FBX SDK

FBX SDK Programmer's Guide 2011

April 2010

Autodesk® FBX® 2011 SDK

Contents

Welcome to FBX SDK

1

Welcome to the documentation for Autodesk FBX SDK 2011. Get started by looking at any of these topics:

- What's new/What's changed on page 3
  Summarizes differences between this version of FBX SDK and the previous version.
- Introduction on page 3
  Introduces FBX technology, the *.fbx* file format, and FBX SDK.
- Installing and Configuring on page 17
  Shows you how to install FBX SDK on your development platform, and how to configure your development tools.
- Sample programs on page 25
  Describes the sample programs included in FBX SDK.
- Getting started with file import/export on page 41
  Shows you how to get started using FBX SDK to import, convert, and export files in any of the file formats supported by FBX technology.
- Scripting with Python FBX on page 101
  Python programmer? Start here!

For detailed information on each class, method, enumeration types, and so on, consult *FBX SDK Reference*, part of *FBX SDK Help*. *FBX SDK Help* is located in the `<yourFBXSDKpath>\doc` directory of the FBX SDK distribution.

NOTE For the most recent information and downloads, visit *http://www.autodesk.com/fbx*

Introduction 2

Autodesk® FBX® SDK is a C++ software development kit (SDK) that lets you create plug-ins, converters, and other applications that use Autodesk FBX technology. FBX technology allows you to translate and exchange 3D assets and media from a variety of sources quickly and easily.

What's new/What's changed

Details about all changes since the previous version of FBX SDK are in `readme.txt`, located in the directory where you installed FBX SDK.

Here are the highlights:

New animation system

The FBX animation system has been completely redesigned.

Take nodes, take node containers, current takes, and FCurves have all been replaced by animation stacks (`KFbxAnimStack`), animation layers (`KFbxAnimLayer`) animation curve nodes (`KFbxAnimCurveNode`), and animation curves (`KFbxAnimCurve`).

The *animation stack* is now the highest-level container for animation. You can think of an animation stack as a stack of *animation layers*, e.g., Layer0, Layer1, Layer2, ... etc. An animation stack must contain at least one animation layer (Layer0, also called the base layer).

Advantages of the new animation system

- Supports blended animation.
- Allows you to export a scene to an FBX file that contains nothing but animation curves. You can then import the animation curves into a scene that contains the characters or other models to be animated.

The new animation system is more transparent, more consistent, and easier to learn:

- All the new animation classes are FBX objects (subclasses from `KFbxObject`), and therefore inherit a rich library of member functions.
- The data managed by each class are contained in FBX properties (see FBX properties), which means that you can query, get, and set data using the member functions of `KFbxProperty` and `KFbxTypedProperty`.
- Objects connect to each other with OO and OP connections (see Connections), which means that you can query objects to determine their relationships to other objects.

- Animation curves can now be used in an order-independent way. This was not possible in the old animation system.
- Important concepts such as animation stacks and evaluation are encapsulated in classes (`KFbxAnimStack` and `KFbxAnimEvaluator` respectively), rather than scattered among several classes.

New evaluation system

All evaluation can be done with functions of class `KFbxAnimEvaluator`. These functions replace `GetGlobalFromCurrentTake()` and all its variants.

You simply query the scene for its evaluator, and then you can query the evaluator for the global or local positions of any nodes, as well as the value of any property, at the specified time.

The evaluation data is now stored in `KFbxAnimEvalState`, rather than in `KFbxNode` objects.

Advantages of the new evaluation system

- Reduces the memory used by `KFbxNode`.
- Allows you to re-use animation data. For example, you can evaluate different times while keeping the evaluation results in buffers that you can manage.
- You can write your own evaluator by creating a subclass of `KFbxAnimEvaluator`. That's what we did for `KFbxAnimEvalClassic`, FBX SDK's default evaluator.

Scripting with Python FBX

Python FBX is a *Python binding* for the C++ library of FBX SDK. It allows you to write Python scripts that can use most of the classes and member functions of FBX SDK.

New file format: FBX 7.1

The default file format for FBX SDK is now 7.1. This new file format permits smaller FBX files and therefore faster loading and saving of FBX files.

Instancing

If a scene has 50 robots, and the robots are all alike, *instancing* means that the FBX file needs to store only one set of mesh data for all the chairs.

Levels of detail for meshes

A new class, `KFbxLodGroup`, allows you to have different versions of a mesh, each with its own level of detail (LOD).

File referencing

A new class, `KFbxReference` supports the file referencing feature in Autodesk Maya.

What you must know

The documentation for FBX SDK assumes that you have very good knowledge of:

- Object-oriented programming in the C++ programming language, including templates, virtual functions, containers, etc.
- 3D graphics, including modeling and animation.

- Your development tools (Microsoft Visual Studio, Xcode, gcc, ...).

What is Autodesk FBX technology

Autodesk FBX SDK is part of Autodesk FBX technology, a family of tools that allow 3D content developers to import and export 3Ddata. Autodesk FBX enables organizations creating films, games, etc., to design workflows built around multiple 2D and 3D digital content creation applications.

The FBX family of tools includes:

- FBX file format for 3D scenes on page 5.
- FBX plug-ins for Autodesk 3ds Max and Autodesk Maya on page 7. These plug-ins import and export FBX files.
- FBX Converter on page 8, a file conversion utility.
- FBX for QuickTime on page 9, for viewing and interacting with 3D scenes.
- FBX Software Development Kit on page 9, which lets you create applications, plug-ins, etc.

See also:

- What content developers can do with FBX technology on page 10.
- What designers can do with FBX technology on page 10
- Applications that support FBX technology on page 10

FBX file format for 3D scenes

FBX files (*.fbx*) are normally saved in a binary (or native) format, but they can also be saved in ASCII format. Binary FBX files and ASCII FBX files both use the *.fbx* filename extension.

Here is an shortened version of a small FBX file in ASCII format. Deleted lines are indicated by . . ., and we have manually added some comments. Comments begin with a semicolon (“;“) anywhere on a line.

```
; FBX 7.1.0 project file
; Copyright (C) 1997-2010 Autodesk Inc. and/or its licensors.
; All rights reserved.
; ----------------------------------------------------
FBXHeaderExtension: {
       ; header information: global file information.
 FBXHeaderVersion: 1003
 FBXVersion: 7100
 CreationTimeStamp:  {
 Version: 1000
 Year: 2010
 Month: 1
 Day: 19
 Hour: 16
 Minute: 30
 Second: 28
 Millisecond: 884
 }
 Creator: "FBX SDK/FBX Plugins version 2011.2"
 SceneInfo: "SceneInfo::GlobalInfo", "UserData" {
 ...
}
GlobalSettings: {
 Version: 1000
 Properties70:  {
 P: "UpAxis", "int", "Integer", "",1
 P: "UpAxisSign", "int", "Integer", "",1
 P: "FrontAxis", "int", "Integer", "",2
 P: "FrontAxisSign", "int", "Integer", "",1
 P: "CoordAxis", "int", "Integer", "",0
 P: "CoordAxisSign", "int", "Integer", "",1
 P: "OriginalUpAxis", "int", "Integer", "",-1
 P: "OriginalUpAxisSign", "int", "Integer", "",1
 P: "UnitScaleFactor", "double", "Number", "",1
 P: "OriginalUnitScaleFactor", "double", "Number", "",1
 P: "AmbientColor", "ColorRGB", "Color", "",0,0,0
 P: "DefaultCamera", "KString", "", "", "Producer Perspective"
 P: "TimeMode", "enum", "", "",6
 P: "TimeSpanStart", "KTime", "Time", "",0
 P: "TimeSpanStop", "KTime", "Time", "",46186158000
 }
}
 ...
; Object definitions
;------------------------------------------------------------------
Definitions: {
 Version: 100
 Count: 2251
 ObjectType: "GlobalSettings" {
 Count: 1
 }
 ObjectType: "Model" {
 Count: 86
 PropertyTemplate: "KFbxNode" {
 Properties70:  {
 P: "QuaternionInterpolate", "bool", "", "",0
```

```
    P: "RotationOffset", "Vector3D", "Vector", "",0,0,0
    P: "RotationPivot", "Vector3D", "Vector", "",0,0,0
    P: "ScalingOffset", "Vector3D", "Vector", "",0,0,0
    P: "ScalingPivot", "Vector3D", "Vector", "",0,0,0
    ...}
    ObjectType: "Material" {
    Count: 1
    PropertyTemplate: "KFbxSurfacePhong" {
    Properties70:  {
    P: "ShadingModel", "KString", "", "", "Phong"
    P: "MultiLayer", "bool", "", "",0
    P: "EmissiveColor", "ColorRGB", "Color", "",0,0,0
    P: "EmissiveFactor", "double", "Number", "",1
    P: "AmbientColor", "ColorRGB", "Color", "",0.2,0.2,0.2
    ...}
    Model: 21883936, "Model::Humanoid:Hips", "LimbNode" {
    Version: 232
    Properties70:  {
    P: "ScalingMin", "Vector3D", "Vector", "",1,1,1
    P: "NegativePercentShapeSupport", "bool", "", "",0
    P: "DefaultAttributeIndex", "int", "Integer", "",0
    P: "Lcl Translation", "Lcl Translation", "", "A+",-271.281097412109,-
  762.185852050781,528.336242675781
    P: "Lcl Rotation", "Lcl Rotation", "", "A+",-1.35128843784332,2.6148145198822,0.42334708571434
    P: "Lcl Scaling", "Lcl Scaling", "", "A+",1,0.99999988079071,1
    ...
```

As you can see, FBX files store data about cameras, lights, meshes, NURBS, and the other *elements* of a 3D scene. Applications such as Autodesk 3ds Max, Autodesk Maya, and Autodesk MotionBuilder can import all or part of the scene data from an FBX file, or export data from their scenes to an FBX file.

NOTE The FBX file format is not documented. Applications use FBX SDK to import scene data to and from FBX files (and other file formats supported by FBX. The FBX plug-ins for Autodesk 3ds Max and Autodesk Maya on page 7 are examples of programs that use FBX SDK to import and export scene data.

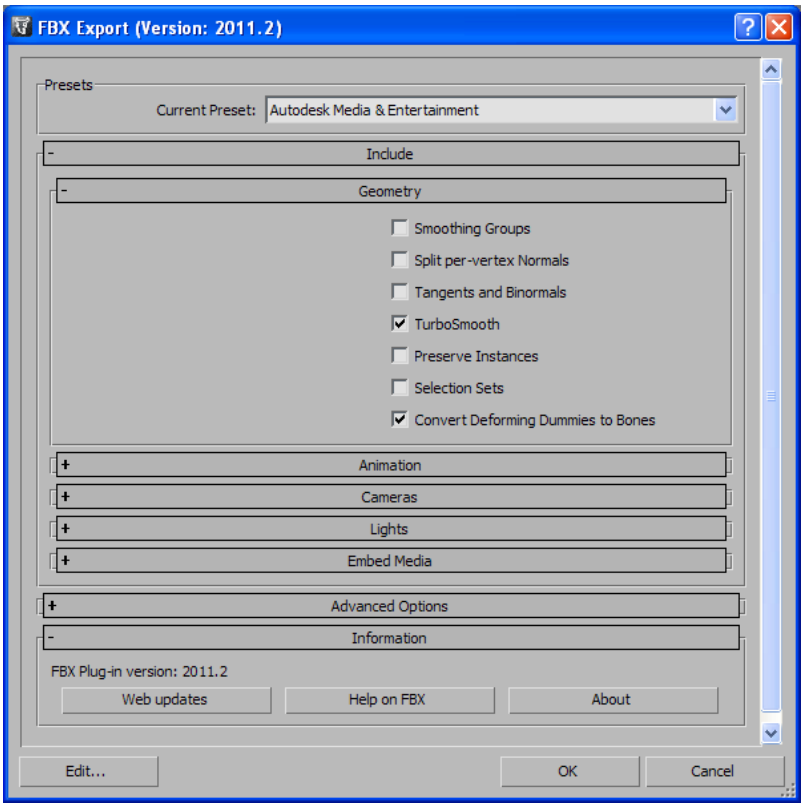
Binary FBX files and ASCII FBX files both use the *.fbx* filename extension. You can convert FBX files from binary to ASCII using FBX SDK, of course, but you can also use a utility program: see FBX Converter on page 8.

One way to get a feel for FBX technology is to view a small ASCII FBX file in a 3D viewer, study the file in a text editor, and study the program that created the file or that processes it in some way.

NOTE For a free viewer of FBX files, see FBX for QuickTime on page 9. All sample programs (see Sample programs on page 25) can import or export FBX files in ASCII format.

FBX plug-ins for Autodesk 3ds Max and Autodesk Maya

3ds Max allows users to import all or part of a scene stored in an FBX file into a 3ds Max scene, and to export all or part of a 3ds Max scene to an FBX file. Here is the FBX Export dialog box for 3ds Max:



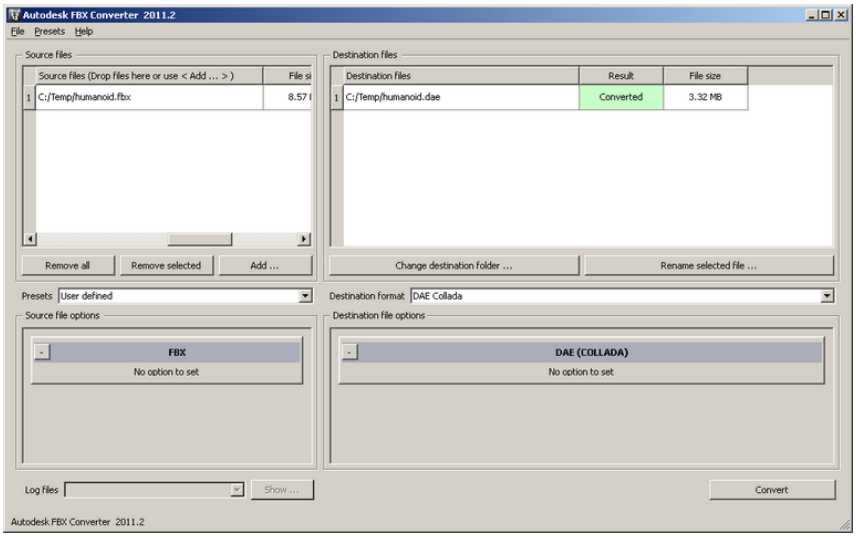
Maya provides equivalent import/export functionality for Maya scenes.

Both 3ds Max and Maya provide FBX functionality as a plug-in. This means users can upgrade to a new FBX plug-in without having to wait for a new version of 3ds Max or Maya.

The plug-ins are written with FBX SDK. They import and export not only FBX files, but some of the other file formats supported by FBX technology.

FBX Converter

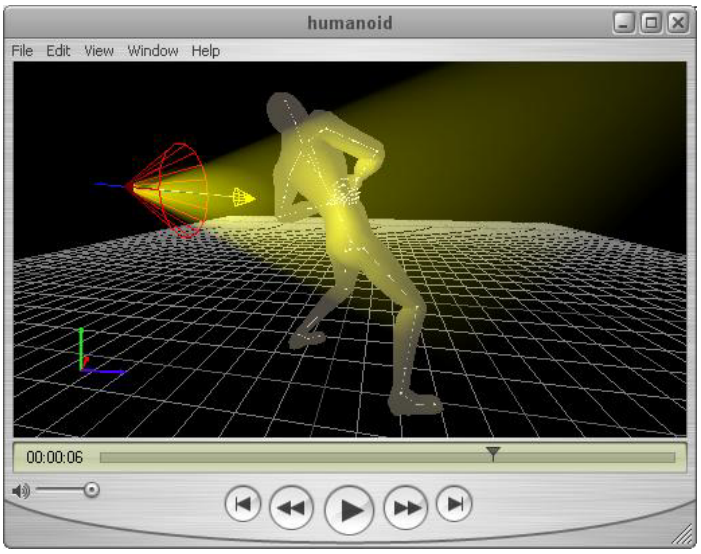
FBX Converter is a standalone application that lets you convert FBX files into other file formats and versions, as well as from other file formats into FBX files.



FBX Converter uses FBX SDK to perform all these conversions.

FBX for QuickTime

FBX for QuickTime is a component plug-in for Apple QuickTime that lets you play back and interact with FBX files inside QuickTime.



You can interact with the scene by changing a camera's position, hiding or displaying grid marks, switching between multiple cameras, switching between multiple takes of animation, etc.

FBX for QuickTime (which uses FBX SDK) is useful for anyone who wants to view and manipulate FBX files without installing and running a 3D modeling application.

FBX Software Development Kit

The FBX Software Development Kit (FBX SDK) allows software developers to create applications that use FBX technology, or to integrate FBX technology into existing applications. The documentation that you are now reading is for FBX SDK.

For more information on FBX SDK:

- FBX SDK features on page 11

FBX Extensions SDK

FBX Extensions SDK allows you to create extensions to Autodesk FBX technology:

- Extensions to FBX Plug-in for 3ds Max/Maya.
- Extensions to FBX SDK to support your own custom file formats.

NOTE FBX Extensions SDK also provides the source code for the exporters and importers used by FBX SDK to write and read files in two file formats: FBX and Collada.

What content developers can do with FBX technology

Content developers are the people who work on 3D modeling and animation for films, television and games. Here are some of the ways that content developers use FBX technology:

Sharing scene assets (interoperability)	Moving models and other scene assets from tool to tool, and from production house to production house. For example, Studio 1 creates a character in 3ds Max, then exports it to an FBX file. Studio 2 imports the FBX file into MotionBuilder, adds motion capture data to the character, then exports it to an FBX file. Studio 3 imports the second file into Maya, touches up the model's skinning, then exports it to a third FBX file. Studio 4 imports the third file into its proprietary tool... and so forth.
Storing scene assets	Storing scene assets in a durable file format. Each new release of FBX can read earlier versions of FBX files.
Converting data	Converting data using conversions built in to FBX SDK. For example: converting textures to TIFF; converting NURBS and patches to meshes.
Processing animation	For example, applying a filter to animation curves.
Packaging models for sale	Vendors of 3D models use FBX as a file format.

What designers can do with FBX technology

Architects, designers, engineers, and design visualization specialists use products such as Autodesk 3ds Max Design and Autodesk Revit Architecture to design and visualize buildings and other objects.

These designers use FBX technology to save models, metadata, and other assets in a file format that can be used by the content developers who prepare the advertising and marketing collateral.

Applications that support FBX technology

The following Autodesk products use FBX technology to import and export files:

Product	Description
Autodesk 3ds Max and Autodesk 3ds Max Design	3D animation, modeling, and rendering solution. Includes plug-in to import and export FBX files.
Autodesk Maya	3D modeling, animation, and rendering solution. Includes plug-in to import and export FBX files.
Autodesk MotionBuilder	Productivity suite for 3D character animation. Imports and exports FBX files.

Product	Description
Autodesk Softimage	A complete 3D software for visual effects and game production. Imports and exports FBX files.
Autodesk Softimage Mod Tool	A free 3D modeling and animation package for creating non-commercial games and modding (modifying games). Exports FBX files.
Autodesk Mudbox	Digital sculpting and texture painting software for 3D modelers and texture artists. Imports and exports FBX files.
Autodesk Flame	Real-time visual effects design and compositing system. Imports and exports FBX files.
Autodesk Flint	Advanced visual effects system for post-production and broadcast graphics. Imports and exports FBX files.
Autodesk Inferno	Interactive design system for high-resolution visual effects. Imports and exports FBX files.
Autodesk Smoke	Integrated editing and finishing system for SD, HD, 2K film and above. Imports and exports FBX files.
AutoCAD Revit Architecture	Building information modeling (BIM) application. Exports FBX files.
AutoCAD	General CAD modeling package. Imports and exports FBX files.

As well, many third-party software products use FBX SDK to import and export files. For more information, see *http://www.autodesk.com/fbx*.

FBX SDK features

This section summarizes the scene elements supported by FBX SDK, the file formats that it imports and exports, the texture file formats that it can embed or reference in an FBX file, and other features of FBX SDK.

Scene elements supported

The FBX SDK lets you access, create, or modify the following elements of a scene:

- Mesh, NURBS, patch, trimmed NURBS, NURBS curves (all referred to as *geometries*).
- Texture mapping over a geometry.
- Material mapping over a geometry.
- Normals, color of vertex, edge visibility, smoothing groups, and user-defined attributes over a mesh.

- Cluster constraints on the control points of a geometry.
- Shape constraints on the control points of a geometry.
- Vertex cache animation on the control points of a geometry.
- Scene settings that provide Up-Axis (X/Y/Z) and scene scaling (units).
- Position, Rotation, Scale, Parent, Single Chain IK, and Aim constraints.
- Multiple cameras. Includes stereo cameras for 3D.
- Multiple lights and gobos.
- Markers.
- Nulls.
- Skeleton segments (root, limb, and limb node).
- Animation curves.
- Global camera, light, and time settings.
- Bind pose for a list of nodes (bones and geometry).
- Rest pose for a list of nodes (bones and geometry).

In addition, the FBX SDK provides tools to:

- Manage and evaluate animation data, including blended animation.
- Triangulate meshes, NURBS, and patches.

File formats imported and exported

FBX SDK:

- Imports FBX files compatible with FBX file format versions 7.1, 7.0, 6.1, and 6.0
- Exports FBX files compatible with FBX file format version 7.1 (default), 7.0, and 6.1.

FBX SDK imports and exports FBX files compatible with the following applications:

Software	Version
MotionBuilder	Version 5.5 and later.
FBX Plug-in for 3ds Max	All versions.
FBX Plug-in for Maya	All versions.
Flame	Version 8.0 and later.
Flint	Version 8.0 and later.
Inferno	Version 5.0 and later.
Smoke	Version 6.0 and later.

Software	Version
Revit Architecture	Version 2009.1 and later.
AutoCAD	Version 2011 and later.

As well, FBX SDK imports and exports files in the following file formats:

File format	Version
Autodesk AutoCAD DXF (*.dxf*)	Version 13 and earlier.
Collada DAE (*.dae*).	Version 1.5 and earlier.
3D Studio 3DS (*.3ds*)	All versions.
Alias OBJ (*.obj*)	All versions.

Texture file formats embedded or referenced

FBX SDK allows you to embed texture files in any 2D file format into an FBX file. This means you can deliver a scene as a single file.

Alternately, you can embed relative references to texture files, and then deliver the texture files along with the FBX file.

Sample programs and tutorials

The `\examples` directory of the FBX SDK distribution contains a library of sample programs. You can use these samples as starting points for your own applications.

For more information, including a brief description of each sample program, see Sample programs on page 25.

Platform requirements

FBX SDK runs on the 32- and 64-bit versions of these platforms:

Platform	Requirements
Microsoft Windows	Windows XP (SP 2 and above). Windows Vista. Windows 7.
Linux	Any Linux implementation that provides GCC version 4.0.2 and above.
Mac OS	Versions 10.4 and above, with Intel processors.

For more information, see Recommended development environments on page 17.

Sources of information

Source	Notes
FBX SDK Programmer's Guide	The document you are now reading. You may be reading it as a PDF file, or as part of *FBX SDK Help*.
FBX SDK Help	Help system distributed with FBX SDK. Includes: ■ *FBX SDK Programmer's Guide* (this document). ■ *FBX SDK Reference*, which containing documentation for each class, member function, enum, etc.
readme.txt	Distributed with FBX SDK. Contains last-minute documentation, as well as what's new and what's changed since the previous version of FBX SDK.
FBX Plug-in for 3ds Max; FBX Plug-in for Maya; FBX Converter; FBX for QuickTime	These FBX tools and their documentation are a good introduction to what you can do with FBX and therefore with FBX SDK. If you are not familiar with 3D modeling and animation concepts, consider learning how to use 3ds Max or Maya.
Discussion forums	Free, user-to-user discussion forums. See Discussion Forums on page 15.
http://www.autodesk.com/fbx	Home page for FBX. Look here for white papers, product sheets, new releases, and other information.
Compatibility/interoperability charts	Detailed information on how FBX supports the interchange of 3D data types between 3ds Max, Maya, MotionBuilder, and FBX file format. Available at *http://www.autodesk.com/fbx*.
The Area	Visit *http://area.autodesk.com*, the Autodesk community for digital entertainment and visualization. You'll find news, tutorials, discussion forums, downloads, etc.

