

Kicad Plugins

May 6, 2018

Contents

1 Introduction to the KiCad plugin system **2**

1.1 Plugin Classes . . . 2

1.1.1 Plugin Class: PLUGIN_3D . . . 3

2 Tutorials: 3D Plugin Class **3**

2.1 Basic 3D Plugin . . . 4

2.2 Advanced 3D Plugin . . . 9

3 Application Programming Interface (API) **14**

3.1 Plugin Class APIs . . . 15

3.1.1 API: Base Kicad Plugin Class . . . 15

3.1.2 API: 3D Plugin Class . . . 15

3.2 Scenegraph Class APIs . . . 16

KiCad Plugin System

Copyright

This document is Copyright © 2016 by it's contributors as listed below. You may distribute it and/or modify it under the terms of either the GNU General Public License (http://www.gnu.org/licenses/gpl.html), version 3 or later, or the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), version 3.0 or later.

All trademarks within this guide belong to their legitimate owners.

Contributors

Cirilo Bernardo

Feedback

Please direct any bug reports, suggestions or new versions to here:

- About KiCad document: https://github.com/KiCad/kicad-doc/issues
- About KiCad software: https://bugs.launchpad.net/kicad
- About KiCad software i18n: https://github.com/KiCad/kicad-i18n/issues

Publication date and software version

Published on January 29, 2016.

1 Introduction to the KiCad plugin system

The KiCad plugin system is a framework for extending the capabilities of KiCad using shared libraries. One of the main advantages of using a plugin is that it is not necessary to rebuild the KiCad suite while developing a plugin; in fact, plugins can be built with the aid of a very small set of headers from the KiCad source tree. Removing the requirement to build KiCad during plugin development greatly increases productivity by ensuring that the developer only compiles code directly related to the plugin which is being developed and thus reducing the time required for each build and test cycle.

Plugins were initially developed for the 3D model viewer to make it possible to support more types of 3D models without requiring major changes to the KiCad source for each new model type supported. The plugin framework was later generalized so that in the future developers can create different classes of plugins. Currently only 3D plugins are implemented within KiCad but it is envisioned that a PCB plugin will eventually be developed to make it possible for users to implement data Importers and Exporters.

1.1 Plugin Classes

Plugins are divided into Plugin Classes since each plugin addresses problems in a specific domain and therefore requires an interface unique to that domain. For example, the 3D model plugins load 3D model data from files and translate that data into a format which can be displayed by the 3D viewer. A PCB Import/Export plugin would take PCB data and export to other electrical or mechanical data formats, or translate a foreign format into a KiCad PCB. At the moment only the 3D Plugin Class has been developed and it will be the focus of this document.

Implementing a Plugin Class requires creating code within the KiCad source tree which manages the loading of plugin code. Within the KiCad source tree, the file `plugins/ldr/pluginldr.h` declares the base class for all plugin loaders. This class declares the most basic functions which we would expect to find in any KiCad plugin (boilerplate code) and its implementation provides basic checks on version compatibility between the plugin loader and the available plugins. The header `plugins/ldr/3d/pluginldr3D.h` declares a loader for the 3D Plugin Class. The loader is responsible for loading a given plugin and making its functions available to KiCad. Each instance of a plugin loader represents an actual plugin implementation and acts as a transparent bridge between KiCad and the plugin's features. The loader is not the only code required within KiCad to support plugins: we also need code to discover the plugins and code to invoke the functions of the plugins via the plugin loader. In the case of the 3D plugins the discovery and invocation functions are all contained within the S3D_CACHE class.

Plugin developers do not need to be concerned with the details of KiCad's internal code for managing plugins unless a new Plugin Class is being developed; a plugin only needs to define the functions declared by their specific plugin class.

The header `include/plugins/kicad_plugin.h` declares the generic functions required of all KiCad plugins; these functions identify the Plugin Class, provide the name of the specific plugin, provide version information for the Plugin Class API, provide version information for the specific plugin, and provide a basic version compatibility check on the Plugin Class API. In brief, these functions are:

```
/* Return a UTF-8 string naming the Plugin Class */
char const* GetKicadPluginClass( void );

/* Return version information for the Plugin Class API */
void GetClassVersion( unsigned char* Major, unsigned char* Minor,
     unsigned char* Patch, unsigned char* Revision );

/*
   Return true if the version check implemented in the plugin
   determines that the given Plugin Class API is compatible.
 */
bool CheckClassVersion( unsigned char Major,
    unsigned char Minor, unsigned char Patch, unsigned char Revision );

/* Return the name of the specific plugin, for example "PLUGIN_3D_VRML" */
const char* GetKicadPluginName( void );

/* Return version information for the specific plugin */
void GetPluginVersion( unsigned char* Major, unsigned char* Minor,
     unsigned char* Patch, unsigned char* Revision );
```

1.1.1 Plugin Class: PLUGIN_3D

The header `include/plugins/3d/3d_plugin.h` declares the functions which must be implemented by all 3D plugins and defines a number of functions which are required by the plugin and which the user must not reimplement. The defined functions which the user must not reimplement are:

```
/* Returns the Plugin Class name "PLUGIN_3D" */
char const* GetKicadPluginClass( void );

/* Return version information for the PLUGIN_3D API */
void GetClassVersion( unsigned char* Major, unsigned char* Minor,
     unsigned char* Patch, unsigned char* Revision );

/*
   Performs basic version checks enforced by the developers of
   the loader for the PLUGIN_3D class and returns true if the
   checks pass
 */
bool CheckClassVersion( unsigned char Major, unsigned char Minor,
     unsigned char Patch, unsigned char Revision );
```

The functions which the user must implement are as follows:

```
/* Return the number of extension strings supported by the plugin */
int GetNExtensions( void );

/*
   Return the requested extension string; valid values are 0 to
   GetNExtensions() - 1
 */
char const* GetModelExtension( int aIndex );

/* Return the total number of file filters supported by the plugin */
int GetNFilters( void );

/*
   Return the file filter requested; valid values are 0 to
   GetNFilters() - 1
 */
char const* GetFileFilter( int aIndex );

/*
    Return true if the plugin can render this type of 3D model.
    In some cases a plugin may not yet provide a visual model
    and must return false.
 */
bool CanRender( void );

/* Load the specified model and return a pointer to its visual model data */
SCENEGRAPH* Load( char const* aFileName );
```

2 Tutorials: 3D Plugin Class

This section contains a description of two very simple plugins of the PLUGIN_3D class and walks the user through the setup and building of the code.

2.1 Basic 3D Plugin

This tutorial walks the user through the development of a very basic 3D plugin named "PLUGIN_3D_DEMO1". The purpose of this tutorial is only to demonstrate the construction of a very basic 3D plugin which does nothing other than provide a few filter strings which permit the KiCad user to filter file names while browsing for 3D models. The code demonstrated here is the absolute minimum requirement for any 3D plugin and can be used as a template for creating more advanced plugins.

