,vPatchIndex;

   if (callback == NULL) return MC_S_OK;

   TeapotPatch.fu[0]=0.0f;
   TeapotPatch.fu[1]=0.0f;
   TeapotPatch.fv[0]=0.0f;
   TeapotPatch.fv[1]=0.0f;
   TeapotPatch.fUVSpace=0;
   TeapotPatch.fReserved=0;


   for (indexPatch=0;indexPatch<NUM_PATCHES;indexPatch++)
   {
      for (uPatchIndex=0;uPatchIndex<4;uPatchIndex++)
      {
         for (vPatchIndex=0;vPatchIndex<4;vPatchIndex++)
         {
            TeapotPatch.fVertices[uPatchIndex][vPatchIndex][0]=ver-
      tex[vertex_index[indexPatch][uPatchIndex][vPatchIndex]-1][0] * kTeapot-
      Size;
            TeapotPatch.fVertices[uPatchIndex][vPatchIndex][1]=ver-
      tex[vertex_index[indexPatch][uPatchIndex][vPatchIndex]-1][1] * kTeapot-
      Size;
            TeapotPatch.fVertices[uPatchIndex][vPatchIndex][2]=ver-
      tex[vertex_index[indexPatch][uPatchIndex][vPatchIndex]-1][2] * kTeapot-
      Size;
         }
      }
    callback(&TeapotPatch,privData);
   }
  return MC_S_OK;
```

```
}
```

Example 4: Sphere

Description

This example implements a geometric sphere, as well as a custom method for ray-tracing the primitive. There are no user inputer parameters. This example provides an icon which looks like a sphere as a UImg resource.

A Sphere

Note the inheritance in the object definition in Sphere.h includes **IRaytracablePrimitive:**

```
class Sphere : public TBasicPrimitive, public IRaytracablePrimitive
```

This multiple-inheritance requires a custom implementation of **AddRef()** and **QueryInterface()** to specify which ID should be returned.
EnumPatches() is used to create the geometry in a manner similar to the TeaPot example. The sphere example differs in that each patch is created with a mathematical formula forming one eighth of a sphere. See the full implementation below.
The boundingbox and UV space implementation are also similar to the previous examples.

Functions

Sphere::UV2XYZ

```
MCCOMErr Sphere::UV2XYZ( TVector2* uv, uint32 uvSpaceID, TVector3* resultPosi-
      tion, boolean* inUVSpace)
```

This example implements **UV2XYZ()**. This method is used to convert from UV coordinates to XYZ coordinates.

```
MCCOMErr Sphere::UV2XYZ( TVector2* uv, uint32 uvSpaceID, TVector3* resultPosi-
      tion, boolean* inUVSpace)
{
   real phi,theta;
   real sinphi,cosphi;
   real sintheta,costheta;

   *inUVSpace=true;
   if (((*uv)[0]<0.0f) || ((*uv)[0]>=kUmax)) *inUVSpace=false;
   if (((*uv)[1]<0.0f) || ((*uv)[1]>=kVmax)) *inUVSpace=false;

   if (*inUVSpace == boolean(true))
   {
      phi=(*uv)[0];
      theta=(*uv)[1]-(kVmax/2);

      // Spherical Coordinates To XYZ-Coordinates
```

```
        sinphi=sin(phi);
        cosphi=cos(phi); //phi.DegreeGetSinCos(sinphi,cosphi);
        sintheta=sin(theta);
        costheta=cos(theta); //theta.DegreeGetSinCos(sintheta,costheta);

        (*resultPosition)[0]=cosphi*costheta;
        (*resultPosition)[1]=sinphi*costheta;
        (*resultPosition)[2]=sintheta;
        (*resultPosition)[0] *= kDefaultSphereRadius;
        (*resultPosition)[1] *= kDefaultSphereRadius;
        (*resultPosition)[2] *= kDefaultSphereRadius;
    }
    return MC_S_OK;
}
```

Sphere::RayHit

```
MCCOMErr Sphere::RayHit( boolean* didHit, Ray3D* aR, RayHitParameters*
RayHitParams, RayHit3D* hit)
```

A custom ray-trace method can now be implemented. If you can give the real coordinate of an intersection between a ray and the surface of the object, it is possible to optimize the ray-tracing methods. The final image is better because the object is not approximated.

RayHit() is implemented to determine if an intersection between the ray being calculated and the sphere occurred:

```
MCCOMErr Sphere::RayHit( boolean* didHit, Ray3D* aR, RayHitParameters*
RayHitParams, RayHit3D* hit)
{
    TVector3 OriginToCenter;
    real    DirectionNorm2;
    real    dotProduct;
    real    t;              // Sphere Center = Ray Origin + t * Ray Direction
    TVector3 CH;             // position of the center projected on the ray
    real    distCH2;        // square of the distance of the projected point
    real    resT;
    real    delta,delta2;
    real    radius2=kDefaultSphereRadius*kDefaultSphereRadius;

    OriginToCenter[0]=-aR->fOrigin[0];
    OriginToCenter[1]=-aR->fOrigin[1];
    OriginToCenter[2]=-aR->fOrigin[2];

    DirectionNorm2 = aR->fDirection[0]*aR->fDirection[0] +
                  aR->fDirection[1]*aR->fDirection[1] +
                  aR->fDirection[2]*aR->fDirection[2];

    dotProduct = aR->fDirection[0]*OriginToCenter[0] +
                 aR->fDirection[1]*OriginToCenter[1] +
               aR->fDirection[2]*OriginToCenter[2];

    t=dotProduct/DirectionNorm2;

    CH[0] = aR->fOrigin[0] + aR->fDirection[0]*t;
    CH[1] = aR->fOrigin[1] + aR->fDirection[1]*t;
    CH[2] = aR->fOrigin[2] + aR->fDirection[2]*t;
    distCH2 = CH[0]*CH[0] + CH[1]*CH[1] + CH[2]*CH[2];

    if (distCH2 > radius2)
    {
        *didHit = false;
     }
    else if (distCH2 == radius2)
```

```
    {
        resT = t;
        *didHit = true;
    }
    else
    {
        // distCH2<radius2
        *didHit = true;
        delta2 = (radius2-distCH2)/DirectionNorm2;
        delta = sqrt(delta2); //delta2.GetSquareRoot(delta);
        resT = t-delta;
        if (resT<=RayHitParams->tmin)
        {
            resT = t+delta;
        }
    }
    if (resT<=RayHitParams->tmin)
    {
        *didHit = false;
    }

    if (resT>RayHitParams->tmax)
    {
        *didHit=false;
    }

    if (*didHit== boolean(true))
    {
        hit->fPointLoc[0] = aR->fOrigin[0] + aR->fDirection[0]*resT;
        hit->fPointLoc[1] = aR->fOrigin[1] + aR->fDirection[1]*resT;
        hit->fPointLoc[2] = aR->fOrigin[2] + aR->fDirection[2]*resT;
        hit->fNormalLoc[0] = hit->fPointLoc[0] / kDefaultSphereRadius;
        hit->fNormalLoc[1] = hit->fPointLoc[1] / kDefaultSphereRadius;
        hit->fNormalLoc[2] = hit->fPointLoc[2] / kDefaultSphereRadius;
        hit->ft = resT;
        hit->fCalcInfo = GetDetails;
    }

    return MC_S_OK;
}
```

Sphere::GetRayHitDetails

```
MCCOMErr Sphere::GetRayHitDetails(RayHit3D* hit)
```

If a hit occurs, **GetRayHitDetails()** is called:

```
MCCOMErr Sphere::GetRayHitDetails(RayHit3D* hit)
{
    real phi,theta,rxy;
    real sinphi,cosphi,sintheta,costheta;

    if (!hit->fShouldSetUV && !hit->fShouldSetIsoUV) return MC_S_OK;

    phi = atan2(hit->fNormalLoc[1],hit->fNormalLoc[0]); //phi.DegreeSetFromSin-
        Cos(hit->fNormalLoc[1],hit->fNormalLoc[0]);
    rxy = hit->fNormalLoc[0]*hit->fNormalLoc[0]+hit->fNormalLoc[1]*hit->fNormal-
        Loc[1];
    rxy = sqrt(rxy); //rxy.GetSquareRoot(rxy);
    theta = atan2(hit->fNormalLoc[2],rxy); //theta.DegreeSetFromSinCos(hit->fNor-
        malLoc[2],rxy);
```

```
    if (hit->fShouldSetUV)
    {
        hit->fUV[0] = phi;
        if (theta>kVmax)
        {
            theta = theta - (kVmax*2);
        }
        hit->fUV[1] = theta+(kVmax/2);
    }
    if (hit->fShouldSetIsoUV)
    {
        sinphi = sin(phi);
        cosphi = cos(phi); //phi.DegreeGetSinCos(sinphi,cosphi);
        sintheta = sin(theta);
        costheta = cos(theta); //theta.DegreeGetSinCos(sintheta,costheta);

        hit->fIsoU[0] = -sinphi*kDefaultSphereRadius;
        hit->fIsoU[1] = cosphi*kDefaultSphereRadius;
        hit->fIsoV[2] = 0.0f;
        hit->fIsoV[0] = cosphi*sintheta*kDefaultSphereRadius;
        hit->fIsoV[1] = sinphi*sintheta*kDefaultSphereRadius;
        hit->fIsoV[2] = costheta*kDefaultSphereRadius;
    }

    return MC_S_OK;

}
```

MCCOMErr Sphere::IsInfiniteBB

```
MCCOMErr Sphere::IsInfiniteBB(boolean &isInfinite)
```

IsInfinteBB() is implemented to indicate that the sphere has finite bounds:

```
MCCOMErr Sphere::IsInfiniteBB(boolean &isInfinite)
{
    isInfinite = false;
    return MC_S_OK;

}
```

Writing a Shader

Family ID : 'shdr'
Interface ID : IID_I3DExShader
Interface file : I3DExShader.h
Basic Implementation file: BasicShader.h, BasicShader.cpp

Overview

Shaders define a value for each point in the UV space of an object. They are used by the renderer to calculate the appearance of an object in the rendered image. A shader is comprised of a group of channels (color, transparency, reflections, etc.) which are combined to form a complete shader.
The most common type of shader extension is a sub-shader. A sub-shader is like a building block which can be combined with other sub-shaders by the user to create a more complex shader. A sub-shader is added to an individual channel and returns a value, color, or vector by implementing at least one or more of the following functions:

- **GetValue(real& result,boolean& fullArea,ShadingIn& shadingIn)**
- **GetColor(TMCColorRGB& result,boolean& fullArea,ShadingIn& shadingIn)**
- **GetVector(TVector3& result,ShadingIn& shadingIn)**

In each of these examples, the input *ShadingIn* describes a single point in the UV space.
In addition to sub-shaders, it is possible to implement a shader which controls all channels by implementing an additional method:

- **DoShade(ShadingOut &result, ShadingIn& shadingIn)**

All shaders and sub-shaders must also implement the following methods:

- **IsEqualTo(I3DExShader* aShader)**
- **GetShadingFlags(ShadingFlags& theFlags)**
- **GetImplementedOutput()**

IsEqualTo() is used by the shell to compare two shaders. **GetShadingFlags()** tells the shell which parameters to use. For more details on the *ShadingFlags* structure, see the descriptions of the data structure in the reference guide. **GetImplementedOutput()** indicates which method will be implemented (**GetValue()**, **GetColor()**, etc.), and if a *ShadedArea* implementation will be used.

Location in the User Interface

Shaders and sub-shaders are found in the Texture room. Sub-Shaders are added to each channel of a shader individually. They can be combined with functions (mix, add, etc.) to form more complex shaders. Shaders are selected from the shader list.

Example 1: Checker

Description

The first example is a sub-shader which returns a value to create a checkerboard across the surface of an object. It implements both the *shadingIn* and *ShadedArea* methods.

Parameters

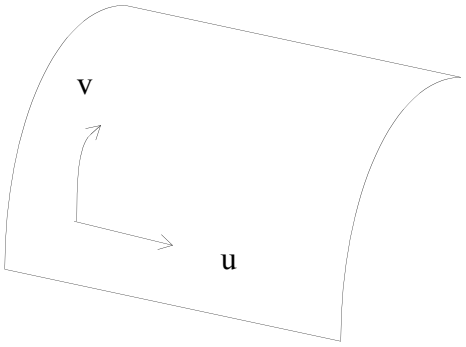
The following variable for the user-input parameters is used:

fNb[2]: The horizontal and vertical number of checkers.

This is defined in Checker.h:

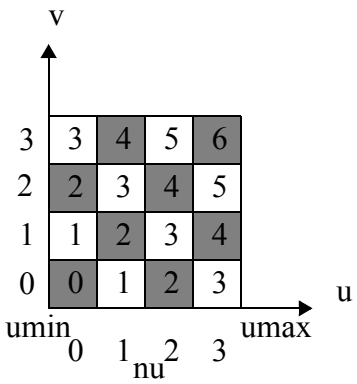
```
int32  fNb[2];// Contains the Horizontal (fNb[0]) and Vertical (fNb[1]) values
```

The elements within the array are mapped to the UI using a basic PMap. Note that they must be in the same order as in the PMap.
The Horizontal and Vertical checkers are mapped along the UV space of the object.



U-V Coordinates on a Surface

The UV coordinates of the object are labeled nbSquareU and nbSquareV:



The following formula is used to calculate the pattern:

$$n_u = \frac{u - u_{min}}{u_{max} - u_{min}} \cdot nbSquareU$$

Each number in the square is the addition of the integer part of nu and nv. The even and odd values form a checkerboard.

Functions

Checker::IsEqualTo

```
boolean Checker::IsEqualTo(I3DExShader* aShader)
```

This simply returns True if this shader and the shader passed to the function are equal, and False otherwise:

