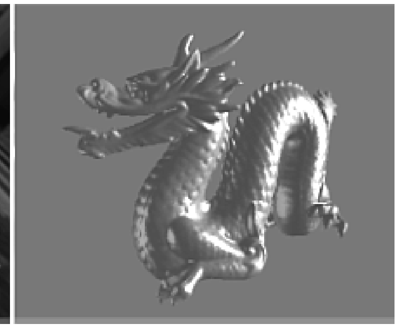
Asset-Importer-Lib

Release September 2019

Apr 12, 2020

General

CHAPTER 1

Introduction

assimp is a library to load and process geometric scenes from various data formats. It is tailored at typical game scenarios by supporting a node hierarchy, static or skinned meshes, materials, bone animations and potential texture data. The library is *not* designed for speed, it is primarily useful for importing assets from various sources once and storing it in a engine-specific format for easy and fast every-day-loading. assimp is also able to apply various post processing steps to the imported data such as conversion to indexed meshes, calculation of normals or tangents/bitangents or conversion from right-handed to left-handed coordinate systems.

assimp currently supports the following file formats (note that some loaders lack some features of their formats because some file formats contain data not supported by assimp, some stuff would require so much conversion work that it has not been implemented yet and some (most . . .) formats lack proper specifications):

- **Collada** (.dae, .xml)
- **Blender** (.blend)
- **Biovision BVH** (.bvh)
- **3D Studio Max 3DS** (.3ds)
- **3D Studio Max ASE** (.ase)
- **Wavefront Object** (.obj)
- **Stanford Polygon Library** (.ply)
- **AutoCAD DXF** (.dxf)
- **IFC-STEP** (.ifc)
- **Neutral File Format** (.nff)
- **Sense8 WorldToolkit** (.nff)
- **Valve Model** (.smd, .vta)
- **Quake I** (.mdl)
- **Quake II** (.md2)
- **Quake III** (.md3

- **Quake 3 BSP** (.pk3)
- **RtCW** (.mdc)
- **Doom 3** (.md5mesh, .md5anim, .md5camera)
- **DirectX X** (.x)
- **Quick3D** (.q3o, .q3s)
- **Raw Triangles** (.raw)
- **AC3D** (.ac)
- **Stereolithography** (.stl)
- **Autodesk DXF** (.dxf)
- **Irrlicht Mesh** (.irrmesh, .xml)
- **Irrlicht Scene** (.irr, .xml)
- **Object File Format** (.off)
- **Terragen Terrain** (.ter)
- **3D GameStudio Model** (.mdl)
- **3D GameStudio Terrain** (.hmp)
- **Ogre** (.mesh.xml, .skeleton.xml, .material)
- **Milkshape 3D** (.ms3d)
- **LightWave Model** (.lwo)
- **LightWave Scene** (.lws)
- **Modo Model** (.lxo)
- **CharacterStudio Motion** (.csm)
- **Stanford Ply** (.ply)
- **TrueSpace** (.cob, .scn)

See the _ai_importer_notes for information, what a specific importer can do and what not. Note that although this paper claims to be the official documentation, README.md is usually the most up-to-date list of file formats supported by the library.

Assimp is independent of the Operating System by nature, providing a C++ interface for easy integration with game engines and a C interface to allow bindings to other programming languages. At the moment the library runs on any little-endian platform including **X86/Windows/Linux/Mac** and **X64/Windows/Linux/Mac**. Special attention was paid to keep the library as free as possible from dependencies.

Big endian systems such as PPC-Macs or PPC-Linux systems are not officially supported at the moment. However, most formats handle the required endian conversion correctly, so large parts of the library should work.

The assimp linker library and viewer application are provided under the BSD 3-clause license. This basically means that you are free to use it in open- or closed-source projects, for commercial or non-commercial purposes as you like as long as you retain the license information and take own responsibility for what you do with it. For details see the LICENSE file.

You can find test models for almost all formats in the <assimp_root>/test/models directory. Beware, they're *free*, but not all of them are **open-source**. If there's an accompagning **'<file>source.txt'** file don't forget to read it.

1.1 Installation

assimp can be used in two ways: linking against the pre-built libraries or building the library on your own. The former option is the easiest, but the assimp distribution contains pre-built libraries only for Visual C++ 2013, 2015 and 2017. For other compilers you'll have to build assimp for yourself. Which is hopefully as hassle-free as the other way, but needs a bit more work. Both ways are described at the @link install Installation page. @endlink If you want to use assimp on Ubuntu you can install it via the following command:

```
sudo apt-get install assimp
```

If you want to use the python-assimp-port just follow these instructions: PyAssimp Doc

1.2 Usage

When you're done integrating the library into your IDE / project, you can now start using it. There are two separate interfaces by which you can access the library: a C++ interface and a C interface using flat functions. While the former is easier to handle, the latter also forms a point where other programming languages can connect to. Up to the moment, though, there are no bindings for any other language provided. Have a look at the @link usage Usage page @endlink for a detailed explanation and code examples.

1.3 Data Structures

When the importer successfully completed its job, the imported data is returned in an aiScene structure. This is the root point from where you can access all the various data types that a scene/model file can possibly contain. The @link data Data Structures page @endlink describes how to interpret this data.

1.4 Extending the library

There are many 3d file formats in the world, and we're happy to support as many as possible. If you need support for a particular file format, why not implement it yourself and add it to the library? Writing importer plugins for assimp is considerably easy, as the whole postprocessing infrastructure is available and does much of the work for you. See the @link extend Extending the library @endlink page for more information.

1.5 Support & Feedback

If you have any questions/comments/suggestions/bug reports you're welcome to post them in our Github-Issue-Tracker. Alternatively there's a mailing list, assimp-discussions .

1.6 Using the pre-built libraries with Visual-Studio

If you develop at Visual Studio 2013, 2015, 2017 or 2019, you can simply use the pre-built linker libraries provided in the distribution. Extract all files to a place of your choice. A directory called "assimp" will be created there. Add the assimp/include path to your include paths (Menu->Extras->Options->Projects and Solutions->VC++

Directories->Include files) and the assimp/lib/<Compiler> path to your linker paths (Menu->Extras->Options->Projects and Solutions->VC++ Directories->Library files). This is necessary only once to setup all paths inside you IDE.

To use the library in your C++ project you can simply generate a project file via cmake. One way is to add the assimp-folder as a subdirectory via the cmake-command

```
addsubdiectory(assimp)
```

Now just add the assimp-dependency to your application:

```
TARGET_LINK_LIBRARIES(my_game assimp)
```

If done correctly you should now be able to compile, link, run and use the application.

1.7 Build on all platforms using vcpkg

You can download and install assimp using the [vcpkg](https://github.com/Microsoft/vcpkg/) dependency manager:

```
bash
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg integrate install
vcpkg install assimp
```

The assimp port in vcpkg is kept up to date by Microsoft team members and community contributors. If the version is out of date, please [create an issue or pull request](https://github.com/Microsoft/vcpkg) on the vcpkg repository.