Naming conventions

Header files and sample programs generally follow the following naming conventions:

Prefix	Notes
`KFbx`	Most FBX SDK class names begin with `KFbx`. Examples: `KFbxNode`, `KFbxScene`, `KFbxCamera`
`K`	Many utility classes begin with upper case "K". Examples: `KString`, `KTime`
`p`	Parameters passed to a member function begin with lower case "P". Examples: `pWriteFileFormat`, `pScene`, `pFilename`
`l`	Local variables begin with lower-case "L". Examples: `lWriteFileFormat`, `lScene`, `lFilename`
`g`	Global variables begin with lower-case "G". Examples: `gStart`, `gStop`, `gCurrentTime`
`m`	Member data (member variables) begin with lower-case "M". Examples: `mDescription`, `mImportname`, `mSelect`

Getting technical support

Autodesk provides developers with technical support, discussion forums, and the email address for the FBX SDK development team.

Autodesk Developer Network (Sparks)

Technical support for the FBX SDK is provided through the Autodesk Developer Network (ADN) Sparks 3rd party developer program. The ADN Sparks program is based on an annual membership fee. Program details, benefits, and pricing are found on the Autodesk Developer Center web page (*http://www.autodesk.com/adn*).

Discussion Forums

For free, user-to-user discussion forums for developers and others, visit the FBX section of AREA, Autodesk's community for digital entertainment and visualization: *http://area.autodesk.com/forum/autodesk-fbx/*

Email the FBX SDK development team

The development team for FBX SDK welcomes your feedback at *fbxsdk@autodesk.com*.

Installing and Configuring 3

This section describes:

- How to install FBX SDK.
- How to configure your development environment for FBX SDK.

NOTE If you are planning to use Python FBX rather than the C++ interface to FBX SDK, see Scripting with Python FBX on page 101.

Recommended development environments

The following table lists the platforms and compilers recommended for developing plug-ins, converters, and other applications with FBX SDK:

Supported platform	Supported development environment or compiler
Windows XP (SP2 and above). Windows Vista. Windows 7.	Visual Studio 2005. Visual Studio 2008.
Linux	Any implementation that provides gcc version 4.0.2 and above.
Mac OS X	gcc 4.0.x on Intel x86.

Notes:

- If your platform or compiler is not listed above, you may not be able to compile programs that use FBX SDK.
- For information about which platforms can run applications developed with FBX SDK, see Platform Requirements.

Downloading and installing

There are separate distributions of FBX SDK for Linux, Mac OS, and Windows.

The structure of the FBX SDK distribution directory is, however, identical on all these platforms. See Directory structure on page 18.

For each platform, there is more than one version of library file for FBX SDK. See Runtime libraries available in different flavors on page 18.

To download and install FBX SDK, follow the instructions for your development platform, either Linux, Mac OS, or Windows on page 20.

Directory structure

The structure of the FBX SDK distribution directory (*<yourFBXSDKpath>*) is identical on all platforms:

Directory	Description
<yourFBXSDKpath>\	The root of the FBX distribution directory. Contains a readme file, the license text, and the uninstaller.
*<yourFBXSDKpath>**doc*\	Contains: ■ *FBX_SDK_Programmers_Guide_2011_2.pdf* ■ A directory containing *FBX SDK Help* ■ *FBX_SDK_Help.html* To start *FBX SDK Help*, open *FBX_SDK_Help.html*.
*<yourFBXSDKpath>**examples*\	Sample programs, each in its own subfolder. Contains platform-specific project files, makefiles, etc.
*<yourFBXSDKpath>**include*\	Header files for FBX SDK.
*<yourFBXSDKpath>**lib*\	Library runtimes for FBX SDK.

Runtime libraries available in different flavors

The library runtime files (located in the *<yourFBXSDKpath>**lib*\ subdirectory of your FBX SDK distribution) are different for each platform (Linux, Mac OS, and Windows).

Library runtimes for Windows

You can link your application to a runtime library for Windows which is:

- For Visual Studio 2008 or 2005.
- For projects that build statically linked libraries (`/MT` compiler option) or dynamically linked libraries (`/MD`).
- For libraries in debug mode or release mode.

■ For 32-bit processors or 64-bit processors.

Library file name	Description
`fbxsdk_mt2008.lib`	Static, 32-bit, release, VS 2008
`fbxsdk_mt2008d.lib`	Static, 32-bit, debug, VS 2008
`fbxsdk_md2008.lib`	Dynamic, 32-bit, release, VS 2008
`fbxsdk_md2008d.lib`	Dynamic, 32-bit, debug, VS 2008
`fbxsdk_mt2008_amd64.lib`	Static, 64-bit, release, VS 2008
`fbxsdk_mt2008_amd64d.lib`	Static, 64-bit, debug, VS 2008
`fbxsdk_md2008_amd64.lib`	Dynamic, 64-bit, release, VS 2008
`fbxsdk_md2008_amd64d.lib`	Dynamic, 64-bit, debug, VS 2008
`fbxsdk_md2005_amd64.lib`	Dynamic, 64-bit, release, VS 2005
`fbxsdk_md2005.lib`	Dynamic, 32-bit, release, VS 2005
`fbxsdk_mt2005.lib`	Static, 32-bit, release, VS 2005
`fbxsdk_mt2005_amd64.lib`	Static, 64-bit, release, VS 2005
`fbxsdk_md2005_amd64d.lib`	Dynamic, 64-bit, debug, VS 2005
`fbxsdk_md2005d.lib`	Dynamic, 32-bit, debug, VS 2005
`fbxsdk_mt2005d.lib`	Static, 32-bit, debug, VS 2005
`fbxsdk_mt2005_amd64d.lib`	Static, 64-bit, debug, VS 2005

Notes:

■ All of these library runtimes are static libraries.

■ These libraries can all be linked to Microsoft's multithreaded C libraries. FBX SDK code, however, is not guaranteed to be thread-safe.

■ Libraries for 64-bit processors are named **_amd64.lib*. These libraries run on Intel and AMD 64-bit processors.

Library runtimes for Mac OS

Here are the names of the library runtimes for Mac OS:

Library file name	Description
`libfbxsdk_gcc4_ub.a`	Universal Binary: gcc 4.0 compiler, release, Intel 32-bit (i386) and Intel 64-bit (x86_64)

Library file name	Description
libfbxsdk_gcc4_ubd.a	Universal Binary: gcc 4.0 compiler, debug, Intel 32-bit (i386) and Intel 64-bit (x86_64) architectures

NOTE All of these library runtimes are static libraries.

Library runtimes for Linux

Here are the names of the library runtimes for Linux:

Library file name	Description
libfbxsdk_gcc4.a	gcc 4.0 compiler, release, 32-bit
libfbxsdk_gcc4d.a	gcc 4.0 compiler, debug, 32-bit
libfbxsdk_gcc4_x64.a	gcc 4.0 compiler, release, 64-bit
libfbxsdk_gcc4_x64d.a	gcc 4.0 compiler, debug, 64-bit

NOTE All of these library runtimes are static libraries.

Windows

Downloading and installing FBX SDK

To download and install FBX SDK on your Windows computer:

1. Go to *http://www.autodesk.com/fbx*.
2. Navigate to the Downloads page, and follow any instructions.
3. Find the version of FBX SDK 2011.2 for Windows.
4. Download the distribution file for Windows to your computer. The distribution file is a Setup program, i.e., an executable.
5. Execute the Setup program, and follow the instructions.
6. The Setup program will let you specify a destination folder. Specify a folder which is new or empty. We will call this as *<yourFBXSDKpath>*, your *distribution directory* for FBX SDK.
7. Read *<yourFBXSDKpath>\readme.txt*. This Readme file contains detailed information about changes in FBX SDK since the previous version, as well as any last-minute documentation.

Notes:

- The Setup program does not modify the Windows Registry or the Windows Start menu.
- You can have more than one version of FBX SDK installed on your computer, providing you install each version in a separate folder.

Uninstalling FBX SDK

To remove FBX SDK from your computer:

➤ Run *<yourFBXSDKpath>\uninstall.exe*.

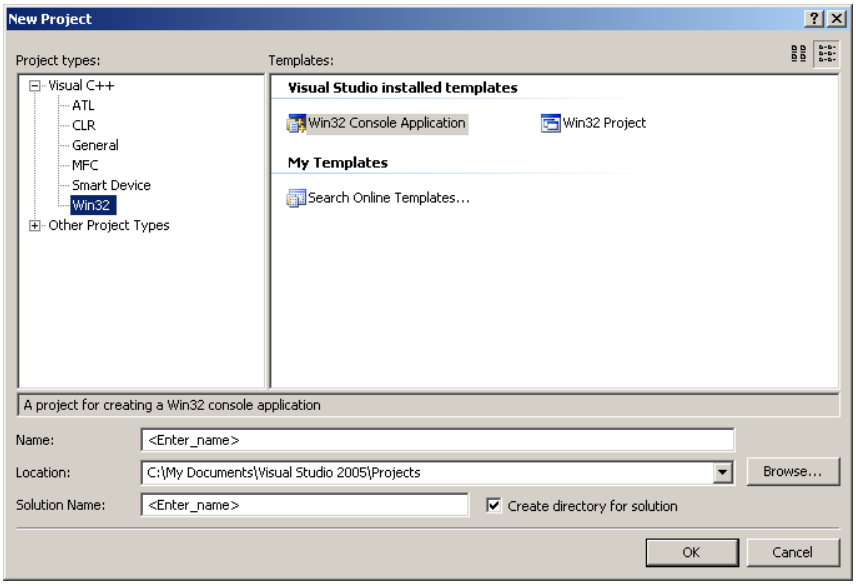
Configuring Visual Studio

The instructions in this section are based on Visual Studio 2005. Except where noted, there are no significant differences for Visual Studio 2008.

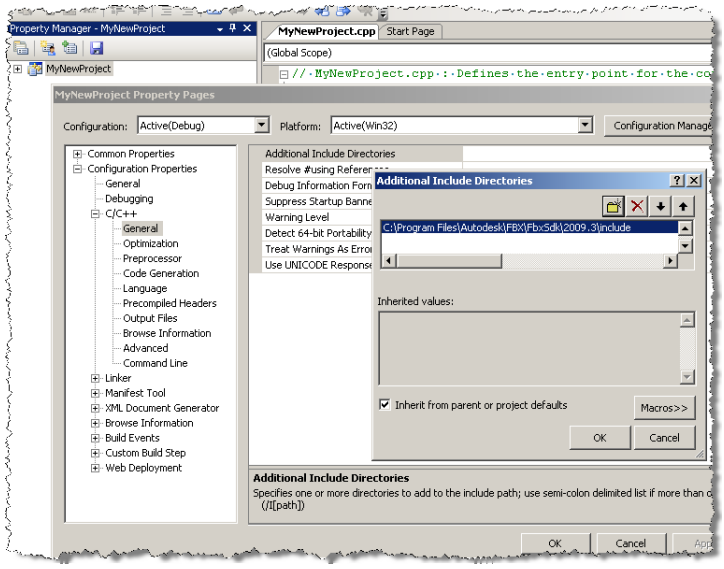
NOTE To build and run the sample programs for FBX SDK, see Building and running the sample programs on page 39. For general information, see Sample programs on page 25.

To create a new Visual Studio solution that uses FBX SDK:

1. Start Visual Studio.
2. To create a new project, choose New > Project from the File menu:



3. In Project Types, choose Visual C++ > Win32.
4. Continue as usual, then click Finish. Your new project and solution will appear.
5. To open the Property Manager tab, choose Property Manager from the View Manager:
6. In the Property Manager, right-click on the project, and choose Properties:



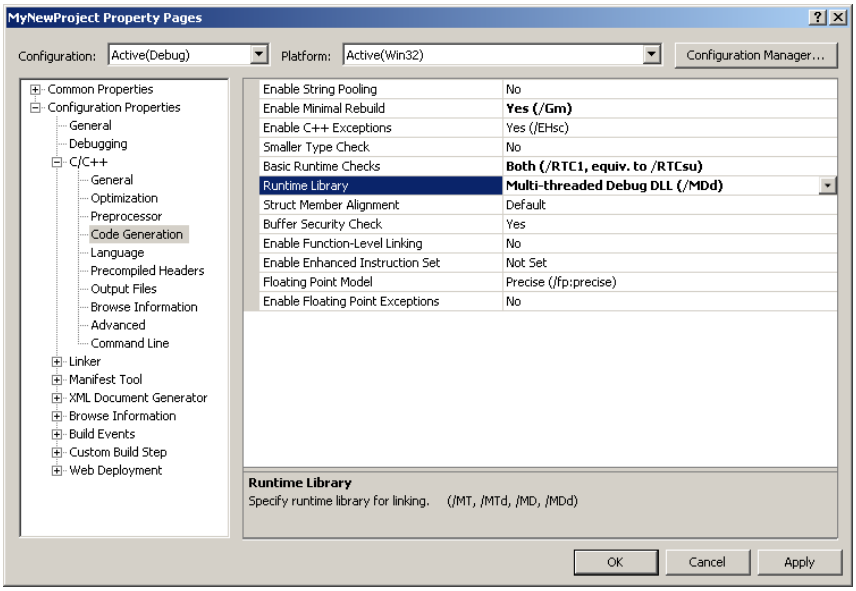
7 In the property tree on the left of the Property Pages dialog box, choose Configuration Properties > C/C++ > General.

8 In the properties sheet on the right of that same dialog box, choose Additional Include Directories. The Additional Include Directories dialog box will open (as shown before the previous step).

9 In the list box at the top of the Additional Include Directories dialog box, click the top blank line until you see a control to browse for directories.

10 Use this control to add the full path for the *<yourFBXSDKpath>\include* directory. By default, the path is: *C:\Program_Files\Autodesk\FBX\FbxSdk\2011.2\include\fbxfilessdk*

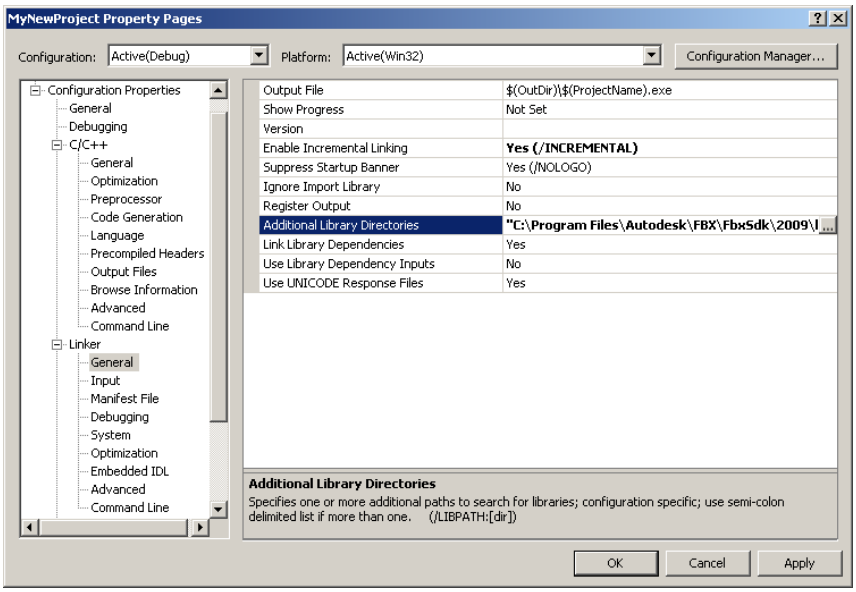
11 In the tree on the left of the dialog box, choose Code Generation. In the property sheet on the right, choose Runtime Library:



12 Use the drop-down list box to choose the type of run-time library that your project will be generating. The list correspond to Visual C++ compiler options: `/MD`, `/MDd`, `/MT`, `/MTd`, etc. Not all compiler options are available for all versions of Visual Studio (see Library runtimes for Windows on page 18).

13 In the Property Pages dialog box, choose Linker > General:

14 Go to Configuration Properties -> Linker:



15 Click Additional Library Dependencies, then use the browse control to enter the full path for the *<yourFBXSDKpath>\lib* folder of the FBX SDK distribution. By default, the full path is: *C:\Program_Files\Autodesk\FBX\FbxSdk\2011.2\lib*

16 Click Input on the left, then choose Additional Dependencies on the right. Use the Additional Dependencies dialog box to add the appropriate FBX SDK library. For a table describing the available FBX SDK library files, see Library runtimes for Windows on page 18.

17 If in the previous step, you selected an FBX SDK library that is either for creating /MTd or /MDd runtimes, then you must do the following:

Click Input on the left, then choose Ignore Specific Library on the right. In the Ignore Specific Library dialog box, enter `LIBMCT`.

18 Click Input on the left. In the Additional Dependencies field on the right, enter `wininet.lib`.

Macintosh OS

Downloading and installing FBX SDK

To download and install FBX SDK on your Macintosh computer:

1 Go to *http://www.autodesk.com/fbx*.

2 Navigate to the Downloads page, and follow any instructions.

3 Find the distribution file of FBX SDK 2011.2 for Mac OS.

4 Download the distribution file to your computer. The distribution file is a compressed Installer program (**.pkg.tgz*).

5 Uncompress the Installer.

6 Execute the Installer, and follow the instructions.

7 The Installer program will let you specify a destination directory. Specify a directory which is new or empty. We call this *<yourFBXSDKpath>*, your distribution directory for FBX SDK.

8 Read *<yourFBXSDKpath>\readme.txt*. This Readme file contains detailed information about changes in FBX SDK since the previous version, as well as any last-minute documentation.

Use sample makefiles as a guide

You will find sample programs in *<yourFBXSDKpath>\examples* (see Sample programs on page 25). Each sample program contains a makefile. Use one of these makefiles as a model for your own project.

Linux

Downloading and installing FBX SDK

To download and install FBX SDK on your Linux computer:

1 Go to *http://www.autodesk.com/fbx*.

2 Navigate to the Downloads page, and follow any instructions.

3 Find the distribution file of FBX SDK 2011.2 for Linux.

4 Download the distribution file to your computer. The distribution file is a compressed tar file (*.tar.gz*).

5 Extract the contents of the file to a new directory that we call *<yourFBXSDKpath>*, your distribution directory for FBX SDK.

6 Follow the instructions in *<yourFBXSDKpath>\Install_FbxFileSdk.txt*.

7 Read *<yourFBXSDKpath>/readme.txt*. This Readme file contains detailed information about changes in FBX SDK since the previous version, as well as any last-minute documentation.

Use sample makefiles as a guide

You will find sample programs in *<yourFBXSDKpath>\examples* (see Sample programs on page 25). Each sample program contains a makefile. Use one of these makefiles as a model for your own project.

Sample programs

4

The purpose of this section is to help you find the sample program that illustrates the programming problem that you need to solve.

FBX SDK includes two kinds of sample programs that demonstrate how to create features of the FBX SDK. See:

- Long tutorial programs aimed at developers who are new to FBX SDK. These programs are Windows-based, have a practical purpose, show results that are visible and meaningful, and introduce FBX features in a sequence that shows newcomers how to use FBX SDK to solve real problems.
- Brief sample programs that demonstrate particular features of the SDK. These programs are console-based, contain the minimum code to demonstrate one or two features, and may not show meaningful results. Some samples are very basic, while others are quite advanced.

Tutorial programs

These tutorial programs are aimed at programmers who are learning to use FBX SDK. These tutorials run on Windows only. Each tutorial program:

- Introduces FBX features and requirements in a sequence that suits learners.
- Has a practical purpose.
- Demonstrates typical workflows within an application.
- Show meaningful results.
- Corresponds to one or more tutorial sections of *FBX SDK Programmer's Guide*:

Tutorial program	Tutorial section	Description
ImportExport tutorial program on page 31	Getting started with file import/export on page 41.	Shows how to import a file and how to export a file. Builds and runs on Windows only.
SceneTreeView tutorial program on page 32	Traversing the Scene Graph on page 49.	Shows how to traverse all the nodes in an FBX scene, and how to determine the content of each node, i.e., whether a node contains a camera, a light, a mesh, etc. Builds and runs on Windows only.

Tutorial program	Tutorial section	Description
CubeCreator tutorial program on page 34	Applying Textures and Materials to Meshes on page 57.	Shows how to add textures, materials, and animation to meshes representing a model. Builds and runs on Windows only.
CubeCreator tutorial program on page 34	Animating a Scene on page 69.	Shows how to animate meshes, cameras, and other objects in a scene.

Finding the source code and project files

Folders for each of these programs are located in `<yourFBXSDKpath>\examples\UI Examples\`. Each folder contains the source code for the tutorial program and Visual Studio project files to help you build and run the program.

Common code used by all tutorial programs

The folder `<yourFBXSDKpath>\examples\UI Examples\Common\`contains utility functions used by the tutorial programs.

It also contains `librarychooser.h`, a header file that selects the appropriate FBX SDK library file based on compiler options and defined symbols.

Brief sample programs

The FBX SDK ships with the following sample programs that run on all platforms supported by FBX SDK.

Brief sample program	Description
Common on page 27	Functions used by other sample programs to initializing objects, destroying objects, loading scenes, and saving scenes.
MyOwnWriterReader on page 28	Functions used by other sample programs to write and read files in a custom file format.
Animation on page 27	Shows how to create animation and how to evaluate a scene containg animation. Illustrates the use of animation stacks, animation layers, animation curves and animation curve nodes.
ExportScene01 on page 28	Creates and exports an animated scene containing a cylinder deformed by a skeleton made of two segments.
ExportScene02 on page 28	Creates and exports an animated scene containing a sphere morphed by two shapes.
ExportScene03 on page 29	Creates and exports an animated scene containing a textured cube, and a pyramid with materials mapped on its faces.

Brief sample program	Description
ExportScene04 on page 29	Creates and exports a scene containing a group of lights, a marker, and a camera. The animation rotates the lights and moves the camera.
ExportScene05 on page 29	In FBX, the position of a node is expressed in coordinates relative to the node's parent. This example shows how to convert between FBX coordinates and a global co-ordinate system.
ImportScene on page 30	Displays the content of any FBX file.
Layers on page 30	Creates a scene containing a cube with layered textures and materials mapped on each of the cube's faces. Illustrates how to use layers.
UserProperties on page 30	Illustrates how to create user properties, attach the properties to a cube, and then animate the properties; how to create a constraint between a pyramid and a cube.
ViewScene on page 30	Shows how to evaluate the animation data and compute deformations. This is necessary for displaying the content of any FBX SDK-supported file in a graphical window.Uses the OpenGL Utility Toolkit (GLUT). A menu lets you select the view, the current camera, the current animation stack, etc.

Folders for each of these programs are located in `<yourFBXSDKpath>\examples\`. Each folder contains Visual Studio project files (Windows only) or makefiles (Mac OS or Linux) to help you build and run the program.

Common

The `Common` folder does not contain a standalone example.

It contains a `common.cxx` file, which shows how to use FBX SDK to perform basic operations such as creating and initializing scenes, destroying scenes, loading scenes, and saving scenes. The other sample programs call the functions defined in this file.

Animation

This sample program creates a scene containing only one property a few animation curves, and the data structures that animates the property by connecting the curves to the property. The program then shows how to evaluate the scene using KFbxEvaluator.

This sample program illustrates:

- Creating an animation stack (the highest level container for animation).
- Creating animation layers.
- Setting blending modes for each animation layer.

- Bypassing blending for specified data types.
- Creating a curve node to connect animation curves to animatable properties in a scene.
- Animating a property of an animation curve.
- Adding and removing channels.
- Creating animation curves by creating keyframes (animation keys at the beginning and end of the curve.
- Animating channels.
- Adding curve nodes to an animation layer.
- Connecting curves to channels.
- Getting a scene's evaluator.
- Specifying the animation stack to be evaluated.
- Stepping the evaluator through an animation curve one key at a time.
- Returning evaluated values (to be used for rendering a scene).

MyOwnWriterReader

This sample code shows how to use FBX SDK to write and read files in a custom file format, i.e., a file format not directly supported by FBX SDK. The sample code reads and writes "CustomWriter" files.

The MyOwnWriterReader folder does not contain a standalone sample program. Instead, it contains sample functions that support CustomWriter files. The sample functions are in turn used by other sample programs (ExportScene05, ImportScene, and ViewScene) to write and read CustomWriter files.

For more information, see Supporting additional file formats on page 97.

ExportScene01

The scene created in this example is a cylinder linked to a skeleton made of two segments. Two animation stacks of animation show the influence of the skeleton segments over the cylinder. ExportScene01 illustrates how to:

- Create a patch.
- Create a skeleton segment.
- Create a link.
- Store the bind pose.
- Store one arbitrary rest pose.
- Create multiple stacks of animation.
- Create meta-data for a scene.
- Export a scene in an *.fbx* file (in ASCII mode).