```
boolean Checker::IsEqualTo(I3DExShader* aShader)
{
    return ((fNb[0]==((Checker*)aShader)->fNb[0]) &&
```

```
        (fNb[1]==((Checker*)aShader)->fNb[1]));
}
```

Checker::GetShadingFlags

```
MCCOMErr Checker::GetShadingFlags(ShadingFlags& theFlags)
```

GetShadingFlags() tells the 3D Shell which parameters the sub-shader uses. This way, only the minimal number of parameters are calculated (for more details on the *ShadingFlags* structure, see the descriptions of the data structure in the reference guide).
In this implementation, the need for UV-Coordinates is indicated:

```
MCCOMErr Checker::GetShadingFlags(ShadingFlags& theFlags)
{
    theFlags.fNeedsUV = true;// We need UV coordinates
    theFlags.fConstantChannelsMask = kNoChannel;
    return MC_S_OK;
}
```

Checker::GetImplementedOutput

```
EShaderOutput Checker::GetImplementedOutput()
```

In the function **GetImplementedOutput()**, *kUsesGetValue* is returned to indicate that we will use **GetValue()** (the *shadingIn* implementation).

```
EShaderOutput Checker::GetImplementedOutput()
{
    return kUsesGetValue;// We use GetValue
}
```

Checker::GetValue

```
MCCOMErr Checker::GetValue(real& result,boolean& fullArea, ShadingIn& shadingIn)
```

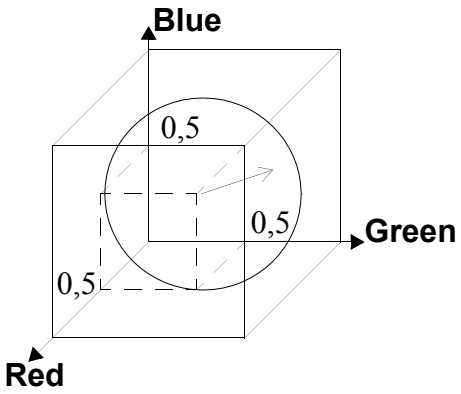
In this example, a value of either 0.0 or 1.0 is returned to define the checkerboard. In the first implementation of **GetValue()**, a value is returned for each UV point. *ShadingIn* defines the point, and the return value is assigned to the variable *result*. *fMul* was assigned to *fNb* in **ExtensionDataChange()**:

```
MCCOMErr Checker::GetValue(real& result,boolean& fullArea, ShadingIn& shadingIn)
        // Without shading area
{
    real tempx = shadingIn.fUV[0] * fMul[0];
    real tempy = shadingIn.fUV[1] * fMul[1];
    if (MyEven(int32(RealFloor(tempx) + RealFloor(tempy))))
    {
        result = kRealZero;
    }
    else
    {
        result = kRealOne;
    }
    fullArea = true;

    return MC_S_OK;
}
```

Example 2: Rainbow

Description

The second example is a sub-shader which returns a color for a point. This sub-shader uses the surface normal, with either Local or Global Coordinates, to generate a color in the RGB color cube.



Parameters

The following variables are used for the user-input parameters:

fIntensity: The intensity, as a percentage.
fModeLocalOrGlobal: Flag to use local or global coordinates.

These are defined in Rainbow.h as follows:

```
struct RainbowShaderPublicData
{
    int16     fIntensity; // Intensity of the rainbow
    int32     fModeLocalOrGlobal;// Mode of the rainbow ( Local or Global )
};
```

These are then mapped to the user interface parameters using the PMap resource in ColorBalance.r. Note that the parameters within the *RainbowShaderPublicData* struct must be in the same order as in the PMap.

Functions

Rainbow::GetShadingFlags

```
MCCOMErr Rainbow::GetShadingFlags(ShadingFlags& theFlags)
```

Unlike the checker example, the Rainbow Shader uses the Normal rather than the UV coordinates. **GetShadingFlags()** is implemented to reflect this:

```
MCCOMErr Rainbow::GetShadingFlags(ShadingFlags& theFlags)
{
    theFlags.fNeedsUV = false;
    theFlags.fConstantChannelsMask = kNoChannel;

    return MC_S_OK;
}
```

Rainbow::GetImplementedOutput

```
uint32 Rainbow::GetImplementedOutput()
```

As in the checker example, **GetImplementedOutput()** is used to indicate that **GetColor()** is implemented, with only the *ShadingIn* method used:

```
uint32 Rainbow::GetImplementedOutput()
{
    return kUsesGetColor ;// the Rainbow Shader is a Color Shader
}
```

Rainbow::GetColor

```
MCCOMErr Rainbow::GetColor(TMCColorRGBA& result,boolean& fullArea,ShadingIn&
       shadingIn)
```

GetColor() is implemented to convert the UV coordinates to a point in the color cube:

```
MCCOMErr Rainbow::GetColor(TColorRGBA& result, ShadingIn& shadingIn)
{

    real temp = RainbowPublicData.fIntensity;
    temp/=100.0;

    if (RainbowPublicData.fModeLocalOrGlobal==1)
    {
        result.R = (((shadingIn.fNormalLoc.x)*temp)/2)+0.5;
        result.G = (((shadingIn.fNormalLoc.y)*temp)/2)+0.5;
        result.B = (((shadingIn.fNormalLoc.z)*temp)/2)+0.5;
    }
    else
    {
        result.R = (((shadingIn.fGNormal.x)*temp)/2)+0.5;
        result.G = (((shadingIn.fGNormal.y)*temp)/2)+0.5;
        result.B = (((shadingIn.fGNormal.z)*temp)/2)+0.5;
     }
    fullArea = true;