1.8 Building the library from scratch

First you need to install cmake. Now just get the code from github or download the latest version from the webside. to build the library just open a command-prompt / bash, navigate into the repo-folder and run cmake via:

```
cmake CMakeLists.txt
```

A project-file of your default make-system (like gnu-make on linux or Visual-Studio on Windows) will be generated. Run the build and you are done. You can find the libs at assimp/lib and the dll's / so's at bin.

1.9 Windows DLL Build

The Assimp-package can be built as DLL. You just need to run the default cmake run.

1.10 Assimp static lib

The Assimp-package can be build as a static library as well. Do do so just set the configuration variable BUILD_SHARED_LIBS to off during the cmake run.

CHAPTER 2

Access by C++ class interface

The assimp library can be accessed by both a class or flat function interface. The C++ class interface is the preferred way of interaction: you create an instance of class Assimp::Importer, maybe adjust some settings of it and then call Assimp::Importer::ReadFile(). The class will read the files and process its data, handing back the imported data as a pointer to an aiScene to you. You can now extract the data you need from the file. The importer manages all the resources for itsself. If the importer is destroyed, all the data that was created/read by it will be destroyed, too. So the easiest way to use the Importer is to create an instance locally, use its results and then simply let it go out of scope.

C++ example:

```
#include <assimp/Importer.hpp>      // C++ importer interface
#include <assimp/scene.h>           // Output data structure
#include <assimp/postprocess.h>     // Post processing flags

bool DoTheImportThing( const std::string& pFile) {
  // Create an instance of the Importer class
  Assimp::Importer importer;

  // And have it read the given file with some example postprocessing
  // Usually - if speed is not the most important aspect for you - you'll
  // probably to request more postprocessing than we do in this example.
  const aiScene* scene = importer.ReadFile( pFile,
    aiProcess_CalcTangentSpace       |
    aiProcess_Triangulate            |
    aiProcess_JoinIdenticalVertices  |
    aiProcess_SortByPType);

  // If the import failed, report it
  if( !scene) {
    DoTheErrorLogging( importer.GetErrorString());
    return false;
  }

  // Now we can access the file's contents.
  DoTheSceneProcessing( scene);
```

(continues on next page)

(continued from previous page)

```
  // We're done. Everything will be cleaned up by the importer destructor
  return true;
}
```

What exactly is read from the files and how you interpret it is described at the @ref data page. @endlink The post processing steps that the assimp library can apply to the imported data are listed at #aiPostProcessSteps. See the @ref pp Post processing page for more details.

Note that the aiScene data structure returned is declared 'const'. Yes, you can get rid of these 5 letters with a simple cast. Yes, you may do that. No, it's not recommended (and it's suicide in DLL builds if you try to use new or delete on any of the arrays in the scene).

2.1 Access by plain-c function interface

The plain function interface is just as simple, but requires you to manually call the clean-up after you're done with the imported data. To start the import process, call aiImportFile() with the filename in question and the desired postprocessing flags like above. If the call is successful, an aiScene pointer with the imported data is handed back to you. When you're done with the extraction of the data you're interested in, call aiReleaseImport() on the imported scene to clean up all resources associated with the import.

C-Example:

```
#include <assimp/cimport.h>        // Plain-C interface
#include <assimp/scene.h>          // Output data structure
#include <assimp/postprocess.h>    // Post processing flags

bool DoTheImportThing( const char* pFile) {
  // Start the import on the given file with some example postprocessing
  // Usually - if speed is not the most important aspect for you - you'll t
  // probably to request more postprocessing than we do in this example.
  const aiScene* scene = aiImportFile( pFile,
    aiProcess_CalcTangentSpace       |
    aiProcess_Triangulate            |
    aiProcess_JoinIdenticalVertices  |
    aiProcess_SortByPType);

  // If the import failed, report it
  if( nullptr != scene) {
    DoTheErrorLogging( aiGetErrorString());
    return false;
  }

  // Now we can access the file's contents
  DoTheSceneProcessing( scene);

  // We're done. Release all resources associated with this import
  aiReleaseImport( scene);
  return true;
}
```

2.2 Using custom IO logic with the C++ class interface

The assimp library needs to access files internally. This of course applies to the file you want to read, but also to additional files in the same folder for certain file formats. By default, standard C/C++ IO logic is used to access these files. If your application works in a special environment where custom logic is needed to access the specified files, you have to supply custom implementations of IOStream and IOSystem. A shortened example might look like this:

```
#include <assimp/IOStream.hpp>
#include <assimp/IOSystem.hpp>

// My own implementation of IOStream
class MyIOStream : public Assimp::IOStream {
  friend class MyIOSystem;

protected:
  // Constructor protected for private usage by MyIOSystem
  MyIOStream();

public:
  ~MyIOStream();
  size_t Read( void* pvBuffer, size_t pSize, size_t pCount) { ... }
  size_t Write( const void* pvBuffer, size_t pSize, size_t pCount) { ... }
  aiReturn Seek( size_t pOffset, aiOrigin pOrigin) { ... }
  size_t Tell() const { ... }
  size_t FileSize() const { ... }
  void Flush () { ... }
};

// Fisher Price - My First Filesystem
class MyIOSystem : public Assimp::IOSystem {
  MyIOSystem() { ... }
  ~MyIOSystem() { ... }

  // Check whether a specific file exists
  bool Exists( const std::string& pFile) const {
    ..
  }

  // Get the path delimiter character we'd like to see
  char GetOsSeparator() const {
    return '/';
  }

  // ... and finally a method to open a custom stream
  IOStream* Open( const std::string& pFile, const std::string& pMode) {
    return new MyIOStream( ... );
  }

  void Close( IOStream* pFile) { delete pFile; }
};
```

Now that your IO system is implemented, supply an instance of it to the Importer object by calling

```
Assimp::Importer::SetIOHandler().
```

An example:

```
void DoTheImportThing( const std::string& pFile) {
  Assimp::Importer importer;
  // put my custom IO handling in place
  importer.SetIOHandler( new MyIOSystem());

  // the import process will now use this implementation to access any file
  importer.ReadFile( pFile, SomeFlag | SomeOtherFlag);
}
```

2.3 Using custom IO logic with the plain-c function interface

The C interface also provides a way to override the file system. Control is not as fine-grained as for C++ although surely enough for almost any purpose. The process is simple:

- Include cfileio.h
- Fill an aiFileIO structure with custom file system callbacks (they're self-explanatory as they work similar to the CRT's fXXX functions)
-

2.4 Logging

The assimp library provides an easy mechanism to log messages. For instance if you want to check the state of your import and you just want to see, after which preprocessing step the import-process was aborted you can take a look into the log. Per default the assimp-library provides a default log implementation, where you can log your user specific message by calling it as a singleton with the requested logging-type. To see how this works take a look to this:

```
using namespace Assimp;

// Create a logger instance
DefaultLogger::create("", Logger::VERBOSE);

// Now I am ready for logging my stuff
DefaultLogger::get()->info("this is my info-call");

// Kill it after the work is done
DefaultLogger::kill();
```

At first you have to create the default-logger-instance (create). Now you are ready to rock and can log a little bit around. After that you should kill it to release the singleton instance.

