

Carrara SDK

Extension Architecture

Reference Manual

©2001 Eovia
All rights reserved

Version 1.1
Preliminary Draft August 2001

Copyright ©2001 Eovia, Corp. All rights reserved.

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 Eovia, Corp.

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 Eovia, Corp. Eovia, Corp. 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 Eovia 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 Eovia 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.

Eovia 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 Eovia HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Chapter 1 - Table of Contents

Chapter 1 - Table of Contents 3

Chapter 2 - MCCommon IEx* Reference 15

2.1 IExtension Interface 15

2.2 IExDataExchange Interface 16

2.3 IExternalView Interface 21

2.4 IExDataExchange Interface 33

IExDataExchange::Prepare(IExDataExchange**, IMCUnknown*) 33

IExDataExchange::IsEqual(IExDataExchange*) 33

IExDataExchange::GetParameter(IDType, void*) 33

IExDataExchange::SetParameter(IDType, void*) 34

IExDataExchange::GetExtensionDataBuffer() 34

IExDataExchange::ExtensionDataChanged() 34

IExDataExchange::HandleEvent(MessageID, IMFResponder*, void*) 34

IExDataExchange::GetResID() 34

2.5 I3DExExportFilter Interface 37

I3DExExportFilter::Prepare(I3DShScene*, I3DShElement*) 37

I3DExExportFilter::WantsOptionDialog() 37

I3DExExportFilter::NeededDiskSpace(IMCFile*, I3DShScene*, I3DShTreeElement*, uint32*) 37

I3DExExportFilter::DoExport(IMCFile*, I3DShScene*, I3DShElement*) 38

I3DExExportFilter::WantsPreview() 38

I3DExExportFilter::GetPreviewSize(int16*, int16*) 38

I3DExExportFilter::SetPreview(TMCPixelBucket*) 38

2.6 I3DExImportFilter Interface 41

I3DExImportFilter::Prepare(IMCFile*, I3DShScene*, I3DShElement*) 41

I3DExImportFilter::WantsOptionDialog() 41

I3DExImportFilter::DoImport(IMCFile*, I3DShScene*, I3DShElement*) 41

I3DExImportFilter::WantsTopScene() 42

2.7 IExRegisterer Interface 43

IExRegisterer::GetComponentCount() 43

IExRegisterer::GetComponentData(uint32, APICOMPData&, APIPrivateData&, APIGUIDData&) ... 43

IExRegisterer::UnregisterComponents() 43

Chapter 3 - Data Structures Reference. 45

TVECTOR2 45

TVECTOR3 47

TBBBOX2D 47

TBBBOX3D 48

TMATRIX33 48

TTRANSFORM3D 51

TTREETRANSFORM 52

TMCCOLORRGB 54

ETreeDeformBBoxModes	56
----------------------------	----

Chapter 4 - Shading Data Structures Reference 57

ShadingIn	57
ShadingFlags	59
ShadingOut	61
UVSpaceInfo	62
ShadedArea	62

Chapter 5 - Geometry Data Structures Reference. 63

TVERTEX3D	63
TFACET3D	63
TPATCH3D	63
FacetMesh	64

Chapter 6 - Ray Tracing Data Structures Reference 65

Ray3D	65
RayHit3D	66
RayHitParameters	67
HitInfo	68

Chapter 7 - 3D IEx* Reference 71

7.1 I3DExAmbientLight Interface..... 73

I3DExAmbientLight::GetColor	73
-----------------------------------	----

7.2 I3DExAtmosphericShader Interface..... 75

I3DExAtmosphericShader::SegmentFilter	75
I3DExAtmosphericShader::DirectionFilter	75

7.3 I3DExBackground Interface 77

I3DExBackground::GetBackgroundColor	77
---	----

7.4 I3DExBackdrop Interface 79

I3DExBackdrop::GetBackdropColor	79
---------------------------------------	----

7.5 I3DExCamera Interface 81

I3DExCamera::SetTransform(const TTransform3D&)	81
I3DExCamera::CreateRay(const TVector2&, TVector3&, TVector3&)	82
I3DExCamera::CreateRay(const TVector2&, const TVector2&, Ray3D&)	82
I3DExCamera::IsProjectionAffine(void)	82
I3DExCamera::PreProject3DTo2D(const TVector3&, TVector4&)	83
I3DExCamera::Project3DTo2D(const TVector3&, TVector3&)	83
I3DExCamera::GetPrimitiveID()	83
I3DExCamera::GetStandardCameraInfo(TStandardCameraInfo&)	83
I3DExCamera::SetHalfWidth(real)	84

7.6 I3DExConstraint 87

I3DExConstraint::GetNbrFreedom()	87
I3DExConstraint::IncrementFreedomValue(int16 , real&)	87
I3DExConstraint::GetMotionLimit (int16, ELimitType&)	87
I3DExConstraint::GetFreedomRange(int16, real&, real&)	88
I3DExConstraint::GetTransform(TTransform3D&)	88
I3DExConstraint::GetTransformPartialDerivate(int16, TTransform3D&)	88

I3DExConstraint::GetTransformSecondPartialDerivative (int16, int16, TTransform3D&)	89
I3DExConstraint::SetTransform(TTransform3D&)	89
I3DExConstraint::IsTransformResetRequired()	89
I3DExConstraint::ResetConstraint()	89
7.7 I3DExDataComponent	91
I3DExDataComponent:: GetPostRenderList(TMCArray<IDType> &)	91
I3DExDataComponent::IsActive(I3DShTreeElement*)	91
7.8 I3DExFinalRenderer Interface	93
I3DExFinalRenderer::SetTreeTop	93
I3DExFinalRenderer::SetCamera	93
I3DExFinalRenderer::SetEnvironment	94
I3DExFinalRenderer::SetAmbientLight	94
I3DExFinalRenderer::Dehydrate	94
I3DExFinalRenderer::GetRenderingTime	94
I3DExFinalRenderer::SetFieldRenderingData	94
I3DExFinalRenderer::GetRenderStatistics	95
I3DExFinalRenderer::PrepareDraw	95
I3DExFinalRenderer::DrawRect	96
I3DExFinalRenderer::FinishDraw	96
I3DExFinalRenderer::IsSMPAware	97
I3DExFinalRenderer::GetPreferredTiling	97
7.9 I3DExInvertibleTween Interface	99
I3DExTween::Inverse()	99
7.10 I3DExLightsource Interface.....	101
I3DExLightsource::SetTransform(TTransform3D*)	101
I3DExLightsource::GetDirection(TVector3&, TVector3&, real&)	101
I3DExLightsource::GetColor(TVector3&, TVector3&, real, TMCColorRGB&, real&)	102
I3DExLightsource::IsVisibleInPerspective()	102
I3DExLightsource::GetPrimitiveID()	102
I3DExLightsource::ForEachShadowBuffer(ForEachShadowBufferCallback, void*)	103
I3DExLightsource::GetLightParameter(int32, void*)	103
I3DExLightsource::GetLightInfo(boolean&, boolean&, real&, TTransform3D&)	103
I3DExLightsource::GetUVfromDirection(TVector2&, const TVector3&)	103
I3DExLightsource::GetNewTrace(int16*)	104
I3DExLightsource::GetStandardLight(TStandardLight&)	104
7.11 I3DExLightsourceGel Interface.....	105
I3DExLightsourceGel::GetGelValues(const TVector2&, TMCColorRGB&)	105
7.12 I3DExModifier Interface	107
I3DExModifier::Apply(I3DShTreeElement)	107
I3DExModifier::GetMaxTimeWithoutComputation (I3DShTreeElement*)	107
I3DExModifier::SetBoundingBox(const TBBBox3D&)	107
I3DExModifier::DeformPoint(TVector3&, TVector3&)	108
I3DExModifier::DeformVertex3D(TVertex3D&, TVertex3D&)	108
I3DExModifier::DeformFacetMesh(real, FacetMesh*, FacetMesh**)	108
I3DExModifier::DeformBBox(const TBBBox3D&, TBox3D&)	108
I3DExModifier::GetNeededLOD(const real, real&)	109
I3DExModifier::GetDeformerInfo(real, TDeformInfo&)	109
7.13 I3DExModule Interface	111
I3DExModule::Initialize(IMCUnknown*)	112
I3DExModule::Destroy()	112
I3DExModule::Hydrate()	112
I3DExModule::Dehydrate()	113
I3DExModule::Activate()	113

I3DExModule::Deactivate()	113
I3DExModule::Import(IMCUnknown*, void*)	113
I3DExModule::BuildMenuBar()	113
I3DExModule::GetPreferredRoom(IDType&)	114
I3DExModule::WantsToBeInCurrentRoom()	114
I3DExModule::SubModuleClosedMainWindow(I3DShModule*)	114
I3DExModule::SubModuleAboutToBeDestroyed(I3DShModule*)	114
I3DExModule::SubModuleAboutToBeHydrated(I3DShModule*)	114
I3DExModule::AboutToCloseMainWindow()	115
I3DExModule::SavesInNativeFormat()	115
7.14 I3DExPostRenderer Interface	117
I3DExPostRenderer::GetBufferNeeds(uint32&, void*)	117
I3DExPostRenderer::PrepareDraw(I3DShRasterLayer*, const TMCRect&, I3DShChannel*, I3DShScene*, I3DShCamera*, TBBBox2D&, TBBBox2D&)	118
I3DExPostRenderer::DrawRect(const TMCRect&, const TChannelDataBucket*, TBBBox2D&)	119
I3DExPostRenderer::FinishDraw()	119
I3DExPostRenderer::BeginRender(MicroTick, MicroTick, unit32, uint32, boolean)	119
I3DExPostRenderer::EndRender()	119
7.15 I3DExGeometricPrimitive Interface	121
I3DExGeometricPrimitive::GetBoundingBox(TBox3D*)	121
I3DExGeometricPrimitive::EnumPatches(EnumPatchesCallback, void*, boolean&)	122
I3DExGeometricPrimitive::GetNbrLOD(int16&)	122
I3DExGeometricPrimitive::GetLOD(int16, real&)	122
I3DExGeometricPrimitive::GetFacetMesh(uint32, FacetMesh**)	123
I3DExGeometricPrimitive::GetFacetMesh(real, FacetMesh**)	123
I3DExGeometricPrimitive::CanBeSplit()	123
I3DExGeometricPrimitive::SplitPrimitive(TMCCountedPtrArray<I3DExGeometricPrimitive>, TMCCArray<TTransform3d>&)	124
I3DExGeometricPrimitive::GetUVSpaceCount()	124
I3DExGeometricPrimitive::GetUVSpace(uint32, UVSpaceInfo*)	124
I3DExGeometricPrimitive::UV2XYZ(TVector2*, uint32, TVector3*, boolean*)	124
I3DExGeometricPrimitive::GetUVSpaceRDS5(uint32, UVSpaceInfoRDS5*)	125
I3DExGeometricPrimitive::AppendToRenderables(const TTransform3D&, TRenderableAndTfmArray&)	125
I3DExGeometricPrimitive::AutoSwitchToModeler()	125
I3DExGeometricPrimitive::IsBBBoxPickable()	126
7.16 I3DExReflectionFeature	127
I3DExReflectionFeature::GetReflectionNeeds(uint32 &)	127
I3DExReflectionFeature::CalcReflection(RenderFeatureIO&)	127
7.17 I3DExRefractionFeature	129
I3DExRefractionFeature::GetRefractionNeeds(uint32 &)	129
I3DExRefractionFeature::CalcRefraction(RenderFeatureIO&)	129
7.18 I3DExShader Interface	131
I3DExShader::IsEqualTo(I3DExShader*)	131
I3DExShader::GetShadingFlags(ShadingFlags&)	132
I3DExShader::GetImplementedOutput()	132
I3DExShader::DoShade(ShadingOut&, ShadingIn&)	133
I3DExShader::GetValue(real&, boolean, ShadingIn&)	133
I3DExShader::GetColor(TMCColorRGB&, boolean, ShadingIn&)	133
I3DExShader::GetVector(TVector3&, ShadingIn&)	134
I3DExShader::ConvertToPoly(TMCCArray<TVector2>*)	134
I3DExShader::OutlineInPreviewMode()	134
I3DExShader::Contains(ShadingIn&, TVector2&)	134
7.19 I3DExTweeners Interface	135

I3DExTween::SimpleTween(real&, int32, int32, int32)	135
I3DExTween::Tween(I3DShKeyFrame*, int32, I3DShTweenChainLink*)	135
7.20 I3DExTweenEdition Interface	137
I3DExTween::PrepareEdit(I3DShTweenChainLink*)	137
7.21 I3DExWireFrame Interface	139
I3DExWireFrame::TrackWireFrame(I3DShWireFrame*, int16, I3DShTreeElement*, int16, const TRACKINFO&, const TRACKINFO&, const TRACKINFO&)	139
I3DExWireFrame::DataToWireFrame(I3DShWireFrame*, int16, I3DShTreeElement*)	140
I3DExWireFrame::WireFrameToData(I3DShWireFrame*, int16, I3DShTreeElement*)	140
I3DExWireFrame::GetWireFrameBBBox(I3DShTweenChainLink*)	140
I3DExWireFrame::Activate()	141
I3DExWireFrame::Deactivate()	141
I3DExWireFrame::HandlesTool(int16)	141
7.22 IRaytracablePrimitive Interface	143
IRaytracablePrimitive::RayHit(boolean*, Ray3D*, RayHitParameters*, RayHit3D*)	143
IRaytracablePrimitive::GetRayHitDetails(RayHit3D*)	144
IRaytracablePrimitive::IsInfiniteBB(boolean&)	144
7.23 ISceneDocument Interface	145
ISceneDocument::GetScene(I3DShScene**)	145
ISceneDocument::GetSceneSelection(ISceneSelection**)	145
ISceneDocument::GetSceneSelectionChannel(IChangeChannel**)	145
ISceneDocument::GetTreeHierarchyChannel(IChangeChannel**)	146
ISceneDocument::GetDocumentWindowByID(IMFPart**, IDType)	146
ISceneDocument::GetModuleByID(I3DShModule**, IDType)	146
ISceneDocument::GetSceneRenderingModule(I3DRenderingModule**)	146
ISceneDocument::StopAnimPreview()	146
ISceneDocument::StartAnimPreview()	146
7.24 IVolumePrimitive Interface	149
IVolumePrimitive::GetVolumeDensity(const TVector3&, TMCCColorRGB&, TMCCColorRGB&)	149
IVolumePrimitive::GetVolumeAttenuation(const TVector3&, const TVector3&, TMCCColorRGB&) ..	150

Chapter 8 - MCCommon ISh* Reference 151

8.1 IShAction Interface..... 153

IShAction::Do()	153
IShAction::Undo()	153
IShAction::Redo()	153
IShAction::WillCauseChange()	153
IShAction::CanUndo()	154
IShAction::GetPartToRedraw(IMFPart**, int32)	154
IShAction::GetName(TMCString&)	154
IShAction::NeedsToPrepareMenus(void)	154

8.2 IShActionManager Interface

IShActionManager::PostAction(IShAction*, int32, IMFResponder*)	155
IShActionManager::PostMouseAction(IShMouseAction*, int32, IMFResponder* IMFPart*, const TMCPoint&)	155
IShActionManager::IsOptionDown()	156
IShActionManager::IsShiftDown()	156
IShActionManager::IsCommandDown()	156
IShActionManager::IsControlDown()	156
IShActionManager::IsLongStaticClick()	156
IShActionManager::IsShortStaticClick()	156

8.3 IShComponent Interface

IShComponent::GetFamilySignature()	157
IShComponent::GetClasSignature()	157
IShComponent::GetClassName(TMCString&)	157
IShComponent::IsDefaultComponent()	157
IShComponent::InvalidatePreview()	158
8.4 IShComponentUtilities Interface	159
IShComponentUtilities::CoCreateInstance(const MCCLSID&, IMCUnknown*, uint32, const MCID&, void**)	159
IShComponentUtilities::CreateComponent(IDType, IDType, IShComponent**)	160
IShComponentUtilities::CreateIUnknown(IDType, IDType, IMCUnknown*, IMCUnknown**)	160
IShComponentUtilities::CreateDefaultComponent(IDType, IShComponent**)	160
IShComponentUtilities::GetComponentClassInfoArrayByFamilyID(IDType, TMCArray<TComponentClassInfo>&)	161
IShComponentUtilities::WriteComponent(IShTokenStream*, IShParameterComponent*)	161
IShComponentUtilities::WriteComponent(IShTokenStream*, IDType, IShParameterComponent*)	161
IShComponentUtilities::ReadComponent(IShTokenStream*, IShParameterComponent**)	161
IShComponentUtilities::ReadComponent(IShTokenStream*, IShParameterComponent**, IDType)	162
IShComponentUtilities::ReadComponentData(TMCiostream*, IShParameterComponent*)	162
IShComponentUtilities::AskForParams(IShParameterComponent*)	162
8.5 IShParameterComponent Interface	163
IShParameterComponent::GetFamilySignature()	163
IShParameterComponent::GetClasSignature()	163
IShParameterComponent::GetClassName(TMCString&)	163
IShParameterComponent::GetParameter(IDType, void*)	164
IShParameterComponent::SetParameter(IDType, void*, boolean)	164
IShParameterComponent::GetParameterType(IDType, IDType&)	164
IShParameterComponent::GetParameterCount()	164
IShParameterComponent::GetParameterInfo(uint32, TMapElement&)	164
IShParameterComponent::Inval(boolean, boolean)	165
8.6 IShUtilities Interface	167
IShUtilities::BeginProgress(TMCString&, IMCUnknown**, real32)	168
IShUtilities::EndProgress(IMCUnknown*)	168
IShUtilities::SetProgressPrompt(TMCString&, IMCUnknown*)	168
IShUtilities::IncrementProgress(real32, IMCUnknown*)	168
IShUtilities::SetProgressValue(real32, IMCUnknown*)	169
IShUtilities::SetProgressValue0to1(real32, IMCUnknown*)	169
IShUtilities::YieldProcesses(uint32)	169
IShUtilities::TickCount()	170
IShUtilities::Random()	170
IShUtilities::SetRandomSeed(uint32)	170
IShUtilities::RandomStep(int32)	170
IShUtilities::GetAPINumber()	170
IShUtilities::Gestalt(void*, uint32)	170
IShUtilities::DoAlert(EAlerts, TMCString&, TMCString&, TMCString&, TMCString&, int32, TMCString&, TMCString&, TMCString&)	171
IShUtilities::ErrorToString(MCCOMErr, TMCString&)	172
IShUtilities::BuildAlertMessage(MCCOMErr, TMCString&, TMCString&)	172
IShUtilities::GetResource(int32, int32, void**)	172
IShUtilities::SetupResources(int32, int32, void**)	173
IShUtilities::RestoreResources(void*)	173
IShUtilities::BlitUIImage(IMCGraphicContext*, TMCRect&, TMCRect&, uint32)	173
IShUtilities::GetUIImageInfo(uint32, int32&, int32&, int32)	173
IShUtilities::GetUIImageBucket(uint32, TMCPixelBucket*)	174
IShUtilities::GetUIImage(uint32, TMCUIImage**)	174
IShUtilities::CreateCompatiblePixelBucket(int32, int32, int32, TMCCompatiblePixelBucket**)	174
IShUtilities::DestroyCompatiblePixelBucket(TMCCompatiblePixelBucket*)	175

IShUtilities::GetAsciiTranslationTable()	175
IShUtilities::GetOpenDocumentsCount()	175
IShUtilities::GetOpenDocumentByIndex(IMFDocument**, index)	175
IShUtilities::GetLastOpenDocPath(TMCString&)	175
IShUtilities::GetLastSaveDocPath(TMCString&)	175
IShUtilities::SetLastOpenDocPath(TMCString&)	176
IShUtilities::SetLastSaveDocPath(TMCString&)	176
IShUtilities::UseMMX()	176
IShUtilities::IsPentiumProcessor()	176
IShUtilities::IsPentiumIIProcessor()	176
IShUtilities::IsPentiumIIIProcessor()	176
IShUtilities::StringWidth(TMCTextStyle&, TMCString&)	176
IShUtilities::GetTextMetrics(TMCTextStyle&, TMCTextMetrics&)	177
IShUtilities::GetCreator()	177
IShUtilities::SetCursor(int32)	177
IShUtilities::GetFontList(IMCFontList**)	177
IShUtilities::CheckForUserCancel()	177
IShUtilities::GetCurrentModifiers()	177
IShUtilities::CheckKeyDown(int16)	178

Chapter 9 - 3D ISh* Reference 179

9.1 I3DShAtmosphericShader Interface 181

I3DShAtmosphericShader::GetScene(I3DShScene**)	181
--	-----

9.2 I3DShBackdrop Interface 183

I3DShBackdrop::GetScene(I3DShScene**)	183
---------------------------------------	-----

9.3 I3DShBackground Interface 185

I3DShBackground::GetScene(I3DShScene**)	185
---	-----

9.4 I3DShCamera Interface 187

I3DShCamera::GetGlobalToCameraTransform(TTTransford3d*)	187
I3DShCamera::Preset(uint32, const TVector3*, real)	187
I3DShCamera::Project3DTo2D(const TVector3*, TVector2*, real*)	188
I3DShCamera::CreateRay(const TVector2*, TVector3*, TVector3*)	188
I3DShCamera::GetRenderUVBox(TBBox2d*, uint32, uint32, real)	189
I3DShCamera::SetCameraComponent(int32)	189
I3DShCamera::SetCameraComponent(IShComponent*, ECloneMode)	189
I3DShCamera::GetCameraComponent(IShComponent**)	189
I3DShCamera::PreProject3DTo2D(const TVector3&, TVector4&)	190
I3DShCamera::IsProjectionAffine(void)	190
I3DShCamera::GetStandardCameraInfo(TStandardCameraInfo&)	190
I3DShCamera::SetCanonicalCameraType(ECanonicalCameraType)	191
I3DShCamera::GetCanonicalCameraType()	191
I3DShCamera::SetProductionOffset(const TVector2&)	192
I3DShCamera::GetProductionOffset()	192
I3DShCamera::SetProductionZoom(real)	192
I3DShCamera::GetProductionZoom()	192

9.5 I3DShConstraint 193

9.6 I3DShEnvironment Interface 195

I3DShEnvironment::SegmentFilter(const TVector3&, const TVector3&, TMCCColorRGB&, boolean)	195
I3DShEnvironment::SegmentFilter(const TVector3&, const TVector3&, TMCCColorRGB&)	196
I3DShEnvironment::DirectionFilter(const TVector3, const TVector3&, TMCCColorRGB&)	196
I3DShEnvironment::GetBackdropColor(const TVector2&, boolean&, const TBBox2D&, TMCCColorRGB&)	196
I3DShEnvironment::GetEnvironmentColor(const TVector3&, boolean&, TMCCColorRGB&)	197

I3DShEnvironment::HasAtmosphere()	197
I3DShEnvironment::HasBackdrop()	197
9.7 I3DShExternalPrimitive Interface	199
I3DShExternalPrimitive::SetPrimitiveComponent(I3DShComponent*, ECloneMode)	199
I3DShExternalPrimitive::GetPrimitiveComponent(I3DShComponent**)	199
9.8 I3DShObject Interface	201
I3DShObject::SetName(const TMCString&)	201
I3DShObject::GetName(TMCString&)	201
I3DShObject::GetInstancesCount()	202
I3DShObject::GetInstanceArray(TMCCountedPtrArray<I3DShInstance>&)	202
I3DShObject::GetUVSpaceCount()	202
I3DShObject::GetUVSpace(uint32, UVSpaceInfo*)	202
I3DShObject::GetScene(I3DShScene**)	202
I3DShObject::IsInfiniteObject()	203
I3DShObject::GetBoundingBox(TBBox3D&)	203
I3DShObject::GetFacetBoundingBox(TBBox3D&)	203
I3DShObject::Clone(I3DShObject**, ECloneMode) const	203
I3DShObject::GetEditor()	203
I3DShObject::GetObjectID()	204
9.9 I3DShPatchArray	205
I3DShPatchArray::Init(uint32, uint32)	205
I3DShPatchArray::SetPatch(uint32, uint32, TPatch3D*)	205
I3DShPatchArray::GetPatch(uint32, uint32, TPatch3D*)	206
I3DShPatchArray::MeshSetU(uint32, real)	206
I3DShPatchArray::MeshSetV(uint32, real)	207
I3DShPatchArray::MeshGetU(uint32, real*)	207
I3DShPatchArray::MeshGetV(uint32, real*)	207
I3DShPatchArray::MeshCalcUV()	207
I3DShPatchArray::GetSize(uint32&, uint32&)	208
I3DShPatchArray::BuildFacetMesh(real, FacetMesh**)	208
9.10 I3DShPatchList	209
I3DShPatchList::Init(boolean)	209
I3DShPatchList::AddPatch(TPatch3D*)	209
I3DShPatchList::GetIndPatches(uint32)	210
I3DShPatchList::CountPatches()	210
I3DShPatchList::SetHasUVSpace(boolean)	210
I3DShPatchList::SetUVSpace(UVSpaceInfo*)	210
I3DShPatchList::GetUVSpace(uint32, UVSpaceInfo*)	211
I3DShPatchList::GetUVSpaceCount()	211
I3DShPatchList::BuildFacetMesh(real, FacetMesh**)	211
9.11 I3DShPolygonArray	213
I3DShPolygonArray::Init(uint32, uint32, boolean, boolean)	213
I3DShPolygonArray::MeshSetVertex(uint32, uint32, TVector3*, TVector3*)	214
I3DShPolygonArray::MeshGetVertex(uint32, uint32, TVector3*, TVector3*)	214
I3DShPolygonArray::MeshSetU(uint32, real)	215
I3DShPolygonArray::MeshSetV(uint32, real)	215
I3DShPolygonArray::MeshGetU(uint32, real*)	216
I3DShPolygonArray::MeshGetV(uint32, real*)	216
I3DShPolygonArray::MeshCalcUV()	216
I3DShPolygonArray::MeshCalcNormals(boolean)	217
I3DShPolygonArray::GetSize(uint32&, uint32&, boolean&, boolean&)	217
9.12 I3DShPolygonList Interface	219
I3DShPolygonList::Init(boolean, boolean)	219

I3DShPolygonList::PreAllocateFacets(uint32)	220
I3DShPolygonList::AddFacet(TFacet3D*)	220
I3DShPolygonList::CountFacets()	220
I3DShPolygonList::CalcNormals(real)	221
I3DShPolygonList::GetIndFacets(uint32)	221
I3DShPolygonList::SetHasUVSpace(boolean)	221
I3DShPolygonList::SetUVSpace(UVSpaceInfo*)	221
I3DShPolygonList::GetUVSpace(uint32, UVSpaceInfo*)	221
I3DShPolygonList::GetUVSpaceCount()	222
9.13 I3DShPostRenderer Interface	223
I3DShPostRenderer::SetScene(I3DShScene*)	223
I3DShPostRenderer::GetBufferNeeds(uint32&)	223
I3DShPostRenderer::PrepareDraw(I3DShRasterLayer*const TMCRect& I3DShChannel* I3DShScene* TBox2D& TBox2D&)	224
I3DShPostRenderer::DrawRect(const TMCRect&, const TChannelDataBucket*, TBox2D)	225
I3DShPostRenderer::FinishDraw()	225
I3DShPostRenderer::BeginRender(MicroTick, MicroTick, uint32, uint32, boolean)	226
I3DShPostRenderer::EndRender()	226
9.14 I3DShMasterShader Interface	227
I3DShMasterShader::SetName(const TMCString&)	227
I3DShMasterShader::GetName(TMCString&)	227
I3DShMasterShader::SetShader(I3DShShader*, ECloneMode)	227
I3DShMasterShader::GetShader(I3DShShader**)	228
I3DShMasterShader::Clone(I3DShMasterShader**, ECloneMode)	228
I3DShMasterShader::GetScene(I3DShScene**)	228
I3DShMasterShader::IsDefaultMaster()	228
9.15 I3DShPrimitive Interface	229
I3DShPrimitive::IsPatchBased()	229
I3DShPrimitive::EnumPatches(EnumPatchesCallback, void*, boolean)	229
I3DShPrimitive::ChangedData()	230
I3DShPrimitive::GetFMesh(real, FacetMesh**)	230
I3DShPrimitive::IsVolumic()	230
I3DShPrimitive::UV2XYZ(const TVector2&, int16, TVector3&, TVector3*)	230
9.16 I3DShPrimitiveComponent Interface	233
I3DShPrimitiveComponent::GetPrimitive(I3DShPrimitive**)	233
I3DShPrimitiveComponent::GetExternalPrimitive(IMCUnknown**)	233
9.17 I3DShShader Interface	235
I3DShShader::Clone(I3DShShader**, ECloneMode) const	235
I3DShShader::IsConstant()	235
I3DShShader::AssignsChannels()	236
I3DShShader::GetShadingFlags(ShadingFlags&)	236
I3DShShader::DoShade(ShadingOut&, ShadingIn&)	236
I3DShShader::GetValue(real&, boolean, ShadingIn&)	236
I3DShShader::GetColor(TMCColourRGB&, boolean, ShadingIn&)	237
I3DShShader::GetVector(TVector3&, ShadingIn&)	237
I3DShShader::GetBump(TVector3&, ShadingIn&)	237
I3DShShader::GetImplementedOutput()	238
I3DShShader::GetOriginalParametricTextureMap(uint32&, uint32&, I3DShTextureMap**)	238
9.18 I3DShTreeElement Interface	239
I3DShTreeElement::SetScene(I3DShScene*)	241
I3DShTreeElement::GetScene(I3DShScene**)	241
I3DShTreeElement::GetSceneNoAddRef()	241
I3DShTreeElement::SetName(const TMCString&)	241

I3DShTreeElement::GetName(TMCString&)	241
I3DShTreeElement::GetFirst(I3DShTreeElement**)	241
I3DShTreeElement::GetLast(I3DShTreeElement**)	242
I3DShTreeElement::GetLeft(I3DShTreeElement**)	242
I3DShTreeElement::GetRight(I3DShTreeElement**)	242
I3DShTreeElement::GetTop(I3DShTreeElement**)	242
I3DShTreeElement::Unlink(I3DShTreeElement**, RelinkTreeElementInfo*)	242
I3DShTreeElement::Relink(RelinkTreeElementInfo*)	242
I3DShTreeElement::InsertFirst(I3DShTreeElement*)	243
I3DShTreeElement::InsertLast(I3DShTreeElement*)	243
I3DShTreeElement::InsertLeft(I3DShTreeElement*)	243
I3DShTreeElement::InsertRight(I3DShTreeElement*)	243
I3DShTreeElement::GetHotPoint(TVector3&)	243
I3DShTreeElement::SetHotPoint(const TVector3&)	243
I3DShTreeElement::CenterElementOnHotPoint()	244
I3DShTreeElement::CenterHotPointOnElement()	244
I3DShTreeElement::TranslateHotPoint(TVector3&)	244
I3DShTreeElement::ComClone(I3DShTreeElement**, ECloneMode, boolean)	244
I3DShTreeElement::InvalidateBoundingBox()	244
I3DShTreeElement::GetBoundingBoxWithoutLightsAndCameras(TBBox3D&)	245
I3DShTreeElement::GetBoundingBox(TBBox3D&, ETreeDeformBBoxModes)	245
I3DShTreeElement::CanBeDistorted()	245
I3DShTreeElement::BeginMove()	245
I3DShTreeElement::PostMoveChange(boolean)	245
I3DShTreeElement::EndMove(boolean)	246
I3DShTreeElement::GetXYZScaling(TVector3&)	246
I3DShTreeElement::SetXYZScaling(TVector3&)	246
I3DShTreeElement::GetNbrFreedom()	246
I3DShTreeElement::GetFreedomRange(int16, real&, real&)	246
I3DShTreeElement::GetFreedomTransform(TMatrix33&, TVector3&)	247
I3DShTreeElement::GetFreedomDerivative(int16, TMatrix33&, TVector3&)	247
I3DShTreeElement::GetFreedomSecondDerivative(int16, int16, TMatrix33&, TVector3&)	247
I3DShTreeElement::IncrementFreedomValue(int16, real*)	248
I3DShTreeElement::GetFullName(TMCString&)	248
I3DShTreeElement::InsertModifier(I3DShModifier*, uint32, ECloneMode)	248
I3DShTreeElement::RemoveModifier(uint32)	248
I3DShTreeElement::GetModifiersCount()	248
I3DShTreeElement::GetModifierByIndex(I3DShModifier**, uint32)	249
I3DShTreeElement::SetModifiers(const TMCCountedPtrArray<I3DShModifier>*, ECloneMode)	249
I3DShTreeElement::LinksChanged()	249
I3DShTreeElement::BeginGetRenderables(const TRenderableAndTfmArray*&, boolean)	249
I3DShTreeElement::EndGetRenderables()	250
I3DShTreeElement::GetConstraint(I3DShConstraint**)	250
I3DShTreeElement::SetConstraint(I3DShConstraint*, ECloneMode)	250
I3DShTreeElement::SetConstraintByID(int32)	250
I3DShTreeElement::GetDataComponentsCount()	250
I3DShTreeElement::GetDataComponentByIndex(I3DShDataComponent**, uint32)	250
I3DShTreeElement::SetDataComponentByID(I3DShDataComponent*, ECloneMode)	251
I3DShTreeElement::GetDataComponentByID(I3DShDataComponent**, IDType)	251
I3DShTreeElement::InsertDataComponent(I3DShDataComponent*)	251
I3DShTreeElement::InsertDataComponentByID(int32, int32)	251
I3DShTreeElement::RemoveDataComponent(uint32)	252
I3DShTreeElement::SetData(TDataKey, IMCUnknown*)	252
I3DShTreeElement::GetData(TDataKey, IMCUnknown**)	252
I3DShTreeElement::ReleaseData(TDataKey)	252
I3DShTreeElement::IsOpen()	252

I3DShTreeElement::GetEditor()	252
I3DShTreeElement::GetAnimationMethod(I3DShAnimationMethod**)	253
I3DShTreeElement::SetAnimationMethod(I3DShAnimationMethod*, ECloneMode, real)	253
I3DShTreeElement::GetLocalTreeTransform(TTreeTransform&)	253
I3DShTreeElement::SetLocalTreeTransform(const TTreeTransform&)	253
I3DShTreeElement::GetGlobalTreeTransform(TTreeTransform&)	253
I3DShTreeElement::SetGlobalTreeTransform(const TTreeTransform&, int16, boolean)	254
I3DShTreeElement::GetGlobalTreeTransform3D(TTreeTransform3D&)	254
I3DShTreeElement::SetGlobalTransform3D(const TTransform3D&, int16)	254
I3DShTreeElement::GetFatherGlobalTransform3D(TTransform3D&)	254
I3DShTreeElement::CloneSubTree(ECloneMode, I3DShTreeElement**)	255
I3DShTreeElement::GetUndoInfo(TTreeUndoInfo&)	255
I3DShTreeElement::SetFromUndoInfo(const TTreeUndoInfo&, const TTreeUndoInfo*)	255
I3DShTreeElement::GetTopIfConstrained(I3DShTreeElement**)	255
I3DShTreeElement::AddWireFrame(I3DExWireFrame*)	255
I3DShTreeElement::RemoveWireFrame(I3DExWireFrame*)	256
I3DShTreeElement::WireFrameArrayFree()	256
I3DShTreeElement::GetWireFrameArray(TMCCClassArray<TWireFrameInfo>&)	256
I3DShTreeElement::IsPickable()	256
I3DShTreeElement::IsBBBoxPickable()	256
I3DShTreeElement::RemoveConstraint(I3DShConstraint**)	256
I3DShTreeElement::RestoreConstraint(I3DShConstraint*)	256
I3DShTreeElement::GetExists()	257
I3DShTreeElement::SetExists(boolean)	257
I3DShTreeElement::GetIsShown()	257
I3DShTreeElement::SetIsShown(boolean)	257
I3DShTreeElement::SetShadowCasting(boolean)	257
I3DShTreeElement::SetShadowCatching(boolean)	257
I3DShTreeElement::GetShadowCasting()	258
I3DShTreeElement::GetShadowCatching()	258
I3DShTreeElement::GetAnimateTree()	258
I3DShTreeElement::SetAnimateTree(boolean)	258
I3DShTreeElement::ClearInstanceCache()	258
9.19 I3DShTweeners Interface	259
I3DShTweeners::Tween(I3DShKeyFrame*, int32, I3DShTweenersChainLink)	259
I3DShTweeners::Edit(I3DShTweeners**, I3DShTweenersChainLink*)	259
I3DShTweeners::SetLink(I3DShTweenersChainLink*)	260
I3DShTweeners::GetLinkNoAddRef()	260
9.20 I3DShTweenersChainLink Interface	261
I3DShTweenersChainLink::Clone(I3DShTweenersChainLink**, I3DShParamTimeLine*)	261
I3DShTweenersChainLink::CopyData(I3DShTweenersChainLink*)	262
I3DShTweenersChainLink::GetTweenersNoAddRef(boolean, boolean)	262
I3DShTweenersChainLink::GetTweeners(I3DShTweeners**, boolean, boolean)	262
I3DShTweenersChainLink::SetTweeners(I3DShTweeners*)	262
I3DShTweenersChainLink::GetKeyFrameNoAddRef()	263
I3DShTweenersChainLink::GetKeyFrame(I3DShKeyFrame**)	263
I3DShTweenersChainLink::SetKeyFrame(I3DShKeyFrame**)	263
I3DShTweenersChainLink::GetTime()	263
I3DShTweenersChainLink::SetTime(MicroTick)	263
I3DShTweenersChainLink::GetNextNoAddRef()	263
I3DShTweenersChainLink::GetNext(I3DShTweenersChainLink**)	264
I3DShTweenersChainLink::SetNext(I3DShTweenersChainLink*)	264
I3DShTweenersChainLink::GetPrevNoAddRef()	264
I3DShTweenersChainLink::GetPrev(I3DShTweenersChainLink**)	264
I3DShTweenersChainLink::SetPrev(I3DShTweenersChainLink*)	264

I3DShTweenChainLink::GetFatherNoAddRef()	265
I3DShTweenChainLink::RemoveFromChain()	265
I3DShTweenChainLink::InsertAfter(I3DShTweenChainLink*)	265
I3DShTweenChainLink::OffsetLink(MicroTick)	265
I3DShTweenChainLink::GetTimeInfo(TTimeInfo&)	265
I3DShTweenChainLink::IsTweenSelected()	265
I3DShTweenChainLink::IsKeyFrameSelected()	265
I3DShTweenChainLink::SelectKeyFrame(boolean)	266
I3DShTweenChainLink::SelectTweenChainLink(boolean)	266
9.21 I3DShWireFrame Interface	267
I3DShWireFrame::AllocateFMesh(int32, int32, int32)	267
I3DShWireFrame::GetFMesh(FacetMesh**)	267
I3DShWireFrame::AllocateWireFrame(int16, int16)	268
I3DShWireFrame::GetWireFrame(WFData&)	268
I3DShWireFrame::Connect(int16, int16)	268
I3DShWireFrame::SetPoint(int16, EWFHandleType, const TVector3&)	268
I3DShWireFrame::Transform(const TTreeProjectionTransform&)	269
I3DShWireFrame::TestHit(TVector2&, real)	269
I3DShWireFrame::AddRenderables(TMCPtrArray<I3DShRenderable>&)	269
I3DShWireFrame::GetColor()	269
I3DShWireFrame::SetColor(const TMCCColorRGB&)	269
I3DShWireFrame::ExcludeFromZBuffer(boolean, boolean)	270
I3DShWireFrame::SetSelectedState(const boolean)	270
I3DShWireFrame::ExcludeFromProjection(boolean)	270
Chapter 10 - DataBase Overview	271
10.1 Scene	271
10.2 Coordinate Systems	273
10.3 Geometry	278
10.4 Color	285
Chapter 11 - Index	287

Chapter 2 - MCCommon IEx* Reference

2.1 IExtension Interface

Interface ID: IID_IExtension

Interface file: IExtension.h

The **IExtension** interface simply adds a Clone method and a generic Shell utilities initialization method to the **IUnknown** interface. All 3D Component interfaces inherit from this interface.

```
DECLARE_INTERFACE_(IExtension, IUnknown) {
    // IUnknown methods
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG, AddRef) (THIS) PURE;
    STDMETHOD_(ULONG, Release) (THIS) PURE;

    // IExtension methods
    STDMETHOD(ShellUtilitiesInit) (THIS_ IShUtilities* shellUtilities) PURE;
    STDMETHOD_(IExtension*, Clone) (THIS) PURE;
};
```

IExtension::ShellUtilitiesInit

HRESULT IExtension::ShellUtilitiesInit(*shellUtilities*);

IShUtilities* *shellUtilities*

IExtension::ShellUtilitiesInit is not called by the shell anymore. Call **InitCoFailure(*shellUtilities*)** in the **DllInitRDCom** function instead of this function. This will allow the failure handling to work between the Extension and the 3D Shell.

It used to be called by the 3D Shell just after the Component was instantiated to give to the Component access to some Shell services.

Parameter  *shellUtilities*
Pointer on the 3D Shell services object.

Comments Do NOT use this function. Use **InitCoFailure(*shellUtilities*)** instead.

IExtension::Clone

IExtension* **IExtension::Clone();**

IExtension::Clone is called when a component needs to be cloned.

Return Value A pointer on the cloned object through a IExtension interface.

Comments This method must absolutely be implemented. This routine needs to duplicate all private data you may have, and return a new component that has values identical to the former one, but is totally independent. In other words, they should not share any blocks in memory so they can be freed independently.

Do not clone the elements of the pMAP, it will be done by the shell.

Make sure you call AddRef() on the returned extension before giving it.

2.2 IExDataExchanger Interface

Interface ID: IID_IExDataExchanger9

Interface file: IExtension.h

The **IExDataExchanger** interface is responsible for handling the user interface of the component and communicating data between the 3D Shell and the 3D Component.

See the “Managing the User Interface” in the Overview for all details about implementing the user interface of your 3D Component.

```
DECLARE_INTERFACE_(IExDataExchanger9, IExtension) {
    // IUnknown methods
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG, AddRef) (THIS) PURE;
    STDMETHOD_(ULONG, Release) (THIS) PURE;

    // IExtension methods
    STDMETHOD(ShellUtilitiesInit) (THIS_ IShUtilities* shellUtilities) PURE;
    STDMETHOD_(IExtension*, Clone) (THIS) PURE;
    // IExDataExchanger methods
    STDMETHOD_(ExtensionDataMap*, GetExtensionDataMap) (THIS) PURE; // Return
        NULL if GetResID is to be used
    STDMETHOD_(void*, GetExtensionDataBuffer) (THIS) PURE;
    STDMETHOD(ExtensionDataChanged) (THIS) PURE;
    STDMETHOD(HandleEvent) (THIS_ ULONG sourceID, IShView* view, ULONG message)
        PURE;
    STDMETHOD_(short, GetResID) (THIS) PURE;
}
```

IExDataExchanger::GetExtensionDataMap

ExtensionDataMap* IExDataExchanger::GetExtensionDataMap();

IExDataExchanger::GetExtensionDataMap is called when the 3D Shell needs to learn about the data the 3D Component wishes to make public. This allows the 3D Shell to take care of the user interface of the 3D Component.

Return Value Returns a pointer on a **ExtensionDataMap** block. The 3D Shell will be responsible for deleting this block. Return NULL if you want the 3D Shell to use a ‘PMAP’ resource instead.
Reserved for future use. Do not implement if at this time, Make a pMAP or PMAP resource.

IExDataExchanger::GetExtensionDataBuffer

void* IExDataExchanger::GetExtensionDataBuffer();

IExDataExchanger::GetExtensionDataBuffer is called when the 3D Shell needs to get the address of the Component public data. This data must correspond to the mapping provided by the routine **IExDataExchanger::GetExtensionDataMap**.

Return Value	A pointer on the Data Buffer of the Component. This points to some internal structure inside the Component that was allocated when the Component was instantiated. The Component will free it when freeing itself.
Comments	The 3D Shell reads and write data directly in the Data Buffer using the Extension Data Map as a road map to calculate the addresses and to use the right data formatting.

IExDataExchanger::ExtensionDataChanged

HRESULT IExDataExchanger::ExtensionDataChanged();

IExDataExchanger::ExtensionDataChanged is called after the 3D Shell writes any new data in the Data Buffer. This allows the Component to perform any postprocessing or update any internal data based on this new information.

Return value S_OK

IExDataExchanger::HandleEvent

HRESULT IExDataExchanger::HandleEvent(sourceID);

ULONG sourceID

IExDataExchanger::HandleEvent is called every time an element of the user interface has sent the message mCallXCode = 2600.

Parameters  sourceID
The ID of the element that has been used. This ID is the one used in the 'View' resource to identify the element (button, check box, etc.)

Return value S_OK

Comments Use this call to react to button hits, and more generally to all controls that are action controls and not state controls. State controls like sliders, clusters of radio buttons, check boxes, etc. show the state of some data inside the Data Buffer. Buttons which are action trigger, do not correspond to anything in the Data Buffer because they go back to their original state after being clicked.

IExDataExchanger::GetResID

short IExDataExchanger::GetResID();

IExDataExchanger::GetResID is called when the 3D Shell needs to know the ID of all Component resources: 'COMP', 'PMAP', 'View', etc. Therefore, all resources of a component must have the same ID.

Return Value The resource ID number.
It MUST return the same value as the resource ID, and all resources MUST have the same ID.

IExDataExchanger9::HandleEvent

HRESULT IExDataExchanger9::HandleEvent(sourceID, view, message);

ULONG sourceID

IShView* view

ULONG message

IExDataExchanger9::HandleEvent is called every time an element of the user interface sends a message.

Parameters
 *sourceID*

The ID of the element that has been used. This ID is the one used in the ‘View’ resource to identify the element (button, check box, etc.)

 *view*

This is the view that recieved the message.

 *message*

This is the ID of the message. All the possible IDs are described in the table below.

Return value

S_OK

Comments

Use this call to react to button hits, and more generally to all controls that are action controls and not state controls. State controls like sliders, clusters of radio buttons, check boxes, etc. show the state of some data inside the Data Buffer. Buttons which are action trigger, do not correspond to anything in the Data Buffer because they go back to their original state after being clicked.

Table 1: Message IDs

ID	Constant	Meaning
0	kShNoEvent	No event to handle.
1	kShOKHT	Hit the default OK button.
2	kShCancelHT	Hit the default Cancel button.
3	kShButtonHT	Hit a button.
4	kShCheckBoxHT	Hit a check box.
5	kShClusterHT	Hit a cluster.
6	kShEditTextHT	Hit an edit text.
7	kShIconHT	Hit an icon.
8	kShListItemHT	Hit a listen item.
9	kShListScrollBarHT	Hit a list scroll bar.
10	kShPictureHT	Hit a picture.
11	kShPopupHT	Hit a pop-up.
12	kShRadioHT	Hit a radio button.
13	kShStaticTextHT	Hit a static text.
14	kShHScrollBarHT	Hit a horizontal scroll bar.
15	kShVScrollBarHT	Hit a vertical scroll bar.
16	kShEditTabKey	Press the tab key in an edit text.
17	kShEditReturnKey	Press the return key in an edit text.
18	kShEditEnterKey	Press the enter key in an edit text.

Table 1: Message IDs

ID	Constant	Meaning
19	kShPatternHT	Hit a pattern.
20	kShControlHT	Hit a control.
21	kShCancelKey	Press the cancel key.
22	kShDefaultKey	Press the default key.
23	kShSmallIconHT	Hit a small icon.
24	kShBecameWinFirstResponder	The window became first responder.
25	kShResignedAsWinFirstResponder	The window resigned as first responder.
26	kShBecameFirstResponder	The view became first responder.
27	kShResignedAsFirstResponder	The view resigned as first responder.
28	kShClosed	Closing message.
29	kShAboutToLoseControl	Clipboard is going to be emptied.
30	kShScrapChanged	Clipboard has just been filled.
31	kShOn	Set ON.
32	kShOff	Set OFF.
33	kShToggle	Toggle.
34	kShDismiss	Dismiss.
35	kShFreeClipboardView	Free clipboard.
36	kShClipViewChanged	Change clipboard.
37	kShRestoreDefault	Restore.
38	kShBeDefault	Set as default.
39	kShBeNotDefault	Un-set as default.
40	kShRemoveDefault	Remove default.
41	kShToolbarChanged	Toolbar changed.
42	kShUpSpinBoxHT	Hit the spin up box.
43	kShDownSpinBoxHT	Hit the spin down box.
44	kShSliderSpinBoxHT	Hit the spin slider box.
45	kShTabViewHT	(backward compatibility)
45	kShTabViewHT	Hit a tab.
46	kShTableCellHit	Hit a cell.
200	kShPlatformEvent	Warn for a platform event.

2.3 IExternalView Interface

Family ID : 'view'

Interface ID : IID_IExternalView

Interface file : IExView.h

This interface allows you to open external views (windows) in the workspace of Ray Dream.

You can not create an external view alone. An external view must be associated to a **I3DExModule**. This way, the view is managed by the shell, and you will only have to worry about what happens between the user and the module you created.

To create a view :

- In your 'View' resource editor (for example : AdLib), create a standard 'TView' element, and set its Class Name to 'TExternalView'.

- In the "User Value" of this view, put the Class ID of your External View (the one that you put in your 'COMP' resource). This is how the Shell will find out which component to instantiate.

- Usually, External Views are used in the user interface of some other component. It is most likely that this component will want to communicate with this view, for example to set and get data. The trick is simply to add an entry in the component's 'pMAP' resource. This entry should contain the ID of the External View and be of 'long' type. The Shell will then store an IUnknown pointer of the View in the Data Buffer of the component. Just cast the long value to IUnknown*, and then do a QueryInterface() to get a pointer to your External View's specific Interface.