In order to build the demo project we require the following:

- CMake
- KiCad plugin headers
- KiCad Scene Graph library `kicad_3dsg`

To automatically detect the KiCad headers and library we shall use a CMake FindPackage script; the script supplied in this tutorial should work on Linux and Windows if the relevant header files are installed to `${KICAD_ROOT_DIR}/kicad` and the KiCad Scene Graph library is installed in `${KICAD_ROOT_DIR}/lib`.

To start let's create a project directory and the FindPackage script:

```
mkdir demo && cd demo
export DEMO_ROOT=${PWD}
mkdir CMakeModules && cd CMakeModules
cat > FindKICAD.cmake << _EOF
find_path( KICAD_INCLUDE_DIR kicad/plugins/kicad_plugin.h
    PATHS ${KICAD_ROOT_DIR}/include $ENV{KICAD_ROOT_DIR}/include
    DOC "Kicad plugins header path."
    )

if( NOT ${KICAD_INCLUDE_DIR} STREQUAL "KICAD_INCLUDE_DIR-NOTFOUND" )

    # attempt to extract the version information from sg_version.h
    find_file( KICAD_SGVERSION sg_version.h
        PATHS ${KICAD_INCLUDE_DIR}
        PATH_SUFFIXES kicad/plugins/3dapi
        NO_DEFAULT_PATH )

    if( NOT ${KICAD_SGVERSION} STREQUAL "KICAD_SGVERSION-NOTFOUND" )

        # extract the "#define KICADSG_VERSION*" lines
        file( STRINGS ${KICAD_SGVERSION} _version REGEX "^#define.*KICADSG_VERSION.*" )

        foreach( SVAR ${_version} )
            string( REGEX MATCH KICADSG_VERSION_[M,A,J,O,R,I,N,P,T,C,H,E,V,I,S]* _VARNAME $ ↩
                {SVAR} )
            string( REGEX MATCH [0-9]+ _VALUE ${SVAR} )

            if( NOT ${_VARNAME} STREQUAL "" AND NOT ${_VALUE} STREQUAL "" )
                set( _${_VARNAME} ${_VALUE} )
            endif()

        endforeach()

        #ensure that NOT SG3D_VERSION* will evaluate to '0'
        if( NOT _KICADSG_VERSION_MAJOR )
            set( _KICADSG_VERSION_MAJOR 0 )
        endif()

        if( NOT _KICADSG_VERSION_MINOR )
            set( _KICADSG_VERSION_MINOR 0 )
        endif()
```

```
        if( NOT _KICADSG_VERSION_PATCH )
            set( _KICADSG_VERSION_PATCH 0 )
        endif()

        if( NOT _KICADSG_VERSION_REVISION )
            set( _KICADSG_VERSION_REVISION 0 )
        endif()

        set( KICAD_VERSION ${_KICADSG_VERSION_MAJOR}.${_KICADSG_VERSION_MINOR}.${ ↩
            _KICADSG_VERSION_PATCH}.${_KICADSG_VERSION_REVISION} )
        unset( KICAD_SGVERSION CACHE )

    endif()
endif()


find_library( KICAD_LIBRARY
    NAMES kicad_3dsg
    PATHS
        ${KICAD_ROOT_DIR}/lib $ENV{KICAD_ROOT_DIR}/lib
        ${KICAD_ROOT_DIR}/bin $ENV{KICAD_ROOT_DIR}/bin
    DOC "Kicad scenegraph library path."
    )

include( FindPackageHandleStandardArgs )
FIND_PACKAGE_HANDLE_STANDARD_ARGS( KICAD
    REQUIRED_VARS
        KICAD_INCLUDE_DIR
        KICAD_LIBRARY
        KICAD_VERSION
    VERSION_VAR KICAD_VERSION )


mark_as_advanced( KICAD_INCLUDE_DIR )
set( KICAD_VERSION_MAJOR ${_KICADSG_VERSION_MAJOR} CACHE INTERNAL "" )
set( KICAD_VERSION_MINOR ${_KICADSG_VERSION_MINOR} CACHE INTERNAL "" )
set( KICAD_VERSION_PATCH ${_KICADSG_VERSION_PATCH} CACHE INTERNAL "" )
set( KICAD_VERSION_TWEAK ${_KICADSG_VERSION_REVISION} CACHE INTERNAL "" )
_EOF
```

Kicad and its plugin headers must be installed; if they are installed to a user directory or under `/opt` on Linux, or you are using Windows, you will need to set the `KICAD_ROOT_DIR` environment variable to point to the directory containing the KiCad `include` and `lib` directories. For OS X the FindPackage script presented here may require some adjustments.

To configure and build the tutorial code we will use CMake and create a `CMakeLists.txt` script file:

```
cd ${DEMO_ROOT}
cat > CMakeLists.txt << _EOF
# declare the name of the project
project( PLUGIN_DEMO )

# check that we have a version of CMake with all required features
cmake_minimum_required( VERSION 2.8.12 FATAL_ERROR )

# inform CMake of where to find the FindKICAD script
set( CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/CMakeModules )

# attempt to discover the installed kicad headers and library
# and set the variables:
#     KICAD_INCLUDE_DIR
#     KICAD_LIBRARY
```

```
find_package( KICAD 1.0 REQUIRED )

# add the kicad include directory to the compiler's search path
include_directories( ${KICAD_INCLUDE_DIR}/kicad )

# create a plugin named s3d_plugin_demo1
add_library( s3d_plugin_demo1 MODULE
    src/s3d_plugin_demo1.cpp
    )

_EOF
```

The first demo project is very basic; it consists of a single file with no external link dependencies other than the compiler defaults. We start by creating a source directory:

```
cd ${DEMO_ROOT}
mkdir src && cd src
export DEMO_SRC=${PWD}
```

Now we create the plugin source itself:

s3d_plugin_demo1.cpp

```
#include <iostream>

// the 3d_plugin.h header defines the functions required of 3D plugins
#include "plugins/3d/3d_plugin.h"

// define the version information of this plugin; do not confuse this
// with the Plugin Class version which is defined in 3d_plugin.h
#define PLUGIN_3D_DEMO1_MAJOR 1
#define PLUGIN_3D_DEMO1_MINOR 0
#define PLUGIN_3D_DEMO1_PATCH 0
#define PLUGIN_3D_DEMO1_REVNO 0

// implement the function which provides users with this plugin's name
const char* GetKicadPluginName( void )
{
    return "PLUGIN_3D_DEMO1";
}

// implement the function which provides users with this plugin's version
void GetPluginVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision )
{
    if( Major )
        *Major = PLUGIN_3D_DEMO1_MAJOR;

    if( Minor )
        *Minor = PLUGIN_3D_DEMO1_MINOR;

    if( Patch )
        *Patch = PLUGIN_3D_DEMO1_PATCH;

    if( Revision )
        *Revision = PLUGIN_3D_DEMO1_REVNO;

    return;
}

// number of extensions supported; on *NIX systems the extensions are
// provided twice - once in lower case and once in upper case letters
```

```
#ifdef _WIN32
    #define NEXTS 7
#else
    #define NEXTS 14
#endif

// number of filter sets supported
#define NFILS 5

// define the extension strings and filter strings which this
// plugin will supply to the user
static char ext0[] = "wrl";
static char ext1[] = "x3d";
static char ext2[] = "emn";
static char ext3[] = "iges";
static char ext4[] = "igs";
static char ext5[] = "stp";
static char ext6[] = "step";

#ifdef _WIN32
static char fil0[] = "VRML 1.0/2.0 (*.wrl)|*.wrl";
static char fil1[] = "X3D (*.x3d)|*.x3d";
static char fil2[] = "IDF 2.0/3.0 (*.emn)|*.emn";
static char fil3[] = "IGESv5.3 (*.igs;*.iges)|*.igs;*.iges";
static char fil4[] = "STEP (*.stp;*.step)|*.stp;*.step";
#else
static char ext7[] = "WRL";
static char ext8[] = "X3D";
static char ext9[] = "EMN";
static char ext10[] = "IGES";
static char ext11[] = "IGS";
static char ext12[] = "STP";
static char ext13[] = "STEP";

static char fil0[] = "VRML 1.0/2.0 (*.wrl;*.WRL)|*.wrl;*.WRL";
static char fil1[] = "X3D (*.x3d;*.X3D)|*.x3d;*.X3D";
static char fil2[] = "IDF 2.0/3.0 (*.emn;*.EMN)|*.emn;*.EMN";
static char fil3[] = "IGESv5.3 (*.igs;*.iges;*.IGS;*.IGES)|*.igs;*.iges;*.IGS;*.IGES";
static char fil4[] = "STEP (*.stp;*.step;*.STP;*.STEP)|*.stp;*.step;*.STP;*.STEP";
#endif

// instantiate a convenient data structure for accessing the
// lists of extension and filter strings
static struct FILE_DATA
{
    char const* extensions[NEXTS];
    char const* filters[NFILS];

    FILE_DATA()
    {
        extensions[0] = ext0;
        extensions[1] = ext1;
        extensions[2] = ext2;
        extensions[3] = ext3;
        extensions[4] = ext4;
        extensions[5] = ext5;
        extensions[6] = ext6;
        filters[0] = fil0;
        filters[1] = fil1;
        filters[2] = fil2;
        filters[3] = fil3;
        filters[4] = fil4;
```

```
#ifndef _WIN32
        extensions[7] = ext7;
        extensions[8] = ext8;
        extensions[9] = ext9;
        extensions[10] = ext10;
        extensions[11] = ext11;
        extensions[12] = ext12;
        extensions[13] = ext13;
#endif
        return;
    }

} file_data;


// return the number of extensions supported by this plugin
int GetNExtensions( void )
{
    return NEXTS;
}

// return the indexed extension string
char const* GetModelExtension( int aIndex )
{
    if( aIndex < 0 || aIndex >= NEXTS )
        return NULL;

    return file_data.extensions[aIndex];
}

// return the number of filter strings provided by this plugin
int GetNFilters( void )
{
    return NFILS;
}

// return the indexed filter string
char const* GetFileFilter( int aIndex )
{
    if( aIndex < 0 || aIndex >= NFILS )
        return NULL;

    return file_data.filters[aIndex];
}

// return false since this plugin does not provide visualization data
bool CanRender( void )
{
    return false;
}

// return NULL since this plugin does not provide visualization data
SCENEGRAPH* Load( char const* aFileName )
{
    // this dummy plugin does not support rendering of any models
    return NULL;
}
```

This source file meets all the minimum requirements to implement a 3D plugin. The plugin does not produce any data for rendering models but it can provide KiCad with a list of supported model file extensions and file extension filters to enhance the

3D model file selection dialog. Within KiCad the extension strings are used to select the plugins which may be used to load a specified model; for example, if the plugin is `wrl` then KiCad will invoke each plugin which claims to support the extension `wrl` until a plugin returns visualization data. The file filters provided by each plugin are passed to the 3D file selector dialog to improve the browsing UI.

To build the plugin:

```
cd ${DEMO_ROOT}
# export KICAD_ROOT_DIR if necessary
mkdir build && cd build
cmake .. && make
```

The plugin will be built but not installed; you must copy the plugin file to KiCad's plugin directory if you wish to load the plugin.

2.2 Advanced 3D Plugin

This tutorial walks the user through the development of a 3D plugin named "PLUGIN_3D_DEMO2". The purpose of this tutorial is to demonstrate the construction of a very basic scene graph which the KiCad previewer can render. The plugin claims to handle files of type `txt`. Although the file must exist in order for the cache manager to invoke the plugin, the file contents are not processed by this plugin; instead, the plugin simply creates a scene graph containing a pair of tetrahedra. This tutorial assumes that the first tutorial had been completed and that the CMakeLists.txt and FindKICAD.cmake script files have been created.

Place the new source file in the same directory as the previous tutorial's source file and we will extend the previous tutorial's CMakeLists.txt file to build this tutorial. Since this plugin will create a scene graph for KiCad we need to link to KiCad's scene graph library `kicad_3dsg`. KiCad's Scene Graph Library provides a set of classes which can be used to build the Scene Graph Object; the Scene Graph Object is an intermediate data visualization format used by the 3D Cache Manager. All plugins which support model visualization must translate the model data into a scene graph via this library.

The first step is to extend `CMakeLists.txt` to build this tutorial project:

```
cd ${DEMO_ROOT}
cat >> CMakeLists.txt << _EOF
add_library( s3d_plugin_demo2 MODULE
    src/s3d_plugin_demo2.cpp
    )

target_link_libraries( s3d_plugin_demo2 ${KICAD_LIBRARY} )
_EOF
```

Now we change to the source directory and create the source file:

```
cd ${DEMO_SRC}
```