ExportScene02

The scene created in this example is a sphere morphed by two shapes. A animation stack of animation shows the influence of the shapes over the sphere. ExportScene02 illustrates how to:

- Create a NURBS.

- Map a shape over NURBS.
- Map a texture over NURBS on material channel Diffuse and Ambient.
- Map a material over a NURBS.
- Create animation.
- Export a scene in an *.fbx* file.

ExportScene03

The scene created in this example is a textured cube and a pyramid with materials mapped on its faces. The animation displays six different angles of both models. ExportScene03 illustrates how to:

- Create a mesh.
- Extend the SDK by creating your own version of class `KFbxMesh`.
- Map a texture over a mesh or your own mesh type.
- Map a material over a mesh or your own mesh type.
- Extend the SDK by applying your own user-defined Layer Element on a mesh.
- Use your own mesh type with your own User Data Layer Element.
- Create a vertex cache deformer to deform a simple model.
- Create animation.
- Animate the vertex cache.
- Export a scene in an *.fbx* file.

ExportScene04

The scene created in this example contains a group of lights, a marker, and a camera. Animation rotates the lights and moves the camera. ExportScene04 illustrates how to:

- Create a light and assign a gobo.
- Set global light settings.
- Create a marker.
- Create a camera and link it to a point of interest.
- Create animation.
- Export a scene in an *.fbx* file.

ExportScene05

The scene created in this example contains a skeleton made of three segments. The position of a node in an *.fbx* file is expressed in coordinates relative to its parent. This example shows how to convert to and from a global position. ExportScene05 illustrates how to:

- Use a custom memory allocator.
- Create a skeleton segment.
- Get the global default position of a node.

- Set the global default position of a node.
- Set limits, rotation order, and pre/post pivots.
- Export a scene in an *.fbx* file.

ImportScene

This example illustrates how to detect if a scene is password protected, and how to import and browse the scene to access node and animation information. This example displays the content of an *.fbx* file that is passed as program argument. It shows how to traverse an FBX scene graph and retrieve its content. You can try this example using any of the FBX SDK-supported files output by the export examples.

Layers

In this example, a scene is created containing a cube with layered textures and materials mapped on each of the cube's faces. This example illustrates how to use texture layers. (Note: Texture layers are not the same as animation layers.)

UserProperties

In this example, a scene is created containing a cube, a pyramid, and user properties. This example illustrates how to:

- Create user properties
- Attach user properties to a cube.
- Animate attached user properties.
- Create a constraint that constrains a pyramid to a cube.

NOTE User properties are also called *custom properties*.

ViewScene

This example illustrates how to display the content of an *.fbx* or *.obj* file in a graphical window. This program is based on the OpenGL® Utility Toolkit (GLUT). Start the program on the command line by providing the name of any FBX SDK-supported file. A menu opens letting you select the current camera and the current animation stack. The ViewScene example illustrates how to:

- Import a scene from *.fbx* or *.obj* files.
- Convert the NURBS and patch attribute types of a scene into mesh node attributes.
- Get the list of all the cameras in the scene.
- Find the current camera.
- Get the relevant settings of a camera depending on its projection type and aperture mode.
- Compute the local and global positions of each node.
- Compute the shape deformation of mesh vertices.
- Compute the cluster deformation of mesh vertices.
- Display the point cache simulation of a mesh.
- Get a list of all poses in the scene.
- Display the scene with a specific pose.

ImportExport tutorial program

The ImportExport tutorial program shows how to convert a file from one file format (FBX binary, FBX ASCII, DAE, etc.) to another. Operations include:

- Building and running a sample program using Visual Studio.
- Managing memory with SDK Manager.
- Creating an empty scene.
- Creating a file importer.
- Loading the import file into the scene.
- Creating a file exporter.
- Specifying the file format of the export file.
- Specifying whether media are embedded in the export file.
- Exporting the scene to the export file.
- Cleaning up and shutting down.

NOTE Builds and runs on Windows only.

See Getting started with file import/export on page 41.

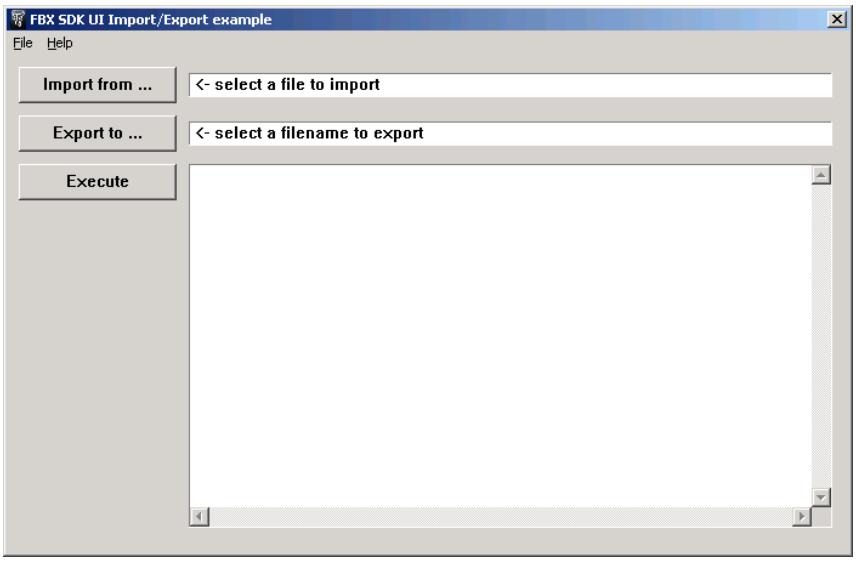
Organization of the project

The ImportExport project is located at `<yourFBXSDKpath>\examples\UI Examples\importexport\`.

ImportExport has two source files:

Source file	Description
ImportExport.cpp	Platform-independent functions to convert files from one FBX-supported file format to another. The function with the main logic is `ImportExport()`.
UI.cpp	User interface to `ImportExport()`. The user interface code depends on the platform for which you downloaded the FBX SDK. We won't look at this source file.

The user interface



User interface to ImportExport tutorial program. Messages appear in the large textbox beside the Execute button.

The main logic

ImportExport's `ImportExport()` function contains the main logic of the file conversion program. `ImportExport()` does the following:

- Accepts the following parameters from the calling program (i.e., `UI.cpp`):
  - The path and name of the file to be imported;
  - The name and path of the file to be exported;
  - The file format for the file to be exported.
- Creates an SDK manager object to manage the memory used by other FBX objects.
- Creates a scene object to hold the data from the import file.
- Creates an importer object to load the file into the scene.
- Loads the data from the import file into the scene.
- Saves the data from the scene to the export file.
- Destroys the SDK manager, which makes sure that all FBX objects are destroyed, i.e., that all memory used by the SDK is released.
- Returns control to the calling program.

SceneTreeView tutorial program

Shows how to traverse all the nodes in an FBX scene, and how to determine the content of each node, i.e., whether a node contains a camera, a light, a mesh, etc. Operations include:

- Getting a reference to the root node of a scene.
- Getting references to each of the child nodes.

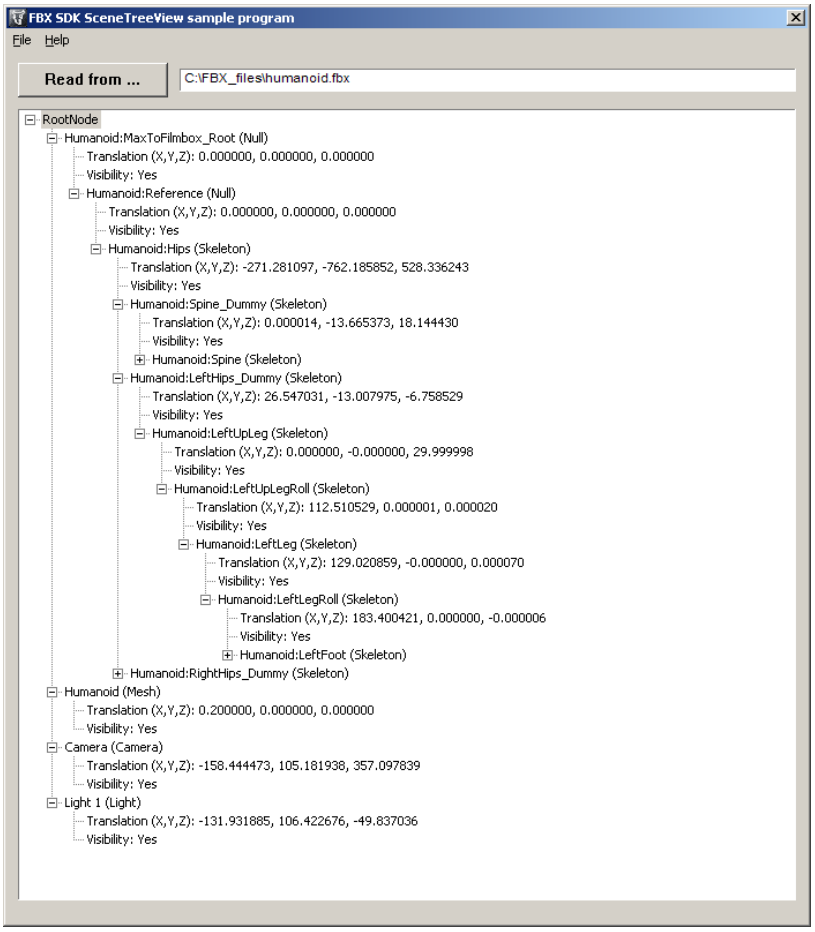
- Getting references in both directions of a two-way relationship.
- Getting a node's properties as a point in space.
- Getting a node's attribute type and contents.

NOTE Builds and runs on Windows only.

See Traversing the Scene Graph on page 49.

SceneTreeView: the user interface

The SceneTreeView sample program imports an FBX file into an empty scene, then displays the scene graph. The scene graph is more precisely a tree, also called a node hierarchy.



The nodes of the tree refer to the scene elements—the cameras, lights, meshes, skeletons, NURBS, and so forth—that were imported from the FBX file. To display this tree, SceneTreeView must traverse the entire tree, that is, it must visit each node.

SceneTreeView: Organization of the project

The SceneTreeView project is located at `<yourFBXSDKpath>\examples\UI Examples\SceneTreeView\`.

SceneTreeView has two source files:

Source file	Description
SDK_Utility.cxx	Contains platform-independent code that calls FBX SDK to create the scene, import the file, retrieve data from each node in the scene graph, and add materials, textures, and animation to the cubes. This tutorial explains much of the code in this file.
UI.cxx	Contains the code that interacts with the user and with Windows, and traverses the scene graph to display the tree view of the scene. The code contains some calls to FBX SDK.

SceneTreeView: the main logic

SceneTreeView begins by:

- Accepting from the user interface the name and path of the file to be imported.
- Loading the import file into the scene.
- Getings a reference to the root node of the scene.

Starting with the root node, SceneTreeView traverses the tree recursively. At each node, SceneTreeView:

- Displays the name and attribute type of the node.
- Displays selected properties of the node, including properties that are stored in the node attribute.
- Gets a reference to each of the children of that node.

CubeCreator tutorial program

This tutorial program shows how to add textures, materials, and animation to meshes representing a model (in this case, a cube). Operations include:

High level operations	Low level operations
Constructing the baseline scene	■ Defining the animation stack and animation layer that will contain the animation. ■ Creating a marker to be used as a reference point. ■ Creating a camera. ■ Creating one texture available to all cubes. ■ Embedding media files in an FBX file. ■ Processing FBX files that contain embedded media. ■ Processing scene files with references to media. ■ Creating an array of materials to be shared.

High level operations	Low level operations
	■ Pointing the camera at the marker. ■ Setting the position of the marker. ■ Setting the position of the camera.
Adding animation to the camera	■ Constructing the initial scene graph. ■ Setting a camera as the scene's default camera. ■ Adding a cube to the scene. ■ Setting the cube's name, position, and rotation. ■ Applying the user's specifications to the cube. ■ Setting the cube's position in the scene. ■ Creating a mesh for the cube. ■ Defining the cube's vertices and the normals. ■ Defining the cube's faces. ■ Using layer 0 to store a normal to each face. ■ Creating a layer element for the materials and textures. ■ Saving memory by sharing a mesh among multiple nodes. ■ Adding animation to a cube. ■ Adding a texture to a cube.
Adding a material to a cube	No lower-level operations.

NOTE Builds and runs on Windows only.

For information about the concepts behind this sample program, and the operations it performs, see:

- Applying Textures and Materials to Meshes on page 57.
- Animating a Scene on page 69

CubeCreator: Organization of the Project

The CubeCreator project is located at `<yourFBXSDKpath>\examples\UI Examples\CubeCreator\`.

CubeCreator has two source files:

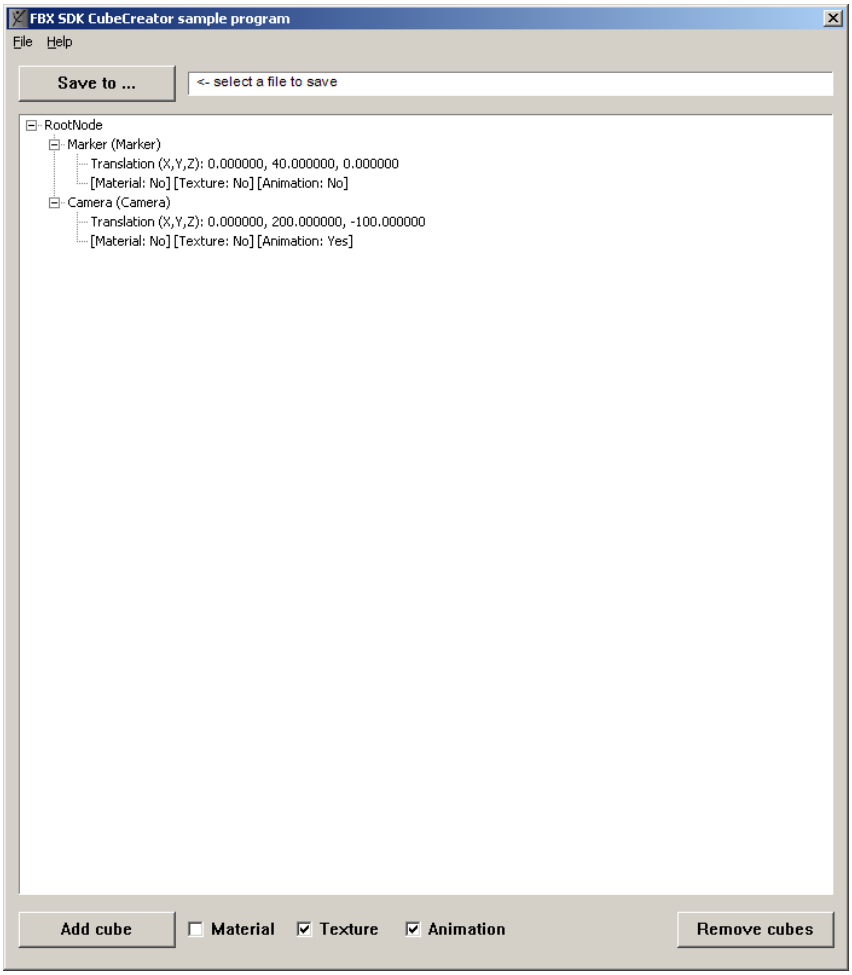
Source file	Description
SDK_Utility.cxx	Contains all the code that calls FBX SDK to create the scene, add the cubes, and add materials, textures and animation to the cubes.This tutorial explains much of the code in this file.

Source file	Description
UI.cxx	Contains the code that displays the tree view of the scene and that interacts with the user. The code uses FBX SDK to retrieve the scene data that the UI displays in its tree view.This tutorial does not explain any of the code in this file.

CubeCreator: The user interface

The CubeCreator sample program shows you how to add textures, materials, and animation to models in a scene.

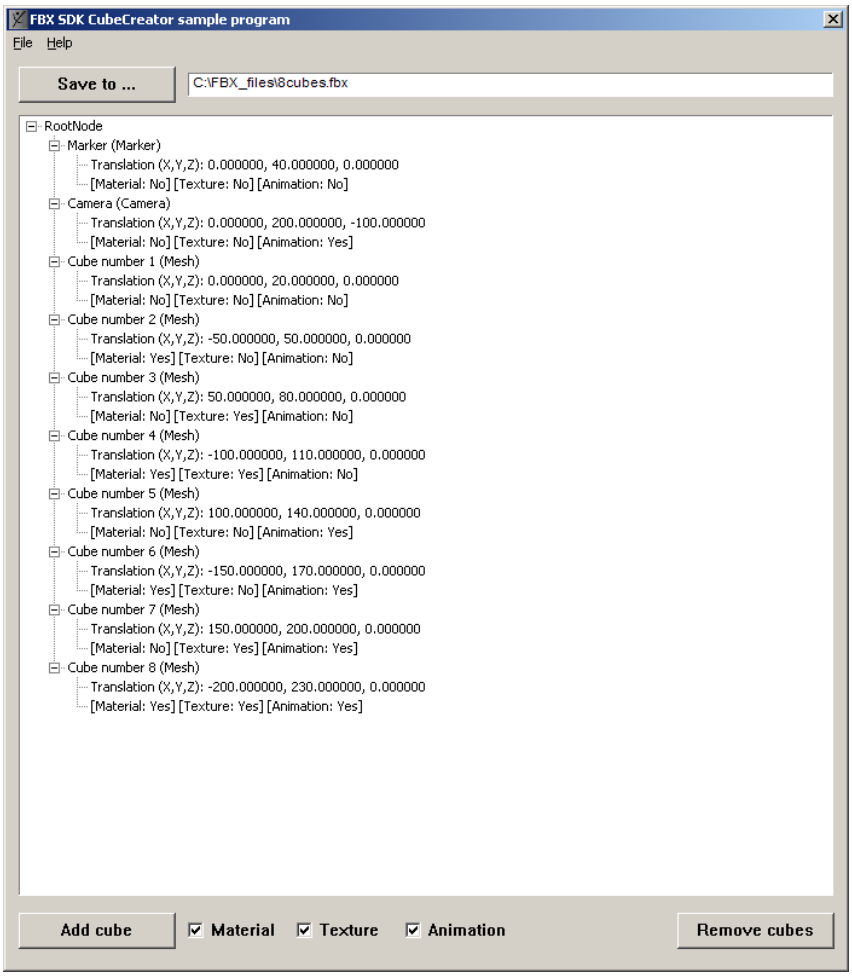
At startup, CubeCreator displays a tree view of a scene containing only a marker and a camera.



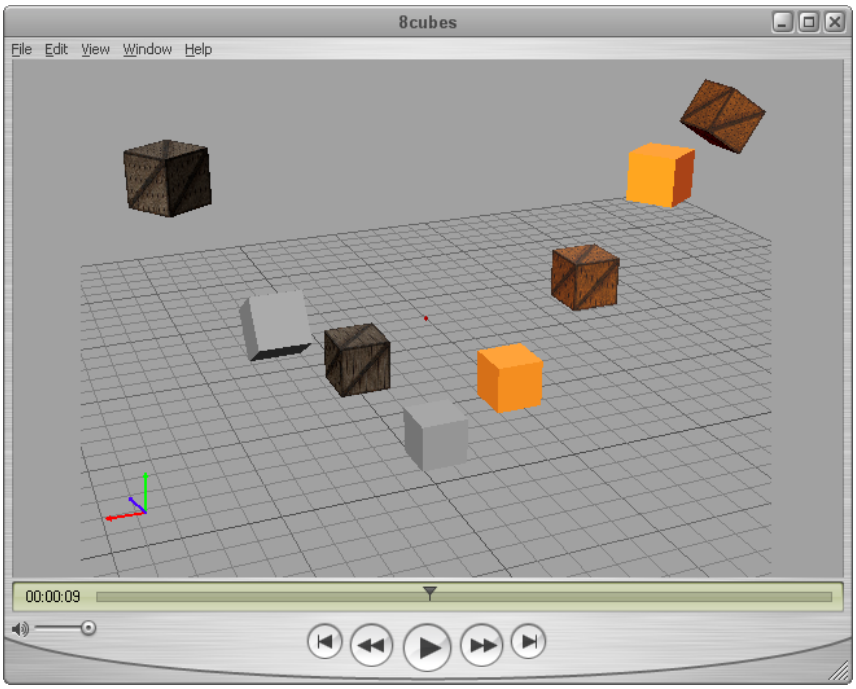
As a user, you can add cubes one at a time to the scene. For each cube, you can:

- Add a texture, or not.
- Add materials, or not.
- Add animation, or not.

You can save the scene as an FBX file (or in any file format exported by FBX SDK).



Here is what `8cubes.fbx` (the scene shown as a tree, above) looks like when opened in FBX for QuickTime:



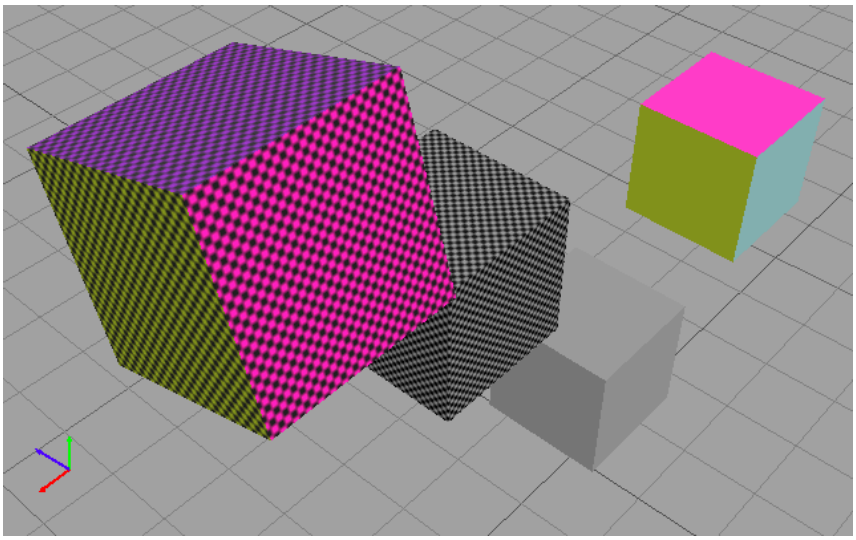
CubeCreator: Location of texture file at run time

<yourFBXSDKpath>\examples\UI Examples\CubeCreator\ contains a file named crate.jpg, which CubeCreator uses as a texture file.

You must copy this texture file into the folder where the executable for CubeCreator will start. Unless you change the settings for the CubeCreator project:

- At build time, CubeCreator.exe will be saved in a subdirectory of *<yourFBXSDKpath>*\bin\CubeCreator\.
- At run time, CubeCreator.exe will execute in that same directory. It will search for the texture file in that location.

If FBX for QuickTime cannot find a texture file for a surface, it renders the surface as a checkerboard pattern. Here is how that looks for the faces of our cubes:



CubeCreator: The main logic

CubeCreator does the following operations:

- Constructs a baseline scene containing a marker, a camera, and a light.
- Points the camera at the marker, then adds animation to the camera. The animation is always the same: a view of the cubes that sweeps back and up.
- Defines one material that you (the user) can apply to any or all cubes.
- Defines one texture that you can apply to any or all cubes.
- Defines animation that you can apply to any or all cubes. If applied to a cube, the animation causes the cube to rotate around its axis of rotation.
- Lets you add cubes, one at a time. The cubes are mesh objects.
- Places the first cube near the marker.
- Places subsequent cubes alternately: left of and above the leftmost cube, then right of and above the rightmost cube.
- Gives each cube an axis of rotation: either X, Y, or Z.
- Lets you (the user) add 0 to *n* cubes. To each cube added, you have the option of adding any or all of a material, a texture, and animation.
- Lets you save the scene as an FBX file (or in another FBX-supported file format).
- Invites you to use FBX for QuickTime to view and interact with the scene stored in the saved file.

Building and running the sample programs

This section describes how to use Visual Studio to build and run ImportExport, one of the tutorial programs. Tutorial programs are available only for Windows.

For Windows, the procedure is essentially the same for any of the sample programs.

For Linux and Mac OS, the procedure is trivial, and therefore omitted.

Visual Studio

To build and run ImportExport using Visual Studio:

1. Start Visual Studio.
2. Click File > Open > Project/Solution.
3. Navigate to the folder that contains the version of ImportExport for your version of Visual Studio, e.g. `C:\Program Files\Autodesk\FBX\FbxSdk\2011.2\examples\UI Examples\ImportExport\`
4. Open the project file for your version of Visual Studio. For Visual Studio 2005, the file is named `ImportExport_net2005.vcproj`. Visual Studio will create a new solution, and open the project inside that solution.
5. To build the file, click Build > Build Solution. Unless you modify the project, Visual Studio builds a debug version of the application (named `ImportExport.exe`), stored in the project's `Debug\` directory.
6. To run ImportExport within Visual Studio, choose Start Debugging from the Debug menu. The dialog box for ImportExport will open. You can use it to select files for conversion to various file formats.