API of COM External Views, based on the 'View' resource format.

```
DECLARE_INTERFACE_(IExternalView, I3DExtension) {
    // IUnknown methods
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
    STDMETHOD_(ULONG, AddRef) (THIS) PURE;
    STDMETHOD_(ULONG, Release) (THIS) PURE;

    // I3DExtension methods
    STDMETHOD(ShellUtilitiesInit) (THIS_ IShUtilities* shellUtilities) PURE;
    STDMETHOD_(I3DExtension*, Clone) (THIS) PURE;

    // IExternalView methods
    STDMETHOD(Draw) (THIS_ IShGraphicDevice* gd, LRECT3D* zone) PURE;
    STDMETHOD(PrepareMenus) (THIS) PURE;
    STDMETHOD_(BOOLEAN, OnMenuAction) (THIS_ IShContext* context, IShView* shView, IShGraphicDevice* graphicDevice, long actionNumber) PURE;
    STDMETHOD_(BOOLEAN, OnMouseAction) (THIS_ IShContext* context, IShView* shView, IShGraphicDevice* graphicDevice, LPOINT3D* mousePos, PLATFORMEVENT* event, POINT3D* threshold) PURE;
    STDMETHOD_(BOOLEAN, OnKeyEvent) (THIS_ IShContext* context, IShView* shView, IShGraphicDevice* graphicDevice, PLATFORMEVENT* event) PURE;
    STDMETHOD_(BOOLEAN, OnCommandKey) (THIS_ IShContext* context, IShView* shView, IShGraphicDevice* graphicDevice, PLATFORMEVENT* event) PURE;
    STDMETHOD(SetBounds) (THIS_ LRECT3D* newBounds) PURE;
    STDMETHOD(CursorEnteringWindow) (THIS_ BOOLEAN entering, IShGraphicDevice* graphicDevice) PURE;
    STDMETHOD(SetCursor) (THIS_ LPOINT3D* mousePos, IShGraphicDevice* graphicDevice) PURE;
}
```

```

    STDMETHOD_(short, GetCursorID) (THIS_ LPOINT3D* mousePos, IShGraphicDevice*
        graphicDevice) PURE;
    STDMETHOD_(BOOLEAN, WantsToBeFirstResponder) (THIS) PURE;
    STDMETHOD_(OnMessage) (THIS_ long messageID) PURE;
    STDMETHOD_(OnIdle) (THIS_ long idleStage) PURE;
    STDMETHOD_(SuperviewChangedBounds) (THIS_ IShView* shView, LRECT3D* oldBounds,
        LRECT3D* newBounds, BOOLEAN inval) PURE;
    STDMETHOD_(SubviewChangedBounds) (THIS_ IShView* shView, IShView* theSubview,
        LRECT3D* oldBounds, LRECT3D* newBounds, BOOLEAN inval) PURE;
    STDMETHOD_(UpdateCoordinates) (THIS_ IShView* shView, IShGraphicDevice* graph-
        icDevice) PURE;
    // Drag and drop support
    STDMETHOD_(BOOLEAN, GetDropArea) (THIS_ LPOINT3D* mousePos, long offeredType)
        PURE;
    STDMETHOD_(ReceiveDrop) (THIS_ IShContext* context, IShView *shView, IShGraph-
        icDevice* gd, long offeredType, void* data, BOOLEAN moveOrCopy, LPOINT3D*
        mousePos) PURE;
};

```

IExternalView::Draw

HRESULT IExternalView::Draw(*gd*, *zone*)

IShGraphicDevice* *gd*

LRECT3D* *zone*

IExternalView::Draw is called when the view or part of it needs to be redrawn.

Parameters

 *gd*

This is an interface for the graphic device to be used for drawing.

 *zone*

This is the rectangle that needs to be redrawn.

Return Value **S_OK**

IExternalView::PrepareMenus

HRESULT IExternalView::PrepareMenus()

Each time the menu needs to be updated, the shell calls this routine. Beware : this routine is called quite often : each time the data change, each time the user changes view, and more. The best way to code this is to store the information needed to set properly the menus items valid or not in an internal cache.

Return Value **S_OK**

IExternalView::OnMenuAction

BOOLEAN IExternalView::OnMenuAction(*context*, *shView*, *graphicDevice*, *actionNumber*)

IShContext* *context*

IShView* *shView*

IShGraphicDevice* *graphicDevice*

long *actionNumber*

This routine is called when the user selects an item in a menu, or presses the shortcut's keys corresponding

to one of the menu items. Your view should then treat the action by posting an **IShAction** when the user chose an undoable item.

Parameters

→ *context*

This is an interface for the context of the view. Just use that argument with **IShActionManager**, when posting and action for example.

→ *shView*

This is the shell's data for the view. It is generally just passed to **IShActionManager::PostMouseAction**, so you will probably not use it here.

→ *graphicDevice*

This is the graphic device of the external view. Use it to draw in the external view (See **IShGraphicDevice**).

→ *actionNumber*

This is the number of the menu item.

Return Value TRUE

IExternalView::OnMouseAction

BOOLEAN IExternalView::OnMouseAction(*context, shView, graphicDevice, mousePos, event, treshold*)

IShContext* *context*

IShView* *shView*

IShGraphicDevice* *graphicDevice*

LPOINT3D* *mousePos*

PLATFORMEVENT* *event*

POINT3D* *treshold*

Parameters

→ *context*

This is an interface for the context of the view. Just use that argument with **IShActionManager**, when posting and action for example.

→ *shView*

This is the shell's control for the view. It is generally just passed to **IShActionManager::PostMouseAction**, so you will probably not use it here.

→ *graphicDevice*

This is the graphic device of the external view. Use it to draw in the external view (See **IShGraphicDevice**).

→ *mousePos*

This is the position of the mouse in the view.

 *event*

This is the details of the event.

 *threshhold*

This is the precision of the mouse click.

Return Value TRUE if the action has been handled.
FALSE if the action has not been recognized. Nothing was done.

IExternalView::OnKeyEvent

BOOLEAN IExternalView::OnKeyEvent(*context, shView, graphicDevice, event*)

IShContext* *context*

IShView* *shView*

IShGraphicDevice* *graphicDevice*

PLATFORMEVENT* *event*

This routine is called when the user presses keys that are NOT shortcuts. You view should then treat the action by posting an action. Beware : if the user presses the Cmd/Ctrl key, the shell will call **IExternalView::OnCommandKey** instead of **IExternalView::OnKeyEvent**.

Parameters

 *context*

This is an interface for the context of the view. Just use that argument with **IShActionManager**, when posting and action for example.

 *shView*

This is the shell's data for the view. It is generally just passed to **IShActionManager::PostMouseAction**, so you will probably not use it here.

 *graphicDevice*

This is the graphic device of the external view. Use it to draw in the external view (See **IShGraphicDevice**).

 *event*

This is the details of the event.

Return Value TRUE if the action has been handled.
FALSE if the action has not been recognized by the module. Nothing was done.

IExternalView::OnCommandKey

BOOLEAN IExternalView::OnCommandKey(*context, shView, graphicDevice, event*)

IShContext* *context*

IShView* *shView*

IShGraphicDevice* *graphicDevice*

PLATFORMEVENT* *event*

This routine is called when the user presses Cmd/Ctrl key and another key. That is exactly like

IExternalView::OnKeyEvent, but the Cmd/Ctrl key is down.

Parameters

→ *context*

This is an interface for the context of the view. Just use that argument with **IShActionManager**, when posting and action for example.

→ *shView*

This is the shell's data for the view. It is generally just passed to **IShActionManager::PostMouseAction**, so you will probably not use it here.

→ *graphicDevice*

This is the graphic device of the external view. Use it to draw in the external view (See **IShGraphicDevice**).

→ *event*

This is the details of the event.

Return Value TRUE if the action has been handled.
FALSE if the action has not been recognized by the module. Nothing was done.

IExternalView::SetBounds

HRESULT IExternalView::SetBounds(*newBounds*)

LRECT3D* *newBounds*

This routine is called each time the size of the view changes.

Parameters

→ *newBounds*

This is the new size of the view.

Return Value S_OK

IExternalView::CursorEnteringWindow

HRESULT IExternalView::CursorEnteringWindow(*entering*, *graphicDevice*)

BOOLEAN *entering*

IShGraphicDevice* *graphicDevice*

This routine is called when the mouse enters the external view. This is very usefull for custom mouse pointers that need constant redraw.

Parameters

→ *entering*

TRUE if entering, FALSE is exiting.

→ *graphicDevice*

This is the graphic device of the external view. Use it to draw in the external view (See **IShGraphicDevice**).

Return Value S_OK

IExternalView::SetCursor

HRESULT IExternalView::SetCursor(mousePos, graphicDevice)

LPOINT3D* mousePos

IShGraphicDevice* graphicDevice

This routine allows you to set the cursor. If you just have a 'curs' resource as a cursor, you do not need to implement this routine, and can use **IExternalView::GetCursorID** instead. **IExternalView::SetCursor** is intended for more complex cursor management..

Parameters

 mousePos

This is the position of the mouse in the view.

 graphicDevice

This is the graphic device of the external view. Use it to draw in the external view (See **IShGraphicDevice**).

Return Value

S_OK

E_NOTIMPL Tell the Shell to use **IExternalView::GetCursorID**() instead.

Comments

See also **IExternalView::GetCursorID** for more simple cursor management.

IExternalView::GetCursorID

short IExternalView::GetCursorID(mousePos, graphicDevice)

LPOINT3D* mousePos

IShGraphicDevice* graphicDevice

This routine allows you to set the cursor by using a 'curs' resource (a 'curs' resource is a common Macintosh resource type) that you provide in your .dta file.

Parameters

 mousePos

This is the position of the mouse in the view.

 graphicDevice

This is the graphic device of the external view. You probably do not need it at this point, but it is still provided for your convenience.

Return Value

The ID of your 'curs' resource, or one of the preset following values:

kShNoCursor	-1	Hide the cursor
kShArrowCursor	1	Standard arrow cursor
kShWaitCursor	2	Wait cursor
kShIBeamCursor	3	Text insertion cursor
kShColumnSizingCursor	4	Column resizing cursor

IExternalView::WantsToBeFirstResponder

BOOLEAN IExternalView::WantsToBeFirstResponder()

Return TRUE if the view should be the first one to get events in the window.

Return Value TRUE or FALSE

Comments You usually return TRUE, since you probably want to get events. Whatever you return, you will always get dedicated events such as mouse clicks. But if you want to catch menu and key events, and be able to set up menus in **PrepareMenus**, then you should return TRUE. It is also advised to set the *Wants to be Target* check box in AdLib for your view (Target = First Responder).

IExternalView::OnMessage

HRESULT IExternalView::OnMessage(messageID)

long messageID

This is a general purpose event routine, reserved for future use. At this time, you view will be told about some events, but usually you do not need to process them here. Specific events are treated in specific routines, such as **IExternalView::OnMouseAction**, **IExternalView::OnMenuAction**, etc.

Parameters  messageID
The identifier code of the message.

Return Value S_OK

IExternalView::OnIdle

HRESULT IExternalView::OnIdle(idleStage)

long idleStage

IExternalView::OnIdle allows you to get control while nothing is happening (idle time). You can use this opportunity to update your view, or perform any background process. A clock view would be a typical example.

Parameters  idleStage
The phase of the idle stage. There are 3 stages: **kStartIdle**, **kContinueIdle** and **kFinishIdle**. **kStartIdle** indicates that the application is starting to idel because no event is pending. **kContinueIdle** will be passed repeatedly according to the Idle Frequency (see below), until a new event shows up, in which case **kFinishIdle** will be passed.

kStartIdle	0	Beginning of idle phase
kContinueIdle	1	Idle phase continuing
kFinishIdle	2	End of Idle Phase

Return Value S_OK

Notes Make sure the Idle check for is on in your view resource in AdLib. Then set the **Idle Frequency** (in ticks, i.e. 1/60 seconds) in AdLib. The Idle Frequency is a time period. The CPU will *try* to call you after each of this time period. If you set a value of 0, this means that you want to be called as often as possible (but this is not a very cooperative approach). If you set a value of 30, then it means that you would like to be called

every 1/2 second. Of course, there is no garanty that you will be called in 1/2 second, because the CPU may be very busy on something else. But if nothing significant happens, then you will be called every 30 ticks.

IExternalView::SuperViewChangedBounds

HRESULT IExternalView::SuperViewChangedBounds(*shView*, *oldBounds*, *newBounds*, *inval*)

IShView* *shView*

LRECT3D* *oldBounds*

LRECT3D* *newBounds*

BOOLEAN *inval*

IExternalView::SuperViewChangedBounds will be called every time the “Super View” of the view have changed in size or position. The Super View is the view containing this view in the hierachy of views that you build with AdLib. For example it can be the Window.

Parameters

 *shView*

This is the Shell’s data for our own view. It can be passed to the **IShViewManager** calls, if necessary.

 *oldBounds*

This is the old frame of the super view.

 *newBounds*

This is the new frame of the super view.

 *inval*

Tells us if we should invalidate whatever change we might have to do, in order to trigger a future redraw event. For example, if the super view is a window that is being enlarged, *inval* will be TRUE because the revealed part will have to be updated.

Return Value S_OK

Comments See also **IExternalView::SubViewChangedBounds**

IExternalView::SubViewChangedBounds

HRESULT IExternalView::SubViewChangedBounds(*shView*, *theSubView*, *oldBounds*, *newBounds*, *inval*)

IShView* *shView*

IShView* *theSubView*

LRECT3D* *oldBounds*

LRECT3D* *newBounds*

BOOLEAN *inval*

This routine is similar to **IExternalView::SuperViewChangedBounds**, but it applies to our Sub-Views (views for which we are a Super View).

Parameters

 *shView*

This is the Shell’s data for our own view. It can be passed to the **IShViewManager** calls, if necessary.

 *theSubView*

The Shell’s data of the sub-view. It can be passed to the **IShViewManager** calls, if you need to perform

manipulations on the sub-view.

→ *oldBounds*

This is the old frame of the sub view.

→ *newBounds*

This is the new frame of the sub view.

→ *inval*

Tells us if we should invalidate whatever change we might have to do, in order to trigger a future redraw event.

Return Value S_OK

Comments See also **IExternalView::SuperViewChangedBounds**

IExternalView::UpdateCoordinates

HRESULT IExternalView::UpdateCoordinates(*shView*, *graphicDevice*)

IShView* *shView*

IShGraphicDevice* *graphicDevice*

IExternalView::UpdateCoordinates is called to tell us that our global coordinate system has changed. This is a low-level call that you will probably never used. Global coordinate system can change if the view has been moved or resized, or if the visible part of the view has changed. In the first case, it is better and more simple to rely on **IExternalView::SetBounds**. If you are interested in the second case (visible part has changed), the only way you can catch it it by implementing this routine.

Parameters → *shView*

This is the Shell's data for our own view. It can be passed to the **IShViewManager** calls, if necessary.

→ *graphicDevice*

This is the graphic device of the external view. You probably do not need it at this point, but it is still provided for your convenience.

Return Value S_OK

E_NOTIMPL

IExternalView::GetDropArea

BOOLEAN IExternalView::GetDropArea(*mousePos*, *offeredType*)

LPOINT3D* *mousePos*

long *offeredType*

Drag and drop support. Answer TRUE if you accept 'offeredType' at the 'mousePos' location. If you return TRUE, the view will be highlighted so that the user understands it is a drop candidate.

Parameters → *mousePos*

The mouse pointer location in your view. Use it if you view has only specific sensitive areas.

➦ *offeredType*

This is a code identifying the type of data offered by the Shell. Some items that the user drags have multiple data types attached to them. Therefore **IExternalView::GetDropArea** may be called several times for the same item. This is normal.

At this time, most of the data types are private. However, there is one which is public: the tools from the toolbars:

```
#define kShModuleToolDropType IDTYPE('m', 't', 'd', 'r')
```

Return Value TRUE if the offered data is accepted.

Comments See **IExternalView::ReceiveDrop**.

IExternalView::ReceiveDrop

HRESULT IExternalView::ReceiveDrop(context, shView, gd, offeredType, data, moveOrCopy, mousePos)

IShContext* context

IShView* shView

IShGraphicDevice* gd

long offeredType

void* data

BOOLEAN moveOrCopy

LPOINT3D* mousePos

Drag and drop support. **IExternalView::ReceiveDrop** will be called if **IExternalView::GetDropArea** accepted the dragged item.

Parameters ➦ *context*

The current context the view belongs to. You will need it if you have to post an action using the **IShActionManager** interface.

➦ *shView*

This is the Shell's data for our own view. It can be passed to the **IShViewManager** and **IShActionManager** calls, if necessary.

➦ *graphicDevice*

This is the graphic device of the external view. You probably do not need it at this point, but it is still provided for your convenience.

➦ *offeredType*

This is a code identifying the type of data offered by the Shell. Same parameter as in **IExternalView::GetDropArea**.

➦ *data*

A pointer on the private offered data, or a long value. The content is completely dependent of the type of data offered. For tools drag and drop (*offeredType* == kShModuleToolDropType), cast *data* to a **long** to get the tool ID:

```
long toolID = (long) data;
```

 *moveOrCopy*

The Shell will tell you if the current item is being moved or copied.

 *mousePos*

The mouse pointer location in your view.

Return Value S_OK

Comments See **IExternalView::GetDropArea**.

2.4 IExDataExchange Interface

Interface ID: IID_IExDataExchange

Interface file: IExtension.h

Description

The **IExDataExchange** Interface contains utility methods that are complements to IMCUnknown, like clone(). It allows the Shell to handle the UI of the Component and the Component's public data. It also handles the communication between the Component and the Shell.

Methods Index

“IExDataExchange::Prepare(IExDataExchange**, IMCUnknown*)”
 “IExDataExchange::IsEqual(IExDataExchange*)”
 “IExDataExchange::GetParameter(IDType, void*)”
 “IExDataExchange::SetParameter(IDType, void*)”
 “IExDataExchange::GetExtensionDataBuffer()”
 “IExDataExchange::ExtensionDataChanged()”
 “IExDataExchange::HandleEvent(MessageID, IMFResponder*, void*)”
 “IExDataExchange::GetResID()”

IExDataExchange::Prepare(IExDataExchange**, IMCUnknown*)

void MCOMAPI Clone(IExDataExchange,IMCUnknown* pUnkOuter)**

Clone() clones a data exchanger. <zot needs description.>

Parameter  **IExDataExchange****
 A pointer that specifies the source data exchanger.

 **IMCUnknown* pUnkOuter**
 A pointer to the component.

IExDataExchange::IsEqual(IExDataExchange*)

boolean MCOMAPI IsEqual(IExDataExchange * otherDataExchange)

IsEqual() returns True if the two pointers are referring to the same data exchanger.

Parameter  **IExDataExchange*otherDataExchange**
 A pointer to the data exchanger.

IExDataExchange::GetParameter(IDType, void*)

MCCOMErr MCOMAPI GetParameter(IDType keyword, void *parameter)

GetParameter() <zot needs description> .

Parameter

- **IDType** *keyword*
<zot needs description> .
- **void*** *parameter*
A pointer that specifies the parameter.

IExDataExchanger::SetParameter(IDType, void*)

MCCOMErr MCCOMAPI SetParameter(IDType keyword, void *parameter)
SetParameter() <zot needs description> .

Parameter

- **IDType** *keyword*
<zot needs description> .
- **void*** *parameter*
A pointer that specifies the parameter.

IExDataExchanger::GetExtensionDataBuffer()

void* MCCOMAPI GetExtensionDataBuffer()
GetExtensionDataBuffer() returns a pointer to the extension data buffer.

IExDataExchanger::ExtensionDataChanged()

MCCOMErr MCCOMAPI ExtensionDataChanged()
ExtensionDataChanged() <zot needs description>.

IExDataExchanger::HandleEvent(MessageID, IMFResponder*, void*)

MCCOMErr MCCOMAPI HandleEvent(MessageID message, IMFResponder* source, void* data)
HandleEvent() handles a message event.

Parameter

- **MessageID** *message*
All MessageIDs are preceeded with "kMsg".
- **IMFResponder*** *source*
A pointer that specifies the source of the event.
- **void*** *data*
A pointer that specifies the data.

IExDataExchanger::GetResID()

int16 MCCOMAPI GetResID()
GetResID() gets the resource ID.

2.5 I3DExExportFilter Interface

Interface ID: IID_I3DExExportFilter

Interface file: I3DExImportExport.h

Description

The **I3DExExportFilter** Interface filters Carrara data for exporting, and allows you to set up preview information that gets stored with the file.

Methods Index

“I3DExExportFilter::Prepare(I3DShScene*, I3DShElement*)”
 “I3DExExportFilter::WantsOptionDialog()”
 “I3DExExportFilter::NeededDiskSpace(IMCFile*, I3DShScene*, I3DShTreeElement*, uint32*)”
 “I3DExExportFilter::DoExport(IMCFile*, I3DShScene*, I3DShElement*)”
 “I3DExExportFilter::WantsPreview()”
 “I3DExExportFilter::GetPreviewSize(int16*, int16*)”
 “I3DExExportFilter::SetPreview(TMCPixelBucket*)”

I3DExExportFilter::Prepare(I3DShScene*, I3DShElement*)

boolean MCCOMAPI Prepare(I3DShScene* scene, I3DShTreeElement* fatherTree)

Prepare() prepares to export the specified tree element in the specified scene . Returns True when <zot needs more description>

Parameter

→ **I3DShScene* scene**

A pointer that specifies which scene to prepare to export.

→ **I3DShTreeElement* fatherTree**

A pointer to the top or father of the tree element for the item being prepared for export.

I3DExExportFilter::WantsOptionDialog()

boolean MCCOMAPI WantsOptionDialog()

WantsOptionDialog() returns True if the Options dialog box is needed to specify options before the scene can be saved.

I3DExExportFilter::NeededDiskSpace(IMCFile*, I3DShScene*, I3DShTreeElement*, uint32*)

MCCOMErr MCCOMAPI NeededDiskSpace(IMCFile * file, I3DShScene* scene, I3DShTreeElement* fatherTree, uint32* diskSpace)

NeededDiskSpace() determines the amount of disk spaced needed to store a file containing the specified scene and specified tree element .

Parameter	 IMCFile* <i>file</i> A pointer that specifies the file name.
	 I3DShScene* <i>scene</i> A pointer that specifies which scene to prepare to export.
	 I3DShTreeElement* <i>fatherTree</i> A pointer to the top or father of the tree element for the item being prepared for export.
	 uint32* <i>diskSpace</i> A pointer to the amount of disk space needed to store the file.

I3DExExportFilter::DoExport(IMCFile*,I3DShScene*, I3DShElement*)

MCCOMErr MCCOMAPI DoExport(IMCFile * file, I3DShScene* scene, I3DShTreeElement* fatherTree)

DoExport() performs the export to the specified file of the specified scene and tree element.

Parameter	 IMCFile* <i>file</i> A pointer that specifies the file name.
	 I3DShScene* <i>scene</i> A pointer that specifies which scene to prepare to export.
	 I3DShTreeElement* <i>fatherTree</i> A pointer to the top or father of the tree element for the item being prepared for export.

I3DExExportFilter::WantsPreview()

boolean MCCOMAPI WantsPreview()

WantsPreview() returns True when preview information should be saved with the file.

I3DExExportFilter::GetPreviewSize(int16*, int16*)

MCCOMErr MCCOMAPI GetPreviewSize(int16* sizeh, int16* sizev)

GetPreviewSize() calculates the preview size in pixels .

Parameter	 int16* <i>sizeh</i> A pointer that specifies the horizontal preview dimension in pixels.
	 int16* <i>sizev</i> A pointer that specifies the vertical preview dimension in pixels.

I3DExExportFilter::SetPreview(TMCPixelBucket*)

MCCOMErr MCCOMAPI SetPreview(TMCPixelBucket* preview)

SetPreview() displays the specified preview image .

Parameter  **TMCPixelBucket*** *preview*
A pointer that specifies which preview to display.

2.6 I3DExImportFilter Interface

Interface ID: IID_I3DExImportFilter

Interface file: I3DExImportExport.h

Description

The **I3DExImportFilter** Interface filters data for importing into Carrara.

Methods Index

“I3DExImportFilter::Prepare(IMCFile*, I3DShScene*, I3DShElement*)”
 “I3DExImportFilter::WantsOptionDialog()”
 “I3DExImportFilter::DoImport(IMCFile*,I3DShScene*, I3DShElement*)”
 “I3DExImportFilter::WantsTopScene()”

I3DExImportFilter::Prepare(IMCFile*, I3DShScene*, I3DShElement*)

boolean MCCOMAPI Prepare(IMCFile * file, I3DShScene* scene, I3DShTreeElement* fatherTree)

Prepare() prepares to export the specified tree element in the specified scene . Returns True when <zot needs more description>

Parameter

➡ **IMCFile* file**

A pointer that specifies the file to import.

➡ **I3DShScene* scene**

A pointer that specifies which scene into which to import.

➡ **I3DShTreeElement* fatherTree**

A pointer to the top or father of the tree element for the item being prepared for import.

I3DExImportFilter::WantsOptionDialog()

boolean MCCOMAPI WantsOptionDialog()

WantsOptionDialog() returns True if the Options dialog box is needed to specify options before the scene can be imported.

I3DExImportFilter::DoImport(IMCFile*,I3DShScene*, I3DShElement*)

MCCOMErr MCCOMAPI DoImport(IMCFile * file, I3DShScene* scene, I3DShTreeElement* fatherTree)

DoImport() performs the import of the specified file to the specified scene and tree element.

Parameter

➡ **IMCFile* file**

A pointer that specifies the file name.

➡ **I3DShScene* scene**

A pointer that specifies which scene to prepare to export.

→ **I3DShTreeElement*** *fatherTree*

A pointer to the top or father of the tree element for the item being prepared for export.

I3DExImportFilter::WantsTopScene()

boolean MCCOMAPI **WantsTopScene()**

WantsTopScene() returns True when <zot need more info>.

2.7 IExRegisterer Interface

Interface ID: IID_IExRegisterer

Interface file: IExIRegisterer.h

Description

The **IExRegisterer** Interface can register multiple components, typically when the existence of components is conditioned by other installed software.

Methods Index

“IExRegisterer::GetComponentCount()”

“IExRegisterer::GetComponentData(uint32, APICOMPData&, APIPrivateData&, APIGUIDData&)”

“IExRegisterer::UnregisterComponents()”

IExRegisterer::GetComponentCount()

uint32 MCOMAPI GetComponentCount()

GetComponentCount() returns the number of components to be registered. <zot needs more description>

IExRegisterer::GetComponentData(uint32, APICOMPData&, APIPrivateData&, APIGUIDData&)

MCCOMErr MCOMAPI GetComponentData(uint32 index, APICOMPData& compData, APIPrivateData& privateData, APIGUIDData& guidData)

GetComponentData() gets all of the information needed to register a component. This includes the component data, any private data needed by that component, and the guid data.

Parameter

→ **uint32** *index*

<zot needs description>.

→ **APICOMPData&** *compData*

A reference to the component data.

→ **APIPrivateData&** *privateData*

A reference to any private data needed by that type of component.

→ **APIGUIDData&** *guidData*

A reference to the guid data for the component.

IExRegisterer::UnregisterComponents()

MCCOMErr MCOMAPI UnregisterComponents()

UnregisterComponents() should be called when a component is uninstalled <zot need more info>.

Chapter 3 - Data Structures Reference

Basic Data Structures

These are common data structures found throughout the SDK.

TVECTOR2

TVECTOR2 is used to manipulate two-dimensional data: 2D points or 2D vectors.

Constructors

```
TVector2::TVECTOR2(void)
TVector2::TVECTOR2(real, real)
```

Constants

Name	X value	Y value
TVector2::kZero	0.0	0.0
TVector2::kOnes	1.0	1.0
TVector2::kUnitX	1.0	0.0
TVector2::kUnitY	0.0	1.0
TVector2::kNegativeX	-1.0	0.0
TVector2::kNegativeY	0.0	-1.0

Element Access Operators

```
TVector2::TVector2()
TVector2(0) returns the X value and TVector2(1) returns the Y value.
```

```
TVector2::TVector2[]
TVector2[0] returns the X value and TVector2[1] returns the Y value.
```

```
TVector2::TVector2.X
TVector2::TVector2.Y
```

Methods

```
TVector2::SetValues(real, real)
Assigns the X and Y values.
```

```
TVector2::GetDimension(void)
Returns the dimensions of a TVector2.
```

`TVector2::GetMagnitude()`

Returns $\sqrt{x^2 + y^2}$

`TVector2::GetMagnitudeSquared()`

Returns the square of the magnitude:

$$x^2 + y^2$$

`TVector2::GetDistance(const TVector2& )`

Returns the distance of vector.

`TVector2::SetMagnitude(const real)`

Sets the magnitude of the vector equal to the real given in the parameter.

`TVector2::SetToCenter(const TVector2&, const TVector2&)`

Sets the point at the center of the two points given in parameters.

`TVector2::Normalize(const real)`

Determines if the magnitude of the vector is greater than the real given in the parameter. If so, it normalizes the vector by setting the magnitude equal to 1. Returns True if it normalized the vector.

`TVector2::Normalize(TVector2& )`

Normalizes the vector if it is not (0,0). Returns True if it normalized the vector.

`TVector2::NormalizeOrReplaceWith(const TVector2& ,const real)`

If the magnitude of the vector is greater than the real, then the vector is normalized. Otherwise, it is replaced by the vector given in the parameter.

`IsEqual(const TVector2& , const real)`

Returns True if the vector and the vector given in the paramter are the same or within the tolerance specified by the real.

`ClampValue(const TVector2& , const TVector2& )`

Bounds the point within the range fixed by the two points given in the parameters. The first is the minimum value, the second represents the upper limit.

`Negate(void)`

Returns the negation of "this".

`Interpolate(const TVector2& , const TVector2& , const float )`

Interpolates the point "this" between the two points given in parameters; if hte float is equal to 0, "this" will be equal to the first paramter. If the float is equal to 1, "this will be equal to the second parameter.

`Swap(TVector2& )`

Swaps "this" and the vector given in the paramter.

zot There was more in Vector2.h not sure how much to document.

Operators

These are used for fast vector operations.

```
inline TVector2&operator/=(const real scalar);
inline TVector2&operator*=(const real scalar);
inline TVector2&operator+=(const TVector2& v);
inline TVector2&operator-=(const TVector2& v);
```

TVECTOR3

TVECTOR3 is similar to **TVECTOR2** but for three-dimensional data: 3D points or 3D vectors.

TBBOX2D

TBBOX2D is used for 2D Bounding Boxes. A 2D Bounding Box is a rectangular area that encloses a set of 2D points.

Constructors

```
TBBox2D();  
TBBox2D(const TVector2& min, const TVector2& max);  
TBBox2D(const TVector2& max);
```

Methods

```
void SetMax(const TVector2& max) {fMax = max;}  
Sets the upper right corner of the bounding box.
```

```
void SetMin(const TVector2& min) {fMin = min;}  
Sets the bottom left corner of the bounding box.
```

```
void GetCenter(TVector2& center) const  
Returns the coordinates for the center of the bounding box.
```

```
void GetDelta(TVector2& delta) const  
Gets the difference between the minimum and maximum values.
```

```
real GetMax() const  
Returns either the width of the bounding box if it is greater than the length, or the length if it is greater than or equal to the width.
```

```
boolean IsEmpty() const  
Returns True if the bounding box is not set to a value.
```

```
void AddPoint(const TVector2 &PP)  
Resets the bounding box if the point PP wasn't within. PP becomes either the upper right or bottom left corner of the new bounding box.
```

```
void Inset(const TVector2 &PP)  
Shrinks the bounding box of the vector given in the parameter.
```

```
TVector2 operator[](int16 ii) const  
Transforms the bounding box in an array. The first element is the bottom-left corner, the second is the upper-left corner, the third is the upper-right corner, and the fourth is the bottom-right corner.
```

```
void FixInvertedFields()  
If the bottom-left corner of the bounding box is located further to the right than the upper-right corner, then both corner x-coordinates are set to 0. If the bottom-left corner is greater than the upper-left, then both of their Y coordinates are set to 0.
```

```
inline real GetWidth() const  
{ return fMax.x - fMin.x; }  
Returns the width of the bounding box.
```

```
inline real GetHeight( void )
{ return fMax.y - fMin.y; }
```

Returns the height of the bounding box.

Constants

TVector2 fMin

Bottom-left corner of the bounding box.

TVector2 fMax

Upper-right corner of the bounding box.

TBBOX3D

TBBOX3D is similar to TBBBox 2D except that it is used for 3D Bounding Boxes. A 3D Bounding Box is a cubic volume that encloses a set of 3D points.

TMATRIX33

TMATRIX33 is used for 3D rotation.

If you are porting code from RDS, be careful about converting code from TMatrix3D to TMatrix33 since the void ctor for TMatrix33 does not set the matrix to identity. Use explicit initialization to the constant TMatrix33::kIdentity if necessary.

not the section that follows is from the old sdk. How much of this is stil relevant?

A convenient way to look at such a 3 x 3 Matrix is to consider it as three column vectors I, J and K. In this perspective, here is how one could write the transformation of an (x, y, z) point in the mathematical form:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} ix & jx & kx \\ iy & jy & ky \\ iz & jz & kz \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The reason of its convenience is that these vectors represent the unit vectors of the Object Local System defined by this matrix, and that the Ix, Iy, etc. values are the coordinates of these vectors in the Global Coordinates System.

Comments To define a 3D rotation, such a matrix must be normalized.

All row and column vectors of the matrix must be unit vectors:

$$ix^2 + iy^2 + iz^2 = 1.0$$

$$jx^2 + jy^2 + jz^2 = 1.0$$

$$kx^2 + ky^2 + kz^2 = 1.0$$

$$ix^2 + jx^2 + kx^2 = 1.0$$

$$iy^2 + jy^2 + ky^2 = 1.0$$

$$iz^2 + jz^2 + kz^2 = 1.0$$

All row and column vectors of the matrix must be orthogonal to each other:

$$ix*jx + iy*jy + iz*jz = 0.0$$

$$ix*kx + iy*ky + iz*kz = 0.0$$

$$jx*kx + jy*ky + jz*kz = 0.0$$

$$ix*iy + jx*jy + kx*ky = 0.0$$

$$ix*iz + jx*jz + kx*kz = 0.0$$

$$iy*iz + jy*jz + ky*kz = 0.0$$

If the matrix is normalized, then inverting it to get the inverse transformation is easy: the inverse matrix is obtained by simple transposition:

$$\text{Inverse Matrix} = \begin{matrix} & \begin{matrix} ix & iy & iz \end{matrix} \\ \begin{matrix} ix & jx & kx \end{matrix} & \begin{matrix} jy & jz & ky & kz \end{matrix} \end{matrix}$$

Example

Here is a rotation around the Z axis of an alpha angle:

$$\begin{matrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{matrix}$$

And the inverse rotation is written like this after transposition:

$$\begin{matrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{matrix}$$

Which does correspond to a -alpha rotation because $\cos(\alpha) = \cos(-\alpha)$ and $-\sin(\alpha) = \sin(-\alpha)$:

$$\begin{matrix} \cos(-\alpha) & \sin(-\alpha) & 0 \\ \sin(-\alpha) & \cos(-\alpha) & 0 \\ 0 & 0 & 1 \end{matrix}$$

zot end old material

Constants

kIdentity is the matrix identity.

kZero is Matrix 0.

Constructors

```
inline TMatrix33 (void);
inline TMatrix33 (const TMatrix33& original);
inline TMatrix33 (const real m00, const real m01, const real m02, const real m10,
    const real m11, const real m12, const real m20, const real m21, const real
    m22);
```

Methods

```
inline uint32 GetRowDimension (void)
```

Gets the number of rows.

```
inline uint32 GetColumnDimension (void)
```

Gets the number of columns.

```
inline real operator() (const uint32 row, const uint32 col)
```

Use this for element access, since there is no [] [] operator.

```
inline real& operator() (const uint32 row, const uint32 col)
```

Use this for element access, since there is no [] [] operator.

```
inline const real* operator[] (const uint32 row)
```

Use this method for [][] access only; the second index is unchecked.

```
inline real* operator[] (const uint32 row)
```

Use this method for [][] access only; the second index is unchecked.

```
void GetRow (const uint32 row, TVector3& v)
```

Gets a row for row access:

```
void SetRow (const uint32 row, const TVector3& v)
```

Sets a row of the matrix.

```
void GetColumn (const uint32 col, TVector3& v)
```

Gets a column for column access:

```
void SetColumn (const uint32 col, const TVector3&v)
```

Sets a column of the matrix.

```
void SetToTranspose (void)
```

Use this method for matrix conversion. Transposes the matrix "this".

```
void SetToTransposeOf (const TMatrix33& source)
```

Use this method for matrix conversion. Sets the matrix equal to the transpose of the matrix source.

```
TMatrix33 GetTranspose (void)
```

Returns the transpose of the matrix "this" without changing it.

```
TMatrix33 GetAdjoint ()
```

Returns the adjoint of the matrix "this".

```
TMatrix33 GetOrthogonal (boolean* success)
```

Returns the rotation associated with the matrix ; an orthogonal matrix returns itself.

```
real GetTrace (void)
Returns the trace of the matrix "this".
```

```
real GetDeterminant (void)
Returns the determinant of the matrix "this".
```

```
void Multiply (const TMatrix33& m, const TMatrix33& n)
Sets the matrix "this" equal to the multiplication of the parameters m and n.
```

The following methods are used for fast matrix operation:

```
void operator*= (const real scalar);
TMatrix33& operator= (const TMatrix33& original);
These routines were added to support existing code in RealMatrix33:
```

```
void SetColumns (const TVector3& c0, const TVector3& c1, const TVector3& c2);
void GetColumns (TVector3& c0, TVector3& c1, TVector3& c2) const;
void SetToDiagonalTfm (const real x, const real y, const real z);
boolean IsEqual (const TMatrix33& rhs, const real zeroTolerance = 0.0f) const;
These are the deprecated names from RealMatrix33, use the direct equivalents when writing new code:
```

```
inline void SetIJK (const TVector3& c0, const TVector3& c1, const TVector3& c2)
{ SetColumns( c0, c1, c2 ); }
```

```
inline void GetIJK (TVector3& c0, TVector3& c1, TVector3& c2) const
{ GetColumns( c0, c1, c2 ); }
```

```
inline void SetDiagonal (const real x, const real y, const real z)
{ SetToDiagonalTfm( x, y, z ); }
```

```
inline void Reset ()
{ *this = TMatrix33::kIdentity; }
```

```
inline boolean IsIdentity () const
{
    return ( this == &TMatrix33::kIdentity ) ||
           this->IsEqual( TMatrix33::kIdentity );
}
```

```
inline TMatrix33 Transpose () const
{ return GetTranspose(); }
```

Use `TMtxMath::SolveInverseGeneralCR` instead of this:

```
TMatrix33 Inverse(boolean* success) const;
```

TTRANSFORM3D

TTRANSFORM3D is used to define a combination of a Rotation and a Translation.

Constants

⇒ *fRotationAndScale*

Matrix composed of: UniformScaling, XYZScaling, and Rotation around local 0,0,0.

Constructor

```
TTransform3D()
```

Methods

```
TVector3 TransformPoint(const TVector3 &PP) const;
```

Transforms a three dimensional point.

```
TVector3 TransformVector(const TVector3 &VV) const;
```

Transforms a three dimensional vector.

Comments This kind of transformation can be written as this:

$$P' = [R] P + T$$

when P is the original 3D point and P' the same point after transformation.

TREETRANSFORM

TREETRANSFORM is used to define a Tree Element Transformation.

Constructors and Destructor

```
TTreeTransform();
TTreeTransform(const TTreeTransform &aT);
~TTreeTransform();
```

Methods

```
void      Replicate(const TTreeTransform& repTransform)
boolean   IsIdentity()
boolean   IsDirect()
void      Reset()
void      AlignOnGravity()
void      PointAt(const TVector3 &point)
void      Expand(real scale)
void      Invert()
TTreeTransformInverse()
void      Align()
void      Orthogonalize()
```

Operators

```
boolean operator==(const TTreeTransform& tt)
void operator= (const TTreeTransform& tt)
TTreeTransform operator*(const TTreeTransform& b)
TVector3 operator*(const TVector3& point)
```

Accessors

```
void SetOffset(const TVector3& offset){fOffset= offset;}
void SetRotation(const TMatrix33& rotation){fRotation= rotation;}
void SetUniformScaling(const real& scaling){fUniformScaling = scaling;}
void SetHotPoint(const TVector3& hotPoint){fHotPoint= hotPoint;}
void SetXYZScaling(const TVector3& XYZScaling){fXYZScaling= XYZScaling;}
void SetPhyThetaPsy(real phy, real theta, real psy, boolean isDirect);
const TVector3& GetOffset() const {return fOffset;}
```

```

const TMatrix33& GetRotation() const {return fRotation;}
const real&GetUniformScaling() const {return fUniformScaling;}
const TVector3& GetHotPoint() const {return fHotPoint;}
const TVector3& GetXYZScaling() const {return fXYZScaling;}
void    GetPhyThetaPsy(real& phy, real& theta, real& psy, boolean& isDirect) const;
void    GetTransform3D(TTransform3D &aT) const;
void    GetTransform3D(TMatrix33& RR, TVector3& TT) const;
void    GetCombinedTranslation(TVector3 &TT) const;
void    GetPlane(TPlane& aplane) const;

```

Translation

```

void    Translate(const TVector3& vector);
void    TranslateHP(const TVector3& vector);
void    TranslateHPL(const TVector3& vector);

```

Rotation

```

void    RotateAxis(const TVector3& center, const real sin, const real cos, const
TVector3& AXIS, const real& constrainAngle = kRealZero);
void    Rotate(const TVector3& center, const TVector3& PP, const TVector3& QQ,
const TVector3& AXIS, const real& constrainAngle = kRealZero);
void    RotateL(const TMatrix33& a);
void    RotateR(const TMatrix33& b);

```

Composition

```

void    ComposeL(const TTreeTransform& transform);
void    ComposeR(const TTreeTransform& transform);

```

Transformations

```

void    TransformCS (const TTreeTransform& a);
void    InverseTransformCS(const TTreeTransform& a);

TVector3& Transform (const TVector3& point, TVector3& result) const;
TVector3& InverseTransform(const TVector3& point, TVector3& result) const;

TVector3& TransformV (const TVector3& point, TVector3& result) const;
TVector3& InverseTransformV(const TVector3& point, TVector3& result) const;

TPlane&   ransformPlane(const TPlane& plane, TPlane& result) const;
TPlane&   InverseTransformPlane(const TPlane& plane, TPlane& result) const;

TBBBox3D  TransformBBBox(const TBBBox3D& box) const;
TBBBox3D  InverseTransformBBBox(const TBBBox3D& box) const;

TVector3  ScaleV (const TVector3&) const; // scaling only

```

Variables

```

TMatrix33fRotation;
Rotation matrix around HotPoint.

TVector3fOffset;
Vector from parent 0,0,0 to HotPoint.

real    fUniformScaling;
Uniform scaling. This scaling is applied in all directions.

TVector3fHotPoint;

```

Vector from local 0,0,0 to HotPoint.

TVector3fXYZScaling;
XYZ (non-uniform) scaling.

Comments This kind of transformation can be written as this:

$$P' = S * ([R] P) + T$$

when P is the original 3D point and P' the same point after transformation.

TMCCOLORRGB

This is the floating point color used for most calculations since it can do interpolation without converting int-to-float.

Variables

```
real kMaxIntensity = 1;
real kInvMaxIntensity = 1.0 / kMaxIntensity;

TMCColorRGB kBlack = (0,0,0);
TMCColorRGB kWhite = (kMaxIntensity,kMaxIntensity,kMaxIntensity);
```

Constructors

```
inline TMCColorRGB(void);
inline TMCColorRGB(const real rr,const real gg, const real bb);
inline TMCColorRGB(const TMCRGBColor inColor);
```

Methods

```
inline void Set (const real rr, const real gg, const real bb);
Sets the Red, Green, and Blue values.
```

```
boolean IsEqual (const TMCColorRGB& rhs, const real epsilon = 0.0) const;
Returns True if the color "this" is equal to rhs, or within the tolerance specified. (epsilon by default is set to 0.)
```

Magnitudes

```
inline real GetMagnitude(void);
Returns the magnitude.
```

```
inline real GetMagnitudeSquared(void);
Returns the magnitude squared.
```

```
inline real GetMinOfRGB(void) const;
Finds the minium intensity.
```

```
inline real GetMaxOfRGB(void) const;
Finds the maximum intensity.
```

```
boolean IsClamped(void) const;
Use this method to see if the color falls in between [0, 1]; mainly for debugging.
```

```
void Clamp (void);
Clamp the color value to between [0, 1].
```

```
void Interpolate (const TMColorRGB&from, const TMColorRGB& to, const real
interpolant);
void Interpolate (const TVector2& interpolants, const TMColorRGB& lowerLeft,
const TMColorRGB& lowerRight, const TMColorRGB& upperLeft, const TMCo-
lorRGB& upperRight);
```

Linear/bilinear interpolation, interpolants must be between 0 and 1. X values are left/right. Y values are lower/upper.

```
inline void AccumRGB(const TMColorRGB&rhs, const real scalar );
Color accumulation is used so often and we want to take advantage of fmul-add. This is equiv to *this +=
rhs * scalar; no clamping is done. In TMColorRBB, the alpha information is ignored.
```

Operators

These are operators on all the components:

```
inline boolean operator==(const TMColorRGB& rhs ) const;
inline boolean operator!=(const TMColorRGB& rhs ) const;
inline boolean operator<(const real t) { return red<t || green<t || blue<t; }
inline boolean operator>(const real t) { return red>t || green>t || blue>t; }
```

```
inline TMColorRGB& operator+=(const TMColorRGB& rhs);
inline TMColorRGB& operator-=(const TMColorRGB& rhs);
inline TMColorRGB& operator*=(const real s);
inline TMColorRGB& operator/=(const real s);
```

```
inline TMColorRGB operator+(const TMColorRGB& rhs) const;
inline TMColorRGB operator-(const TMColorRGB& rhs) const;
inline TMColorRGB operator*(const real s) const;
inline TMColorRGB operator/(const real s) const;
```

Dot, term-by-term product:

```
inline real operator*(const TMColorRGB& rhs) const;
inline TMColorRGB operator%(const TMColorRGB& rhs) const;
```

Conversion operations to the other colors:

```
TMColorRGBA8&ToRGBA8(TMColorRGBA8&c, const uint8 alphaValue) const;
TMColorRGBA8&ToRGBA8Unclamped(TMColorRGBA8& c, const uint8 alphaValue) const;

TMColorRGBA&ToRGBA(TMColorRGBA& c, const float alphaValue) const;
```

Conversion from other colors:

```
inline TMColorRGB& operator=(class TMColorRGBA &c);

inline const real* GetAddrFirstComponent(void) const;
Returns the address of first component in the structure.
```

To facilitate a simpler move away from old colors, do not use R,G,B,A in new code. Use the full names instead.

```
union { real red; real R; };
union { real green; real G; };
union { real blue; real B; };
```

Fields*Red*

The Red value in RGB mode, or the Cyan value in CMYK. From 0.0 to 1.0.

*Green*

The Green value in RGB mode, or the Magenta value in CMYK. From 0.0 to 1.0.

*Blue*

The Blue value in RGB mode, or the Yellow value in CMYK. From 0.0 to 1.0.

Comments

All rendering and shading calculations are done in RGB mode.

Example

White = (1.0, 1.0, 1.0)

Black = (0.0, 0.0, 0.0)

ETreeDeformBBoxModes

ETreeBoundingBoxModes is used to indicate which kind of bounding box is to be given by GetBoundingBox. It is a method of I3DShGroup, I3DShInstance, and I3DShTreeElement.

Variables

kApplyNoDeformer	0	Gives the original box (without applying any deformer).
kApplyDeformersLocally	1	At each tree (group) level accumulates the bboxes of the children (recursively computed with kApplyDeformersLocally) and then applies the tree deformers on the resulting box.
kApplyAllDeformersAtInstanceLevel	2	Valid only for instances, deforms the original box of the instance by all the deformers of the tree the instance is on.
kUnusedApplyDeformerConstant	0xFFFFFFFF	To ensure "castable" as uint32.

Chapter 4 - Shading Data Structures Reference

ShadingIn

Description

The ShadingIn data structure is used in conjunction with the other Shading data structures to perform shading. It contains all the input parameters provided to the Shader according to how the “*ShadingFlags*” were set earlier by the Shader. Fields that were not requested in the “*ShadingFlags*” record have undefined values. Do not use them.

See the “**I3DExShader Interface**” documentation for further reference. Also read the description of the other Shading data structures, especially “*ShadingFlags*” and “*ShadingOut*”.

Field Index

<Section>Appendix , “TVector3 fPoint,<Section>”
 <Section>Appendix , “TVector3 fGNormal,<Section>”
 <Section>Appendix , “TVector3 fIsoU,<Section>”
 <Section>Appendix , “TVector3 fIsoV,<Section>”
 <Section>Appendix , “TVector2 fUV,<Section>”
 <Section>Appendix , “uint32 fUVSpaceID,<Section>”
 <Section>Appendix , “TVector3 fPointLoc,<Section>”
 <Section>Appendix , “TVector3 fNormalLoc,<Section>”
 <Section>Appendix , “boolean fBumpOn,<Section>”
 <Section>Appendix , “uint32 fCurrentCompletionMask,<Section>”
 <Section>Appendix , “I3DShInstance* fInstance,<Section>”
 <Section>Appendix , “UVSpaceInfo* fUVInfo,<Section>”
 “TVector3 fPointx and fPointy”
 “TVector3 fNormalx and fNormaly”
 “TVector3 fPointLocx and fPointLocy”
 “TVector3 fNormalLocx and fNormalLocy”
 “TVector2 fUVx,fUVy”

<zot>Please check the descriptions for all of these fields.</zot>

TVector3 fPoint

Surface point in Global Coordinates. Examples of such shaders are volumetric procedural shaders for objects that are “cut” in a global matter. An example would be a “Mountain” texture where the rock color depends on the altitude.

TVector3 fGNormal

The Normal to the surface, in Global Coordinates System. An example of such a shader could be a “Snowy Mountains shader”: when the Normal is looking up vertically (which mean that the surface is horizontal), the shader would return a snow color. When the Normal is horizontal, it would return a gray rocky color (because the snow does not hold on steep slopes).

TVector3 fIsoU

The U iso-parametric vector in Local Coordinates. See the “Database Overview” chapter for more details on these vectors. Briefly, they can be described as the vectors that are tangent to the u,v Iso-parametric curves. These curves are the curves on the surface that have a constant u (or v) value. For example, the meridians and parallels on a globe are Iso-parametric curves.

These vectors are necessary to perform Bump effects; if the Normal is to be perturbed based on the u and v values, one needs to know in which direction to perturb it. *fIsoU* and *fIsoV* provide that information.

TVector3 fIsoV

The V iso-parametric vector in Local Coordinates. See the “Database Overview” chapter for more details on these vectors. Briefly, they can be described as the vectors that are tangent to the u,v Iso-parametric curves. These curves are the curves on the surface that have a constant u (or v) value. For example, the meridians and parallels on a globe are Iso-parametric curves.

These vectors are necessary to perform Bump effects; if the Normal is to be perturbed based on the u and v values, one needs to know in which direction to perturb it. *fIsoU* and *fIsoV* provide that information.

TVector2 fUV

The u,v texture values. See the “Database Overview” chapter for more details on UV Space. Briefly, a UV Space is a 2D coordinate system wrapped on the object surface (like the longitude and latitude parameters of a globe). There can be several UV Spaces on an object: each one has a unique UV Space ID.

uint32 fUVSpaceID

The UV Space ID of the UV Space to which the *fUV* values belong. It can use UVSpaceID enum.

TVector3 fPointLoc

The surface point in Local Coordinates. Examples of such shaders are volumetric procedural shaders like a wood function. The shading will not depend on the position of the object in the Scene.

TVector3 fNormalLoc

The surface Normal in Local Coordinates. See the Rainbow shader example in the toolkit. A shader can use *fNormalLoc* to perform shading that does not depend on the orientation of the object in the Scene.

boolean fBumpOn

The global bump setting.

uint32 fCurrentCompletionMask

Internal flags for shaders. Must be set to 0 before calling **DoShade()**.

I3DShInstance* fInstance

Used by ShadingMode to calculate the bounding Box.

UVSpaceInfo* fUVInfo

UV Space information will be filled by I3DShInstance::**DoShade()**.

TVector3 fPointx and fPointy

Surface point derivatives in Global Coordinates.

TVector3 fNormalx and fNormaly

Surface Normal derivatives in Global Coordinates.

TVector3 fPointLocx and fPointLocy

Surface point derivatives in Local Coordinates.

TVector3 fNormalLocx and fNormalLocy

Surface Normal derivatives in Local Coordinates.

TVector2 fUVx, fUVy

Texture derivatives.