If you want to integrate the assimp-log into your own GUI it my be helpful to have a mechanism writing the logs into your own log windows. The logger interface provides this by implementing an interface called LogStream. You can attach and detach this log stream to the default-logger instance or any implementation derived from Logger. Just derivate your own logger from the abstract base class LogStream and overwrite the write-method:

```
// Example stream
class myStream : public LogStream {
public:
    // Write womethink using your own functionality
    void write(const char* message) {
        ::printf("%s\n", message);
```

(continues on next page)

(continued from previous page)

```
    }
};

// Select the kinds of messages you want to receive on this log stream
const unsigned int severity = Logger::Debugging|Logger::Info|Logger::Err|Logger::Warn;

// Attaching it to the default logger
Assimp::DefaultLogger::get()->attachStream( new myStream, severity );
```

The severity level controls the kind of message which will be written into the attached stream. If you just want to log errors and warnings set the warn and error severity flag for those severities. It is also possible to remove a self defined logstream from an error severity by detaching it with the severity flag set:

```
unsigned int severity = 0;
severity |= Logger::Debugging;

// Detach debug messages from you self defined stream
Assimp::DefaultLogger::get()->attachStream( new myStream, severity );
```

If you want to implement your own logger just derive from the abstract base class #Logger and overwrite the methods debug, info, warn and error.

If you want to see the debug-messages in a debug-configured build, the Logger-interface provides a logging-severity. You can set it calling the following method:

```
Assimp::DefaultLogger::get()->setLogSeverity( LogSeverity log_severity );
```

The normal logging severity supports just the basic stuff like, info, warnings and errors. In the verbose level very fine-grained debug messages will be logged, too. Note that this kind kind of logging might decrease import performance.

2.5 Data Structures

The assimp library returns the imported data in a collection of structures. aiScene forms the root of the data, from here you gain access to all the nodes, meshes, materials, animations or textures that were read from the imported file. The aiScene is returned from a successful call to assimp::Importer::ReadFile(), aiImportFile() or aiImportFileEx() - see the @link usage Usage page @endlink for further information on how to use the library.

By default, all 3D data is provided in a right-handed coordinate system such as OpenGL uses. In this coordinate system, +X points to the right, +Y points upwards and +Z points out of the screen towards the viewer. Several modeling packages such as 3D Studio Max use this coordinate system as well (or a rotated variant of it). By contrast, some other environments use left-handed coordinate systems, a prominent example being DirectX. If you need the imported data to be in a left-handed coordinate system, supply the #aiProcess_MakeLeftHanded flag to the ReadFile() function call.

The output face winding is counter clockwise. Use #aiProcess_FlipWindingOrder to get CW data.

```
x2

            x1
    x0
```

Outputted polygons can be literally everything: they're probably concave, self-intersecting or non-planar, although our built-in triangulation (#aiProcess_Triangulate postprocessing step) doesn't handle the two latter.

The output UV coordinate system has its origin in the lower-left corner:

```
0x|1y ---------- 1x|1y
 |                |
 |                |
 |                |
0x|0y ---------- 1x|0y
```

Use the #aiProcess_FlipUVs flag to get UV coordinates with the upper-left corner als origin.

A typical 4x4 matrix including a translational part looks like this:

```
X1  Y1  Z1  T1
X2  Y2  Z2  T2
X3  Y3  Z3  T3
 0   0   0   1
```

with <tt>(X1, X2, X3)</tt> being the local X base vector, <tt>(Y1, Y2, Y3)</tt> being the local Y base vector, <tt>(Z1, Z2, Z3)</tt> being the local Z base vector and <tt>(T1, T2, T3)</tt> being the offset of the local origin (the translational part). All matrices in the library use row-major storage order. That means that the matrix elements are stored row-by-row, i.e. they end up like this in memory: <tt>[X1, Y1, Z1, T1, X2, Y2, Z2, T2, X3, Y3, Z3, T3, 0, 0, 0, 1]</tt>.

Note that this is neither the OpenGL format nor the DirectX format, because both of them specify the matrix layout such that the translational part occupies three consecutive addresses in memory (so those matrices end with <tt>[…, T1, T2, T3, 1]</tt>), whereas the translation in an Assimp matrix is found at the offsets 3, 7 and 11 (spread across the matrix). You can transpose an Assimp matrix to end up with the format that OpenGL and DirectX mandate. To be very precise: The transposition has nothing to do with a left-handed or right-handed coordinate system but 'converts' between row-major and column-major storage format.

11.24.09: We changed the orientation of our quaternions to the most common convention to avoid confusion. However, if you're a previous user of Assimp and you update the library to revisions beyond SVNREV 502, you have to adapt your animation loading code to match the new quaternion orientation.

2.6 The Node-Hierarchy

Nodes are little named entities in the scene that have a place and orientation relative to their parents. Starting from the scene's root node all nodes can have 0 to x child nodes, thus forming a hierarchy. They form the base on which the scene is built on: a node can refer to 0..x meshes, can be referred to by a bone of a mesh or can be animated by a key sequence of an animation. DirectX calls them "frames", others call them "objects", we call them aiNode.

A node can potentially refer to single or multiple meshes. The meshes are not stored inside the node, but instead in an array of aiMesh inside the aiScene. A node only refers to them by their array index. This also means that multiple nodes can refer to the same mesh, which provides a simple form of instancing. A mesh referred to by this way lives in the node's local coordinate system. If you want the mesh's orientation in global space, you'd have to concatenate the transformations from the referring node and all of its parents.

Most of the file formats don't really support complex scenes, though, but a single model only. But there are more complex formats such as .3ds, .x or .collada scenes which may contain an arbitrary complex hierarchy of nodes and meshes. I for myself would suggest a recursive filter function such as the following pseudocode:

```
void CopyNodesWithMeshes( aiNode node, SceneObject targetParent, Matrix4x4␣
↪accTransform) {
  SceneObject parent;
  Matrix4x4 transform;

  // if node has meshes, create a new scene object for it
```

(continues on next page)

(continued from previous page)

```
  if( node.mNumMeshes > 0) {
        SceneObjekt newObject = new SceneObject;
        targetParent.addChild( newObject);
        // copy the meshes
        CopyMeshes( node, newObject);

        // the new object is the parent for all child nodes
        parent = newObject;
        transform.SetUnity();
  } else {
        // if no meshes, skip the node, but keep its transformation
        parent = targetParent;
        transform = node.mTransformation * accTransform;
  }

  // continue for all child nodes
  for( all node.mChildren) {
        CopyNodesWithMeshes( node.mChildren[a], parent, transform);
  }
}
```

This function copies a node into the scene graph if it has children. If yes, a new scene object is created for the import node and the node's meshes are copied over. If not, no object is created. Potential child objects will be added to the old targetParent, but there transformation will be correct in respect to the global space. This function also works great in filtering the bone nodes - nodes that form the bone hierarchy for another mesh/node, but don't have any mesh themselves.

2.7 Meshes

All meshes of an imported scene are stored in an array of aiMesh* inside the aiScene. Nodes refer to them by their index in the array and providing the coordinate system for them, too. One mesh uses only a single material everywhere - if parts of the model use a different material, this part is moved to a separate mesh at the same node. The mesh refers to its material in the same way as the node refers to its meshes: materials are stored in an array inside aiScene, the mesh stores only an index into this array.