    return MC_S_OK;
}
```

Writing a Lighting Model

Family ID : ‘shdr’
Interface ID : IID_I3DExShader
Interface file : I3DExShader.h
Basic Implementation file: BasicShader.h, BasicShader.cpp

Overview

A lighting model describes the way the light interacts with the objects it intersects. A very well-known and widely used model is Phong’s model, based on three components: ambient light, diffuse light and specular light. Yet, other models are possible and sometimes physically more accurate.
The interface of the Carrara shaders enables you to implement a lighting model as a shader, which provides the user with infinite possibilities (having several objects with different behaviors regarding their lightings).

We want this particular type of shader to control all channels, so besides the methods seen in “Writing a Shader”, we need to implement:

- **DoShade(ShadingOut &result, ShadingIn& shadingIn)**

In the basic shader class (*TBasicShader*), the method which deals with the lighting is:

- **ShadeAndLight(LightingDetail& result, const LightingContext& lightingContext, I3DShShader* inShader)**

But, since this method calls subroutines, you can choose to only override the subroutines:

- **CalculateDirectLighting(LightingDetail& result, const LightingContext& lightingContext)** (which is most often the only one you have to override)
- **CalculateReflection(TMCColorRGB& reflectionColor, const LightingContext& hightingContext, const ShadingOut& shading)**
- **CalculateCaustics(TMCColorRGB& causticColor, const LightingContext& hightingContext, const ShadingOut& shading)**
- **CalculateIndirectLighting(TMCColorRGB& indirectDiffuseColor, const LightingContext& lightingContext, const ShadingOut& shading)**
- **ApplyTransparency(TMCColorRGBA& resColor, const LightingContext& hightingContext, const ShadingOut& shading)**
- **ApplyAlpha(TMCColorRGBA& resColor, const LightingContext& hightingContext, const ShadingOut& shading)**

Location in the User Interface

A Lighting Model is accessed through the Top Shader menu of the Shader dialog of any object (in the Texture Room).

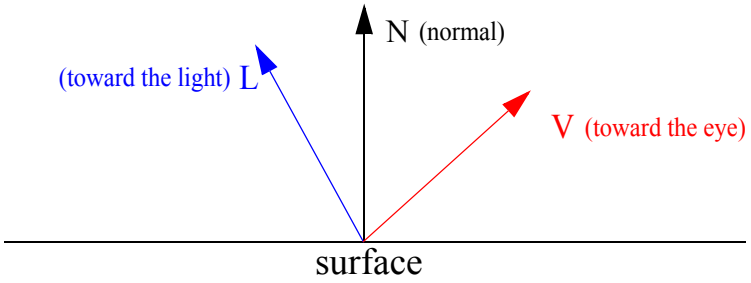
Example: Anisotropic Lighting

Description

In this example, we implement an anisotropic lighting model, which means that the light interacts with the object given a particular spatial direction, whereas there is no such direction in Phong’s model. Nonetheless, we keep the three-component base idea of the latter. We actually only change the normal vector used in the default model.
We want a lighting model that fits some specific materials for which the default model is not good enough. Indeed, Phong’s model enables the lighting of surfaces whereas some materials are best regarded as a set of

lines (instead of a continuous surface). It's the case for Christmas satin balls or vinyl records. Therefore, the lighting model we'll use must be designed for lines.
And the big difference between a line and a surface is that you have multiple normals for a point on a line, and only one normal for a point on a surface. Thus, we need to decide which one we'll pick for the calculation. This vector will replace the normal vector in Phong's model.

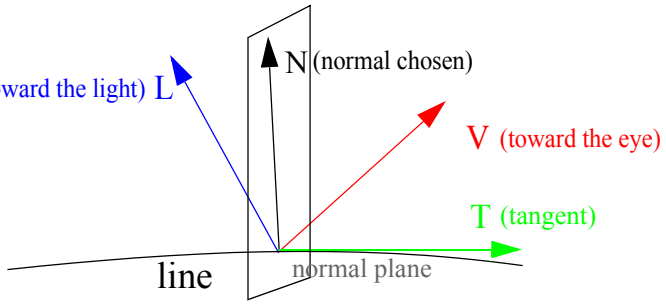
Here are the commonly used notations for the Phong model :



and I is the outcoming light, I_a is the ambient light, I^0 is the incoming light, k_d and k_s are the diffuse and specular colors, **R** is the reflection of **L** at **N** and n is a roughness parameter.

$$I = I_a + (k_d \times \vec{N} \bullet \vec{L} + k_s \times (\vec{R} \bullet \vec{V})^n) \times I^0$$

Whereas, when it comes to lightening a line, you have several choices for **N** (because there is only one tangent direction). Thus, the new model becomes:



The normal vector we'll use is the projection of the incoming light vector **L** into the normal plane (defined as perpendicular to the tangent vector). As a result, the lighting model depends on the choice of a particular direction, which is why it is called anisotropic. You'll see that the lighting of a simple sphere will significantly change when you merely translate it.

Now, when you want to apply such a lighting model to an object (as a shader), there is no obvious line on the object's surface! So, we need to create these lines by defining a tangent vector.

Parameters

The following variables are used for the user-input parameters:

```
fClusterValue: ID of the lines configuration
fUVAngle:      customizable orientation of the lines
```

This is defined in *SimpleLightingModel.h* as follows:

```
struct PMap
{
    TMCCountedPtr<IShParameterComponent>fShader; // sublevel shader
    int32                               fClusterValue;
    real32                              fUVAngle;
};
```

This is then mapped to the user interface parameters using the PMap resource in *SimpleLightingModel.r*.
Note that the parameters within the *PMap struct* must be in the same order as in the PMap.

Functions

As for every given shader, we need to implement **GetImplementedOutput** and **GetShadingFlags**. Moreover, we'll use **DoShade** to compute the shading accordingly to the first sub-shader (it's a Multi-Channel shader by default). Finally, the lighting model will be implemented in **CalculateDirectLighting**.

TSimpleLightingModel::GetImplementedOutput

```
EShaderOutput TSimpleLightingModel::GetImplementedOutput()
```

In the function **GetImplementedOutput()**, *kUsesDoShade* is returned to indicate that **DoShade()** will be used for the shading.

```
EShaderOutput TSimpleLightingModel::GetImplementedOutput()
{
    return kUsesDoShade;
}
```

TSimpleLightingModel::GetShadingFlags

```
MCCOMErr TSimpleLightingModel::GetShadingFlags(ShadingFlags& theFlags)
```

In the function **GetShadingFlags()**, we browse all the sub-shaders to get all the necessary flags and we finally ask for the UV isoparameters (the two vectors that describe the curvature of the surface on a given point), that we'll use in our lighting model.

```
MCCOMErr TSimpleLightingModel::GetShadingFlags(ShadingFlags& theFlags)
{
    TMCPtrArray<I3DShShader>::iterator iter(*fShaderList);
    for(I3DShShader* theShader = iter.First() ; iter.More() ; theShader =
        iter.Next())
    {
        if (theShader)
        {
            ShadingFlags subFlags;
            theShader->GetShadingFlags(subFlags);
            theFlags.CombineFlagsWith(subFlags);
        }
    }

    theFlags.fNeedsIsoUV = true;

    return MC_S_OK;
}
```

TSimpleLightingModel::DoShade

```
MCCOMErr TSimpleLightingModel::DoShade(ShadingOut &result, ShadingIn& theShadin-
        gIn)
```

DoShade is used to call the **DoShade** method of the sub-level shader, if there is one.