ShadingFlags

Description

Shading flags are used by the 3D Shell to determine which fields of the *ShadingIn* structure the Shader requires. This allows the 3D Shell to use the minimum data and to convert between different types.

See the *ShadingIn* data structure for complete details on the meaning of each field.

Field Index

Zot didn't find fCallOnce. Please confirm it should be removed.

Fields

☞ *fCallOnce*

Constant Shader flag. If the Shader is constant, i.e. if it does not depend on any fields of the **ShadingIn** structure, then it should return TRUE in this field. Then its shading procedure will be called only once.

Such shaders are a regular Color Shader or a Value Shader.

↳ *fNeedsColor*

This field is for future use.

↳ *fNeedsPoint*

The Shader needs the surface point in Global Coordinates (**ShadingIn::fPoint**).

↳ *fNeedsNormal*

The Shader needs the Normal to the surface, in the Global Coordinates System (**ShadingIn::fNormal**).

↳ *fNeedsIsoUV*

The Shader needs the U or V iso-parametric vector in Local Coordinates (**ShadingIn::fIsoU** and **ShadingIn::fIsoV**).

↳ *fNeedsUV*

The Shader needs the u,v texture values (**ShadingIn::fUV**).

↳ *fNeedsPointLoc*

The Shader needs the surface point in Local Coordinates (**ShadingIn::fPointLoc**).

↳ *fNeedsNormalLoc*

The Shader needs the surface Normal in Local Coordinates (**ShadingIn::fNormalLoc**).

↳ *fNeedsPixelRatio*

The Shader needs the Pixel Ratio. This field is not used at this time, and reserved for future Pixel Ratio support.

↳ *fChangesNormal*

The Shader will change the Normal. This allows the Shader to tell the 3D Shell that a new Normal (**ShadingOut::fChangedNormal**) will have to be read after calling the Shader. This is intended for Shaders that changes the Normal like Bump mapping Shaders.

↳ *fUVSpaceShaders*

Returns true if there are different shaders in UVSpaceId.

↳ *fConstantChannelsMask*

zot needs description, here is the comment from the code: // use these enum:
kColorChannel,kSpecularityChannel,kShininessChannel,kNormalChannel...

Comments

See the **I3DExShader** Interface documentation for further reference. Also read the description of the other Shading data structures, especially **ShadingIn** and **ShadingOut**.

ShadingOut

Description

The **ShadingOut** data structure is filled out by Shaders that implement the **I3DExShader::DoShade** call. It is also used by renderers that call **I3DShInstance::DoShade** to get the shading parameters of an Instance in order to calculate the lighting on the surface (using a Phong Illumination Model for example). In both cases, the **ShadingOut** data structure is included in a **ShadingInOut** data structure.

Field Index

Fields

- ⇒ *fColor*
Diffuse Color (Kd).
- ⇒ *fSpecularColor*
Specular Color (Ks).
- ⇒ *fSpecularSize*
Specular highlight size (Ns).
- ⇒ *fAmbient*
Ambient factor (Fa).
- ⇒ *fLambert*
Diffuse factor (Fd).
- ⇒ *fReflection*
Reflectivity (Rx).
- ⇒ *fTransparency*
Transparency (Tx).
- ⇒ *fGlow*
Glow color (G)
- ⇒ *fRefractiveIndex*
Refraction index (Nr)
- ⇒ *fChangedNormal*
Changed Normal in Global Coordinates. The Shader will set this field if it told the 3D Shell that it will change the Normal (see **ShadingFlags::fChangesNormal**).
- ⇒ *fOpacity*
Opacity index. Used when mixing shaders together such as a Layers List shader.

↳ *fFullAreaDone*

True if the shader was evaluated on the area.

Comments See the description of the Illumination Model in the “Writing a Renderer” chapter in the Cookbook.

UVSpaceInfo

Description

The **UVSpaceInfo** record is filled out by **I3DExGeometricPrimitive::GetUVSpace**, **I3DShObject::GetUVSpace**, and **I3DShInstance::GetUVSpace** to give all the necessary information on the UV Space(s) of the Objects and Instances.

Field Index

Fields

↳ *fID*

The UV Space ID identifying this UV Space.

↳ *fName*

The UV Space ID name.

↳ *fMin*

The minimum u and v values of the UV Space

↳ *fMax*

The maximum u and v values of the UV Space

↳ *fWraparound*

This array of two boolean values indicate if the UV Space is closed in the u or v direction. For example, a hollow cylinder would be closed in one direction (for instance the v). In this case, Shading Elements can cross over the v=vmax boundary.

ShadedArea

zot I only found one reference to this in I3DShInstance. should it be removed?

Chapter 5 - Geometry Data Structures Reference

TVERTEX3D

Used with a facet mesh.

```
TVector3 fVertex;  
X, y, z vertex coordinates.
```

```
TVector3 fNormal;  
Nx, Ny, Nz normal values at that vertex.
```

```
TVector2 fUV;  
Texture u,v values at that vertex.
```

TFACET3D

The structure for a facet mesh.

```
TVertex3D fVertices[3];  
The facet's three vertices. This includes it's vertex coordinates, normal values, and texture uv values.
```

```
int16 fUVSpace;  
The UV Space ID to which this facet belongs.
```

```
int16 fReserved;  
Reserved - 0. zot does this mean set it to zero?
```

TPATCH3D

<zot>need description here

```
TVector3fVertices[4][4];  
The patches' 16 vertices.
```

```
real fu[2];  
u values at the patch boundaries.
```

```
real fv[2];  
v values at the patch boundaries.
```

```
int16 fUVSpace;  
UV Space ID to which this patch belongs.
```

```

        int16      fReserved;
Reserved - 0.

};

```

FacetMesh

A FacetMesh is a representation for triangular polygon data using corresponding arrays of data.

There are several more routines in PublicUtilities.h for managing facet meshes. Be careful about maintaining the polygon eqn and edge lists.

<zot> what is a "eqn"?

Methods

```
static void Create(FacetMesh** outMesh)
Creates a facet mesh.
```

```
void Clone (FacetMesh** clone);
Clone's a facet mesh.
```

```
inline boolean IsEdgeListInvalid() const
{ return fEdgeList.GetElemCount() == 0; }
Call this routine anytime the facets change, the edge generators rely on this to generate the edge lists.
```

```
inline void MarkEdgeListInvalid()
{ fEdgeList.SetElemCount( 0 ); }
Call this routine anytime the facets change, the plane generators rely on it to generate the polygon equations.
This method doesn't release the memory.
```

```
inline boolean IsPolygonEqnListInvalid() const
{ return fPolygonEqns.GetElemCount() == 0; }
zot needs description.
```

```
inline void MarkPolygonEqnListInvalid()
{ fPolygonEqns.SetElemCount( 0 ); }
This method doesn't release memory.
```

```
inline void CalcPolygonEqnList()
Call this method to calculate the list of plane equations.
```

These are the plane equations for the polygons (a combination of unit normal and plane coefficients.) MetaCreations uses the plane eqn of the form for compute the plane coefficient::

$$A x + B y + C z + D = 0$$

zot add ECloneMode here? (APITypes.h), or to the basic data structs list?

Chapter 6 - Ray Tracing Data Structures Reference

Ray3D

The Ray3D structure is used to describe a light ray. This structure is mostly used by ray-tracing algorithms.

Fields

⇒ *fOrigin*

Origin of the ray in Global Coordinates.

⇒ *fDirection*

Direction of the ray in Global Coordinates. It must be a unit vector.

⇒ *fOx*

Partial derivative of the origin.

⇒ *fOy*

Partial derivative of the origin.

⇒ *fDx*

Partial derivative of the direction.

⇒ *fDy*

Partial derivative of the direction.

zot didn't find these: Remove?

⇒ *fFocalPoint*

Coordinate of the focal point of the light beam. This coordinate is a coordinate along the ray's line, i.e. the focal point is at: $fOrigin + fFocalPoint * fDirection$

⇒ *fAngle*

Angle of the light beam. In fact, this is not really an angle, but more the sinus of the light beam angle. It controls how the light beam spreads out. See the formula below.

⇒ *fWidthAtFocal*

Width of the light beam at the focal point.

Comments

fFocalPoint, *fAngle* and *fWidthAtFocal* are made to calculate the "pixel size", i.e. the size that the light beam covers at a specific point if it is considered as a cone of light rather than a straight line. The formula works like this:

$$Size(t) = fWidthAtFocal + (t - fFocalPoint) * fAngle$$

Where 't' is the "ray parameter": any point on the ray can be written as:

$$fOrigin + t * fDirection \quad \text{with } t \text{ as a real value.}$$

RayHit3D

The RayHit3D structure is used by **I3DExGeometricPrimitive::RayHit** and **I3DExGeometricPrimitive::RayAllHits**. Its purpose is to return ray-tracing hit information when a hit occurs between a ray and a Primitive's surface.

Constructor

```
RayHit3D()
```

Fields

```
fShouldSetUV=true;
fShouldSetIsoUV=true;
fRenderInfinitePrimitives=true;
fShouldSetNormalDerivative=false;
```

Methods

```
void CalcInfo(ShadingFlags &shadingFlags, boolean calcDerivatives, Ray3D &ray);
```

Variables

```
real    ft;
    t parameter along the ray: fPosition = RayOrigin + t * RayDirection. Ray parameter of fPosition.
    fPosition = RayOrigin + ft * RayDirection.
```

```
TVector3 fReflectDir;
    Direction of reflection of the ray.
```

```
TTransform3DfT;
zot Needs description.
```

```
TTransform3DfInvT;
zot Needs description.
```

```
TMatrix33fNormalDerivative;
    Derivative of the normal depending on the position in local coordinates.
```

```
TVector3fUVDerivative[2];
    Derivative of the UV depending on the position in local coordinates
```

```
FacetInfo*fFacetInfo;
    Private data of the raytracer ( used in fCalcInfo ).
```

```
real    fBaryCoord[3];
```

Barycentric coordinates of the hit point on a hot facet.

```
int32 fFacetIndex;
```

Facet hit (index in facetMesh with best Level Of Detail).

```
FacetMesh*fFacetMesh;
```

Pointer to the facet mesh

```
void (*fCalcInfo)(RayHit3D &me);
```

zot needs description.

```
boolean fShouldSetUV;
```

Returns True if fUV should be set.

```
boolean fShouldSetIsoUV;
```

Returns True if fIsoU and fIsoV should be set.

```
boolean fRenderInfinitePrimitives;
```

Returns True if infinite primitives are rendered.

```
boolean fShouldSetNormalDerivative;
```

Returns True if fNormalDerivate should be set (it is only needed for reflection and transparency.)

```
boolean fNormalFlipped;
```

Returns True if the Normal is flipped.

```
boolean fIsShadowCatcher;
```

True if the object is a shadowcatcher.

```
real fShadowCatchingCoef;
```

Returns the percentage of the light that is not intercepted by an object (computed only for shadow catcher.)

Comments See the sphere example in the geometric primitive cookbook for more information on this structure.

RayHitParameters

The RayHitParameters structure is used by **I3DShInstance::RayHit** and **I3DExGeometricPrimitive::RayHit**. to calculate the hit of a ray on an object.

Fields

➡ *tmin*

Minimum value for the parametric variable t.

➡ *tmax*

Maximum value for the parametric variable t, that mean that for this value, the ray hit an object and don't go thru. If no object have been found yet, the maximum value is set instead.

➡ **ray*

zot needs description.

➡ **hit*

zot needs description.

Comments See **Ray3D** structure for more details.

HitInfo

The HitInfo data structure works in conjunction with the RayHit3D structure. If a hit occurs and that the ray-tracing needs all the information on this hit, then **I3DExGeometricPrimitive::GetRayHitDetails** is called.

Constructor

```
HitInfo()
```

Variables

```
fInstance=nil;
fNeedsColor=false;
fNeedsDistance=false;
fNeedsPosition=false;
fNeedsNormal=false;
fNeedsAlpha=false;
fNeedsIndex=false;
fNeedsUV=false;
fZBShading=false;
fFilterInstanceProc=nil;
```

Output data

```
TInstance* fInstance;
Instance that was hit by the ray.
```

```
VECTOR2D fUV;
(u,v) texture values at the intersection point. To be filled only if fNeedsUV is TRUE.
```

```
uint32 fUVSpaceID;
UV Space ID at the intersection point.
```

```
VECTOR3D fPosition;
Intersection point in Global Coordinates. To be filled only if fNeedsPosition is TRUE.
```

```
VECTOR3D fNormal;
Normal at the intersection point in Global Coordinates. To be filled only if fNeedsNormal is TRUE.
```

```
COLOR3D fColor;
Rendered final color (all lighting and shading included). To be filled only if fNeedsColor is TRUE.
```

```
NUM3D fAlpha;
Alpha-channel value. To be filled only if fNeedsAlpha is TRUE.
```

```
uint32 fIndex;
Instance unique index. To be filled only if fNeedsIndex is TRUE.
```

```
NUM3D fDistance;
Distance from the point of view to the intersection point. To be filled only if fNeedsDistance is TRUE.
```

Input flags

```
boolean fNeedsColor;
```

Returns True if Needs fColor.

`boolean fNeedsDistance;`
Returns True if Needs fDistance.

`boolean fNeedsPosition;`
Returns True if Needs fPosition.

`boolean fNeedsNormal;`
Returns True if Needs Normal.

`boolean fNeedsAlpha;`
Returns True if Needs Alpha.

`boolean fNeedsIndex;`
Returns True if Needs Instance index.

`boolean fNeedsUV; //`
Returns True if Needs (u,v) values. These flags control the calculation of the returned data.

`boolean fZBShading;`
zot, this was comment: Ask for Z-Buffer-like shading: no transparencies, no reflections and no shadows

`FilterHitProc fFilterInstanceProc;`
Filterproc for instances. Used only in fast hit mode when fNeedsColor is false.

Comments See the sphere example in the geometric primitive cookbook chapter for more information on this structure.

Chapter 7 - 3D IEx* Reference

7.1 I3DExAmbientLight Interface

Family ID: 'ambi'

Interface ID: IID_I3DExAmbientLight

Interface file: I3DExPrimitive.h

Basic Implementation file: BasicEnvironment.h, BasicEnvironment.cpp

Description

This Ambient Light shines from all directions, and therefore does not cast any shadows.

I3DExAmbientLight::GetColor

MCCOMErr I3DExAmbientLight::GetColor(TMCColorRGB* resultColor);

Gets the color of the ambient light.

Parameters  **TMCColorRGB* resultColor**
The color of the ambient light to return. This is global for the scene.

Return Value MC_E_NOTIMPL

7.2 I3DExAtmosphericShader Interface

Family ID: 'atmo'

Interface ID: IID_I3DExAtmosphericShader

Interface file: I3DExPrimitive.h

Basic Implementation file: BasicEnvironment.h, BasicEnvironment.cpp

Description

The atmospheric shader is used to create effects such as fog, clouds, etc. The atmospheric shader filters the light beams and can modify them at will.

The following two functions are implemented:

- **SegmentFilter**(TVector3& beg, TVector3& end, TMCColorRGB& inOutFilter)
- **DirectionFilter**(TVector3& origin, TVector3& direction, TMCColorRGB& inOutFilter)

Methods

"I3DExAtmosphericShader::SegmentFilter"

"I3DExAtmosphericShader::DirectionFilter"

I3DExAtmosphericShader::SegmentFilter

MCCOMErr I3DExAtmosphericShader::SegmentFilter(TVector3& beg, TVector3& end, TMCColorRGB& inOutFilter)

SegmentFilter is called to filter the light beam color between two points.

Parameters

↗ **TVector3& beg**

This point is the beginning point of the light beam in Global Coordinates.

↗ **TVector3& end**

This point is the ending point of the light beam in Global Coordinates.

↔ **TMCColorRGB& inOutFilter**

The Light beam color is given as an input and output parameter. The component should modify this color to perform an atmospheric effect.

Return Value

False is MC_E_NOTIMPL

True is S_OK

Comments

The Atmospheric filtering can be based on the values of the beginning and ending points, the distance between the two points, etc.

I3DExAtmosphericShader::DirectionFilter

MCCOMErr I3DExAtmosphericShader::DirectionFilter(TVector3& origin, TVector3& direction, TMCColorRGB& inOutFilter)

DirectionFilter works like **SegmentFilter**, but is called to filter any light beam going away to the infinite. In this case, there is no ending point, but a direction vector instead.

Parameters

→ *beg*

This point is the beginning point of the light beam in Global Coordinates.

→ *direction*

The direction of the light beam in Global Coordinates. This is a unit vector.

↔ *filterOut*

The Light beam color is given as an input and output parameter. The component should modify this color to perform an atmospheric effect.

Return Value

False is MC_E_NOTIMPL

True is S_OK

Comments

The Atmospheric filtering should be based on the values of the beginning point and the direction.

Example

Fog at the horizon can be done this way: if the light beam is going to the horizon (i.e. *direction* is horizontal), filtering would be performed; if the light beam is going up to the sky, not filtering would be done. The result is a foggy horizon and a clear sky.

7.3 I3DExBackground Interface

Family ID: 'back'

Interface ID: IID_I3DExBackground

Interface file: I3DExPrimitive.h

Basic Implementation file: BasicEnvironment.h, BasicEnvironment.cpp

Description

Backgrounds are 3D environments that can be reflected on the 3D objects surfaces (like a sky).

I3DExBackground::GetBackgroundColor

MCCOMErr I3DExBackground::GetBackgroundColor(*TMCColorRGB& color, boolean& fullAreaDone, const TVector3& direction*);

I3DExBackground::GetBackgroundColor is called to perform environment backgrounds (sometimes called environment mapping). Environments are reflected on reflecting objects.

Parameters

→ **TMCColorRGB& color**

Return the calculated color here.

→ **boolean& fullAreaDone**

Return True if the full pixel area has been shaded (not used, return false for the moment).

→ **TVECTOR3D& direction**

The unit vector defining the direction of the light beam.

Return Value MC_E_NOTIMPL

Comments

The direction *parameter* is enough because one can consider that the environment is an image mapped on the inside surface of a infinite sphere around the scene.

Example

This is a white to blue sky, with a gray ground:

7.4 I3DExBackdrop Interface

Family ID: 'drop'

Interface ID: IID_I3DExBackdrop

Interface file: I3DExPrimitive.h

Basic Implementation file: BasicEnvironment.h, BasicEnvironment.cpp

Description

Backdrops are image at infinity (behind every objects) in the axis of the camera. A backdrop can not be reflected on the objects.

The following function is implemented:

- **GetBackdropColor**(TMCColorRGB& color,boolean& fullAreaDone,const TBBBox2D& screenMinMax,const TVector2& uv)

I3DExBackdrop::GetBackdropColor

MCCOMErr I3DExBackdrop::GetBackdropColor(TMCColorRGB& color,boolean& fullAreaDone,const TBBBox2D& screenMinMax,const TVector2& uv);

GetBackdropColor is called to return the backdrop color information.

Parameters

- ✎ **TMCColorRGB& color**
Return the calculated color here.
- ✎ **boolean& fullAreaDone**
Return True if the full area of the pixel has been shaded.
- ✎ **TBBBox2D& screenMinMax**
Box corresponding to the production frame in screen coordinates.
- ✎ **const TVector2& uv**
Position on the screen.

Return Value

S_OK

Comments

Remember that backdrops are like composing the rendered image with another background image. Backdrops are not reflected on reflective objects. Use backgrounds for this.

You need to use the screenMinMax coordinates because the uv coordinates that are given to you might be outside the production frame (for example when the marquee rendering tools are used in the interactive preview).

7.5 I3DExCamera Interface

Family ID : 'came'

Interface ID : IID_I3DExCamera

Interface file : I3DExCamera.h

Basic Implementation files: BasicCameraLightGel.h, BasicCameraLightGel.cpp

Description

Cameras define the projection from the 3D world to the 2D screen on which the image is rendered. There are two common forms of projection: orthographic and perspective.

An orthographic camera takes rays parallel to the z-axis and projects them on the x-y plane by removing z.

A perspective camera takes all rays coming to a point and intersects those rays with a plane some distance from the point. The distance to this plane is the focal length. As with a real camera, it is possible to implement various lenses, focal values, zoom effects, etc.

Projection is not restricted to the conical projection (the natural projection in your eye), so isometric or fish-eye cameras can be designed.

There are three basic methods. **Project3DTo2D()** is used to create the projection from 3D space onto the screen. This call is used by the hybrid raytracer. The two **CreateRay()** function calls create a 3D ray from a screen point. These calls are used for the ray-tracing. (You need to implement both **CreateRay()** calls.)

For your camera to be used by the interactive renderer, you must implement **GetStandardCameraInfo()**. For performance reasons, the interactive renderer requires a standard isometric or canonical camera. The parameters for this standard camera are supplied by **GetStandardCameraInfo()**.

Method Index

```
"I3DExCamera::SetTransform(const TTransform3D&)"
"I3DExCamera::CreateRay(const TVector2&, TVector3&, TVector3& )"
"I3DExCamera::CreateRay(const TVector2&, const TVector2&, Ray3D&)"
"I3DExCamera::IsProjectionAffine(void)"
"I3DExCamera::PreProject3DTo2D(const TVector3&, TVector4&)"
"I3DExCamera::Project3DTo2D(const TVector3&, TVector3&)"
"I3DExCamera::GetPrimitiveID()"
"I3DExCamera::GetStandardCameraInfo(TStandardCameraInfo&)"
"I3DExCamera::SetHalfWidth(real)"
```

I3DExCamera::SetTransform(const TTransform3D&)

MCCOMErr I3DExCamera::SetTransform(const TTransform3D& transform)

I3DExCamera::SetTransform() is used to store the camera transformation data.

Parameter  **const TTransform3D& transform**
 transform is a reference to the transformation data. Copy its contents to an internal variable.

Return Value MC_S_OK

Comments Pre-processing calculations for the camera extension can occur in **SetTransform()**. You should cache this transform in your camera and use it to implement **CreateRay()** .

I3DExCamera::CreateRay(const TVector2&, TVector3&, TVector3&)

boolean I3DExCamera::CreateRay(const TVector2& screenPosition, TVector3& worldOrigin, TVector3& worldDirection)

I3DExCamera::CreateRay() is called when a ray (i.e. a straight line) from a screen position needs to be created.

Parameters

- ✚ **const TVector2& screenPosition**
Position on the screen in screen coordinates. (This can be any floating point value as the camera screen is considered infinite.) Values between -1 and 1 corresponds to the Production Frame.
- ✚ **TVector3& worldOrigin**
worldOrigin must return the screen point, in 3D, in the Global Coordinates System. You need to use the transformation provided by SetTransform() to perform those calculations.
- ✚ **TVector3& worldDirection**
worldDirection must return the direction vector of the straight line. This must be a unit vector.

Return Value MC_S_OK

I3DExCamera::CreateRay(const TVector2&, const TVector2&, Ray3D&)

boolean CreateRay(const TVector2& screenPosition, const TVector2& screenDerivate, Ray3D& ray)

I3DExCamera::CreateRay() is called by the renderer to get a ray and its derivatives.

Parameters

- ✚ **const TVector2& screenPosition**
Position on the screen (can be any floating point value as the camera screen is considered infinite). Values between -1 and 1 corresponds to the Production Frame.
- ✚ **TVector2& screenDerivate**
screenDerivate specifies the size of a pixel on the screen in screen coordinates.
- ✚ **Ray3D& ray**
ray specifies the specific ray being created. You must fill fOx,fOy with the variation of the origin of the ray over a pixel and fDx,fDy with the variation of the direction over a pixel. You should use the value given in screeDerivate for that.

Return Value MC_S_OK

I3DExCamera::IsProjectionAffine(void)

boolean IsProjectionAffine(void)

I3DExCamera::IsProjectionAffine returns true if the w component from the projection is always 1.0.

Return Value

TRUE	w is always 1.0
FALSE	w is not always 1.0

I3DExCamera::PreProject3DTo2D(const TVector3&, TVector4&)

boolean PreProject3DTo2D(const TVector3& cameraPosition, TVector4& projectedPosition)
I3DExCamera::PreProject3DTo2D is called to project a 3D point on the screen.

Parameters

-  **const TVector3& cameraPosition**
The 3D position of the camera.
-  **TVector4& projectedPosition**
Projection on the screen.

Return Value

TRUE	The point is visible.
FALSE	The point is not visible.

I3DExCamera::Project3DTo2D(const TVector3&, TVector3&)

boolean Project3DTo2D(const TVector3& cameraPosition, TVector3& screenPosition)
I3DExCamera::Project3DTo2D is called to project a 3D point on the screen.

Parameters

-  **const TVector3& cameraPosition**
The 3D position in camera space.
-  **TVector3& screenPosition**
Projection on the screen in screen coordinates.

Return Value

TRUE	The point is visible.
FALSE	The point is not visible.

I3DExCamera::GetPrimitiveID()

uint32 GetPrimitiveID();
I3DExCamera::GetPrimitiveID is called to get the primitive ID of the camera.

Return Value The primitive ID.

I3DExCamera::GetStandardCameraInfo(TStandardCameraInfo&)

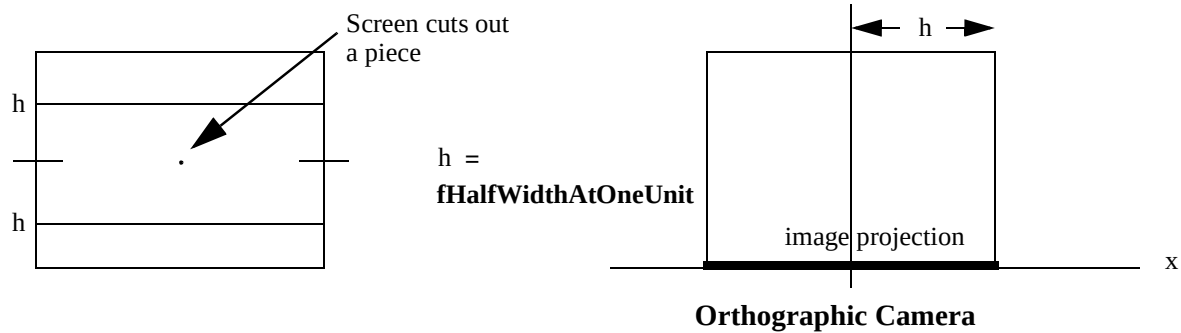
void GetStandardCameraInfo(TStandardCameraInfo& cameraInfo)
I3DExCamera::GetStandardCameraInfo is used to return standard camera info used by the interactive renderer.

TStandardCameraInfo has the following fields:

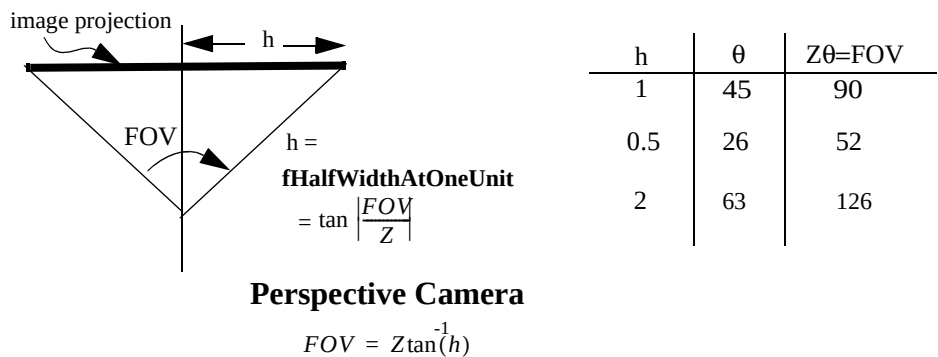
- **EProjectionType fProjectionType** - pick the type of camera that you are most closely approximating between conical (perspective) and isometric (orthographic).
- **ECanonicalCameraType fCanonicalCameraType** - for internal use only (ignore).

- **real fHalfWidthAtOneUnit** - the amount of camera space units at one unit from the camera location. In the case of an orthographic camera, this represents the zoom factor. For the perspective camera, this represents the field of view (FOV), or the angle of the most extreme rays that are in the image. Any rays outside of the FOV are not represented in the image. (See diagrams below for graphical description.)
- **boolean flsMatching** - indicates whether the standard camera info is an exact or approximate conversion. For example, a spherical lens may only approximate itself with one of the types.

As you can see from the diagram below, for an orthographic camera, a large **fHalfWidthAtOneUnit** means zoom out.



As shown below, with a perspective camera, a large **fHalfWidthAtOneUnit** means the FOV is large, like a wide angle lens, and the image is zoomed out.



Parameters **TStandardCameraInfo& cameraInfo**
The camera info.

I3DExCamera::SetHalfWidth(real)

MCCOMErr SetHalfWidth(real value)

I3DExCamera::SetHalfWidth is used to explicitly specify the viewable area. It is typically used for isometric (orthographic) cameras.

Parameters **real value**
value is the HalfWidth of the camera screen. For an isometric camera, *value* is 20.

Return Value MC_S_OK

7.6 I3DExConstraint

Interface ID: IID_I3DExConstraint

Interface File: I3DExConstraint.h

Description

I3DExConstraint provides constraints between Tree Elements. A constraint, known as a link in RDS, defines motion links between Tree Elements. It allows you to define “mechanical” constraints (degrees of freedom) between one Tree Element and its father.

Method Index

“I3DExConstraint::GetNbrFreedom()”
 “I3DExConstraint::IncrementFreedomValue(int16 , real&)”
 “I3DExConstraint::GetMotionLimit (int16, ELimitType&)”
 “I3DExConstraint::GetFreedomRange(int16, real&, real&)”
 “I3DExConstraint::GetTransform(TTransform3D&)”
 “I3DExConstraint::GetTransformPartialDerivate(int16, TTransform3D&)”
 “I3DExConstraint::GetTransformSecondPartialDerivative (int16, int16, TTransform3D&)”
 “I3DExConstraint::SetTransform(TTransform3D&)”
 “I3DExConstraint::IsTransformResetRequired()”
 “I3DExConstraint::ResetConstraint()”

I3DExConstraint::GetNbrFreedom()

int16 MCCOMAPI GetNbrFreedom()

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.

I3DExConstraint::IncrementFreedomValue(int16 , real&)

MCCOMErr MCCOMAPI IncrementFreedomValue(int16 index, real& increment)

<zot>Need method description</zot> The Shell can’t directly modify the freedom value, so you need to implement **IncrementFreedomValue()**.

Parameters

 **int16 index**
<zot>Need Description</zot>

 **real& increment**
<zot>Need Description</zot>

Return Value MC_S_OK

I3DExConstraint::GetMotionLimit (int16, ELimitType&)

MCCOMErr MCCOMAPI GetMotionLimit (int16 index, ELimitType& limit)

<zot>Need method description</zot>

Parameters

- **int16** *index*
<zot>Need Description</zot>
- **ELimitType** *limit*
<zot>Need Description</zot>

I3DExConstraint::GetFreedomRange(int16, real&, real&)

MCCOMErr MCCOMAPI GetFreedomRange(int16 index, real& min, real& max)

GetFreedomRange() returns the return the maximum range of increment and decrement.

Parameters

- **uint32** *index*
<zot>Need Description</zot>
- **real** *min*
min is the minimum increment value.
- **real** *max*
max is the maximum increment value.

I3DExConstraint::GetTransform(TTransform3D&)

MCCOMErr MCCOMAPI GetTransform(TTransform3D& transform)

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.

Parameters

- **TTransform3D&** *transform*
Reference to the rotation matrix and translation vector. See TTransform in the data structures chapter.

I3DExConstraint::GetTransformPartialDerivate(int16, TTransform3D&)

MCCOMErr MCCOMAPI GetTransformPartialDerivate(int16 index, TTransform3D& transform)

To allow the inverse kinematics mechanism to work with a constraint, implement the **GetTransformPartialDerivate()** function. A partial derivative transformation is a transformation where each element is derived by a freedom degree.

Parameters

- **int16** *index*
<zot>Need Description</zot>
- **TTransform3D&** *transform*
Reference to the rotation matrix and translation vector. See TTransform in the data structures chapter.

I3DExConstraint::GetTransformSecondPartialDerivative (int16, int16, TTransform3D&)

MCCOMErr MCCOMAPI GetTransformSecondPartialDerivative (int16 index1, int16 index2, TTransform3D& transform)

<zot>Need method description</zot>

Parameters

➦ **int16 index1**
<zot>Need Description</zot>

➦ **int16 index2**
<zot>Need Description</zot>

➦ **TTransform3D& transform**
Reference to the rotation matrix and translation vector. See TTransform in the data structures chapter.

I3DExConstraint::SetTransform(TTransform3D&)

MCCOMErr MCCOMAPI SetTransform(TTransform3D& transform)

SetTransform() sets the rotation matrix and translation vector .

Parameters

➦ **TTransform3D& transform**
Reference to the rotation matrix and translation vector. See TTransform in the data structures chapter.

I3DExConstraint::IsTransformResetRequired()

virtual boolean MCCOMAPI IsTransformResetRequired()

Call **IsTransformResetRequired()** after an event in the constraint chooser. A reset is required when the constraint has changed in nature or in axis. That is when the transform will no longer match. The default is return true if you did not handle the change.

I3DExConstraint::ResetConstraint()

MCCOMErr MCCOMAPI ResetConstraint()

Call **ResetConstraint()** to reset the constraint after **IsTransformResetRequired()** returns True.

7.7 I3DExDataComponent

Interface ID: IID_I3DExDataComponent

Interface File: I3DRenderFeature.h

Description

I3DExDataComponent allows you to check whether a data component has an effect on an object. You can also get a list of the postrenderers that are enabled by a data component. See the chapter in the cookbook **zot** Creating Your Own Component for more information on data components.

Method Index

“I3DExDataComponent:: GetPostRenderList(TMCArray<IDType> &)”
 “I3DExDataComponent::IsActive(I3DShTreeElement*)”

I3DExDataComponent:: GetPostRenderList(TMCArray<IDType> &)

MCCOMErr MCCOMAPI GetPostRenderList(TMCArray<IDType> &idArray)

GetPostRenderList() fills the array with the class IDs of the postrenderers that are enabled by this dataComponent.

Parameters  **TMCArray<IDType> & idArray**
 <zot>need description</zot> Array containing a list of postrenderer class IDs.

Returns MC_E_NOTIMPL

I3DExDataComponent::IsActive(I3DShTreeElement*)

boolean MCCOMAPI IsActive(I3DShTreeElement* tree)

IsActive() returns false if the dataComponent has no effect on this object. By default data components are always active.

Parameters  **I3DShTreeElement* tree**
 The tree of the object.

Returns True

7.8 I3DExFinalRenderer Interface

Family ID: 'frnd'

Interface ID: IID_I3DExFinalRenderer

Interface file: I3DExFinalRenderer.h

Basic Implementation files: BasicFinalRenderer.h, BasicFinalRenderer.cpp

Description

Rendering means building a 2D image from a 3D scene. Once the image is rendered, the post render filters can be applied.

Method Index

"I3DExFinalRenderer::SetTreeTop"
 "I3DExFinalRenderer::SetCamera"
 "I3DExFinalRenderer::SetEnvironment"
 "I3DExFinalRenderer::SetAmbientLight"
 "I3DExFinalRenderer::Dehydrate"
 "I3DExFinalRenderer::GetRenderingTime"
 "I3DExFinalRenderer::SetFieldRenderingData"
 "I3DExFinalRenderer::GetRenderStatistics"
 "I3DExFinalRenderer::PrepareDraw"
 "I3DExFinalRenderer::DrawRect"
 "I3DExFinalRenderer::FinishDraw"
 "I3DExFinalRenderer::IsSMPAware"
 "I3DExFinalRenderer::GetPreferredTiling"

I3DExFinalRenderer::SetTreeTop

virtual MCOMErr MCOMAPI SetTreeTop(I3DShGroup* treeTop)

The **SetTreeTop()** function specifies which tree needs to be rendered (generally the universe).

Parameter  **I3DShGroup* treeTop**
 treeTop is the top of the hierarchy tree.

Return Value MC_S_OK

I3DExFinalRenderer::SetCamera

virtual MCOMErr MCOMAPI SetCamera(I3DShCamera* camera)

The **SetCamera()** function specifies the rendering camera.

Parameter  **I3DShCamera* camera**
 camera is the specified camera.

Return Value MC_S_OK

I3DExFinalRenderer::SetEnvironment

virtual MCCOMErr MCCOMAPI SetEnvironment(I3DShEnvironment* environment)

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

Parameter  **I3DShEnvironment*** *environment*
environment is the specified environment.

Return Value MC_S_OK

I3DExFinalRenderer::SetAmbientLight

virtual MCCOMErr MCCOMAPI SetAmbientLight(const TMCCColorRGB& ambientColor)

The **SetAmbientLight()** function sets the ambient light color.

Parameter  **TMCCColorRGB&** *ambientColor*
ambientColor is the color of the ambient light.

Return Value MC_S_OK

Comments

I3DExFinalRenderer::Dehydrate

virtual MCCOMErr MCCOMAPI Dehydrate(int16 level)

The **Dehydrate()** function specifies which buffers owned by the component could be freed in case Carrara lacked memory. **Dehydrate()** is called when the Render room is to be exited.

Parameter  **int16** *level*
level is the level of the buffers.

Return Value MC_S_OK

I3DExFinalRenderer::GetRenderingTime

virtual uint32 MCCOMAPI GetRenderingTime()

The **GetRenderingTime()** function gets data to update a display option in the Properties tray: Output Statistics panel.

Return Value 0

I3DExFinalRenderer::SetFieldRenderingData

virtual MCCOMErr MCCOMAPI SetFieldRenderingData(boolean useFieldRendering,int16 firstFrame)

The **SetFieldRenderingData()** function is used to optimized the rendering of FieldRendering for video output. This function is called once before the rendering starts. If FieldRendering is on, the rendering Module will render twice as many frames as it will output, then it will mix the even and odd lines from two frames to build one frame.

- Parameter**
- **boolean** *useFieldRendering*
useFieldRendering is true if *fieldRendering* is on.
 - **int16** *firstFrame*
firstFrame specifies the number of the first frame to be rendered.

Return Value MC_S_OK

I3DExFinalRenderer::GetRenderStatistics

virtual void MCCOMAPI GetRenderStatistics(I3DRenderStatistics renderstats)**

The **GetRenderStatistics()** function returns a pointer to the current renderStatistics. It is called several time during the rendering to read the progress of the rendering and update the rendering statistic panel in the Render room. If you did not implement any rendering statistics you can return NULL.

If you want to add render statistics, your renderer must create the renderStatistics object which implements I3DRenderStatistics (for more information on renderStatistics see I3DRenderStatistics.).

- Parameter**
- **I3DRenderStatistics**** *renderstats*
renderstats is a pointer to the renderStatistics of the renderer.

I3DExFinalRenderer::PrepareDraw

virtual MCCOMErr MCCOMAPI PrepareDraw(const TMCPoint&size, const TBBBox2D&uvBox, const TBBBox2D &productionFrame, boolean(*callback) (int16 vv, void* priv), void*priv)

The **PrepareDraw()** function is called at the beginning of the rendering of each frame. You can use it to precalculate and store some information that will be needed for each tile of the frame. For example you can get the list of the instances and some information about them.

Note: this function is called only once per frame even in SMP (multi processor mode).

If you want to use the shadow features of the lights you want to initialize them here by calling **I3DExShadowFeature::BeginRenderFrame** on each of them.

- Parameter**
- **TMCPoint&** *size*
size defines the size in pixels of the image being rendered.
 - **TBBBox2D&** *uvBox*
uvBox is the box in screen coordinates (camera uv) corresponding to the image being rendered. Note that it is not always within the Production Frame, for example, when you do a Test Render in the Assemble room.
 - **TBBBox2D&** *productionFrame*
productionFrame is the box in screen coordinates (camera uv) corresponding to the Production Frame (the image being rendered in final rendering). You can use this parameter to make sure that the size of some effects does not change when you use the Test Render tool. (For example, when you do a Test Render of the backdrop which expects screen coordinates between 0 and 1 matching the Production Frame.)
 - **boolean (* callback) (int16 vv, void* priv)**
callback is used for updating the progress bar. You need to call it when you want the progress bar

information to be updated. The second parameter, *priv*, is passed to **PrepareDraw**.

→ **void* priv**
priv is the parameter of the progress bar callback.

Return Value MC_S_OK

I3DExFinalRenderer::DrawRect

virtual MCOMErr MCOMAPI DrawRect(const TMCRect& rect, const TBBBox2D& uvBox, RTData* pixels)

The **DrawRect()** function is called for each tile of the rendered image. You must implement your rendering code here and you must output the final image in the image buffer here.

You usually want to put most of your rendering code here so that it can take advantage of multiprocessing.

Note: You need to call **IShUtilities::YieldProcesses** from time to time during the rendering if you want the user interface to be responsive while you render.

This call must be reentrant if you want your renderer to support SMP (if not return false in **IsSmpAware**). That means that you cannot use global variables in the call, and that you must protect with critical section any code that cannot be executed by two processors at the same time. If you need to build some caches shared by all tiles you can do that in **PrepareDraw** which is called only once per frame.

Parameter

→ **TMCRect& rect**
rect is the area in pixels of the tile. It is always included in the rect (0,0,size.x,size.y) where size is the size of the image passed in **PrepareDraw**.

→ **const TBBBox2D& uvBox**
uvBox of the tile (included in the *uvBox* passed to **PrepareDraw**.)

→ **RTData* pixels**
pixels is a pointer to the output buffer for the image and the GBuffers.

```
struct RTData
{
    uint16  *r,*g,*b; // pointer to the color buffers
    uint32  **buffers; // array of pointers to the GBuffers data (there are as
                      // many pointers in this array as there are GBuffers allocated)
    uint32  columnBytes; // number of bytes per pixel in the colors buffers
    uint32  rowBytes; // number of bytes to go to next line in a color buffer (currently 2)
};
```

I3DExFinalRenderer::FinishDraw

virtual MCOMErr MCOMAPI FinishDraw()

The **FinishDraw()** function is called at the end of the rendering of each frame. You can use this function to delete the temporary rendering caches.

Note, that it is called only once even in SMP.

Return Value MC_S_OK

I3DExFinalRenderer::IsSMPAware

virtual boolean MCCOMAPI IsSMPAware()

The **IsSMPAware()** function returns True if the renderer supports SMP (multiprocessors) which means that *drawRect* is reentrant, returns False otherwise.

Return Value True if SMP is supported, otherwise, False.

I3DExFinalRenderer::GetPreferredTiling

virtual MCCOMErr MCCOMAPI GetPreferredTiling(const TMCRect& area, TMCPPoint& tileSize)

The **GetPreferredTiling()** function returns the preferred tile size of your renderer. For example if your renderer has a high overhead per tile you might want to ask for bigger tiles. If you support SMP, you might want to have small tiles so that the processor time is evenly shared between the processors.

Note that this is only a preferred size which means that some tiles might be smaller or bigger. This means that you must support tiles of any size.

Parameter  **const TMCRect& area**
 area is a rectangle of the image being rendered (in pixels).

 **const TMCPPoint& tileSize**
 tileSize is preferred size of the tiles in pixels.

Return Value **MC_E_NOTIMPL**

7.9 I3DExInvertibleTweeners Interface

Interface ID: IID_I3DExInvertibleTweeners

Interface file: I3DExTweeners.h

Description

The **I3DExInvertibleTweeners** Interface defines invertible Animation Tweeners. Tweeners are 3D Components that describe the interpolation between key frames.

I3DExTweeners::Inverse()

MCCOMErr MCCOMAPI Inverse()

Inverse() is called by the 3D Shell to perform an inverse of a twener.

7.10 I3DExLightsource Interface

Family ID: 'lite'

Interface ID: IID_I3DExLightsource

Interface file: I3DExLight.h

Basic Implementation file: BasicCameraLightGel.h, BasicCameraLightGel.cpp

Description

Light sources define the lighting in a 3D scene. All kind of light sources can be designed: spot lights, bulb lights, distant lights (sun, moon), etc. Various features can be coded in a Light Source Extension, because the Light Source has complete control of the light intensity returned to the 3D Shell.

Light sources create light that illuminate object surfaces. The user can combine lights with gels to make additional lighting effects (see the **I3DExLightsourceGel** interface).

Method Index

```

"I3DExLightsource::SetTransform(TTransform3D*)"
"I3DExLightsource::GetDirection(TVector3&, TVector3&, real&)"
"I3DExLightsource::GetColor(TVector3&, TVector3&, real, TMCCColorRGB&, real&)"
"I3DExLightsource::IsVisibleInPerspective()"
"I3DExLightsource::GetPrimitiveID()"
"I3DExLightsource::ForEachShadowBuffer(ForEachShadowBufferCallback, void*)"
"I3DExLightsource::GetLightParameter(int32, void*)"
"I3DExLightsource::GetLightInfo(boolean&, boolean&, real&, TTransform3D&)"
"I3DExLightsource::GetUVfromDirection(TVector2&, const TVector3&)"
"I3DExLightsource::GetNewTrace(int16*)"
"I3DExLightsource::GetStandardLight(TStandardLight&)"

```

I3DExLightsource::SetTransform(TTransform3D*)

MCCOMErr MCCOMAPI SetTransform(TTransform3D* transform)

SetTransform is called to tell a light source its position and orientation in space. This function is used to cache transform information to speed up **GetDirection()** and **GetColor()**.

Parameters  **TTransform3D*** *transform*
Points to the transformation of the light source.

I3DExLightsource::GetDirection(TVector3&, TVector3&, real&)

MCCOMErr MCCOMAPI GetDirection(const TVector3&point, TVector3&resultDirection, real&resultDistance)

GetDirection() is called prior to **GetColor()** to perform preliminary calculations and return the distance between the surface point and light source. The surface point is the point on the object where the light is about to be calculated.

Parameters	→ TVector3& point The position of the point on the object's surface in Global Coordinates. → TVector3& resultDirection Returns the direction vector between the point and the light source in Global Coordinates. This vector is pointing from the surface point to the light source. This vector should be normalized (norm equal to 1). → real& resultDistance Returns the distance between the point and the light source.
Comment	A light bulb typically calculates <i>resultDirection</i> by subtracting the position point to the light source origin, then normalizes the results. A simulated sun light returns its light direction in <i>resultDirection</i> , and the maximum possible value in <i>resultDistance</i> .

I3DExLightsource::GetColor(TVector3&, TVector3&, real, TMCCColor- RGB&, real&)

**boolean MCCOMAPI GetColor(const TVector3&point, const TVector3&direction, const realdistance,/ TMCCColor-
RGB&result, real&ShadowIntensity)**

GetColor is called to calculate the color of light seen through a given point. The component has total control over the color to return.

Parameters	→ TVector3& point The 3D position of the point to be lit, in Global coordinates. → TVector3& direction The direction from the point to the light, in Global coordinates, as calculated by GetDirection(). → const real distance The distance from the light to the point, as calculated by GetDirection(). → TMCCColorRGB& result The resulting calculated color. (Note that the value can be more than 1.0f.) → real& ShadowIntensity Returns how clearly this light cast shadows. Between 0.0f (no shadows) and 1.0f (no light in shadow).
-------------------	--

Comment When using a spot, use the *position* parameter and the light source position and orientation to determine whether a point is in the light beam or not.
When using a sun, use the *direction* parameter to calculate the intensity.
When lighting is based on distance, (such as distance fall-off), use the *distance* parameter.

I3DExLightsource::IsVisibleInPerspective()

boolean MCCOMAPI IsVisibleInPerspective()

IsVisibleInPerspective() is called by the shell to determine if this light source can be displayed in a 3D view or not.

I3DExLightsource::GetPrimitiveID()

uint32 MCCOMAPI GetPrimitiveID()

GetPrimitiveID() is called to get the primitive ID of the light source.

I3DExLightsource::ForEachShadowBuffer(ForEachShadowBufferCallback, void*)

MCCOMErr MCCOMAPI ForEachShadowBuffer(ForEachShadowBufferCallback proc, void* priv)

ForEachShadowBuffer() is called when the shell needs to get a list of the shadow buffer planes from the light source to calculate shadows. This call is used by the finalrenderer.

- Parameters**
- **ForEachShadowBufferCallback** *proc*
The callback procedure you should call for each shadow buffer plane of your light source.
 - **void*** *priv*
Give this parameter to the callback procedure.

I3DExLightsource::GetLightParameter(int32, void*)

MCCOMErr MCCOMAPI GetLightParameter(int32 keyword, void* parameter)

GetLightParameter is called to get the light parameters from the PMap.

- Parameters**
- **int32** *keyword*
The ID of the parameter in the PMap.
 - **void*** *parameter*
Points to where to store the light parameters.

I3DExLightsource::GetLightInfo(boolean&, boolean&, real&, TTransform3D&)

MCCOMErr MCCOMAPI GetLightInfo(boolean&hasLightCone, boolean&hasLightSphere, real&halfAngle, TTransform3D&transform)

GetLightInfo() is called to get some specific information about a light.

- Parameters**
- **boolean&** *hasLightCone*
Returns True if the light has a light cone.
 - **boolean&** *hasLightSphere*
Returns True if the light has a light sphere.
 - **real&** *halfAngle*
Returns the half angle of the lighth cone if there is any.
 - **TTransform3D&** *transform*
Returns the transformation information for the light.

I3DExLightsource::GetUVfromDirection(TVector2&, const TVector3&)

boolean MCCOMAPI GetUVfromDirection(TVector2 &uv, const TVector3 &direction)

GetUVfromDirection() is called to get uv information for a gel, for the ray direction.

- Parameters**
- **TVector2&** *uv*
Returns the uv values for the given direction.
 - **const TVector3&** *direction*
Specifies the direction of the ray.

I3DExLightsource::GetNewTrace(int16*)

LightTraceElement* MCCOMAPI GetNewTrace(int16* nb)

GetNewTrace() zot needs description. (RDS 5 docs say it was unused...)

Parameters  **int16* nb**
zot needs description.

I3DExLightsource::GetStandardLight(TStandardLight&)

MCCOMErr MCCOMAPI GetStandardLight(TStandardLight& light)

GetStandardLight() retrieves a standard description of the light for use in the interactive renderer.

Parameters  **TStandardLight& light**
Returns the standard description of a light.

7.11 I3DExLightsourceGel Interface

Family ID: 'gel'

Interface ID: IID_I3DExLightsourceGel

Interface file: I3DExGel.h

Basic Implementation file: BasicCameraLightGel.h, BasicCameraLightGel.cpp

Description

The user can combine lights with gels to make additional lighting effects (see the **I3DExLightsource** interface to learn more about lights). A Gel can be thought of as a colored slide placed in front of a light source to modify the color and intensity of the light beam.

I3DExLightsourceGel::GetGelValues(const TVector2&, TMCColorRGB&)

boolean MCCOMAPI GetGelValues(const TVector2& gelScreenPosition, TMCColorRGB& result)

GetGelValues is called to change the color of a light beam.

Parameters

- **TVector2& gelScreenPosition**
2D position where the light beam crosses the gel screen. The gel screen is a 2D screen in front of the light source with boundaries from -1.0 to +1.0 on both axis.
- **TMCColorRGB& result**
Returns the factors by which the light beam color is going to be multiplied.

7.12 I3DExModifier Interface

Family ID: 'modi'

Interface ID: IID_I3DExModifier

Interface file: I3DExModifier.h

Basic Implementation file: BasicModifier.h, BasicModifier.cpp

Description

The **I3DExModifier** Interface is used to apply modifiers 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.