s3d_plugin_demo2.cpp

```
#include <cmath>
// 3D Plugin Class declarations
#include "plugins/3d/3d_plugin.h"
// interface to KiCad Scene Graph Library
#include "plugins/3dapi/ifsg_all.h"

// version information for this plugin
#define PLUGIN_3D_DEMO2_MAJOR 1
#define PLUGIN_3D_DEMO2_MINOR 0
#define PLUGIN_3D_DEMO2_PATCH 0
#define PLUGIN_3D_DEMO2_REVNO 0

// provide the name of this plugin
const char* GetKicadPluginName( void )
{
```

```
    return "PLUGIN_3D_DEMO2";
}

// provide the version of this plugin
void GetPluginVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision )
{
    if( Major )
        *Major = PLUGIN_3D_DEMO2_MAJOR;

    if( Minor )
        *Minor = PLUGIN_3D_DEMO2_MINOR;

    if( Patch )
        *Patch = PLUGIN_3D_DEMO2_PATCH;

    if( Revision )
        *Revision = PLUGIN_3D_DEMO2_REVNO;

    return;
}


// number of extensions supported
#ifdef _WIN32
#define NEXTS 1
#else
#define NEXTS 2
#endif

// number of filter sets supported
#define NFILS 1

static char ext0[] = "txt";

#ifdef _WIN32
static char fil0[] = "demo (*.txt)|*.txt";
#else
static char ext1[] = "TXT";

static char fil0[] = "demo (*.txt;*.TXT)|*.txt;*.TXT";
#endif


static struct FILE_DATA
{
    char const* extensions[NEXTS];
    char const* filters[NFILS];

    FILE_DATA()
    {
        extensions[0] = ext0;
        filters[0] = fil0;

#ifndef _WIN32
        extensions[1] = ext1;
#endif
        return;
    }

} file_data;
```

```
int GetNExtensions( void )
{
    return NEXTS;
}


char const* GetModelExtension( int aIndex )
{
    if( aIndex < 0 || aIndex >= NEXTS )
        return NULL;

    return file_data.extensions[aIndex];
}


int GetNFilters( void )
{
    return NFILS;
}


char const* GetFileFilter( int aIndex )
{
    if( aIndex < 0 || aIndex >= NFILS )
        return NULL;

    return file_data.filters[aIndex];
}


// return true since this plugin can provide visualization data
bool CanRender( void )
{
    return true;
}


// create the visualization data
SCENEGRAPH* Load( char const* aFileName )
{
    // For this demonstration we create a tetrahedron (tx1) consisting
    // of a SCENEGRAPH (VRML Transform) which in turn contains 4
    // SGSHAPE (VRML Shape) objects representing each of the sides of
    // the tetrahedron. Each Shape is associated with a color (SGAPPEARANCE)
    // and a SGFACESET (VRML Geometry->indexedFaceSet). Each SGFACESET is
    // associated with a vertex list (SGCOORDS), a per-vertex normals
    // list (SGNORMALS), and a coordinate index (SGCOORDINDEX). One shape
    // is used to represent each face so that we may use per-vertex-per-face
    // normals.
    //
    // The tetrahedron in turn is a child of a top level SCENEGRAPH (tx0)
    // which has a second SCENEGRAPH child (tx2) which is a transformation
    // of the tetrahedron tx1 (rotation + translation). This demonstrates
    // the reuse of components within the scene graph hierarchy.

    // define the vertices of the tetrahedron
    // face 1: 0, 3, 1
    // face 2: 0, 2, 3
    // face 3: 1, 3, 2
    // face 4: 0, 1, 2
    double SQ2 = sqrt( 0.5 );
```

```
    SGPOINT vert[4];
    vert[0] = SGPOINT( 1.0, 0.0, -SQ2 );
    vert[1] = SGPOINT( -1.0, 0.0, -SQ2 );
    vert[2] = SGPOINT( 0.0, 1.0, SQ2 );
    vert[3] = SGPOINT( 0.0, -1.0, SQ2 );


    // create the top level transform; this will hold all other
    // scenegraph objects; a transform may hold other transforms and
    // shapes
    IFSG_TRANSFORM* tx0 = new IFSG_TRANSFORM( true );

    // create the transform which will house the shapes
    IFSG_TRANSFORM* tx1 = new IFSG_TRANSFORM( tx0->GetRawPtr() );

    // add a shape which we will use to define one face of the tetrahedron;
    // shapes hold facesets and appearances
    IFSG_SHAPE* shape = new IFSG_SHAPE( *tx1 );

    // add a faceset; these contain coordinate lists, coordinate indices,
    // vertex lists, vertex indices, and may also contain color lists and
    // their indices.

    IFSG_FACESET* face = new IFSG_FACESET( *shape );

    IFSG_COORDS* cp = new IFSG_COORDS( *face );
    cp->AddCoord( vert[0] );
    cp->AddCoord( vert[3] );
    cp->AddCoord( vert[1] );

    // coordinate indices - note: enforce triangles;
    // in real plugins where it is not necessarily possible
    // to determine which side a triangle is visible from,
    // 2 point orders must be specified for each triangle
    IFSG_COORDINDEX* coordIdx = new IFSG_COORDINDEX( *face );
    coordIdx->AddIndex( 0 );
    coordIdx->AddIndex( 1 );
    coordIdx->AddIndex( 2 );

    // create an appearance; appearances are owned by shapes

    // magenta
    IFSG_APPEARANCE* material = new IFSG_APPEARANCE( *shape);
    material->SetSpecular( 0.1, 0.0, 0.1 );
    material->SetDiffuse( 0.8, 0.0, 0.8 );
    material->SetAmbient( 0.2, 0.2, 0.2 );
    material->SetShininess( 0.2 );

    // normals
    IFSG_NORMALS* np = new IFSG_NORMALS( *face );
    SGVECTOR nval = S3D::CalcTriNorm( vert[0], vert[3], vert[1] );
    np->AddNormal( nval );
    np->AddNormal( nval );
    np->AddNormal( nval );

    //
    // Shape2
    // Note: we reuse the IFSG* wrappers to create and manipulate new
    // data structures.
    //
    shape->NewNode( *tx1 );
    face->NewNode( *shape );
```

```
    coordIdx->NewNode( *face );
    cp->NewNode( *face );
    np->NewNode( *face );

    // vertices
    cp->AddCoord( vert[0] );
    cp->AddCoord( vert[2] );
    cp->AddCoord( vert[3] );

    // indices
    coordIdx->AddIndex( 0 );
    coordIdx->AddIndex( 1 );
    coordIdx->AddIndex( 2 );

    // normals
    nval = S3D::CalcTriNorm( vert[0], vert[2], vert[3] );
    np->AddNormal( nval );
    np->AddNormal( nval );
    np->AddNormal( nval );
    // color (red)
    material->NewNode( *shape );
    material->SetSpecular( 0.2, 0.0, 0.0 );
    material->SetDiffuse( 0.9, 0.0, 0.0 );
    material->SetAmbient( 0.2, 0.2, 0.2 );
    material->SetShininess( 0.1 );

    //
    // Shape3
    //
    shape->NewNode( *tx1 );
    face->NewNode( *shape );
    coordIdx->NewNode( *face );
    cp->NewNode( *face );
    np->NewNode( *face );

    // vertices
    cp->AddCoord( vert[1] );
    cp->AddCoord( vert[3] );
    cp->AddCoord( vert[2] );

    // indices
    coordIdx->AddIndex( 0 );
    coordIdx->AddIndex( 1 );
    coordIdx->AddIndex( 2 );

    // normals
    nval = S3D::CalcTriNorm( vert[1], vert[3], vert[2] );
    np->AddNormal( nval );
    np->AddNormal( nval );
    np->AddNormal( nval );

    // color (green)
    material->NewNode( *shape );
    material->SetSpecular( 0.0, 0.1, 0.0 );
    material->SetDiffuse( 0.0, 0.9, 0.0 );
    material->SetAmbient( 0.2, 0.2, 0.2 );
    material->SetShininess( 0.1 );

    //
    // Shape4
    //
    shape->NewNode( *tx1 );
```

```
    face->NewNode( *shape );
    coordIdx->NewNode( *face );
    cp->NewNode( *face );
    np->NewNode( *face );

    // vertices
    cp->AddCoord( vert[0] );
    cp->AddCoord( vert[1] );
    cp->AddCoord( vert[2] );

    // indices
    coordIdx->AddIndex( 0 );
    coordIdx->AddIndex( 1 );
    coordIdx->AddIndex( 2 );

    // normals
    nval = S3D::CalcTriNorm( vert[0], vert[1], vert[2] );
    np->AddNormal( nval );
    np->AddNormal( nval );
    np->AddNormal( nval );

    // color (blue)
    material->NewNode( *shape );
    material->SetSpecular( 0.0, 0.0, 0.1 );
    material->SetDiffuse( 0.0, 0.0, 0.9 );
    material->SetAmbient( 0.2, 0.2, 0.2 );
    material->SetShininess( 0.1 );

    // create a copy of the entire tetrahedron shifted Z+2 and rotated 2/3PI
    IFSG_TRANSFORM* tx2 = new IFSG_TRANSFORM( tx0->GetRawPtr() );
    tx2->AddRefNode( *tx1 );
    tx2->SetTranslation( SGPOINT( 0, 0, 2 ) );
    tx2->SetRotation( SGVECTOR( 0, 0, 1 ), M_PI*2.0/3.0 );

    SGNODE* data = tx0->GetRawPtr();

    // delete the wrappers
    delete shape;
    delete face;
    delete coordIdx;
    delete material;
    delete cp;
    delete np;
    delete tx0;
    delete tx1;
    delete tx2;

    return (SCENEGRAPH*)data;
}
```