```
MCCOMErr TSimpleLightingModel::DoShade(ShadingOut &result, ShadingIn& theShadin-
        gIn)
{
    I3DShShader* const shader = (*fShaderList)[0];

    if (shader)
    {
        shader->DoShade(result,theShadingIn);
    }
    return MC_S_OK;
}
```

TSimpleLightingModel::CalculateDirectLighting

```
void TSimpleLightingModel::CalculateDirectLighting(LightingDetail& result, const
       LightingContext& lightingContext)
```

It's in this method that we'll compute the lighting model. The goal is to fill **result.*fAmbientLight***, **result.*fDiffuseLight*** and **result.*fSpecularLight***. In addition, **result.*fShadingOut*** provides a lot of information regarding the point that is being shaded (color, normal vector, glow color, specular color...).

The **lightingContext** parameter provides us with valuable information related to the lighting, such as:

- **lightingContext.*fLightingFlags*** which gives information about the lighting settings that have been chosen (see the definition of the *LightingFlags* structure for more information),
- **lightingContext.*fRaytracer*** which enables us to reach the different lights of the scene,
- **lightingContext.*fReflectDir*** which is the reflection of the viewer vector **V** at the normal vector **N**.
- **...**

To be able to compute the diffuse and specular components, we have to determine how lines are drawn on the surface of the object. The user can choose between three modes:

- the lines are horizontal (just like the latitude lines on the globe),
- the lines are vertical (just like the longitude lines on the globe),
- the lines are in an intermediate mode, whose angle is given by the user.

The choice of the mode leads to the determination of a tangent vector. At this point, we use the principle described above to choose a normal vector and then apply the Phong model.

For the ambient light, use:

```
const TMCColorRGBA& ambientLight = lightingContext.GetAmbientLight();
```

For the diffuse and specular lights:

```
result.fDiffuseLight = TMCColorRGB::kBlack;
result.fSpecularLight = TMCColorRGB::kBlack;
```

These are the diffuse and specular colors, used in the Phong model:

```
const TMCColorRGB&diffColor= shading.fColor;
const TMCColorRGB&specColor= shading.fSpecularColor;
```

Use the raytracer to get the number of lights and to reach information from each of them:

```
I3DExRaytracer* raytracer = lightingContext.fRaytracer;
const int32 lightCount = raytracer->GetLightCount();
```

Store the direct lighting information (color and direction) of each light:

```
DirectLighting directLighting;
const TMCColorRGB& lighting = directLighting.fLightColor;
const TVector3& lightDirection = directLighting.fLightDirection;

for (int32 lightIndex = 0;lightIndex<lightCount; lightIndex++)
{
   if (raytracer->GetLightIntensity(directLighting,lightIndex,lightingCon-
      text,lightingContext.fIllumSettings.fShadowsOn))
   {
   ...
   }
}
```

Once we have selected a mode for the lines, we determine the tangent vector, then the normal vector so that we can compute the diffuse light and finally we use the vector in the viewing point direction to compute the specular light.

Writing a Terrain Filter

Family ID : 'tfil'
Interface ID : IID_I3DExTerrainFilter
Interface file : I3DExTerrainFilter.h
Basic Implementation file: BasicTerrainFilter.h, BasicTerrainFilter.cpp

Overview

Terrain Filters are used to generate a terrain (in this case, we call them *Generators*) or to bring high-level changes to the shape of an existing terrain. For instance, once you have created mountains, you may want them to appear eroded. A Terrain Filter is designed to perform this kind of transformation on the height map of a terrain. To do so, you need to implement these *I3DExTerrainFilter* methods:

- **Shuffle()**
- **Filter (TMCArray<real>& heightField, TVector2& heightBound, const TIndex2& size, const TVector2& cellSize)**
- **CanBuildPreview()**

Shuffle() is the method in which you want to initialize the seed of a random distribution. **Filter()** is the method which computes the transformation of the height field. **CanBuildPreview**() returns true if the filter can build a preview of the final map by working on a smaller map.

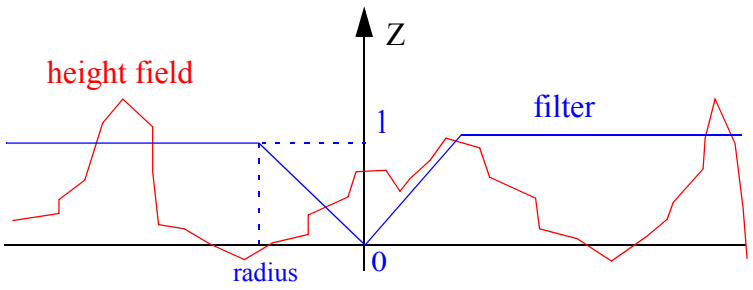
Location in the User Interface

All Terrain Filters are accessed through the Filter menu of the Terrain Modeler dialog.

Example: Conical Crater

Description

This example tends to dig a conical crater at the middle of an existing terrain. The user can choose the radius of the crater. The filter merely applies a multiplicative factor (between 0 and 1) to the height field.



Parameters

The following variable is used for the user-input parameters:

```
fRadius:        radius of the conical crater.
```

This is defined in Camera.h as follows:

```
struct TParameterMap
{
    TParameterMap();
```

```
    real fRadius;
};
```

This is then mapped to the user interface parameters using the PMap resource in *SimpleTerrainFilter.r*. Note that the parameters within the *TParameterMap* struct must be in the same order as in the PMap.

A Terrain Filter can be seen as a function of a height field which generates a new height field. In this simple example, the transformation applied to the height field is a multiplicative coefficient that only depends on the distance between the map's points and the map's center. For each point, if this distance is greater than the crater's radius (specified by the user), then the factor is equal to 1 (we do not modify the terrain). Otherwise, the factor is linearly interpolated between 0 (at the map's center) and 1 (on the crater's limit).

Functions

In this case, we do not need any random numbers so **Shuffle()** will have no instructions. **Filter()** actually performs the transformation for every point of the height field (terrain map). We can have a preview of this filter on small maps so **CanBuildPreview()** will return true.

TSimpleTerrainFilter::Filter