An aiMesh is defined by a series of data channels. The presence of these data channels is defined by the contents of the imported file: by default there are only those data channels present in the mesh that were also found in the file. The only channels guaranteed to be always present are aiMesh::mVertices and aiMesh::mFaces. You can test for the presence of other data by testing the pointers against NULL or use the helper functions provided by aiMesh. You may also specify several post processing flags at Importer::ReadFile() to let assimp calculate or recalculate additional data channels for you.

At the moment, a single aiMesh may contain a set of triangles and polygons. A single vertex does always have a position. In addition it may have one normal, one tangent and bitangent, zero to AI_MAX_NUMBER_OF_TEXTURECOORDS (4 at the moment) texture coords and zero to AI_MAX_NUMBER_OF_COLOR_SETS (4) vertex colors. In addition a mesh may or may not have a set of bones described by an array of aiBone structures. How to interpret the bone information is described later on.

2.8 Materials

See the @link materials Material System Page. @endlink

2.9 Bones

A mesh may have a set of bones in the form of aiBone objects. Bones are a means to deform a mesh according to the movement of a skeleton. Each bone has a name and a set of vertices on which it has influence. Its offset matrix declares the transformation needed to transform from mesh space to the local space of this bone.

Using the bones name you can find the corresponding node in the node hierarchy. This node in relation to the other bones' nodes defines the skeleton of the mesh. Unfortunately there might also be nodes which are not used by a bone in the mesh, but still affect the pose of the skeleton because they have child nodes which are bones. So when creating the skeleton hierarchy for a mesh I suggest the following method:

a) Create a map or a similar container to store which nodes are necessary for the skeleton. Pre-initialise it for all nodes with a "no".
 b) For each bone in the mesh: <br> b1) Find the corresponding node in the scene's hierarchy by comparing their names.
 b2) Mark this node as "yes" in the necessityMap.
 b3) Mark all of its parents the same way until you 1) find the mesh's node or 2) the parent of the mesh's node.
 c) Recursively iterate over the node hierarchy <br> c1) If the node is marked as necessary, copy it into the skeleton and check its children <br> c2) If the node is marked as not necessary, skip it and do not iterate over its children.

Reasons: you need all the parent nodes to keep the transformation chain intact. For most file formats and modelling packages the node hierarchy of the skeleton is either a child of the mesh node or a sibling of the mesh node but this is by no means a requirement so you shouldn't rely on it. The node closest to the root node is your skeleton root, from there you start copying the hierarchy. You can skip every branch without a node being a bone in the mesh - that's why the algorithm skips the whole branch if the node is marked as "not necessary".

You should now have a mesh in your engine with a skeleton that is a subset of the imported hierarchy.

2.10 Animations

An imported scene may contain zero to x aiAnimation entries. An animation in this context is a set of keyframe sequences where each sequence describes the orientation of a single node in the hierarchy over a limited time span. Animations of this kind are usually used to animate the skeleton of a skinned mesh, but there are other uses as well.

An aiAnimation has a duration. The duration as well as all time stamps are given in ticks. To get the correct timing, all time stamp thus have to be divided by aiAnimation::mTicksPerSecond. Beware, though, that certain combinations of file format and exporter don't always store this information in the exported file. In this case, mTicksPerSecond is set to 0 to indicate the lack of knowledge.

The aiAnimation consists of a series of aiNodeAnim's. Each bone animation affects a single node in the node hierarchy only, the name specifying which node is affected. For this node the structure stores three separate key sequences: a vector key sequence for the position, a quaternion key sequence for the rotation and another vector key sequence for the scaling. All 3d data is local to the coordinate space of the node's parent, that means in the same space as the node's transformation matrix. There might be cases where animation tracks refer to a non-existent node by their name, but this should not be the case in your every-day data.

To apply such an animation you need to identify the animation tracks that refer to actual bones in your mesh. Then for every track: * Find the keys that lay right before the current anim time. * Optional: interpolate between these and the following keys. * Combine the calculated position, rotation and scaling to a transformation matrix * Set the affected node's transformation to the calculated matrix.

If you need hints on how to convert to or from quaternions, have a look at the Matrix & Quaternion FAQ. I suggest using logarithmic interpolation for the scaling keys if you happen to need them - usually you don't need them at all.

2.11 Blenshapes

ToDo!

2.12 Textures

Normally textures used by assets are stored in separate files, however, there are file formats embedding their textures directly into the model file. Such textures are loaded into an aiTexture structure.

In previous versions, the path from the query for *AI_MATKEY_TEXTURE(textureType, index)* would be *<*index*> where <*index*> is the index of the texture in aiScene::mTextures. Now this call will return a file path for embedded textures in FBX files. To test if it is an embedded texture use aiScene::GetEmbeddedTexture. If the returned pointer is not null, it is embedded und can be loaded from the data structure. If it is null, search for a separate file. Other file types still use the old behaviour.
 If your rely on the old behaviour, you can use Assimp::Importer::SetPropertyBool with the key #AI_CONFIG_IMPORT_FBX_EMBEDDED_TEXTURES_LEGACY_NAMING to force the old behaviour.

There are two cases:

- The texture is NOT compressed. Its color data is directly stored in the aiTexture structure as an array of aiTexture::mWidth * aiTexture::mHeight aiTexel structures. Each aiTexel represents a pixel (or "texel") of the texture image. The color data is stored in an unsigned RGBA8888 format, which can be easily used for both Direct3D and OpenGL (swizzling the order of the color components might be necessary). RGBA8888 has been chosen because it is well-known, easy to use and natively supported by nearly all graphics APIs.
- This applies if aiTexture::mHeight == 0 is fulfilled. Then, texture is stored in a compressed format such as DDS or PNG. The term "compressed" does not mean that the texture data must actually be compressed, however the texture was found in the model file as if it was stored in a separate file on the harddisk. Appropriate decoders (such as libjpeg, libpng, D3DX, DevIL) are required to load theses textures. aiTexture::mWidth specifies the size of the texture data in bytes, aiTexture::pcData is a pointer to the raw image data and aiTexture::achFormatHint is either zeroed or contains the most common file extension of the embedded texture's format. This value is only set if assimp is able to determine the file format.

2.13 Material-System

2.13.1 General Overview

All materials are stored in an array of aiMaterial inside the aiScene.

Each aiMesh refers to one material by its index in the array. Due to the vastly diverging definitions and usages of material parameters there is no hard definition of a material structure. Instead a material is defined by a set of properties accessible by their names. Have a look at assimp/material.h to see what types of properties are defined. In this file there are also various functions defined to test for the presence of certain properties in a material and retrieve their values.

2.13.2 Textures

Textures are organized in stacks, each stack being evaluated independently. The final color value from a particular texture stack is used in the shading equation. For example, the computed color value of the diffuse texture stack (aiTextureType_DIFFUSE) is multiplied with the amount of incoming diffuse light to obtain the final diffuse color of a pixel.

Stack	Resulting equation
Constant base color	color
Blend operation 0	•
Strength factor 0	0.25*
Texture 0	texture_0
Blend operation 1	x
Strength factor 1	1.0*
Texture 1	texture_1

2.13.3 Constants

All material key constants start with 'AI_MATKEY' as a prefix.