Getting started with file import/export

5

This tutorial shows you how to the most commonly used classes of FBX SDK: those dealing with memory management, importing scene data from a file, and exporting scene data to a file.

For a sample program that demonstrates most of the concepts introduced here, see ImportExport.

Managing memory with the SDK manager

Class `KFbxSdkManager` (sometimes called the *SDK manager*) is the memory manager for FBX SDK. You will use it, directly or indirectly, whenever you instantiate an FBX SDK object, and again whenever you destroy an FBX SDK object. An FBX application normally:

- Begins by creating an SDK manager object.
- Uses that object to create a scene (i.e., an instance of class `KFbxScene`.
- Uses the scene object to create objects that are instances of most of the classes of FBX SDK.

Normally, an FBX application needs only one SDK manager object.

Almost all classes in FBX SDK have a `Create()` member function and a Destroy() member function. Whenever you call these functions, SDK Manager automatically looks after allocating and releasing memory.

Creating an SDK manager

This code snippet declares an SDK manager object and then instantiates it:

```
KFbxSdkManager* mySdkManager = NULL;
...
mySdkManager = KFbxSdkManager::Create();
```

Using SDK manager to create a scene object

When you create a scene object, you must pass the SDK manager object that will manage memory for that scene:

```
// Create a scene object
KFbxScene* myScene = KFbxScene::Create(mySdkManager, "");
```

Most FBX applications need only one scene. But if, for example, you wish to load (i.e., import) several FBX files into memory and work with them all at the same time, you can create a separate scene for each file.

NOTE You can use FBX SDK to import and export FBX files as well as other formats. See File formats imported and exported on page 12

Using a scene object to create other FBX objects

Most classes in FBX SDK have a `Create()` member function which you use in place of a constructor.

To instantiate (create) an object of most FBX classes:

1. Call the class's `Create()` member function.
2. For the first parameter, pass the scene object to which the object belongs.

For example:

```
// Create a node object
KFbxNode* myNode = KFbxNode::Create (myScene, "");

// Create an mesh object
KFbxImporter* myMesh = KFbxMesh::Create (myScene, "");
```

The scene that you pass as a parameter knows to which SDK manager it belongs. That SDK manager is responsible for allocating memory for the node, mesh, or other FBX object that you are instantiating.

NOTE When you export a scene to a file, all objects created with the scene object will be exported.

Specifying names for FBX objects

The second parameter of `Create()` is an optional name for the object. You can specify any name as a string: the name does not need to be unique. You can also specify an empty string. Examples:

```
// Create a scene object named My Scene
KFbxScene* myScene = KFbxScene::Create(mySdkManager, "My Scene");

// Create a node object named My Own Node
KFbxNode* myNode = KFbxNode::Create (myScene, "My Own Node");

// Create a mesh object named My Very Own Mesh
KFbxMesh* myMesh = KFbxMesh::Create (myScene, "My Very Own Mesh");

// Create a camera object with no name
KFbxCamera* myCamera = KFbxCamera::Create (myScene, "");
```

Because the names of FBX objects need not be unique, you cannot simply "`get`" an FBX object by specifying its name. But you can write code to enumerate all the objects in a scene that have a specified name.

Creating FBX objects that belong to no scene

If for any reason you wish to create an object that is not a part of any specified scene, you can pass an SDK manager object rather than a scene object:

```
// Create a camera manipulator object to be used in several scenes
KFbxCameraManipulator* myCameraManipulator =
    KFbxCameraManipulator::Create (mySDKManager, "");
```

NOTE Consider an application with two FBX scenes in memory, `scene1` and `scene2`. If an object has been created with `scene1` as the first parameter, and then `scene2` is exported, the object will not be exported. If the object has been created with the SDK manager as the first parameter, then the object will not be exported when either `scene1` or `scene2` is exported.

Destroying objects

Always destroy an FBX object using its `Destroy()` member function. SDK Manager will automatically free all memory allocated for the object. SDK Manager will also update all *connections* to that object that are stored in other FBX objects (see Connections on page 78).

To destroy any object created with Create():

- Call the object's `Destroy()` member function:

```
// Destroy these objects
myMesh->Destroy();      // Destroy the mesh
myNode->Destroy();      // Destroy the node
myScene->Destroy();     // Destroy the scene and its objects
mySDKManager->Destroy() // Destroy SDK Manager and its objects
```

Notes on destroying an object:

- The SDK Manager can reuse a destroyed object's memory for new objects.
- When you call `Destroy()`, you don't need to specify the object's scene or SDK manager: each FBX object "knows" the SDK manager object that allocated its memory.
- When you destroy a scene, SDK manager automatically destroys any objects created with that scene object. (See Using a scene object to create other FBX objects on page 42).

To clean up memory after using FBX SDK:

- Call the SDK Manager's `Destroy()` member function.

This will also destroy any remaining SDK objects managed by that SDK Manager.

```
// Destroy the FBX SDK manager.
// All the objects that
// (1) have been allocated by the memory manager, AND that
// (2) have not been explicitly destroyed
// will be automatically destroyed.
if (mySdkManager) mySdkManager->Destroy();
```

Specifying your own memory allocation routines

NOTE Skip this topic unless your application is using its own memory management routines.

FBX implements its own *memory allocation routines* (i.e., `malloc`, `calloc`, `realloc`, `free`, and `msize`). By default, whenever you `create()` an FBXobject, SDK Manager allocates and manages memory for that object using the FBX memory allocation routines.

If you want FBX SDK to use another set of routines, call `KFbxSdkManager::SetMemoryAllocator` before creating any `KFbxSdkManager` object. `SetMemoryAllocator()` takes a `KFbxMemoryAllocator` object as a parameter.

Creating an empty scene

A *scene* is a container for 3D *scene elements* such as meshes, NURBS, cameras, lights, etc. One FBX file must contain one and only one scene. Your FBX application, on the other hand, may contain more than one scene.

The scene elements in an FBX scene are stored in the nodes of a *scene graph*. For FBX, the scene graph is actually a *tree* (i.e., a *node hierarchy*). Each node has a *node attribute* that determines whether the node contains a mesh, a camera, a light, etc. For more detail on scenes, nodes, node attributes, etc., see Traversing the Scene Graph on page 49.

Before you can import an FBX file (or any other file format supported by FBX SDK), you must create a scene object to hold the data stored in the file:

```
// Create an FBX scene object
KFbxScene* myScene = KFbxScene::Create(mySdkManager, "");
```

Creating a file importer

To *import a file* into an FBX scene (i.e., to *load* the scene with the contents of a file), you must first create an *importer* object:

```
// Create an importer
KFbxImporter* myImporter = KFbxImporter::Create(mySdkManager, "");
```

FBX can import many different kinds of files: FBX files in a *binary* (or *native*) format, FBX files in an ASCII format, FBX files using earlier versions of the FBX file format, and some non-FBX file formats, such as Collada (see File formats imported and exported on page 12). Any file that is to be imported into a program using FBX is called an *import file*.

```
const char *myImportFile;   // Full path\filename of the file to be imported
```

Use the `Initialize()` member function to specify the path and name of the file to be imported.

```
const bool lImportStatus =
    myImporter->Initialize(myImportFile, -1, pSdkManager->GetIOSettings());
if( !lImportStatus )
    ...// Problem with the file to be imported
```

Checking version numbers

There are several versions of FBX SDK that are in current use: FBX SDK 2011.2, FBX SDK 2011.1, FBX SDK 2010.2, FBX SDK 2010.0, FBX SDK 2009, etc.

FBX file formats also have version numbers. FBX SDK 2010.2 exports FBX file format 7.1 (default), 7.0, and 6.1, and imports FBX file format 7.1, 7.0, 6.1, and 6.0 (seeFile formats imported and exported on page 12.)

You can check whether the version of the file format used by an FBX file is supported by the version of FBX SDK to which your application is linked:

```
// Get the FBX file format version number for the FBX files generated
// by the version of FBX SDK that you are using.
myImporter->GetFileVersion(lFileMajor, lFileMinor, lFileRevision);

// Get the FBX file format version number of the import file
myImporter->GetFileVersion(lFileMajor, lFileMinor, lFileRevision);
```

Error handling

For many member functions of FBX classes, if a call to the function triggers an error:

- The function returns `false`.
- *`objectname`*`->GetLastErrorString()` returns a string with an error message.
- *`objectname`*`->GetLastErrorID()` returns an integer value. You can use FBX enumerated values or defined values to handle the error appropriately.

Here is an example of error-handing code for a file import operation:

```
const bool lImportStatus =
    myImporter->Initialize(myImportFile, -1, pSdkManager->GetIOSettings() );
if( !lImportStatus ) // Problem with the import file
{
    UI_Printf("Call to KFbxImporter::Initialize() failed.");
    UI_Printf("Error returned: %s", myImporter->GetLastErrorString());

    if (myImporter->GetLastErrorID() ==
        KFbxIO::eFILE_VERSION_NOT_SUPPORTED_YET ||
        myImporter->GetLastErrorID() ==
        KFbxIO::eFILE_VERSION_NOT_SUPPORTED_ANYMORE)
        {    // Handle the error...
```

Loading all or part of an import file

You can import into an FBX scene all or only some of the data types that are stored in an import file. For example, you can choose whether or not to import the cameras, lights, animation, etc. These choices are called *import options*.

Import options (as well as export options) are managed by your SDK manager, and stored in an IO settings object:

```
// Create an IOSettings object.
// Let's assume that mySDKManager has already been created.
KFbxIOSettings * ios = KFbxIOSettings::Create(mySdkManager, IOSROOT );
mySdkManager->SetIOSettings(ios); // Store IO settings here
```

The following code snippet sets some import options:

```
// Import options determine what kind of data is to be imported.
// True is the default, but here we'll set some to true explicitly, and others to false.
(*(pSdkManager->GetIOSettings())).SetBoolProp(IMP_FBX_MATERIAL,        true);
(*(pSdkManager->GetIOSettings())).SetBoolProp(IMP_FBX_TEXTURE,         true);
(*(pSdkManager->GetIOSettings())).SetBoolProp(IMP_FBX_LINK,            false);
(*(pSdkManager->GetIOSettings())).SetBoolProp(IMP_FBX_SHAPE,           false);
(*(pSdkManager->GetIOSettings())).SetBoolProp(IMP_FBX_GOBO,            false);
(*(pSdkManager->GetIOSettings())).SetBoolProp(IMP_FBX_ANIMATION,       true);
(*(pSdkManager->GetIOSettings())).SetBoolProp(IMP_FBX_GLOBAL_SETTINGS, true);
```

Here is the call to import a file into our scene::

```
lStatus = myImporter->Import(myScene);
```

The importer reads the import file, extracts the scene data (as specified by the import options) from the file, and loads it into the FBX scene (which may or may not be previously empty). In other words, the FBX scene graph is populated with data extracted from the import file.

If the import file is not an FBX file, the importer does the necessary conversions to load it into an FBX scene.

In the ImportExport sample program, we load a scene from a file, but we do not look at the contents of the scene. For more detail on scenes, nodes, node attributes, etc., see Traversing the Scene Graph on page 49, which explains the SceneTreeView sample program.

Notes on import options:

- For each import option, the default value is `true`. This means that by default, the importer imports all the data types that FBX SDK can handle.
- The default import options are used for all import files unless you explicitly change one or more of the options.
- Once import options are changed, the then-current set of options are used for all import files until (a) further changes are made or (b) the program terminates.
- For a full list of import options, see header file `kfbxiosettingspath.h`. Import options have symbols that begin with "`IMP_`".

Saving all or part of a scene

Saving (*exporting*) a scene to an *export file* is very similar to importing a file:

- You must create an exporter object using class `KFbxExporter`.
- You can control what scene elements are exported and other aspects of the export by specifying *export options*. For a full list of export options, see header file `kfbxiosettingspath.h`. Export options have symbols that begin with "`EXP_`".
- You must specify the file format for the export file.
- You can choose to embed *media* (textures, sound, movies, etc.) in the export file, or save them as separate files.

Embedding media in FBX files

One of the export options specifies how FBX handles any *media* (textures, sound, movies, etc.) in the scene to be exported:

```
bool myEmbedMediaFlag;

... // Set myEmbedMediaFlag to true or false.
EXP_FBX_EMBEDDED is an export option. Set it to myEmbedMediaFlag.

(*(pSdkManager->GetIOSettings())).SetBoolProp(EXP_FBX_EMBEDDED, myEmbedMediaFlag);
```

There are two choices:

- If `true`, the media are physically embedded in the export file. This option is available only when exporting to FBX binary files.
- if `false`, no media files are embedded. Instead, the FBX file contains relative references to the media files.

When imported by any program that uses FBX SDK, embedded media files are extracted into *myExportFile*.fbm\, which is a subdirectory of the directory where the FBX file *myExportFile*.fbx is located. *myExportFile*.fbm\ is created if it does not already exist.

Traversing the Scene Graph 6

This tutorial:

- Introduces you to the FBX scene graph.
- Shows how to traverse the nodes in the scene.
- Shows how to discover the data type of each node in the scene.
- Shows how FBX SDK handles relationships between nodes.
- Introduces coordinate systems as well as translation, rotation, and scaling.

FBX SDK includes two sample programs which demonstrate these concepts:

- SceneTreeView (Windows only).
- ImportScene (all FBX development platforms).

Both of these programs traverse the scene graph and display information about what it finds in each node in the scene.

See also:

- SceneTreeView tutorial program on page 32.
- ImportScene on page 30.

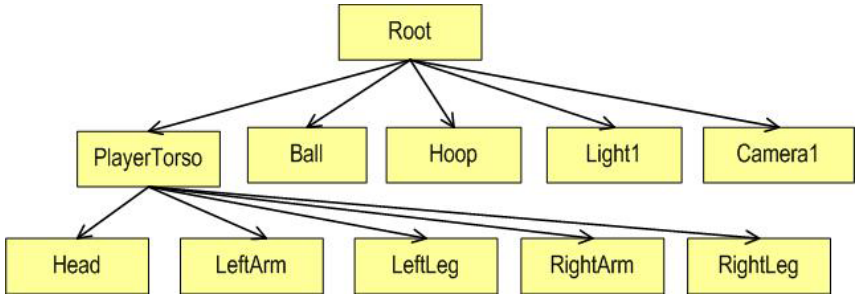
Introducing the FBX scene

In FBX, a *scene* is a container for all the meshes, NURBS, cameras, lights and other scene elements that make up a 3D scene.

Each element in the scene is called a *node*. The data structure used to organize all the nodes in a 3D scene is commonly called the *scene graph.* In FBX, the scene graph is specifically a *tree*:

- The root node of the scene graph has no parent.
- Every other node in the scene graph has exactly one parent.
- Every node can have zero, one, or many children.

Here is a diagram showing an extremely simple scene containing a camera, a light, a ball, a hoop, and a stick figure of a basketball player:



Notes on FBX scenes:

- An FBX application must have a root node. But the root node is never saved in an FBX file. Accordingly, do not use the root node to represents anything "real" in the scene that you may want to save in a file.
- The scene object points to the root node of the scene. Each node, including the root node, points to its children. Accordingly, with the scene object as a starting point, you can traverse the entire tree.
- An FBX application can have more than one scene. But an FBX file can store only one scene.

Creating a scene and getting its root node

FBX applications using create the scene object immediately after creating the SDK Manager object:

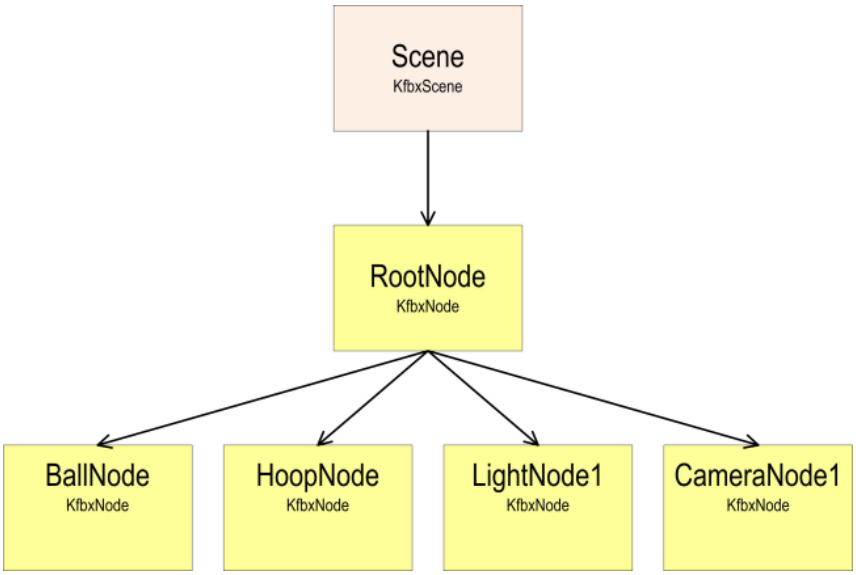
```
// Declarations
KFbxSdkManager* mySdkManager = NULL;
KFbxScene*      myScene      = NULL;
const KFbxNode* myRootNode   = NULL;
...
// Create SDK manager to manage memory
mySdkManager = KFbxSdkManager::Create();

// Create scene object
myScene = KFbxScene::Create(gSdkManager, "");
```

In FBX, a scene has exactly one root node. KFbxScene has a function `GetRootNode()` which returns a reference to the root node of a scene:

```
//Get the scene's root node
KFbxNode* rootNode = myScene->GetRootNode();
```

All nodes in an FBX scene graph are, directly or indirectly, children of the root node:



Scene objects are instances of Class `KFbxScene`. Node objects are instances of Class `KFbxNode`.

The root node referenced by a `KFBXScene` object is not saved in an export file. So if you want, for example, to apply a global translation to all objects in the scene:

1. Create a node (named, for example, RootNodeForGlobals) below RootNode and above the BallNode, HoopNode, etc.
2. Apply the global translation to RootNodeForGlobals.

RootNodeForGlobals will be saved in the export file with the global translation properties.

Getting child nodes recursively

To display the scene graph, sample program ImportScene displays the root node, then visits and displays each of the children. More precisely, it recursively visits and displays all the children, grandchildren, etc. of each of the root node's children:

```
// Non-recursive DisplayContent() is called once
void DisplayContent(KFbxScene* pScene)
{
    int i;
    KFbxNode* lNode = pScene->GetRootNode();
    if(lNode)
    {
        for(i = 0; i < lNode->GetChildCount(); i++)
        {
            // Call recursive DisplayContent() once for each child of the root node
            DisplayContent(lNode->GetChild(i));
        }
    }
}

// Recursive DisplayContent() is called for each node below the root node
void DisplayContent(KFbxNode* pNode)
{
...
    for(i = 0; i < pNode->GetChildCount(); i++)
    {
        DisplayContent(pNode->GetChild(i));
    }
}
```

Most relationships are two-way

You can get from a node a reference to its parent, as well as to the scene that contains it:

```
KFbxNode myParentNode = pNode->GetParent();
KFbxScene myScene = pNode->GetScene();
```

In general, all relationships in FBX are two-way: if FBX object A has a reference to FBX object B, then object B will also have a reference to object A. For example:

- A child node "knows" its parent, and a parent node knows each of its children.
- A node knows its scene, and a scene knows the root node of the tree that contains all nodes in the scene.
- A node knows its constraints, and a constraint knows which nodes it constrains.

These two-way relationships between FBX objects are managed by *connections*, an FBX data structure. See Connections on page 78.

Getting the properties of a node as a point in space

A node represents a point in space. Properties that are generic to any point in space are generally accessed using class `KFbxNode`. The properties of `KFbxNode` include those of the node's *translation*, *rotation*, and *scaling* (the *TRS properties*)

TRS properties are usually expressed as values of X, Y, and Z. These three values are usually stored in a vector of four elements, i.e., as `KFbxVector4` objects.

NOTE The TRS properties of a node are expressed as offsets to the TRS properties of the node's parent, which in turn are expressed as offsets of its parent's TRS properties, and so on.

Since the location of an FBX node is expressed as a translation from the parent node's location, we need to get those translation values. More precisely, we need to get the node's *default translation* values.

NOTE A node's actual location may be affected by limits on the X, Y, and Z values, by constraints, and by animation curves (FCurves). These are discussed in Applying Textures and Materials to Meshes on page 57.

In FBX SDK, a triplet of X, Y, and Z values are often stored in a four-element vector, an instance of class `KFbxVector4`. Here is a function that shows how to get the default translation vector and how to access the X, Y, and Z values:

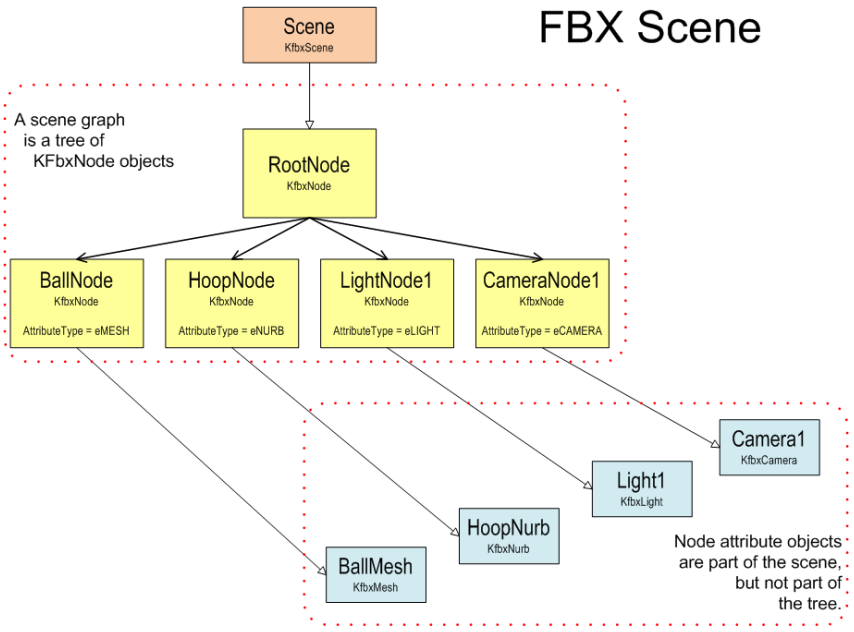
```
// Return a node's default translation values as a text string
KString GetDefaultTranslationInfo(
                                  const KFbxNode* pNode
                                  )
{
    KFbxVector4 v4; // Default translation values
    v4 = ((KFbxNode*)pNode)->LclTranslation.Get();

    return KString("Translation (X,Y,Z): ") + KString(v4.GetAt(0)) + ", " +
        KString(v4.GetAt(1)) + ", " + KString(v4.GetAt(2));
}
```

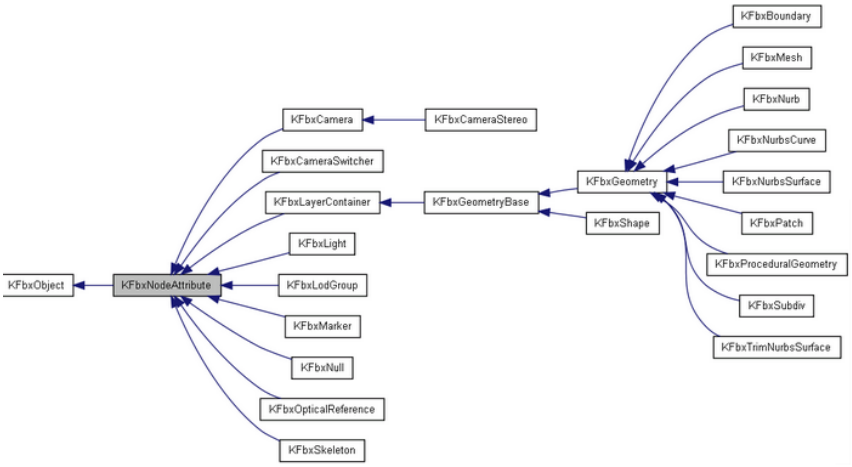
Getting the attribute type and contents of a node

FBX node objects store the generic properties for a point in space.

The data for the "contents" of a given node—e.g., the data particular to a camera, light, mesh, or NURBS—is stored in another object. That object must be an instance of the appropriate class: `KFbxCamera`, `KFbxLight`, `KFbxMesh`, `KFbxNURBS`, etc.