```
void TSimpleTerrainFilter::Filter(TMCArray<real>& heightField, TVector2& height-
      Bound, const TIndex2& size, const TVector2& cellSize)
```

heightField is a 1-dimensional array that contains the heights of the map's points. To be updated.
heightBound contains the minimum and maximum values of the height field. To be updated .
size gives the number of points in the map along the x-axis and y-axis.
cellSize gives the number of points in each cell (portion of a plane) along the x-axis and y-axis.

Given these parameters, we can go through each point of the map, calculate the multiplicative coefficient from its coordinates and apply it to the existing height of this point.
Note that the filter must be independent of the size of the map (the filter should work the same whatever the size). Therefore, the map's coordinates must be scaled to the [0 ; 1] interval (or any other determined interval) before calculating the coefficient.

Start with *index* = 0 ; *min* = INFINITE ; *max* = - INFINITE ;
For each point (x, y) of the map, do:

- scale the coordinates: xx = x / size.x and yy = y / size.y,
- calculate the factor A(xx, yy),
- multiplies **heightField**[*index*] by A(xx, yy),
- if necessary, update *min* and *max*,
- increments *index*.

Note: since the **heightField** is one-dimensional, the order you choose to browse the points of the map does matter. The convention is : *for every y, for every x, do.*

Writing a Volumetric Effect

Family ID : 'Volu'
Interface ID : IID_I3DExVolumetricEffec
Interface file : I3DExVolumetricEffect.h
Basic Implementation file: BasicVolumetricEffect.h, BasicVolumetricEffect.cpp

Overview

A Volumetric Effect aims at modifying the perceived colors of the objects and background of the scene, just like an Atmosphere does. However, as any other effect, a Volumetric Effect applies to a certain type of object that you choose (lights, primitives, etc.). The idea is that the color is altered along the ray of light that starts from a given position and reaches the object or the background. For instance, the effect can be produced by a light (e.g. a light cone effect) or any given object (e.g. aura effect). This enables you to simulate some nice and complex lighting effects.

For that purpose, you need to implement the following interface methods from *I3DExVolumetricEffect*:

- **Initialise(I3DShScene* scene)**
- **DirectionFilter(const TVector3& origin, const TVector3& direction, real maxt, TMCColorRGBA& inOutFilter, boolean isShadowCasting)**

Initialise() is called at the beginning of the rendering of a frame and allows you to create caches for your effect. **DirectionFilter()** should filter a color along a ray with the Volumetric Effect. When the function is called, *inOutFilter* contains the color of an object at the end point. When it returns, *inOutFilter* should have been modified to include the filtering effect.

You will also have to implement a Basic Data Component to connect your Volumetric Effect to a certain type of object (and to the Shell). Two interface methods from the *I3DExDataComponent* structure (see I3dExRenderFeature.h) will have to be overridden:

- **IsActive(I3DShTreeElement *tree)**
- **GetVolumetricList(TMCArray<IDType>& idArray)**

See "Writing a Data Component" for more information on data components.

Location in the User Interface

A Volumetric Effect is accessed through the Effects panel of the Properties tray, in the Assemble room. For instance, if it applies to lights, you will have to select a light to be able to see the Volumetric Effect in the panel.

Example: Light Cone Effect

Description

The light rays of a basic spot light are only made visible by the energy reflected by the objects it sees. What you may want to see from your light source, is a light cone. It enables you to easily produce some nice effects, such as car lights in a foggy night or laser beams in a night club.
To apply such a Volumetric Effect, we need to reach the lights that use this effect and compute the filtering effect produced by each of them.

The principle of the algorithm is then quite straightforward:

- for each light *L* in the scene that has the Light Cone Effect activated, do:
    - calculate the intersection of the ray with the light cone of *L*

- in general, we have two points of intersection, thus the length of the ray inside the cone
- calculate the effect's intensity coefficient *I* from this length
- add to the object's color a color inferred from *I* and *L*'s color.

Note that we can easily add a fall-off parameter and an effect's color to the formula. In addition, the atypical cases of intersection must be addressed and properly handled.

In the following example, we have also used a Post Render Filter for the preview of our Volumetric Effect in the user interface (see "Writing a Post Render Filter").

Parameters

The following variables are used for the user-input parameters:

```
fIntensity:    Intensity (%)
fRadius:       Light source radius (in)
fFogColor:     Fog effect color
fFallOff:      quality of fall off
```

This is defined in *lcPMAP.h* as follows:

```
struct LightConeInfo
{
   LightConeInfo();

   real         fIntensity;  // Intensity (%).
   real         Radius;      // Light source radius (in).
   TMCColorRGBA fFogColor;   // Fog color
   real         fFallOff;    // quality of fall off
};
```

This is then mapped to the user interface parameters using the PMap resource in *SimpleVolumetricEffect.r*. Note that the parameters within the *LightConeInfo* structure must be in the same order as in the PMap. The PMap resource will be declared as part of the data component, to which we'll add a 'data' resource in order to specify the type of object that can use our volumetric effect:

```
resource 'data' (R_SimpleLightConeControlID)
{
   {
   'li  '   // can be applied to lights
   }
};
```

Functions

We want to use a Post Render Filter for the preview of the effect. Therefore we will share data between the Volumetric Effect object (TLightConeVolumetric) and the Post Render Filter object (TLightConePostRenderer) by deriving these two classes from a base class (called TSimpleLightCone) which will contain all the necessary fields and methods. Besides, we'll have to implement the methods of the Data Component class (see "Writing a Data Component").

TLightConeVolumetric::Initialise

```
void MCCOMAPI Initialise(I3DShScene* scene)
```

Initialise is needed by both the effect and the post renderer, so we actually define it in the base class.

```
void TSimpleLightCone::Initialise(I3DShScene* scene)
{
...
```

Access the information related to the lights of the scene thanks to the passed-in parameter *scene*:

```
uint32 lightNbr=scene->GetLightsourcesCount();
TMCCountedPtr<I3DShLightsource> light;
```

Reach each light of the scene with:

```
scene->GetLightsourceByIndex(&light,index);
```

We want to know for each light whether the volumetric effect has been activated. So we need to get the light cone component associated to the current light:

```
TMCCountedPtr<IShParameterComponent> component;
DataComponentUtils::GetLightDataComponent(light,SimpleLightConeControlID,&compo-
      nent);
```

Note that *SimpleLightConeControlID* is the ID of our Data Component class.

```
boolean enabled=false;
component->GetParameter('ENBL',(void*)&enabled);
```

'ENBL' is the ID of the checkbox of the activation option of the light cone effect, in the user interface.

Make sure that the current light's type supports this effect (is *hasLightCone* true?):

```
light->GetLightInfo(hasLightCone,hasLightSphere,halfAngle,transform);
```

If it is the case, we add this light to the list of lights that asked for the Light Cone Effect. In addition, we store information for the calculation (performed in **DirectionFilter**):

```
fLights.AddElem(light);
fZScale.AddElem(tan(PI*halfAngle/180.0));

TMatrix33 scale( 1,0,0, 0,1,0, 0,0,a);
transform=Inverse(transform);
transform.fRotationAndScale=scale*transform.fRotationAndScale;
transform.fTranslation=scale*transform.fTranslation;
fTransforms.AddElem(transform);
```

Fill in the PMAP from the component (i.e. the user interface):

```
LightConeInfo data;
component->GetParameter(IDTYPE('I','N','T','S'),(void*)&data.fIntensity);
component->GetParameter(IDTYPE('R','A','D','I'),(void*)&data.fRadius);
component->GetParameter(IDTYPE('C','O','L','O'),(void*)&data.fFogColor);
component->GetParameter(IDTYPE('F','A','L','L'),(void*)&data.fFallOff);
fLightData.AddElem(data);
```

Store the colors of the selected lights:

```
uint32 lightConeNbr=fLights.GetElemCount();

fLightColorBuffers.SetElemCount(lightConeNbr);
fLightColor.SetElemCount(lightConeNbr);

for (i=0;i<lightConeNbr;i++)
{
   fLightColorBuffers[i]=GetLightColorBuffer(i,fLights[i],fTransforms[i],fLight-
      Color[i])
}
}
```

TLightConeVolumetric::DirectionFilter

```
void TSimpleLightCone::DirectionFilter(const TVector3& rayOrigin,const TVector3&
      rayDirection,real maxt,TMCColorRGBA& inOutFilter,boolean isShadowCasting)
```

DirectionFilter is needed by both the effect and the post renderer, so we actually define it in the base class.