Name	Data Type	Default Value	Meaning	Notes
NAME	aiString	n/a	The name of the material, if available.	Ignored by <tt>aiProcess_RemoveRedundantMaterials. Materials are considered equal even if their names are different.
COLOR_DIFFUSE	aiColor3D	black (0,0,0)	Diffuse color of the material. This is typically scaled by the amount of incoming diffuse light (e.g. using gouraud shading)	n/a
COLOR_SPECULAR	aiColor3D	black (0,0,0)	Specular color of the material. This is typically scaled by the amount of incoming specular light (e.g. using phong shading)	n/a
COLOR_AMBIENT	aiColor3D	black (0,0,0)	Ambient color of the material. This is typically scaled by the amount of ambient light	n/a
COLOR_EMISSIVE	aiColor3D	black (0,0,0)	Emissive color of the material. This is the amount of light emitted by the object. In real time applications it will usually not affect surrounding objects, but raytracing applications may wish to treat emissive objects as light sources.	n/a
COLOR_TRANSPARENT	aiColor3D	black (0,0,0)	Defines the transparent color of the material, this is the color to be multiplied with the color of translucent light to construct the final 'destination color' for a particular position in the screen buffer.	n/a
COLOR_REFLECTIVE	aiColor3D	<tt>black (0,0,0)	Defines the reflective color of the material. This is typically scaled by the amount of incoming light from the direction of mirror reflection. Usually combined with an environment lightmap of some kind for real-time applications.	n/a
REFLECTIVITY	float	0.0	Scales the reflective color of the material.	n/a
WIREFRAME	int	false	Specifies whether wireframe rendering must be turned on for the material. 0 for false, !0 for true.	n/a
TWOSIDED	int	false	Specifies whether meshes using this material must be rendered without backface culling. 0 for false, !0 for true.	Some importers set this property if they don't know whether the output face order is right. As long as it is not set, you may safely enable backface culling.
SHADING_MODEL	int	gouraud	One of the #aiShadingMode enumerated values. Defines the library shading model to use for (real time) rendering to approximate the original look of the material as closely as possible.	The presence of this key might indicate a more complex material. If absent, assume phong shading only if a specular exponent is given.
BLEND_FUNC	int	false	One of the #aiBlendMode enumerated values. Defines how the final color value in the screen buffer is computed from the given color at that position and the newly computed color from the material. Simply said, alpha blending settings.	n/a
OPACITY	float	1.0	Defines the opacity of the material in a range between 0..1.	Use this value to decide whether you have to activate alpha blending for rendering. <tt>OPACITY</tt> != 1 usually also implies TWOSIDED=1 to avoid cull artifacts.
SHININESS	float	0.f	Defines the shininess of a phong-shaded material. This is actually the exponent of the phong specular equation	SHININESS</tt>=0 is equivalent to <tt>SHADING_MODEL</tt>=<tt>aiShadingM
SHININESS_STRENGTH	float	1.0	Scales the specular color of the material.	This value is kept separate from the specular color by

2.13.4 C++-API

Retrieving a property from a material is done using various utility functions. For C++ it's simply calling aiMaterial::Get()

```
aiMaterial* mat = .....

// The generic way
if(AI_SUCCESS != mat->Get(<material-key>,<where-to-store>)) {
   // handle epic failure here
}
```

Simple, isn't it? To get the name of a material you would use

```
aiString name;
mat->Get(AI_MATKEY_NAME,name);
```

Or for the diffuse color ('color' won't be modified if the property is not set)

```
aiColor3D color (0.f,0.f,0.f);
mat->Get(AI_MATKEY_COLOR_DIFFUSE,color);
```

Note: Get() is actually a template with explicit specializations for aiColor3D, aiColor4D, aiString, float, int and some others. Make sure that the type of the second parameter matches the expected data type of the material property (no compile-time check yet!). Don't follow this advice if you wish to encounter very strange results.

2.13.5 C-API

For good old C it's slightly different. Take a look at the aiGetMaterialGet<data-type> functions.

```
aiMaterial* mat = .....

if(AI_SUCCESS != aiGetMaterialFloat(mat,<material-key>,<where-to-store>)) {
   // handle epic failure here
}
```

To get the name of a material you would use

```
aiString name;
aiGetMaterialString(mat,AI_MATKEY_NAME,&name);
```

Or for the diffuse color ('color' won't be modified if the property is not set)

```
aiColor3D color (0.f,0.f,0.f);
aiGetMaterialColor(mat,AI_MATKEY_COLOR_DIFFUSE,&color);
```

2.14 How to map UV channels to textures (MATKEY_UVWSRC)

The MATKEY_UVWSRC property is only present if the source format doesn't specify an explicit mapping from textures to UV channels. Many formats don't do this and assimp is not aware of a perfect rule either.

Your handling of UV channels needs to be flexible therefore. Our recommendation is to use logic like this to handle most cases properly:

::

have only one uv channel? assign channel 0 to all textures and break

for all textures

have uvwsrc for this texture? assign channel specified in uvwsrc

else assign channels in ascending order for all texture stacks, i.e. diffuse1 gets channel 1, opacity0 gets channel 0.

2.14.1 Pseudo Code Listing

For completeness, the following is a very rough pseudo-code sample showing how to evaluate Assimp materials in your shading pipeline. You'll probably want to limit your handling of all those material keys to a reasonable subset suitable for your purposes (for example most 3d engines won't support highly complex multi-layer materials, but many 3d modellers do).

Also note that this sample is targeted at a (shader-based) rendering pipeline for real time graphics.

```
// ---------------------------------------------------------------------------------
↪----
// Evaluate multiple textures stacked on top of each other
float3 EvaluateStack(stack) {
  // For the 'diffuse' stack stack.base_color would be COLOR_DIFFUSE
  // and TEXTURE(aiTextureType_DIFFUSE,n) the n'th texture.

  float3 base = stack.base_color;
  for (every texture in stack) {
    // assuming we have explicit & pretransformed UVs for this texture
    float3 color = SampleTexture(texture,uv);

    // scale by texture blend factor
    color *= texture.blend;

    if (texture.op == add)
      base += color;
    else if (texture.op == multiply)
      base *= color;
    else // other blend ops go here
  }
  return base;
}

// ---------------------------------------------------------------------------------
↪----
// Compute the diffuse contribution for a pixel
float3 ComputeDiffuseContribution() {
  if (shading == none)
     return float3(1,1,1);

  float3 intensity (0,0,0);
  for (all lights in range) {
    float fac = 1.f;
    if (shading == gouraud)
      fac =  lambert-term ..
    else // other shading modes go here

    // handling of different types of lights, such as point or spot lights
    // ...
```

(continues on next page)