Methods Index

"I3DExModifier::Apply(I3DShTreeElement)"
 "I3DExModifier::GetMaxTimeWithoutComputation (I3DShTreeElement*)"
 "I3DExModifier::SetBoundingBox(const TBBBox3D&)"
 "I3DExModifier::DeformPoint(TVector3&, TVector3&)"
 "I3DExModifier::DeformVertex3D(TVertex3D&, TVector3D&)"
 "I3DExModifier::DeformFacetMesh(real, FacetMesh*, FacetMesh**)"
 "I3DExModifier::DeformBBox(const TBBBox3D&, TBox3D&)"
 "I3DExModifier::GetNeededLOD(const real, real&)"
 "I3DExModifier::GetDeformerInfo(real, TDeformInfo&)"

I3DExModifier::Apply(I3DShTreeElement)

MCCOMErr **MCCOMAPI** Apply(I3DShTreeElement* tree)

Apply gives access to all the parameters of the tree the modifier is on. To implement a behavior, just change the transforms.

Parameter  **I3DShTreeElement* tree**
The tree of the element to be modified.

I3DExModifier::GetMaxTimeWithoutComputation (I3DShTreeElement*)

MicroTick **MCCOMAPI** GetMaxTimeWithoutComputation (I3DShTreeElement* tree)

GetMaxTimeWithoutComputation tells the shell how much cached data the behavior has (starting at zero).

Parameter  **I3DShTreeElement* tree**
The tree of the element to be modified.

I3DExModifier::SetBoundingBox(const TBBBox3D&)

MCCOMErr I3DExModifier::SetBoundingBox(const TBBBox3D& bbox);

SetBoundingBox is called by the 3D Shell to tell the 3D deformation modifier in which boundaries the 3D data to deform is going to be.

Parameter  **TBBBox3D& bbox**
The 3D bounding box of the 3D data to be deformed.

I3DExModifier::DeformPoint(TVector3&, TVector3&)

MCCOMErr I3DExModifier::DeformPoint(TVector3& point, TVector3& result);

Computes result of the deformation of a given point (in local coordinates). **DeformPoint** is called by the shell if you do not implement **DeformFacetMesh** (the shell will automatically subdivide the facet mesh to achieve a smooth deformation).

Parameter  **TVector3& point**
The 3D point to be deformed, in the coordinate relative to the bounding box.
 **TVector3& result**
The 3D point after the deformation.
Return Value MC_E_NOTIMPL

I3DExModifier::DeformVertex3D(TVertex3D&, TVector3D&)

MCCOMErr MCCOMAPI DeformVertex3D(const TVertex3D& vertex, TVector3D& result);

DeformVertex3D is called by the 3D Shell and can be used to deform using a vertex.

Parameter  **TVertex3D& vertex**
The 3D vertex to be deformed, in the coordinate relative to the bounding box.
 **TVector3* result**
The 3D vertex after the deformation.
Return Value MC_E_NOTIMPL

I3DExModifier::DeformFacetMesh(real, FacetMesh*, FacetMesh**)

MCCOMErr I3DExModifier::DeformFacetMesh(real lod, FacetMesh* in, FacetMesh outMesh);**

DeformFacetMesh is called to deform or create 3D facets. Given the original facet mesh of a primitive, it produces a (new) deformed mesh. You can modify whatever you want on the mesh: you can add or remove vertices, normals, facets, or change their position.

But you should NOT modify the *in* mesh. You need to create a new mesh and return it in *outMesh* (yet you can clone *in* to *outMesh* and then change *outMesh*).

If the parameters of your deformer won't deform the mesh, you can have *outMesh* reference *in* thus:

```
TMCCountedGetHelper<FacetMesh> result(outMesh); result= in;
```

Parameters  **real lod**
The level of detail.
 **FacetMesh* in**
A pointer to the input representation of triangular polygon data using corresponding arrays of data.
 **FacetMesh** outMesh**
The output mesh after the deformation occurs.
Returns MC_E_NOTIMPL

I3DExModifier::DeformBBox(const TBBBox3D&, TBox3D&)

MCCOMErr MCCOMAPI DeformBBox(const TBBBox3D& in, TBox3D& out)

DeformBBox is called to get an approximation of how the modifier affects a given zone of space con-

	tained in the box <i>in</i> , and returns the box that encompass the deformed space in <i>out</i> .
Parameters	 TBBox& in This is the reference to the input bounding box.  TBBox& out This is the reference to the output bounding box.
Returns	MC_S_OK

I3DExModifier::GetNeededLOD(const real, real&)

MCCOMErr **MCCOMAPI** GetNeededLOD(const real outputLOD, real& neededLod)

GetNeededLOD is used by the shell to know what level of detail the modifier needs to produce a result at the precision *outputLOD*. The default implementation does:
neededLOD= outputLOD;

Parameters	 const real outputLOD This is the output level of detail.  real& neededLod This is the reference to the needed level of detail.
Returns	MC_S_OK

I3DExModifier::GetDeformerInfo(real, TDeformInfo&)

MCCOMErr **MCCOMAPI** GetDeformerInfo(real lod, TDeformInfo& deformInfo)

GetDeformerInfo is called to get information about the modifier. If you implement only **DeformPoint** or **DeformVertex3D** and not **DeformFacetMesh**, use this function to give the shell some directions to guide the facet mesh subdivision process (see I3dExModifier.h).

Parameters	 real lod This is the level of detail.  TDeformInfo& deformInfo This is the reference deformation information.
Returns	MC_S_OK

7.13 I3DExModule Interface

Family ID: 'modu'

Interface ID: IID_I3DExModule

Interface file: I3DExModule.h

Basic Implementation file: BasicModule.h, BasicModule.cpp

Description

The **I3DExModule** Interface is used to create module of any kind, and to associate as many views as you want. The cookbook example <zot xref later>shows how to create a modeler. A module has Preferences, which can be set by the user through the File menu> Preferences command just like any other module. A module also has its own tool set and menu bar.

There are two entry points to a module. First, a module can edit data of some known type. Second, if the type of data is not known, the module may import the data into a familiar type. Refer to **I3DExModule::Edit()** for details.

Resources:

In addition to the usual 'Comp' and 'PMap' resources, you need to provide a 'Modu' resource of the same ID to specify parameters such as the 'MBAR' (Menu Bar) ID, the 'TBAR' (Tool Bar) ID, etc. See ExternalAPI.r.

Preferences:

The Shell can read and write your Module Preferences data automatically both from and to the Preferences file. You can give this data to your Module using the **I3DExModule::SetPrefs()** call just before calling **I3DExModule::Edit()**.

1. In the 'Modu' resource, choose a Preferences View ID and a 'Prfs' ID (the 'Prfs' resource contains the default values if a Preferences file has to be created from scratch by the Shell. To make a 'Prfs' resource, copy/paste the part of the Preferences File related to your Module into the 'Prfs' resource.
2. Add to the Module's 'PMap'. You will note that the same PMap is used for the Module's View and its Preferences. This is not a real problem as you use distinct IDs for the UI elements (buttons, check boxes, etc.). If you do not wish to have Preferences, put 0 in the Preferences View ID of the 'Modu' resource.

Drag and drop from Tool Bar: **zot** this was in the header file, does someone want to mention something about it here? If not, I'll remove it.

Actions:

zot The RDS text discusses actions and refers to a group of interfaces that are no longer used (IShActionManager, IShAction, and IShMouseAction). What should be said about how to post undoable actions to enter the undo/redo process.

Methods Index

"I3DExModule::Initialize(IMCUnknown*)"

"I3DExModule::Destroy()"

"I3DExModule::Hydrate()"

```

“I3DExModule::Dehydrate()”
“I3DExModule::Activate()”
“I3DExModule::Deactivate()”
“I3DExModule::Import(IMCUnknown*, void*)”
“I3DExModule::BuildMenuBar()”
“I3DExModule::GetPreferredRoom(IDType&)”
“I3DExModule::WantsToBeInCurrentRoom()”
“I3DExModule::SubModuleClosedMainWindow(I3DShModule*)”
“I3DExModule::SubModuleAboutToBeDestroyed(I3DShModule*)”
“I3DExModule::SubModuleAboutToBeHydrated(I3DShModule*)”
“I3DExModule::AboutToCloseMainWindow()”
“I3DExModule::SavesInNativeFormat()”

```

I3DExModule::Initialize(IMCUnknown*)

MCCOMErr MCCOMAPI Initialize(IMCUnknown* inElement)

Initialize() is called during creation of the module to initialize UI elements, and set the editing element. inElement is the element that the module edits. Override this function to create UI elements such as windows and initialize any data for your module.

Use QueryInterface on the inElement object to get the specific interface you need. Scene Document Modules will get an I3DShGroup object. Modelers get two calls to Initialize. First, they get passed the universe group as all Scene Document Modules do. Next, they get passed the master object in the scene selection which they are to edit. In addition, if the user leaves the modeling room and re-enters with a different object selected, **Initialize()** is called again with the new object to edit.

Global Modules do not get any object, they have a NULL inElement.

I3DExModule::Destroy()

MCCOMErr MCCOMAPI Destroy()

Destroy() is called once when the module is to be destroyed (e.g., Scene Document Modules are called when the document closes, Global Modules are called when the application is quit). Override this function to destroy UI elements and cleanup any data. However, you need destroy your slave modules, since you are in charge of them.

I3DExModule::Hydrate()

MCCOMErr MCCOMAPI Hydrate()

Hydrate() is called when the module is to be hydrated; that is, the room that it occupies is entered.

If this module is a main module (i.e., it defines a room), override this function to show any main windows and call **Hydrate()** on slave modules.

If this module is a slave module, override this function to prepare to be activated and call **Hydrate()** on any subslave modules. This may involve showing windows, but should not install parts in shared windows since **Hydrate()** does not mean the module is active (see **Activate()** below). Also note that your window may have been closed by someone else (in this case you had been told through **AboutToCloseMainWindow()**), so you will have to recreate your window.

I3DExModule::Dehydrate()

MCCOMErr MCCOMAPI Dehydrate()

Dehydrate() is called when the module is to be dehydrated; that is, the room that it occupies is exited. If this module is a main module (i.e., it defines a room), override this function to hide any main windows and call **Dehydrate()** on slave modules.

If this module is a slave module, override this function to destroy large caches and parts and call **Dehydrate()** on any subslave modules.

NOTE: **Deactive()** is called before **Dehydrate()** on an active module.

I3DExModule::Activate()

MCCOMErr MCCOMAPI Activate()

Activate() is called when the module is to be activated. If this module is a main module (i.e., it defines a room), one of its main windows has been activated; override this function to call **Activate()** on slave modules. If this module is a slave module, override this function to activate and call **Activate()** on any subslave modules.

This may involve showing windows, and may install parts in shared windows.

I3DExModule::Deactivate()

MCCOMErr MCCOMAPI Deactivate()

Deactivate() is called when the module is to be deactivated. If this module is a main module (i.e., it defines a room), all of its main windows has been deactivated; override this function to call **Deactivate()** on slave modules. If this module is a slave module, override this function to deactivate and call **Deactivate()** on any subslave modules.

This may involve hiding windows, and may uninstall parts out of shared windows.

I3DExModule::Import(IMCUnknown*, void*)

MCCOMErr MCCOMAPI Import(IMCUnknown* inElement, void* inImportData)

Import() is called when the module is asked to import some foreign data. For example, a modeler could be asked to import the facets of another primitive.

Parameters

✎ **IMCUnknown*** *inElement*

It is up to the module to do something or not with *inElement*. The module should rely on **inElement->QueryInterface** to decide if it can do anything with it or not. In any case, *inElement* should be left intact.

✎ **void*** *inImportData*

inImportData points to some data block depending on the type import. For example, for modelers, *inImportData* will be a **real32*** containing the Fidelity.

I3DExModule::BuildMenuBar()

MCCOMErr MCCOMAPI BuildMenuBar()

BuildMenuBar() gives you opportunity to add menus dynamically (instead of using a MBAR resource).

Use *gShMenuUtilities* to add menu items.

I3DExModule::GetPreferredRoom(IDType&)

MCCOMErr MCCOMAPI GetPreferredRoom(IDType& roomId)

GetPreferredRoom() is called to determine in which room the module wants to be put in, when the module does not define a room (eg the rendering module).

By default the module will be added in the current room.

Parameters  **IDType& roomId**
The ID of the preferred room.

I3DExModule::WantsToBeInCurrentRoom()

boolean MCCOMAPI WantsToBeInCurrentRoom()

WantsToBeInCurrentRoom() is called to determine if the module can be inserted in the current room.

I3DExModule::SubModuleClosedMainWindow(I3DShModule*)

boolean MCCOMAPI SubModuleClosedMainWindow(I3DShModule* subModule)

SubModuleClosedMainWindow() will be called by any sub module while its Main Window is being closed, so you can react and tell it that it is OK to commit suicide (return true).

Note: You tell a sub module that you are its "master" by calling **I3DShModule::SetMasterModule()**.

Parameters  **I3DShModule* subModule**
A pointer to the sub module that is having its Main Window closed.

I3DExModule::SubModuleAboutToBeDestroyed(I3DShModule*)

MCCOMErr MCCOMAPI SubModuleAboutToBeDestroyed(I3DShModule* subModule)

SubModuleAboutToBeDestroyed() will be called by any sub module that is about to be destroyed, so you can react.

Note: You tell a sub module that you are its "master" by calling **I3DShModule::SetMasterModule()**.

Parameters  **I3DShModule* subModule**
A pointer to the sub module that is about to be destroyed.

I3DExModule:: SubModuleAboutToBeHydrated(I3DShModule*)

MCCOMErr MCCOMAPI SubModuleAboutToBeHydrated(I3DShModule* subModule)

SubModuleAboutToBeHydrated() will be called by any sub module that is about to be hydrated, so you can react.

Note: You tell a sub module that you are its "master" by calling **I3DShModule::SetMasterModule()**.

Parameters **I3DShModule*** *subModule*

A pointer to the sub module that is about to be hydrated.

I3DExModule::AboutToCloseMainWindow()

MCCOMErr MCCOMAPI AboutToCloseMainWindow()

AboutToCloseMainWindow() is called when your main window is about to be closed by someone else.

It is very important to release any references that your module may have on your window. You do not need to close it, since this "someone" is going to do it.

I3DExModule::SavesInNativeFormat()

booleanMCCOMAPI SavesInNativeFormat()

SavesInNativeFormat() is called to determine whether the module wants to save data in the nativefile format.

7.14 I3DExPostRenderer Interface

Family ID: 'post'

Interface ID: IID_I3DExPostRenderer

Interface file: I3DExPostRenderer.h

Basic Implementation File: BasicPostRenderer.h, BasicPostRenderer.cpp

Description

The **I3DExPostRenderer** Interface is used to apply additional effects (filters) to the image after the rendering.

Method Index

"I3DExPostRenderer::GetBufferNeeds(uint32&, void*)"
 "I3DExPostRenderer::PrepareDraw(ISHRasterLayer*, const TMCRect&, ISHChannel*, I3DShScene*, I3DShCamera*, TBBBox2D&, TBBBox2D&)"
 "I3DExPostRenderer::DrawRect(const TMCRect&, const TChannelDataBucket*, TBBBox2D&)"
 "I3DExPostRenderer::FinishDraw()"
 "I3DExPostRenderer::BeginRender(MicroTick, MicroTick, uint32, uint32, boolean)"
 "I3DExPostRenderer::EndRender()"

I3DExPostRenderer::GetBufferNeeds(uint32&, void*)

void GetBufferNeeds(uint32& needs, void* renderer)

GetBufferNeeds gets information needed when you initialize the postrenderer. This call is called before the rendering starts.

Parameter

 **uint32& needs**

Returns a set of flags to determine which GBuffers should be filled during the rendering. The following constants are defined to help you (they define the bit number to be set so to get the buffer of the NormalX and the distance you should return $2^{\text{k32Distance}} + 2^{\text{k32NormalX}}$):

Name	Value	Description
k32Distance	0	Distance from the camera to the object visible at a point in the image.
k32PositionX	1	X position of a point if not the background
k32PositionY	2	Y position of a point if not the background
k32PositionZ	3	Z position of a point if not the background
k32NormalX	4	X direction of an object's surface Normal if not the background
k32NormalY	5	Y direction of an object's surface Normal if not the background
k32NormalZ	6	Z direction of an object's surface Normal if not the background

Name	Value	Description
k32Alpha	7	The alpha channel with black (0) as the background
k32Index	8	UV space index
k32SurfaceU	9	U value of the UV space of an object at a point
k32SurfaceV	10	V value of the UV space of an object at a point
k32DiffuseColor	11	Corresponds to the Color channel of the object's shader
k32SpecularColor	12	Corresponds to the Highlight channel of the object's shader
k32SpecularSizeValue	13	Corresponds to the Shininess channel of the object's shader
k32AmbientValue	14	
k32LambertValue	15	
k32ReflectionColor	16	Corresponds to the Reflection channel of the object's shader
k32TransparencyColor	17	Corresponds to the Transparency channel of the object's shader
k32RefractionValue	18	Corresponds to the Refraction channel of the object's shader
k32GlowColor	19	Corresponds to the Glow channel of the object's shader

❏ **void* renderer**

This element is not available yet.

I3DExPostRenderer::PrepareDraw(IShRasterLayer*, const TMCRect&, IShChannel*, I3DShScene*, I3DShCamera*, TBBBox2D&, TBBBox2D&)

MCCOMErr PrepareDraw(IShRasterLayer* input, const TMCRect& bounds, IShChannel* buffers[], I3DShScene* scene, I3DShCamera* renderingCamera, TBBBox2D& uvBox, TBBBox2D& productionFrame)

PrepareDraw() is called prior to the rendering. You should store most of the info provided.

Parameters

❏ **IShRasterLayer* input**

Is a pointer to the buffer of the input image. (The input image is composed of 3 planes of 16 bits each for the 3 colors.)

❏ **const TMCRect& bounds**

Is a rectangle corresponding to the image.

❏ **IShChannel* buffers[]**

This is a table of pointers to the buffers requested by **GetBufferNeeds()**. Each element in the buffers array corresponds to the value of each buffer (buffer[0] for distance through buffer[19] for glow color). These pointers are null when the buffer is not allocated.

See the table in “I3DExPostRenderer::GetBufferNeeds(uint32&, void*)” on page 117 for the complete list.

→ **I3DShScene*** *scene*

This is a pointer to the scene structure.

→ **I3DShCamera*** *renderingCamera*

This is a pointer to the rendering camera.

→ **TBBBox2D&** *uvBox*

uvBox of the image being rendered in screen coordinates.

→ **TBBBox2D&** *Production Frame*

uv Box of the Production Frame in screen coordinates (this might be different from the uvBox when doing marquee rendering in the interactive preview).

Return Value MC_S_OK

Comments You need to use the Production Frame and uvBox to make sure Area Rendering in the interactive preview will work.

I3DExPostRenderer::DrawRect(const TMCRect&, const TChannelDataBucket*, TBBBox2D&)

MCCOMErr DrawRect(const TMCRect& outputRect, const TChannelDataBucket* output[], TBBBox2D &uvBox)

DrawRect() is called for each tile of the image.

Parameters → **TMCRect&** *outputRect*

Coordinates for the tile to be rendered in pixels. Should be inside the bounds rectangle given in **PrepareDraw()**. It matches an actual tile in the input and output buffers.

→ **TChannelDataBucket*** *output[]*

PixelBucket corresponding to the tile to be rendered in the output image. You can be sure it is an actual tile of the output buffer.

I3DExPostRenderer::FinishDraw()

MCCOMErr FinishDraw()

FinishDraw() is called for cleanup at the end of the post-rendering.

I3DExPostRenderer::BeginRender(MicroTick, MicroTick, uint32, uint32, boolean)

MCCOMAPI BeginRender(MicroTick beginTime, MicroTick endTime, uint32 framePerSeconds, int32 nbrFrames, boolean isRenderMovie)

BeginRender() is called for set up at the beginning of the post-rendering.

I3DExPostRenderer::EndRender()

MCCOMAPI EndRender()

EndRender() is called for cleanup at the end of the post-rendering.

7.15 I3DExGeometricPrimitive Interface

Family ID: 'prim'

Interface ID: IID_I3DExGeometricPrimitive

Interface file: I3DExPrimitive.h

Basic Implementation file: BasicPrimitive.h, BasicPrimitive.cpp

Description

Geometric primitives define a geometric object. They return a list of facets or bicubic patches and can give information on an object's UV Space. The shell will check for patches first by calling **EnumPatches()** and then call **GetFacetMesh()** if **EnumPatches()** fails. At least one or both of these two methods must be implemented.

In addition to describing geometry, a geometric primitive can define a custom ray-tracing routine rather than relying only on facets or patches. This is done by inheriting from the "**IRaytracablePrimitive Interface**" in addition to **I3DExGeometricPrimitive** (or TBasicPrimitive).

Rather than defining geometry, a primitive can also describe a volume. This is achieved by inheriting from the "**IVolumePrimitive Interface**" as well as **I3DExGeometricPrimitive** (or TBasicPrimitive).

Method Index

```
"I3DExGeometricPrimitive::GetBoundingBox(TBox3D*)"
"I3DExGeometricPrimitive::EnumPatches(EnumPatchesCallback, void*, boolean&)"
"I3DExGeometricPrimitive::GetNbrLOD(int16&)"
"I3DExGeometricPrimitive::GetLOD(int16, real&)"
"I3DExGeometricPrimitive::GetFacetMesh(uint32, FacetMesh**)"
"I3DExGeometricPrimitive::GetFacetMesh(real, FacetMesh**)"
"I3DExGeometricPrimitive::CanBeSplit()"
"I3DExGeometricPrimitive::SplitPrimitive(TMCCountedPtrArray<I3DExGeometricPrimitive>&, TMCAr-
ray<TTransform3d>&)"
"I3DExGeometricPrimitive::GetUVSpaceCount()"
"I3DExGeometricPrimitive::GetUVSpace(uint32, UVSpaceInfo*)"
"I3DExGeometricPrimitive::UV2XYZ(TVector2*, uint32, TVector3*, boolean*)"
"I3DExGeometricPrimitive::GetUVSpaceRDS5(uint32, UVSpaceInfoRDS5*)"
"I3DExGeometricPrimitive::AppendToRenderables(const TTransform3D&, TRenderableAndTfmArray&)"
"I3DExGeometricPrimitive::AutoSwitchToModeler()"
"I3DExGeometricPrimitive::IsBBoxPickable()"
```

I3DExGeometricPrimitive::GetBoundingBox(TBox3D*)

MCCOMErr I3DExGeometricPrimitive::GetBoundingBox(TBox3D* bbox)

GetBoundingBox() is called to pass the bounding box of the primitive to the 3D Shell. This call is optional. If this call is not implemented the bounding box will be calculated automatically from the patches or facets list.

Parameter

↳ **TBox3D* bbox**

The 3D bounding box of the object.

Return Value

MC_S_OK	
MC_E_NOTIMPL	Call not implemented.

I3DExGeometricPrimitive::EnumPatches(EnumPatchesCallback, void*, boolean&)

MCCOMErr MCCOMAPI EnumPatches(EnumPatchesCallback callback, void* privData, boolean& closed)

I3DExGeometricPrimitive::EnumPatches is called by the 3D Shell to get the primitive as a list of bicubic patches. Try not to use it. Use **GetFacetMesh()** instead.

Parameters

➦ **EnumPatchesCallback** *callback*

The callback procedure should be called for each Patch generated by the Primitive. Its interface is as follows:

```
typedef void (*EnumPatchesCallback) (TPatch3D* patch, void* privData);
```

The 3D Shell will not keep the *patch* pointer. It will copy its content.

➦ **void* privData**

Points to private data from the Shell. Pass this pointer to *callback*.

➦ **boolean& closed**

Returns True when the surface is closed. This helps set the backface mask correctly.

Return

MC_E_NOTIMPL;

I3DExGeometricPrimitive::GetNbrLOD(int16&)

MCCOMErr GetNbrLOD(int16& nbrLod)

GetNbrLOD() is called to get the number of levels of detail of the primitive. This value is used as the *lodIndex* in **GetLOD()**.

Parameters

➦ **int16& nbrLod**

This is the number of levels of detail. A level of detail is the threshold used to display your primitive.

Return value

MC_S_OK	
MC_E_NOTIMPL	Function is not implemented.

I3DExGeometricPrimitive::GetLOD(int16, real&)

MCCOMErr GetLOD(int16 lodIndex, real& lod)

GetLOD() is called to get the level of detail of the primitive. Refer to the Cookbook chapter on creating a primitive for a detailed discussion on level of detail.

Parameters

➦ **int16 lodIndex**

The level of detail index.

⇨ **real& lod**
The level of detail.

Return value

MC_S_OK	
MC_E_NOTIMPL	Function is not implemented.

I3DExGeometricPrimitive::GetFacetMesh(uint32, FacetMesh)**

MCCOMErr MCCOMAPI GetFacetMesh(uint32 lodIndex, FacetMesh outMesh)**

GetFacetMesh() is an alternate method for creating a facet mesh. It is more efficient than using **EnumFacets()** as it allows for calculating less facets in conditions that require less detail. Implement this version.

Parameters

⇨ **uint32 index**
The index of the level of detail.

⇨ **FacetMesh** outMesh**
A pointer to the output facet mesh.

Return value A FacetMesh* defining the geometric primitive. Refer to “FacetMesh” on page 64 to learn more.

I3DExGeometricPrimitive::GetFacetMesh(real, FacetMesh)**

MCCOMErr MCCOMAPI GetFacetMesh(real lod, FacetMesh outMesh)**

Never implement this version. It is reserved for future use (with primitives with an infinite number of resolutions.)

Parameters

⇨ **real lod**
The level of detail.

⇨ **FacetMesh** outMesh**
A pointer to the output facet mesh.

Return value A FacetMesh* defining the geometric primitive. Refer to “FacetMesh” on page 64 to learn more.

I3DExGeometricPrimitive::CanBeSplit()

boolean CanBeSplit()

CanBeSplit() returns true if the primitive can split itself into sub parts of the same type.

Return value

True	SplitPrimitive() will be called.
False	SplitPrimitive() will not be called.

I3DExGeometricPrimitive::SplitPrimitive(TMCCountedPtrArray<I3DExGeometricPrimitive>&, TMCArray<TTransform3d>&)

MCCOMErr SplitPrimitive(TMCCountedPtrArray<I3DExGeometricPrimitive>& subParts, TMCArray<TTransform3d>& subPartPositions)

SplitPrimitive() builds a list of sub-parts of itself. Both arrays need to be of same size. The original primitive remains intact.

Parameters

→ **TMCCountedPtrArray<I3DExGeometricPrimitive>& subParts**
The split parts of the primitive.<zot>Could use better description here</zot>

→ **TMCArray<TTransform3d>& subPartPositions**
subPartPosition gives the relative position of each sub part in the object's local coordinates.

I3DExGeometricPrimitive::GetUVSpaceCount()

uint32 GetUVSpaceCount()

GetUVSpaceCount() is called to get the number of UV Spaces as defined in “UVSpaceInfo” on page 62.

Return value Number of UV Spaces for this object. If it is not possible to do direct parametric mapping and there are no UV values then return 0.

I3DExGeometricPrimitive::GetUVSpace(uint32, UVSpaceInfo*)

MCCOMErr GetUVSpace(uint32 uvSpaceID, UVSpaceInfo* uvSpaceInfo)

GetUVSpace() is called to define the UV Space(s) of the primitive.

Parameters

→ **uint32 uvSpaceID**
The UV Space ID on which the Component needs to return information, indexed from 0 to **GetUVSpaceCount()-1**.

→ **UVSpaceInfo* uvSpaceInfo**

The UVSpace info pointer. Refer to “UVSpaceInfo” on page 62 to learn more.

Return Value

MC_S_OK	
MC_E_NOTIMPL	Call not implemented.

I3DExGeometricPrimitive::UV2XYZ(TVector2*, uint32, TVector3*, boolean*)

MCCOMErr UV2XYZ(TVector2* uv, uint32 uvSpaceID, TVector3* resultPosition, boolean* inUVSpace)

UV2XYZ() is called when a point on the surface defined only by its uv coordinates needs to be converted in x, y and z values. This call is optional. If this call is not implement then the default rules will apply by using the patches or the facets of the Primitive. When possible implement

I3DExGeometricPrimitive::UV2XYZ for optimal performance and accuracy.

Parameters **TVector2*** *uv*

The given uv values.

 **uint32** *uvSpaceID*

The identifier of the UV space.

 **TVector3*** *resultPosition*

The resulting 3D coordinates in the object Local Coordinates System.

 **boolean*** *inUVSpace*

Return TRUE if the uv values are within the UV Space boundaries.

Return Value

MC_S_OK	
MC_E_NOTIMPL	Call not implemented.

I3DExGeometricPrimitive::GetUVSpaceRDS5(uint32, UVSpaceInfoRDS5*)

MCCOMErr GetUVSpaceRDS5(uint32 *uvSpaceID*, UVSpaceInfoRDS5* *uvSpaceInfoRDS5*)

<zot>need full description</zot>

Parameters **uint32** *uvSpaceID*

The identifier of the UV space

 **UVSpaceInfoRDS5 *** *uvSpaceInfoRDS5*

zot need description here.

Return Value**MC_E_NOTIMPL**

I3DExGeometricPrimitive::AppendToRenderables(const TTransform3D&, TRenderableAndTfmArray&)

MCCOMErr AppendToRenderables(const TTransform3D& *worldFromModelTfm*, TRenderableAndTfmArray& *renderableAndTfm*)

<zot>need full description</zot> Retrieves the renderables

Parameters **TTransform3D&** *worldFromModelTfm*

zot need description here

 **TRenderableAndTfmArray&** *renderableAndTfm*

zot need description here.

I3DExGeometricPrimitive::AutoSwitchToModeler()

virtual boolean AutoSwitchToModeler()**AutoSwitchToModeler()** automatically switches to the modeler after the primitive is inserted.**Return value**

True	True if the primitive should be open in its modeler immediately after its insertion
False	False if the primitive should not automatically open its modeler.

I3DExGeometricPrimitive::IsBBoxPickable()

booleanMCCOMAPI IsBBoxPickable()

Return true if the primitive can be picked in the 3D View when clicking on its bounding box. Usually, you want to reply no, since the picking is done thru a ray-tracing hit test. However, some primitives such as volumic primitives or particles systems may want to answer yes.

Default implementation in TBasicPrimitive returns false.

7.16 I3DExReflectionFeature

Interface ID: IID_I3DExReflectionFeature

Interface File: I3DRenderFeature.h

Description

I3DExReflectionFeature allows you to identify the reflection needs and calculate the reflections.

Method Index

“I3DExReflectionFeature::GetReflectionNeeds(uint32 &)”

“I3DExReflectionFeature::CalcReflection(RenderFeatureIO&)”

I3DExReflectionFeature::GetReflectionNeeds(uint32 &)

MCCOMErr MCCOMAPI GetReflectionNeedFlags(uint32& needFlags)

GetReflectionNeeds() counts the number of needed reflections based on *needFlags*.

Parameters  **uint32 & needFlags**
 <zot>need description</zot> .

I3DExReflectionFeature::CalcReflection(RenderFeatureIO&)

MCCOMErr MCCOMAPI CalcReflection(RenderFeatureIO& theFIOParms)

CalcReflection() calculates the reflections.

Parameters  **RenderFeatureIO& theFIOParms**
 <zot>need description</zot>.

7.17 I3DExRefractionFeature

Interface ID: IID_I3DExRefractionFeature

Interface File: I3DRenderFeature.h

Description

I3DexRefractionFeature allows you to identify the refraction needs and calculate the refractions.

Method Index

“I3DExRefractionFeature::GetRefractionNeeds(uint32 &)”

“I3DExRefractionFeature::CalcRefraction(RenderFeatureIO&)”

I3DExRefractionFeature::GetRefractionNeeds(uint32 &)

MCCOMErr MCCOMAPI GetRefractionNeedFlags(uint32& needFlags)

GetRefractionNeeds() counts the number of needed refractions based on *needFlags*.

Parameters ↗ **uint32 & needFlags**
 <zot>need description</zot> .

I3DExRefractionFeature::CalcRefraction(RenderFeatureIO&

MCCOMErr MCCOMAPI CalcRefraction(RenderFeatureIO& theFIOParms)

CalcRefraction() calculates the refractions.

Parameters **RenderFeatureIO** & *theFIOParms*
<zot>need description</zot>.

7.18 I3DExShader Interface

Family ID: 'shdr'

Interface ID: IID_I3DExShader

Interface file: I3DExShader.h

Basic Implementation file: BasicShader.h, BasicShader.cpp

Description

Shaders are used to give photorealistic appearance to an object's surface. They 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.

Shaders are designed to be re-entrant such that the output from one shader can be used as the input for another shader. This is accomplished by implementing methods to return a value, color, or complete shading structure and then using **GetShadingFlags()** and **GetImplementedOutput()** to define which implementation was used. Methods that return simply a value, color, or vector use the "*ShadingIn*" struct, filling in the fields as defined by **GetShadingFlags()** independent of the channel in which the shader is placed. A shader that implements an entire shading structure uses a "*ShadingOut*" structure that sets values for a ShadingIn structure for each channel of the shader.

Besides implementing methods to define a value for each point, which uses a *ShadingIn* or *ShadingOut*, you can also implement an additional version of each function that uses a *ShadedArea*. A *ShadedArea* is defined by four *ShadingIn* points. If the *ShadedArea* version of the function is implemented, the *ShadingIn* version must also be implemented.

By using a *ShadedArea* you can find the average of a pixel. In implementing this method you can control the sampling level and greatly increase the speed at which your shader is rendered. This is particularly easy to implement for shaders which produce the same or similar result over an extended area.

Method Index

```
"I3DExShader::IsEqualTo(I3DExShader*)"
"I3DExShader::GetShadingFlags(ShadingFlags&)"
"I3DExShader::GetImplementedOutput()"
"I3DExShader::DoShade(ShadingOut&, ShadingIn&)"
"I3DExShader::GetValue(real&, boolean, ShadingIn&)"
"I3DExShader::GetColor(TMCCColorRGB&, boolean, ShadingIn&)"
"I3DExShader::GetVector(TVector3&, ShadingIn&)"
"I3DExShader::ConvertToPoly(TMCArrary<TVector2>*)"
"I3DExShader::OutlineInPreviewMode()"
"I3DExShader::Contains(ShadingIn&, TVector2&)"
```

I3DExShader::IsEqualTo(I3DExShader*)

boolean IsEqualTo(I3DExShader* aShader)

IsEqualTo() is used to compare the values of two shaders. **IsEqualTo()** must first get a pointer on the same interface as itself before comparing the values.

Parameter  **I3DExShader*** *aShader*
 Pointer to the comparison shader.

Return Value	Value	Meaning
	TRUE	Shaders have identical values
	FALSE	Shaders are different

I3DExShader::GetShadingFlags(ShadingFlags&)

MCCOMErr GetShadingFlags(ShadingFlags& theFlags)

GetShadingFlags() specifies which parameters the shader will need to perform shading calculations. If the shader is a *constant shader* which values does not depend on any of the *ShadingIn* parameters then it should set the *fCallOnce* flag in *theFlags* to true. See the description of the “*ShadingFlags*” structure for more details.

Parameter  **ShadingFlags&** *theFlags*
 Points to a *ShadingFlags* record containing all the flags.

Return Value MC_S_OK

I3DExShader::GetImplementedOutput()

uint32 GetImplementedOutput()

GetPreferredOutput() is called by the 3D Shell to determine which methods the Shader actually implements. Return only one of the following values if you implement only one of the methods. If more than one is implemented, such as the *ShadedArea* implementation, then return the result as the OR of the two values (i.e. return `kUsesGetValueArea | kUsesGetValue`).

Return Value

Shading Defines

Constant	Value	Meaning
kUsesGetValue	1	I3DExShader::GetValue is implemented
kUsesGetColor	2	I3DExShader::GetColor is implemented
kUsesGetVector	4	I3DExShader::GetVector is implemented
kUsesDoShade	16	I3DExShader::DoShade is implemented
kUsesGetValueArea	32	I3DExShader::GetValue with <i>ShadedArea</i> is implemented
kUsesGetColorArea	64	I3DExShader::GetColor with <i>ShadedArea</i> is implemented
kUsesGetVectorArea	128	I3DExShader::GetVector with <i>ShadedArea</i> is implemented
kUsesDoShadeArea	256	I3DExShader::DoShade with <i>ShadedArea</i> is implemented

I3DExShader::DoShade(ShadingOut&, ShadingIn&)

MCCOMErr DoShade(ShadingOut& result, ShadingIn& shadingIn)

DoShade() has total control of the input and output parameters. See the description of “*ShadingOut*” and “*ShadingIn*” for more details.

Parameters

➞ **ShadingOut& result**

Reference to the complete input/output shading data structure.

➞ **ShadingIn& shadingIn**

The Shading element associated with the shader.

Return Value

MC_S_OK	DoShade() is implemented by this shader.
MC_E_NOTIMPL	DoShade() is not implemented by this shader.

Comments

I3DExShader::GetValue(real&, boolean, ShadingIn&)

MCCOMErr GetValue(real& result, boolean fullArea, ShadingIn& shadingIn)

GetValue() returns a value between 0.0 and 1.0. Typical examples are noise functions, fractal functions, gradients, etc. See the description of “*ShadingIn*” for more details.

Parameters

➞ **real& result**

Returns a result between 0.0 and 1.0.

➞ **boolean fullArea**

<zot> needs description </end zot>

➞ **ShadingIn& shadingIn**

Reference to the input shading data structure. The shader uses the parameters requested by “*I3DExShader::GetShadingFlags(ShadingFlags&)*”.

Return Value

MC_S_OK	GetValue() is implemented by this shader.
MC_E_NOTIMPL	GetValue() is not implemented by this shader.

I3DExShader::GetColor(TMCCColorRGB&, boolean, ShadingIn&)

MCCOMErr GetColor(TMCCColorRGB& result, boolean fullArea, ShadingIn& shadingIn)

GetColor() returns a color. Typical examples are constant colors or texture maps. See the description of “*ShadingIn*” for more details.

Parameters

➞ **TMCCColorRGB & result**

Return the color values.

➡ **boolean *fullArea***
 <zot> needs description </end zot>

➡ **ShadingIn & *in***
 Points to the input shading data structure. The shader uses the parameters requested by “I3DExShader::GetShadingFlags(ShadingFlags&)”.

Return Value

MC_S_OK	GetColor() is implemented by this shader.
MC_E_NOTIMPL	GetColor() is not implemented by this shader.

Comments

I3DExShader::GetVector(TVector3&, ShadingIn&)

MCCOMErr GetVector(TVector3& result, ShadingIn& shadingIn)

GetVector is reserved for a future use.

I3DExShader::ConvertToPoly(TMCArray<TVector2>*)

void ConvertToPoly(TMCArray<TVector2>* vertices)

ConvertToPoly() must be implemented to allow a shader to display an outline. This is typically used by layers such as 'rectangle', 'ellipse', and 'polygon'.

Parameters ➡ **TMCArray<TVector2>* vertices**
 An array of 2D vertices used to draw a layout around the layer. Those vertices' coordinates are in the UVSpace of the layer.

I3DExShader::OutlineInPreviewMode()

boolean OutlineInPreviewMode()

Return Value Return True if the shader will draw an outline. **ConvertToPoly()** must then be implemented.

I3DExShader::Contains(ShadingIn&, TVector2&)

boolean Contains(ShadingIn& shadingIn, TVector2& localUV)

Contains() is used for selecting the layer in the Preview window of the Texture Room. It must return true if the layer is selected and false otherwise.

Parameters ➡ **ShadingIn& shadingIn**
 Reference to the *ShadingIn* structure.

➡ **TVector2& localUV**
localUV returns the uv coordinates found in the uv coordinates of the layer.

7.19 I3DExTweener Interface

Family ID: 'twee'

Interface ID: IID_I3DExTweener

Interface file: I3DExTweener.h

Description

The **I3DExTweener** Interface defines Animation Tweeners. Tweeners are 3D Components that describe the interpolation between key frames.

Method Index

"I3DExTweener::SimpleTween(real&, int32, int32, int32)"

"I3DExTweener::Tween(I3DShKeyFrame*,int32, I3DShTweenerChainLink*)"

I3DExTweener::SimpleTween(real&, int32, int32, int32)

virtual MCOMErr MCOMAPI SimpleTween(real& result, int32 time, int32 time1, int32 time2)

SimpleTween() is called by the 3D Shell to perform interpolation at a given time. Replaces I3DExTweener::DoTweening from RDS.

Parameter

→ **real& result**

A value that tells the shell where to put the frame by interpolating linearly between the two extreme key frames.

→ **int32 time**

The time when to calculate the animation. Its unit is the ticks.

→ **int32 time1**

The time of the first key frame.

→ **int32 time2**

The time of the second key frame.

Comments

When *time* is equal to *time1*, return results=0.0; when *time* is equal to *time2*, return results=1.0.

For a linear tweener, the formula is:

*result=FixedToReal(time-time1)/FixedToReal(time2-time1);

I3DExTweener::Tween(I3DShKeyFrame*,int32, I3DShTweenerChainLink*)

virtual MCOMErr MCOMAPI 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.

Parameter	 I3DShKeyFrame* <i>res</i> A pointer to the current key frame.  int32 <i>time</i> The time when to calculate the animation. Its unit is the ticks.  I3DShTweenerChainLink* <i>alink</i> A pointer to the entire chain.
------------------	--

7.20 I3DExTweenEdition Interface

Family ID: 'twee'

Interface ID: IID_I3DExTweenEdition

Interface file: I3DExTween.h

Description

The **I3DExTweenEdition** Interface does **zot**. Tweeners are 3D Components that describe the interpolation between key frames.

I3DExTween::PrepareEdit(I3DShTweenChainLink*)

MCCOMErr MCCOMAPI PrepareEdit(I3DShTweenChainLink* alink)

PrepareEdit() zot needs description .

Parameter

 **I3DShTweenChainLink* alink**
A pointer to which link to edit.

7.21 I3DExWireFrame Interface

Interface ID: IID_I3DExWireFrame

Interface file: I3DWireFrame.h

Description

The **I3DExWireFrame** Interface allows you to activate, deactivate, and position wire frames. Wire frames are the lines projected by an object that display on the planes of the working box.

Method Index

```

“I3DExWireFrame::TrackWireFrame(I3DShWireFrame*, int16, I3DShTreeElement*, int16, const
TRACKINFO&, const TRACKINFO&, const TRACKINFO&)”
“I3DExWireFrame::DataToWireFrame(I3DShWireFrame*, int16, I3DShTreeElement*)”
“I3DExWireFrame::WireFrameToData(I3DShWireFrame*, int16, I3DShTreeElement*)”
“I3DExWireFrame::GetWireFrameBBox(I3DShTweenerChainLink*)”
“I3DExWireFrame::Activate()”
“I3DExWireFrame::Deactivate()”
“I3DExWireFrame::HandlesTool(int16)”

```

I3DExWireFrame::TrackWireFrame(I3DShWireFrame*, int16, I3DShTreeElement*, int16, const TRACKINFO&, const TRACKINFO&, const TRACKINFO&)

MCCOMErr MCCOMAPI TrackWireFrame(I3DShWireFrame* wireFrame, int16 proj, I3DShTreeElement* tree, int16 handle, const TRACKINFO& startinfo, const TRACKINFO& previnfo, const TRACKINFO& nextinfo)

TrackWireFrame() tracks the location of the specified wireframe, on the specified projection of the working box, for the specified tree element.

- Parameter**
- ➡ **I3DShWireFrame* wireFrame**
A pointer to the specified wire frame to track.
 - ➡ **int16 proj**
Specifies the projection of the working box.
 - ➡ **I3DShTreeElement* tree**
A pointer to tree of the object.
 - ➡ **int16 handle**
zot needs description.
 - ➡ **const TRACKINFO& startinfo**
zot needs description.
 - ➡ **const TRACKINFO& previnfo**
zot needs description.

→ **const TRACKINFO& nextinfo**
zot needs description.

I3DExWireFrame::DataToWireFrame(I3DShWireFrame*, int16, I3DShTreeElement*)

MCCOMErr MCCOMAPI DataToWireFrame(I3DShWireFrame* wireFrame, int16 proj, I3DShTreeElement* tree)
DataToWireFrame() orients the working box to the location of the wire frame.

Parameter

- **I3DShWireFrame* alink**
A pointer to the specified wire frame.
- **int16 proj**
Specifies the projection of the working box.
- **I3DShTreeElement* tree**
A pointer to tree of the object.

I3DExWireFrame::WireFrameToData(I3DShWireFrame*, int16, I3DShTreeElement*)

MCCOMErr MCCOMAPI WireFrameToData(I3DShWireFrame* wireFrame, int16 proj, I3DShTreeElement* tree)
WireFrameToData() orients the wire frame to the location of the working box.

Parameter

- **I3DShWireFrame* alink**
A pointer to the specified wire frame.
- **int16 proj**
Specifies the projection of the working box.
- **I3DShTreeElement* tree**
A pointer to tree of the object.

I3DExWireFrame::GetWireFrameBBox(I3DShTweenChainLink*)

MCCOMErr MCCOMAPI GetWireFrameBBox(TBBox3D* outBBox, int16proj, I3DShTreeElement*tree)
GetWireFrameBBox() gets the bounding box for the wire frame.

Parameter

- **TBBox3D* outBBox**
A pointer to the resulting bounding box.
- **int16 proj**
Specifies the projection of the working box.
- **I3DShTreeElement* tree**
A pointer to tree of the object.

I3DExWireFrame::Activate()

MCCOMErr MCCOMAPI Activate()

Activate() activates the wire frame.

I3DExWireFrame::Deactivate()

MCCOMErr MCCOMAPI Deactivate()

Deactivate() deactivates the wire frame.

I3DExWireFrame::HandlesTool(int16)

boolean MCCOMAPI HandlesTool(int16 inTool)

HandlesTool() zot needs description .

Parameter  **int16 inTool**
zot needs description.

7.22 IRaytracablePrimitive Interface

Interface ID: IID_IRaytracablePrimitive

Interface file: I3DExPrimitive.h

Description

In addition to describing geometry a geometric primitive can define a custom ray-tracing routine rather than relying only on facets or patches. This is done by inheriting from the “**IRaytracablePrimitive Interface**” in addition to the “**I3DExGeometricPrimitive Interface**” (or TBasicPrimitive).

Methods Index

“IRaytracablePrimitive::RayHit(boolean*, Ray3D*, RayHitParameters*, RayHit3D*)”

“IRaytracablePrimitive::GetRayHitDetails(RayHit3D*)”

“IRaytracablePrimitive::IsInfiniteBB(boolean&)”

IRaytracablePrimitive::RayHit(boolean*, Ray3D*, RayHitParameters*, RayHit3D*)

MCCOMErr RayHit(boolean* didHit, Ray3D* ray, RayHitParameters* hitParams, RayHit3D* hit)

RayHit() is called to allow the primitive to perform specific ray-tracing calculations instead of relying on the facets or patches. For example, a sphere can provide its intersection routine so the ray-tracing renders perfect spheres.

Call **IRaytracablePrimitive::GetRayHitDetails** if there is a hit and that you want more details on it.

If **IRaytracablePrimitive::RayHit** is implemented, then **IRaytracablePrimitive::GetRayHitDetails** must be implemented as well.

If they are not implemented, ray-tracing intersection will be based on the patches or the facets.

Parameters

➞ **boolean* didHit**

Return TRUE in this parameter if a hit occurred.

➞ **Ray3D* ray**

This is the ray provided by the 3D Shell.

➞ **RayHitParameters* hitParams**

These parameter give the minimum and the maximum 't' Ray parameter above which the hit is allowed to occur. See the Data Structure reference for “RayHitParameters” for more information.

➞ **RayHit3D* hit**

Points to a record that must be filled-in if a hit occurred. See the description of the data structure for all details. Depending on the flags set in the structure, more or less data should be calculated and stored in *hit*.

Return Value

MC_S_OK	
MC_E_NOTIMPL	Call not implemented.

IRaytracablePrimitive::GetRayHitDetails(RayHit3D*)**MCCOMErr GetRayHitDetails(RayHit3D* hit)**

GetRayHitDetails() is called after a ray-tracing hit occurred. If **IRaytracablePrimitive::GetRayHitDetails** is implemented, then **IRaytracablePrimitive::RayHit** must be implemented as well.

If they are not implemented ray-tracing intersection will be based on the patches or the facets.

Parameter

➞ **RayHit3D* hit**

Points to a record that must be filled-in. See the Data Structure on “RayHit3D”.

Return Value

MC_S_OK	
MC_E_NOTIMPL	Call not implemented.

IRaytracablePrimitive::IsInfiniteBB(boolean&)**MCCOMErr IsInfiniteBB(boolean& isInfinite)**

If the primitive has an infinite bounding box (i.e. the infinite plane) this method must return TRUE.

Parameter

➞ **boolean& isInfinite**

TRUE if the primitive has an infinite bounding box; FALSE if not.

Return Value

MC_S_OK	
MC_E_NOTIMPL	Call not implemented.

7.23 ISceneDocument Interface

Interface ID: IID_ISceneDocument

Interface file: ISceneDocument.h

Description

zot needs description. A scene is a collection of objects stored in a file and accessed through IDShScene. A scene also includes documents which are accessed through ISceneDocument. You can access the document windows, specify which room (module) you want to use, specify how you want the scene to be rendered in the window, and start or stop animation previews.

Method Index

```

"ISceneDocument::GetScene(I3DShScene**)"
"ISceneDocument::GetSceneSelection(ISceneSelection**)"
"ISceneDocument::GetSceneSelectionChannel(IChangeChannel**)"
"ISceneDocument::GetTreeHierarchyChannel(IChangeChannel**)"
"ISceneDocument::GetDocumentWindowByID(IMFPart**, IDType)"
"ISceneDocument::GetModuleByID(I3DShModule**, IDType)"
"ISceneDocument::GetSceneRenderingModule(I3DRenderingModule**)"
"ISceneDocument::StopAnimPreview()"
"ISceneDocument::StartAnimPreview()"

```

ISceneDocument::GetScene(I3DShScene**)

void MCCOMAPI GetScene(I3DShScene)**

GetScene() is called to get a scene. **zot** needs more description.

Parameters  **I3DShScene****
Specifies the scene.

ISceneDocument::GetSceneSelection(ISceneSelection**)

void MCCOMAPI GetSceneSelection(ISceneSelection)**

GetSceneSelection() is called to get a scene selection. **zot** needs more description.

Parameters  **ISceneSelection****
Specifies the scene selection.

ISceneDocument::GetSceneSelectionChannel(IChangeChannel**)

void MCCOMAPI GetSceneSelectionChannel(IChangeChannel)**

GetSceneSelectionChannel() is called to get a scene selection channel. **zot** needs more description.

Parameters  **IChangeChannel****
Specifies the selection channel.

ISceneDocument::GetTreeHierarchyChannel(IChangeChannel**)

void MCCOMAPI GetTreeHierarchyChannel(IChangeChannel)**

GetTreeHierarchyChannel() is called to get a tree hierarchy channel. **zot** needs more description.

Parameters  **IChangeChannel****
Specifies the selection channel.

ISceneDocument::GetDocumentWindowByID(IMFPart**, IDType)

void MCCOMAPI GetDocumentWindowByID(IMFPart, IDType inWindowID)**

GetDocumentWindowByID() is called to get a document window by specifying the window ID. **zot** needs more description.

Parameters  **IMFPart****
zot needs description.

 **IDType inWindowID**
zot needs description.

ISceneDocument::GetModuleByID(I3DShModule**, IDType)

void MCCOMAPI GetModuleByID(I3DShModule, IDType inModuleID)**

ModuleByID() is called to get a module by specifying the ID. **zot** needs more description.

Parameters  **I3DShModule****
zot needs description.

 **IDType inModuleID**
zot needs description.

ISceneDocument::GetSceneRenderingModule(I3DRenderingModule**)

void MCCOMAPI GetSceneRenderingModule(I3DRenderingModule renderingModule)**

GetSceneRenderingModule() is called to get a rendering module. **zot** needs more description.

Parameters  **I3DRenderingModule** renderingModule**
zot needs description.

ISceneDocument::StopAnimPreview()

boolean MCCOMAPI StopAnimPreview()

StopAnimPreview() is called to stop an animation preview. **zot** needs more description.

ISceneDocument::StartAnimPreview()

void MCCOMAPI StartAnimPreview()

StartAnimPreview() is called to start an animation preview. **zot** needs more description.

7.24 IVolumePrimitive Interface

Interface ID: IID_IVolumePrimitive

Interface file: I3DExPrimitive.h

Description

Rather than defining geometry, a primitive can also describe a volume. This is achieved by inheriting from the “**IVolumePrimitive Interface**” as well as the “**I3DExGeometricPrimitive Interface**” (or TBasicPrimitive).

There are two methods for implementing a volume primitive. **IVolumePrimitive::GetVolumedensity** provides a filter color and a glow color for each point of the volume primitive. **IVolumePrimitive::GetVolumeAttenuation** calculates the color from a ray.

Method Index

“IVolumePrimitive::GetVolumeDensity(const TVector3&, TMCColorRGB&, TMCColorRGB&)”
“IVolumePrimitive::GetVolumeAttenuation(const TVector3&, constTVector3&, TMCColorRGB&)”

IVolumePrimitive::GetVolumeDensity(const TVector3&, TMCColor- RGB&, TMCColorRGB&)

MCCOMErr GetVolumeDensity(const TVector3& point, TMCColorRGB& filter, TMCColorRGB& glow)

GetVolumeDensity() is called during rendering to calculate a color from a ray going thru the volume primitive. The shell calls this method to get a filtered color and a glow color for many points on the ray. It begins with a base color from an object or background color behind the volumetric primitive to calculate the color within the volume:

Volume Color = (base color * filtered color) + glow

Parameters

-  **const TVector3& point**
The point within the volume in local coordinates.
-  **TMCColorRGB& filter**
The resulting filtered color.
-  **TMCColorRGB& glow**
The resulting glow value.

Return Value

MC_S_OK	
MC_E_NOTIMPL	Call not implemented.

IVolumePrimitive::GetVolumeAttenuation(const TVector3&, const TVector3&, TMCCColorRGB&)

MCCOMErr GetVolumeAttenuation(const TVector3& from, const TVector3& to, TMCCColorRGB& attenuation)

GetVolumeAttenuation() is called by the shell to get a color from a ray going thru a volume primitive.

Parameters

✚ **const TVector3& from**

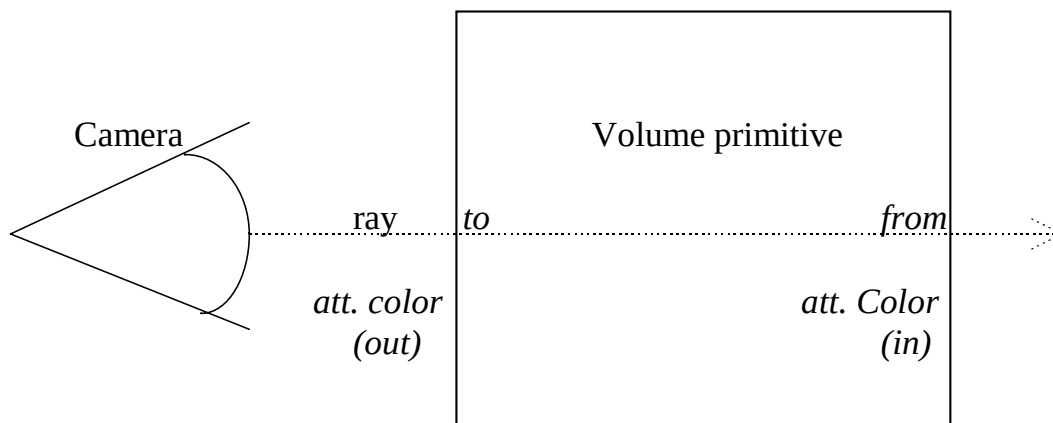
The intersection of the ray with the back of the volume primitive. This is where the calculation begins.

✚ **const TVector3& to**

The intersection of the ray with the front of the volume primitive. This is where the calculation ends.

✚ **TMCCColorRGB& attenuation**

The color behind the object is given. The resulting color is returned.



Return Value

MC_S_OK	
MC_E_NOTIMPL	This method is not supported. IVolumePrimitive::GetVolumeDensity will be used.

Chapter 8 - MCCCommon ISh* Reference

8.1 IShAction Interface

Interface ID: IID_IShAction

Interface file: IShUtilities.h

Description

The **IShAction** Interface contains methods to do and undo changes. <zot needs more description>

Methods Index

“IShAction::Do()”
“IShAction::Undo()”
“IShAction::Redo()”
“IShAction::WillCauseChange()”
“IShAction::CanUndo()”
“IShAction::GetPartToRedraw(IMFPart**, int32)”
“IShAction::GetName(TMCString&)”
“IShAction::NeedsToPrepareMenus(void)”

IShAction::Do()

MCCOMErr MCCOMAPI Do()

Do() is called to do the action. Since the constructor of the action is not protected by the exception mechanism, no memory allocations (like 'pointer = new ...') should be made in the constructor. Make only initializations (like 'pointer = NULL;').

IShAction::Undo()

MCCOMErr MCCOMAPI Undo()

Undo() is called to undo the action. When undone, an action must restore the database in the EXACT SAME state than before Do is called. This means that if a pointer is cloned, the action must store the original pointer in Do(), and apply its modifications to the clone. And therefore when Undo() is called, the action simply replaces the cloned and modified pointer by the original and unmodified one.

If CanUndo() returns False, just Undo() should return MC_E_NOTIMPL.

IShAction::Redo()

MCCOMErr MCCOMAPI Redo()

Redo() is called to redo the action.

If CanUndo() returns False, just Redo() should return MC_E_NOTIMPL.

IShAction::WillCauseChange()

boolean MCCOMAPI WillCauseChange()

WillCauseChange() returns True if the action will modify the database.

IShAction::CanUndo()

boolean MCCOMAPI CanUndo()

CanUndo() returns True if the action is undoable. If CanUndo() returns False, just Undo() and Redo() should return MC_E_NOTIMPL.

IShAction::GetPartToRedraw(IMFPart**, int32)

void MCCOMAPI GetPartToRedraw(IMFPart outPart, int32 stage)**

GetPartToRedraw() provides an opportunity to the action to tell the Shell that the returned view should be redrawn after Do/Undo/Redo. Return NULL if no redraw is needed.

Parameter  **IMFPart** outPart**
The resulting part that is redrawn.

 **int32 stage**
The area to be redrawn.

IShAction::GetName(TMCString&)

MCCOMErr MCCOMAPI GetName(TMCString& name)

GetName() returns the name to display in the undo/redo list. To avoid localization problems, don't use hardcoded strings such as `char name[] = "My Tool";` Use strings from resources instead.

Parameter  **TMCString& name**
The name to be displayed in the undo/redo list.

IShAction::NeedsToPrepareMenus(void)

boolean MCCOMAPI NeedsToPrepareMenus(void)

NeedsToPrepareMenus() returns True if execution (Do()/Undo()) of the action should cause the menus to be prepared. For example, the contents of the menus need to be enabled or disabled.

Parameter  **void**
<zot needs description>.

8.2 IShActionManager Interface

Interface ID: IID_IShActionManager

Interface file: IShUtilities.h

Description

The **IShActionManager** Interface contains methods to launch and measure mouse actions and keyboard clicks. <zot needs more description>

Methods Index

“IShActionManager::PostAction(IShAction*, int32, IMFResponder*)”
 “IShActionManager::PostMouseAction(IShMouseAction*, int32, IMFResponder* IMFPart*, const TMCPoint&)”
 “IShActionManager::IsOptionDown()”
 “IShActionManager::IsShiftDown()”
 “IShActionManager::IsCommandDown()”
 “IShActionManager::IsControlDown()”
 “IShActionManager::IsLongStaticClick()”
 “IShActionManager::IsShortStaticClick()”

IShActionManager::PostAction(IShAction*, int32, IMFResponder*)

MCCOMErr MCCOMAPI PostAction(IShAction* action,int32 actionNbr,IMFResponder *context)

PostAction() launches an action.

Parameter

- ➦ **IShAction* action**
A pointer to the action.
- ➦ **int32 actionNbr**
The action number.
- ➦ **IMFResponder* context**
The context should be the IMFResponder interface of the document.

IShActionManager::PostMouseAction(IShMouseAction*, int32, IMFResponder* IMFPart*, const TMCPoint&)

MCCOMErr MCCOMAPI PostMouseAction(IShMouseAction* action,int32 actionNbr,IMFResponder *context, IMFPart *part,const TMCPoint &mouse)

PostMouseAction() launches a mouse action.

Parameter

- ➦ **IShAction* action**
A pointer to the action.
- ➦ **int32 actionNbr**
The action number.

- **IMFResponder*** *context*
The context should be the IMFResponder interface of the document.
 - **IMFPart*** *part*
A pointer to the part.
 - **const TMCPoint&** *mouse*
A reference to the mouse location.
-

IShActionManager::IsOptionDown()

boolean MCCOMAPI IsOptionDown()

IsOptionDown() returns True if the Option key is depressed.

IShActionManager::IsShiftDown()

boolean MCCOMAPI IsShiftDown()

IsShiftDown() returns True if the Shift key is depressed.

IShActionManager::IsCommandDown()

boolean MCCOMAPI IsCommandDown()

IsCommandDown() returns True if the Command key is depressed.

IShActionManager::IsControlDown()

boolean MCCOMAPI IsControlDown()

IsControlDown() returns True if the Control key is depressed on a Macintosh.

IShActionManager::IsLongStaticClick()

boolean MCCOMAPI IsLongStaticClick()

IsLongStaticClick() returns True if click was short and the mouse did not move too much. Call this method if the mouse is already down.

IShActionManager::IsShortStaticClick()

boolean MCCOMAPI IsShortStaticClick()

IsShortStaticClick() returns True if click was long and the mouse did not move too much. Call this method if the mouse is already down.

8.3 IShComponent Interface

Interface ID: IID_IShComponent

Interface file: IShComponent.h

Description

The **IShComponent** Interface contains methods for identifying a component. <zot needs more description>

Methods Index

“IShComponent::GetFamilySignature()”
“IShComponent::GetClassSignature()”
“IShComponent::GetClassName(TMCString&)”
“IShComponent::IsDefaultComponent()”
“IShComponent::InvalidatePreview()”

IShComponent::GetFamilySignature()

IDTypeMCCOMAPI GetFamilySignature()

GetFamilySignature() gets the family signature. <zot needs description.>

<zot: is the signature the same as the Family ID in the COMP resource? If so, I can add more to this.>

IShComponent::GetClassSignature()

IDTypeMCCOMAPI GetClassSignature()

GetClassSignature() gets the signature of the class. <zot needs description>.

<zot: is signature the same as the Class ID in the COMP resource? If so, I can add more to this.>

IShComponent::GetClassName(TMCString&)

MCCOMErrMCCOMAPI GetClassName(TMCString& name)

GetClassName() gets the name of the class.

Parameter

 **TMCString& name**
A reference to the name of the class .

IShComponent::IsDefaultComponent()

booleanMCCOMAPI IsDefaultComponent()

IsDefaultComponent() usually returns False, returns True for components who use a default instance (like

dataComponents).

IShComponent::InvalidatePreview()

voidMCCOMAPI InvalidatePreview()

InvalidatePreview() invalidates the component preview (if there is any).

8.4 IShComponentUtilities Interface

Interface ID: IID_IShComponentUtilities

Interface file: IShUtilities.h

Description

The **IShComponentUtilities** Interface contains methods to create, read, and write components. <zot needs more description>

Methods Index

```

“IShComponentUtilities::CoCreateInstance(const MCCLSID&, IMCUnknown*, uint32, const MCIID&, void**)”
“IShComponentUtilities::CreateComponent(IDType, IDType, IShComponent**)”
“IShComponentUtilities::CreateIUnknown(IDType, IDType, IMCUnknown*, IMCUnknown**)”
“IShComponentUtilities::CreateDefaultComponent(IDType, IShComponent**)”
“IShComponentUtilities::GetComponentClassInfoArrayByFamilyID(IDType, TMCArray<TComponent-
ClassInfo>&)”
“IShComponentUtilities::WriteComponent(IShTokenStream*, IShParameterComponent*)”
“IShComponentUtilities::WriteComponent(IShTokenStream*, IDType, IShParameterComponent*)”
“IShComponentUtilities::ReadComponent(IShTokenStream*, IShParameterComponent**)”
“IShComponentUtilities::ReadComponent(IShTokenStream*, IShParameterComponent**, IDType)”
“IShComponentUtilities::ReadComponentData(TMCiostream*, IShParameterComponent*)”
“IShComponentUtilities::AskForParams(IShParameterComponent*)”

```

IShComponentUtilities::CoCreateInstance(const MCCLSID&, IMCUnknown*, uint32, const MCIID&, void**)

MCCOMErr MCCOMAPI CoCreateInstance(const MCCLSID& rclsid, IMCUnknown* pUnkOuter, uint32 dwClsContext, const MCIID& riid, void ppv)**

CoCreateInstance() creates an instance of a component.

Parameter

- **const MCCLSID& rclsid**
A reference to the MetaCreations class id.
- **IMCUnknown* pUnkOuter**
A pointer to the <zot needs definition>.
- **uint32 dwClsContext**
The class context value.
- **const MCIID& riid**
The class IID_ value.
- **void** ppv**
The <zot needs defintion>.

IShComponentUtilities::CreateComponent(IDType, IDType, IShComponent**)

MCCOMErr MCCOMAPI CreateComponent(IDType familyID, IDType classID, IShComponent component)**

CreateComponent() creates a component, given the family id, the class id, and returns a pointer to the component. For example, CreateInstanceByID('gel ', 'blnd', &gel) creates a "blinds" gel.

Parameter

- **IDType** *familyID*
The family id of the component being created.
- **IDType** *classID*
The class id of the component being created.
- **IShComponent**** *component*
A pointer to the component created using the first two arguments.

IShComponentUtilities::CreateUnknown(IDType, IDType, IMCUnknown*, IMCUnknown**)

MCCOMErr MCCOMAPI CreateUnknown(IDType familyID, IDType classID, IMCUnknown* pUnkOuter, IMCUnknown component)**

CreateUnknown() creates a component, given the family id, the class id, and returns a pointer to the component.

Parameter

- **IDType** *familyID*
The family id of the component being created.
- **IDType** *classID*
The class id of the component being created.
- **IMCUnknown*** *pUnkOuter*
A pointer to the <zot needs definition>.
- **IMCUnknown**** *component*
A pointer to the component created using the first two arguments.

IShComponentUtilities::CreateDefaultComponent(IDType, IShComponent**)

MCCOMErr MCCOMAPI CreateDefaultComponent(IDType familyID, IShComponent component)**

CreateDefaultComponent() creates a component, given the family id, and returns a pointer to the component.

Parameter

- **IDType** *familyID*
The family id of the component being created.
- **IShComponent**** *component*
A pointer to the component created using the first argument.

IShComponentUtilities::GetComponentClassInfoArrayByFamilyID(IDType, TMCArray<TComponentClassInfo>&)

MCCOMErr MCCOMAPI GetComponentClassInfoArrayByFamilyID(IDType familyID, TMCArray<TComponentClassInfo>& componentClassInfoArray)

GetComponentClassInfoArrayByFamilyID() gets the component class information array, given the family id, and returns a reference to the array.

Parameter

- **IDType** *familyID*
The family id of the component.
- **TMCArray<TComponentClassInfo>&** *componentClassInfoArray*
A reference to the array of the component indicated using the first argument.

IShComponentUtilities::WriteComponent(IShTokenStream*, IShParameterComponent*)

MCCOMErr MCCOMAPI WriteComponent(IShTokenStream* stream, IShParameterComponent* component)

WriteComponent() writes a specified component to the stream.

Parameter

- **IShTokenStream*** *stream*
A pointer to the stream.
- **IShParameterComponent*** *component*
A pointer to the component.

IShComponentUtilities::WriteComponent(IShTokenStream*, IDType, IShParameterComponent*)

MCCOMErr MCCOMAPI WriteComponent(IShTokenStream* stream, IDType keyword, IShParameterComponent* component)

WriteComponent() writes a specified component with a specified keyword to the stream.

Parameter

- **IShTokenStream*** *stream*
A pointer to the stream.
- **IDType** *keyword*
The keyword specifies the id type.
- **IShParameterComponent*** *component*
A pointer to the component.

IShComponentUtilities::ReadComponent(IShTokenStream*, IShParameterComponent**)

MCCOMErr MCCOMAPI ReadComponent(IShTokenStream* stream, IShParameterComponent component)**

ReadComponent() reads a specified component from the stream.

Parameter	 IShTokenStream* <i>stream</i> A pointer to the stream.
	 IShParameterComponent** <i>component</i> A pointer to the component.

IShComponentUtilities::ReadComponent(IShTokenStream*, IShParameterComponent**, IDType)

MCCOMErr MCCOMAPI ReadComponent(IShTokenStream* stream, IShParameterComponent component, IDType familyID)**

ReadComponent() reads a specified component from the stream. Use this method when the family token has already been read (the begin token is still in the stream).

Parameter	 IShTokenStream* <i>stream</i> A pointer to the stream.
	 IShParameterComponent** <i>component</i> A pointer to the component.
	 IDType <i>familyID</i> The family id of the component being read.

IShComponentUtilities::ReadComponentData(TMCiostream*, IShParameterComponent*)

MCCOMErr MCCOMAPI ReadComponentData(TMCiostream* stream, IShParameterComponent* component)

ReadComponentData() reads the data from a specified component from the stream.

Parameter	 IShTokenStream* <i>stream</i> A pointer to the stream.
	 IShParameterComponent* <i>component</i> A pointer to the component.

IShComponentUtilities::AskForParams(IShParameterComponent*)

boolean MCCOMAPI AskForParams(IShParameterComponent *component)

AskForParams() returns True if there are parameters for the component.

Parameter	 IShParameterComponent* <i>component</i> A pointer to the component.
------------------	---

8.5 IShParameterComponent Interface

Interface ID: IID_IShParameterComponent

Interface file: IShComponent.h

Description

The **IShParameterComponent** Interface contains methods for identifying and accessing a parameter component. <zot needs more description>

Methods Index

“IShParameterComponent::GetFamilySignature()”
 “IShParameterComponent::GetClasSignature()”
 “IShParameterComponent::GetClassName(TMCString&)”
 “IShParameterComponent::GetParameter(IDType, void*)”
 “IShParameterComponent::SetParameter(IDType, void*, boolean)”
 “IShParameterComponent::GetParameterType(IDType, IDType&)”
 “IShParameterComponent::GetParameterCount()”
 “IShParameterComponent::GetParameterInfo(uint32, TMapElement&)”
 “IShParameterComponent::Inval(boolean, boolean)”

IShParameterComponent::GetFamilySignature()

IDTypeMCCOMAPI GetFamilySignature()

GetFamilySignature() gets the family signature. <zot needs description.>

<zot: is the signature the same as the Family ID in the COMP resource? If so, I can add more to this.>

IShParameterComponent::GetClasSignature()

IDTypeMCCOMAPI GetClassSignature()

GetClassSignature() gets the signature of the class. <zot needs description>.

<zot: is signature the same as the Class ID in the COMP resource? If so, I can add more to this.>

IShParameterComponent::GetClassName(TMCString&)

MCCOMErrMCCOMAPI GetClassName(TMCString& name)

GetClassName() gets the name of the class.

Parameter

 **TMCString& name**

A reference to the name of the class .

IShParameterComponent::GetParameter(IDType, void*)

MCCOMErr MCCOMAPI GetParameter(IDType keyword, void* parameter)

GetParameter() gets the parameter, given the IDType.

- Parameter**
- ✚ **IDType** *keyword*
<zot needs description>.
 - ✚ **void*** *parameter*
A pointer to the parameter.

IShParameterComponent::SetParameter(IDType, void*, boolean)

MCCOMErr MCCOMAPI SetParameter(IDType keyword, void* parameter, boolean inNotify=true)

SetParameter() sets the parameter, given the IDType, and returns True unless <zot needs description>.

- Parameter**
- ✚ **IDType** *keyword*
<zot needs description>.
 - ✚ **void*** *parameter*
A pointer to the parameter.
 - ✚ **boolean** *inNotify*
Set to True, <zot needs description>. Returns False if <zot needs description>

IShParameterComponent::GetParameterType(IDType, IDType&)

MCCOMErr MCCOMAPI GetParameterType(IDType keyword, IDType& parameterType)

GetParameterType() gets the parameter type, given the IDType, updates the reference to that IDType <zot needs description>.

- Parameter**
- ✚ **IDType** *keyword*
<zot needs description>.
 - ✚ **IDType&** *parameterType*
A reference to the parameter type.

IShParameterComponent::GetParameterCount()

uint32 MCCOMAPI GetParameterCount()

GetParameterCount() returns the count of the parameters.

IShParameterComponent::GetParameterInfo(uint32, TMapElement&)

MCCOMErr MCCOMAPI GetParameterInfo(uint32 index, TMapElement& info)

GetParameterInfo() gets the parameter information given an index, and returns a reference to the information. <zot needs description>.

Parameter	 uint32 <i>index</i> <zot needs description>.
	 TPMapElement& <i>info</i> A reference to <zot needs description>

IShParameterComponent::Inval(boolean, boolean)

void MCCOMAPI Inval(boolean inNotifyExternal = true, boolean inNotifyUI = false)

Inval() <zot needs description>.

Parameter	 boolean <i>inNotifyExternal</i> Returns True when <zot needs description>. Returns False when <zot needs description>.
	 boolean <i>inNotifyUI</i> Returns False when <zot needs description>. Returns True when <zot needs description>.

8.6 IShUtilities Interface

Interface ID: IID_IShUtilities

Interface file: IShUtilities.h

Description

The **IShUtilities** Interface contains methods for accessing a progress bar, tracking and reporting information about the progress. It also manages resources related to the process. <zot needs more description>

Methods Index

```

“IShUtilities::BeginProgress(TMCString&, IMCUnknown**, real32)”
“IShUtilities::EndProgress(IMCUnknown*)”
“IShUtilities::SetProgressPrompt(TMCString&, IMCUnknown*)”
“IShUtilities::IncrementProgress(real32, IMCUnknown*)”
“IShUtilities::SetProgressValue(real32, IMCUnknown*)”
“IShUtilities::SetProgressValue0to1(real32, IMCUnknown*)”
“IShUtilities::YieldProcesses(uint32)”
“IShUtilities::TickCount()”
“IShUtilities::Random()”
“IShUtilities::SetRandomSeed(uint32)”
“IShUtilities::RandomStep(int32)”
“IShUtilities::GetAPINumber()”
“IShUtilities::Gestalt(void*, uint32)”
“IShUtilities::DoAlert(EAlerts, TMCString&, TMCString&, TMCString&, TMCString&, int32, TMC-
String&, TMCString&, TMCString&)”
“IShUtilities::ErrorToString(MCCOMErr, TMCString&)”
“IShUtilities::BuildAlertMessage(MCCOMErr, TMCString&, TMCString&)”
“IShUtilities::GetResource(int32, int32, int32, void**)”
“IShUtilities::SetupResources(int32, int32, void**)”
“IShUtilities::RestoreResources(void*)”
“IShUtilities::BlitUIImage(IMCGraphicContext*, TMCRect&, TMCRect&, uint32)”
“IShUtilities::GetUIImageInfo(uint32, int32&, int32&, int32)”
“IShUtilities::GetUIImageBucket(uint32, TMCPixelBucket*)”
“IShUtilities::GetUIImage(uint32, TMCUIImage**)”
“IShUtilities::CreateCompatiblePixelBucket(int32, int32, int32, TMCCompatiblePixelBucket**)”
“IShUtilities::DestroyCompatiblePixelBucket(TMCCompatiblePixelBucket*)”
“IShUtilities::GetAsciiTranslationTable()”
“IShUtilities::GetOpenDocumentsCount()”
“IShUtilities::GetOpenDocumentByIndex(IMFDocument**, index)”
“IShUtilities::GetLastOpenDocPath(TMCString&)”
“IShUtilities::GetLastSaveDocPath(TMCString&)”
“IShUtilities::SetLastOpenDocPath(TMCString&)”
“IShUtilities::SetLastSaveDocPath(TMCString&)”
“IShUtilities::UseMMX()”
“IShUtilities::IsPentiumProcessor()”
“IShUtilities::IsPentiumIIProcessor()”
“IShUtilities::IsPentiumIIIProcessor()”
“IShUtilities::StringWidth(TMCTextStyle&, TMCString&)”
“IShUtilities::GetTextMetrics(TMCTextStyle&, TMCTextMetrics&)”
“IShUtilities::GetCreator()”
“IShUtilities::SetCursor(int32)”

```

“IShUtilities::GetFontList(IMCFontList**)”
 “IShUtilities::CheckForUserCancel()”
 “IShUtilities::GetCurrentModifiers()”
 “IShUtilities::CheckKeyDown(int16)”

IShUtilities::BeginProgress(TMCString&, IMCUnknown**, real32)

void MCCOMAPI BeginProgress(TMCString& prompt, IMCUnknown outProgressKey, real32 inMaxValue = 100.0f)**

BeginProgress() begins a progress bar. BeginProgress returns a progressKey which is to be passed into the other methods---if you pass NULL as a key, it is assumed to be the main/first progress.

- Parameter**
- ➦ **TMCString& prompt**
A reference to the string that displays with the progress bar.
 - ➦ **IMCUnknown** outProgressKey**
A pointer to the progress key used to specify which progress the bar is tracking, and tracks the progress of the process.
 - ➦ **real32 inMaxValue**
The maximum value of a progressKey is set to 100.0f by default. <zot what is the range or typical range?>

IShUtilities::EndProgress(IMCUnknown*)

void MCCOMAPI EndProgress(IMCUnknown* inProgressKey)

EndProgress() ends the progress bar for the progress specified by the progress key. <zot needs description.>

- Parameter**
- ➦ **IMCUnknown** inProgressKey**
Identifies which progress is being ended. BeginProgress returns a progressKey which is to be passed into the other methods---if you pass NULL as a key, it is assumed to be the main/first progress.

IShUtilities::SetProgressPrompt(TMCString&, IMCUnknown*)

void MCCOMAPI SetProgressPrompt(TMCString& inPrompt, IMCUnknown* inProgressKey = NULL)

SetProgressPrompt() sets up the string that displays while process is occurring.

- Parameter**
- ➦ **TMCString& inPrompt**
A reference to the string that displays while the progress bar is progressing.
 - ➦ **IMCUnknown* inProgressKey**
Identifies which progress is being set. BeginProgress returns a progressKey which is to be passed into the other methods---if you pass NULL as a key, it is assumed to be the main/first progress. Progress values are between 0.0f and max value.

IShUtilities::IncrementProgress(real32, IMCUnknown*)

void MCCOMAPI IncrementProgress(real32 inIncrement, IMCUnknown* inProgressKey = NULL)

IncrementProgress() changes the value of the progress bar.

- Parameter**
- ➦ **TMCString& inPrompt**
A reference to the string that displays during the process.
 - ➦ **IMCUnknown* inProgressKey**
Identifies which progress is being set. BeginProgress returns a progressKey which is to be passed into the other methods---if you pass NULL as a key, it is assumed to be the main/first progress. Progress values are between 0.0f and max**Value**.

IShUtilities::SetProgressValue(real32, IMCUnknown*)

voidMCCOMAPI SetProgressValue(real32 inValue, IMCUnknown* inProgressKey = NULL)

SetProgressValue() sets the value for the progress bar.

- Parameter**
- ➦ **real32 inValue**
The initial progress value. Progress values are between 0.0f and max**Value**.
 - ➦ **IMCUnknown* inProgressKey**
Identifies which progress is being set. BeginProgress returns a progressKey which is to be passed into the other methods---if you pass NULL as a key, it is assumed to be the main/first progress.

IShUtilities::SetProgressValue0to1(real32, IMCUnknown*)

voidMCCOMAPI SetProgressValue0to1(real32 inValue, IMCUnknown* inProgressKey = NULL)

SetProgressValue() sets the value for the progress bar to a range of 0 to 1.

- Parameter**
- ➦ **real32 inValue**
The initial progress value.
 - ➦ **IMCUnknown* inProgressKey**
Identifies which progress is being set. BeginProgress returns a progressKey which is to be passed into the other methods---if you pass NULL as a key, it is assumed to be the main/first progress.

IShUtilities::YieldProcesses(uint32)

MCCOMErr MCCOMAPI YieldProcesses(uint32 ticks)

YieldProcesses() allows background processes to receive processing time. This method is often used by renderers to allow background rendering.

- Parameter**
- ➦ **uint32 ticks**
The number of ticks (a tick is approximately 1/60th of a second) indicating the amount of time you want to stay in your code before allowing processing time to other processes. Time between calls to Yield are accumulated until ticks is reached. Then switching to other processes occurs at the next Yield, and the counter is reset when coming back to your code.

Note

This call can trigger a Failure if the current process is about to be killed. In that case, you should perform whatever clean-up is necessary before transmitting the Failure. See the Overview <zot xref here> for more information on the Failure Handling system.

IShUtilities::TickCount()

uint32 MCCOMAPI TickCount()

TickCount() gets an arbitrary time in 1/60 of a second (depending on the platform, it can be the number of ticks since the last boot or something else).

Return Value

Number of ticks (1/60th of a second).

Comments

Use this call to perform any time-based calculations.

IShUtilities::Random()

uint32 MCCOMAPI Random()

Random() returns a random integer.

IShUtilities::SetRandomSeed(uint32)

MCCOMErr MCCOMAPI SetRandomSeed(uint32 newSeed)

SetRandomSeed() changes the seed value of a random function.

Parameter

 **uint32 newSeed**
The new seed value.

IShUtilities::RandomStep(int32)

int32 MCCOMAPI RandomStep(int32 before)

RandomStep() changes the step value in a random function < zot, BIG guess on my part...>. RandomStep uses the initial step value to calculate and return a new step value.

Parameter

 **int32 before**
The initial step value.

IShUtilities::GetAPINumber()

uint32 MCCOMAPI GetAPINumber()

GetAPINumber() gets the version of the API. This is used to determine component version compatibility.

IShUtilities::Gestalt(void*, uint32)

MCCOMErr MCCOMAPI Gestalt(void* data,uint32 id)

Gestalt() gets information on the system <zot needs description>.

Parameter ➤ **void* data**
A pointer to where to store the information.

➤ **uint32 id**
The type of information.

ID	Type of data	Version	Meaning
Vers	long	1	The actual version of Gestalt.
FN83	Boolean	1	TRUE if the name is 8 + 3 characters

For example, set *id* to **IDTYPE('F', 'N', '8', '3')** to know if the file names are 8 characters long plus 3 characters for the extension.

IShUtilities::DoAlert(EAlerts, TMCString&, TMCString&, TMCString&, TMCString&, int32, TMCString&, TMCString&, TMCString&)

int16 MCCOMAPI DoAlert(EAlerts alertSeverity, const TMCString& mainMessage, const TMCString& arg0, const TMCString& arg1, const TMCString& arg2, int32 buttonsCount, const TMCString& button0, const TMCString& button1, const TMCString& button2)

DoAlert() opens a standard alert. Use DoAlert to post easily an alert and get some response from the user. The returned value is the number of the hit button (0, 1 or 2). If you give empty strings for buttons, you get the following default buttons:

```
// buttonsCount == 1   => OK
// buttonsCount == 2   => OK Cancel
// buttonsCount == 3   => Yes No Cancel
```

Parameter ➤ **IDType alertSeverity**
Alert severity icon. See kShNotificationAlert, kShCautionAlert and kShStopAlert constants.

➤ **TMCString& mainMessage**
A reference to the Main message to display.

➤ **TMCString& arg0, arg1, arg2**
A reference to the Substitution strings : any "^0", "^1" and "^2" token in 'mainMessage' will be replaced by 'arg0', 'arg1', and 'arg2'.

➤ **int32 buttonsCount**
A reference to the Number of buttons (1, 2 or 3).

➤ **TMCString& button0**
A reference to the text of the default button.

➤ **TMCString& button1**
A reference to the text of the second button.

- **TMCString& *button2***
A reference to the text of the third button.

IShUtilities::ErrorToString(MCCOMErr, TMCString&)

MCCOMErr MCCOMAPI ErrorToString(MCCOMErr inErr, TMCString& outErrorString)

ErrorToString() writes the error message to a string, so that it can be used as text for an alert.

- Parameter**
- **MCCOMErr *inErr***
The error to convert to a string.
 - **TMCString& *outErrorString***
A reference to the string containing the error message text.

IShUtilities::BuildAlertMessage(MCCOMErr, TMCString&, TMCString&)

void MCCOMAPI BuildAlertMessage(MCCOMErr inErr, const TMCString& inContext, TMCString& outMessage)

BuildAlertMessage() builds the alert, using the error message string, as text for an alert.

- Parameter**
- **MCCOMErr *inErr***
The error that generates the need for an alert.
 - **TMCString& *inContext***
A reference to the string containing the error message text.
 - **TMCString& *outMessage***
A reference to the string containing the error text as it appears in the alert.

IShUtilities::GetResource(int32, int32, int32, void**)

MCCOMErr MCCOMAPI GetResource(int32 family, int32 classID, int32 resType, void data)**

GetResource() gets a resource with the specified family, class ID, resource type, and provides a pointer to its data.

- Parameter**
- **int32 *family***
The family id.
 - **int32 *classID***
The class ID.
 - **int32 *resType***
The resource type.
 - **void** *data***
A pointer to data for that resource.

IShUtilities::SetupResources(int32, int32, void**)

MCCOMErr MCCOMAPI SetupResources(int32 family, int32 classID, void oldResources)**

SetupResource() forces the Shell to use a given resource file. Just ask for the family and class of the extension you currently consider, and **SetupResources** sets the correct resource file.

- Parameter**
- **int32 family**
The family id.
 - **int32 classID**
The class ID.
 - **void** oldResources**
This is the previous resource file information. Store this value, and give it back when restoring the previous resource file via **IShUtilities::RestoreResources**.

IShUtilities::RestoreResources(void*)

MCCOMErr MCCOMAPI RestoreResources(void* oldResources)

RestoreResources() restores a previous resource file.

- Parameter**
- **void* oldResources.**
This is the previous resource file information. Give this value back when restoring the previous resource file.

IShUtilities::BlitUIImage(IMCGraphicContext*, TMCRect&, TMCRect&, uint32)

MCCOMErr MCCOMAPI BlitUIImage(IMCGraphicContext* inGraphicContext, TMCRect& inRect, const TMCRect& inUpdateBBox, uint32 inResourceID)

BlitUIImage() forces the Shell to update a specific portion of the screen specified by the update bounding box. <zot needs description>

- Parameter**
- **IMCGraphicContext* inGraphicContext**
A pointer to the context for the <zot needs description>.
 - **TMCRect& inRect**
A reference to the <zot needs description>.
 - **TMCRect& inUpdateBox**
A reference to the <zot needs description>.
 - **uint32 inResourceID**
The resource ID of the image being updated? <zot needs description>.

IShUtilities::GetUIImageInfo(uint32, int32&, int32&, int32)

MCCOMErr MCCOMAPI GetUIImageInfo(uint32 inResourceID, int32& width, int32& height, int32& planes)

GetUIImageInfo() allocates the UI image. It must be allocated before you can call GetUIImageBucket.

<zot needs description>

- Parameter**
- **uint32 inResourceID**
The resource ID. <zot needs description>.
 - **int32& width**
A reference to the width of the image <zot needs description>.
 - **int32& height**
A reference to the height of the image <zot needs description>.
 - **int32& planes**
A reference to the planes of the image <zot needs description>.

IShUtilities::GetUIImageBucket(uint32, TMCPixelBucket*)

MCCOMErr MCCOMAPI GetUIImageBucket(uint32 inResourceID, TMCPixelBucket* bucket)

GetUIImageBucket() blits the UI uimage into a pixel bucket - it must be allocated before calling (that's what the previous call is for).

- Parameter**
- **uint32 inResourceID**
The resource ID of the object. <zot needs description>.
 - **TMCPixelBucket* bucket**
A pointer to the destination pixel bucket.<zot needs description>.

IShUtilities::GetUIImage(uint32, TMCUIImage**)

MCCOMErr MCCOMAPI GetUIImage(uint32 inResourceID, TMCUIImage outUiImage)**

GetUIImage() directly gets the UI uimage.

- Parameter**
- **uint32 inResourceID**
The resource ID of the object. <zot needs description>.
 - **TMCUIImage** outUiImage**
Apointer to the resulting UI Image. <zot needs description>.

IShUtilities::CreateCompatiblePixelBucket(int32, int32, int32, TMC-CompatiblePixelBucket**)

MCCOMErr MCCOMAPI CreateCompatiblePixelBucket(int32 widthParam, int32 heightParam, int32 depthParam, TMCCompatiblePixelBucket outBucket)**

CreateCompatiblePixelBucket() creates a TMCCompatiblePixelBucket with the specified dimensions. <zot needs description>

- Parameter**
- **int32 widthParam**
The width value for the resulting pixel bucket. <zot needs description>.
 - **int32& heightParam**
The heigh value for the resulting pixel bucket. <zot needs description>.

- ➦ **int32& depthParam**
The depth value for the resulting pixel bucket. <zot needs description>.
- ➦ **TMCCompatiblePixelBucket** outBucket**
A pointer to the resulting pixel bucket. <zot needs description>.

IShUtilities::DestroyCompatiblePixelBucket(TMCCompatiblePixelBucket*)

MCCOMErr MCCOMAPI DestroyCompatiblePixelBucket(TMCCompatiblePixelBucket* outBucket)

DestroyCompatiblePixelBucket() destroys the specified a TMCCompatiblePixelBucket. <zot needs description>

Parameter ➦ **TMCCompatiblePixelBucket* outBucket**
A pointer to the specified pixel bucket. <zot needs description>.

IShUtilities::GetAsciiTranslationTable()

TMCAsciiTranslationTable* MCCOMAPI GetAsciiTranslationTable()

GetAsciiTranslationTable() returns a pointer to a TMCAsciiTranslationTable. <zot needs description>

IShUtilities::GetOpenDocumentsCount()

uint32 MCCOMAPI GetOpenDocumentsCount()

GetOpenDocumentsCount() returns the number of open documents. <zot needs description>

IShUtilities::GetOpenDocumentByIndex(IMFDocument**, index)

MCCOMErr MCCOMAPI GetOpenDocumentByIndex(IMFDocument document, uint32 index)**

GetOpenDocumentByIndex() gets the document specified by the index. <zot needs description>

Parameter ➦ **IMFDocument** document**
A pointer to the specified document. <zot needs description>.

➦ **uint32 index**
The index to a specific open document. <zot needs description>.

IShUtilities::GetLastOpenDocPath(TMCString&)

void MCCOMAPI GetLastOpenDocPath(TMCString& path)

GetLastOpenDocPath() gets the last open document path stored in path. <zot needs description>

Parameter ➦ **TMCString& path**
A reference to the string containing the path information. <zot needs description>.

IShUtilities::GetLastSaveDocPath(TMCString&)

void MCCOMAPI GetLastSaveDocPath(TMCString& path)

GetLastSaveDocPath() gets the last save document path stored in path. <zot needs description>

Parameter  **TMCString& path**
A reference to the string containing the path information. <zot needs description>.

IShUtilities::SetLastOpenDocPath(TMCString&)

void MCCOMAPI SetLastOpenDocPath(const TMCString& path)

SetLastOpenDocPath() sets the last open document path stored in path. <zot needs description>

Parameter  **const TMCString& path**
A reference to the string containing the path information. <zot needs description>.

IShUtilities::SetLastSaveDocPath(TMCString&)

void MCCOMAPI SetLastSaveDocPath(TMCString& path)

SetLastSaveDocPath() sets the last save document path stored in path. <zot needs description>

Parameter  **TMCString& path**
A reference to the string containing the path information. <zot needs description>.

IShUtilities::UseMMX()

boolean MCCOMAPI UseMMX()

UseMMX() returns True if the processor supports MMX and if we want to use it.

IShUtilities::IsPentiumProcessor()

boolean MCCOMAPI IsPentiumProcessor()

IsPentiumProcessor() returns True if the processor is a pentium family (P,Ppro,PII,PIII...).

IShUtilities::IsPentiumIIProcessor()

boolean MCCOMAPI IsPentiumIIProcessor()

IsPentiumIIProcessor() returns True if the processor is a PII.

IShUtilities::IsPentiumIIIProcessor()

boolean MCCOMAPI IsPentiumIIIProcessor()

IsPentiumIIIProcessor() returns True if the processor is a PIII.

IShUtilities::StringWidth(TMCTextStyle&, TMCString&)

int16 MCCOMAPI StringWidth(const TMCTextStyle& inStyle, const TMCString& inString)

StringWidth() returns the width needed for the specified string, using the specified text style.

Parameter ↗ **const TMCTextStyle& inStyle**
A reference to the style to use.

 ↗ **const TMCString& inString**
A reference to the string.

IShUtilities::GetTextMetrics(TMCTextStyle&, TMCTextMetrics&)

void MCCOMAPI GetTextMetrics(const TMCTextStyle& inStyle, TMCTextMetrics& outMetrics)

GetTextMetrics() updates the text metrics for the given text style. <zot needs more information. Please provide more details about the metrics or an example.>

Parameter ↗ **const TMCTextStyle& inStyle**
A reference to the style to use to calculate the text metrics.

 ↗ **TMCTextMetrics& outMetrics**
A reference to the metrics output by the function.

IShUtilities::GetCreator()

IDType MCCOMAPI GetCreator()

GetCreator() returns the ID type.

IShUtilities::SetCursor(int32)

void MCCOMAPI SetCursor(int32 cursorId)

SetCursor() changes the cursor to the specified cursor.

Parameter ↗ **int32 cursorID**
The id of the specified cursor.

IShUtilities::GetFontList(IMCFontList**)

void MCCOMAPI GetFontList(IMCFontList outFontList)**

GetFontList() updates the font list.

Parameter ↗ **IMCFontList** outFontList**
A pointer to the resulting FontList.

IShUtilities::CheckForUserCancel()

boolean MCCOMAPI CheckForUserCancel()

CheckForUserCancel() returns True if the user has canceled the operation.

IShUtilities::GetCurrentModifiers()

const TMCModifiers& MCCOMAPI GetCurrentModifiers()

GetCurrentModifiers() returns a reference to the current modifier.

IShUtilities::CheckKeyDown(int16)

boolean MCCOMAPI CheckKeyDown(int16 inVirtualKey)

CheckKeyDown() returns True if the key specified by inVirtualKey is down.

Parameter

 **int16** *inVirtualKey*

The value of the key being checked.

Chapter 9 - 3D ISh* Reference

9.1 I3DShAtmosphericShader Interface

Interface ID: IID_I3DShAtmosphericShader

Interface file: I3DShEnvironment.h

Description

The **I3DShAtmosphericShader** interface lets you get some information about the atmospheric shader of the scene.

The atmospheric shader is used to create effects such as fog, clouds, etc. The atmospheric shader filters the light beams and can modify them at will.

I3DShAtmosphericShader::GetScene(I3DShScene**)

MCCOMAPI GetScene(I3DShScene** scene)

Call **GetScene()** to return the scene of the object.

Parameters

 **I3DShScene** scene**
Points to the scene of the object.

9.2 I3DShBackdrop Interface

Interface ID: IID_I3DShBackdrop

Interface file: I3DShEnvironment.h

Description

The **I3DShBackdrop** interface lets you get some information on the backdrop of the scene. <zot>Beef up description </zot>

Backdrops are image at infinity (behind every objects) in the axis of the camera. A backdrop can not be reflected on the objects.

I3DShBackdrop::GetScene(I3DShScene**)

MCCOMAPI GetScene(I3DShScene** scene)

Call **GetScene()** to return the scene of the object.

Parameters

 **I3DShScene** scene**
Points to the scene of the object.

9.3 I3DShBackground Interface

Interface ID: IID_I3DShBackground

Interface file: I3DShEnvironment.h

Description

The **I3DShBackground** interface lets you get some information on the background of the scene. <zot>Beef up description </zot>

Backgrounds are 3D environment that can be reflected on the 3D objects surfaces (like a sky).

I3DShBackground::GetScene(I3DShScene**)

MCCOMAPI GetScene(I3DShScene** scene)

Call **GetScene()** to return the scene of the object.

Parameters

 **I3DShScene** scene**
Points to the scene of the object.

9.4 I3DShCamera Interface

Interface ID: IID_I3DShCamera

Interface file: I3DShCamera.h

Description

The **I3DShCamera** interface is the shell interface for cameras.

I3DShCamera defines a camera as a tree element. It can return a I3DShTreeElement interface on demand.

Methods Index

```

“I3DShCamera::GetGlobalToCameraTransform(TTransform3d*)”
“I3DShCamera::Preset(uint32, const TVector3*, real)”
“I3DShCamera::Project3DTo2D(const TVector3*, TVector2*, real)”
“I3DShCamera::CreateRay(const TVector2*, TVector3*, TVector3*)”
“I3DShCamera::GetRenderUVBox(TBBox2d*, uint32, uint32, real)”
“I3DShCamera::SetCameraComponent(int32)”
“I3DShCamera::SetCameraComponent(ISHComponent*, ECloneMode)”
“I3DShCamera::GetCameraComponent(ISHComponent**)”
“I3DShCamera::PreProject3DTo2D(const TVector3&, TVector4&)”
“I3DShCamera::IsProjectionAffine(void)”
“I3DShCamera::GetStandardCameraInfo(TStandardCameraInfo&)”
“I3DShCamera::SetCanonicalCameraType(ECanonicalCameraType)”
“I3DShCamera::GetCanonicalCameraType()”
“I3DShCamera::SetProductionOffset(const TVector2&)”
“I3DShCamera::GetProductionOffset()”
“I3DShCamera::SetProductionZoom(real)”
“I3DShCamera::GetProductionZoom()”

```

I3DShCamera::GetGlobalToCameraTransform(TTransform3d*)

MCCOMErr GetGlobalToCameraTransform(TTransform3D* transform)

Gives the transformation from the Global Coordinate System to the Camera Coordinate System.

Parameters  **TTRANSFORM3D*** *transform*
transform is a reference to the transformation data. Copy its contents to an internal variable.

Return Value MC_S_OK

I3DShCamera::Preset(uint32, const TVector3*, real)

MCCOMErr Preset(uint32 preset, const TVector3* aimedPoint, real distance)

Sets up preset values for the camera.

Parameters **uint32** *preset*
 Specifies which preset is being set in ECameraPreset: kReferencePreset, kTopPreset, kBottomPreset, kLeftPreset, bkRightPreset, or kFrontPreset.

TVector3* *aimedPoint*

The 3D location of the point aimed by the camera.

real *distance*

distance between the Camera and the *aimedPoint*.

Return Value MC_S_OK

I3DShCamera::Project3DTo2D(const TVector3*, TVector2*, real*)

boolean Project3DTo2D(const TVector3* *position*, TVector2* *resultScreenPosition*, real* *resultDistanceToScreen*)

Project3DTo2D() is called to project a 3D point on the screen.

Parameters  **const TVector3*** *position*
The 3D position in camera space.

 **TVector2*** *resultScreenPosition*
Projection on the screen.

 **real*** *resultDistanceToScreen*
The Z value.

Return Value

TRUE	The point is visible.
FALSE	The point is not visible.

I3DShCamera::CreateRay(const TVector2*, TVector3*, TVector3*)

boolean CreateRay(const TVector2* *screenPosition*, TVector3* *resultOrigin*, TVector3* *resultDirection*)

CreateRay() is called when a ray (i.e. a straight line) from a screen position needs to be created.

Parameters  **const TVector2*** *screenPosition*
Position on the screen; a relative parameter between -1 and 1.

 **TVector3*** *resultOrigin*
resultOrigin must return the same screen point, in 3D, in the Global Coordinates System. You need to use the transformation provided by SetTransform() to perform those calculations.

 **TVector3*** *resultDirection*
resultDirection must return the direction vector of the straight line. This must be a unit vector.

Return Value

TRUE	A ray could be calculated.
FALSE	Calculation was not possible.

You must always return true to be able to ray-trace an entire image.

I3DShCamera::GetRenderUVBox(TBBox2d*, uint32, uint32, real)

MCCOMErr GetRenderUVBox(TBox2D* uvBox, uint32 width, uint32 height, real pixelAspectRatio)

Gives the uvBox of the productionFrame given the width and height, and pixel aspect ratio of the screen.

Parameters

-  **TBBox2D*** *uvBox*
uvBox of the productionFrame.
-  **Uint32** *width*
The screen width.
-  **Uint32** *height*
The screen height.
-  **real** *pixelAspectRatio*
The screen pixel aspect ratio.

Return Value MC_S_OK

I3DShCamera::SetCameraComponent(int32)

MCCOMErr SetCameraComponent(int32 kindID)

<zot>Description here</zot>

Parameters

-  **int32** *kindID*
<zot>Description here</zot>

Return Value MC_S_OK

I3DShCamera::SetCameraComponent(IShComponent*, ECloneMode)

MCCOMErr SetCameraComponent(IShComponent * cameraComponent, ECloneMode cloneMode)

<zot>Description here</zot>

Parameters

-  **IShComponent*** *cameraComponent*
<zot>Description here</zot>
- ECloneMode** *cloneMode*
<zot>Description here</zot>

Return Value MC_S_OK

I3DShCamera::GetCameraComponent(IShComponent**)

MCCOMErr GetCameraComponent(IShComponent ** cameraComponent)

<zot>Description here</zot>

Parameters  **IShComponent** cameraComponent**
 <zot>Description here</zot>

Return Value MC_S_OK

I3DShCamera::PreProject3DTo2D(const TVector3&, TVector4&)

boolean PreProject3DTo2D(const TVector3& cameraPosition, TVector4& projectedPosition)

PreProject3DTo2D() is called to project a 3D point on the screen.

Parameters  **const TVector3& cameraPosition**
 The 3D position of the camera.

 **TVector4& projectedPosition**
 Projection on the screen.

Return Value

TRUE	The point is visible.
FALSE	The point is not visible. It may be behind the camera.

I3DShCamera::IsProjectionAffine(void)

boolean IsProjectionAffine(void)

IsProjectionAffine() returns true if the w component from the projection is always 1.0.

Return Value

TRUE	w is always 1.0
FALSE	w is not always 1.0

I3DShCamera::GetStandardCameraInfo(TStandardCameraInfo&)

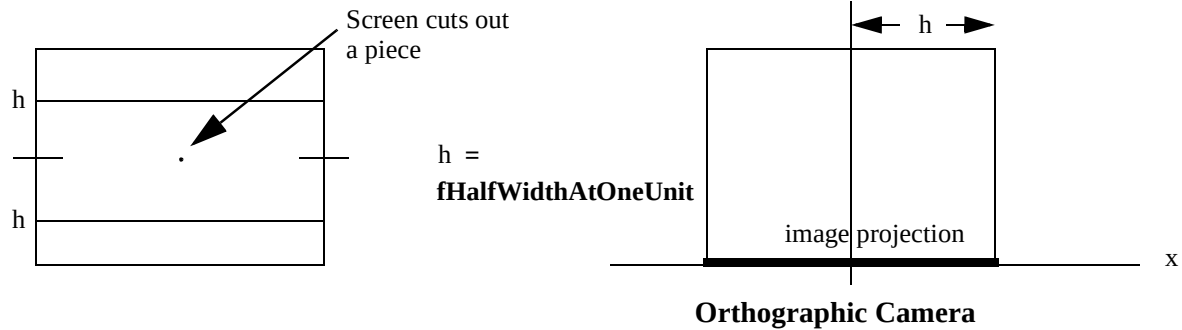
void GetStandardCameraInfo(TStandardCameraInfo& cameraInfo)

GetStandardCameraInfo() is used to return standard camera info used by the interactive renderer.

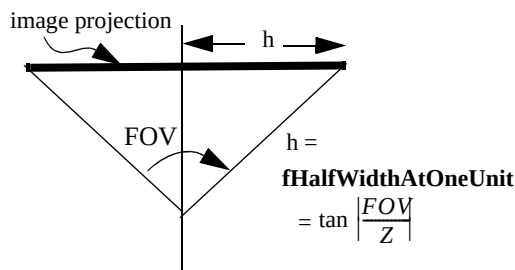
TStandardCameraInfo has the following fields:

- **EProjectionType fProjectionType** - pick the type of camera that you are most closely approximating between conical (perspective) and isometric (orthographic)
- **ECanonicalCameraType fCanonicalCameraType** - internal use only (ignore)
- **real fHalfWidthAtOneUnit** - the amount of camera space units at one unit from the camera location. In the case of an orthographic camera, this represent the zoom factor. For the perspective camera, this represents the field of view (FOV), or angle of the most extreme rays that are in the image. Any rays outside of the FOV are not represented in the image. (See diagrams below for graphical description.)
- **boolean flsMatching** - indicates whether the standard camera info is an exact or approximate conversion. For example, a spherical lens may only approximate itself with one of the types.

As you can see from the diagram below, for an orthographic camera, a large **fHalfWidthAtOneUnit** means a zoom out.



As shown below, with a perspective camera, a large **fHalfWidthAtOneUnit** means the FOV is large, like a wide angle lens, and the image is zoomed out.



h	θ	$Z\theta=FOV$
1	45	90
0.5	26	52
2	63	126

$$FOV = Z \tan^{-1}(h)$$

Parameters  **TStandardCameraInfo** & *cameraInfo*
The camera info.

I3DShCamera::SetCanonicalCameraType(ECanonicalCameraType)

void SetCanonicalCameraType(ECanonicalCameraType canonicalCameraType)

Chooses between canonical (standard) cameras (default cameras).

Parameters **ECanonicalCameraType** *canonicalCameraType*
 kCanonicalCameraType_NotCanonical
 kCanonicalCameraType_Left
 kCanonicalCameraType_Right
 kCanonicalCameraType_Top
 kCanonicalCameraType_Bottom
 kCanonicalCameraType_Front
 kCanonicalCameraType_Back

I3DShCamera::GetCanonicalCameraType()

ECanonicalCameraType GetCanonicalCameraType()

Returns the canonical camera type or kCanonicalCameraType_NotCanonical if this camera is not canonical.

Return Value The Canonical Camera Type.

I3DShCamera::SetProductionOffset(const TVector2&)

void SetProductionOffset(const TVector2& inOffset)

Sets the camera screen offset

Parameters  **TVECTOR2& inOffset**
The offset value.

I3DShCamera::GetProductionOffset()

TVector2 GetProductionOffset()

Gets the camera screen offset.

Return Value The production Offset.

I3DShCamera::SetProductionZoom(real)

void SetProductionZoom(real zoom)

Sets the camera screen zoom value.

Parameters **real zoom**
The zoom value.

I3DShCamera::GetProductionZoom()

real GetProductionZoom()

Gets the camera screen zoom value.

Return Value The Zoom value.

9.5 I3DShConstraint

Interface ID: IID_I3DShConstraint

Interface File: I3DShConstraint.h

Description

I3DShConstraint provides constraints between Tree Elements. A constraint, known as a link in RDS, defines motion links between Tree Elements. It allows you to define “mechanical” constraints (degrees of freedom) between one Tree Element and its father.

zot there are no methods to document in I3DShConstraint.h as of 11/22/99.

9.6 I3DShEnvironment Interface

Interface ID: IID_I3DShEnvironment

Interface file: I3DShEnvironment.h

Description

The **I3DShEnvironment** interface lets you get some information on the environment of the scene.
<zot>Beef up description </zot>

Methods Index

“I3DShEnvironment::SegmentFilter(const TVector3& ,const TVector3&, TMCColorRGB&, boolean)”
 “I3DShEnvironment::SegmentFilter(const TVector3&,const TVector3&,TMCColorRGB&)”
 “I3DShEnvironment::DirectionFilter(const TVector3 ,const TVector3&, TMCColorRGB&)”
 “I3DShEnvironment::GetBackdropColor(const TVector2&,boolean&,const TBBBox2D&, TMCColor-
 RGB&)”
 “I3DShEnvironment::GetEnvironmentColor(const TVector3&, boolean&, TMCColorRGB&)”
 “I3DShEnvironment::HasAtmosphere()”
 “I3DShEnvironment::HasBackdrop()”
 “I3DShEnvironment::HasBackground()”

I3DShEnvironment::SegmentFilter(const TVector3& ,const TVector3&, TMCColorRGB&, boolean)

MCCOMErr MCCOMAPI SegmentFilter(const TVector3 &beg,const TVector3 &end,TMCColorRGB &inOutFilter, boolean isShadowCasting)

Call **SegmentFilter()** to get the color of a ray when you know the color at one point of the ray. For example, you may know the color of an object where the ray hits and you want to get the color seen because the ray may go thru a fog or a volume primitive.

zot need to say **more** about the differences between the two SegmentFilter functions.

Parameters

- ✚ **const TVector3& beg**
The point where you want to get the color.
- ✚ **const TVector3& end**
The point where you know the color.
- ✚ **TMCColorRGB& inOutFilter**
Gives the color of the ray at point *end*, you get back the color at point *beg* of the ray.
- ✚ **boolean isShadowCasting**
Returns True if the object casts shadows.

I3DShEnvironment::SegmentFilter(const TVector3&,const TVector3&,TMCColorRGB&)

MCCOMErr MCCOMAPI SegmentFilter(const TVector3 &beg,const TVector3 &end,TMCColorRGB &inOutFilter)

Call **SegmentFilter()** to get the color of a ray when you know the color at one point of the ray. For example, you may know the color of an object where the ray hits and you want to get the color seen because the ray may go thru a fog or a volume primitive.

Parameters

- **const TVector3& beg**
The point where you want to get the color.
- **const TVector3& end**
The point where you know the color.
- **TMCColorRGB& inOutFilter**
Gives the color of the ray at point *end*, you get back the color at point *beg* of the ray.

I3DShEnvironment::DirectionFilter(const TVector3 ,const TVector3&, TMCColorRGB&)

MCCOMErr MCCOMAPI DirectionFilter(const TVector3 &origin,const TVector3 &direction, TMCColorRGB &inOutFilter)

Call **I3DShEnvironment::DirectionFilter** to get the color of a ray that reaches no object.

Parameters

- **const TVector3& origin**
The point where the ray starts.
- **const TVector3& direction**
The direction of the ray.
- **TMCColorRGB& inOutFilter**
Gives the color of the background, and you get back the resulting color.

I3DShEnvironment::GetBackdropColor(const TVector2&,boolean&,const TBBBox2D&, TMCColorRGB&)

MCCOMErr MCCOMAPI GetBackdropColor(const TVector2 &screenPoint,boolean &fullAreaDone,const TBBBox2D &screenMinMax, TMCColorRGB &resultColor)

GetBackdropColor() is called to get the backdrop color at a given point on the screen.

Parameters

- **const TVector2& screenPoint**
The screen point in **zot Local coordinates? Global coordinates?**
- **boolean& fullAreaDone**
Return True if the full area is covered.
- **TBBBox2D& screenMinMax**
The boundaries (minimum and maximum values) of the *fullAreaDone* range of values.

→ **TMCColorRGB& color**
Return the calculated color here.

I3DShEnvironment::GetEnvironmentColor(const TVector3&, boolean&, TMCColorRGB&)

MCCOMErr MCCOMAPI GetEnvironmentColor(const TVector3 &direction,boolean &fullAreaDone,TMCColor-
RGB &resultColor)

Call **I3DShEnvironment::GetEnvironmentColor** to get the color of the background.

Parameters

→ **const TVector3& direction**
The direction of the ray.

→ **boolean& fullAreaDone**
Return True if the full area is covered.

→ **TMCColorRGB& color**
Return the calculated color here.

I3DShEnvironment::HasAtmosphere()

boolean MCCOMAPI HasAtmosphere()

Call **HasAtmosphere()** to find out if the current Environment has an atmospheric effect or not. Returns True if it has an atmospheric effect.

I3DShEnvironment::HasBackdrop()

boolean MCCOMAPI HasBackdrop()

Call **HasBackdrop()** to find out if the current Environment has a backdrop effect or not. Returns True if it has a backdrop effect.

I3DShEnvironment::HasBackground()

boolean MCCOMAPI HasBackground()

Call **HasBackground()** to find out if the current Environment has a background effect or not. Returns True if it has a background effect.

9.7 I3DShExternalPrimitive Interface

Interface ID: IID_I3DShExternalPrimitive

Interface file: I3DShObject.h

Description

I3DShExternalPrimitive is a host for external primitives. It points to an “**I3DShPrimitiveComponent Interface**” that points to the External primitive itself. To get to the external primitive itself call **GetPrimitiveComponent()**. To get the component part, query the **I3DShPrimitiveComponent** interface, and call **I3DShPrimitiveComponent::GetExternalPrimitive**.

Method Index

“I3DShExternalPrimitive::SetPrimitiveComponent(ISHComponent*, ECloneMode)”

“I3DShExternalPrimitive::GetPrimitiveComponent(ISHComponent**)”

I3DShExternalPrimitive::SetPrimitiveComponent(ISHComponent*, ECloneMode)

MCCOMErr SetPrimitiveComponent(ISHComponent * primitiveComponent, ECloneMode cloneMode)

<zot>Need method description</zot>

Parameter  **I3DShComponent*** *primitiveComponent*
primitiveComponent **zot needs description.**

 **ECloneMode** *cloneMode*
cloneMode specifies whether or not animation information is included? **zot needs description.**

I3DShExternalPrimitive::GetPrimitiveComponent(ISHComponent**)

MCCOMErr GetPrimitiveComponent(ISHComponent ** primitiveComponent)

<zot>Need method description</zot> **GetPrimitiveComponent()** gets the component part of an external primitive.

Parameter  **I3DShComponent*** *primitiveComponent*
primitiveComponent **zot needs description.**

9.8 I3DShObject Interface

Interface ID: IID_I3DShObject

Interface file: I3DShObject.h

Description

I3DShObject contains the geometry and shading information of objects. There are two types of Objects: Primitives and Scenes. They are kept in a list within a scene called the objects list. An object can be instantiated as many times as needed in a scene.

Method Index

```

“I3DShObject::SetName(const TMCString&)”
“I3DShObject::GetName(TMCString&)”
“I3DShObject::GetInstancesCount()”
“I3DShObject::GetInstanceArray(TMCCountedPtrArray<I3DShInstance>&)”
“I3DShObject::GetUVSpaceCount()”
“I3DShObject::GetUVSpace(uint32, UVSpaceInfo*)”
“I3DShObject::GetScene(I3DShScene**)”
“I3DShObject::IsInfiniteObject()”
“I3DShObject::GetBoundingBox(TBBox3D&)”
“I3DShObject::GetFacetBoundingBox(TBBox3D&)”
“I3DShObject::Clone(I3DShObject**, ECloneMode) const”
“I3DShObject::GetEditor()”
“I3DShObject::GetObjectID()”

```

I3DShObject::SetName(const TMCString&)

MCCOMErr SetName(const TMCString& name)

SetName() sets the name of the object. This is the same call as **I3DShMaster::SetName**<zot>Need Cross-ref</zot>.

Parameters  **TMCString& name**
The name of the object.

Return Value MC_S_OK

I3DShObject::GetName(TMCString&)

MCCOMErr GetName(TMCString& name)

GetName() gets the name of the object. This is the same call as **I3DShMaster::GetName**<zot>Need Cross-Ref</zot>.

Parameters  **TMCString& name**
The name of the object.

Return Value MC_S_OK

I3DShObject::GetInstancesCount()

uint32 GetInstancesCount()

GetInstancesCount() returns the number of instances of the object. This is the same call as **I3DShMaster::GetInstancesCount**<zot>Need Cross-Ref</zot>.

Return Value The number of instances of the object.

I3DShObject::GetInstanceArray(TMCCountedPtrArray<I3DShInstance>&)

void GetInstanceArray(TMCCountedPtrArray<I3DShInstance>& instances)

GetInstanceArray() builds an array of all the instances of an object. This is the same call as **I3DShMaster::GetInstanceArray**<zot>Need Cross-Ref</zot>.

Parameters  **TMCCountedPtrArray<I3DShInstance>& instances**
An array of all instances of the object.

I3DShObject::GetUVSpaceCount()

uint32 GetUVSpaceCount()

GetUVSpaceCount() returns the number of UV spaces for an object. If 0 is returned only projection mapping can be performed on the object.

Return Value The number of UV Spaces for an object.

I3DShObject::GetUVSpace(uint32, UVSpaceInfo*)

virtual MCCOMErrMCCOMAPI GetUVSpace(uint32 uvSpaceID, UVSpaceInfo* uvSpaceInfo)

Get information on a specific UV Space

Parameters  **uint32 uvSpaceId**
UV Space ID, starting at 0, up to GetUVSpaceCount()-1.

 **UVSpaceInfo uvSpaceInfo**
Returned data of this UVSpace (you provide a pointer on the storage struct).

I3DShObject::GetScene(I3DShScene**)

virtual voidMCCOMAPI GetScene(I3DShScene scene)**

Returns the scene of the object.

Parameters  **I3DShScene scene**
Returns the scene.

I3DShObject::IsInfiniteObject()

virtual boolean MCCOMAPI IsInfiniteObject()

IsInfiniteObject() returns TRUE if the object is infinite (like the infinite plane).

I3DShObject::GetBoundingBox(TBBox3D&)

virtual void MCCOMAPI GetBoundingBox(TBBox3D& bbox)

Gets the bounding box of the object (in local coordinates).

Parameters

➞ **TBBox3D** *bbox*

Returns the coordinates for the bounding box.

I3DShObject::GetFacetBoundingBox(TBBox3D&)

virtual void MCCOMAPI GetFacetBoundingBox(TBBox3D &bbox, TMatrix33 &mappingTransform, void *listener, uint32 matrixChangedPhase)

[The bounding box calculated here exactly fits the facets of the object in a given direction. The FacetBounding boxes are cached inside the Object to reduce calculation time. You don't have to compare mappingTransform (TMatrix33) in the look-up table. A last hit is provide inside the GetFacetBoundingBox method to make the Get on same listener very fast. Each time the BoundingBox of the object changed , the list is flushed.

Parameters

➞ **TBBox3D&** *bbox*

Returns the coordinates for the bounding box.

➞ **TMatrix33&** *mappingTransform*

zot needs description

➞ **void*** *listener*

Accelerates the Get.

➞ **uint32** *matrixChangedPhase*

zot needs description.

I3DShObject::Clone(I3DShObject**, ECloneMode) const

virtual MCCOMErr MCCOMAPI Clone(I3DShObject** clone, ECloneMode cloneMode) const

<zot>Clones an object.

Parameters

➞ **I3DShObject**** *clone*

Returns the cloned object.

➞ **ECloneMode** *cloneMode*

Specifies the clone mode (with or without the animation data).

I3DShObject::GetEditor()

virtual int32 MCCOMAPI GetEditor()

GetEditor() returns the ID of the Editor (i.e. the module) capable of editing this object in the Modeling room.

I3DShObject::GetObjectID()

virtual int32MCCOMAPI GetObjectID()

GetObjectID() returns the id of an object. In the case of a primitive, it returns the class id. For a Scene, it returns 'sc '.

9.9 I3DShPatchArray

Interface ID: IID_I3DShPatchArray

Interface File: I3DShObject.h

Description

I3DShPatchArray is an array of cubic 4 x 4 Bezier patches. A Patch Array is a Primitive. Therefore it supports the **I3DShObject** and **I3DShPrimitive** Interfaces.

The "U" direction is the direction of the U axis of the UV Space. Same thing for the V direction. For example, do not confuse the uIndex, which is an integer describing the index of a point in the array along the U direction, and the uValue at that index, which is the U value in terms of UV Space for all 3D points at the uIndex index.

Using Polygon Arrays or Patch Arrays when possible is better than using Polygon Lists because it is easier to calculate a UV Space and perform Normal smoothing. There are even calls to do all that for you !

Method Index

```

“I3DShPatchArray::Init(uint32, uint32)”
“I3DShPatchArray::SetPatch(uint32, uint32, TPatch3D*)”
“I3DShPatchArray::GetPatch(uint32, uint32, TPatch3D*)”
“I3DShPatchArray::MeshSetU(uint32, real)”
“I3DShPatchArray::MeshSetV(uint32, real)”
“I3DShPatchArray::MeshGetU(uint32, real*)”
“I3DShPatchArray::MeshGetV(uint32, real*)”
“I3DShPatchArray::MeshCalcUV()”
“I3DShPatchArray::GetSize(uint32&, uint32&)”
“I3DShPatchArray::BuildFacetMesh(real, FacetMesh**)”

```

I3DShPatchArray::Init(uint32, uint32)

MCCOMErr Init(uint32 uCount, uint32 vCount)

Initializes a patch array.

Parameters

-  **uint32 uCount**
Number of patches in the U direction.
-  **uint32 vCount**
Number of patches in the V direction.

Return Value MC_S_OK

I3DShPatchArray::SetPatch(uint32, uint32, TPatch3D*)

MCCOMErr SetPatch(uint32 uIndex, uint32 vIndex, TPatch3D* patch)

Use **I3DShPatchArray::SetPatch** to set Patch values in the Patch Array.

Parameters	✚ uint32 uIndex Patch index in the U direction, between 0 and uCount-1. ✚ uint32 vIndex Patch index in the V direction, between 0 and vCount-1. ✚ TPatch3D* patch Points to the values for this patch. See the description of PATCH3D for more details (it is an array of 16 3D Points). The values are copied in the Patch Array internal data.
Return Value	MC_S_OK
Comments	Use this call to set patches one after the other in the Patch Array. See the "3D Database Overview" chapter for more details on Patch Arrays, especially on the conditions to have Geometric Continuity across patches.

I3DShPatchArray::GetPatch(uint32, uint32, TPatch3D*)

MCCOMErr GetPatch(uint32 uIndex, uint32 vIndex, TPatch3D* patch)

Parameters	✚ uint32 uIndex Patch index in the U direction, between 0 and uCount-1 ✚ uint32 vIndex Patch index in the V direction, between 0 and vCount-1 ✚ TPatch3D* patch Points to where to return the values for this patch. See the description of PATCH3D for more details (it is an array of 16 3D Points). The values are copied from the Patch Array internal data.
Return Value	MC_S_OK

I3DShPatchArray::MeshSetU(uint32, real)

MCCOMErr MeshSetU(uint32 uIndex, real uValue)

Use **I3DShPatchArray::MeshSetU** to set the U coordinate of all points at uIndex.

Parameters	✚ uint32 uIndex Index in the U direction. Index in between 0 and uCount-1 if the Array is open in the U direction, or 0 and uCount if it is closed. ✚ real uValue The U value for all points at uIndex. This value is between 0 and 1.
Return Value	MC_S_OK

I3DShPatchArray::MeshSetV(uint32, real)

MCCOMErr MeshSetV(uint32 vIndex, real vValue)

Use **I3DShPatchArray::MeshSetV** to set the V coordinate of all points at vIndex.

- Parameters**
-  **uint32 vIndex**
Index in the V direction. Index in between 0 and vCount-1 if the Array is open in the V direction, or 0 and vCount if it is closed.
 -  **real vValue**
The V value for all points at vIndex. This value is between 0 and 1.

Return Value MC_S_OK

I3DShPatchArray::MeshGetU(uint32, real*)

MCCOMErr MeshGetU(uint32 uIndex, real* uValue)

Use **I3DShPatchArray::MeshGetU** to get the U coordinate of all points at uIndex.

- Parameters**
-  **uint32 uIndex**
Index in the U direction. Index in between 0 and uCount-1 if the Array is open in the U direction, or 0 and uCount if it is closed.
 -  **real* uValue**
Where to copy the U value for all points at uIndex. This value is between 0 and 1.

Return Value MC_S_OK

I3DShPatchArray::MeshGetV(uint32, real*)

MCCOMErr MeshGetV(uint32 vIndex, real* vValue)

Use **I3DShPatchArray::MeshGetV** to get the V coordinate of all points at vIndex.

- Parameters**
-  **uint32 vIndex**
Index in the V direction. Index in between 0 and vCount-1 if the Array is open in the V direction, or 0 and vCount if it is closed.
 -  **real* vValue**
Where to copy the V value for all points at vIndex. This value is between 0 and 1.

Return Value MC_S_OK

I3DShPatchArray::MeshCalcUV()

MCCOMErr MeshCalcUV()

Call **I3DShPatchArray::CalcUV** if you want to have all the UV values calculated for you. You must call this procedure if you did not set any UV values using

I3DShPatchArray::MeshSetU and **I3DShPatchArray::MeshSetV**.

Return Value MC_S_OK

I3DShPatchArray::GetSize(uint32&, uint32&)

MCCOMErr GetSize(uint32& uCount, uint32& vCount)

Gets the size of the patch array in the U and V directions.

Parameters ➦ **uint32** *uCount*
Where to copy the number of patches in the U direction.

➦ **uint32** *vCount*
Where to copy the number of patches in the V direction.

Return Value MC_S_OK

I3DShPatchArray::BuildFacetMesh(real, FacetMesh)**

virtual MCCOMErr MCCOMAPI BuildFacetMesh(real lod, FacetMesh outMesh)**

This is a low-level procedure that always returns the facet mesh at the requested LOD (different than when using the **I3DShPrimitive::GetFacetMesh()** system of sampling LODs values.)

Parameters ➦ **real** *lod*
The level of detail.

➦ **FacetMesh**** *outMesh*
The created mesh that represents the same object than the patch array but with facets.

Return Value MC_S_OK

9.10 I3DShPatchList

Interface ID: IID_I3DShPatchList

Interface File: I3DShObject.h

Description

A Patch List is simply a bag of bicubic patches in unsorted order. Of course, it will not return patches, only facets: use **I3DShPrimitive::EnumPatches**.

A Patch List is a Primitive. Therefore it supports the **I3DShObject** and **I3DShPrimitive** Interfaces.

A Patch List can have an additional feature: a UV Space.

When you build a Patch List, it is recommended to provide support for UV Space in the patches. Supporting UV Space will enable Direct Parametric Mapping, which is the best technique to apply 2D Shaders on a surface. Otherwise, Projection Mapping will have to be used, which may result in some texture deformations on the objects.

Method Index

```
"I3DShPatchList::Init(boolean)"
"I3DShPatchList::AddPatch(TPatch3D*)"
"I3DShPatchList::GetIndPatches(uint32)"
"I3DShPatchList::CountPatches()"
"I3DShPatchList::SetHasUVSpace(boolean)"
"I3DShPatchList::SetUVSpace(UVSpaceInfo*)"
"I3DShPatchList::GetUVSpace(uint32, UVSpaceInfo*)"
"I3DShPatchList::GetUVSpaceCount()"
"I3DShPatchList::BuildFacetMesh(real, FacetMesh**)"
```

I3DShPatchList::Init(boolean)

MCCOMErr Init(boolean hasUV)

I3DShPatchList::Init must be right after creating a Patch List object.

Parameters  **boolean** *hasUV*
TRUE if this object supports a UV Space

Return Value MC_S_OK

I3DShPatchList::AddPatch(TPatch3D*)

MCCOMErr AddPatch(TPatch3D* patch)

Use **I3DShPatchList::AddPatch** to add patches to a Patch List object one after the other.

Parameters	 TPatch3D* patch Points to the patch to add to the Patch List. The patch content is copied, I3DShPatchList::AddPatch does not take property of the patch pointer.
Return Value	MC_S_OK
Comments	Fill-in the patch data according to the flags provided in I3DShPatchList::Init: give a UV Space info as necessary. See the PATCH3D structure for more information. I3DShPatchList::AddPatch fails if running out of RAM.

I3DShPatchList::GetIndPatches(uint32)

TPatch3D*MCCOMAPI GetIndPatches(uint32 index)

Provides acces to the patches by index.

Parameters	 uint32 index The index of the patch you want to acces, <i>index</i> is 0-based.
Return Value	The index th patch of the list.

I3DShPatchList::CountPatches()

uint32 CountPatches()

Counts the number of patches in the patch list.

Return Value	The number of patches in the patchlist as an uint32.
---------------------	--

I3DShPatchList::SetHasUVSpace(boolean)

MCCOMErr SetHasUVSpace(boolean hasUV)

If false, will remove all previous UV Space definitions.

Parameters	 boolean hasUV the flag for HasUVSpace.
Return Value	MC_S_OK

I3DShPatchList::SetUVSpace(UVSpaceInfo*)

MCCOMErr SetUVSpace(UVSpaceInfo* uvSpaceInfo)

Set a UV Space. The UV space will be added if it was not previously defined. See the Data Structure reference section for more information on the *UVSpaceInfo* structure.

Parameters	 UVSpaceInfo* uvSpaceInfo Pointer to the UVSpaceInfo structure for defining the UV space.
Return Value	MC_S_OK

I3DShPatchList::GetUVSpace(uint32, UVSpaceInfo*)

MCCOMErr GetUVSpace(uint32 uvSpaceID, UVSpaceInfo* uvSpaceInfo)

Returns the *UVSpaceInfo* structure for the corresponding *uvSpaceID*. See the Data Structure reference section for more information on the *UVSpaceInfo* structure.

Parameters ↗ **uint32 uvSpaceID**
The ID of the UV Space to retrieve.

↗ **UVSpaceInfo* uvSpaceInfo**
The *UVSpaceInfo* structure.

Return Value MC_S_OK

I3DShPatchList::GetUVSpaceCount()

uint32MCCOMAPI GetUVSpaceCount()

Gets the count of UV spaces.

Return Value The count of UV spaces as an uint32.

I3DShPatchList::BuildFacetMesh(real, FacetMesh**)

MCCOMErrMCCOMAPI BuildFacetMesh(real lod, FacetMesh outMesh)**

This is a low-level procedure that always returns the facet mesh at the requested LOD (different than when using the **I3DShPrimitive::GetFacetMesh()** system of sampling LODs values.)

Parameters ↗ **real lod**
The level of detail.

↗ **FacetMesh** outMesh**
The created mesh that represents the same object than the patch array but with facets.

Return Value MC_S_OK

9.11 I3DShPolygonArray

Interface ID: IID_I3DShPolygonArray

Interface File: I3DShObject.h

Description

A Polygon Array is an array of 3D points and Normals that describe a surface. See the "3D Database Overview" chapter for more details on Polygon Arrays.

Polygon Arrays are facet-based 3D Objects.

A Polygon Array is a Primitive. Therefore it supports the **I3DShObject** and **I3DShPrimitive** Interfaces.

The "U" direction is the direction of the U axis of the UV Space. Same thing for the V direction. For example, do not confuse the uIndex, which is an integer describing the index of a point in the array along the U direction, and the uValue at that index, which is the U value in terms of UV Space for all 3D points at the uIndex index.

Using Polygon Arrays or Patch Arrays when possible is better than using Polygon Lists because it is easier to calculate a UV Space and perform Normal smoothing. There are even calls to do all that for you !

Method Index

```
"I3DShPolygonArray::Init(uint32, uint32, boolean, boolean)"
"I3DShPolygonArray::MeshSetVertex(uint32, uint32, TVector3*, TVector3*)"
"I3DShPolygonArray::MeshGetVertex(uint32, uint32, TVector3*, TVector3*)"
"I3DShPolygonArray::MeshSetU(uint32, real)"
"I3DShPolygonArray::MeshSetV(uint32, real)"
"I3DShPolygonArray::MeshGetU(uint32, real*)"
"I3DShPolygonArray::MeshGetV(uint32, real*)"
"I3DShPolygonArray::MeshCalcUV()"
"I3DShPolygonArray::MeshCalcNormals(boolean)"
"I3DShPolygonArray::GetSize(uint32&, uint32&, boolean&, boolean&)"
```

I3DShPolygonArray::Init(uint32, uint32, boolean, boolean)

MCCOMErr Init(uint32 uCount, uint32 vCount, boolean uClosed, boolean vClosed)

Call **I3DShPolygonArray::Init** just after creating a **I3DShPolygonArray** object to initialize all internal data structures.

Parameters

uint32 uCount
Number of vertices in the U direction.

uint32 vCount
Number of vertices in the V direction.

boolean *uClosed*

TRUE if the Array is closed in the U direction.

boolean *vClosed*

TRUE if the Array is closed in the V direction.

Return Value MC_S_OK

Comments **I3DShPolygonArray::Init** allocates the Polygon Array in RAM so you can access its points through **I3DShPolygonArray::SetVertex** and **I3DShPolygonArray::GetVertex**.

I3DShPolygonArray::MeshSetVertex(uint32, uint32, TVector3*, TVector3*)

MCCOMErr MeshSetVertex(uint32 uIndex, uint32 vIndex, TVector3* vertex, TVector3* normal)

Use **I3DShPolygonArray::SetVertex** to set the Vertex and the Normal at a specific point of the Array. Normal can be NULL.

Parameters

 **uint32** *uIndex*

Index in the U direction. Index in between 0 and uCount-1.

 **uint32** *vIndex*

Index in the V direction. Index in between 0 and vCount-1

 **TVector3*** *vertex*

Points to the Vertex coordinates. The values will be copied in the internal Polygon Array data.

 **TVector3*** *normal*

Points to the Normal coordinates. The values will be copied in the internal Polygon Array data. The pointer can be NULL if you do not want to provide Normals now, and intend to calculate them later using **I3DShPolygonArray::CalcNormals**.

Return Value MC_S_OK

Comments It is always better to provide the Normals yourself. Note that because there is only one Normal at each point of the Array, a Array surface is always smooth. If you do not want it to be smooth, do not provide Normals and call **I3DShPolygonArray::CalcNormals(FALSE)**.

I3DShPolygonArray::MeshGetVertex(uint32, uint32, TVector3*, TVector3*)

MCCOMErr MeshGetVertex(uint32 uIndex, uint32 vIndex, TVector3* vertex, TVector3* normal)

Use **I3DShPolygonArray::GetVertex** to get the Vertex and the Normal at a specific point of the Array.

Parameters	 uint32 <i>uIndex</i> Index in the U direction. Index in between 0 and uCount-1.  uint32 <i>vIndex</i> Index in the V direction. Index in between 0 and vCount-1.  TVector3* <i>vertex</i> Where to return the Vertex coordinates. The values will be copied from the internal Polygon Array data into this place.  TVector3* <i>normal</i> Where to return the Normal coordinates. The values will be copied from the internal Polygon Array data into this place, unless <i>normal</i> is NULL.
Return Value	MC_S_OK

I3DShPolygonArray::MeshSetU(uint32, real)

MCCOMErr MeshSetU(uint32 uIndex, real uValue)

Use **I3DShPolygonArray::MeshSetU** to set the U coordinate of all points at uIndex.

Parameters	 uint32 <i>uIndex</i> Index in the U direction. Index in between 0 and uCount-1 if the Array is open in the U direction, or 0 and uCount if it is closed.  real <i>uValue</i> The U value for all points at uIndex. This value is between 0 and the U value of the last vertex.
Return Value	MC_S_OK
Note	Note that all points at the same uIndex have the same texture U value. Same thing on V. The UV Space is "parallel" to the array of points: points at the same uIndex have the same uValue.

I3DShPolygonArray::MeshSetV(uint32, real)

MCCOMErr MeshSetV(uint32 vIndex, real vValue)

Use **I3DShPolygonArray::MeshSetV** to set the V coordinate of all points at vIndex.

Parameters	 uint32 <i>vIndex</i> Index in the V direction. Index in between 0 and vCount-1 if the Array is open in the V direction, or 0 and vCount if it is closed.  real <i>vValue</i> The V value for all points at vIndex. This value is between 0 and the V value of the last vertex.
-------------------	---

Return Value MC_S_OK

Note Note that all points at the same vIndex have the same texture V value. Same thing on U. The UV Space is "parallel" to the array of points: points at the same vIndex have the same vValue.

I3DShPolygonArray::MeshGetU(uint32, real*)

MCCOMErr MeshGetU(uint32 uIndex, real* uValue)

Use **I3DShPolygonArray::MeshGetU** to get the U coordinate of all points at uIndex.

Parameters

-  **uint32 uIndex**
Index in the U direction. Index in between 0 and uCount-1 if the Array is open in the U direction, or 0 and uCount if it is closed.
-  **real* uValue**
Where to copy the U value for all points at uIndex. This value is between 0 and the U value of the last vertex.

Return Value MC_S_OK

Note Note that all points at the same uIndex have the same texture U value. Same thing on V. The UV Space is "parallel" to the array of points.

I3DShPolygonArray::MeshGetV(uint32, real*)

MCCOMErr MeshGetV(uint32 vIndex, real* vValue)

Use **I3DShPolygonArray::MeshGetU** to get the V coordinate of all points at vIndex.

Parameters

-  **uint32 vIndex**
Index in the V direction. Index in between 0 and vCount-1 if the Array is open in the V direction, or 0 and vCount if it is closed.
-  **real* vValue**
Where to copy the V value for all points at vIndex. This value is between 0 and the V value of the last vertex.

Return Value MC_S_OK

Note Note that all points at the same vIndex have the same texture V value. Same thing on U. The UV Space is "parallel" to the array of points.

I3DShPolygonArray::MeshCalcUV()

MCCOMErr MeshCalcUV()

Call **I3DShPolygonArray::CalcUV** if you want to have all the UV values calculated for you.

Return Value MC_S_OK

Comments You must call this procedure if you did not set any UV values using **I3DShPolygonArray::MeshSetU** and **I3DShPolygonArray::MeshSetV**.

I3DShPolygonArray::MeshCalcNormals(boolean)

MCCOMErr MeshCalcNormals(boolean smooth)

Use **I3DShPolygonArray::CalcNormals** to have all Normals calculated for you.

Return Value MC_S_OK

Comments You must call this procedure if you did not set any Normals using **I3DShPolygonArray::SetVertex**.

I3DShPolygonArray::GetSize(uint32&, uint32&, boolean&, boolean&)

MCCOMErr GetSize(uint32& uCount, uint32& vCount, boolean& uClosed, boolean& vClosed)

Retrives the information passed to **I3DShPolygonArray::Init**.

Parameters

-  **uint32& uCount**
-  **uint32& vCount**
-  **boolean& uClosed**
-  **boolean& vClosed**

Return Value MC_S_OK

9.12 I3DShPolygonList Interface

Interface ID: IID_I3DShPolygonList

Interface File: I3DShObject.h

Description

A Polygon List is simply a bag of triangular facets in unsorted order. For better efficiency you should use the **FacetMeshPrimitive**.

A Polygon List is a Primitive. Therefore it supports the **I3DShObject** and **I3DShPrimitive** Interfaces.

A Polygon List can have two additional features: Normals and a UV Space.

When you build a Polygon List, it is highly recommended to provide Normals in the facets. This will ensure best and smooth shading. If Normals cannot be provided, then you must call **I3DShPolygonList::CalcNormals** to reconstruct the Normals. However this is a costly and long process that is not 100% reliable.

If you can, it is also recommended to provide support for UV Space in the facets. Supporting UV Space will enable Direct Parametric Mapping, which is the best technique to apply 2D Shaders on a surface. Otherwise, Projection Mapping will have to be used, which may result in some texture deformations on the objects.

Method Index

```

“I3DShPolygonList::Init(boolean, boolean)”
“I3DShPolygonList::PreAllocateFacets(uint32)”
“I3DShPolygonList::AddFacet(TFacet3D*)”
“I3DShPolygonList::CountFacets()”
“I3DShPolygonList::CalcNormals(real)”
“I3DShPolygonList::GetIndFacets(uint32)”
“I3DShPolygonList::SetHasUVSpace(boolean)”
“I3DShPolygonList::SetUVSpace(UVSpaceInfo*)”
“I3DShPolygonList::GetUVSpace(uint32, UVSpaceInfo*)”
“I3DShPolygonList::GetUVSpaceCount()”

```

I3DShPolygonList::Init(boolean, boolean)

MCCOMErr Init(boolean hasNormals, boolean hasUV)

I3DShPolygonList::Init must be called right after creating a Polygon List object.

Parameters

 **boolean** *hasNormals*
Indicates if the polygon list has surface normals.

 **boolean** *hasUV*
Indicates if the polygon list has UV values.

Return Value MC_S_OK

I3DShPolygonList::PreAllocateFacets(uint32)

MCCOMErr PreAllocateFacets(uint32 nbFacets)

Call **I3DShPolygonList::PreAllocateFacets** to pre-allocate the memory space needed for *nbFacets* facets.

Parameters  **uint32** *nbFacets*
Number of facets that will be added later with I3DShPolygonList::AddFacet.

Return Value MC_S_OK

Comments Use this call if you know in advance how many facets you will build and want to make sure that there will be enough RAM to store them. This allows the code to fail immediately if the 3D Shell is running out of RAM, instead of failing later after many facets may already be calculated and stored. It is also more efficient, because the data block will be allocated only once instead of being regularly re-allocated and copied. The use of this call is optional.

I3DShPolygonList::AddFacet(TFacet3D*)

MCCOMErr AddFacet(TFacet3D* facet)

Use **I3DShPolygonList::AddFacet** to add facets to a Polygon List object one after the other.

Parameters  **TFacet3D*** *facet*
Points to the facet to add to the Polygon List. The facet content is copied, I3DShPolygonList::AddFacet does not take property of the facet pointer

Return Value MC_S_OK

Comments Fill-in the facet data according to the flags provided in I3DShPolygonList::Init: give Normals and UV Space info as necessary. See the FACET3D structure in the "Database Overview" chapter for more information. I3DShPolygonList::AddFacet fails if running out of RAM (see I3DShPolygonList::PreAllocateFacets about memory management).

I3DShPolygonList::CountFacets()

uint32 CountFacets()

Returns the number of facets in the Polygon List.

Return Value The number of facets as an uint32.

I3DShPolygonList::CalcNormals(real)

MCCOMErr CalcNormals(real smoothAngle)

Calculates the surface normals with the given smooth angle.

Parameters  **real** *smoothAngle*
Angle under which smoothing should occur between facets. This angle is in degrees. 0° means no smoothing, 90° means always smooth. 30° is a good default value.

Return Value MC_S_OK

Comments **I3DShPolygonList::CalcNormals** must be called if `hasNormals` was FALSE when calling **I3DShPolygonList::Init**.

I3DShPolygonList::GetIndFacets(uint32)

TFacet3D* GetIndFacets(uint32 index)

Use **I3DShPolygonList::GetIndFacets** to get a facet in the list.

Parameters  **uint32** *index*
Index of the facet, must be between 0 and **I3DShPolygonList::CountFacets** - 1

Return Value A **TFacet3D*** referenced by the index.

I3DShPolygonList::SetHasUVSpace(boolean)

MCCOMErr SetHasUVSpace(boolean hasUV)

If *hasUV* is false **SetHasUVSpace()** removes all previous UV Space definitions.

I3DShPolygonList::SetUVSpace(UVSpaceInfo*)

MCCOMErr SetUVSpace(UVSpaceInfo* uvSpaceInfo)

Set a UV Space. The UV space will be added if it was not previously defined. See the Data Structure reference section for more information on the *UVSpaceInfo* structure.

Parameters  **UVSpaceInfo*** *uvSpaceInfo*
Pointer to the *UVSpaceInfo* structure for defining the UV space.

Return Value MC_S_OK

I3DShPolygonList::GetUVSpace(uint32, UVSpaceInfo*)

MCCOMErr GetUVSpace(uint32 uvSpaceID, UVSpaceInfo* uvSpaceInfo)

Returns the *UVSpaceInfo* structure for the corresponding *uvSpaceID*. See the Data Structure reference section for more information on the *UVSpaceInfo* structure.

Parameters  **uint32** *uvSpaceID*
The ID of the UV Space to retrieve.

⇒ **UVSpaceInfo*** *uvSpaceInfo*
The *UVSpaceInfo* structure.

Return Value MC_S_OK

I3DShPolygonList::GetUVSpaceCount()

uint32MCCOMAPI **GetUVSpaceCount()**
Gets the count of UV spaces.

Return Value The count of UV spaces as an uint32.

9.13 I3DShPostRenderer Interface

Interface ID : IID_I3DShPostRenderer

Interface file : I3DShPostRenderer.h

Description

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.

Method Index

“I3DShPostRenderer::SetScene(I3DShScene*)”
 “I3DShPostRenderer::GetBufferNeeds(uint32&)”
 “I3DShPostRenderer::PrepareDraw(ISHRasterLayer*, const TMCRect&, IShChannel*, I3DShScene*, TBox2D&, TBox2D&)”
 “I3DShPostRenderer::DrawRect(const TMCRect&, const TChannelDataBucket*, TBox2D)”
 “I3DShPostRenderer::FinishDraw()”
 “I3DShPostRenderer::BeginRender(MicroTick, MicroTick, uint32, uint32, boolean)”
 “I3DShPostRenderer::EndRender()”

I3DShPostRenderer::SetScene(I3DShScene*)

MCCOMErr SetScene(I3DShScene* scene)

<zot>Need description here</zot> Sets a scene as the current scene.

Parameter  **I3DShScene* scene**
 A pointer to the scene.

Return Value MC_S_OK

I3DShPostRenderer::GetBufferNeeds(uint32&)

void GetBufferNeeds(uint32& needs)

GetBufferNeeds() is used to indicate which G-Buffers are needed.

Parameter  **uint32_t** *needs*

Return a set of flags for the informations you need as found in the table below:

Name	Value	Description
k32Distance	0	Distance from the camera to the object visible at a point in the image.
k32PositionX	1	X position of a point if not the background
k32PositionY	2	Y position of a point if not the background
k32PositionZ	3	Z position of a point if not the background
k32NormalX	4	X direction of an object's surface Normal if not the background
k32NormalY	5	Y direction of an object's surface Normal if not the background
k32NormalZ	6	Z direction of an object's surface Normal if not the background
k32Alpha	7	The alpha channel with black (0) as the background
k32Index	8	UV space index
k32SurfaceU	9	U value of the UV space of an object at a point
k32SurfaceV	10	V value of the UV space of an object at a point
k32DiffuseColor	11	Corresponds to the Color channel of the object's shader
k32SpecularColor	12	Corresponds to the Highlight channel of the object's shader
k32SpecularSizeValue	13	Corresponds to the Shininess channel of the object's shader
k32AmbientValue	14	<zot></zot>
k32LambertValue	15	Corresponds to the<zot></zot> channel of the object's shader
k32ReflectionColor	16	Corresponds to the Reflection channel of the object's shader
k32TransparencyColor	17	Corresponds to the Transparency channel of the object's shader
k32RefractionValue	18	Corresponds to the Refraction channel of the object's shader
k32GlowColor	19	Corresponds to the Glow channel of the object's shader

I3DShPostRenderer::PrepareDraw(IShRasterLayer*, const TMCRect&, IShChannel*, I3DShScene*, TBox2D&, TBox2D&)

MCCOMErr PrepareDraw(IShRasterLayer* input, const TMCRect& bounds, IShChannel* buffers[], I3DShScene* scene, I3DShCamera* renderingCamera, TBox2D& uvBox, TBox2D& productionFrame)

PrepareDraw() is called prior to rendering each block of the image.

Parameters	 IShRasterLayer* <i>input</i> <zot>desc.</zot>  const TMCRect& <i>bounds</i> <zot>desc.</zot>  IShChannel* <i>buffers[]</i> This is a table of pointers to the buffers requested by GetBufferNeeds(). Each element in the buffers array corresponds to the value of each buffer (buffer[0] for distance through buffer[19] for glow color). See the table in GetBufferNeeds() for the complete list.  I3DShScene* <i>scene</i> This is a pointer to the scene structure.  I3DShCamera* <i>renderingCamera</i> This is a pointer to the rendering camera.  TBBox2D& <i>uvBox</i> Screen coordinates of the image.  TBBox2D& <i>productionFrame</i> Screen coordinates of the production frame.
Return Value	MC_S_OK

I3DShPostRenderer::DrawRect(const TMCRect&, const TChannelDataBucket*, TBBox2D)

MCCOMErr DrawRect(const TMCRect& outputRect, const TChannelDataBucket* output[], TBBox2D& uvBox)
DrawRect() is called for each tile of the image.

Parameters	 TMCRect& <i>outputRect</i> <zot>desc.</zot>screen coordinates for the output  TChannelDataBucket* <i>output[]</i> <zot>desc.</zot>  TBBox2D& <i>uvBox</i> Screen coordinates of the image.
-------------------	---

I3DShPostRenderer::FinishDraw()

MCCOMErr FinishDraw()
FinishDraw() is called for cleanup at the end of the post-rendering.

I3DShPostRenderer::BeginRender(MicroTick, MicroTick, uint32, uint32, boolean)

MCCOMErrMCCOMAPI BeginRender(MicroTick beginTime, MicroTick endTime, uint32 framePerSeconds, int32 nbrFrames, boolean isRenderMovie)

BeginRender() is called to start the post-rendering.

Parameters

- **MicroTick** *beginTime*
Specifies the starting time of the movie for the render.
- **MicroTick** *endTime*
Specifies the ending time of the movie for the render.
- **uint32** *framePerSeconds*
Specifies the frames per second.
- **uint32** *nbrFrames*
Specifies the frame rate.
- **boolean** *isRenderMovie*
Is True when rendering a movie.

I3DShPostRenderer::EndRender()

MCCOMErr EndRender()

EndRender() is called for cleanup at the end of the post-rendering.

9.14 I3DShMasterShader Interface

Interface ID : IID_I3DShMasterShader

Interface file : I3DShShader.h

Description

zot add description

Method Index

“I3DShMasterShader::SetName(const TMCString&)”
 “I3DShMasterShader::GetName(TMCString&)”
 “I3DShMasterShader::SetShader(I3DShShader*, ECloneMode)”
 “I3DShMasterShader::GetShader(I3DShShader**)”
 “I3DShMasterShader::Clone(I3DShMasterShader**, ECloneMode)”
 “I3DShMasterShader::GetScene(I3DShScene**)”
 “I3DShMasterShader::IsDefaultMaster()”

I3DShMasterShader::SetName(const TMCString&)

MCCOMErr MCCOMAPI SetName(const TMCString& name)

<zot>Sets the name of a master shader.

Parameters  **TMCString& name**
The name of the master shader.

I3DShMasterShader::GetName(TMCString&)

MCCOMErr MCCOMAPI GetName(TMCString& name)

<zot>Gets the name of a master shader.

Parameters  **TMCString& name**
The name of the master shader.

I3DShMasterShader::SetShader(I3DShShader*, ECloneMode)

MCCOMErr MCCOMAPI SetShader(I3DShShader* shader, ECloneMode cloneMode)

SetShader() specifies which shader is set.

Parameter  **I3DShShader shader**
Specifies which shader to set.

 **ECloneMode cloneMode**
Specifies whether to clone with or without animation information.

I3DShMasterShader::GetShader(I3DShShader**)

MCCOMErr MCCOMAPI GetShader(I3DShShader shader)**

GetShader() gets a shader.

Parameters  **I3DShShader** shader**
Specifies which shader to get.

I3DShMasterShader::Clone(I3DShMasterShader**, ECloneMode)

MCCOMErr MCCOMAPI Clone(I3DShMasterShader clone, ECloneMode cloneMode)**

Clone() clones a master shader.

Parameters  **I3DShMasterShader** clone**
Specifies which shader to get.

 **ECloneMode cloneMode**
Specifies whether to clone with or without animation information.

I3DShMasterShader::GetScene(I3DShScene**)

MCCOMErr MCCOMAPI GetScene(I3DShScene scene)**

GetScene() returns a scene.

Parameters  **I3DShScene** scene**
Specifies the scene.

I3DShMasterShader::IsDefaultMaster()

boolean MCCOMAPI IsDefaultMaster()

IsDefaultMaster() specifies whether a master shader is the default master shader.

9.15 I3DShPrimitive Interface

Interface ID: IID_I3DShPrimitive

Interface file: I3DShObject.h

Description

A Primitive is a simple geometric object. **I3DShPrimitive** contains the methods needed to create the geometry.

Method Index

“I3DShPrimitive::IsPatchBased()”
 “I3DShPrimitive::EnumPatches(EnumPatchesCallback, void*, boolean)”
 “I3DShPrimitive::ChangedData()”
 “I3DShPrimitive::GetFMesh(real, FacetMesh**)”
 “I3DShPrimitive::IsVolumic()”
 “I3DShPrimitive::UV2XYZ(const TVector2&, int16, TVector3&, TVector3*)”

I3DShPrimitive::IsPatchBased()

boolean IsPatchBased()

This method simply indicates if an object has patches or is comprised only of facets.

Return Value

True	The Primitive is Patch based.
False	The Primitive is not Patch based.

I3DShPrimitive::EnumPatches(EnumPatchesCallback, void*, boolean)

MCCOMErr EnumPatches(EnumPatchesCallback callback, void* privData, boolean surfaceClosed)

I3DExGeometricPrimitive::EnumPatches is called by the 3D Shell to get the primitive as a list of bicubic patches.

Parameters

 **EnumPatchesCallback** *callback*

The callback procedure should be called for each Patch generated by the Primitive. Its interface is as follows:

```
typedef void (*EnumPatchesCallback) (TPatch3D* patch, void* privData);
```

The 3D Shell will not keep the *patch* pointer. It will copy its content.

 **void* privData**

Points to private data from the Shell. Pass this pointer to *callback*.

 **boolean surfaceClosed**

Returns True when the surface is closed . This helps set the backface mask correctly.

Return Value

MC_S_OK	Patches were generated correctly.
MC_E_NOTIMPL	Patches can not be generated.

I3DShPrimitive::ChangedData()

MCCOMErr ChangedData()

<zot>below is the comment from the code - needs clarification</zot>

Call this if you changed some data of the Primitive so the Shell hears about it.

I3DShPrimitive::GetFMesh(real, FacetMesh**)

MCCOMErr GetFMesh(real lod, FacetMesh mesh)**

<zot>Needs description here</zot>

GetFMesh() is an alternate method for creating a facet mesh. It is more efficient than using **EnumFacets()** as it allows for calculating less facets in conditions that require less detail.

Parameters

→ **real lod**

The level of detail.

→ **FacetMesh** mesh**

A FacetMesh defining the primitive.

I3DShPrimitive::IsVolumic()

boolean IsVolumic()

<zot>below is the comment from the code - needs clarification</zot>

"Volumic" primitives are special primitives such as particle systems, fog, fountain, etc.

I3DShPrimitive::UV2XYZ(const TVector2&, int16, TVector3&, TVector3*)

boolean UV2XYZ(const TVector2& uv, int16 uvSpaceID, TVector3& thePos3D, TVector3* normal)

<zot>Needs descrition of how this method differs from I3DExGeometricPrimitive::UV2XZ</zot>

I3DShPrimitive::UV2XYZ is called when a point on the surface defined only by its uv coordinates needs to be converted in x, y and z values. This call is optional. If this call is not implement then the default rules will apply by using the patches or the facets of the Primitive. When possible implement **I3DShPrimitive::UV2XYZ** for optimal performance and accuracy.

Parameters

→ **const TVector2& uv**

The given uv values.

→ **int16 uvSpaceID**

The identifier of the UV space.

⇒ **TVector3& thePos3D**

The resulting 3D coordinates in the object Local Coordinates System.

⇒ **TVector3* normal**

<zot>Need description.</zot>

Return Value

True	<zot>Need description.</zot>
False	<zot>Need description.</zot>

9.16 I3DShPrimitiveComponent Interface

Interface ID: IID_I3DShPrimitiveComponent

Interface File: I3DShObject.h

Description

I3DShPrimitiveComponent points to the External primitive itself. See the “**I3DShExternalPrimitive Interface**”.

I3DShPrimitiveComponent provides access to the owner primitive from the component. This primitive interface then permits access to the scene and other elements through **QueryInterface()**. It also gives a way to get to the External Primitive attached to it such as I3DExGeometricPrimitive. IShComponent is also supported.

Method Index

“I3DShPrimitiveComponent::GetPrimitive(I3DShPrimitive**)”

“I3DShPrimitiveComponent::GetExternalPrimitive(IMCUnknown**)”

I3DShPrimitiveComponent::GetPrimitive(I3DShPrimitive**)

void GetPrimitive(I3DShPrimitive primitive)**

This provides access to the parent **I3DShExternalPrimitive**.

<zot>Need method description</zot>

Parameters  **I3DShPrimitive** primitive**
 <zot>Need Description</zot>

I3DShPrimitiveComponent::GetExternalPrimitive(IMCUnknown**)

MCCOMErr GetExternalPrimitive(IMCUnknown extPrimitive)**

Provides access to the external primitive itself (**I3DExGeometricPrimitive**, etc.).

<zot>Need method description</zot>

Parameters  **IMCUnknown** extPrimitive**
 <zot>Need Description</zot>

9.17 I3DShShader Interface

Interface ID : IID_I3DShShader

Interface file : I3DShShader.h

Description

With the exceptions of **IsConstant()** and **AssignsChannels()** the methods in **I3DShShader** call the **I3DShShader** method with the passed arguments.

Method Index

“I3DShShader::Clone(I3DShShader**, ECloneMode) const”
“I3DShShader::IsConstant()”
“I3DShShader::AssignsChannels()”
“I3DShShader::GetShadingFlags(ShadingFlags&)”
“I3DShShader::DoShade(ShadingOut&, ShadingIn&)”
“I3DShShader::GetValue(real&, boolean, ShadingIn&)”
“I3DShShader::GetColor(TMCColorRGB&, boolean, ShadingIn&)”
“I3DShShader::GetVector(TVector3&, ShadingIn&)”
“I3DShShader::GetBump(TVector3&, ShadingIn&)”
“I3DShShader::GetImplementedOutput()”
“I3DShShader::GetOriginalParametricTextureMap(uint32&,uint32&, IShTextureMap**)”

I3DShShader::Clone(I3DShShader**, ECloneMode) const

voidMCCOMAPI Clone(I3DShShader clone, ECloneMode cloneMode) const**

<zot>Clones a shader.

Parameters

-  **I3DShShader** clone**
Returns the cloned shader.
-  **ECloneMode cloneMode**
Specifies the clone mode (with or without the animation data).

I3DShShader::IsConstant()

boolean IsConstant()

If a shader is constant a single value is applied to the entire surface. This is used for basic components such as the color shader.

Return Value

True	The shader value is constant across the surface.
False	The shader value is not constant across the surface.

I3DShShader::AssignsChannels()

boolean AssignsChannels()

This function indicates if the **DoShade()** method is implemented by the shader.

Return Value

True	DoShade() is implemented.
False	DoShade() is not implemented.

I3DShShader::GetShadingFlags(ShadingFlags&)

MCCOMErr GetShadingFlags(ShadingFlags& theFlags)

GetShadingFlags specifies which parameters the shader will need to perform shading calculations. If the shader is a *constant shader* which values does not depend on any of the *ShadingIn* parameters then it should set the *fCallOnce* flag in *theFlags* to true. See the description of the “ShadingFlags” structure for more details.

Parameter

 **ShadingFlags& theFlags**
Points to a *ShadingFlags* record containing all the flags.

Return Value **MC_S_OK**

I3DShShader::DoShade(ShadingOut&, ShadingIn&)

MCCOMErr DoShade(ShadingOut& result, ShadingIn& shadingIn)

I3DShShader::DoShade has total control of the input and output parameters. See the description of “ShadingFlags” and “ShadingIn” for more details.

Parameters

 **ShadingOut& result**
Reference to the complete input/output shading data structure.

 **ShadingIn& shadingIn**
The Shading element associated with the shader.

Return Value

MC_S_OK	DoShade() is implemented by this shader.
MC_E_NOTIMPL	DoShade() is not implemented by this shader.

Comments

I3DShShader::GetValue(real&, boolean, ShadingIn&)

MCCOMErr GetValue(real& result, boolean fullArea, ShadingIn& shadingIn)

GetValue returns a value between 0.0 and 1.0. Typical examples are noise functions, fractal functions, gradients, etc. See the description of “ShadingIn” for more details.

Parameters

→ **real& result**

Returns a result between 0.0 and 1.0.

→ **boolean fullArea**

<zot> needs description </end zot> guess: Returns True if the full area uses the same value.

→ **ShadingIn& shadingIn**

Reference to the input shading data structure. The shader uses the parameters requested by “I3DShShader::GetShadingFlags(ShadingFlags&)”.

Return Value

MC_S_OK	GetValue() is implemented by this shader.
MC_E_NOTIMPL	GetValue() is not implemented by this shader.

I3DShShader::GetColor(TMCCColorRGB&, boolean, ShadingIn&)

MCCOMErr GetColor(TMCCColorRGB& result, boolean fullArea, ShadingIn& shadingIn)

GetColor returns a color. Typical examples are constant colors or texture maps. See the description of “ShadingIn” for more details.

Parameters

→ **TMCCColorRGB& result**

Return the color values.

→ **boolean fullArea**

<zot> needs description </end zot>

→ **ShadingIn & in**

Points to the input shading data structure. The shader uses the parameters requested by “I3DShShader::GetShadingFlags(ShadingFlags&)”.

Return Value

MC_S_OK	GetColor() is implemented by this shader.
MC_E_NOTIMPL	GetColor() is not implemented by this shader.

Comments

I3DShShader::GetVector(TVector3&, ShadingIn&)

MCCOMErr GetVector(TVector3& result, ShadingIn& shadingIn)

I3DShShader::GetVector is reserved for a future use.

I3DShShader::GetBump(TVector3&, ShadingIn&)

MCCOMErr GetBump(TVector3& result, ShadingIn& shadingIn)

Returns the bump channel only (usually implemented with DoShade).

Parameters

→ **TVector3& result**

Return the bump values.

→ **ShadingIn &shadingIn**

Points to the input shading data structure. The shader uses the parameters requested by “I3DShShader::GetShadingFlags(ShadingFlags&)”.

I3DShShader::GetImplementedOutput()

uint32 GetImplementedOutput()

GetImplementedOutput is called by the 3D Shell to determine which methods the Shader actually implements. Return only one of the following values if you implement only one of the methods. If more than one is implemented, such as the *ShadedArea* implementation, then return the result as the OR of the two values (i.e. return `kUsesGetValueArea | kUsesGetValue`).

Return Value

Shading Defines

Constant	Value	Meaning
kUsesGetValue	1	I3DShShader::GetValue is implemented
kUsesGetColor	2	I3DShShader::GetColor is implemented
kUsesGetVector	4	I3DShShader::GetVector is implemented
kUsesDoShade	16	I3DShShader::DoShade is implemented
kUsesGetValueArea	32	I3DShShader::GetValue with <i>ShadedArea</i> is implemented
kUsesGetColorArea	64	I3DShShader::GetColor with <i>ShadedArea</i> is implemented
kUsesGetVectorArea	128	I3DShShader::GetVector with <i>ShadedArea</i> is implemented
kUsesDoShadeArea	256	I3DShShader::DoShade with <i>ShadedArea</i> is implemented

I3DShShader::GetOriginalParametricTextureMap(uint32&,uint32&, IShTextureMap**)

MCCOMErrMCCOMAPI GetOriginalParametricTextureMap(const uint32& uvSpaceID, const uint32& channelIndex, IShTextureMap outMap)**

Gets the original parametric texture map.

Parameters

→ **uint32& uvSpaceID**

Returns the UV Space ID .

→ **uint32& channelIndex**

Return the index to bump values.

→ **IShTextureMap** outMap**

Points to the output shading data structure.

9.18 I3DShTreeElement Interface

Interface ID : IID_I3DShTreeElement

Interface file : I3DShTreeElement.h

Description

I3DShTreeElement is the abstract class of all elements in a tree (called "tree elements"). These elements can be groups, object instances, lights sources, cameras, and so on. You can always get this interface from a **I3DShGroup**, **I3DShInstance**, etc.

Method Index

```

“I3DShTreeElement::SetScene(I3DShScene*)”
“I3DShTreeElement::GetScene(I3DShScene**)”
“I3DShTreeElement::GetSceneNoAddRef()”
“I3DShTreeElement::SetName(const TMCString&)”
“I3DShTreeElement::GetName(TMCString&)”
“I3DShTreeElement::GetFirst(I3DShTreeElement**)”
“I3DShTreeElement::GetLast(I3DShTreeElement**)”
“I3DShTreeElement::GetLeft(I3DShTreeElement**)”
“I3DShTreeElement::GetRight(I3DShTreeElement**)”
“I3DShTreeElement::GetTop(I3DShTreeElement**)”
“I3DShTreeElement::Unlink(I3DShTreeElement**, RelinkTreeElementInfo*)”
“I3DShTreeElement::Relink(RelinkTreeElementInfo*)”
“I3DShTreeElement::InsertFirst(I3DShTreeElement*)”
“I3DShTreeElement::InsertLast(I3DShTreeElement*)”
“I3DShTreeElement::InsertLeft(I3DShTreeElement*)”
“I3DShTreeElement::InsertRight(I3DShTreeElement*)”
“I3DShTreeElement::GetHotPoint(TVector3&)”
“I3DShTreeElement::SetHotPoint(const TVector3&)”
“I3DShTreeElement::CenterElementOnHotPoint()”
“I3DShTreeElement::CenterHotPointOnElement()”
“I3DShTreeElement::TranslateHotPoint(TVector3&)”
“I3DShTreeElement::ComClone(I3DShTreeElement**, ECloneMode, boolean)”
“I3DShTreeElement:: InvalidateBoundingBox()”
“I3DShTreeElement::GetBoundingBoxWithoutLightsAndCameras(TBBox3D&)”
“I3DShTreeElement::GetBoundingBox(TBBox3D&, ETreeDeformBBoxModes)”
“I3DShTreeElement::CanBeDistorted()”
“I3DShTreeElement::BeginMove()”
“I3DShTreeElement::PostMoveChange(boolean)”
“I3DShTreeElement::EndMove(boolean)”
“I3DShTreeElement::GetXYZScaling(TVector3&)”
“I3DShTreeElement::SetXYZScaling(TVector3&)”
“I3DShTreeElement::GetNbrFreedom()”
“I3DShTreeElement::GetFreedomRange(int16, real&, real&)”
“I3DShTreeElement::GetFreedomTransform(TMatrix33&, TVector3&)”
“I3DShTreeElement::GetFreedomDerivative(int16, TMatrix33&, TVector3&)”
“I3DShTreeElement::GetFreedomSecondDerivative(int16, int16, TMatrix33&, TVector3&)”
“I3DShTreeElement::IncrementFreedomValue(int16 , real*)”
“I3DShTreeElement::GetFullName(TMCString&)”
“I3DShTreeElement::InsertModifier(I3DShModifier* , uint32, ECloneMode)”
“I3DShTreeElement::RemoveModifier(uint32)”

```

```

“I3DShTreeElement::GetModifiersCount()”
“I3DShTreeElement::GetModifierByIndex(I3DShModifier**, uint32)”
“I3DShTreeElement::SetModifiers(const TMCCountedPtrArray<I3DShModifier>*, ECloneMode)”
“I3DShTreeElement::LinksChanged()”
“I3DShTreeElement::BeginGetRenderables(const TRenderableAndTfmArray*&, boolean)”
“I3DShTreeElement::EndGetRenderables()”
“I3DShTreeElement::GetConstraint(I3DShConstraint**)”
“I3DShTreeElement::SetConstraint(I3DShConstraint*, ECloneMode)”
“I3DShTreeElement::SetConstraintByID(int32)”
“I3DShTreeElement::GetDataComponentsCount()”
“I3DShTreeElement::GetDataComponentByIndex(I3DShDataComponent**, uint32)”
“I3DShTreeElement::SetDataComponentByID(I3DShDataComponent*, ECloneMode)”
“I3DShTreeElement::GetDataComponentByID(I3DShDataComponent**, IDType)”
“I3DShTreeElement::InsertDataComponent(I3DShDataComponent*)”
“I3DShTreeElement::InsertDataComponentByID(int32, int32)”
“I3DShTreeElement::RemoveDataComponent(uint32)”
“I3DShTreeElement::SetData(TDataKey, IMCUnknown*)”
“I3DShTreeElement::GetData(TDataKey, IMCUnknown**)”
“I3DShTreeElement::ReleaseData(TDataKey)”
“I3DShTreeElement::IsOpen()”
“I3DShTreeElement::GetEditor()”
“I3DShTreeElement::GetAnimationMethod(I3DShAnimationMethod**)”
“I3DShTreeElement::SetAnimationMethod(I3DShAnimationMethod*, ECloneMode, real)”
“I3DShTreeElement::GetLocalTreeTransform(TTreeTransform&)”
“I3DShTreeElement::SetLocalTreeTransform(const TTreeTransform&)”
“I3DShTreeElement::GetGlobalTreeTransform(TTreeTransform&)”
“I3DShTreeElement::SetGlobalTreeTransform(const TTreeTransform&, int16, boolean)”
“I3DShTreeElement::GetGlobalTreeTransform3D(TTreeTransform3D&)”
“I3DShTreeElement::SetGlobalTransform3D(const TTransform3D&, int16)”
“I3DShTreeElement::GetFatherGlobalTransform3D(TTransform3D&)”
“I3DShTreeElement::CloneSubTree(ECloneMode, I3DShTreeElement**)”
“I3DShTreeElement::GetUndoInfo(TTreeUndoInfo&)”
“I3DShTreeElement::SetFromUndoInfo(const TTreeUndoInfo&, const TTreeUndoInfo*)”
“I3DShTreeElement::GetTopIfConstrained(I3DShTreeElement**)”
“I3DShTreeElement::AddWireFrame(I3DExWireFrame*)”
“I3DShTreeElement::RemoveWireFrame(I3DExWireFrame*)”
“I3DShTreeElement::WireFrameArrayFree()”
“I3DShTreeElement::GetWireFrameArray(TMCClassArray<TWireFrameInfo>&)”
“I3DShTreeElement::IsPickable()”
“I3DShTreeElement::IsBBoxPickable()”
“I3DShTreeElement::RemoveConstraint(I3DShConstraint**)”
“I3DShTreeElement::RestoreConstraint(I3DShConstraint*)”
“I3DShTreeElement::GetExists()”
“I3DShTreeElement::SetExists(boolean)”
“I3DShTreeElement::GetIsShown()”
“I3DShTreeElement::SetIsShown(boolean)”
“I3DShTreeElement::SetShadowCasting(boolean)”
“I3DShTreeElement::SetShadowCatching(boolean)”
“I3DShTreeElement::GetShadowCasting()”
“I3DShTreeElement::GetShadowCatching()”
“I3DShTreeElement::GetAnimateTree()”
“I3DShTreeElement::SetAnimateTree(boolean)”

```

I3DShTreeElement::SetScene(I3DShScene*)

MCCOMErrMCCOMAPI SetScene(I3DShScene* scene)

SetScene() must be called after creating a Tree Element to set which scene it belongs to.

Parameters  **I3DShScene* scene**
The scene to which the tree element belongs.

I3DShTreeElement::GetScene(I3DShScene**)

MCCOMErrMCCOMAPI GetScene(I3DShScene scene)**

GetScene() gets the scene a tree element belongs to.

Parameters  **I3DShScene** scene**
Address of a counted pointer that is set to the scene to which the tree element belongs (the scene is addrefed).

I3DShTreeElement::GetSceneNoAddRef()

I3DShScene* MCCOMAPI GetSceneNoAddRef()

Returns a pointer to the scene to which the tree element belongs (without addrefing it).

I3DShTreeElement::SetName(const TMCString&)

MCCOMErr MCCOMAPI SetName(const TMCString& name)

Use **SetName()** to change the name of the tree element.

Parameters  **const TMCString& name**
The new the name (31 characters maximum.)

Comments The pointer is not kept; the string content it points to is copied into the I3DShTreeElement internal space.

I3DShTreeElement::GetName(TMCString&)

MCCOMErr MCCOMAPI GetName(TMCString& name)

Gets the name of a tree element.

Parameters  **TMCString& name**
Points to where to store the name (31 characters maximum.)

I3DShTreeElement::GetFirst(I3DShTreeElement**)

MCCOMAPI GetFirst(I3DShTreeElement)**

Used for tree navigation. Gets the first son of a tree element.

Parameter  **I3dShTreeElement****
Returns the first son, NULL if there is none.

I3DShTreeElement::GetLast(I3DShTreeElement**)

MCCOMAPI GetLast(I3DShTreeElement)**

Used for tree navigation. Gets the last son of a tree element

Parameter  **I3dShTreeElement****
Returns the last son, NULL if there is none.

I3DShTreeElement::GetLeft(I3DShTreeElement**)

MCCOMAPI GetLeft(I3DShTreeElement)**

Used for tree navigation. Gets the left brother of a tree element.

Parameter  **I3dShTreeElement****
Returns the left brother, NULL if there is none.

I3DShTreeElement::GetRight(I3DShTreeElement**)

MCCOMAPI GetRight(I3DShTreeElement)**

Used for tree navigation. Gets the right brother of a tree element.

Parameter  **I3dShTreeElement****
Returns the right brother, NULL if there is none.

I3DShTreeElement::GetTop(I3DShTreeElement**)

MCCOMAPI GetTop(I3DShTreeElement)**

GetTop() is used for tree navigation. Gets the father of a tree element.

Parameter  **I3dShTreeElement****
Returns the father, NULL if there is none.

I3DShTreeElement::Unlink(I3DShTreeElement**, RelinkTreeElementInfo*)

MCCOMErr MCCOMAPI Unlink(I3DShTreeElement treeElement, RelinkTreeElementInfo* outRelink=NULL)**

Unlink() unhooks a tree element from its tree. The whole sub-tree is unhooked along with it.

Parameters  **I3DShTreeElement** treeElement**
Specifies the element to unhook within the tree.

 **RelinkTreeElementInfo outRelink**
Returns the old father.

I3DShTreeElement::Relink(RelinkTreeElementInfo*)

MCCOMErr MCCOMAPI Relink(RelinkTreeElementInfo* inRelink)

Relink() rehooks a tree element based upon relink.

Parameters  **RelinkTreeElementInfo*** *inRelink*
Returns a pointer to the old father.

I3DShTreeElement::InsertFirst(I3DShTreeElement*)

MCCOMErr MCCOMAPI InsertFirst(I3DShTreeElement* newSon)

InsertFirst() inserts an element as the first son of the tree element.

Parameters  **I3DShTreeElement*** *newSon*
Points to the element to insert.

I3DShTreeElement::InsertLast(I3DShTreeElement*)

MCCOMErr MCCOMAPI InsertLast(I3DShTreeElement* newSon)

InsertLast() inserts an element as the last son of a tree element.

Parameters  **I3DShTreeElement*** *newSon*
Points to the element to insert.

I3DShTreeElement::InsertLeft(I3DShTreeElement*)

MCCOMErr MCCOMAPI InsertLeft(I3DShTreeElement* newBrother)

InsertLeft() inserts an element as the first left brother of a tree element.

Parameters  **I3DShTreeElement*** *newBrother*
Points to the element to insert.

I3DShTreeElement::InsertRight(I3DShTreeElement*)

MCCOMErr MCCOMAPI InsertRight(I3DShTreeElement* newBrother)

InsertLeft() inserts an element as the first right brother of a tree element.

Parameters  **I3DShTreeElement*** *newBrother*
Points to the element to insert.

I3DShTreeElement::GetHotPoint(TVector3&)

MCCOMErr MCCOMAPI GetHotPoint(TVector3& hotPoint)

GetHotPoint returns a hot point in local coordinates. The hot point is the point around which rotation and scaling occur.

Parameters  **TVector3&** *hotPoint*
Points to where to store the coordinates.

I3DShTreeElement::SetHotPoint(const TVector3&)

MCCOMErr MCCOMAPI SetHotPoint(const TVector3& hotPoint)

SetHotPoint sets a hot point in local coordinates. The hot point is the point around which rotation and

scaling occur.

Parameters  **const TVector3& hotPoint**
Points to where to store the coordinates.

Comments Changing the hot point position of a tree element changes the object global coordinates, and not the hot point position in global coordinates.

I3DShTreeElement::CenterElementOnHotPoint()

MCCOMErr MCCOMAPI CenterElementOnHotPoint()

CenterElementOnHotPoint() centers an element on a hot point, by moving the TreeElement so that the bounding box is centered on the hot point. The hot point is the point around which rotation and scaling occur.

I3DShTreeElement::CenterHotPointOnElement()

MCCOMErr MCCOMAPI CenterHotPointOnElement()

CenterHotPointOnElement() centers a hot point on an element, by moving the hot point to the center of the bounding box. The hot point is the point around which rotation and scaling occur.

I3DShTreeElement::TranslateHotPoint(TVector3&)

MCCOMErr MCCOMAPI TranslateHotPoint(TVector3& delta)

Translates the hot point of *delta*, in local coordinates. The hot point is the point around which rotation and scaling occur.

Parameters  **TVector3& delta**
Points to where to store the coordinates of the point *delta*.

I3DShTreeElement::ComClone(I3DShTreeElement**, ECloneMode, boolean)

MCCOMErr MCCOMAPI ComClone(I3DShTreeElement treeElement, ECloneMode cloneMode, boolean cloneAll)**

ComClone() . Duplicates a tree element. The new tree element is unhooked from the tree the original tree element is a member of, and becomes an independent branch.

Parameters  **I3DShTreeElement** treeElement**
Where to put the cloned tree element.

 **ECloneMode cloneMode**
Specifies whether to clone with or without animation information.

 **boolean cloneAll**
zot needs description.

I3DShTreeElement:: InvalidateBoundingBox()

MCCOMAPI InvalidateBoundingBox()

Invalidates the current bounding box. **zot** need more description

I3DShTreeElement::GetBoundingBoxWithoutLightsAndCameras(TBBox3D&)

MCCOMAPI GetBoundingBoxWithoutLightsAndCameras(TBBox3D& bbox)

GetBoundingBoxWithoutLightsAndCameras() gets the specified bounding box without lights and camera information. This is used to get a bounding box on only the geometric information in a tree hierarchy.

Parameters  **TBBox3D& bbox**
The coordinates of the bounding box.

I3DShTreeElement::GetBoundingBox(TBBox3D&, ETreeDeformBBoxModes)

MCCOMAPI GetBoundingBox(TBBox3D& bbox, ETreeDeformBBoxModes deformMode)

GetBoundingBox() gets the bounding box of a tree element.

Parameters  **TBBox3D& bbox**
The coordinates of the bounding box.

 **ETreeDeformBBoxModes deformMode**
Indicates how to compute the bbox when deformers are applied on the tree or its parents. Refer to the data structures chapter (zot xref) for more information.

I3DShTreeElement::CanBeDistorted()

boolean MCCOMAPI CanBeDistorted()

CanBeDistorted() returns True if the bounding box can be distorted.

I3DShTreeElement::BeginMove()

MCCOMErr MCCOMAPI BeginMove()

BeginMove() is used by the Shell to know which tree element will change to perform some optimization in the drawing of the scene. It must be called before you change the parameters of the tree element.

I3DShTreeElement::PostMoveChange(boolean)

void MCCOMAPI PostMoveChange(boolean invalidateDeformedGeometry = false)

PostMoveChange() tells the listeners that the tree moved. It is called by I3DShTreeElement::EndMove().

Parameters  **boolean invalidateDeformedGeometry**
Should the instances with deformers invalidate their deformed meshes (false by default).

I3DShTreeElement::EndMove(boolean)

void MCCOMAPI EndMove(boolean invalidateDeformedGeometry = false)

EndMove() is used by the Shell to know which tree element moved to perform some optimization in the drawing of the scene. It must be called after you change the parameters of the tree element.

Parameters ➦ **boolean** *invalidateDeformedGeometry*
See I3DShTreeElement::PostMoveChange().

I3DShTreeElement::GetXYZScaling(TVector3&)

MCCOMErr MCCOMAPI GetXYZScaling(TVector3& xyzScaling)

GetXYZScaling() is called to get the scaling in the X, Y, and Z directions of a tree element.

Parameters ➦ **TVector3&** *xyzScaling*
References the coordinates for the X, Y, and Z scaling.

I3DShTreeElement::SetXYZScaling(TVector3&)

MCCOMErr MCCOMAPI SetXYZScaling(TVector3& xyzScaling)

Call **SetXYZScaling()** to set the scaling in the X, Y, and Z directions of a tree element.

Parameters ➦ **TVector3&** *xyzScaling*
References the coordinates for the X, Y, and Z scaling.

I3DShTreeElement::GetNbrFreedom()

int16 MCCOMAPI GetNbrFreedom()

Call **GetNbrFreedom()** to get the number of degrees of freedom in a tree element. This is the total number of possible translations and rotations. A totally free tree element has 6 degrees of freedom. Inverse Kinematics uses this method to calculate movement.

I3DShTreeElement::GetFreedomRange(int16, real&, real&)

MCCOMErr MCCOMAPI GetFreedomRange(int16 index, real& min, real& max)

GetFreedomRange() returns the return the range of variation fo a given freedom. Inverse Kinematics uses this method to calculate movement.

Parameters ➦ **uint32** *index*
Select the freedom.

➦ **real** *min*
min is the minimum increment value.

➦ **real** *max*
max is the maximum increment value.

I3DShTreeElement::GetFreedomTransform(TMatrix33&, TVector3&)

MCCOMErr MCCOMAPI GetFreedomTransform(TMatrix33& RR,TVector3& TT)

GetFreedomTransform() gets the transformation of a tree element given by a method. Inverse Kinematics uses this method to calculate movement.

Parameters

→ **TMatrix33& RR**

Reference to the rotation matrix. See TMaxtrix33 in the data structures chapter.

→ **TVector3& TT**

Indicates a three dimensional vector for transformation **zot**. Refer to the data structures chapter (zot xref) for more information.

I3DShTreeElement::GetFreedomDerivative(int16, TMatrix33&, TVector3&)

MCCOMErr MCCOMAPI GetFreedomDerivative(int16 index,TMatrix33& RR,TVector3& TT)

GetFreedomDerivative() gets the derivative of a tree element given by a modifier. Inverse Kinematics uses this method to calculate movement. The derviative corresponds to the difference of freedom transformation when the given freedom value increases by 1. That information is used by the 3DShell to know which freedom value must be changed when the user drags the tree element.

Parameters

→ **int16 index**

Selects the freedom.

→ **TMatrix33& RR**

Reference to the rotation matrix. See TMaxtrix33 in the data structures chapter.

→ **TVector3& TT**

Reference to a three dimensional vector for transformation **zot**. Refer to the data structures chapter (zot xref) for more information.

I3DShTreeElement::GetFreedomSecondDerivative(int16, int16, TMatrix33&, TVector3&)

MCCOMErr MCCOMAPI GetFreedomSecondDerivative(int16 index1, int16 index2, TMatrix33& RR,TVector3& TT)

GetFreedomSecondDerivative() **zot** needs more information.

Parameters

→ **int16 index1**

zot needs description: Selects the first freedom?

→ **int16 index2**

zot needs description: Selects the second freedom?

→ **TMatrix33& RR**

Reference to the rotation matrix. See TMaxtrix33 in the data structures chapter.

→ **TVector3& TT**

Indicates a three dimensional vector for transformation **zot**. Refer to the data structures chapter (zot

xref) for more information.

I3DShTreeElement::IncrementFreedomValue(int16 , real*)

MCCOMErr MCCOMAPI IncrementFreedomValue(int16 index, real* value)

IncrementFreedomValue() increases the freedom value of a given freedom.

Parameters ↗ **int16** *index*
Selects the freedom.

↗ **real*** *value*
Points to the value to add to the freedom.

I3DShTreeElement::GetFullName(TMCString&)

MCCOMErr MCCOMAPI GetFullName(TMCString& fullName)

GetFullName() returns the path name of the tree element (nodes separated by kPathNameSeparator). This is the name of the tree element and all of the fathers starting from the universe.

Parameters ↗ **TMCString&** *fullName*
The full path name of the tree element.

I3DShTreeElement::InsertModifier(I3DShModifier* , uint32, ECloneMode)

MCCOMErr MCCOMAPI InsertModifier(I3DShModifier* modifier, uint32 index, ECloneMode cloneMode)

Inserts a modifier into the tree's modifier list at the given index.

Parameters ↗ **I3DShModifier*** *modifier*
The modifier to insert.

↗ **uint32** *index*
Where to insert the modifier.

↗ **ECloneMode** *cloneMode*
Currently unused.

I3DShTreeElement::RemoveModifier(uint32)

MCCOMErr MCCOMAPI RemoveModifier(uint32 index)

Removes the modifier at the given index from the tree's modifier list.

Parameters ↗ **uint32** *index*
The modifier to remove.

I3DShTreeElement::GetModifiersCount()

uint32 MCCOMAPI GetModifiersCount()

Return value The number of modifiers applied to the tree element.

I3DShTreeElement::GetModifierByIndex(I3DShModifier**, uint32)

MCCOMErr MCCOMAPI GetModifierByIndex(I3DShModifier modifier, uint32 index)**

Gets the indexth modifier applied to the tree.

Parameters ↗ **I3DShModifier** modifier**
Returns the modifier.

↗ **uint32 index**
Index of the modifier (zero based).

I3DShTreeElement::SetModifiers(const TMCCountedPtrArray<I3DShModifier>*, ECloneMode)

MCCOMErr MCCOMAPI SetModifiers(const TMCCountedPtrArray<I3DShModifier>* modifiers, ECloneMode cloneMode)

Sets the whole list of modifiers applied to the tree.

Parameters ↗ **const TMCCountedPtrArray<I3DShModifier>* modifiers**
The new modifier list.

↗ **ECloneMode cloneMode**
Specifies whether to clone with or without animation information.

I3DShTreeElement::LinksChanged()

MCCOMErr MCCOMAPI LinksChanged()

Call **LinksChanged()** to tell the Shell that a link of the tree element has changed.

I3DShTreeElement::BeginGetRenderables(const TRenderableAndTfmArray*&, boolean)

MCCOMErr MCCOMAPI BeginGetRenderables(const TRenderableAndTfmArray*& instances, boolean inOnlyShown = false)

Gets the renderables of the tree.

Parameters ↗ **const TRenderableAndTfmArray*& instances**
The list of renderables to fill.

↗ **boolean inOnlyShown**
Set *inOnlyShown* to true to filter out elements that have been hidden by the user.

I3DShTreeElement::EndGetRenderables()

MCCOMErr MCCOMAPI EndGetRenderables()

<zot>Need method description</zot> Call after BeginGetRenderables.

I3DShTreeElement::GetConstraint(I3DShConstraint**)

MCCOMErr MCCOMAPI GetConstraint(I3DShConstraint link)**

Gets the constraint applied to the tree.

Parameters  **I3DShConstraint**** *link*

I3DShTreeElement::SetConstraint(I3DShConstraint*, ECloneMode)

MCCOMErr MCCOMAPI SetConstraint(I3DShConstraint* link, ECloneMode cloneMode)

Sets a constraint on the tree element.

Parameters  **I3DShConstraint*** *link*
The constraint to be applied.

 **ECloneMode** *cloneMode*
Specifies whether to clone with or without animation information.

I3DShTreeElement::SetConstraintByID(int32)

MCCOMErr MCCOMAPI SetConstraintByID(int32 kindID)

Sets a constraint on a tree element by specifying the constraint's ID.

Parameters  **int32** *kindID*
The constraint's ID

I3DShTreeElement::GetDataComponentsCount()

uint32 MCCOMAPI GetDataComponentsCount()

Gets the number of the data components attached to the tree element.

I3DShTreeElement::GetDataComponentByIndex(I3DShDataComponent**, uint32)

MCCOMErr MCCOMAPI GetDataComponentByIndex(I3DShDataComponent dataComp, uint32 index)**

Gets the indexth data component of the tree.

Parameters  **I3DShDataComponent**** *dataComp*

→ **uint32** *index*
Index of the data component (zero based).

I3DShTreeElement::SetDataComponentByID(I3DShDataComponent*, ECloneMode)

MCCOMErr MCCOMAPI SetDataComponentByID(I3DShDataComponent* dataComp, ECloneMode cloneMode)

Adds a data component to the tree. If a data component with the same ID is found in the data components list, it is replaced by the new one.

Parameters → **I3DShDataComponent*** *dataComp*
The new data component to add to the tree.

→ **ECloneMode** *cloneMode*
Specifies whether to clone with or without animation information.

I3DShTreeElement::GetDataComponentByID(I3DShDataComponent**, IDType)

MCCOMErr MCCOMAPI GetDataComponentByID(I3DShDataComponent dataComp, IDType kindID)**

Gets the data component which class signature is equal to kindID.

Parameters → **I3DShDataComponent**** *dataComp*
Returns the data component.

→ **IDType** *kindID*
The class signature of the data component to return.

I3DShTreeElement::InsertDataComponent(I3DShDataComponent*)

MCCOMErr MCCOMAPI InsertDataComponent(I3DShDataComponent* dataComp)

Inserts a data component into the tree's data components list.

Parameters → **I3DShDataComponent*** *dataComp*
The data component to be inserted.

I3DShTreeElement::InsertDataComponentByID(int32, int32)

MCCOMErr MCCOMAPI InsertDataComponentByID(int32 kindID, int32 index)

Inserts a data component of the specified ID into the tree's data components list at the given index.

Parameters → **int32** *kindID*
The ID of the data component to be inserted.

→ **int32** *index*
Where to insert the data component.

I3DShTreeElement::RemoveDataComponent(uint32)

MCCOMErr MCCOMAPI RemoveDataComponent(uint32 index)

Removes the data component at the given index from the tree's data components list.

Parameters  **uint32 index**
The index of the data component to remove.

I3DShTreeElement::SetData(TDataKey, IMCUnknown*)

void MCCOMAPI SetData(TDataKey dataKey, IMCUnknown* data)

<zot>Need method description</zot> comment in file says: // DataByKey

Parameters  **TDataKey dataKey**
<zot>Need Description</zot>

 **IMCUnknown* data**
<zot>Need Description</zot>

I3DShTreeElement::GetData(TDataKey, IMCUnknown**)

void MCCOMAPI GetData(TDataKey dataKey, IMCUnknown data)**

<zot>Need method description</zot> comment in file says: // DataByKey

Parameters  **TDataKey dataKey**
<zot>Need Description</zot>

 **IMCUnknown** data**
<zot>Need Description</zot>

I3DShTreeElement::ReleaseData(TDataKey)

void MCCOMAPI ReleaseData(TDataKey dataKey)

<zot>Need method description</zot> comment in file says: // DataByKey

Parameters  **TDataKey dataKey**
<zot>Need Description</zot>

I3DShTreeElement::IsOpen()

boolean MCCOMAPI IsOpen()

Tells if the tree (group) is opened or closed in the hierarchy.

I3DShTreeElement::GetEditor()

int32 MCCOMAPI GetEditor()

Return the ID of the Editor (i.e. the module) capable of editing this tree in the Modeling room. Returns zero when no editor is available

I3DShTreeElement::GetAnimationMethod(I3DShAnimationMethod**)

MCCOMErr MCCOMAPI GetAnimationMethod(I3DShAnimationMethod animationMethod)**

Gets the animation method of the tree.

Parameters  **I3DShAnimationMethod**** *animationMethod*
Returns the animation method of the tree.

I3DShTreeElement::SetAnimationMethod(I3DShAnimationMethod*, ECloneMode, real)

void MCCOMAPI SetAnimationMethod(I3DShAnimationMethod* animationMethod, ECloneMode cloneMode, real conversionQuality=0.0f)

<zot>Need method description</zot> Sets the animation method.

Parameters  **I3DShAnimationMethod*** *animationMethod*
<zot>Need Description</zot>

 **ECloneMode** *cloneMode*
Specifies whether to clone with or without animation information.

 **real** *conversionQuality*
By default, set to 0.0f.

I3DShTreeElement::GetLocalTreeTransform(TTreeTransform&)

void MCCOMAPI GetLocalTreeTransform(TTreeTransform& mat)

Gets the local transformation of the tree.

Parameters  **TTreeTransform&** *mat*
Returns the local transformation of the tree.

I3DShTreeElement::SetLocalTreeTransform(const TTreeTransform&)

void MCCOMAPI SetLocalTreeTransform(const TTreeTransform& mat)

Sets the local transformation of the tree.

Parameters  **TTreeTransform&** *mat*
The new local transformation of the tree.

I3DShTreeElement::GetGlobalTreeTransform(TTreeTransform&)

void MCCOMAPI GetGlobalTreeTransform(TTreeTransform& mat)

Gets the global transformation of the tree.

Parameters ↗ **TTreeTransform& mat**
Returns the global transformation of the tree.

I3DShTreeElement::SetGlobalTreeTransform(const TTreeTransform&, int16, boolean)

void MCCOMAPI SetGlobalTreeTransform(const TTreeTransform& mat, int16 mode = kXTreeBehaviorDefault, boolean withXYZScaling = true)
Sets the global transformation of the tree.

Parameters ↗ **TTreeTransform& mat**
The new global transformation of the tree.

↗ **int16 mode**
<zot>Need Description</zot>mode is set to kXTreeBehaviorDefault.

↗ **boolean withXYZScaling**
Default to true, unused.

I3DShTreeElement::GetGlobalTreeTransform3D(TTreeTransform3D&)

void MCCOMAPI GetGlobalTransform3D(TTransform3D& mat)
Gets the global transformation of the tree.

Parameters ↗ **TTreeTransform3D& mat**
Returns the global transformation of the tree.

I3DShTreeElement::SetGlobalTransform3D(const TTransform3D&, int16)

void MCCOMAPI SetGlobalTransform3D(const TTransform3D& mat, int16 mode = kXTreeBehaviorDefault)
Sets the global transformation of the tree.

Parameters ↗ **const TTreeTransform& mat**
The new global transformation of the tree.

↗ **int16 mode**
<zot>Need Description</zot>mode is set to kXTreeBehaviorDefault.

I3DShTreeElement::GetFatherGlobalTransform3D(TTransform3D&)

void MCCOMAPI GetFatherGlobalTransform3D(TTransform3D& mat)
<zot>Need method description</zot>.

Parameters ↗ **TTreeTransform3D& mat**
<zot>Need Description</zot>

I3DShTreeElement::CloneSubTree(ECloneMode, I3DShTreeElement**)

MCCOMErr MCCOMAPI CloneSubTree(ECloneMode cloneMode, I3DShTreeElement clone)**

<zot>Need method description</zot>Clones part of a tree element.

Parameters

- ➦ **ECloneMode** *cloneMode*
Specifies whether to clone with or without animation information.
- ➦ **I3DShTreeElement**** *clone*
<zot>Need Description</zot>

I3DShTreeElement::GetUndoInfo(TTreeUndoInfo&)

void MCCOMAPI GetUndoInfo(TTreeUndoInfo& undoInfo)

<zot>Need method description</zot>Gets information to undo changes.

Parameters

- ➦ **TTreeUndoInfo&** *undoInfo*
<zot>Need Description</zot>

I3DShTreeElement::SetFromUndoInfo(const TTreeUndoInfo&, const TTreeUndoInfo*)

void MCCOMAPI SetFromUndoInfo(const TTreeUndoInfo& toInfo, const TTreeUndoInfo* fromInfo)

<zot>Need method description</zot>.

Parameters

- ➦ **TTreeUndoInfo&** *undoInfo*
<zot>Need Description</zot>
- ➦ **const TTreeUndoInfo&** *fromInfo*
<zot>Need Description</zot>

I3DShTreeElement::GetTopIfConstrained(I3DShTreeElement**)

MCCOMErr MCCOMAPI GetTopIfConstrained(I3DShTreeElement treeElement)**

<zot>Need method description</zot>If a constraint is applied, returns the top of the tree element.

Parameters

- ➦ **I3DShTreeElement**** *treeElement*
<zot>Need Description</zot>

I3DShTreeElement::AddWireFrame(I3DExWireFrame*)

MCCOMErr MCCOMAPI AddWireFrame(I3DExWireFrame* wireFrame)

<zot>Need method description</zot>.

Parameters

- ➦ **I3DExWireFrame*** *wireFrame*
<zot>Need Description</zot>

<zot>Need method description</zot>Restores a constraint to a tree element.

Parameters  **I3DShConstraint*** *inLink*
 <zot>Need Description</zot>

I3DShTreeElement::GetExists()

boolean MCCOMAPI GetExists()

<zot>Need method description</zot>Gets the animated visibility/existence parameter. Returns True when the parameter exists.

I3DShTreeElement::SetExists(boolean)

void MCCOMAPI SetExists(boolean inExists)

<zot>Need method description</zot>Returns True if it sets the animated visibility/existence parameter.

Parameters  **boolean** *inExists*
 <zot>Need Description</zot>

I3DShTreeElement::GetIsShown()

boolean MCCOMAPI GetIsShown()

<zot>Need method description</zot>Gets the non-animated visibility parameter in the 3D views.

I3DShTreeElement::SetIsShown(boolean)

void MCCOMAPI SetIsShown(boolean inIsShown)

<zot>Need method description</zot>Sets the non-animated visibility parameter in the 3D views.

Parameters  **boolean** *inIsShown*
 <zot>Need Description</zot>

I3DShTreeElement::SetShadowCasting(boolean)

MCCOMErr MCCOMAPI SetShadowCasting(boolean shadowCasting)

<zot>Need method description</zot>Returns True if shadows are cast.

Parameters  **boolean** *shadowCasting*
 <zot>Need Description</zot>

I3DShTreeElement::SetShadowCatching(boolean)

MCCOMErr MCCOMAPI SetShadowCatching(boolean shadowCatching)

<zot>Need method description</zot>Returns True if shadows are caught.

Parameters  **boolean** *shadowCatching*
 <zot>Need Description</zot>

I3DShTreeElement::GetShadowCasting()

boolean MCCOMAPI GetShadowCasting()

<zot>Need method description</zot> Gets whether shadows are cast.

I3DShTreeElement::GetShadowCatching()

boolean MCCOMAPI GetShadowCatching()

<zot>Need method description</zot> Gets whether shadows are caught.

I3DShTreeElement::GetAnimateTree()

boolean MCCOMAPI GetAnimateTree()

<zot>Need method description</zot>

I3DShTreeElement::SetAnimateTree(boolean)

void MCCOMAPI SetAnimateTree(boolean animateTree)

<zot>Need method description</zot>

Parameters

 **boolean** *animateTree*

<zot>Need Description</zot>

I3DShTreeElement::ClearInstanceCache()

void MCCOMAPI ClearInstanceCache()

Clears the renderable cache that is stored at the Tree Element level.

9.19 I3DShTweener Interface

Family ID: 'twee'

Interface ID: IID_I3DShTweener

Interface file: I3DShTweener.h

Description

The **I3DShTweener** interface is the wrapper for an I3DExTweener. Tweeners are 3D Components that describe the interpolation between key frames.

Method Index

"I3DShTweener::Tween(I3DShKeyFrame*, int32, I3DShTweenerChainLink*)"
 "I3DShTweener::Edit(I3DShTweener**, I3DShTweenerChainLink*)"
 "I3DShTweener::SetLink(I3DShTweenerChainLink*)"
 "I3DShTweener::GetLinkNoAddRef()"

I3DShTweener::Tween(I3DShKeyFrame*, int32, I3DShTweenerChainLink*)

MCCOMErr MCCOMAPI Tween(I3DShKeyFrame* res, int32 time, I3DShTweenerChainLink* alink)

Tween() fills a key frame with interpolated data given the time and the chain link to use. The given chain link contains the needed time line information, the previous key even, the tweener, as well as the next key event (contained in the chain link).

Parameter

-  **I3DShKeyFrame* res**
The specified key frame that will be filled with interpolated data.
-  **int32 time**
The time line information.
-  **I3DShTweenerChainLink* alink**
The given chain link.

I3DShTweener::Edit(I3DShTweener**, I3DShTweenerChainLink*)

void MCCOMAPI Edit(I3DShTweener tweener, I3DShTweenerChainLink* chainLink)**

Edit() is not needs description. Check with Greg

Parameter

-  **I3DShTweener** tweener**
not needs definition.
-  **I3DShTweenerChainLink* alink**
not needs definition.

I3DShTweener::SetLink(I3DShTweenerChainLink*)

voidMCCOMAPISetLink(I3DShTweenerChainLink* inLink)

SetLink() is used to access a chain link with a tweener.

Parameter  **I3DShTweenerChainLink*** *inlink*
zot needs definition.

I3DShTweener::GetLinkNoAddRef()

I3DShTweenerChainLink* MCCOMAPIGetLinkNoAddRef()

GetLinkNoAddRef() is used to access a chain link with a tweener. Use this function with caution because **GetLink** can return a NULL pointer if the tweener is not the one on the time line.

9.20 I3DShTweenChainLink Interface

Interface ID: IID_I3DShTweenChainLink

Interface file: I3DShTween.h

Description

The **I3DShTweenChainLink** interface is the key data structure for animation. Tweeners are 3D Components that describe the interpolation between key frames. Each track in the time line is made of a two-way linked list of **I3DShTweenChainLink**. **I3DShTweenChainLink** corresponds to a key event in the time line. It also contains with the tweener immediately following, and a link to the next **I3DShTweenChainLink**.

Method Index

```

“I3DShTweenChainLink::Clone(I3DShTweenChainLink**, I3DShParamTimeLine* )”
“I3DShTweenChainLink::CopyData(I3DShTweenChainLink*)”
“I3DShTweenChainLink::GetTweenNoAddRef(boolean, boolean)”
“I3DShTweenChainLink::GetTween(I3DShTween**, boolean, boolean)”
“I3DShTweenChainLink::SetTween(I3DShTween*)”
“I3DShTweenChainLink::GetKeyFrameNoAddRef()”
“I3DShTweenChainLink::GetKeyFrame(I3DShKeyFrame**)”
“I3DShTweenChainLink::SetKeyFrame(I3DShKeyFrame**)”
“I3DShTweenChainLink::GetTime()”
“I3DShTweenChainLink::SetTime(MicroTick)”
“I3DShTweenChainLink::GetNextNoAddRef()”
“I3DShTweenChainLink::GetNext(I3DShTweenChainLink**)”
“I3DShTweenChainLink::SetNext(I3DShTweenChainLink*)”
“I3DShTweenChainLink::GetPrevNoAddRef()”
“I3DShTweenChainLink::GetPrev(I3DShTweenChainLink**)”
“I3DShTweenChainLink::SetPrev(I3DShTweenChainLink*)”
“I3DShTweenChainLink::GetFatherNoAddRef()”
“I3DShTweenChainLink::RemoveFromChain()”
“I3DShTweenChainLink::InsertAfter(I3DShTweenChainLink*)”
“I3DShTweenChainLink::OffsetLink(MicroTick)”
“I3DShTweenChainLink::GetTimeInfo(TTimeInfo&)”
“I3DShTweenChainLink::IsTweenSelected()”
“I3DShTweenChainLink::IsKeyFrameSelected()”
“I3DShTweenChainLink::SelectKeyFrame(boolean)”
“I3DShTweenChainLink::SelectTween(boolean)”

```

I3DShTweenChainLink::Clone(I3DShTweenChainLink**, I3DShParamTimeLine*)

MCCOMErr MCCOMAPI Clone(I3DShTweenChainLink outLink, I3DShParamTimeLine*inFatherTimeLine)**

Call **Clone()** to get a clone of a tweener chain link.

Parameter  **I3DShTweenChainLink** outLink**
 Contains a brand new clone of *inFatherTimeLine*.

✚ **I3DShParamTimeLine*** *inFatherTimeLine*
Contains the object to clone.

I3DShTweenerChainLink::CopyData(I3DShTweenerChainLink*)

MCCOMErrMCCOMAPI CopyData(I3DShTweenerChainLink* inDest)

CopyData() copies a tweener chain link. Overwrites the data in the already existing *inDest*.

Parameter ✚ **I3DShTweenerChainLink*** *inDest*
A pointer to a tweener chain link.

I3DShTweenerChainLink::GetTweenerNoAddRef(boolean, boolean)

I3DShTweener*MCCOMAPI GetTweenerNoAddRef(boolean inDefaultIfNone = false, boolean inCreateIfNone = false)

GetTweenerNoAddRef() is used to access a tweener. or to create one if none exists.

Use this function to avoid doing an unnecessary AddRef. Refer to Creating COM objects in the Overview to learn more.

Parameter ✚ **boolean*** *inDefaultNone*
inDefaultNone is set to false. If *inDefaultNone* is set to true, it will create a default tweener if none already exists.

✚ **boolean*** *inCreateIfNone*
inCreateIfNone is set to false. <zot> needs more description.

I3DShTweenerChainLink::GetTweener(I3DShTweener**, boolean, boolean)

MCCOMErrMCCOMAPI GetTweener(I3DShTweener outTweener, boolean inDefaultIfNone = false, boolean inCreateIfNone = false)**

GetTweener() is used to access a tweener, or to create one if none exists.

Parameter ✚ **I3DShTweener**** *outTweener*
outTweener is a pointer to the output tweener.

✚ **boolean*** *inDefaultNone*
inDefaultNone is set to false. If *inDefaultNone* is set to true, it will create a default tweener if none already exists and store it in the I3DShTweenerChainLink.

✚ **boolean*** *inCreateIfNone*
inCreateIfNone is set to false. <zot> needs more description.

I3DShTweenerChainLink::SetTweener(I3DShTweener*)

MCCOMErrMCCOMAPI SetTweener(I3DShTweener* inTweener)

SetTween() is used to store a tweener in the I3DShTweenChainLink.

Parameter  **I3DShTween*** *inTween*
inTween is a pointer to the tweener to be stored.

I3DShTweenChainLink::GetKeyFrameNoAddRef()

I3DShKeyFrame* **MCCOMAPI GetKeyFrameNoAddRef()**

GetTweenNoAddRef() is used to access key frame data.

Use this function to avoid doing an unnecessary AddRef.

I3DShTweenChainLink::GetKeyFrame(I3DShKeyFrame**)

MCCOMErr MCCOMAPI GetKeyFrame(I3DShKeyFrame outKeyFrame)**

GetKeyFrame() is used to access key frame data.

Parameter  **I3DShKeyFrame**** *outKeyFrame*
outKeyFrame is <zot> needs more description.

I3DShTweenChainLink::SetKeyFrame(I3DShKeyFrame**)

MCCOMErrMCCOMAPI SetKeyFrame(I3DShKeyFrame* inKeyFrame)

SetKeyFrame() is used to access key frame data. <zot> needs more description.

Parameter  **I3DShKeyFrame**** *inKeyFrame*
inKeyFrame is <zot> needs more description.

I3DShTweenChainLink::GetTime()

MicroTickMCCOMAPI GetTime()

GetTime() is used to access key event time.

I3DShTweenChainLink::SetTime(MicroTick)

MCCOMErrMCCOMAPI SetTime(MicroTick inTime)

SetTime() is used to access key event time.

Parameter  **MicroTick** *inTime*
inTime is <zot> needs more description.

I3DShTweenChainLink::GetNextNoAddRef()

I3DShTweenChainLink* **MCCOMAPI GetNextNoAddRef()**

GetNextNoAddRef() is used to navigate the chain links. A track of time line is composed of a 2-way linked list of I3DShTweenerChainLink.

Use this function to avoid doing an unnecessary AddRef.

I3DShTweenerChainLink::GetNext(I3DShTweenerChainLink**)

MCCOMErrMCCOMAPI GetNext(I3DShTweenerChainLink outNext)**

GetNext() is used to navigate the chain links. A track of time line is composed of a 2-way linked list of I3DShTweenerChainLink.

Parameter  **I3DShTweenerChainLink** outNext**
outNext is <zot> needs more description.

I3DShTweenerChainLink::SetNext(I3DShTweenerChainLink*)

MCCOMErr MCCOMAPI SetNext(I3DShTweenerChainLink* inNext)

GetNext() is used to navigate the chain links. A track of time line is composed of a 2-way linked list of I3DShTweenerChainLink.

Parameter  **I3DShTweenerChainLink* inNext**
inNext is <zot> needs more description.

I3DShTweenerChainLink::GetPrevNoAddRef()

I3DShTweenerChainLink*MCCOMAPI GetPrevNoAddRef()

GetPrevNoAddRef() is used to navigate the chain links. A track of time line is composed of a 2-way linked list of I3DShTweenerChainLink.

Use this function to avoid doing an unnecessary AddRef.

I3DShTweenerChainLink::GetPrev(I3DShTweenerChainLink**)

MCCOMErrMCCOMAPI GetPrev(I3DShTweenerChainLink outPrev)**

GetPrev() is used to navigate the chain links. A track of time line is composed of a 2-way linked list of I3DShTweenerChainLink.

Parameter  **I3DShTweenerChainLink** outPrev**
outPrev is <zot> needs more description.

I3DShTweenerChainLink::SetPrev(I3DShTweenerChainLink*)

MCCOMErr MCCOMAPI SetPrev(I3DShTweenerChainLink* inPrev)

GetPrev() is used to navigate the chain links. A track of time line is composed of a 2-way linked list of I3DShTweenerChainLink.

Parameter  **I3DShTweenerChainLink* inPrev**
inPrev is <zot> needs more description.

I3DShTweenerChainLink::GetFatherNoAddRef()

I3DShParamTimeLine*MCCOMAPI GetFatherNoAddRef()

GetFatherNoAddRef() is used to retrieve the time line track to whom the chain link belongs.

Use this function to avoid doing an unnecessary AddRef.

I3DShTweenerChainLink::RemoveFromChain()

MCCOMErrMCCOMAPI RemoveFromChain()

RemoveFromChain() is used to remove a link from the chain. It also links the previous and next links together.

I3DShTweenerChainLink::InsertAfter(I3DShTweenerChainLink*)

MCCOMErrMCCOMAPI InsertAfter(I3DShTweenerChainLink* inAfter)

InsertAfter() is used to insert a link after a specific link (inAfter).

Parameter  **I3DShTweenerChainLink* inAfter**
inAfter is a pointer to the specific link.

I3DShTweenerChainLink::OffsetLink(MicroTick)

MCCOMErrMCCOMAPI OffsetLink(MicroTick inDeltaTime)

OffsetLink() is used to increment/decrement the time of a chain link.

Parameter  **MicroTick inDeltaTime**
inDeltaTime is a pointer to delta of time for the offset.

I3DShTweenerChainLink::GetTimeInfo(TTimeInfo&)

MCCOMErrMCCOMAPI GetTimeInfo(TTimeInfo& outTimeInfo)

GetTimeInfo() is used to differentiate key frames that completely overlap and have the same time information.

Parameter  **TTimeInfo& outTimeInfo**
outTimeInfo is a pointer to the time and index of a key event.

I3DShTweenerChainLink::IsTweenerSelected()

boolean MCCOMAPI IsTweenerSelected()

IsTweenerSelected() returns True if the tweener is selected.

I3DShTweenerChainLink::IsKeyFrameSelected()

boolean MCCOMAPI IsKeyFrameSelected()

IsKeyFrameSelected() returns True if the key frame is selected.

I3DShTweenerChainLink::SelectKeyFrame(boolean)

MCCOMErr MCCOMAPI SelectKeyFrame(boolean select)

SelectKeyFrame() is used to select a key frame.

Parameter  **boolean** *select*
select <zot> needs more description.

I3DShTweenerChainLink::SelectTweener(boolean)

MCCOMErr MCCOMAPI SelectTweener(boolean select)

SelectTweener() is used to select a tweener.

Parameter  **boolean** *select*
select <zot> needs more description.

9.21 I3DShWireFrame Interface

Interface ID: IID_I3DShWireFrame

Interface file: I3DWireFrame.h

Description

The **I3DShWireFrame** interface is used for accessing wire frames.

Method Index

```

“I3DShWireFrame::AllocateFMesh(int32, int32, int32)”
“I3DShWireFrame::GetFMesh(FacetMesh**)”
“I3DShWireFrame::AllocateWireFrame(int16, int16)”
“I3DShWireFrame::GetWireFrame(WFData&)”
“I3DShWireFrame::Connect(int16, int16)”
“I3DShWireFrame::SetPoint(int16, EWFHandleType, const TVector3&)”
“I3DShWireFrame::Transform(const TTreeProjectionTransform&)”
“I3DShWireFrame::TestHit(TVector2&, real)”
“I3DShWireFrame::AddRenderables(TMCPtrArray<I3DShRenderable>&)”
“I3DShWireFrame::GetColor()”
“I3DShWireFrame::SetColor(const TMCCColorRGB&)”
“I3DShWireFrame::ExcludeFromZBuffer(boolean, boolean)”
“I3DShWireFrame::SetSelectedState(const boolean)”
“I3DShWireFrame::ExcludeFromProjection(boolean)”

```

I3DShWireFrame::AllocateFMesh(int32, int32, int32)

MCCOMErrMCCOMAPI AllocateFMesh(int32 numFacets, int32 numVertices, int32 facetStreamSize)

Call **AllocateFMesh()** to allocate memory for a facet mesh.

- Parameter**
-  **int32 numFacets**
Specifies the number of facets in the mesh.
 -  **int32 numVertices**
Specifies the number of vertices in the mesh.
 -  **int32 facetStreamSize**
Specifies the <Zot>.

I3DShWireFrame::GetFMesh(FacetMesh**)

MCCOMErrMCCOMAPI GetFMesh(FacetMesh aMesh)**

GetFMesh() gets a facet mesh.

- Parameter**
-  **FacetMesh** aMesh**
A pointer to a facet mesh.

I3DShWireFrame::AllocateWireFrame(int16, int16)

MCCOMErrMCCOMAPI AllocateWireFrame(int16 numPoints, int16 numConnections)

AllocateWireFrame() is used to allocate memory for a wire frame

- Parameter**
-  **int16 numPoints**
Specifies the number of points in the wire frame.
 -  **int16 numConnections**
Specifies the number of connections in the wire frame.
-

I3DShWireFrame::GetWireFrame(WFData&)

MCCOMErrMCCOMAPI GetWireFrame(WFData& aWireFrame)

GetWireFrame() is used to access a wire frame

- Parameter**
-  **WFData& aWireFrame**
The data of the specified wire frame.
-

I3DShWireFrame::Connect(int16, int16)

MCCOMErrMCCOMAPI Connect(int16 point, int16 from, int16 to)

Connect() is used to connect two points. <zot needs more>

- Parameter**
-  **int16 from**
The starting point to connect.
 -  **int16 to**
The ending point to connect.
-

I3DShWireFrame::SetPoint(int16, EWFHandleType, const TVector3&)

MCCOMErrMCCOMAPI SetPoint(int16 point, EWFHandleType type, const TVector3& loc)

SetPoint() is used to set a point in the wire frame in a specified location.

- Parameter**
-  **int16 point**
The specified point.
 -  **EWFHandleType type**
The type of point. <zot verify>
 -  **const TVector3& loc**
The location of the point.

I3DShWireFrame::Transform(const TTreeProjectionTransform&)

MCCOMErrMCCOMAPI Transform(const TTreeProjectionTransform& trans)

Transform() is used to <zot do what?>.

Parameter  **const TTreeProjectionTransform& trans**
 trans is <zot> needs more description.

I3DShWireFrame::TestHit(TVector2&, real)

shortMCCOMAPI TestHit(const TVector2& point, real error)

TestHit() is used to <zot> needs more description.

Parameter  **TVector2& point**
 point is <zot> needs more description.

 **real error**
 error is <zot> needs more description.

I3DShWireFrame::AddRenderables(TMCPtrArray<I3DShRenderable>&)

MCCOMErrMCCOMAPI AddRenderables(TMCPtrArray<I3DShRenderable>& renderables)

AddRenderables() is used to <zot needs description>.

Parameter  **TMCPtrArray<I3DShRenderable>& renderables**
 renderables is <zot> needs more description.

I3DShWireFrame::GetColor()

TMCCColorRGBMCCOMAPI GetColor()

GetColor() returns the color of the wire frame.

I3DShWireFrame::SetColor(const TMCCColorRGB&)

MCCOMErrMCCOMAPI SetColor(const TMCCColorRGB& color)

SetColor() is used to set the color of the wire frame.

Parameter  **const TMCCColorRGB& color**
 color is the new color value for the wire frame.

I3DShWireFrame::ExcludeFromZBuffer(boolean, boolean)

MCCOMErrMCCOMAPI ExcludeFromZBuffer(boolean excludeLines, boolean excludePoints)

ExcludeFromZBuffer() is used to exclude a specified wire frame from being rendered by the ZBuffer renderer.

- Parameter**
-  **boolean** *excludeLines*
Specifies whether the lines are rendered.
 -  **boolean** *excludePoints*
Specifies whether the points are rendered.

I3DShWireFrame::SetSelectedState(const boolean)

MCCOMErrMCCOMAPI SetSelectedState(const boolean selected)

SetSelectedState() is used to set whether the wire frame is selected or not.

- Parameter**
-  **const boolean** *selected*
selected is true when the wire frame is selected.

I3DShWireFrame::ExcludeFromProjection(boolean)

MCCOMErrMCCOMAPI ExcludeFromProjection(boolean exclude)

ExcludeFromProjection() is used to exclude a specific wire from displaying on the projection planes of the working box.

- Parameter**
-  **boolean** *exclude*
exclude is true when the wire frame is excluded.

Chapter 10 - DataBase Overview

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

10.1 Scene

A *Scene* is mostly comprised of 2 structures:

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

Objects

Objects contains the geometric information defining its surface regardless of its position. It also contains a reference to its shading information. The Objects used in a Scene are located in the Objects tab of the Browser window in Carrara.

An Object (I3DShObject) can be of two types: a *Primitive* (I3DShPrimitive) or a *Scene* (I3DShScene).

Primitives are the basic objects that are located in the toolbar as icons. There are different classes of primitives. Carrara defines a certain number of default internal primitives that can be used by extensions. Those primitives will always be there. They are the Spline Primitive, the Polygon Array, the Patch Array, the Polygon List (list of triangular facets), the Flat Primitive, the 3D Text, etc. Some other primitives are external primitives (coded in regular extensions) like the Cube, the Sphere, the Isocaedra, etc. Third party developers can define new classes of primitives.

A Scene is also considered an Object, because it can be instanciated in another Scene, like if it was a complex object. This is sometimes called “Scene Instancing”.

Scene Tree and Tree Elements

The scene tree is a hierarchical structure containing *Tree Elements* (I3DShTreeElement).

There are four types of tree elements: Object Instances (I3DShInstance), Light Sources (I3DShLightsource), Cameras (I3DShCamera) and Groups (I3DShGroup). Those elements are detailed below. All these elements have 2 things in common:

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

Object Instances

An object instance is a tree element that has a reference to an Object in the object list. Therefore there can be multiple instances in a scene of a same object. This object can be either a Primitive or another Scene.

Shading can be “overridden” at the instance level, thus surpassing any shading information the original object may have.

Lights

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

Cameras

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

Groups

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

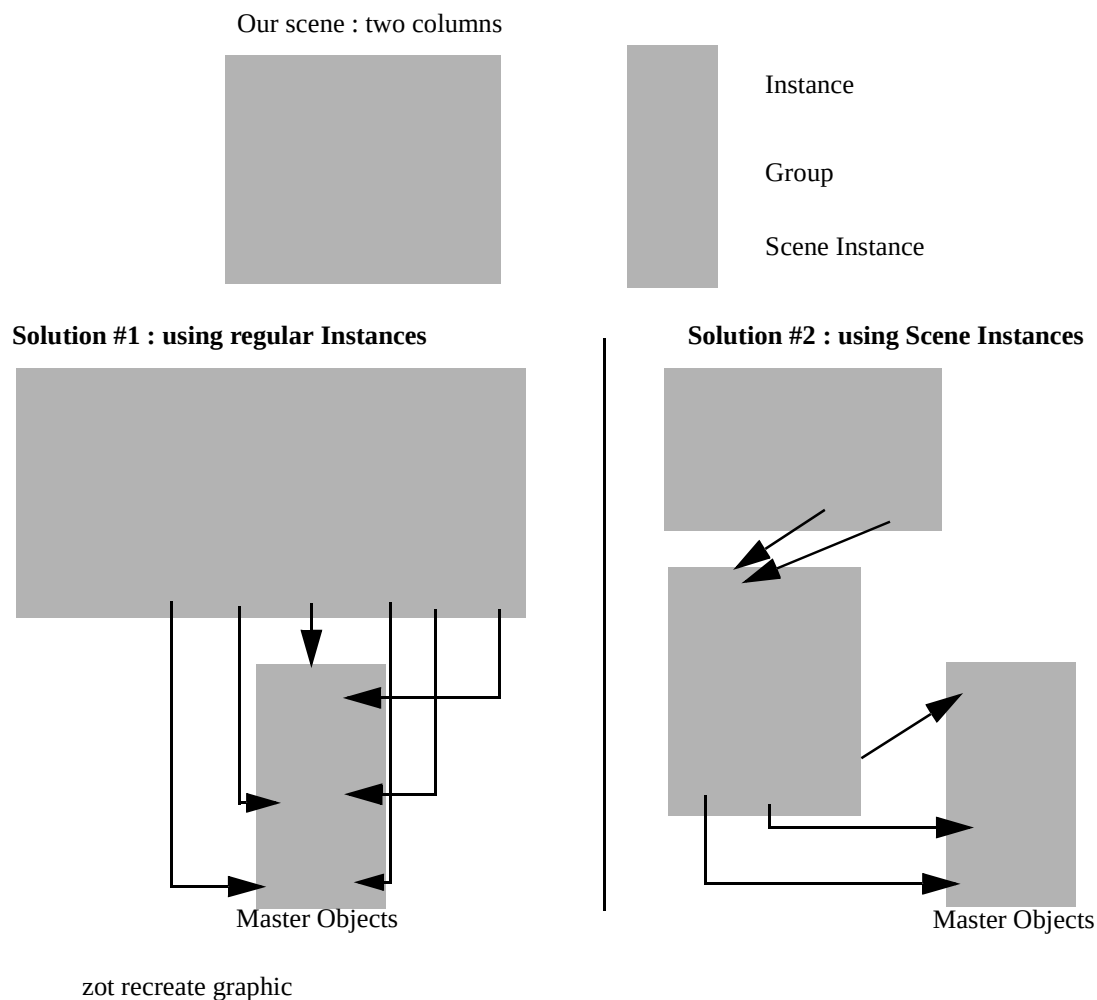
Scene Instance vs regular Instances

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

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

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

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



10.2 Coordinate Systems

Global Coordinate System

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

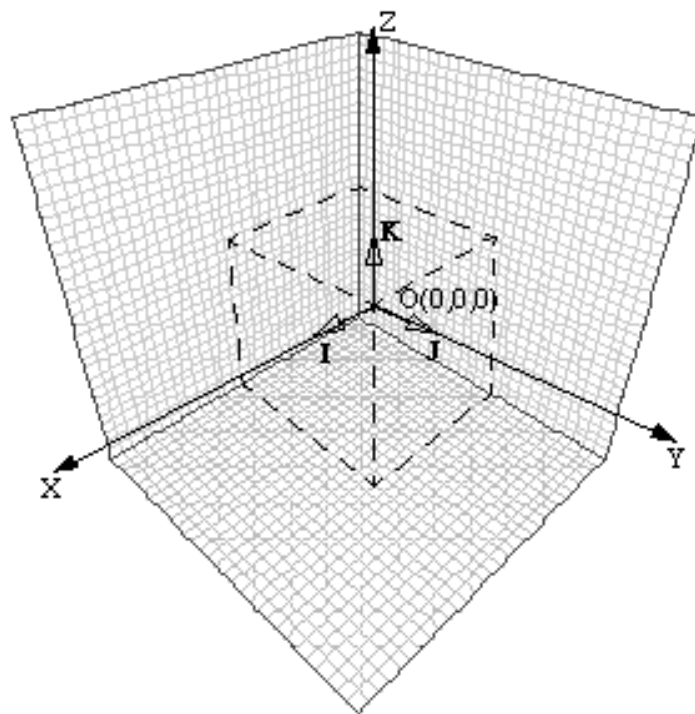


Figure 1. The Global Coordinate System

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

Working Box Coordinate System

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

Local Coordinate System (or Object Coordinate System)

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

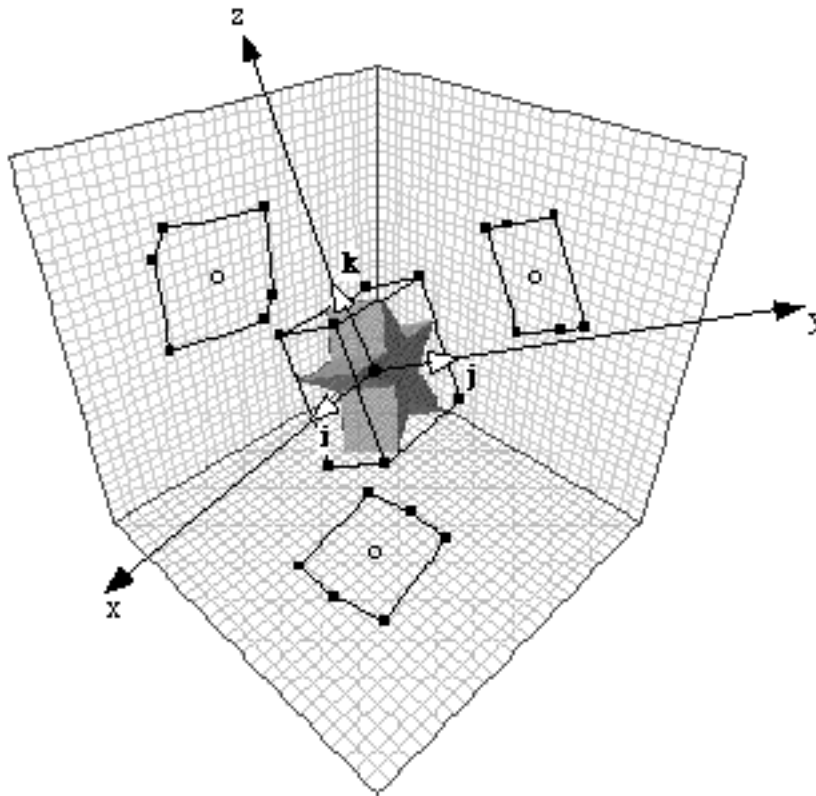


Figure 2. The Local Coordinate System of an object in a scene
The **i**, **j** and **k** vectors are the unit vectors of the x, y and z axis.

Screen Coordinate System

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

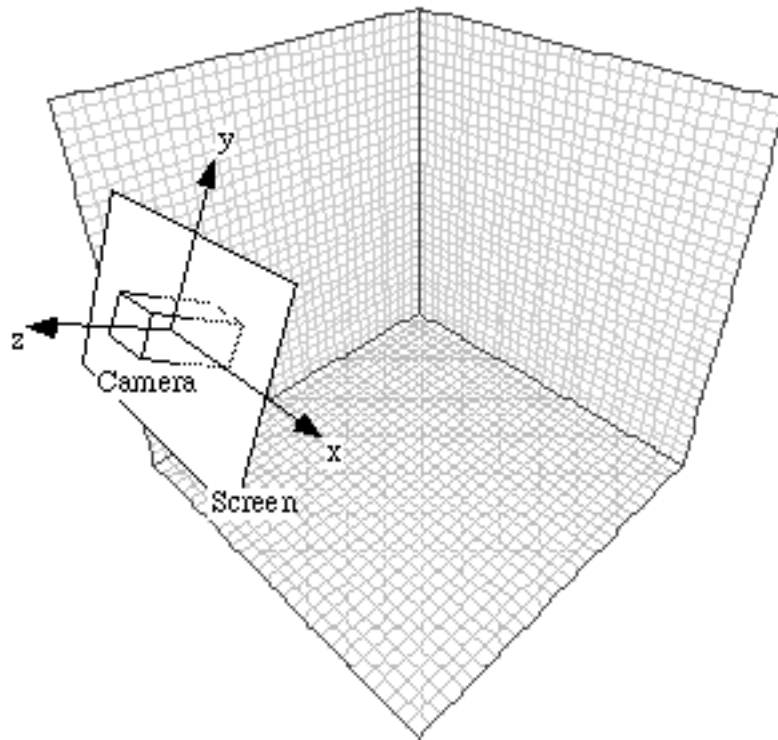


Figure 3. The Screen Coordinate System

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

The Screen Coordinate System versus the Camera's Coordinate System

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

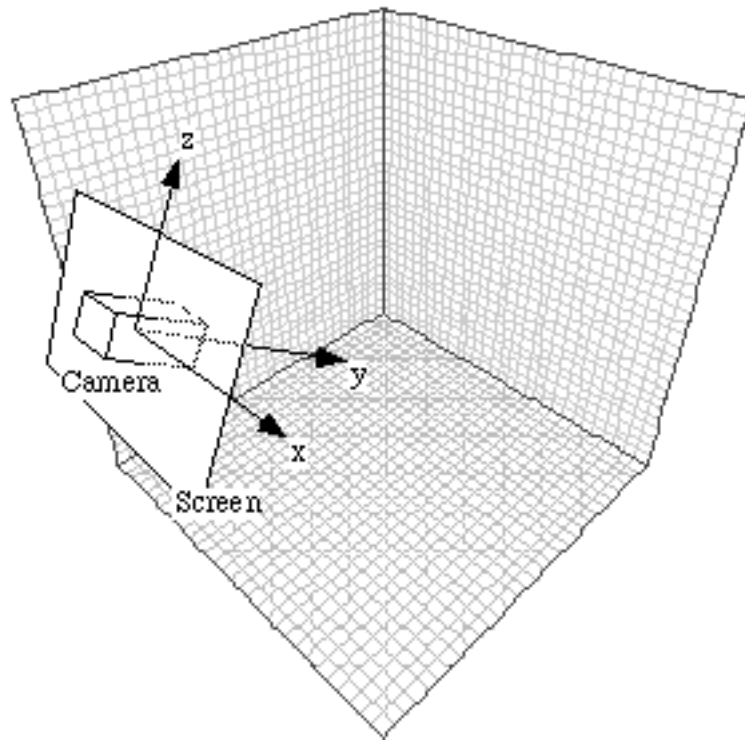


Figure 4. The camera's Local Coordinate System

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

The screen pixels space

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

1 3D unit = 288 points

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

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

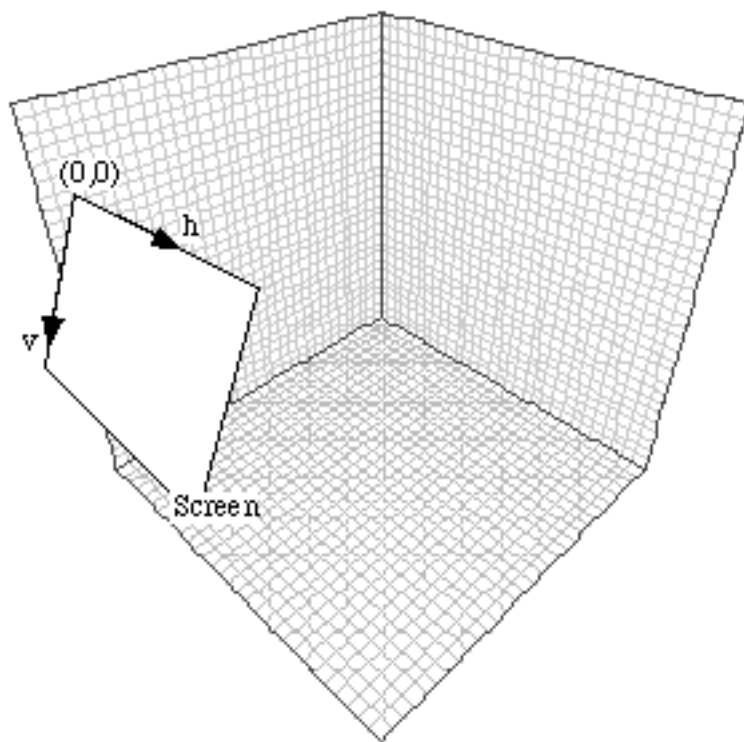


Figure 5. The Screen Pixels Space

10.3 Geometry

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

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

```
typedef float NUM3D;
```

Units System

The units system used in Carrara is defined as follows:

1 3D unit = 4 inches

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

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

Tree Elements Transformation

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

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

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

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

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

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

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

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

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

With :

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

More on the rotation matrix:

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

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

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

Geometry basics

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

Surface Point

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

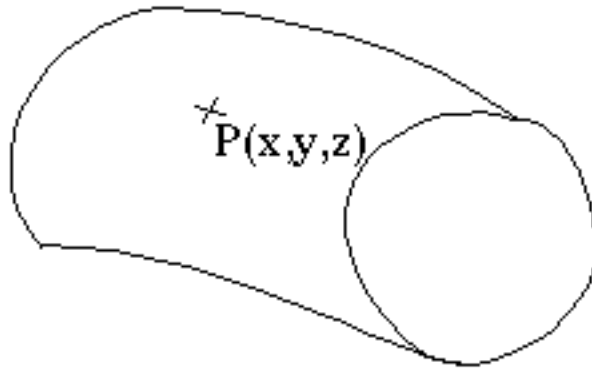


Figure 6. A point on a surface

Surface Normal

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

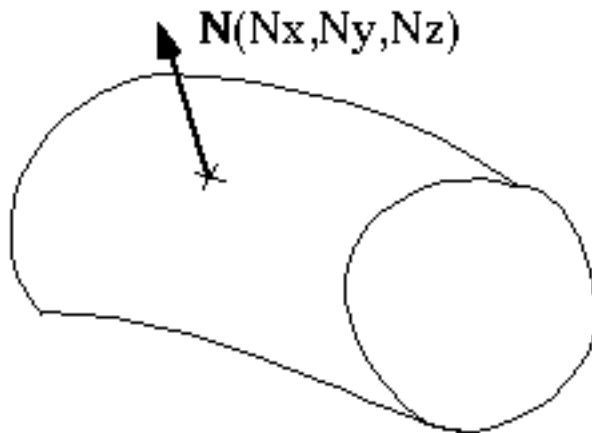


Figure 7. A Normal to the surface

Note that :

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

UV Space (Texture Coordinate System)

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

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

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

Parametric mapping - a better solution

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

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

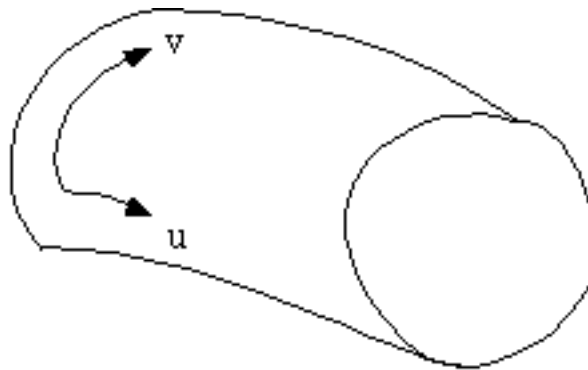


Figure 8. A typical u,v space on the side surface of an object

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

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

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

Shading

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

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

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

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

Facets

There are two kinds of low level geometric data used for rendering, exporting, etc.: **facets** and **bicubic patches**. <zot> this section needs to be rewritten; there are only patches now.

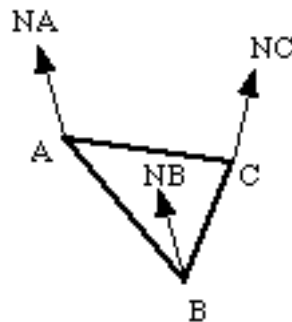


Figure 9. A 3D facet

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

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

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

Interpolating in a facet

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

from:

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

you can interpolate:

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

Don't forget to renormalize the normal:

$$N_{\text{nit}} = \frac{N}{\sqrt{N^2}}$$

Likewise:

$$u = \alpha uA + \beta uB + \gamma uC$$

$$v = \alpha vA + \beta vB + \gamma vC$$

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

Bicubic Patches

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

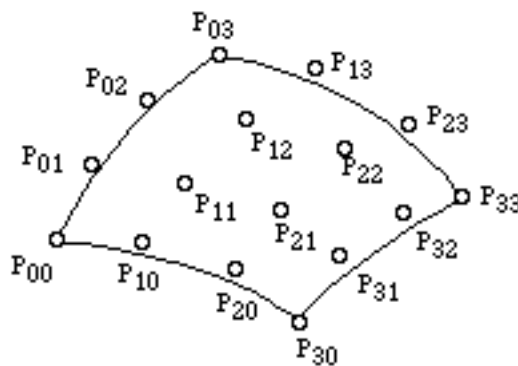


Figure 10. A bicubic patch

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

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

```
short    fReserved;           // Reserved - 0
} PATCH3D;
```

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

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

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

$$0.0 \leq t_u \leq 1.0$$

$$0.0 \leq t_v \leq 1.0$$

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

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

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

Patch normals

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

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

Patch u,v Space

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

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

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

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

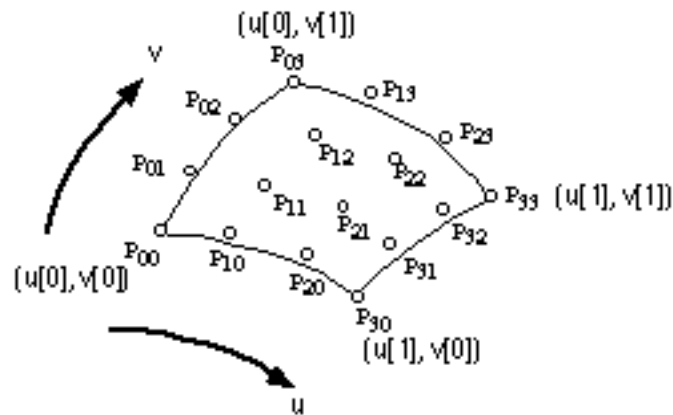


Figure 11. A patch (u,v) Space

To learn more about Bézier bicubic Patches

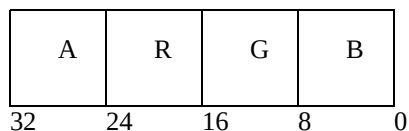
Read the excellent book « Introduction to Computer Graphics » by Folley - Van Dame - , published by Addison-Weisley.

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

10.4 Color

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

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



Color in long (32-bits) format.

Chapter 11 - Index

A

AboutToCloseMainWindow() 115
Activate() 113, 141
AddPatch 209
AddRenderables 269
AddWireFrame (I3DExWireFrame*) 255
AllocateFMesh 267
AllocateWireFrame 268
Apply(I3DShTreeElement) 107
AskForParams 162

B

BeginGetRenderables (const TRenderableAndTfmArray*&, boolean) 249
BeginProgress 168
BeginRender 226
bicubic patch 283
BlitUIImage 173
BOX3D 48
BuildAlertMessage 172
BuildFacetMesh 211
BuildMenuBar() 113

C

Camera 272
camera 272
Camera Coordinate System 276
CanUndo 154
CheckForUserCancel 177
CheckKeyDown 178
Clone 203, 235, 261
Clone()
 I3DExtension 15
CoCreateInstance 159
COLOR3D 54
Connect 268
Coordinate System
 Camera Coordinate System 276
 Global Coordinate System 273
 Local Coordinate System 274
 Object Coordinate System 274

- Working Box Coordinate System 274
- Coordinate Systems
 - Screen Coordinate System 275
- coordinate systems 273
 - global coordinate system 273
 - local (or object) coordinate system 274
 - screen pixels space 277
 - UV space 281
 - working box coordinate system 274
- CopyData (I3DShTweenerChainLink*) 262
- CountPatches 210
- CreateCompatiblePixelBucket 174
- CreateComponent 160
- CreateDefaultComponent 160
- CreateIUnknown 160
- CreateRay()
 - I3DExCamera 82

D

- DataToWireFrame 140
- Deactivate() 113, 141
- DeformBBox() 108
- DeformFacetMesh() 108
- DeformPoint() 108
- Dehydrate() 113
 - I3DExFinalRenderer 94
- Destroy() 112
- DestroyCompatiblePixelBucket 175
- DirectionFilter()
 - I3DExAtmosphericShader 75
- DirectionFilter(const TVector3 ,const TVector3&, TMCColorRGB&) 196
- Do 153
- DoAlert 171
- DoExport 38
- DoImport 41
- DoShade()
 - I3DExShader 133, 236
- Drag and drop 29, 30
- DrawRect 225
- DrawRect()
 - I3DExFinalRenderer 96

E

- EndGetRenderables() 250
- EndProgress 168
- ErrorToString 172
- ExcludeFromProjection 270

ExcludeFromZBuffer 270
ExtensionDataChanged 34
ExtensionDataChanged() 17

F

facet 282
FACET3D 63
Filter() 118
FinishDraw 225
FinishDraw()
 I3DExFinalRenderer 96

G

geometric data type 278
geometry 278
geometry basics 280
Gestalt 170
GetAnimationMethod (I3DShAnimationMethod**) 253
GetAPINumber 170
GetAsciiTranslationTable 175
GetBackdropColor()
 I3DExBackdrop 79
GetBackdropColor(const TVector2&,boolean&,const TBox2D&, TMCColorRGB&) 196
GetBackgroundColor() 77
GetBBox()
 I3DExGeometricPrimitive 121
GetBoundingBox 203
GetBufferNeeds() 117, 223
GetBump() 237
GetClasSignature 157, 163
GetClassName 157, 163
GetColor 269
GetColor()
 I3DExAmbientLight 73
 I3DExShader 133, 237
GetComponentClassInfoArrayByFamilyID 161
GetComponentCount 43
GetComponentData 43
GetConstraint (I3DShConstraint**) 250
GetCreator 177
GetCurrentModifiers 177
GetData (TDataKey, IMCUnknown*) 252
GetDataComponentByID (I3DShDataComponent*, IDType) 251
GetDataComponentByIndex(I3DShDataComponent**, uint32) 250
GetDataComponentsCount () 250
GetEditor 203
GetEditor () 252

GetEnvironmentColor(const TVector3&, boolean&, TMCColorRGB&) 197
 GetExtensionDataBuffer 34
 GetExtensionDataBuffer() 16
 GetExtensionDataMap() 16
 GetFacetBoundingBox 203
 GetFamilySignature 157, 163
 GetFatherGlobalTransform3D(TTransform3D&) 254
 GetFatherNoAddRef() 265
 GetFMesh 267
 GetFontList 177
 GetFreedomRange() 88, 246
 GetFullName(TMCString&) 248
 GetGlobalTreeTransform3D (TTreeTransform3D&) 254
 GetIndPatches 210
 GetInstanceArray 202
 GetInstancesCount 202
 GetKeyFrame (I3DShKeyFrame**) 263
 GetKeyFrameNoAddRef () 263
 GetLastOpenDocPath 175
 GetLastSaveDocPath 175
 GetMaxTimeWithoutComputation (I3DShTreeElement*) 107
 GetModifierByIndex (I3DShModifier**, uint32) 249
 GetMotionLimit() 87
 GetName 154, 201
 GetNbrFreedom() 87, 246
 GetNeededLOD() 109
 GetNext (I3DShTweenChainLink**) 264
 GetNextNoAddRef() 263
 GetObjectID 204
 GetOpenDocumentByIndex 175
 GetOpenDocumentsCount 175
 GetOriginalParametricTextureMap() 238
 GetParameter 33, 164
 GetParameterCount 164
 GetParameterInfo 164
 GetParameterType 164
 GetPartToRedraw 154
 GetPreferredOutput() 132, 238
 GetPreferredRoom(IDType&) 114
 GetPreferredTiling() 97
 GetPrev (I3DShTweenChainLink**) 264
 GetPreviewSize 38
 GetPrevNoAddRef() 264
 GetPrimitiveID()
 I3DExCamera 83
 GetRayHitDetails() 144
 GetRenderingTime()
 I3DExFinalRenderer 94

GetRenderStatistics()
 I3DExFinalRenderer 95
GetResID 34
GetResID()
 I3DExDataExchanger 17
GetResource 172
GetScene 202
GetScene(I3DShScene**) 181, 183, 185
GetShadingFlags()
 I3DExShader 132
GetTextMetrics 177
GetTime() 263
GetTimeInfo(TTimeInfo&) 265
GetTopIfConstrained (I3DShTreeElement**) 255
GetTweenener (I3DShTweenener**, boolean, boolean) 262
GetTweenenerNoAddRef (boolean, boolean) 262
GetUIImage 174
GetUIImageBucket 174
GetUIImageInfo 173
GetUVSpace 202
GetUVSpace()
 I3DExGeometricPrimitive 124
GetUVSpaceCount 202, 211, 222
GetUVSpaceCount()
 I3DExGeometricPrimitive 122, 123, 124
GetValue() 133, 236
GetVector() 134, 237
GetVolumeAttenuation() 150
GetVolumeDensity() 149
GetWireFrame 268
GetWireFrameBBox 140
Global Coordinate System 273
global coordinate system 273
Group 272
group 272

H

HandleEvent 34
HandleEvent() 17
HandlesTool 141
HasAtmosphere() 197
HasBackdrop() 197
HasBackground() 197
HitInfo 68
Hydrate() 112

I

- I3DExAmbientLight interface 73
- I3DExAtmosphericShader interface 75
- I3DExBackdrop interface 79
- I3DExBackground interface 77
- I3DExCamera interface 81
- I3DExDataExchange interface 16
- I3DExExportFilter interface 37
- I3DExFinalRenderer 97
- I3DExFinalRenderer interface 93
- I3DExGeometricPrimitive interface 121
- I3DExImportFilter interface 41
- I3DExLightsource
 - ForEachShadowBuffer(ForEachShadowBufferCallback, void*) 103
 - GetColor(TVector3&, TVector3&, real, TMCColorRGB&, real&) 102
 - GetDirection(TVector3&, TVector3&, real&) 101
 - GetLightInfo(boolean&, boolean&, real&, TTransform3D&) 103
 - GetLightParameter(int32, void*) 103
 - GetNewTrace(int16*) 104
 - GetStandardLight(TStandardLight&) 104
 - GetUVfromDirection(TVector2&, const TVector3&) 103
 - IsVisibleInPerspective() 102
 - SetTransform(TTransform3D*) 101
- I3DExLightsource interface 101, 105
- I3DExLightsourceGel
 - GetGelValues(constTVector2&, TMCColorRGB&) 105
- I3DExModifier
 - Apply(I3DShTreeElement) 107, 112
 - GetMaxTimeWithoutComputation (I3DShTreeElement*) 107
- I3DExModifier interface 107
- I3DExModule
 - AboutToCloseMainWindow() 115
 - Activate() 113
 - BuildMenuBar() 113
 - Deactivate() 113
 - Dehydrate() 113
 - Destroy() 112
 - GetPreferredRoom(IDType&) 114
 - Hydrate() 112
 - Import(IMCUnknown*, void*) 113
 - Initialize(IMCUnknown*) 112
 - SavesInNativeFormat() 115
 - SubModuleAboutToBeDestroyed(I3DShModule*) 114

SubModuleAboutToBeHydrated(I3DShModule*) 114
SubModuleClosedMainWindow(I3DShModule*) 114
WantsToBeInCurrentRoom() 114
I3DExModule interface 111
I3DExPostRenderer interface 117, 119, 225
I3DExShader interface 131
I3DExtension interface 15
I3DExTweeners

Inverse() 99
I3DExTweeners interface 99, 135
I3DExTweenersEdition interface 137
I3DExVolumePrimitive interface 145, 149
I3DExWireFrame interface 139
I3DShAtmosphericShader

GetScene(I3DShScene**) 181
I3DShAtmosphericShader interface 181
I3DShBackdrop

GetScene(I3DShScene**) 183, 185
I3DShBackdrop interface 183, 185
I3DShCamera interface 187
I3DShEnvironment

DirectionFilter(const TVector3 ,const TVector3&, TMCColorRGB&) 196
GetBackdropColor(const TVector2&,boolean&,const TBBox2D&, TMCColorRGB &) 196
GetEnvironmentColor(const TVector3&, boolean&, TMCColorRGB&) 197
HasAtmosphere() 197
HasBackdrop() 197
HasBackground() 197
SegmentFilter(const TVector3& ,const TVector3&, TMCColorRGB&, boolean) 195
SegmentFilter(const TVector3&,const TVector3&,TMCColorRGB&) 196
I3DShEnvironment interface 195
I3DShMasterShader

Clone(I3DShMasterShader**, ECloneMode) 228
GetName(TMCString&) 227
GetScene(I3DShScene**) 228
GetShader(I3DShShader**) 228
IsDefaultMaster() 228
SetShader(I3DShShader*, ECloneMode) 227
I3DShObject 201
I3DShPostRenderer

BeginRender 226
I3DShTreeElement

BeginMove () 245
 CanBeDistorted () 245
 ComClone (I3DShTreeElement**, ECloneMode, boolean) 244
 EndMove(boolean) 246
 GetBoundingBox (TBBBox3D&, ETreeDeformBBoxModes) 245
 GetBoundingBoxWithoutLightsAndCameras(TBBBox3D&) 245
 GetScene (I3DShScene* scene) 241
 GetSceneNoAddRef() 241
 GetXYZScaling(TVector3&) 246
 InvalidateBoundingBox () 244
 IsBBoxPickable 256
 PostMoveChange (boolean) 245
 TranslateHotPoint (TVector3&) 244
 I3DShTreeElement interface 223, 227, 229, 235, 239
 I3DShTweenener
 Edit(I3DShTweenener**,I3DShTweenenerChainLink*) 259
 GetLinkNoAddRef() 260
 SetLink(I3DShTweenenerChainLink*) 260
 I3DShTweenener interface 259
 I3DShTweenenerChainLink
 Clone (I3DShTweenenerChainLink**, I3DShParamTimeLine*) 261
 CopyData (I3DShTweenenerChainLink*) 262, 267
 GetFatherNoAddRef() 265
 GetKeyFrame (I3DShKeyFrame**) 263
 GetKeyFrameNoAddRef () 263
 GetNext (I3DShTweenenerChainLink**) 264
 GetNextNoAddRef() 263
 GetPrev (I3DShTweenenerChainLink**) 264
 GetPrevNoAddRef() 264
 GetTime() 263
 GetTimeInfo(TTimeInfo&) 265
 GetTweenener (I3DShTweenener**, boolean, boolean) 262
 GetTweenenerNoAddRef (boolean, boolean) 262
 InsertAfter(I3DShTweenenerChainLink*) 265
 IsKeyFrameSelected() 265
 IsTweenenerSelected() 265
 OffsetLink(MicroTick) 265
 RemoveFromChain() 265
 SelectKeyFrame(boolean) 266
 SelectTweenener(boolean) 266
 SetKeyFrame (I3DShKeyFrame**) 263, 269
 SetNext (I3DShTweenenerChainLink*) 264
 SetPrev(I3DShTweenenerChainLink*) 264
 SetTime (MicroTick) 263
 SetTweenener (I3DShTweenener*) 262
 I3DShTweenenerChainLink interface 261, 267

Idle Frequency 27
IExDataExchange interface 33
IExRegisterer interface 43
Import(IMCUnknown*, void*) 113
IncrementFreedomValue() 87, 248
IncrementProgress 168
Init 209
Initialize(IMCUnknown*) 112
InsertAfter(I3DShTweenChainLink*) 265
InsertDataComponent (I3DShDataComponent*) 251
InsertDataComponentByID (int32, int32) 251
InsertModifier(I3DShModifier* , uint32, ECloneMode) 248
instance
 object instance 272
interpolating in a facet 282
Inval 165
InvalidatePreview 158
IsCommandDown 156
IsControlDown 156
IsDefaultComponent 157
IsEqual 33
IsEqualTo() 131
IShAction interface 153
IShActionManager interface 155
IShComponent interface 157
IShComponentUtilities interface 159
IShParameterComponent interface 163
IShUtilities interface 167
IsInfiniteObject 203
IsKeyFrameSelected() 265
IsLongStaticClick 156
IsOpen () 252
IsOptionDown 156
IsPentiumIIIProcessor 176
IsPentiumIIProcessor 176
IsPentiumProcessor 176
IsShiftDown 156
IsShortStaticClick 156
IsSMPAware()
 I3DExFinalRenderer 97
IsTweenSelected() 265

K

kShModuleToolDropType 30

L

light 272

Light Source 272
LinksChanged () 249
Local Coordinate System 274
local coordinate system 274

M

Master Objects 271
MATRIX3D 48

N

NeededDiskSpace 37
NeedsToPrepareMenus 154
normal
 patch normal 284
NUM3D 45, 278

O

object 271
Object Coordinate System 274
object coordinate system 274
object instance 272
Object List 271
Objects 271
OffsetLink(MicroTick) 265

P

parametric mapping 281
patch 283
 normal 284
 u,v space 284
PATCH3D 63
PostAction 155
PostMouseAction 155
Prepare 33, 37, 41
PrepareDraw() 224
 I3DExFinalRenderer 95
Primitive 271
Project3DTo2D()
 I3DExCamera 83

R

Random 170
RandomStep 170
Ray3D 65
RayHit()

I3DExGeometricPrimitive 143
RayHit3D 66
RayHitParameters 67
ReadComponent 161, 162
ReadComponentData 162
Redo 153
ReleaseData (TDataKey) 252
RemoveDataComponent (uint32) 252
RemoveFromChain() 265
RemoveModifier 248
RemoveModifier(uint32) 248
RemoveWireFrame (I3DExWireFrame*) 256
RestoreResources 173
rotation matrix 279

S

SavesInNativeFormat() 115
Scene
 Object List 271
 Objects 271
 Scene Tree 271
 Tree Elements 271
scene 271
Scene Instancing 271
Scene Tree 271
scene tree 271
Screen Coordinate System 275
screen pixels space 277
SegmentFilter()
 I3DExAtmosphericShader 75
SegmentFilter(const TVector3&, const TVector3&, TMCColorRGB&, boolean) 195
SegmentFilter(const TVector3&, const TVector3&, TMCColorRGB&) 196
SelectKeyFrame(boolean) 266
SelectTweenener(boolean) 266
SetAmbientLight()
 I3DExFinalRenderer 94
SetAnimationMethod (I3DShAnimationMethod*, ECloneMode, real) 253
SetBoundingBox()
 I3DExModifier 107
SetCamera()
 I3DExFinalRenderer 93
SetColor 269
SetConstraint (I3DShConstraint*, ECloneMode) 250
SetConstraintByID (int32) 250
SetCursor 177
SetData (TDataKey, IMCUnknown*) 252
SetEnvironment()

- I3DExFinalRenderer 94
- SetFieldRenderingData()
 - I3DExFinalRenderer 94
- SetFromUndoInfo (const TTreeUndoInfo&, const TTreeUndoInfo*) 255
- SetGlobalTransform3D (const TTransform3D&, int16) 254
- SetGlobalTreeTransform (const TTreeTransform&, int16, boolean) 254
- SetHasUVSpace 210
- SetKeyFrame (I3DShKeyFrame**) 263
- SetLastOpenDocPath 176
- SetLastSaveDocPath 176
- SetModifiers (const TMCCountedPtrArray*, ECloneMode) 249
- SetName() 201
- SetNext (I3DShTweenChainLink*) 264
- SetParameter 34, 164
- SetPoint 268
- SetPrev(I3DShTweenChainLink*) 264
- SetPreview 38
- SetProgressPrompt 168
- SetProgressValue 169
- SetProgressValue0to1 169
- SetRandomSeed 170
- SetScene 223
- SetScene() 187
- SetSelectedState 270
- SetTime (MicroTick) 263
- SetTransform()
 - I3DExCamera 81
- SetTreeTop()
 - I3DExFinalRenderer 93
- SetTween (I3DShTween*) 262
- SetupResources 173
- SetUVSpace 210
- shading 282
 - parametric mapping 281
- ShadingFlags 59
- ShadingIn 57
- ShadingInOut 62
- ShadingOut 61
- ShellUtilitiesInit() 15
- StringWidth 176
- SubModuleAboutToBeDestroyed(I3DShModule*) 114
- SubModuleAboutToBeHydrated(I3DShModule*) 114
- SubModuleClosedMainWindow(I3DShModule*) 114
- surface normal 280
- surface point 280

T

TBBOX2D 47
TestHit 269
texture coordinate system 281
TickCount 170
TrackWireFrame 139
Transform 269
TRANSFORM3D 51
tree element 271
Tree Elements 271
tree elements transformation 279
TREETRANSFORM3D 52

U

u,v space 281
 patch u,v space 284
Undo 153
units system 278
UnregisterComponents 43
UseMMX 176
UV Space 281
UVShMaps 62
UVSpaceInfo 62

V

VECTOR2D 45
VECTOR3D 47
VERTEX3D 63

W

WantsOptionDialog 37, 41
WantsPreview 38
WantsToBeInCurrentRoom() 114
WillCauseChange 153
WireFrameToData 140
Working Box Coordinate System 274
working box coordinate system 274
WriteComponent 161

Y

YieldProcesses 169