```
void TSimpleLightCone::DirectionFilter(const TVector3& rayOrigin,const TVector3&
      rayDirection,real maxt,TMCColorRGBA& inOutFilter,boolean isShadowCasting)
{
```

We don't want to do anything if we're casting shadows:

```
   if (isShadowCasting) return; // no effect on shadow casting!
```

The needed information is already stored in lists created in **Initialise**:

```
const int32 lightCount = fLights.GetElemCount();

real Rt=0,Gt=0,Bt=0,At=0;
```

Apply the filtering effect according to each of the selected lights:

```
for (int32 lightIndex=0;lightIndex<lightCount;lightIndex++)
{
    real Rl=0,Gl=0,Bl=0,Al=0;
    real intensity=0;
```

We want to have the ray coordinates in the local system of this light:

```
    TVector3 origin=TransformPoint(fTransforms[lightIndex],rayOrigin);
    TVector3 direction=TransformVector(fTransforms[lightIndex],rayDirection);
```

Let's calculate the intersection of the ray with the light cone. Since we have a starting point and a direction, we seek the distance (real number) we have to traverse to intersect the light cone.

```
    // intersection with the cone : x^2+y^2-z^2=0
    real a=direction.x*direction.x+direction.y*direction.y-direction.z*direc-
    tion.z;
    real b=direction.x*origin.x+direction.y*origin.y-direction.z*origin.z;
    real c=origin.x*origin.x+origin.y*origin.y-origin.z*origin.z;

    real delta = b*b-a*c;
    real t1,t2; // solutions of the intersection with the cone
```

We can discard the solutions with 0 or 1 point of intersection and only keep the 2-point solutions:

```
    if (delta>0.000001f && a!=0.0f)
    {
        delta=sqrt(delta);
        if (a>0.0f)
        {
            t1=(-b-delta)/a;
            t2=(-b+delta)/a;
        }
        else if (a<0.0f)
        {
            t1=(-b+delta)/a;
            t2=(-b-delta)/a;
        }
        real z1,z2; // intersection in local coordinates
        z1=t1*direction.z+origin.z;
        z2=t2*direction.z+origin.z;
```

Note that the z-axis of the light is oriented backwards. So, the points located in front of the light source have a negative z.

```
        if (z1<0.0 || z2<0.0) // good side of the cone
        {
```

Here you handle special cases and calculate the length of the ray inside the cone.
You can then calculate the intensity of the effect (including a falloff parameter if desired).

```
            const real t = fLightData[lightIndex].fFallOff;
            intensity= (t2-t1) / ( (1.0f-t) * dist + t * dist*dist );
```

Now you calculate the energy gained along the ray:

```
            Rl=fLightColor[lightIndex].R * intensity;
            Gl=fLightColor[lightIndex].G * intensity;
            Bl=fLightColor[lightIndex].B * intensity;
        }
```

Finally, calculate the contribution of the current light to the filtering effect:

```
        const real lightIntensity = fLightData[lightIndex].fIntensity;
        Rt+=Rl*lightIntensity * fLightData[lightIndex].fFogColor.R;
        Gt+=Gl*lightIntensity * fLightData[lightIndex].fFogColor.G;
```

```
                Bt+=Bl*lightIntensity * fLightData[lightIndex].fFogColor.B;
            }
        }
        inOutFilter.R += Rt;
        inOutFilter.G += Gt;
        inOutFilter.B += Bt;
    }
```

Writing a Scene Command

Family ID : 'scmd'
Interface ID :IID_I3DExSceneCommand
Interface file : I3DExSceneCommand.h
Basic Implementation file: Basic3DCOMImplementations.h, Basic3DCOMImplementations.cpp

Overview

Scene Commands, formerly known as Scene Operations, add features to the scene. They work in a very similar manner to regular actions, with **Do()**, **Undo(),** and **Redo()** methods. They allow you to add menu items.
You can provide a 'scmd' resource to specify additional information.
If you give an Action Number, then no dynamic menu will be created, as it is expected that you will use a menu item defined in an existing menu. If the Action Number is -1, then you can specify in the next field the ID of the target menu. A dynamic menu item will be created in there. If the "Sub Family Name" of the COMP resource is not empty, then a hierarchical sub-menu will be created, and all commands with the same "Sub Family Name" will be grouped there. Then specify the list of the Rooms in which the Scene Commmand is valid. Supplying no name means "all".
Note : One instance of each type of Scene Command will be created at the application startup. This instance will be used only for menu enabling purposes by calling the **SelfPrepareMenus()** method. All other methods of these instances will NEVER be called, including **Init()**. Therefore, the I3DExSceneCommand::**SelfPrepareMenus()** procedure should be completely self-sufficient.
The following methods inherited from IExDataExchanger must be implemented:

- SelfPrepareMenus(ISceneDocument* sceneDocument)
- Init(ISceneDocument* sceneDocument)
- Prepare()

If you plan to support undo/redo functionality, you need to implement the following functions:

- CanUndo()
- Do()
- Undo()
- Redo()

See the reference manual for more specific information on the interface methods.

Location in the User Interface

Scene commands typically add items to one of the menus.

Example: StairCommand

Description

This sample shows how to create a stairway with every selected object. To do this, you need to duplicate and translate an object.

Parameters

The following variables are used for the user-input parameters:

```
fApplyModifier The modifier selected by the user.
fNbStep:       The number of steps.
fDx:           The relative position of x.
```

fDy: **The relative position of y.**
fDz: **The relative position of z.**

These are defined in StairCommand.h as follows:

```
struct StairData
{
    IMCUnknown* fApplyModifier;
    int16 fNbStep;
    real fDx;
    real fDy;
    real fDz;
};
```

These are then mapped to the user interface parameters using the PMap resource in StairCommand.r. Note that the parameters within the *StairData* struct must be in the same order as in the PMap.

Functions

StairCommand::SelfPrepareMenus

```
boolean  StairCommand::SelfPrepareMenus(ISceneDocument* sceneDocument)
```

See note above. **SelfPrepareMenus()** is called ONLY with the one instance created at the application start-up. Return true if the menu item should be enabled. Use 'sceneDocument' to get the selection or any other Scene data you would need to make this decision. Note that this method needs to be self- sufficient (i.e. not rely on anything else but the 'sceneDocument' parameter).

```
boolean  StairCommand::SelfPrepareMenus(ISceneDocument* sceneDocument)
{
    return (sceneDocument != NULL);
}
```

StairCommand::Init

```
MCCOMErr StairCommand::Init(ISceneDocument* sceneDocument)
```

Init() is called after the Scene Command is instantiated. Use sceneDocument to get the selection or any other Scene data you would need to perform your action.

```
MCCOMErr StairCommand::Init(ISceneDocument* sceneDocument)
{
    fSceneDocument = sceneDocument;
    sceneDocument -> GetSceneSelection(&fSelection);
    fSelection -> Clone(&fCloneSelection);
    fSceneDocument -> GetSceneSelectionChannel(&fSceneSelectionChannel);

    fData.fNbStep = 5;
    fData.fDx = 0.0;
    fData.fDy = 1.0;
    fData.fDz = 1.0;
    return MC_S_OK;
}
```

StairCommand::Prepare

```
MCCOMErr StairCommand::Prepare()
```

Prepare() allows you to perform any preparation work needed (even posting a dialog). Make sure you return MC_S_OK if it is OK to proceed with the Scene Command. If your Scene Command has a PMap resource and the corresponding 'Node' UI resources, then the application will create a modal dialog for you just after **Prepare()**.