(continued from previous page)

```
    // and finally sum the contribution of this single light ...
    intensity += light.diffuse_color * fac;
  }
  // ... and combine the final incoming light with the diffuse color
  return EvaluateStack(diffuse) * intensity;
}

// ---------------------------------------------------------------------------------
↪----
// Compute the specular contribution for a pixel
float3 ComputeSpecularContribution() {
  if (shading == gouraud || specular_strength == 0 || specular_exponent == 0)
    return float3(0,0,0);

  float3 intensity (0,0,0);
  for (all lights in range) {
    float fac = 1.f;
    if (shading == phong)
      fac =  phong-term ..
    else // other specular shading modes go here

    // handling of different types of lights, such as point or spot lights
    // ...

    // and finally sum the specular contribution of this single light ...
    intensity += light.specular_color * fac;
  }
  // ... and combine the final specular light with the specular color
  return EvaluateStack(specular) * intensity * specular_strength;
}

// ---------------------------------------------------------------------------------
↪----
// Compute the ambient contribution for a pixel
float3 ComputeAmbientContribution() {
  if (shading == none)
     return float3(0,0,0);

  float3 intensity (0,0,0);
  for (all lights in range) {
    float fac = 1.f;

    // handling of different types of lights, such as point or spot lights
    // ...

    // and finally sum the ambient contribution of this single light ...
    intensity += light.ambient_color * fac;
  }
  // ... and combine the final ambient light with the ambient color
  return EvaluateStack(ambient) * intensity;
}

// ---------------------------------------------------------------------------------
↪----
// Compute the final color value for a pixel
// @param prev Previous color at that position in the framebuffer
```

(continues on next page)

(continued from previous page)

```
float4 PimpMyPixel (float4 prev) {
  // .. handle displacement mapping per vertex
  // .. handle bump/normal mapping

  // Get all single light contribution terms
  float3 diff = ComputeDiffuseContribution();
  float3 spec = ComputeSpecularContribution();
  float3 ambi = ComputeAmbientContribution();

  // .. and compute the final color value for this pixel
  float3 color = diff + spec + ambi;
  float3 opac  = EvaluateStack(opacity);

  // note the *slightly* strange meaning of additive and multiplicative blending here␣
↪...
  // those names will most likely be changed in future versions
  if (blend_func == add)
       return prev+color*opac;
  else if (blend_func == multiply)
       return prev*(1.0-opac)+prev*opac;

   return color;
}
```

2.14.2 How to access shader-code from a texture (AI_MATKEY_GLOBAL_SHADERLANG and AI_MATKEY_SHADER_VERTEX, . . .)

You can get assigned shader sources by using the following material keys:

- *AI_MATKEY_GLOBAL_SHADERLANG* To get the used shader language.
- *AI_MATKEY_SHADER_VERTEX* Assigned vertex shader code stored as a string.
- *AI_MATKEY_SHADER_FRAGMENT* Assigned fragment shader code stored as a string.
- *AI_MATKEY_SHADER_GEO* Assigned geometry shader code stored as a string.
- *AI_MATKEY_SHADER_TESSELATION* Assigned tesselation shader code stored as a string.
- *AI_MATKEY_SHADER_PRIMITIVE* Assigned primitive shader code stored as a string.
- *AI_MATKEY_SHADER_COMPUTE* Assigned compute shader code stored as a string.

2.15 Performance

2.16 Overview

This page discusses general performance issues related to assimp.

2.17 Profiling

Assimp has built-in support for <i>very</i> basic profiling and time measurement. To turn it on, set the <tt>GLOB_MEASURE_TIME</tt> configuration switch to <tt>true</tt> (nonzero). Results are dumped to the log file, so you need to setup an appropriate logger implementation with at least one output stream first (see the @link logging Logging Page @endlink for the details.).

Note that these measurements are based on a single run of the importer and each of the post processing steps, so a single result set is far away from being significant in a statistic sense. While precision can be improved by running the test multiple times, the low accuracy of the timings may render the results useless for smaller files.

A sample report looks like this (some unrelated log messages omitted, entries grouped for clarity):

```
Debug, T5488: START `total`
Info,  T5488: Found a matching importer for this file format


Debug, T5488: START `import`
Info,  T5488: BlendModifier: Applied the `Subdivision` modifier to `OBMonkey`
Debug, T5488: END   `import`, dt= 3.516 s


Debug, T5488: START `preprocess`
Debug, T5488: END   `preprocess`, dt= 0.001 s
Info,  T5488: Entering post processing pipeline


Debug, T5488: START `postprocess`
Debug, T5488: RemoveRedundantMatsProcess begin
Debug, T5488: RemoveRedundantMatsProcess finished
Debug, T5488: END   `postprocess`, dt= 0.001 s


Debug, T5488: START `postprocess`
Debug, T5488: TriangulateProcess begin
Info,  T5488: TriangulateProcess finished. All polygons have been triangulated.
Debug, T5488: END   `postprocess`, dt= 3.415 s


Debug, T5488: START `postprocess`
Debug, T5488: SortByPTypeProcess begin
Info,  T5488: Points: 0, Lines: 0, Triangles: 1, Polygons: 0 (Meshes, X = removed)
Debug, T5488: SortByPTypeProcess finished

Debug, T5488: START `postprocess`
Debug, T5488: JoinVerticesProcess begin
Debug, T5488: Mesh 0 (unnamed) | Verts in: 503808 out: 126345 | ~74.922
Info,  T5488: JoinVerticesProcess finished | Verts in: 503808 out: 126345 | ~74.9
Debug, T5488: END   `postprocess`, dt= 2.052 s

Debug, T5488: START `postprocess`
Debug, T5488: FlipWindingOrderProcess begin
Debug, T5488: FlipWindingOrderProcess finished
Debug, T5488: END   `postprocess`, dt= 0.006 s


Debug, T5488: START `postprocess`
Debug, T5488: LimitBoneWeightsProcess begin
Debug, T5488: LimitBoneWeightsProcess end
Debug, T5488: END   `postprocess`, dt= 0.001 s
```

(continues on next page)

(continued from previous page)

```
Debug, T5488: START `postprocess`
Debug, T5488: ImproveCacheLocalityProcess begin
Debug, T5488: Mesh 0 | ACMR in: 0.851622 out: 0.718139 | ~15.7
Info,  T5488: Cache relevant are 1 meshes (251904 faces). Average output ACMR is 0.
↪718139
Debug, T5488: ImproveCacheLocalityProcess finished.
Debug, T5488: END   `postprocess`, dt= 1.903 s


Info,  T5488: Leaving post processing pipeline
Debug, T5488: END   `total`, dt= 11.269 s
```

In this particular example only one fourth of the total import time was spent on the actual importing, while the rest of the time got consumed by the #aiProcess_Triangulate, #aiProcess_JoinIdenticalVertices and #aiProcess_ImproveCacheLocality postprocessing steps. A wise selection of postprocessing steps is therefore essential to getting good performance. Of course this depends on the individual requirements of your application, in many of the typical use cases of assimp performance won't matter (i.e. in an offline content pipeline).

.._ai_threading:

2.18 Threading

2.19 Overview

This page discusses both assimps scalability in threaded environments and the precautions to be taken in order to use it from multiple threads concurrently.

2.20 Thread-safety / using Assimp concurrently from several threads

The library can be accessed by multiple threads simultaneously, as long as the following prerequisites are fulfilled:

- Users of the C++-API should ensure that they use a dedicated #Assimp::Importer instance for each thread. Constructing instances of #Assimp::Importer is expensive, so it might be a good idea to let every thread maintain its own thread-local instance (which can be used to load as many files as necessary).
- The C-API is thread safe.
- When supplying custom IO logic, one must make sure the underlying implementation is thread-safe.
- Custom log streams or logger replacements have to be thread-safe, too.