These classes are called *node attributes*, and are subclasses of Class `KFbxNodeAttribute`:



Each node has a node attribute (`KFbxNode::GetNodeAttribute()`), whose value is either:

- A pointer to the node attribute object of the node.
- `null`, i.e., there is no node attribute object for the node.

If the node attribute is not `null`, then you can get from the node attribute object its *attribute type* (enum `eAttributeType`).

NOTE Class `KFbxNull` is used to create node attribute objects representing a null node in the tree. These objects are used by applications that need to represent a null node in its scene graph. A pointer to a KFbxNull object is not the same as `null`.

Here is how SceneTreeView creates the string that contains the name of the node and the attribute type:

```
KString GetNodeNameAndAttributeTypeName(const KFbxNode* pNode)
{
    KString s = pNode->GetName();

    KFbxNodeAttribute::EAttributeType lAttributeType;

    if(pNode->GetNodeAttribute() == NULL)
    {
        s += " (No node attribute type)";
    }
    else
    {
        lAttributeType = (pNode->GetNodeAttribute()->GetAttributeType());

        switch (lAttributeType)
        {
        case KFbxNodeAttribute::eMARKER:               s += " (Marker)";              break;
        case KFbxNodeAttribute::eSKELETON:             s += " (Skeleton)";            break;
        case KFbxNodeAttribute::eMESH:                 s += " (Mesh)";                break;
        case KFbxNodeAttribute::eCAMERA:               s += " (Camera)";              break;
        case KFbxNodeAttribute::eLIGHT:                s += " (Light)";               break;
        case KFbxNodeAttribute::eBOUNDARY:             s += " (Boundary)";            break;
        case KFbxNodeAttribute::eTRIM_NURBS_SURFACE:   s += " (Trim nurbs surface)";  break;
         ...
        }
    }

    return s;
}
```

To access properties of a node attribute object that are specific to it being a camera, a NURBS, a light, etc., you can write similar code to cast the object to its appropriate class. This allows you to call the methods for that class:

```
        ...
        case KFbxNodeAttribute::eNURB:
        DisplayNurb(pNode);
        break;

...
void DisplayNurb(KFbxNode* pNode)
{
    KFbxNurb* lNurb = (KFbxNurb*) pNode->GetNodeAttribute ();
    ...
    int lControlPointsCount = lNurb->GetControlPointsCount();
    KFbxVector4* lControlPoints = lNurb->GetControlPoints();
```

Applying Textures and Materials to Meshes

7

Creating the mesh for a cube

This section shows you how to create a cube as a mesh object. Class `KFbxMesh` is the FBX implementation of a mesh.

A *mesh* is a type of geometric model of a three-dimensional object. In a mesh, the basic shape is made up of points, usually called *vertices* or *control points*. The vertices are connected by *edges* to form *polygons*, which are often called *faces*.

The mesh that we are creating is a cube. Therefore:

- The mesh has six faces.
- Each face is a square polygon.
- Each polygon has four equal edges.
- Each polygon has four vertices.

Creating a mesh object

Here is how to create a node in your scene graph that contains a mesh:

```
//Create an SDK manager for your application
KFbxSdkManager* gSdkManager = KFbxSdkManager::Create();

// Create a scene object
gScene = KFbxScene::Create(gSdkManager, "");

// Create a mesh object
KFbxMesh* lMesh = KFbxMesh::Create(pSdkManager, "");

// Get the scene's root node
KFbxNode* lRootNode = gScene->GetRootNode();

// Create a node for the mesh.
KFbxNode* lNode = KFbxNode::Create(pSdkManager, "");

// Set the node as a child of the scene's root node.
lRootNode->AddChild(lNode);

// Set the mesh as the node attribute of the node
lNode->SetNodeAttribute(lMesh);

// set the shading mode to view texture
lNode->SetShadingMode(KFbxNode::eTEXTURE_SHADING);

// Rescale the cube
lNode->LclScaling.Set(KFbxVector4(0.3, 0.3, 0.3));
```

Creating the cube's vertices, faces, and normals

In this section, we create objects that will be used to define the cube's vertices, faces, and normals.

The following code creates eight points in 3D space that define the eight corners of a cube. These corners are the control points (vertices) of the mesh:

```
KFbxVector4 lControlPoint0(-50, 0, 50);
KFbxVector4 lControlPoint1(50, 0, 50);
KFbxVector4 lControlPoint2(50, 100, 50);
KFbxVector4 lControlPoint3(-50, 100, 50);
KFbxVector4 lControlPoint4(-50, 0, -50);
KFbxVector4 lControlPoint5(50, 0, -50);
KFbxVector4 lControlPoint6(50, 100, -50);
KFbxVector4 lControlPoint7(-50, 100, -50);
```

Here we define six normals, one for each face of the cube:

```
KFbxVector4 lNormalXPos(1, 0, 0);
KFbxVector4 lNormalXNeg(-1, 0, 0);
KFbxVector4 lNormalYPos(0, 1, 0);
KFbxVector4 lNormalYNeg(0, -1, 0);
KFbxVector4 lNormalZPos(0, 0, 1);
KFbxVector4 lNormalZNeg(0, 0, -1);
```

Here we tell the mesh object that it needs 24 control points, four control points for each of the cube's six faces:

```
// Create control points.
lMesh->InitControlPoints(24);
```

Here we get a pointer to the mesh's array of 24 control points:

```
KFbxVector4* lControlPoints = lMesh->GetControlPoints();
```

Here we set the values of the mesh's control point array, one face at a time:

```
lControlPoints[0] = lControlPoint0;
lControlPoints[1] = lControlPoint1;
lControlPoints[2] = lControlPoint2;
lControlPoints[3] = lControlPoint3;

lControlPoints[4] = lControlPoint1;
lControlPoints[5] = lControlPoint5;
lControlPoints[6] = lControlPoint6;
lControlPoints[7] = lControlPoint2;

lControlPoints[8] = lControlPoint5;
lControlPoints[9] = lControlPoint4;
lControlPoints[10] = lControlPoint7;
lControlPoints[11] = lControlPoint6;

lControlPoints[12] = lControlPoint4;
lControlPoints[13] = lControlPoint0;
lControlPoints[14] = lControlPoint3;
lControlPoints[15] = lControlPoint7;

lControlPoints[16] = lControlPoint3;
lControlPoints[17] = lControlPoint2;
lControlPoints[18] = lControlPoint6;
lControlPoints[19] = lControlPoint7;

lControlPoints[20] = lControlPoint1;
lControlPoints[21] = lControlPoint0;
lControlPoints[22] = lControlPoint4;
lControlPoints[23] = lControlPoint5;
```

Instancing: sharing a mesh (or other node attribute)

In the CubeCreator tutorial program, every time the user adds a cube, CubeCreator creates a new `KFbxMesh` object and a new `KFbxNode` object containing that mesh. That's two objects for each cube, not to mention all the layer objects, layer element objects, and so forth. If the user needs hundreds or thousands of cubes, the memory requirements could be significant.

Imagine that you need a program where all cubes looked alike, but you needed many thousands of them. You could save memory by creating one `KFbxMesh` object when the program starts up. Then, every time you needed a new cube, you create a new `KFbxNode` object, then point that node to the one mesh.

This is called *instancing*. In general, you can save memory by having many node objects share one node attribute object (i.e., one object of any subclass of `KFbxNodeAttribute`). Each node is an *instance* of the mesh, NURBS, or other scene element.

If you export your scene to an FBX file, instancing also reduces the file size.

You can also save memory by having multiple nodes share textures, materials, animation curves, etc.

Using layers to control how a model is rendered

In FBX SDK, a mesh can have multiple layers. A *layer* (class `KFbxLayer`) is a container for a *layer element* (class `KFbxLayerElement`) that will affect how the mesh will be rendered.

NOTE Unless you need more than one layer (for some advanced purpose), always use layer 0 and only layer 0. MotionBuilder, FBX for QuickTime, and the FBX plug-ins for 3ds Max and Maya only process normals and materials stored in layer 0.

Layers contain materials, textures, UV, and other layer elements

There are many types of layer element: textures (texture files), normals, UV coordinates, specular materials, diffuse materials, and so forth.

For each of these *layer element types*, you may have more than one element. For example, if the element type is textures, you may have more than one texture file; or if the element type is normals, you may have more than one normal.

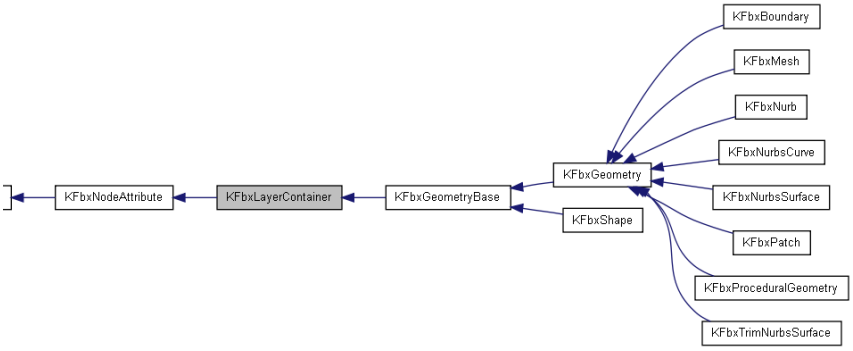
To support multiple elements of each type, a layer element contains an array of elements. What is stored in each element of the array depends upon the layer element type. This array is called *DirectArray*. DirectArray is actually a lookup table, indexed by element number

If you need an additional level of indirection to the elements of a DirectArray, you can use *IndexArray*, an array of index numbers to DirectArray

For example, in the CubeCreator tutorial program, here is how we use the DirectArray and IndexArray for normals:

- We need normals to specify which way each polygon (face) of the cube is facing.
- We need to store six normals (representing Up, Down, Left, Right, Front, and Back): so the DirectArray for this element type must have six array elements.
- We must apply the same normal to all four vertices of each cube face. So we use a 24-element IndexArray to map the six normals to the 24 vertices.

NOTE Meshes are not the only type of FBX object that supports layers. Any subclass of class `KFbxLayerContainer` supports layers.



Any subclass of class `KFbxLayerContainer` supports layers.

Using layer 0 to store a normal to each face

In this section of code, we will assign the normals that we created earlier in a layer element for normals, and we will assign that layer element to Layer 0 of the mesh.

We start by creating layer 0 if it does not already exist:

```
// Set the normals on Layer 0.
KFbxLayer* lLayer = lMesh->GetLayer(0);
if (lLayer == NULL)
{
    lMesh->CreateLayer();
    lLayer = lMesh->GetLayer(0);
}
```

Here we assign *normals* to each vertex of each polygon in our mesh. A normal is a vector that defines which way a face or vertex is pointing. The direction of the normal indicates the front, or outer surface of the face or vertex. Normals are (usually) perpendicular to a polygon.

Since our mesh is a cube:

- There are six faces.
- The faces of the cube are squares, i.e., four-sided polygons.
- Therefore, we assign normals to 24 vertices (control points).

A layer element is composed of two arrays, called DirectArray and IndexArray:

- DirectArray stores the actual pointers to the data. In this case, the data will be the normals: 24 vectors, one for each control point of each face.
- IndexArray stores index numbers to DirectArray. When used, IndexArray provides an additional level of indirection to the vectors.

We start by creating the layer element for the normals.

```
// We want to have one normal for each control point,
// so we set the mapping mode to eBY_CONTROL_POINT.
KFbxLayerElementNormal* lLayerElementNormal= KFbxLayerElementNormal::Create(lMesh, "");

lLayerElementNormal->SetMappingMode(KFbxLayerElement::eBY_CONTROL_POINT);

// Set the normal values for every control point.
lLayerElementNormal->SetReferenceMode(KFbxLayerElement::eDIRECT);
```

The normals for each face of the cube will face the same direction. Accordingly, the first four vertices, which refer to the first face, are assigned the same normal (`lNormalZpos`); the second four vectors, which refer to the second face, are assigned the same normal (`lNormalXpos`); etc.

`GetDirectArray()` returns a pointer to a DirectArray for normals. `Add()` appends a vector to the end of the array.

```
lLayerElementNormal->GetDirectArray().Add(lNormalZPos);
lLayerElementNormal->GetDirectArray().Add(lNormalZPos);
lLayerElementNormal->GetDirectArray().Add(lNormalZPos);
lLayerElementNormal->GetDirectArray().Add(lNormalZPos);

lLayerElementNormal->GetDirectArray().Add(lNormalXPos);
lLayerElementNormal->GetDirectArray().Add(lNormalXPos);
lLayerElementNormal->GetDirectArray().Add(lNormalXPos);
lLayerElementNormal->GetDirectArray().Add(lNormalXPos);

lLayerElementNormal->GetDirectArray().Add(lNormalZNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalZNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalZNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalZNeg);

lLayerElementNormal->GetDirectArray().Add(lNormalXNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalXNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalXNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalXNeg);

lLayerElementNormal->GetDirectArray().Add(lNormalYPos);
lLayerElementNormal->GetDirectArray().Add(lNormalYPos);
lLayerElementNormal->GetDirectArray().Add(lNormalYPos);
lLayerElementNormal->GetDirectArray().Add(lNormalYPos);

lLayerElementNormal->GetDirectArray().Add(lNormalYNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalYNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalYNeg);
lLayerElementNormal->GetDirectArray().Add(lNormalYNeg);
```

We've finished assigning normals to the DirectArray of normals. Now we assign the DirectArray to the layer (layer 0):

```
lLayer->SetNormals(lLayerElementNormal);
```

This completes the construction of the mesh for a cube.

See also:

- Using multiple geometry layers on page 92. This advanced topic uses diagrams to show the relationships between layers, layer elements, etc.
- `KFbxLayerElement::ELayerElementType` is an enum that maps identifiers such as `eUV`, `eNORMAL`, and `eMATERIAL` to their corresponding classes (`KFbxLayerElementUV`, `KFbxLayerElementNormal`, `KFbxLayerElementMaterial`, ...). You must supply the enum as a parameter to `KFbxLayer::GetLayerElementOfType()`.
- `KFbxLayerElement::EMappingMode` is an enum that specifies how the layer element is mapped to the surface.
- `KFbxLayerElement::EReferenceMode` is an enum that specifies how the IndexArray and DirectArray of a layer element are to be referenced.

Creating layer elements for materials and textures

For each cube, CubeCreator allows the user to apply one material and one texture to each of the faces. This section of code does all the necessary low-level setup so that `CreateCubeDetailed()` can apply the material and texture in a few lines of code.

Here we create an array of polygon vertex numbers that we'll use below.

```
// Array of polygon vertices.
int lPolygonVertices[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12, 13,
    14, 15, 16, 17, 18, 19, 20, 21, 22, 23 };
```

CubeCreator uses only one texture, but we still need to use a DirectArray to store the pointer to that texture. The DirectArray will need only one element, i.e., element 0, for that texture.

FBX SDK supports various texture channels, including diffuse, specular, ambient, and shininess. We're going to use the diffuse channel: that allows us to set the color of each face of the cube.

Here, we create the DirectArray for textures:

```
// Set texture mapping for diffuse channel.
KFbxLayerElementTexture* lTextureDiffuseLayer=
    KFbxLayerElementTexture::Create(pMesh, "Diffuse Texture");
lTextureDiffuseLayer->SetMappingMode(KFbxLayerElement::eBY_POLYGON);
```

This time, we are going to use the IndexArray as well as the DirectArray.

```
lTextureDiffuseLayer->SetReferenceMode(KFbxLayerElement::eINDEX_TO_DIRECT);
```

Here we point the mesh's layer to the object that contains the DirectArray and IndexArray for textures:

```
if(pMesh->GetLayer(0))
    pMesh->Getlayer(0)->SetDiffuseTextures(lTextureDiffuseLayer);
else
    return;
```

To correctly map the texture onto the mesh, we need the UV coordinates on the same channel (i.e., diffuse texture). FBX SDK will pair the texture data with the UV coordinates of the same given channel.

```
// Create UV for Diffuse channel.
KFbxLayerElementUV* lUVDiffuseLayer=
    KFbxLayerElementUV::Create(lMesh, "DiffuseUV");
lUVDiffuseLayer->SetMappingMode(KFbxLayerElement::eBY_POLYGON_VERTEX);
```

We are going to use IndexArray to store index numbers for DirectArray:

```
lUVDiffuseLayer->SetReferenceMode(KFbxLayerElement::eINDEX_TO_DIRECT);
lLayer->SetUVs(lUVDiffuseLayer, KFbxLayerElement::eDIFFUSE_TEXTURES);
```

Since we may apply a material to the faces, we also need a layer element for materials:

```
// Set material mapping.
KFbxLayerElementMaterial* lMaterialLayer=KFbxLayerElementMaterial::Create(pMesh, "");
lMaterialLayer->SetMappingMode(KFbxLayerElement::eBY_POLYGON);
lMaterialLayer->SetReferenceMode(KFbxLayerElement::eINDEX_TO_DIRECT);
if(pMesh->GetLayer(0))
    pMesh->Getlayer(0)->SetMaterials(lMaterialLayer);
else
    return;
```

These four vectors define UV coordinates for mapping the the texture onto the face of a cube. UV coordinates are normalized to range from 0.0 to 1.0: the vectors below will map the texture so that it covers the entire surface of the face.

```
KFbxVector2 lVectors0(0, 0);
KFbxVector2 lVectors1(1, 0);
KFbxVector2 lVectors2(1, 1);
KFbxVector2 lVectors3(0, 1);
```

Our DirectArray for diffuse UV coordinates has four elements, one for each coordinate:

```
lUVDiffuseLayer->GetDirectArray().Add(lVectors0);
lUVDiffuseLayer->GetDirectArray().Add(lVectors1);
lUVDiffuseLayer->GetDirectArray().Add(lVectors2);
lUVDiffuseLayer->GetDirectArray().Add(lVectors3);
```

This IndexArray has 24 elements, one for each of the six vertices of the four faces of the cube. We use it to index the DirectArray of four UV coordinates.

```
// Now we have set the UVs as eINDEX_TO_DIRECT reference
// and in eBY_POLYGON_VERTEX mapping mode.
// We must update the size of the index array.
 lUVDiffuseLayer->GetIndexArray().SetCount(24);
```

This IndexArray has has six elements, one for each face of a cube. We use it to index the DirectArray of textures defined in `lTextureDiffuseLayer`.

```
//We are in eBY_POLYGON, so there's only need for 6 index (a cube has 6 polygons).
lTextureDiffuseLayer->GetIndexArray().SetCount(6);
```

For each of the six faces of the cube:

```
// Create polygons. Assign texture and texture UV indices.
for(i = 0; i < 6; i++)
{
```

- For this face, begin to draw the polygon:

```
// all faces of the cube have the same texture
lMesh->BeginPolygon(-1, -1, -1, false);)
```

- For each of the four vertices of face `[i]`:

```
for(j = 0; j < 4; j++)
{
```

- Add polygon vertex `[j]` of face `[i]` to the mesh of the cube:

```
// Control point index
lMesh->AddPolygon(lPolygonVertices[i*4 + j]);
```

- Map the IndexArray elements to the the approriate DirectArray element.
  Earlier, we created the DirectArray with four elements, i.e., 4 UV coordinates. The effect of this code is to map the four UV coordinates to the four corners of each face of the cube. The values of the UV coordinates mean the texture will cover the entire face.

```
// update the index array of the UVs that map the texture to the face
lUVDiffuseLayer->GetIndexArray().SetAt(i*4+j, j);
}
```

- Finish drawing the polygon for face `[i]`.

```
lMesh->EndPolygon ();
}
```

Now we have finished creating the KFbxMesh object.

Managing textures and other media files

Creating a texture object for a texture file

The following code from the CubeCreator tutorial program shows how to create a texture object and to set some of its properties so that it refers to a texture file, `crate.jpg`:

```
KFbxTexture* gTexture = NULL;
...
// Create a global texture for cube.
void CreateTexture(KFbxSdkManager* pSdkManager)
{
    gTexture = KFbxTexture::Create(pSdkManager,"Diffuse Texture");
    // Resource file must be in the application's directory.
    KString lTexPath = gAppPath + "\\Crate.jpg";

    // Set texture properties.
    gTexture->SetFileName(lTexPath.Buffer());
    gTexture->SetTextureUse(KFbxTexture::eSTANDARD);
    gTexture->SetMappingType(KFbxTexture::eUV);
    gTexture->SetMaterialUse(KFbxTexture::eMODEL_MATERIAL);
    gTexture->SetSwapUV(false);
    gTexture->SetTranslation(0.0, 0.0);
    gTexture->SetScale(1.0, 1.0);
    gTexture->SetRotation(0.0, 0.0);
}
```

NOTE The texture file `crate.jpg` must be available to CubeCreator at run time, and available to the renderer (e.g., FBX for QuickTime) of the file exported by CubeCreator. See:

- CubeCreator: Location of texture file at run time on page 38
- Embedding media files in an FBX file on page 66

Applying a texture to a cube

`CreateScene()` calls `CreateCube()`, which calls `CreateCubeDetailed()`, which calls `AddTexture()` only if the user has selected Texture in the UI:

```
void AddTexture(KFbxMesh* pMesh)
{
    // The Layer 0 and the KFbxLayerElementTexture has already been created
    // in the CreateCube function.
    pMesh->GetLayer(0)->GetDiffuseTextures()->
    GetDirectArray().Add(gTexture);
}
```

Embedding media files in an FBX file

When you run CubeCreator, you can save the scene in the file format of your choice and in the folder of your choice.

If you save the scene as a binary FBX file, CubeCreator embeds the texture file `crate.jpg` in the FBX file. This ensures that if you send your FBX file to a co-worker, the FBX file will contain all files referenced by the scene.

In general, you can embed any kind of *media file* in a binary FBX file—providing you set the `EXP_FBX_EMBEDDED` *export option* to `true`:

```
// Create export options object
KFbxStreamOptionsFbxWriter* lExportOptions=
    KFbxStreamOptionsFbxWriter::Create(pSdkManager, "");

// If the file format is FBX binary, then ...
if (pSdkManager->GetIOPluginRegistry()->WriterIsFBX(pFileFormat))
{
    // Embed the media file in the FBX binary file
    IOSREF.SetBoolProp(EXP_FBX_EMBEDDED, pEmbedMedia);
}
```

Processing FBX files that contain embedded media

Applications that render or otherwise process FBX files must access the textures and other media files.

If media files are embedded in a binary FBX file, when you import the file into the application, the FBX SDK importer will extract the media from the FBX file and copy them to a directory. By default:

- The directory for the media is a subdirectory of the directory containing the FBX file.
- The name of that subdirectory is based on the name of the FBX file, but with a *.fbm* extension replacing the *.fbx*. If, for example, the FBX file is `MyScene.fbx`, the subdirectory is named `MyScene.fbm\`.

Processing scene files with references to media

ASCII FBX files, OBJ files, and other FBX-supported file formats cannot contain embedded media. Moreover, you can create binary FBX files that need media files, but set the export option so that the files are not embedded. In these cases, the file will retain the addresses of the media files.

Then you must make sure that FBX for QuickTime (or 3ds Max, Maya, or any other application that you use to render or otherwise process the scene) can find those media.

Here is how FBX SDK searches for each non-embedded media file in, for example, an ASCII FBX file:

- First, FBX SDK uses the absolute *`path\filename`* specified in the FBX file. This was set when the program that exported the file called `SetFileName()`:

```
// Resource file must be in the application's directory.
KString lTexPath = gAppPath + "\\Crate.jpg";
...
gTexture->SetFileName(lTexPath.Buffer());
```

- Then, if it does not find the media file in the absolute path, FBX SDK uses the relative *`path\filename`*. When the FBX file was exported, FBX SDK had determined the relative *`path\filename`* (i.e., relative to the FBX file) of the media file, and stored it in the FBX file.

Managing materials

Creating a material

CubeCreator creates one material that will be applied to each face of the cube. This material is shared by all the cubes in the scene.