3 Application Programming Interface (API)

Plugins are implemented via Application Programming Interface (API) implementations. Each Plugin Class has its specific API and in the 3D Plugin tutorials we have seen examples of the implementation of the 3D Plugin API as declared by the header `3d_plugin.h`. Plugins may also rely on other APIs defined within the KiCad source tree; in the case of 3D plugins, all plugins which support visualization of models must interact with the Scene Graph API as declared in the header `ifsg_all.h` and its included headers.

This section describes the details of available Plugin Class APIs and other KiCad APIs which may be required for implementa-

tions of plugin classes.

3.1 Plugin Class APIs

There is currently only one plugin class declared for KiCad: the 3D Plugin Class. All KiCad plugin classes must implement a basic set of functions declared in the header file `kicad_plugin.h`; these declarations are referred to as the Base Kicad Plugin Class. No implementation of the Base Kicad Plugin Class exists; the header file exists purely to ensure that plugin developers implement these defined functions in each plugin implementation.

Within KiCad, each instance of a Plugin Loader implements the API presented by a plugin as though the Plugin Loader is a class providing the plugin's services. This is achieved by the Plugin Loader class providing a public interface containing function names which are similar to those implemented by the plugin; the argument lists may vary to accommodate the need to inform the user of any problems which may be encountered if, for example, no plugin is loaded. Internally the Plugin Loader uses a stored pointer to each API function to invoke each function on behalf of the user.

3.1.1 API: Base Kicad Plugin Class

The Base Kicad Plugin Class is defined by the header file `kicad_plugin.h`. This header must be included in the declaration of all other plugin classes; for an example see the 3D Plugin Class declaration in the header file `3d_plugin.h`. The prototypes for these functions were briefly described in Plugin Classes. The API is implemented by the base plugin loader as defined in `pluginldr.cpp`.

To help make sense of the functions required by the base KiCad plugin header we must look at what happens in the base Plugin Loader class. The Plugin Loader class declares a virtual function `Open()` which takes the full path to the plugin to be loaded. The implementation of the `Open()` function within a specific plugin class loader will initially invoke the protected `open()` function of the base plugin loader; this base `open()` function attempts to find the address of each of the required basic plugin functions; once the addresses of each function have been retrieved, a number of checks are enforced:

1. Plugin `GetKicadPluginClass()` is invoked and the result is compared to the Plugin Class string provided by the Plugin Loader implementation; if these strings do not match then the opened plugin is not intended for the Plugin Loader instance.
2. Plugin `GetClassVersion()` is invoked to retrieve the Plugin Class API Version implemented by the plugin.
3. Plugin Loader virtual `GetLoaderVersion()` function is invoked to retrieve the Plugin Class API Version implemented by the loader.
4. The Plugin Class API Version reported by the plugin and the loader are required to have the same Major Version number, otherwise they are considered incompatible. This is the most basic version test and it is enforced by the base plugin loader.
5. Plugin `CheckClassVersion()` is invoked with the Plugin Class API Version information of the Plugin Loader; if the Plugin supports the given version then it returns `true` to indicate success. If successful the loader creates a PluginInfo string based on the results of `GetKicadPluginName()` and `GetPluginVersion()`, and the plugin loading procedure continues within the Plugin Loader's `Open()` implementation.

3.1.2 API: 3D Plugin Class

The 3D Plugin Class is declared by the header file `3d_plugin.h` and it extends the required plugin functions as described in Plugin Class: PLUGIN_3D. The corresponding Plugin Loader is defined in `pluginldr3D.cpp` and the loader implements the following public functions in addition to the required API functions:

```
/* Open the plugin specified by the full path "aFullFileName" */
bool Open( const wxString& aFullFileName );

/* Close the currently opened plugin */
void Close( void );

/* Retrieve the Plugin Class API Version implemented by this Plugin Loader */
void GetLoaderVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Revision, unsigned char* Patch ) const;
```

The required 3D Plugin Class functions are exposed via the following functions:

```
/* returns the Plugin Class or NULL if no plugin loaded */
char const* GetKicadPluginClass( void );

/* returns false if no plugin loaded */
bool GetClassVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/* returns false if the class version check fails or no plugin is loaded */
bool CheckClassVersion( unsigned char Major, unsigned char Minor,
    unsigned char Patch, unsigned char Revision );

/* returns the Plugin Name or NULL if no plugin loaded */
const char* GetKicadPluginName( void );

/*
   returns false if no plugin is loaded, otherwise the arguments
   contain the result of GetPluginVersion()
 */
bool GetVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/*
   sets aPluginInfo to an empty string if no plugin is loaded,
   otherwise aPluginInfo is set to a string of the form:
   [NAME]:[MAJOR].[MINOR].[PATCH].[REVISION] where
   NAME = name provided by GetKicadPluginClass()
   MAJOR, MINOR, PATCH, REVISION = version information from
   GetPluginVersion()
 */
void GetPluginInfo( std::string& aPluginInfo );
```

In typical situations, the user would do the following:

1. Create an instance of `KICAD_PLUGIN_LDR_3D`.
2. Invoke `Open( "/path/to/myplugin.so" )` to open a specific plugin. The return value must be checked to ensure that the plugin loaded as desired.
3. Invoke any of the 3D Plugin Class calls as exposed by `KICAD_PLUGIN_LDR_3D`.
4. Invoke `Close()` to close (unlink) the plugin.
5. Destroy the `KICAD_PLUGIN_LDR_3D` instance.