Multiple concurrent imports may or may not be beneficial, however. For certain file formats in conjunction with little or no post processing IO times tend to be the performance bottleneck. Intense post processing together with 'slow' file formats like X or Collada might scale well with multiple concurrent imports.

2.21 Internal threading

Internal multi-threading is not currently implemented.

2.22 Resources

This page lists some useful resources for assimp. Note that, even though the core team has an eye on them, we cannot guarantee the accuracy of third-party information. If in doubt, it's best to ask either on the mailing list or on our forums on SF.net.

- assimp comes with some sample applications, these can be found in the <i>./samples</i> folder. Don't forget to read the <i>README</i> file.
- Assimp-GL-Demo - OpenGl animation sample using the library's animation import facilities.
- Assimp-Animation-Loader is another utility to simplify animation playback.
- Assimp-Animations - Tutorial "Loading models using the Open Asset Import Library", out of a series of OpenGl tutorials.

CHAPTER 3

Importer Notes

3.1 Blender

This section contains implementation notes for the Blender3D importer.

3.2 Overview

assimp provides a self-contained reimplementation of Blender's so called SDNA system (**'Notes on SDNA http://www.blender.org/development/architecture/notes-on-sdna/'_**). SDNA allows Blender to be fully backward and forward compatible and to exchange files across all platforms. The BLEND format is thus a non-trivial binary monster and the loader tries to read the most of it, naturally limited by the scope of the #aiScene output data structure. Consequently, if Blender is the only modeling tool in your asset work flow, consider writing a custom exporter from Blender if assimps format coverage does not meet the requirements.

3.3 Current status

The Blender loader does not support animations yet, but is apart from that considered relatively stable.

@subsection bl_notes Notes

When filing bugs on the Blender loader, always give the Blender version (or, even better, post the file caused the error).

3.4 IFC

This section contains implementation notes on the IFC-STEP importer.

3.5 Overview

The library provides a partial implementation of the IFC2x3 industry standard for automatized exchange of CAE/architectural data sets. See IFC for more information on the format. We aim at getting as much 3D data out of the files as possible.

3.6 Current status

IFC support is new and considered experimental. Please report any bugs you may encounter.

3.7 Notes

- Only the STEP-based encoding is supported. IFCZIP and IFCXML are not (but IFCZIP can simply be unzipped to get a STEP file).
- The importer leaves vertex coordinates untouched, but applies a global scaling to the root transform to convert from whichever unit the IFC file uses to <i>metres</i>.
- If multiple geometric representations are provided, the choice which one to load is based on how expensive a representation seems to be in terms of import time. The loader also avoids representation types for which it has known deficits.
- Not supported are arbitrary binary operations (binary clipping is implemented, though).
- Of the various relationship types that IFC knows, only aggregation, containment and material assignment are resolved and mapped to the output graph.
- The implementation knows only about IFC2X3 and applies this rule set to all models it encounters, regardless of their actual version. Loading of older or newer files may fail with parsing errors.

3.8 Metadata

IFC file properties (IfcPropertySet) are kept as per-node metadata, see aiNode::mMetaData.

3.9 Ogre

ATTENTION: The Ogre-Loader is currently under development, many things have changed after this documentation was written, but they are not final enough to rewrite the documentation. So things may have changed by now!

This section contains implementations notes for the OgreXML importer.

3.10 Overview

Ogre importer is currently optimized for the Blender Ogre exporter, because that's the only one that I use. You can find the Blender Ogre exporter at: OGRE3D forum

3.11 What will be loaded?

Mesh: Faces, Positions, Normals and all TexCoords. The Materialname will be used to load the material.

Material: The right material in the file will be searched, the importer should work with materials who have 1 technique and 1 pass in this technique. From there, the texturename (for 1 color- and 1 normalmap) and the materialcolors (but not in custom materials) will be loaded. Also, the materialname will be set.

Skeleton: Skeleton with Bone hierarchy (Position and Rotation, but no Scaling in the skeleton is supported), names and transformations, animations with rotation, translation and scaling keys.

3.12 How to export Files from Blender

You can find information about how to use the Ogreexporter by your own, so here are just some options that you need, so the assimp importer will load everything correctly:

- Use either “Rendering Material” or “Custom Material” see @ref material
- do not use “Flip Up Axies to Y”
- use “Skeleton name follow mesh”

3.13 XML-Format

There is a binary and a XML mesh Format from Ogre. This loader can only Handle xml files, but don’t panic, there is a command line converter, which you can use to create XML files from Binary Files. Just look on the Ogre page for it.

Currently you can only load meshes. So you will need to import the .mesh.xml file, the loader will try to find the appendant material and skeleton file.

The skeleton file must have the same name as the mesh file, e.g. fish.mesh.xml and fish.skeleton.xml.

@subsection material Materials The material file can have the same name as the mesh file (if the file is model.mesh or model.mesh.xml the loader will try to load model.material), or you can use

```
Importer::Importer::SetPropertyString(AI_CONFIG_IMPORT_OGRE_MATERIAL_FILE,
↪"materiafile.material")
```

to specify the name of the material file. This is especially useful if multiply materials a stored in a single file. The importer will first try to load the material with the same name as the mesh and only if this can’t be open try to load the alternate material file. The default material filename is “Scene.material”.

We suggest that you use custom materials, because they support multiple textures (like colormap and normalmap). First of all you should read the custom material sektion in the Ogre Blender exporter Help File, and than use the assimp.tlp template, which you can find in scripts/OgreImpoter/Assimp.tlp in the assimp source. If you don’t set all values, don’t worry, they will be ignored during import.

If you want more properties in custom materials, you can easily expand the ogre material loader, it will be just a few lines for each property. Just look in OgreImporterMaterial.cpp

3.14 Properties

- IMPORT_OGRE_TEXTURETYPE_FROM_FILENAME: Normally, a texture is loaded as a colormap, if no target is specified in the materialfile. Is this switch is enabled, texture names ending with _n, _l, _s are used as normalmaps, lightmaps or specularmaps.
 Property type: Bool. Default value: false.

- IMPORT_OGRE_MATERIAL_FILE: Ogre Meshes contain only the MaterialName, not the MaterialFile. If there is no material file with the same name as the material, Ogre Importer will try to load this file and search the material in it.
 Property type: String. Default value: guessed.

3.15 Extending the Library

3.16 General

Or - how to write your own loaders. It's easy. You just need to implement the #Assimp::BaseImporter class, which defines a few abstract methods, register your loader, test it carefully and provide test models for it.

OK, that sounds too easy :-). The whole procedure for a new loader merely looks like this:

 Create a header (<tt><i>FormatName</i>Importer.h</tt>) and a unit (<tt><i>FormatName</i>Importer.cpp</tt>) in the <tt><root>/code/</tt> directory Add them to the following workspaces: vc8 and vc9 (the files are in the workspaces directory), CMAKE (code/CMakeLists.txt, create a new source group for your importer and put them also to ADD_LIBRARY(assimp SHARED)) Include <i>AssimpPCH.h</i> - this is the PCH file, and it includes already most Assimp-internal stuff. Open Importer.cpp and include your header just below the <i>(include_new_importers_here)</i> line, guarded by a #define

```
#if (!defined assimp_BUILD_NO_FormatName_IMPORTER)
    ...
#endif
```

Wrap the same guard around your .cpp!