We'll use a Phong shader for the material.

```
KFbxSurfacePhong* gMaterial = NULL;
...

// Create global material for cube.
void CreateMaterial(KFbxSdkManager* pSdkManager)
{
   KString lMaterialName = "material";
   KString lShadingName = "Phong";
   fbxDouble3 lBlack(0.0, 0.0, 0.0);
   fbxDouble3 lRed(1.0, 0.0, 0.0);
   fbxDouble3 lDiffuseColor(0.75, 0.75, 0.0);
   gMaterial = KFbxSurfacePhong::Create(pSdkManager, lMaterialName.Buffer());
```

The following properties are used by the Phong shader to calculate the color for each pixel in the material:

```
   // Generate primary and secondary colors.
   gMaterial->GetEmissiveColor()      .Set(lBlack);
   gMaterial->GetAmbientColor()       .Set(lRed);
   gMaterial->GetDiffuseColor()       .Set(lDiffuseColor);
   gMaterial->GetTransparencyFactor() .Set(40.5);
   gMaterial->GetShadingModel()       .Set(lShadingName);
   gMaterial->GetShininess()          .Set(0.5);
}
```

Applying a material to the faces of a cube mesh

The following code applies a global material to the six faces of a cube mesh:

```
void AddMaterials(KFbxMesh* pMesh)
{
    int i;

    //get the node of mesh, add material for it.
    KFbxNode* lNode = pMesh->GetNode();
    if(lNode == NULL)
        return;

    // cube has 6 faces with the same material
    for (i = 0; i < 6; i++ )
    {
        lNode->AddMaterial(gMaterial);
    }
}
```

Animating a Scene

8

This section is a tutorial on animation, based on the CubeCreator tutorial program. This section is a continuation of Applying Textures and Materials to Meshes on page 57, but it can be read independently.

Working with a camera

This section shows how to create a camera object, how to set its location in the FBX scene, and how to point it at a target. In this case, the target is a marker; we will also create the marker and sets its position in the scene.

NOTE This section is based on the CubeCreator tutorial program, located in *<yourFBXSDKpath>*\examples\UI Examples\CubeCreator\.

Creating a camera

CubeCreator creates a node whose node attribute is a camera.

Let's assume we already have a scene:

```
KFbxSDKManager* mySDKManager  // An initialized and valid SDK manager object
KFbxScene* myScene;           // An initialized and valid scene object
```

First we instantiate a camera object that is contained within our scene:

```
KFbxCamera* myCamera = KFbxCamera::Create(myScene, "");
```

Then we adjust the settings of the camera so that it "films" in a format suitable for (non-high-definition) television :

```
// Set camera properties for classic TV projection with aspect ratio 4:3
myCamera->SetFormat(KFbxCamera::eNTSC);
```

Next we instantiate a node object:

```
KFbxNode* myCameraNode = KFbxNode::Create(myScene, "");
```

We set the camera object as the node attribute of the node object:

```
myCameraNode->SetNodeAttribute(myCamera);
```

Finally, we add the node as a child of the root note of the FBX scene graph:

```
KFbxNode* myRootNode = myScene->GetRootNode()
myRootNode->AddChild(myCameraNode);
```

Creating and positioning the camera's target

CubeCreator points its camera, not at a specific cube, but at a *marker* in the scene. Markers are normally not visible, i.e., they are not rendered. In this case, the purpose of the marker is to be a *target* (sometimes called a *point of interest*) of the camera.

Creating a marker is very similar to creating a camera:

```
KFbxMarker* myMarker = KFbxMarker::Create(myScene, "");
KFbxNode* myMarkerNode = KFbxNode::Create(myScene, "");
myMarkerNode->SetNodeAttribute(myMarker);
myRootNode->AddChild(myMarkerNode);
```

Here we set the marker's translation, rotation, and scaling (the TRS properties):

```
// The marker is positioned above the origin.
// There is no rotation
// and no scaling.
myMarkerNode->LclTranslation.Set (KFbxVector4(0.0, 40.0, 0.0));
myMarkerNode->LclRotation.Set (KFbxVector4(0.0, 0.0, 0.0));
myMarkerNode->LclScaling.Set (KFbxVector4(1.0, 1.0, 1.0));
```

`LclTranslation.Set()` sets the default position of a node, in this case, the node for the marker. Note that:

- This position is relative to the parent node (except for the scene's root node, which has no parent node).
- This position is only the node's *default position*: it may be overridden by any *animation* that is applied to the node, by any *limits* to the position of the node, and by any *constraints* that apply to the position of the node.

Similarly, the rotation and scaling of a node are also defaults and are also relative to the parent node.

Pointing a camera at a target

We call `SetCameraPointOfInterest()` to set the marker as the *target* (i.e., *point of interest*) of the camera. This means that even if the camera and/or the marker moves, the camera will continue to point at the marker.

```
// Set the camera to always point at the marker
myCameraNode->SetTarget(myMarkerNode);
```

Set the camera as the scene's default camera

Since a scene can have many cameras, FBX SDK allows you to switch from camera to camera. The default camera is the camera that will be used when the scene's animation begins.

Even though CubeCreator's scene has only one camera, we must explicitly set it as the default camera:

```
myScene->GetGlobalSettings().SetDefaultCamera((char *) lCamera->GetName());
```

Animating a camera (or any other FBX object)

NOTE This section is loosely based on the CubeCreator tutorial program, located in `<yourFBXSDKpath>\examples\UI Examples\CubeCreator`.

Unless you are using blended animation, setting up the data structures required for animation is very straightforward. You'll need one animation stack; one animation layer; and one animation curve node for every animated property.

Let's say that you need to animate a camera. More specifically, you want to animate the local translation property of a camera's `KFbxNode` object:

```
KFbxScene* myScene;        // An initialized and valid scene object
KFbxNode* myCameraNode;    // An initialized and valid node object

// Set the default local translation values (X, Y, Z) for the camera
myCameraNode.LclTranslation.Set(fbxDouble3(1.1, 2.2, 3.3))
```

Although this example deals with the translation of a camera, you can animate any FBX property of any FBX object, providing that the property is *animatable* (see `eFbxPropertyFlags::eANIMATABLE`).

You always need at least one stack and one layer

Begin by creating one animation stack:

```
// Define an animation stack
KFbxAnimStack* myAnimStack = KFbxAnimStack::Create(pScene, "My stack");
```

You need to define only one animation layer:

```
// Create the base layer (this is mandatory)
KFbxAnimLayer* myAnimBaseLayer = KFbxAnimLayer::Create(pScene, "Layer0");
```

Add this layer to the animation stack:

```
myAnimStack->AddMember(myAnimBaseLayer);
```

Curve nodes connect animation data to the animated property

CubeCreator creates one animation curve node for each axis along which the camera moves. You need one animation curve node to connect the translation data to the camera node's `lclTranslation` property:

```
// Get the camera's curve node for local translation.
// true: If the curve node does not exist, create it.
pCamera->LclTranslation.GetCurveNode(pAnimLayer, true);
```

Notes on animation curve nodes:

- `myAnimCurveNode` was automatically created with three animation channels of type `double`, because the type of `lclTranslation` is `fbxDouble3`. See `KFbxNode::lclTranslation`.
- `myAnimCurveNode` was automatically connected to `myCameraNode.LclTranslation`, the local translation property of the camera that we are animating.
- `myAnimCurveNode` was automatically added as a member curve node to `myAnimBaseLayer`. Animation layers are containers for animation curve nodes.
- For another approach to creating animation curve nodes and animation curves, see the Animation sample program in `<yourFBXSDKpath>\examples\Animation\`

`myAnimCurveNode` is all set up to connect animation curves to the local translation property of `myCameraNode`. All that is missing are the animation curves.

Curve nodes are containers for animation curves

Here, we create the animation curve for translation on the X-axis:

```
KFbxAnimCurve* myTranXCurve = NULL;   // Curve for local translation on X-axis
KTime myTime;                         // For the start and stop keys.
int myKeyIndex = 0;                // Index for the keys that define the curve

// Get the animation curve for local translation of the camera.
// true: If the curve does not exist yet, create it.
myTranXCurve = myCameraNode->LclTranslation.GetCurve<KFbxAnimCurve>
                                  (myAnimLayer, KFCURVENODE_T_X, true);
```

Here, we define the curve's starting and ending keyframes:

```
// First the start keyframe
myAnimCurve->KeyModifyBegin();
myTime.SetSecondDouble(0.0);             // Starting time
myKeyIndex = myAnimCurve->KeyAdd(lTime); // Add the start key; returns 0

myAnimCurve->KeySet(lKeyIndex,           // Set the zero'th key
                    lTime,               // Starting time
                    0.0,                 // Starting X value
KFbxAnimCurveDef::eINTERPOLATION_LINEAR);// Straight line between 2 points

// Then the stop keyframe
lTime.SetSecondDouble(20.0);
lKeyIndex = lCurve->KeyAdd(lTime);
lCurve->KeySet(lKeyIndex, lTime, 500.0, KFbxAnimCurveDef::eINTERPOLATION_LINEAR);
lCurve->KeyModifyEnd();
```

So if you were to run this animation, the camera would take 20 seconds to move 500 units along the X-axis.

Sample programs that demonstrate animation:

- The Animation sample program in *<yourFBXSDKpath>*`\examples\Animation\`.
- The CubeCreator tutorial program in *<yourFBXSDKpath>*`\examples\UI Examples\CubeCreator\`.

See also:

- Animation data structures on page 79

Selected Classes and Data Structures

9

FBX SDK has well over 200 documented classes. But thanks in part to the consistency enforced by inheritance, you need master only a relatively few classes.

This section of the *FBX SDK Programmer's Guide* will:

- Help you navigate through the inheritance tree.
- Briefly describe some of the most important classes and data structures.

FBX SDK inheritance tree

FBX SDK Reference provides three kinds of inheritance graphs for the classes in FBX SDK:

- Click FBX SDK Reference > Class Hierarchy for a text-only representation of the full inheritance tree.
- Click FBX SDK Reference > Graphical Class Hierarchy for a diagram of the full inheritance tree.
- For any class that is involved in an inheritance relationship, the reference page will contain an inheritance diagram. Note that for classes with many levels of derived classes (notably `KFbxObject`), this diagram does not show all derived classes.

This topic will use the inheritance tree to guide you to the classes that deserve careful study.

Notes on the inheritance tree:

- Almost half of the classes in FBX SDK derive from `KFbxObject`. See FBX objects: class KFbxObject on page 74 and the reference page for class `KFbxObject`.
- FBX objects such as `KFbxNode` use FBX properties rather than conventional C++ member variables. FBX properties allow strongly typed data to be dynamically added to FBX objects. See FBX properties: class KFbxProperty and KFbxTypedProperty on page 75.
- The FBX scene graph is composed of instances of class `KFbxNode`. See FBX nodes and node attributes on page 77.

- Whether an FBX node is a camera, a mesh, a NURBS, etc. is an attribute of the node, i.e., a *node attribute*. Classes for camera, mesh, NURBS, etc. are derived from `KFbxNodeAttribute`. See Node attributes: class KFbxNodeAttribute on page 77.
- Most container classes are derived from class `KFbxCollection`, whose member functions include `AddMember`, `RemoveMember()`, `FindMember()`, `IsMember()`, `SetSelected()`, etc.
- Many of the methods for importing and exporting files are in class `KFbxIO`, which is the base class for `KFbxImporter` and `KFbxExporter`. See Getting started with file import/export on page 41, and the reference pages for `KFbxIO`, `KFbxImporter`, and `KFbxExporter`.

FBX objects: class KFbxObject

An *FBX object* is an instance of class `KFbxObject` or of any class derived from `KFbxObject`. Almost half of the documented classes in FBX SDK are derived from `KFbxObject`.

An FBX object:

- Must be created using its `Create()` method, which must specify the SDK manager object responsible for managing memory for the FBX object.
- Must be destroyed using its `Destroy()` method.
- Has a unique identifier that is set by FBX and that you can use but cannot change.
- Has a name (e.g., “Camera_1”, “Upper_Arm”) that typically will appear in a user interface. You can set the name of any FBX object to a null string. This name does not need to be unique for FBX SDK. However, other applications might not support duplicate object names.
- Can have a name that is qualified by a namespace (e.g., “MyNameSpace::Camera_1”).
- Can be selected and deselected.
- Contains a set of properties that store and manage strongly typed data. See FBX properties: class KFbxProperty and KFbxTypedProperty on page 75.
- Can be queried to enumerate all the FBX properties that it contains.
- Can use *connections* to manage its relationships with other FBX objects. See Connections on page 78.
- Can contain user data, i.e., can point to a buffer that contains data that is not managed by FBX SDK. See Customizing FBX SDK on page 96.

Copying an FBX object

To copy an FBX object (i.e., an object of any class derived from `KFbxObject`, use the object’s Copy() member function. For example, to create a copy of a mesh object:

```
// Assume that myScene is a pointer to a valid scene object
KFbxMesh* mySourceMesh = KFbxMesh::Create (myScene, "");

. . . // Define control points, etc. for mySourceMesh

// Thhis mesh will be overwritten
KFbxMesh* myTargetMesh = KFbxMesh::Create (myScene, "");

// Copy the data of mySourceMesh into myTargetMesh
myTargetMesh->Copy(mySourceMesh);
```

Notes on copying FBX objects

- To copy an FBX object, always use the target object's `Copy()` function.
- When you copy an FBX object, all FBX properties of the source object are automatically copied to the target object. The name of the source object is also copied. Connections, however, are not copied.
- The source object and the target object must be instances of the same class.
- You cannot use the assignment operator (`operator=`) to copy FBX objects: it is a private member function.
- The `Clone()` member function is under review: we recommend that you do not use it. It's behavior, however, should be unchanged from previous versions of FBX SDK.

FBX properties: class KFbxProperty and KFbxTypedProperty

FBX includes a property system that allows FBX objects to contain strongly typed member data in the form of an *FBX property*. FBX properties can be accessed anad managed by means of a rich library of member functions: `Create()`, `Destroy()`, `DestroyRecursively()`, `IsValid()`, `Get()`, `Set()`, etc.

Although you can define your own properties and data types, let's begin by looking at what you can do with the properties that are already built in to FBX SDK classes. Here, for example, are a few properties of class `KFbxNode`:

Public and fast access Properties

KFbxTypedProperty< fbxDouble3 >	**LclTranslation** *This property contains the translation information of the node.*
KFbxTypedProperty< fbxDouble3 >	**LclRotation** *This property contains the rotation information of the node.*
KFbxTypedProperty< fbxDouble3 >	**LclScaling** *This property contains the scaling information of the node.*
KFbxTypedProperty< KFbxXMatrix >	**GlobalTransform** *This property contains the global transform information of the node.*
KFbxTypedProperty< fbxDouble1 >	**Visibility** *This property contains the visibility information of the node.*
KFbxTypedProperty< fbxDouble1 >	**Weight** *This property contains the weight information of the node.*
KFbxTypedProperty< fbxDouble3 >	**PoleVector** *This property contains the pole vector information of the node.*
KFbxTypedProperty< fbxDouble1 >	**Twist** *This property contains the twist information of the node.*
KFbxTypedProperty< fbxDouble3 >	**WorldUpVector** *This property contains the world up vector information of the node.*
KFbxTypedProperty< fbxDouble3 >	**UpVector** *This property contains the up vector information of the node.*
KFbxTypedProperty< fbxDouble3 >	**AimVector** *This property contains the aim vector information of the node.*
KFbxTypedProperty< fbxBool1 >	**QuaternionInterpolate** *This property contains the quaternion interpolate flag of the node.*
KFbxTypedProperty< fbxDouble3 >	**RotationOffset** *This property contains the rotation offset information of the node.*
KFbxTypedProperty< fbxDouble3 >	**RotationPivot** *This property contains the rotation pivot information of the node.*
KFbxTypedProperty< fbxDouble3 >	**ScalingOffset** *This property contains the scaling offset information of the node.*
KFbxTypedProperty< fbxDouble3 >	**ScalingPivot** *This property contains the scaling pivot information of the node.*
KFbxTypedProperty< fbxBool1 >	**TranslationActive** *This property contains the translation active information of the node.*
KFbxTypedProperty< fbxDouble3 >	**Translation** *This property contains the translation information of the node.*

Extract from the reference page for class KFbxNode

An FBX property:

- Has a name that you can get, set, or find.
- Is implemented as a template class: you must specify the data type of the property.
- Can have a data type that is created at run time. See, for example, sample program ExportScene05 in `<yourFBXSDKpath>\examples\ExportScene05\`.
- Can only be set to a value that is valid for the property's data type. There are some conversions supported: for example, if a `double` is required but an `int` is specified, the int is converted to a `double`, and the property is set to the `double`.
- Contains a set of flags that you can get, set, and manipulate: see `KFbxPropertyFlags::eFbxPropertyFlags`.
- Can be used to manage the connections between FBX objects (See Connections on page 78).
- Can have connections to other properties (see Connections on page 78).
- Can be used with assignment and copy operators.
- Are automatically copied when the FBX object is copied. Copying an FBX object copies all the properties of the source object, as well as the values of each property. The copy does not, however, contain the connections (if any) of the source object.
- Can be a compound of two or more properties. A compound property is organized as a hierarchy of its component properties. See `KFbxProperty::CreateFrom`, `KFbxProperty::DestroyRecursively`, and related member functions.

When you create an FBX object, its FBX properties are automatically instantiated and initialized. These are called *static properties*.

But you can also instantiate and initialize your own additional properties after the FBX object has been created. These are called *dynamic properties*.

You can query an FBX object to:

- Determine whether it contains an FBX property with a specified name.
- Enumerate all its FBX properties.

NOTE In this documentation, an *FBX property* refers to an object that is instantiated from class `KFbxTypedProperty`. Some FBX objects contain member variables that are not implemented as FBX properties, however.

FBX nodes and node attributes

For an introduction to FBX nodes and node attributes, see Traversing the Scene Graph on page 49.

FBX nodes: class KFbxNode

An *FBX node* is an instance of class `KFbxNode`. Each FBX node is a container for one of the elements in an FBX scene, that is, for a camera object, a light object, a mesh object, a NURBS object, etc.

Notes:

- FBX SDK allows two or more nodes to have the same name. However, the file format you use to store your scene may not allow duplicate node names (a *name clash*). In that case, your "extra" nodes will be renamed to *nodename*`_ncl1`, *nodename*`_ncl2`, ... *nodename*`_ncln`.
- FBX File Format (*.fbx*) versions 1.0 to 6.1 do not allow duplicate node names.
- `KFbxNode` provides access to all properties of a node as a point in space, e.g., the node's translation, rotation, and scaling (the TRS properties). See Getting the transformation matrix for a node on page 89.
- Limits on a node's TRS properties are accessed in various ways, including by a call to `KFbxNode::GetLimits()`, which returns a reference to a `KFbxNodeLimits` object. See class `KFbxNodeLimits` and sample program ExportScene05.

See also:

- Tutorial 1: Traversing the Scene Graph.

Node attributes: class KFbxNodeAttribute

A `KFbxNode` object is usually paired with another object which is one of the elements in the scene, that is, for a `KFbxCamera` object, a `KFbxLight` object, a `KFbxMesh` object, a `KFbxNurb` object, etc. This second object is called the *node attribute*:

- The node object stores generic data for any scene element (translation, rotation, scaling, etc.).
- The node attribute object stores data specific to a camera, light, mesh, NURBS, etc.
- A node attribute is always a subclass of `KFbxNodeAttribute`. Examples include class `KFbxCamera`, `KFbxLight`, `KFbxMesh`, and `KFbxNurb`.

- Corresponding to each node attribute class is a *node attribute type*, e.g., `eCamera`, `eLight`, `eMesh`, `eNURB`. For a complete list, see `KFbxNodeAttribute::EattributeType`.
- To get a pointer to the node attribute (object) of a particular node (object), call the node's `getAttribute()` member function (see `KFbxNode::GetNodeAttribute()`).

See also:

- Traversing the Scene Graph on page 49
- Getting the attribute type and contents of a node on page 53

Connections

A *connection* is an FBX SDK data structure that manages a two-way relationship between FBX objects and /or FBX properties.

In a connection, one side of the relationship is the *source*, and the other side is the *destination*.

"Source" and "destination" are arbitrary terms that are used in method names (e.g., `GetSRCObjectCount()`, `ConnectDstObject()`) and in the reference documentation. The *semantics* (or the meaning) of the relationship (e.g., parent/child, a material applied to a mesh) usually has nothing to do with sources and destinations.

As you will see below, the semantics of the relationship are almost always meaningful if you make the following "translations":

- Replace "**destination**" by "**container**" or "**contains**".
- Replace "**source**" by "**belongs to**" or "**is a member of**" or "**member**".

Connections can involve FBX objects, FBX properties, or both.

There are four types of connections:

- OO: Object (source) to Object (destination).
- OP: Object (source) to Property (destination).
- PO: Property (source) to Object (destination).
- PP: Property (source) to Property (destination).

Translate "source to destination" as "member of container".

Any object or a property can have, at the same time:

- 0..n connections to source objects.
- 0..n connections to source properties.
- 0..n connections to destination objects.
- 0..n connections to destination properties.

Some relationships between FBX objects are implemented by connections, but the use of connections is not explicit. For example:

- A parent/child relationship between two FBX nodes is normally created by the parent node calling `AddChild (*myChildNode)`. The child node is actually a source object. Translation: "The child belongs to the parent."
- A node's relationship to a node attribute is normally created by the node calling `SetNodeAttribute (KFbxNodeAttribute *pNodeAttribute)`. The node attribute is actually a source object. Translation: "The node contains its node attribute."

Many important and complex relationships among many FBX objects are explicitly implemented by connections. For example:

- Materials are connected as sources to their nodes. One node can be connected to many materials, and one material can be connected to many nodes. Translation: "A material can belong to many nodes, and one node can contain many materials."
- Constraints are implemented by connections that can involve several objects.

Connections are not managed as objects of one particular class. Connections are managed by methods of class `KFbxObject` and class `KFbxProperty`.

KFbxObject has methods that manage the relationship between:

- `this` object and 0..n source objects.
- `this` object and 0..n destination objects.
- `this` object and 0..n source properties.
- `this` object and 0..n destination properties.

These methods include `ConnectSrcObject`, `GetSourceObjectCount`, `Disconnect`, etc.

KFbxProperty has methods that manage the relationship between:

- `this` property and 0..n source objects.
- `this` property and 0..n destination objects.
- `this` property and 0..n source properties.
- `this` property and 0..n destination properties.

Both classes also have methods to search for and retrieve objects and properties according to criteria that can be quite complex. See class `KFbxCriteria`, which is used to specify search criteria.

Animation data structures

To create animation in an FBX scene, you must use several closely related FBX classes: `KFbxAnimStack`, `KFbxAnimLayer`, `KFbxAnimCurve`, `KFbxAnimCurveNode`, `KFbxAnimCurve`, and `KFbxAnimCurveKey`.

NOTE Before FBX SDK 2011, the data structures for animation were the take node (or take), the take node container, take info, among others.

- If you are familiar with these earlier data structures, begin by reading Migrating to the new data structures for animation on page 80.
- For a tutorial on (unblended) animation with FBX data structures, see Animating a camera (or any other FBX object) on page 71.
- For a description of the animation data structures, especially if you need to use blended animation, start with Using animation layers to blend animation on page 82.

Migrating to the new data structures for animation

The FBX SDK animation system has been completely redesigned.

Take nodes, take node containers, current takes, and FCurves have all been replaced by animation stacks (`KFbxAnimStack`), animation layers (`KFbxAnimLayer`) animation curve nodes (`KFbxAnimCurveNode`), and animation curves (`KFbxAnimCurve`).

New class	Deprecated classes and concepts	Comments
`KFbxAnimStack`	`KFbxTakeInfo`, take, current take	The *animation stack* replaces the *take* as the highest-level container for animation. For animated scenes, you need at least one animation stack. `KFbxAnimStack` manages the same data as `KFbxTake`. If you needed multiple takes before, now you need multiple animation stacks. An animation stack as a stack of *animation layers*, e.g., Layer0, Layer1, Layer2, ... etc.
No equivalent	`KFbxTakeNode`	
`KFbxAnimLayer`	No equivalent	An *animation layer* is a container for *animation curve nodes*. Unless you are using blended animation, all you need is one animation layer. This first layer (Layer0 in our example) is often called the *base layer*)
`KFbxAnimCurveNode`	`KFCurveNode`	An *animation curve node* connects one animation curve to one FBX property or channel. The new class is simpler than `KFCurveNode` because there is no need for hierarchies.
`KFbxAnimCurve`	`KFCurve`	The new class has essentially the same interface as `KFCurve`.
`KFbxAnimEvaluator`	`KFbxNode::GetGlobalFromCurrentTakeNode` and similar func-	The new evaluation system provides the same behaviour as the old one, but is much better encapsulated. You can write your own version by deriving from `KFbx-`

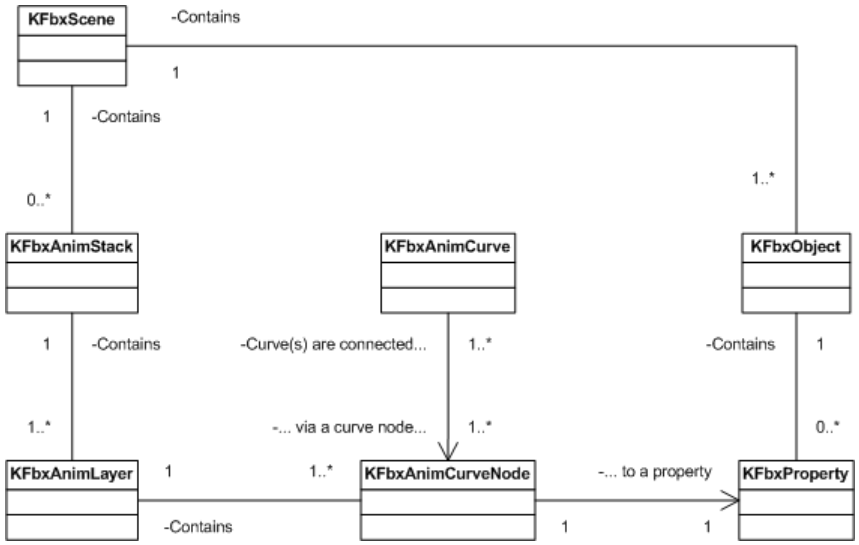
New class	Deprecated classes and concepts	Comments
	tions that `KFbxNode` and many other classes inherited from `KFbx-TakeNodeContainer`	`AnimEvaluator`. See Evaluating the animation in a scene on page 85.

The old animation classes can still be used but have been deprecated (see Avoid deprecated classes and functions on page 98), with the following exceptions:

- `KFbxTakeInfo` has not been deprecated. This means that you can still retrieve take data from an FBX file (into `FbxTakeInfo` objects) without loading the entire file. See Extracting take data (KFbxTakeInfo) from a file on page 85).
- You can initialize a `KFbxAnimStack` object with the data stored in a `KFbxTakeInfo` object. See Extracting take data (KFbxTakeInfo) from a file on page 85 and `KFbxAnimStack::Reset`.

Animation classes and their interrelationships

Here is a UML class diagram showing the relationships between the classes that store animation-related data:



Classes that store animation-related data

- `KFbxScene`: An FBX scene can contain one or more animation stacks. If you do not want animation in your scene or exported to your scene file, then you do not need any animation stacks nor any other animation data.
- `KFbxAnimStack`: The animation stack is the highest level container for animation data. An animation stack is equivalent to one *take* of animation.

  An animation stack contains one or more animation layers. If you are doing blended animation, you need more than one animation layer; otherwise, you need only one animation layer.
- `KFbxAnimLayer`: An animation layer contains one or more animation curve nodes.
- `KFbxAnimCurve`: an animation curve (often called a function curve or FCurve) affects how an animatable attribute of an FBX object varies with time. In FBX, attributes are implemented as FBX properties.

An animation curve can be connected to many curve nodes. Accordingly, one animation curve can animate many properties of many FBX objects.

- `KFbxAnimCurveNode`: An animation curve node is the connection point between animation curves and FBX properties. To connect an animation curve to an FBX property, connect the curve and the property to one curve node.

  A curve node can be connected to only one property of one FBX object. Properties such as `KFbxNode::LclTranslation` contain more than one value (X, Y, Z). If you create a curve node by calling `KFbxAnimCurveNode::CreateTypedCurveNode`, you must specify a property of the data type you need, e.g., `LclTranslation`; function `CreateTypedCurveNode()` will create a curve node with the necessary *animation channels*, in this case, channels X, Y, and Z. The curve node will not actually be connected to the specified property, however. See the Animation sample program in `<yourFBXSDKpath>\examples\Animation\`.

- `KFbxAnimCurveKey` (not shown in diagram): A key (or keyframe) that (typically) marks the beginning or end of an animation curve.

- `KFbxObject`: An FBX object can contain 0 or more FBX properties. Class `KFbxNode` contains the properties that deal with the location of an object as a point in space. Objects that are node attributes (e.g., cameras, meshes, lights) contain the properties that are specific to that node attribute type.

- `KFbxProperty`: An FBX property is strongly typed data that belongs to an FBX object. To animate a scene, animate the appropriate FBX properties (`LclTranslation` is the most common) contained in the FBX objects that are contained in the scene. See FBX properties: class KFbxProperty and KFbxTypedProperty on page 75.

NOTE

FBX SDK needs animation stacks and animation layers to support *blended animation*, a feature of Autodesk MotionBuilder, Autodesk Maya, and other applications.

If you are not familiar with blended animation (and related concepts such as *blend weight* and *blend modes*), the Maya user documentation has a good introduction.

Using animation layers to blend animation

Imagine that you have a set of animation curves that enable a character to walk, and another set that allow the character to run. To create a smooth transition from walking to running, you can *blend* the walking animation with the running animation for perhaps one second.

For blended animation, you need multiple animation layers. For the walk-run example, you would need two animation layers, Layer0 containing the animation curve nodes for the walking animation, and Layer1 containing the animation curve nodes for the running animation.

Animation layers are evaluated in the order that they were added to the animation stack: Layer0, Layer1, Layer2, ... etc.

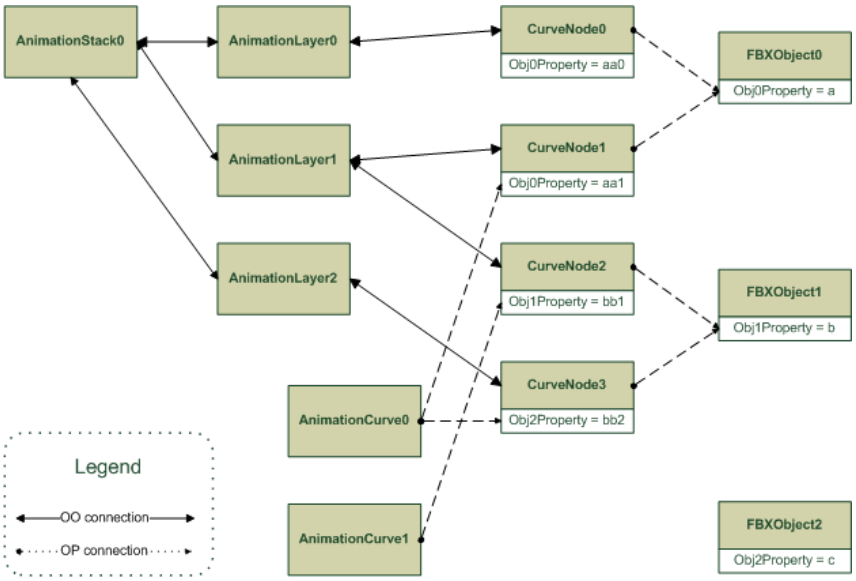
Animation layers have a *Weight* (the blending weight) property, which you can also animate. For example, during the walk-run transition, you would take one-second to change the weight of Layer0 (walking) from .99 to .01, and the weight of Layer1 (running) from .01 to .99.

An animation curve node normally connects one animation curve to one FBX property. But you can also use a curve node to simply *override* the value of an FBX property.

NOTE A curve node also has an unrelated purpose: as a convenient container for the value of FBX property. See Evaluating the animation in a scene on page 85.

Scene showing interrelationships of data structures

Here is a hypothetical scene showing the interrelationships between layers, animation curve nodes, animation layers, FBX objects and their properties:



The scene has three FBX objects (FBXObject0, FBXObject1, and FBXObject2), each of which have one animatable FBX property (Obj0Property, Obj1Property, and Obj2Property). These three properties have default values a, b, and c respectively; these are *default values* because they can be changed by curve nodes and animation curves.

The scene has one animation stack that contains three animation layers:

- AnimationLayer0 (the base layer) has only one animation curve node, CurveNode0. This curve node is not connected to any animation curves, so it is not used to apply animation to an object. The curve node is, however, connected to Obj0Property; this means that in AnimationLayer0, Obj0Property's value of *a* is overridden by the curve node's value of *aa0*.

  In general, in the absence of an animation curve or when ignoring the animation curve, the value of the curve node overrides the value of the property to be animated.

- AnimationLayer1 has two curve nodes:
  - CurveNode1 is connected to Obj0Property and to AnimationCurve0. This means that in AnimationLayer1, the actual value of Obj0Property is neither *a* nor *aa1*, but determined by the values over time of AnimationCurve0.
  - CurveNode2 is connected to Obj1Property and AnimationCurve1. This means that in AnimationLayer1, the actual value of Obj1Property is neither *b* nor *bb1*, but determined by the values over time of AnimationCurve1.

- AnimationLayer 2 has one curve node, CurveNode3, which connects AnimationCurve0 and Obj1Property. This means that in AnimationLayer2, the actual value of Obj1Property is neither *b*nor *bb2*, but determined by the values over time of AnimationCurve0.

Notice that AnimationCurve0 is connected to CurveNode1 and CurveNode3. This means that when the scene is evaluated, AnimationCurve0 will affect two different properties of two different objects in two different animation layers. How this data is evaluated depends not only on the animation curves, but also on the blend mode and the blend weight of each layer.

The FBX animation system gives you a lot of flexibility, but it is your responsibility to ensure that the connections are valid.

Using blend modes to control how a layer blends

The *blend mode* of an animation layer affects how the animation contained in the layer is blended with animation in previous layers in the animation stack. There are three blend modes:

- *Additive mode*: The animation layer "adds" its animation to layers that precede it in the animation stack and that affect the same FBX properties.

  For example, if AnimLayerA and AnimLayerB both contain curve nodes that control the X translation of an FBX node, the resulting X translation is the sum of the X translation values from both layers.

- *Override mode*: The animation layer overrides the animation of any layer that contains curve nodes that control the same FBX properties and that precedes it in the animation stack.

  For example: AnimLayerA and AnimLayerB are both in Override mode. The X translation value for an FBX object on AnimLayerA is 10, and on AnimLayerB X translation has a value of 15. In the resulting animation, the FBX object will have a translateX value of 15.

- *Override-Passthrough mode*: If AnimLayerC is in Override mode, the layer is always completely opaque, blocking all animation from curve nodes in preceding layers that affect FBX properties that are also affected by AnimationLayer3. But when the layer is in Override-Passthrough mode, you control the opacity of the layer by animating its Weight value.

See also:

- enum `KFbxAnimLayer::EBlendMode`
- enum `KFbxAnimLayer::ERotationAccumulationMode`
- enum `KFbxAnimLayer::EScaleAccumulationMode`

Bypassing the blend mode for specified data types

Each animation layer can contain animation curve nodes that affect many FBX properties of different data types. For some of these data types, notably booleans, you probably do not want the animation in one layer to "blend" with the animation in another layer.

Accordingly, each animation layer has a *blend mode bypass* flag for every data type defined in header file `kfbxtype.h`. You can get and set the value of this flag:

- `true`: For FBX properties of the specified data type, the blend mode for this layer will be ignored. Instead, during evaluation (by `KFbxEvaluator`), the values of the properties in this layer will simply override the values in any previous layers.
- `false`: For FBX properties of the specified data type, the blend mode for this layer will be used during evalation.

If you are writing your own evaluator, then you are responsible for processing this flag correctly.

See also:

- `KFbxAnimLayer::GetBlendModeBypass`
- `KFbxAnimLayer::SetBlendModeBypass`

Extracting take data (KFbxTakeInfo) from a file

You can extract and query animation data from FBX files without going through the overhead of loading the entire file. But you'll need to use a concept, a class, and a function from the "old" FBX animation system.

In the new animation system, the animation stack is the highest-level container for animation data. In the old animation system, the *take* is the equivalent to the animation stack.

To extract the animation data, you'll need to first create and initialize an importer object:

```
// Create animporter
KFbxImporter* myImporter = KFbxImporter::Create(mySdkManager,"");

// Initialize by providing a filename
const bool lImportStatus =
    lImporter->Initialize(pFilename, -1, pSdkManager->GetIOSettings() );
```

You can get the number of takes (i.e., animation stacks) in the file:

```
int myTakeCount;
...
myTakeCount = myImporter->GetTakeCount();
```

You can step through the takes, storing the data for each take in a `KFbxTakeInfo` object:

```
for(int i = 0; i < lTakeCount; i++)
{
    KFbxTakeInfo* lTakeInfo = lImporter->GetTakeInfo(i);
...
}
```

And you can use each take info object to initialize the corresponding animation stack object:

```
KfbxAnimstack* myAnimStack // Previously created animation stack object
...
myAnimStack->Reset(myTakeInfo)
```

See also the CubeCreator tutorial program in `<yourFBXSDKpath>\examples\UI Examples\CubeCreator\`.

Evaluating the animation in a scene

To *render* (i.e., display) a scene containing animation, you need to evaluate at many points in time the animated properties of the relevant objects in the scene. For example, if you want to show a moving racecar, and the racecar is represented by a mesh, you must animate the local translation property of the node whose node attribute is the mesh. (see `KFbxNode::LclTranslation`).

Start by getting an evaluator object from the scene object that contains the node:

```
// Let's assume that myScene has been created already
KFbxAnimEvaluator* mySceneEvaluator = MyScene->GetEvaluator();
```

You'll need a time object:

```
KTime myTime;       // The time for each key in the animation curve(s)
myTime.SetSecondDouble(0.0);   // Starting time
```

And a node object:

```
// Create a node object
KFbxNode* myMeshNode = KFbxNode::Create (myScene, "");
... // Code to connect myMeshNode to a mesh object
```

Then you can get the global transformation matrix of the relevant node at the specified time:

```
// Get a reference to node's global transform.
KFbxXMatrix& GlobalMatrix =
    mySceneEvaluator->GetNodeGlobalTransform(myMeshNode, myTime);
```

Or, if you prefer, the node's local transformation matrix at the specified time:

```
// Get a reference to node's local transform.
KFbxXMatrix& GlobalMatrix =
    mySceneEvaluator->GetNodeLocalTransform(myMeshNode, myTime);
```

To animate a camera, you don't need a transformation matrix. All you need is the camera's position, expressed as a vector, at the specified time:

```
// Given a scene, we can create a camera
KFbxCamera* myCamera = KFbxCamera::Create (myScene, "");

// Store the value of the property... in an animation curve node!?
 KFbxAnimCurveNode& myCameraPositionVector;

// Get and store the value of the camera's local translation
myCameraPositionVector = mySceneEvaluator->GetPropertyValue (myCamera->Position, myTime);
```

NOTE We are not using the `KFbxAnimCurveNode` object for its normal purpose, that is, we are not using it as a connection point between an animation curve and an FBX property.

We are using an animation curve node as a convenient place to store the vector of the property returned by the evaluator. That's because the animatable properties of FBX objects come in many data types. Some data types are for scalars; examples include `fbxDouble1`, `fbxInteger1`, and `fbxBool1`. Other data types have values that are (X, Y, Z) triplets that are stored as vectors.

In our code snippet above, `KFbxNode::LclTranslation` is of type `fbxDouble3`, a vector with three elements. `KFbxAnimCurveNode` is a container that can store the value of any FBX property no matter what its type. And `KFbxAnimCurveNode`'s member functions allow you to access the value of each element in a vector by, in this case, getting the value of a channel X, the value of channel Y, and the value of channel Z.

See also:

- `KFbxNode::LclTranslation`
- `KFbxAnimCurveNode::GetChannelCount`, `KFbxAnimCurveNode::GetChannelValue`, and related functions.
- `kfbxdatatypes.h`, for a full list of FBX data types.

Writing and using your own evaluator

Class `KFbxAnimEvaluator` is the class to use when you need to evaluate the animation in a scene:

```
// Let's assume that myScene has been created already
KFbxAnimEvaluator* mySceneEvaluator = MyScene->GetEvaluator();
```

But `KFbxAnimEvaluator` is actually an *abstract class*. When you call `KFbxScene::GetEvaluator` in the above code snippet, the function actually creates a `KFbxAnimEvalClassic` object. `KFbxAnimEvalClassic` is the only implementation of `KFbxAnimEvaluator` in this release of FBX SDK.

You can, however, implement your own evaluator class derived from `KFbxAnimEvaluator`. To use your evaluator class, call `KFbxScene::SetEvaluator`, setting the evaluator to your own evaluator object. Then, make this call, which is exactly the same call that you see in the above code snippet:

```
KFbxAnimEvaluator* mySceneEvaluator = MyScene->GetEvaluator();
```

This time, `mySceneEvaluator` will point to an object of your own evaluator class.

Advanced Topics

10

This section provides brief descriptions of several advanced topics.

Getting the transformation matrix for a node

To get the default translation, rotation, and scaling (*default TRS properties*) of a node, access the node's `LclTranslation`, `LclRotation`, and `LclScaling` properties:

```
KFbxNode myNode;   // A properly initialized node object

// Get the node's default TRS properties
fbxDouble3 myNodeLclTranslation = myNode->LclTranslation.Get();
fbxDouble3 myNodeLclRotation    = myNode->LclRotation.Get();
fbxDouble3 myNodeLclScaling     = myNode->LclScaling.Get();
```

All three member functions return vectors of type `fbxDouble3`. The value of each vector is a triplet of X, Y, and Z coordinates. The value of one of these vectors is an offset from the corresponding default TRS property vector for the parent node. A node's default TRS properties are therefore *local* to the parent node.

The *actual TRS properties* for the node at a given point in time depend on:

- The node's default TRS properties.
- The *limits* that apply to the node (see `KFbxNode::GetLimits`).
- The *constraints* that apply to the node (see `KFbxConstraint` and its subclasses.
- The *animation curves* (often called *function curves* or *FCurves*) of the *current animation stack* (*current take*). (See Animating a Scene on page 69 and Animation data structures on page 79.)

You can get the node's TRS properties in the scene's global coordinate system expressed as a *transformation matrix* (often called the *transform*) by calling the `GetNodeGlobalTransform` member function of the scene's evaluator and passing the node as a parameter (see `KFbxKFbxAnimEvaluator::GetNodeGlobalTransform`). This function allows you to get:

- The node's default TRS properties, returned as a transformation matrix.
- The node's actual TRS properties at a specified point in time, returned as a transformation matrix.

```
KFbxAnimEvaluator* mySceneEvaluator = myScene->getEvaluator();

// Get node's default TRS properties as a transformation matrix
KFbxXMatrix& myNodeDefaultGlobalTransform =
    mySceneEvaluator->GetNodeGlobalTransform(myNode);

// Get transform containing node's actual TRS properties at a point in time
Ktime myTime;    // Defaults to myTime=0
KFbxXMatrix& myNodeActualGlobalTransform =
    mySceneEvaluator->GetNodeGlobalTransform(myNode, myTime);
```

See also:

- Getting the properties of a node as a point in space
- FBX property `KFbxNode::LclTranslation`
- FBX property `KFbxNode::LclRotation`
- FBX property `KFbxNode::LclScaling`
- `KFbxKFbxAnimEvalClassic::GetNodeGlobalTransform`
- Evaluating animation

How transformation matrices are computed

FBX SDK and Maya use the same formula to compute a transformation matrix. The formula used by 3ds Max is different.

NOTE The FBX importers and exporters for 3ds Max automatically convert transformation matrices to and from 3ds Max.

FBX and Maya

Here is how FBX SDK and Maya compute the transformation matrix for a node:

$$WorldTransform = ParentWorldTransform * T * Roff * Rp * Rpre * R * Rpost * Rp^{-1} * Soff * Sp * S * Sp^{-1}$$

Where this term:	Is a 4 x 4 matrix that contains:
`WorldTransform`	Transformation matrix of the node
`ParentWorldTransform`	Transformation matrix of the parent node
`T`	Translation
`Roff`	Rotation offset
`Rp`	Rotation pivot
`Rpre`	Pre-rotation
`R`	Rotation

Where this term:	Is a 4 x 4 matrix that contains:
`Rpost`	Post-rotation
Rp^{-1}	Inverse of the rotation pivot
`Soff`	Scaling offset
`Sp`	Scaling pivot
`S`	Scaling
Sp^{-1}	Inverse of the scaling pivot

Notes:

- Computations are performed from left to right.
- The effect of the formula is that any given vector is first translated, then rotated, then scaled.
- The `R` matrix takes into account the *rotation order*. Because of the mathematical properties of the matrices, `R` is the result of one of the possible combinations of `Ry`, `Ry` and `Rz` (each being matrices also). For example, for the default rotation order of XYZ, `R = Rx * Ry * Rz`

3ds Max

Here is how 3ds Max computes the transformation matrix for a node. All the terms in the equation are the same as in FBX and Maya, except for the three terms that represent geometric transformation:

```
WorldTransform = ParentWorldTransform * T * R * S * OT * OR * OS
```

Where this term:	Is a 4 x 4 matrix that contains:
`WorldTransform`	Transformation matrix of the node
`ParentWorldTransform`	Transformation matrix of the parent node
`T`	Translation
`R`	Rotation
`S`	Scaling
`OT`	Geometric transform translation
`OR`	Geometric transform rotation
`OS`	Geometric transform translation

Notes:

- Computations are performed from left to right.
- Geometric translation, geometric rotation, and geometric scaling relate to the object-offset concept in 3ds Max. These geometric transformations are applied to the node attribute after the node transformations.
- Geometric transformations are not inherited: `ParentWorldTransform` does not contain the OT, OR, OS of `WorldTransform`'s parent node.

- Geometric transformations are implemented in FBX SDK as three properties of `KFbxNode` objects: `KFbxNode::GeometricTranslation`, `KFbxNode::GeometricRotation`, and `KFbxNode::GeometricScaling`.

Using multiple geometry layers

This topic will use diagrams to show how to use multiple geometry layers (class `KFbxLayer`) to create a **layered texture** (class `KFbxLayeredTexture`). A layered texture consists of several textures (class `KFbxTexture`) that are partially transparent, and that are overlaid to create a texture that is a blending of the original textures.

This topic also provides a visual review of *geometry layers* and *geometry layer elements*, two FBX concepts that were introduced in Applying Textures and Materials to Meshes on page 57. That section described a mesh in the shape of a cube.

Here is a mesh in the form of a plane. Like the cube, it has six polygons, numbered 0 through 5:

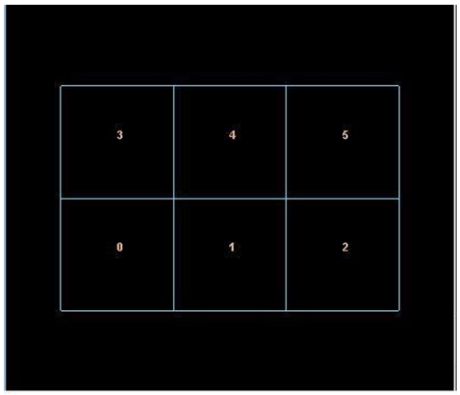


Figure 1 - Polygons IDs on the Mesh

We have five materials (M0 through M4) that we want to apply to the six polygons. We'll apply the red material M0 to polygons 0 and 2:

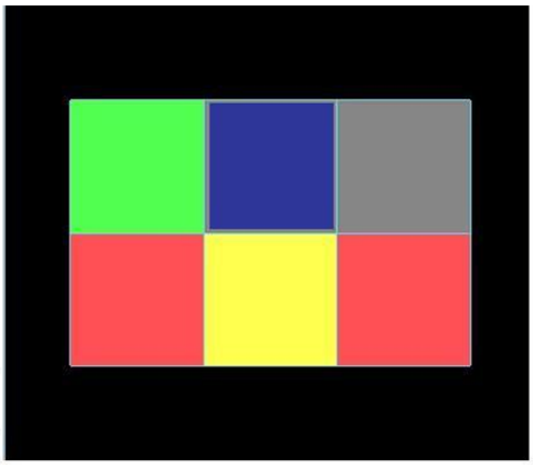
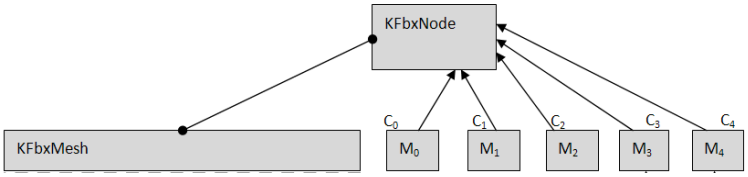


Figure 2 - Materials assigned by polygon

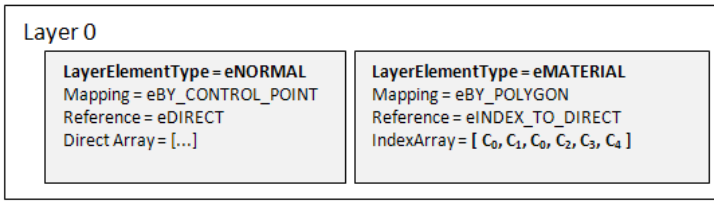
M0 = Red material
M1 = Yellow material
M2 = Green
M3 = Blue
M4 = Grey

Let's look at this in terms of FBX classes:

- The mesh object is the node attribute of a node object (class `KFbxNode`).
- The five material objects are connected to the node object (**not** the mesh object) by five connections, C0 to C4 (see Connections).



To do this, we create a geometry layer as a container for two geometry layer elements: normals and materials. The layer is Layer 0. We recommend that you use Layer 0 and only Layer 0, unless you have a particular reason for using another layer or layers. We'll see a situation that requires multiple layers later on in this topic.



The normals for the 24 vertices (control points) are contained in a `KFbxLayerElementNormal` object. We'll map the normals to the control points of the polygons. And we'll use a DirectArray of 24 items, where each item is a `KFbxVector4` object containing a normal. "Reference = eDIRECT" means that FBX SDK will refer to the elements of the DirectArray directly, i.e., without using an IndexArray.

We also need a `KFbxLayerElementMaterial` object, whose IndexArray contains the index numbers of connections. In our example, there are six elements in the IndexArray, one for each polygon in the mesh. Each IndexArray element contains the index number of the connection for that polygon. This maps the five connections (one connection for each of the five materials) to the six polygons of the mesh. The red material (M0) is applied to polygon 0 and polygon 2: that's why C0 is in element #0 and element #2 of the IndexArray.

NOTE Use eINDEX_TO_Direct for the reference mode, even though class `KFbxLayerElementMaterial` does not use a DirectArray.

Now we want to go two steps farther:

- We apply a texture to polygon 4. The texture contains an image of the number 1.
- We apply a layered texture to polygon 3. The layered texture will be a blending of three textures. The three textures contain the images of the numbers 1, 2, and 3, respectively

When we are done, the mesh will look like this:

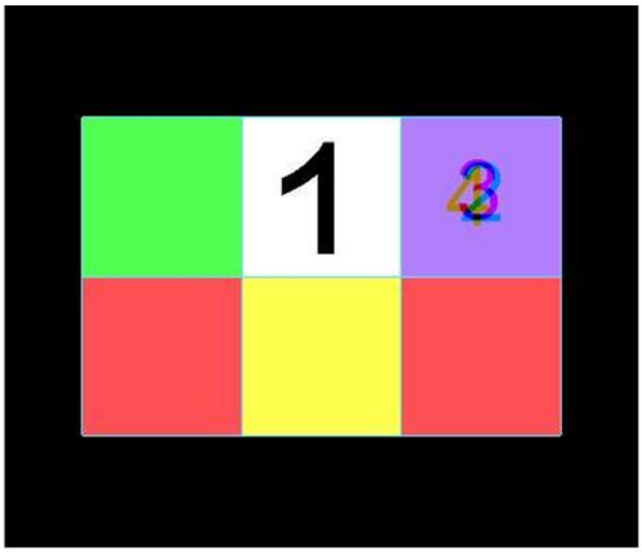


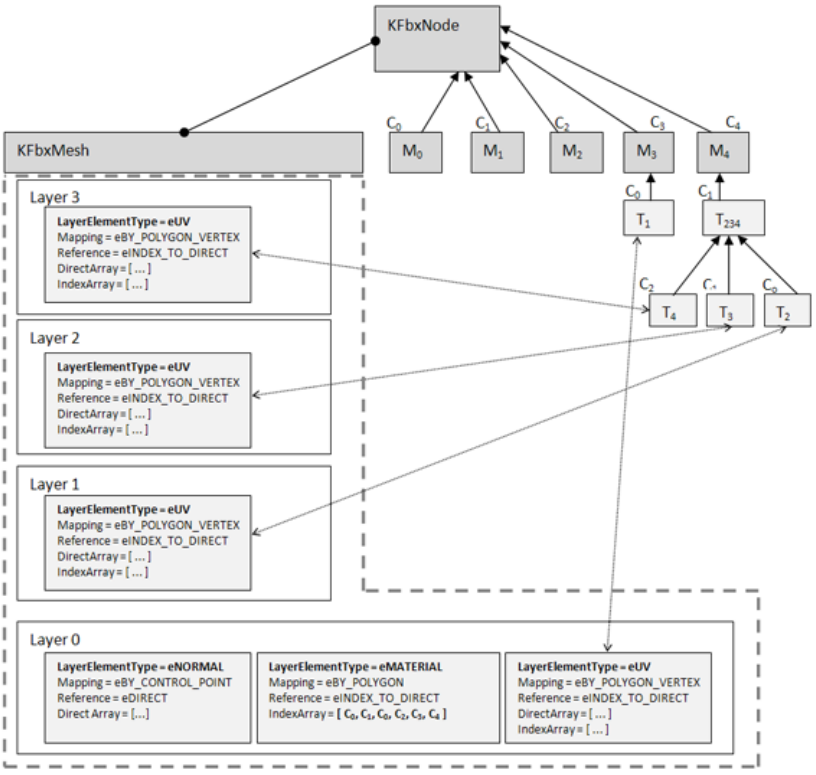
Figure 3 - Textures applied on 2 polygons

T1 = Single texture

T234 = Layered texture composed of textures T2,T3 and T4

The textures with the number 1 is called T1. To apply the texture to polygon 4, we create a layer element for UVs in Layer 0. This layer element is an object of class `KFbxLayerElementUV`. And as you will see from the Figure 3, to apply texture T1 to polygon 4, we connect T1 to the material M3 (which is itself connected to polygon 4), and use the UVs to position the texture on the material.

The texture with number 2 is called T2. The texture with number 3 is called T3. The texture with number 4 is called T4. Each of these three textures needs a separate layer element (Layers 1, 2, and 3, respectively) so they can be blended together into a layered texture (class `KFbxLayeredTexture`).



Layered textures are an example of a case when you cannot use only Layer 0.

See also:

- `KFbxLayerElement::ELayerElementType` is an enum that maps identifiers such as `eUV`, `eNORMAL`, and `eMATERIAL` to their corresponding classes (`KFbxLayerElementUV`, `KFbxLayerElementNormal`, `KFbxLayerElementMaterial`, ...). You must supply the enum as a parameter to `KFbxLayer::GetLayerElementOfType()`.
- `KFbxLayerElement::EMappingMode` is an enum that specifies how the layer element is mapped to the surface.
- `KFbxLayerElement::EReferenceMode` is an enum that specifies how the IndexArray and DirectArray of a layer element are to be referenced.

Storing animation in a vertex cache

In FBX, geometries can be deformed using skinning, shapes, or *vertex caches*. A vertex cache is a way to store directly the vertex animations inside a *cache file*.

For sample code that creates and animates vertices using a vertex cache, see functions `MapVertexCacheOnTriangle()` and `AnimateVertexCacheOnTriangle()` in sample program ExportScene03 (see ExportScene03 on page 29).

For sample code that reads data in a vertex cache, see functions `PreparePointCacheData()` and `ReadVertexCacheData()` of sample program ViewScene (see ViewScene on page 30).

Using hardware shaders to create materials

CGFX and DirectX hardware shaders are supported by FBX SDK starting with version 2010. You can set up a material with a CGFX or DirectX implementation using class `KFbxImplementation` and class `KFbxBindingTable`.

Creating UV sets for different texture channels

You can create different UV sets for different texture channels. In sample program ExportScene03, function `CreateCubeWithTexture()` creates UVs for the diffuse, ambient, and emissive texture channels (see ExportScene03 on page 29).

Creating metadata about nodes

Class `KFbxObjectMetaData` holds metadata about nodes. The metadata are stored as properties.

For an example of creating metadata, look at function `CreateScene()` in sample program ExportScene05 (see ExportScene05 on page 29).

For an example of how to access metadata, look at functions `DisplayMetaData()` and `DisplayMetaDataConnections()` in sample program ImportScene (see ImportScene on page 30).

Customizing FBX SDK

You can customize or extend the features of FBX SDK in various ways, including user data (custom data), custom properties, custom data types, and more.

Defining user data for FBX objects

`KFbxObject` and its subclasses allow you set and get a pointer to any data record (usually an object or a struct). You are responsible for creating, destroying, and otherwise managing this data. See `KFbxObject::SetUserDataPtr` and `KFbxObject::GetUserDataPtr`.

Defining custom properties (user properties)

You can define *custom properties* for any object. To learn how, look at sample program UserProperties (see UserProperties on page 30).

To learn how how to get data from custom properties, see function `DisplayUserProperties()` in ImportExport tutorial program on page 31.

Defining a custom object type

The FBX SDK object model can be extended with a *custom object type*. For an example, see how a new class `MyKFbxMesh` is created from class `KFbxMesh` in sample program ExportScene03 (see ExportScene03 on page 29).

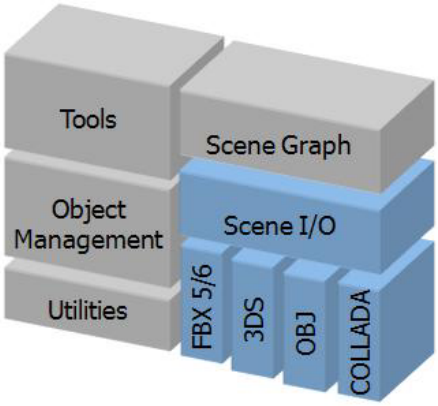
Defining a custom type (user data) for layer elements

To create a layer element with a custom type, use class `KFbxLayerElementUserData`. Like any other layer element, it can be mapped by polygon vertex, by vertex, by polygon, etc.

See function `CreateCubeWithMaterialAndMyKFbxMesh()` in sample program ExportScene03 (see ExportScene03 on page 29). It creates a custom compound based on float and boolean data types, and adds data per vertex.

Supporting additional file formats

FBX imports and exports scene data using several file formats, including FBX, COLLADA, OBJ, etc.



Each of these file formats has its own *writer* class (derived from class `KFbxWriter`) reader and *reader* class (derived from class `KFbxReader`).

FBX SDK implements scene I/O using a *plug-in* architecture. This makes it relatively easy to extend FBX SDK so that it supports an additional file format (usually called a *custom file format*) of your own design.

NOTE Do not confuse scene I/O plug-ins with the FBX plug-ins to 3ds Max/Maya. Scene I/O plug-ins are a component of FBX SDK. The FBX Plug-ins to 3ds Max/Maya are described in FBX plug-ins for Autodesk 3ds Max and Autodesk Maya on page 7.

Sample program MyOwnReaderWriter (see Sample programs on page 25) shows how to use FBX SDK to write and read "`CustomWriter`" files, which use file extension *.abc*. In general, to support your own custom file format, you must:

- Implement your own writer class (see `MyOwnWriter.cpp` and `MyOwnWriter.h`).
- Implement your own reader class (see `MyOwnReader.cpp` and `MyOwnReader.h`).

You must also use class `KFbxIOPluginRegistry` (see `MyOwnWriterReaderPlugin.cpp` and `MyOwnWriterReaderPlugin.h`) to:

- *Register* your reader and writer as a scene I/O plug-in.
- Register your own file extension (which need not be unique).
- Register your own *file description* (which must be unique).

The `MyOwnWriterReader` folder does not contain a stand-alone sample program. Instead, it contains sample functions that support `CustomWriter` files. The sample functions are in turn used by other sample programs (ExportScene05 on page 29, ImportScene on page 30, and ViewScene on page 30) to write and read `CustomWriter` files.

See also Extending FBX or Collada file formats on page 97.

Extending FBX or Collada file formats

FBX Extensions SDK allows you to extend FBX file format (versions 5 and 6) and the Collada file format. You can:

- Extend the FBX and/or Collada file format to support additional kinds of data.

- Extend the functionality of the FBX plug-ins for 3ds Max/Maya to support the reading and writing of files that use your extended file format.
- Use FBX SDK directly to read and write files that use your extended file format, without passing through 3ds Max or Maya.

FBX Extensions SDK contains:

- Source code and project files for the scene I/O plug-ins that read and write scenes to FBX files.
- Source code and project files for the scene I/O plug-ins that read and write scenes to Collada files.
- Library files that link to the 3ds Max/Maya plug-ins.
- Documentation.

You can download FBX Extensions SDK by going to *http://www.autodesk.com/fbx*.

Creating objects that are destroyed with their scenes

This code snippet creates an SDK manager object, and uses it to create a scene object, and then a node object:

```
// Create SDK manager
KFbxSdkManager* mySdkManager = KFbxSdkManager::Create();

// Create a scene object
KFbxScene* myScene = KFbxScene::Create(mySdkManager, "");

// Create a node object
KFbxNode* myNode = KFbxNode::Create(mySdkManager, "");
```

Although passing the application's SDK manager object as a parameter is an acceptable way to create objects, we recommend that objects belonging to a given scene be created by passing the scene object as a parameter instead of the SDK manager. This way, these objects will be automatically destroyed when the scene object is destroyed (since they belong to the scene object).

If they were created with the SDK manager object as parameter, they would not be destroyed until the manager is destroyed (and searching for them from the scene would fail).

The example would then become:

```
// Create SDK manager
KFbxSdkManager* mySdkManager = KFbxSdkManager::Create();

// Create a scene object
KFbxScene* myScene = KFbxScene::Create(mySdkManager, "");

// Create a node object
KFbxNode* myNode = KFbxNode::Create(myScene, "");
```

Avoid deprecated classes and functions

In every release of FBX SDK, new classes and functions are added, while others are *deprecated*. A deprecated function is declared as `K_DEPRECATED`. Here is an example:

```
K_DEPRECATED KFbxTakeNode* GetDefaultTakeNode();
```

You can continue to use deprecated functions, but:

- Your compiler will throw a warning.
- Functions that are deprecated in the current release will be removed completely in the next release.

Support for UTF-8 strings and other formats

FBX SDK uses UTF-8 strings internally. UTF-8 (8-bit UCS/Unicode Transformation Format) is a variable-length character encoding for Unicode. It is able to represent any character in the Unicode standard, yet is backwards compatible with ASCII. For a good introduction to UTF-8, see Wikipedia (wikipedia.com).

FBX will convert its UTF-8 strings to the required string format when calling a function in an operating system's API. For example when calling a Windows API function requiring a Unicode `wide char *`, FBX calls a MACRO of KString to make the conversion.

So, if your application uses non-ASCII `char`, you must convert your strings to UTF-8 `char *` C-strings before passing them to FBX. *KString.h* has six macros that you can use:

Macro
KFBX_UTF8_to_WIDECHAR
KFBX_WIDECHAR_to_UTF8
KFBX_WIDECHAR_to_ANSI
KFBX_ANSI_to_WIDECHAR
KFBX_ANSI_to_UTF8
KFBX_UTF8_to_ANSI

Scripting with Python FBX

11

Python FBX is a *Python binding* for the C++ library of FBX SDK. It allows you to write Python scripts that can use many of the classes and member functions of FBX SDK.

Python FBX is suitable for applications that can:

- Set Import/Export options.
- Browe a scene's node hierarchy.
- Browse the scene's objects to determine their types.
- Get/Set property values for selected objects.
- Use the above features to import/export files of any of the file formats supported by FBX.

This section explains how to install, configure, and script with Python FBX.

Platforms supported

Python FBX is distributed with FBX SDK. There are separate distributions of FBX SDK for Windows, Linux, and Mac OS. Within these distributions, there are separate directories for each combination of Python version (2.6 or 3.1) and machine type (32-bit or 64-bit).

For Windows, these versions of Python FBX are provided:

Directory name	Python version, machine type
Python26_x86	2.6, 32-bit
Python31_x86	3.1, 32-bit
Python26_x64	2.6, 64-bit
Python31_x64	3.1, 64-bit

For Linux, these versions of Python FBX are provided:

Directory name	Python version, machine type
Python26_x64	2.6, 64-bit, 2-byte Unicode characters

Directory name	Python version, machine type
Python31_x64	3.1, 64-bit, 2-byte Unicode characters
Python26_ucs4_x64	2.6, 64-bit, 4-byte Unicode characters
Python31_ucs4_x64	3.1, 64-bit, 4-byte Unicode characters

For Mac OS, these versions of Python FBX are provided:

Directory name	Python version, machine type
Python26	2.6, 32-bit
Python31	3.1, 32-bit

NOTE Remember the directory name of the Python version that you wish to use with Python FBX. You'll need it for Installing Python FBX on page 102.

Installing Python FBX

We assume that you have already installed and configured Python 2.6 or Python 3.1 on your computer. Let's call the directory where you installed Python *yourPythonPath*.

To install Python FBX:

1. Follow the instructions of Installing FBX SDK for your development platform (see Installing and Configuring on page 17).

   Let's call the directory where you installed FBX SDK *<yourFBXSDKpath>*.

2. Determine the directory name (<Python*xxxxxx*>) of the version of Python FBX that you wish to install (see Platforms supported on page 101).

3. Copy the contents of *<yourFBXSDKpath>*\lib\<Python*xxxxxx*>\ to *yourPythonPath*\Lib\site-packages\.

4. Optional: Copy the sample programs for Python FBX to a suitable location. The sample programs are in *<yourFBXSDKpath>*\examples\Python\.

5. Optional: Copy the documentation for FBX SDK to a suitable location. The documentation is in *<yourFBXSDKpath>*\doc.

6. Optional: Delete *<yourFBXSDKpath>* and its contents. If you have copied all the directories mentioned above, you do not need the rest of the distribution for FBX SDK.

Importing FBX libraries into your Python script

To write a script using Python FBX, you must import the Python FBX libraries into your script. Put the following statement at the beginning of your script:

```
from fbx import *
```

Now you can write scripts using any of the classes and member functions of Python FBX.

Classes and member functions

Python FBX provides Python equivalents to most of the classes and member functions of FBX SDK. For documentation about specific classes and member functions, see the documentation for the equivalent C++ class or member function.

Differences between FBX SDK and Python FBX

Python FBX contains most of the classes and member functions that are available in FBX SDK itself. Here are the important differences:

Function templates and class templates not provided

C++ supports *function templates*. These are functions for which the type of data on which the function operates can be specified as a parameter. *Class templates* are classes whose member functions can be function templates.

Python does not support templates. Accordingly, Python FBX cannot have classes and functions that correspond to the C++ class templates and function templates. But Python FBX does provide some support:

- FBX SDK has functions that are *overloaded* (i.e, there are various versions of the function that share the same function name). In some cases (e.g., `KFbx Property::DisconnectAllSrcObjects`), some of the overloaded functions are template functions, while others are ordinary functions. In these cases, Python FBX does provide the ordinary functions.
- For each template function in FBX SDK, Python FBX provides a Python version that accepts `double` parameters and/or returns a `double` return value. That allows you to also use a `float`, `int`, or any other Python data type that can be converted to a `double`.
- For each template function in FBX SDK, Python FBX does provide a version that accepts `str` parameters and/or returns a `str` return value.

An array in C++ becomes a list in Python

In C++, a pointer veriable is sometimes used to pass an array as a parameter, or to return an array as a return value. For example, in this C++ function, `int*` (a pointer to an integer) actually refers to an array of integers:

```
int* KFbxMesh::GetPolygonVertices();
```

In FBX Python, this array of integers is converted to a `list` of type `int`.

Call by reference becomes an extra return value

In C++, a pointer variable (e.g., `int *pLast` is sometimes used with a simple type (such as `int`, `float`, `double`) to support call-by-reference. A call-by-reference parameter allow programmers to not only pass a value to a function, but also to return the new value for the parameter (if the function changes it).

In Python, call-by-reference is not supported. But Python does support functions returning *tuples* (i.e., two return values). C++ does not support tuples as return values.

`KFbxAnimCurve::KeyAdd` is an example of how Python FBX uses tuples as a replacement for call-by-reference. In the C++ version of FBX SDK, the signature for `KFbxAnimCurve::KeyAdd` looks like this:

```
int KeyAdd(KTime pTime, int *pLast);
```

For this function, `int *pLast` is used only to return a value from the function, not to pass a value to it. For the Python FBX, the signature has been converted to this:

```
(int, int) KeyAdd(KTime pTime);
```

Python FBX has converted `int *pLast` to a second `int` return value.

Let's consider an imaginary C++ function where a pointer variable is used both to pass a value and to return a value:

```
int imaginaryFunction(int *myParameter);
```

For Python, the signature would be converted to this:

```
(int, int) imaginaryFunction(int myParameter);
```

In this case, Python FBX makes two changes. It converted `int *myParameter` to:

- `int myParameter` (to pass an integer value into the function).
- The second `int` return value (to return the changed integer value).

List of Python FBX classes

Here are the classes included in Python FBX. In *FBX SDK Help*, each of these classes is linked to the corresponding C++ documentation in *FBX SDK Reference*.

- `FbxPropertyFlags`
- `KArrayTemplate`
- `KDefaultRenderResolution`
- `KError`
- `KFbx`
- `KFbxAnimCurve`
- `KFbxAnimCurveDef`
- `KFbxAnimCurveKey`
- `KFbxAnimCurveNode`
- `KFbxAnimLayer`
- `KFbxAnimStack`
- `KFbxAxisSystem`
- `KFbxBoundary`
- `KFbxCache`
- `KFbxCamera`
- `KFbxCameraStereo`
- `KFbxCameraSwitcher`
- `KFbxCharacter`
- `KFbxCharacterLink`
- `KFbxCharacterPose`
- `KFbxCluster`
- `KFbxCollection`

- KFbxCollectionExclusive
- KFbxColor
- KFbxConnectionPointFilter
- KFbxConstraint
- KFbxConstraintAim
- KFbxConstraintParent
- KFbxConstraintPosition
- KFbxConstraintRotation
- KFbxConstraintScale
- KFbxConstraintSingleChainIK
- KFbxControlSet
- KFbxControlSetLink
- KFbxControlSetPlug
- KFbxCriteria
- KFbxDataType
- KFbxDeformer
- KFbxDisplayLayer
- KFbxDocument
- KFbxDocumentInfo
- KFbxEffector
- KFbxExporter
- KFbxFileHeaderInfo
- KFbxGeometry
- KFbxGeometryBase
- KFbxGlobalCameraSettings
- KFbxGlobalSettings
- KFbxIO
- KFbxIOPluginRegistry
- KFbxIOSettings
- KFbxImporter
- KFbxLayer
- KFbxLayerContainer
- KFbxLayerElement

- KFbxLayerElementArray
- KFbxLayerElementArrayTemplate
- KFbxLayerElementBinormal
- KFbxLayerElementCrease
- KFbxLayerElementMaterial
- KFbxLayerElementNormal
- KFbxLayerElementPolygonGroup
- KFbxLayerElementSmoothing
- KFbxLayerElementTangent
- KFbxLayerElementTemplate
- KFbxLayerElementTexture
- KFbxLayerElementUV
- KFbxLayerElementVertexColor
- KFbxLayerElementVisibility
- KFbxLayeredTexture
- KFbxLight
- KFbxLimits
- KFbxLodGroup
- KFbxMarker
- KFbxMatrix
- KFbxMesh
- KFbxNameFilter
- KFbxNode
- KFbxNodeAttribute
- KFbxNodeLimits
- KFbxNull
- KFbxNurb
- KFbxNurbsCurve
- KFbxNurbsSurface
- KFbxObject
- KFbxObjectFilter
- KFbxPatch
- KFbxPeripheral

- KFbxPlug
- KFbxPose
- KFbxProgress
- KFbxProperty
- KFbxQuaternion
- KFbxReader
- KFbxRenamingStrategy
- KFbxScene
- KFbxSceneRenamer
- KFbxSdkManager
- KFbxSelectionNode
- KFbxSelectionSet
- KFbxShape
- KFbxSkeleton
- KFbxSkin
- KFbxStatistics
- KFbxSubDeformer
- KFbxSurfaceLambert
- KFbxSurfaceMaterial
- KFbxSurfacePhong
- KFbxSystemUnit
- KFbxTakeInfo
- KFbxTexture
- KFbxThumbnail
- KFbxTrimNurbsSurface
- KFbxTypedProperty
- KFbxUserNotification
- KFbxVector2
- KFbxVector4
- KFbxVertexCacheDeformer
- KFbxVideo
- KFbxXMatrix
- KName

- KNumberRenamingStrategy
- KRenamingStrategy
- KString
- KStringList
- KTime
- KTimeSpan
- LockAccessStatus
- fbxDateTime
- fbxDistance
- fbxReference
- fbxVectorTemplate2
- fbxVectorTemplate3
- fbxVectorTemplate4
- kFbxClassId

Index

V

W