3.2 Scenegraph Class APIs

The Scenegraph Class API is defined by the header `ifsg_all.h` and its included headers. The API consists of a number of helper routines with the namespace `S3D` as defined in `ifsg_api.h` and wrapper classes defined by the various `ifsg_*.h` headers; the wrappers support the underlying scene graph classes which, taken together, form a scene graph structure which is compatible with VRML2.0 static scene graphs. The headers, structures, classes and their public functions are as follows:

sg_version.h

```
/*
   Defines version information of the SceneGraph Classes.
   All plugins which use the scenegraph class should include this header
   and check the version information against the version reported by
   S3D::GetLibVersion() to ensure compatibility
 */
```

```
#define KICADSG_VERSION_MAJOR         2
#define KICADSG_VERSION_MINOR         0
#define KICADSG_VERSION_PATCH         0
#define KICADSG_VERSION_REVISION      0
```

sg_types.h

```
/*
   Defines the SceneGraph Class Types; these types
   are closely related to VRML2.0 node types.
 */

namespace S3D
{
    enum SGTYPES
    {
        SGTYPE_TRANSFORM = 0,
        SGTYPE_APPEARANCE,
        SGTYPE_COLORS,
        SGTYPE_COLORINDEX,
        SGTYPE_FACESET,
        SGTYPE_COORDS,
        SGTYPE_COORDINDEX,
        SGTYPE_NORMALS,
        SGTYPE_SHAPE,
        SGTYPE_END
    };
};
```

The `sg_base.h` header contains declarations of basic data types used by the scenegraph classes.

sg_base.h

```
/*
    This is an RGB color model equivalent to the VRML2.0
    RGB model where each color may have a value within the
    range [0..1].
 */

class SGCOLOR
{
public:
    SGCOLOR();
    SGCOLOR( float aRVal, float aGVal, float aBVal );

    void GetColor( float& aRedVal, float& aGreenVal, float& aBlueVal ) const;
    void GetColor( SGCOLOR& aColor ) const;
    void GetColor( SGCOLOR* aColor ) const;

    bool SetColor( float aRedVal, float aGreenVal, float aBlueVal );
    bool SetColor( const SGCOLOR& aColor );
    bool SetColor( const SGCOLOR* aColor );
};


class SGPOINT
{
public:
    double x;
    double y;
    double z;
```

```
public:
    SGPOINT();
    SGPOINT( double aXVal, double aYVal, double aZVal );

    void GetPoint( double& aXVal, double& aYVal, double& aZVal );
    void GetPoint( SGPOINT& aPoint );
    void GetPoint( SGPOINT* aPoint );

    void SetPoint( double aXVal, double aYVal, double aZVal );
    void SetPoint( const SGPOINT& aPoint );
};


/*
    A SGVECTOR has 3 components (x,y,z) similar to a point; however
    a vector ensures that the stored values are normalized and
    prevents direct manipulation of the component variables.
 */
class SGVECTOR
{
public:
    SGVECTOR();
    SGVECTOR( double aXVal, double aYVal, double aZVal );

    void GetVector( double& aXVal, double& aYVal, double& aZVal ) const;

    void SetVector( double aXVal, double aYVal, double aZVal );
    void SetVector( const SGVECTOR& aVector );

    SGVECTOR& operator=( const SGVECTOR& source );
};
```

The `IFSG_NODE` class is the base class for all scenegraph nodes. All scenegraph objects implement the public functions of this class but in some cases a particular function may have no meaning for a specific class.

ifsg_node.h

```
class IFSG_NODE
{
public:
    IFSG_NODE();
    virtual ~IFSG_NODE();

    /**
     * Function Destroy
     * deletes the scenegraph object held by this wrapper
     */
    void Destroy( void );

    /**
     * Function Attach
     * associates a given SGNODE* with this wrapper
     */
    virtual bool Attach( SGNODE* aNode ) = 0;

    /**
     * Function NewNode
     * creates a new node to associate with this wrapper
     */
    virtual bool NewNode( SGNODE* aParent ) = 0;
    virtual bool NewNode( IFSG_NODE& aParent ) = 0;
```

```
    /**
     * Function GetRawPtr()
     * returns the raw internal SGNODE pointer
     */
    SGNODE* GetRawPtr( void );

    /**
     * Function GetNodeType
     * returns the type of this node instance
     */
    S3D::SGTYPES GetNodeType( void ) const;

    /**
     * Function GetParent
     * returns a pointer to the parent SGNODE of this object
     * or NULL if the object has no parent (ie. top level transform)
     * or if the wrapper is not currently associated with an SGNODE.
     */
    SGNODE* GetParent( void ) const;

    /**
     * Function SetParent
     * sets the parent SGNODE of this object.
     *
     * @param aParent [in] is the desired parent node
     * @return true if the operation succeeds; false if
     * the given node is not allowed to be a parent to
     * the derived object.
     */
    bool SetParent( SGNODE* aParent );

    /**
     * Function GetNodeTypeName
     * returns the text representation of the node type
     * or NULL if the node somehow has an invalid type
     */
    const char * GetNodeTypeName( S3D::SGTYPES aNodeType ) const;

    /**
     * Function AddRefNode
     * adds a reference to an existing node which is not owned by
     * (not a child of) this node.
     *
     * @return true on success
     */
    bool AddRefNode( SGNODE* aNode );
    bool AddRefNode( IFSG_NODE& aNode );

    /**
     * Function AddChildNode
     * adds a node as a child owned by this node.
     *
     * @return true on success
     */
    bool AddChildNode( SGNODE* aNode );
    bool AddChildNode( IFSG_NODE& aNode );
};
```

`IFSG_TRANSFORM` is similar to a VRML2.0 Transform node; it may contain any number of child IFSG_SHAPE and IFSG_TRANSFO
nodes and any number of referenced IFSG_SHAPE and IFSG_TRANSFORM nodes. A valid scenegraph must have a single

IFSG_TRANSFORM object as a root.

ifsg_transform.h

```
/**
 * Class IFSG_TRANSFORM
 * is the wrapper for the VRML compatible TRANSFORM block class SCENEGRAPH
 */

class IFSG_TRANSFORM : public IFSG_NODE
{
public:
    IFSG_TRANSFORM( bool create );
    IFSG_TRANSFORM( SGNODE* aParent );

    bool SetScaleOrientation( const SGVECTOR& aScaleAxis, double aAngle );
    bool SetRotation( const SGVECTOR& aRotationAxis, double aAngle );
    bool SetScale( const SGPOINT& aScale );
    bool SetScale( double aScale );
    bool SetCenter( const SGPOINT& aCenter );
    bool SetTranslation( const SGPOINT& aTranslation );

    /* various base class functions not shown here */
};
```

IFSG_SHAPE is similar to a VRML2.0 Shape node; it must contain a single child or reference FACESET node and may contain a single child or reference APPEARANCE node.

ifsg_shape.h

```
/**
 * Class IFSG_SHAPE
 * is the wrapper for the SGSHAPE class
 */

class IFSG_SHAPE : public IFSG_NODE
{
public:
    IFSG_SHAPE( bool create );
    IFSG_SHAPE( SGNODE* aParent );
    IFSG_SHAPE( IFSG_NODE& aParent );

    /* various base class functions not shown here */
};
```

IFSG_APPEARANCE is similar to a VRML2.0 Appearance node, however, at the moment it only represents the equivalent of an Appearance node containing a Material node.

ifsg_appearance.h

```
class IFSG_APPEARANCE : public IFSG_NODE
{
public:
    IFSG_APPEARANCE( bool create );
    IFSG_APPEARANCE( SGNODE* aParent );
    IFSG_APPEARANCE( IFSG_NODE& aParent );

    bool SetEmissive( float aRVal, float aGVal, float aBVal );
    bool SetEmissive( const SGCOLOR* aRGBColor );
    bool SetEmissive( const SGCOLOR& aRGBColor );

    bool SetDiffuse( float aRVal, float aGVal, float aBVal );
    bool SetDiffuse( const SGCOLOR* aRGBColor );
```

```
    bool SetDiffuse( const SGCOLOR& aRGBColor );

    bool SetSpecular( float aRVal, float aGVal, float aBVal );
    bool SetSpecular( const SGCOLOR* aRGBColor );
    bool SetSpecular( const SGCOLOR& aRGBColor );

    bool SetAmbient( float aRVal, float aGVal, float aBVal );
    bool SetAmbient( const SGCOLOR* aRGBColor );
    bool SetAmbient( const SGCOLOR& aRGBColor );

    bool SetShininess( float aShininess );
    bool SetTransparency( float aTransparency );

    /* various base class functions not shown here */

    /* the following functions make no sense within an
       appearance node and always return a failure code

        bool AddRefNode( SGNODE* aNode );
        bool AddRefNode( IFSG_NODE& aNode );
        bool AddChildNode( SGNODE* aNode );
        bool AddChildNode( IFSG_NODE& aNode );
     */
};
```

IFSG_FACESET is similar to a VRML2.0 Geometry node which contains an IndexedFaceSet node. It must contain a single child or reference COORDS node, a single child COORDINDEX node, and a single child or reference NORMALS node; in addition there may be a single child or reference COLORS node. A simplistic normals calculation function is provided to aid the user in assigning normal values to surfaces. The deviations from the VRML2.0 analogue are as follows:

1. Normals are always per-vertex.
2. Colors are always per vertex.
3. The coordinate index set must describe triangular faces only.

ifsg_faceset.h

```
/**
 * Class IFSG_FACESET
 * is the wrapper for the SGFACESET class
 */

class IFSG_FACESET : public IFSG_NODE
{
public:
    IFSG_FACESET( bool create );
    IFSG_FACESET( SGNODE* aParent );
    IFSG_FACESET( IFSG_NODE& aParent );

    bool CalcNormals( SGNODE** aPtr );

    /* various base class functions not shown here */
};
```

ifsg_coords.h

```
/**
 * Class IFSG_COORDS
 * is the wrapper for SGCOORDS
 */
```

```
class IFSG_COORDS : public IFSG_NODE
{
public:
    IFSG_COORDS( bool create );
    IFSG_COORDS( SGNODE* aParent );
    IFSG_COORDS( IFSG_NODE& aParent );

    bool GetCoordsList( size_t& aListSize, SGPOINT*& aCoordsList );
    bool SetCoordsList( size_t aListSize, const SGPOINT* aCoordsList );
    bool AddCoord( double aXValue, double aYValue, double aZValue );
    bool AddCoord( const SGPOINT& aPoint );

    /* various base class functions not shown here */

    /* the following functions make no sense within a
       coords node and always return a failure code

        bool AddRefNode( SGNODE* aNode );
        bool AddRefNode( IFSG_NODE& aNode );
        bool AddChildNode( SGNODE* aNode );
        bool AddChildNode( IFSG_NODE& aNode );
     */
};
```

IFSG_COORDINDEX is similar to a VRML2.0 coordIdx[] set except it must exclusively describe triangular faces, which implies that the total number of indices is divisible by 3.

ifsg_coordindex.h

```
/**
 * Class IFSG_COORDINDEX
 * is the wrapper for SGCOORDINDEX
 */

class IFSG_COORDINDEX : public IFSG_INDEX
{
public:
    IFSG_COORDINDEX( bool create );
    IFSG_COORDINDEX( SGNODE* aParent );
    IFSG_COORDINDEX( IFSG_NODE& aParent );

    bool GetIndices( size_t& nIndices, int*& aIndexList );
    bool SetIndices( size_t nIndices, int* aIndexList );
    bool AddIndex( int aIndex );

    /* various base class functions not shown here */

    /* the following functions make no sense within a
       coordindex node and always return a failure code

        bool AddRefNode( SGNODE* aNode );
        bool AddRefNode( IFSG_NODE& aNode );
        bool AddChildNode( SGNODE* aNode );
        bool AddChildNode( IFSG_NODE& aNode );
     */
};
```

IFSG_NORMALS is equivalent to a VRML2.0 Normals node.

ifsg_normals.h

```
/**
 * Class IFSG_NORMALS
 * is the wrapper for the SGNORMALS class
 */

class IFSG_NORMALS : public IFSG_NODE
{
public:
    IFSG_NORMALS( bool create );
    IFSG_NORMALS( SGNODE* aParent );
    IFSG_NORMALS( IFSG_NODE& aParent );

    bool GetNormalList( size_t& aListSize, SGVECTOR*& aNormalList );
    bool SetNormalList( size_t aListSize, const SGVECTOR* aNormalList );
    bool AddNormal( double aXValue, double aYValue, double aZValue );
    bool AddNormal( const SGVECTOR& aNormal );

    /* various base class functions not shown here */

    /* the following functions make no sense within a
       normals node and always return a failure code

        bool AddRefNode( SGNODE* aNode );
        bool AddRefNode( IFSG_NODE& aNode );
        bool AddChildNode( SGNODE* aNode );
        bool AddChildNode( IFSG_NODE& aNode );
     */
};
```

`IFSG_COLORS` is similar to a VRML2.0 colors[] set.

ifsg_colors.h

```
/**
 * Class IFSG_COLORS
 * is the wrapper for SGCOLORS
 */

class IFSG_COLORS : public IFSG_NODE
{
public:
    IFSG_COLORS( bool create );
    IFSG_COLORS( SGNODE* aParent );
    IFSG_COLORS( IFSG_NODE& aParent );

    bool GetColorList( size_t& aListSize, SGCOLOR*& aColorList );
    bool SetColorList( size_t aListSize, const SGCOLOR* aColorList );
    bool AddColor( double aRedValue, double aGreenValue, double aBlueValue );
    bool AddColor( const SGCOLOR& aColor );

    /* various base class functions not shown here */

    /* the following functions make no sense within a
       normals node and always return a failure code

        bool AddRefNode( SGNODE* aNode );
        bool AddRefNode( IFSG_NODE& aNode );
        bool AddChildNode( SGNODE* aNode );
        bool AddChildNode( IFSG_NODE& aNode );
     */
};
```

The remaining API functions are defined in `ifsg_api.h` as follows:

ifsg_api.h

```
namespace S3D
{
    /**
     * Function GetLibVersion retrieves version information of the
     * kicad_3dsg library
     */
    SGLIB_API void GetLibVersion( unsigned char* Major, unsigned char* Minor,
                                  unsigned char* Patch, unsigned char* Revision );

    // functions to extract information from SGNODE pointers
    SGLIB_API S3D::SGTYPES GetSGNodeType( SGNODE* aNode );
    SGLIB_API SGNODE* GetSGNodeParent( SGNODE* aNode );
    SGLIB_API bool AddSGNodeRef( SGNODE* aParent, SGNODE* aChild );
    SGLIB_API bool AddSGNodeChild( SGNODE* aParent, SGNODE* aChild );
    SGLIB_API void AssociateSGNodeWrapper( SGNODE* aObject, SGNODE** aRefPtr );

    /**
     * Function CalcTriNorm
     * returns the normal vector of a triangle described by vertices p1, p2, p3
     */
    SGLIB_API SGVECTOR CalcTriNorm( const SGPOINT& p1, const SGPOINT& p2, const SGPOINT& p3 ↩
        );

    /**
     * Function WriteCache
     * writes the SGNODE tree to a binary cache file
     *
     * @param aFileName is the name of the file to write
     * @param overwrite must be set to true to overwrite an existing file
     * @param aNode is any node within the node tree which is to be written
     * @return true on success
     */
    SGLIB_API bool WriteCache( const char* aFileName, bool overwrite, SGNODE* aNode,
        const char* aPluginInfo );

    /**
     * Function ReadCache
     * reads a binary cache file and creates an SGNODE tree
     *
     * @param aFileName is the name of the binary cache file to be read
     * @return NULL on failure, on success a pointer to the top level SCENEGRAPH node;
     * if desired this node can be associated with an IFSG_TRANSFORM wrapper via
     * the IFSG_TRANSFORM::Attach() function.
     */
    SGLIB_API SGNODE* ReadCache( const char* aFileName, void* aPluginMgr,
        bool (*aTagCheck)( const char*, void* ) );

    /**
     * Function WriteVRML
     * writes out the given node and its subnodes to a VRML2 file
     *
     * @param filename is the name of the output file
     * @param overwrite should be set to true to overwrite an existing VRML file
     * @param aTopNode is a pointer to a SCENEGRAPH object representing the VRML scene
     * @param reuse should be set to true to make use of VRML DEF/USE features
     * @return true on success
     */
    SGLIB_API bool WriteVRML( const char* filename, bool overwrite, SGNODE* aTopNode,
                    bool reuse, bool renameNodes );
```

```
// NOTE: The following functions are used in combination to create a VRML
// assembly which may use various instances of each SG* representation of a module.
// A typical use case would be:
// 1. invoke 'ResetNodeIndex()' to reset the global node name indices
// 2. for each model pointer provided by 'S3DCACHE->Load()', invoke 'RenameNodes()' once;
//    this ensures that all nodes have a unique name to present to the final output file.
//    Internally, RenameNodes() will only rename the given node and all Child subnodes;
//    nodes which are only referenced will not be renamed. Using the pointer supplied
//    by 'S3DCACHE->Load()' ensures that all nodes but the returned node (top node) are
//    children of at least one node, so all nodes are given unique names.
// 3. if SG* trees are created independently of S3DCACHE->Load() the user must invoke
//    RenameNodes() as appropriate to ensure that all nodes have a unique name
// 4. create an assembly structure by creating new IFSG_TRANSFORM nodes as appropriate
//    for each instance of a component; the component base model as returned by
//    S3DCACHE->Load() may be added to these IFSG_TRANSFORM nodes via 'AddRefNode()';
//    set the offset, rotation, etc of the IFSG_TRANSFORM node to ensure correct
// 5. Ensure that all new IFSG_TRANSFORM nodes are placed as child nodes within a
//    top level IFSG_TRANSFORM node in preparation for final node naming and output
// 6. Invoke RenameNodes() on the top level assembly node
// 7. Invoke WriteVRML() as normal, with renameNodes = false, to write the entire assembly
//    structure to a single VRML file
// 8. Clean up by deleting any extra IFSG_TRANSFORM wrappers and their underlying SG*
//    classes which have been created solely for the assembly output

/**
 * Function ResetNodeIndex
 * resets the global SG* class indices
 *
 * @param aNode may be any valid SGNODE
 */
SGLIB_API void ResetNodeIndex( SGNODE* aNode );

/**
 * Function RenameNodes
 * renames a node and all children nodes based on the current
 * values of the global SG* class indices
 *
 * @param aNode is a top level node
 */
SGLIB_API void RenameNodes( SGNODE* aNode );

/**
 * Function DestroyNode
 * deletes the given SG* class node. This function makes it possible
 * to safely delete an SG* node without associating the node with
 * its corresponding IFSG* wrapper.
 */
SGLIB_API void DestroyNode( SGNODE* aNode );

// NOTE: The following functions facilitate the creation and destruction
// of data structures for rendering

/**
 * Function GetModel
 * creates an S3DMODEL representation of aNode (raw data, no transforms)
 *
 * @param aNode is the node to be transcribed into an S3DMODEL representation
 * @return an S3DMODEL representation of aNode on success, otherwise NULL
```

```
     */
    SGLIB_API S3DMODEL* GetModel( SCENEGRAPH* aNode );

    /**
     * Function Destroy3DModel
     * frees memory used by an S3DMODEL structure and sets the pointer to
     * the structure to NULL
     */
    SGLIB_API void Destroy3DModel( S3DMODEL** aModel );

    /**
     * Function Free3DModel
     * frees memory used internally by an S3DMODEL structure
     */
    SGLIB_API void Free3DModel( S3DMODEL& aModel );

    /**
     * Function Free3DMesh
     * frees memory used internally by an SMESH structure
     */
    SGLIB_API void Free3DMesh( SMESH& aMesh );

    /**
     * Function New3DModel
     * creates and initializes an S3DMODEL struct
     */
    SGLIB_API S3DMODEL* New3DModel( void );

    /**
     * Function Init3DMaterial
     * initializes an SMATERIAL struct
     */
    SGLIB_API void Init3DMaterial( SMATERIAL& aMat );

    /**
     * Function Init3DMesh
     * creates and initializes an SMESH struct
     */
    SGLIB_API void Init3DMesh( SMESH& aMesh );
};
```

For actual usage examples of the Scenegraph API see the Advanced 3D Plugin tutorial above and the KiCad VRML1, VRML2, and X3D parsers.