```
MCCOMErr StairCommand::Prepare()
{
    TMCCountedPtr<ISceneSelection>selection;
    if (fSceneDocument)
    {
```

```
            fSceneDocument->GetSceneSelection(&selection);
            fSceneDocument->GetScene(&fScene);
        }

        if (!fTree)
        {
            TMCCountedPtr<I3DShGroup> group;
            fScene->GetTreeRoot(&group);
            if (group->QueryInterface(IID_I3DShTreeElement, (void **) &fTree) !=
            MC_S_OK)
                return MC_S_OK;
        }
        return MC_S_OK;
    }
```

StairCommand::Do

```
boolean StairCommand::Do()
```

The **Do()** function returns true if you actually changed something in the Scene.

```
boolean StairCommand::Do()
{
    TMCCountedPtr<I3DShTreeElement>newStep;
    TTreeTransform      stepTransform;
    TVector3               Offset;
    TMCCountedPtr<I3DShModifier>fModifier;

    TTreeSelectionIterator iter(fSelection);

    Offset[0] = 0.0f;// we initialize the offset
    Offset[1] = 0.0f;
    Offset[2] = 0.0f;

    for(I3DShTreeElement* iTree=iter.First(); iter.More(); iTree=iter.Next())
    {
        int16 iStep;
        for (iStep=0; iStep<fData.fNbStep; iStep++)
        {
            iTree->ComClone(&newStep, kWithAnim, true);

            if (!newStep)
                return false;

            iTree->InsertRight(newStep);

            newStep->GetLocalTreeTransform(stepTransform);

            Offset[0] += ((real)iStep) * fData.fDx;
            Offset[1] += ((real)iStep) * fData.fDy;
            Offset[2] += ((real)iStep) * fData.fDz;

            stepTransform.SetOffset(Offset);
            newStep->SetLocalTreeTransform(stepTransform);

            if(fApplyModifier)  //If the user choosed a modifier, we apply it on
        the new instances.
            {
                fApplyModifier->QueryInterface (IID_I3DShModifier,(void**)&fModi-
        fier);
                newStep->InsertModifier(fModifier, -1, kWithAnim);
            }
        }
```

```
    }
    return true;
}
```

StairCommand::CanUndo

```
boolean StairCommand::CanUndo()
```

CanUndo() returns true if the Scene Command is undoable. In this example, it is not undoable.

```
boolean StairCommand::CanUndo()
{
    return false;
}
```

StairCommand::Undo

```
boolean StairCommand::Undo()
```

The **Undo()** function returns true if you actually changed something in the Scene.

```
boolean StairCommand::Undo()
{
    return false;
}
```

StairCommand::Redo

```
boolean StairCommand::Redo()
```

The **Redo()** function returns true if you actually changed something in the Scene.

```
boolean StairCommand::Redo()
{
    return false;
}
```

Writing a Tweener

Family ID : 'twee'
Interface ID : IID_I3DExTweener
Interface file : 'I3DExTweener.h'
Basic Implementation file: Basic3DCOMImplementations.h, Basic3DCOMImplementations.cpp

Overview

A Tweener interpolates between two key frames. It positions an object along the path created by the key frames by returning a value between 0.0 and 1.0 to represent a percentage of the total distance.
The *I3DExTweener* interface has the following methods:

- **SimpleTween(real &result, int32 time, int32 time1, int32 time2)**

Simple examples implement **SimpleTween()** with **Tween()** returning *MC_E_NOTIMPL*. The variable *result* should be set to a value between 0.0 and 1.0 to represent a percentage of the total distance between key frames. Values less than 0.0 or greater than 1.0 can be used to 'overshoot' the key frame positions. The variables *time*, *time1*, and *time2* represent the current time, the time of the first key frame, and the time of the second key frame respectively.

- **Tween(I3DShKeyFrame *res, int32 time, I3DShTweenerChainLink *alink)**

Tween() is more powerful than **SimpleTween()**. **Tween()** has access to the entire chain, current key frame, and the time. Anything from the timeline can be used, such as previous key frames and tweeners.
If **Tween()** is implemented **SimpleTween()** will not be called.

Location in the User Interface

Tweeners are edited in the Properties Tray after selecting the area between two key frames within the Sequencer.

Example: Tweener

Description

This Tweener is a variation of the Oscillate tweener. This tweener can 'overshoot' the end point, as well as gradually decrease the amount of oscillation. The formula for this oscillator is :

$$f(t) = \cos\frac{t}{T} \cdot \exp(-r \cdot t)$$

Where T is a pseudo-period and r is used to calculate the decrease. A tweener must return 0.0 at the 1st frame and 1.0 at the last frame. To accommodate for this, the function is rewritten as:

$$f(t) = 1 - (\cos(a \cdot t) \cdot \exp(-r \cdot t))$$

$$\text{with} \quad a = 2\pi\left(NbOscillations + \frac{1}{4}\right)$$

This will be implemented in **SimpleTween()**. The variable *a* can be pre-calculated in **ExtensionDataChanged()** and is stored in the variable *fCosCoef*. The variable *r* is stored in *fExpCoef*.

Parameters

The following variables are used for the user-input parameters:

```
fNbOsc:     The number of Oscillations.
fExpCoef:   The variable 'r' in the formulae above.
```

These are defined is Tweener.h as follows:

```
struct TweenerData
{
    int16 fNbOsc;
    real fExpCoef;
};
```

These are then mapped to the user interface parameters using the PMap resource in Tweener.r. Note that the parameters within the *TweenerData* struct must be in the same order as in the PMap.

Functions

Tweener::ExtensionDataChanged

```
MCCOMErr Tweener::ExtensionDataChanged()
```

To avoid calculating *fCosCoef (a* in the formula*)* each time the Shell requests a value, it is pre-calculated in the function **ExtensionDataChanged()**:

```
MCCOMErr Tweener::ExtensionDataChanged()
{
    fCosCoef = 2 * PI * ( fData.fNbOsc + 0.25f );
    fExpCoef= (double)fData.fExpCoef;
    return MC_S_OK;
}
```

Tweener::SimpleTween

```
MCCOMErr Tweener::SimpleTween(real &result, int32 time, int32 time1, int32
        time2)
```

SimpleTween() implements the formula above using *time*, *time1*, and *time2* to calculate *t*, *fCosCoef* for *a*, and *fExpCoef* for *r*. The output is returned in *Result*:

```
MCCOMErr Tweener::SimpleTween(real &result, int32 time, int32 time1, int32
        time2)
{
    int32 delta = time2 - time1;
    result= 0;
    if (delta == 0)
    {
        return MC_S_OK;
    }
    real t = (1.0f * time - time1) / delta;
    result  = (1.0f - ::cos(t*fCosCoef) * ::exp( -t * fExpCoef));
    return MC_S_OK;
}
```

Index