- Now advance to the <i>(register_new_importers_here)</i> line in the Importer.cpp and register your importer there - just like all the others do.
- Setup a suitable test environment (i.e. use AssimpView or your own application), make sure to enable the #aiProcess_ValidateDataStructure flag and enable verbose logging. That is, simply call before you import anything:

```
DefaultLogger::create("AssimpLog.txt",Logger::VERBOSE)
```

- Implement the Assimp::BaseImporter::CanRead(), Assimp::BaseImporter::InternReadFile() and Assimp::BaseImporter::GetExtensionList(). Just copy'n'paste the template from Appendix A and adapt it for your needs.
- For error handling, throw a dynamic allocated ImportErrorException (see Appendix A) for critical errors, and log errors, warnings, infos and debuginfos with DefaultLogger::get()->[error, warn, debug, info].
- Make sure that your loader compiles against all build configurations on all supported platforms. You can use our CI-build to check several platforms like Windows and Linux (32 bit and 64 bit).
- Provide some _free_ test models in <tt><root>/test/models/<FormatName>/</tt> and credit their authors. Test files for a file format shouldn't be too large (<i>~500 KiB in total</i>), and not too repetive. Try to cover all format features with test data.
- Done! Please, share your loader that everyone can profit from it!

3.17 Properties

You can use properties to chance the behavior of you importer. In order to do so, you have to override BaseImporter::SetupProperties, and specify you custom properties in config.h. Just have a look to the other AI_CONFIG_IMPORT_* defines and you will understand, how it works.

The properties can be set with Importer::SetProperty***() and can be accessed in your SetupProperties function with Importer::GetProperty***(). You can store the properties as a member variable of your importer, they are thread safe.

3.18 Notes for text importers

- Try to make your parser as flexible as possible. Don't rely on particular layout, whitespace/tab style, except if the file format has a strict definition, in which case you should always warn about spec violations. But the general rule of thumb is <i>be strict in what you write and tolerant in what you accept</i>.
- Call Assimp::BaseImporter::ConvertToUTF8() before you parse anything to convert foreign encodings to UTF-8. That's not necessary for XML importers, which must use the provided IrrXML for reading.

3.19 Notes for binary importers

- Take care of endianness issues! Assimp importers mostly support big-endian platforms, which define the <tt>AI_BUILD_BIG_ENDIAN</tt> constant. See the next section for a list of utilities to simplify this task.
- Don't trust the input data! Check all offsets!

3.20 Utilities

Mixed stuff for internal use by loaders, mostly documented (most of them are already included by <i>AssimpPCH.h</i>):

- **ByteSwapper** (*ByteSwapper.h*) - manual byte swapping stuff for binary loaders.
- **StreamReader** (*StreamReader.h*) - safe, endianess-correct, binary reading.
- **IrrXML** (*irrXMLWrapper.h*) - for XML-parsing (SAX.
- **CommentRemover** (*RemoveComments.h*) - remove single-line and multi-line comments from a text file.
- fast_atof, strtoul10, strtoul16, SkipSpaceAndLineEnd, SkipToNextToken .. large family of low-level parsing functions, mostly declared in <i>fast_atof.h</i>, <i>StringComparison.h</i> and <i>ParsingUtils.h</i> (a collection that grew historically, so don't expect perfect organization).
- **ComputeNormalsWithSmoothingsGroups()** (*SmoothingGroups.h*) - Computes normal vectors from plain old smoothing groups.
- **SkeletonMeshBuilder** (*SkeletonMeshBuilder.h*) - generate a dummy mesh from a given (animation) skeleton.
- **StandardShapes** (*StandardShapes.h*) - generate meshes for standard solids, such as platonic primitives, cylinders or spheres.
- **BatchLoader** (*BaseImporter.h*) - manage imports from external files. Useful for file formats which spread their data across multiple files.
- **SceneCombiner** (*SceneCombiner.h*) - exhaustive toolset to merge multiple scenes. Useful for file formats which spread their data across multiple files.

3.21 Filling materials

The required definitions zo set/remove/query keys in #aiMaterial structures are declared in <i>MaterialSystem.h</i>, in a #aiMaterial derivate called #aiMaterial. The header is included by AssimpPCH.h, so you don't need to bother.

```
aiMaterial* mat = new aiMaterial();

const float spec = 16.f;
mat->AddProperty(&spec, 1, AI_MATKEY_SHININESS);

//set the name of the material:
NewMaterial->AddProperty(&aiString(MaterialName.c_str()), AI_MATKEY_NAME);//
↪MaterialName is a std::string

//set the first diffuse texture
NewMaterial->AddProperty(&aiString(Texturename.c_str()), AI_MATKEY_
↪TEXTURE(aiTextureType_DIFFUSE, 0));//again, Texturename is a std::string
```

3.22 Appendix A - Template for BaseImporter's abstract methods

```
// -------------------------------------------------------------------------------
// Returns whether the class can handle the format of the given file.
bool xxxxImporter::CanRead( const std::string& pFile, IOSystem* pIOHandler,
      bool checkSig) const {
    const std::string extension = GetExtension(pFile);
    if(extension == "xxxx") {
        return true;
    }
    if (!extension.length() || checkSig) {
        // no extension given, or we're called a second time because no
        // suitable loader was found yet. This means, we're trying to open
        // the file and look for and hints to identify the file format.
        // #Assimp::BaseImporter provides some utilities:
        //
        // #Assimp::BaseImporter::SearchFileHeaderForToken - for text files.
        // It reads the first lines of the file and does a substring check
        // against a given list of 'magic' strings.
        //
        // #Assimp::BaseImporter::CheckMagicToken - for binary files. It goes
        // to a particular offset in the file and and compares the next words
        // against a given list of 'magic' tokens.

        // These checks MUST be done (even if !checkSig) if the file extension
        // is not exclusive to your format. For example, .xml is very common
        // and (co)used by many formats.
    }
    return false;
}

// -------------------------------------------------------------------------------
// Get list of file extensions handled by this loader
void xxxxImporter::GetExtensionList(std::set<std::string>& extensions) {
    extensions.insert("xxx");
}
```

(continues on next page)

(continued from previous page)

```
// -------------------------------------------------------------------------------
void xxxxImporter::InternReadFile( const std::string& pFile,
      aiScene* pScene, IOSystem* pIOHandler) {
    std::unique_ptr<IOStream> file( pIOHandler->Open( pFile, "rb"));

    // Check whether we can read from the file
    if( file.get() == NULL) {
        throw DeadlyImportError( "Failed to open xxxx file " + pFile + ".");
    }

    // Your task: fill pScene
    // Throw a ImportErrorException with a meaningful (!) error message if
    // something goes wrong.
}
```

3.23 Animation Overview

3.24 Transformations

This diagram shows how you can calculate your transformationmatrices for an animated character: