

# KiCad プラグイン

## Table of Contents

KiCad プラグイン システム について .....	2
プラグインのクラス .....	2
チュートリアル: 3D プラグイン クラス .....	4
基本的な 3D プラグイン .....	4
高度な 3D プラグイン .....	9
アプリケーション プログラミング インタフェース (API) .....	12
プラグイン クラス API .....	12
シーングラフ クラス API .....	14

## KiCad プラグイン システム

### 著作権

このドキュメントは以下の貢献者により著作権所有 © 2016 されています。あなたは、GNU General Public License ( <http://www.gnu.org/licenses/gpl.html> ) のバージョン 3 以降、あるいはクリエイティブ・コモンズ・ライセンス ( <http://creativecommons.org/licenses/by/3.0/> ) のバージョン 3.0 以降のいずれかの条件の下で、配布または変更することができます。

このガイドの中のすべての商標は、正当な所有者に帰属します。

### 貢献者

Cirilo Bernardo

### 翻訳

starfort <starfort AT nifty.com>, 2017.

### フィードバック

バグ報告や提案はこちらへお知らせください:

- KiCad のドキュメントについて: <https://gitlab.com/kicad/services/kicad-doc/issues>
- KiCad ソフトウェアについて: <https://gitlab.com/kicad/code/kicad/issues>
- KiCad ソフトウェアの国際化について: <https://gitlab.com/kicad/code/kicad-i18n/issues>

### 発行日とソフトウェアのバージョン

2016年1月29日 発行

# KiCad プラグイン システム について

KiCad プラグイン システムは、共有ライブラリを用いた KiCad の機能を拡張するためのフレームワークです。プラグインを使用する主な利点の一つは、プラグイン開発中に KiCad パッケージ一式を再構築する必要がないということです。実際、プラグインは、KiCad ソース ツリーにあるヘッダのごく小さなセットを使ってビルドすることができます。開発者が直接プラグインに関するコードをコンパイルするだけで済むことが保証されると、その結果として各ビルドとテストのサイクルに必要な時間を減らすことができるので、プラグイン開発において KiCad をビルドする必要をなくすということは、生産性を大きく向上させることになります。

新しいモデル形式に対して KiCad ソースのメジャーな変更を伴うことなく、より多くの 3D モデル形式をサポートする目的で、プラグインは当初、3D モデルビューア用に開発されました。プラグイン フレームワークは、将来の開発者がプラグインの異なったクラスを作成できるよう、後に一般化されたものです。今のところ、KiCad では 3D プラグインのみ実装されていますが、将来的にはデータのインポートとエクスポートをユーザーによって実装できるようにするための PCB プラグインが開発される見通しです。

## プラグインのクラス

各プラグインはそれぞれ特定領域に関する問題を処理するので、その領域毎に別々のインターフェイスが必要となります。このため、プラグインはプラグイン クラスへと分類されます。例えば、3D モデル プラグインはファイルから 3D モデル データを読み込んで、3D ビューアで表示できるようなフォーマットへとデータを変換します。PCB インポート/エクスポート プラグインは PCB のデータを受け取って別の電気的あるいは機械的なデータ フォーマットへとエクスポートしたり、外部フォーマットを KiCad PCB 形式に変換したりします。現時点では、3D プラグイン クラスのみ開発されており、本文書でも集中して取り上げていきます。

プラグイン クラスを実装するには、KiCad ソース ツリー内でプラグインの読み込み管理を行うコードを作成することが必要です。全てのプラグイン ロードーに対する基本クラスは、KiCad ソース ツリー内のファイル `plugins/ldr/pluginldr.h` で宣言されています。このクラスは、あらゆる KiCad プラグインで見つかるであろう(お約束のコードである)最も基本的な関数を宣言しており、プラグイン ロードーと利用可能なプラグイン間のバージョン互換について最低限のチェックを行う機能を持っています。ヘッダ `plugins/ldr/3d/pluginldr3D.h` には、3D プラグイン クラスのためのロードーが宣言されています。ロードーは与えられたプラグインの読み込みを担当し、プラグインが持っている関数を KiCad で利用可能にします。各プラグイン ロードーのインスタンスはプラグイン実装の実体であり、KiCad とプラグインの関数を透過的に橋渡しするよう振舞います。プラグインをサポートするために KiCad 内部で必要とされるコードはロードーだけではありません: プラグインを見つけるためのコード、プラグイン ロードー経由でプラグインの関数を呼び出すコードも必要です。3D プラグインの場合、この発見と呼び出しの関数は `S3D_CACHE` クラス内に全て含まれています。

新規のプラグイン クラスを開発する場合以外、プラグインの開発者はプラグイン管理に関する KiCad 内部コードの詳細を知る必要はありません; プラグインに自身が属するプラグイン クラスで宣言されている関数を定義していくだけで済みます。

ヘッダ `include/plugins/kicad_plugin.h` には、全ての KiCad プラグインで必要とされるジェネリック関数が宣言されています; これらの関数は、プラグイン クラスを識別し、固有のプラグイン名、プラグイン クラス API に関するバージョン情報、固有のプラグインに関するバージョン情報、プラグイン クラス API 上での最低限のバージョン互換チェック機能を提供します。以下は、これらの関数についての要約です:

```

/* プラグイン クラス名を UTF-8 文字列で返す */
char const* GetKicadPluginClass( void );

/* プラグイン クラス API に関するバージョン情報を返す */
void GetClassVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/*
    プラグインに実装されたバージョンチェックが与えられた
    プラグイン クラス API との互換性を確認できた場合、true を返す
*/
bool CheckClassVersion( unsigned char Major,
    unsigned char Minor, unsigned char Patch, unsigned char Revision );

/* 固有のプラグイン名を返す、(例) "PLUGIN_3D_VRML" */
const char* GetKicadPluginName( void );

/* 固有のプラグインに関するバージョン情報を返す */
void GetPluginVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

```

## プラグインクラス: PLUGIN\_3D

ヘッダ `include/plugins/3d/3d_plugin.h` には、全ての 3D プラグインで実装されなければならない関数が宣言されており、プラグインにとって必要な及びユーザーが再実装する必要がない幾つかの関数が定義されています。ユーザーが再実装する必要がない定義済み関数は以下のとおりです:

```

/* プラグイン クラス名 "PLUGIN_3D" を返す */
char const* GetKicadPluginClass( void );

/* PLUGIN_3D API に関するバージョン情報を返す */
void GetClassVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/*
    PLUGIN_3D クラスに関し、ローダーの開発者による強制的な
    最低限のバージョンチェックを行なう。チェックにパスした場合、
    true を返す
*/
bool CheckClassVersion( unsigned char Major, unsigned char Minor,
    unsigned char Patch, unsigned char Revision );

```

ユーザーが実装しなければならない関数は次のとおりです:

```

/* プラグインでサポートされている拡張子文字列の数を返す */
int GetNExtensions( void );

/*
   要求された拡張子の文字列を返す; 有効な値は 0 から
   GetNExtensions() - 1
*/
char const* GetModelExtension( int aIndex );

/* プラグインでサポートされているファイル フィルタの合計数を返す */
int GetNFilters( void );

/*
   要求されたファイル フィルタを返す; 有効な値は 0 から
   GetNFilters() - 1
*/
char const* GetFileFilter( int aIndex );

/*
   この 3D モデル タイプをレンダリングできる場合、true を返す
   プラグインがビジュアル モードを提供しないことがあるかも知れないが、その場合は
   false を返さなければならない
*/
bool CanRender( void );

/* 指定されたモデルを読み込み、ビジュアル モデル データへのポインタを返す */
SCENEGRAPH* Load( char const* aFileName );

```

## チュートリアル: 3D プラグインクラス

この章では2つのとても単純な PLUGIN\_3D クラス プラグインについて解説し、ユーザーがセットアップとコードのビルドを習得できるよう予行演習を行います。

### 基本的な 3D プラグイン

このチュートリアルでは、"PLUGIN\_3D\_DEMO1" という名前の非常に基本的な 3D プラグインを開発することで予行演習を行います。このチュートリアルの目的は、KiCad ユーザーが 3D モデルをブラウズする際に使用可能なファイル名をフィルタリングする幾つかの文字列を提供するだけという、非常に基本的な 3D プラグインの作成方法をデモすることです。ここでデモしているコードは任意の 3D プラグイン に対する最低必要条件であり、より高度なプラグインを作成するためのテンプレートとして使用することができます。

デモ プロジェクトをビルドするために必要なものは以下のとおりです:

- [CMake](#)
- KiCad プラグイン ヘッド
- KiCad Scene Graph ライブラリ `kicad_3dsg`

自動的に KiCad のヘッドとライブラリを検出するためには、CMake FindPackage スクリプトを使うべきでしょう; 関連するヘッド ファイルが `${KICAD_ROOT_DIR}/kicad` に、KiCad Scene Graph ライブラリが

`${KICAD_ROOT_DIR}/lib` にインストールされているなら、このチュートリアルで提供されているスクリプトは Linux および Windows 上で動作するはずです。

まず手始めに、プロジェクト ディレクトリと FindPackage スクリプトを作成してみましょう:

```

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" )

    # sg_version.h からバージョン情報の取得を試みる
    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" )

        # "#define KICADSG_VERSION*" 行を抜き出す
        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()

        # NOT SG3D_VERSION* が '0' と評価されることを保証する
        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 とそのプラグイン ヘッダーをインストールしておく必要があります; もし Linux でそれらがユーザー ディレクトリまたは `/opt` の下にインストールされているか、或いは Windows を使っているならば、KiCad の `include` と `lib` ディレクトリを含むディレクトリを指すように `KICAD_ROOT_DIR` 環境変数をセットする必要があります。OS Xでは、ここに示された `FindPackage` スクリプトを多少調整する必要があるでしょう。

チュートリアルを組んでビルドするため、CMake を使用して `CMakeLists.txt` スクリプト ファイルを作成します:

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

# 必要な機能を全て備えた CMake のバージョンかどうかチェックする
cmake_minimum_required( VERSION 2.8.12 FATAL_ERROR )

# FindKICAD スクリプトがどこにあるか CMake に通知する
set( CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/CMakeModules )

# インストール済の KiCad ヘッダとライブラリを見つけるよう試みる
# そして変数にセットする:
#     KICAD_INCLUDE_DIR
#     KICAD_LIBRARY
find_package( KICAD 1.0 REQUIRED )

# コンパイラの検索パスに KiCad include ディレクトリを追加する
include_directories( ${KICAD_INCLUDE_DIR}/kicad )

# s3d_plugin_demo1 という名前のプラグインを作成する
add_library( s3d_plugin_demo1 MODULE
    src/s3d_plugin_demo1.cpp
)

_EOF
```

最初のデモ プロジェクトはとても基本的なものです; コンパイラのデフォルト以外は外部リンクの依存関係を持たない単一ファイルで構成されています。まずはソース ディレクトリを作成することから始めます:

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

ではプラグイン ソース自体を作成しましょう:

## s3d\_plugin\_demo1.cpp

```
#include <iostream>

// 3d_plugin.h ヘッダに 3D プラグインで必要とされる関数を定義する
#include "plugins/3d/3d_plugin.h"

// このプラグインのバージョン情報を定義する; 3d_plugin.h で定義されている
// プラ??グイン クラス バージョンと混同しないでください
#define PLUGIN_3D_DEMO1_MAJOR 1
#define PLUGIN_3D_DEMO1_MINOR 0
#define PLUGIN_3D_DEMO1_PATCH 0
#define PLUGIN_3D_DEMO1_REVNO 0

// このプラグイン名をユーザーに提供する関数を実装する
const char* GetKicadPluginName( void )
{
    return "PLUGIN_3D_DEMO1";
}

// このプラグインのバージョンをユーザに提供する関数を実装する
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;
}

// サポートされている拡張子の数; *NIX システムでは拡張子が
// 倍になります - 一つは小文字、もう一つは大文字です
#ifdef _WIN32
    #define NEXTS 7
#else
    #define NEXTS 14
#endif

// サポートされているフィルタ セットの数
#define NFILS 5

// このプラグインがユーザーに提供するフィルタ文字列と
// 拡張子の文字列を定義する
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";
```



このソースファイルは 3D プラグインを実装するための最低必要条件を満足しています。このプラグインはモデルをレンダリングするためのいかなるデータも作り出しませんが、3D モデル ファイル選択ダイアログを機能強化してサポートされているモデル拡張子のリストとファイル拡張子フィルタを KiCad に提供します。KiCad 内で拡張子文字列は特定の 3D モデルを読み込むために使われるプラグインの選択で使用されます; 例えば、もしプラグインが `wrl` ならば、KiCad はプラグインが可視化されたデータを返すまで拡張子 `wrl` をサポートすると宣言された各プラグインを呼び出すでしょう。各プラグインが提供するファイル フィルタはブラウジング UI を改良すべく 3D ファイル選択ダイアログへと渡されます。

プラグインをビルドするには:

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

プラグインはビルドされますが、インストールはされません; プラグインを読み込みたいのであれば、KiCad のプラグイン ディレクトリにプラグイン ファイルをコピーする必要があります。

## 高度な 3D プラグイン

このチュートリアルでは、"PLUGIN\_3D\_DEMO2" という名前の 3D プラグインを開発することで予行演習を行います。このチュートリアルの目的は、レンダリング可能な KiCad プレビューで非常に基本的なシーン グラフの構造図をデモすることです。このプラグインは `txt` タイプのファイルを要求します。キャッシュ マネージャーがプラグインを呼び出すためにはファイルが存在しなければなりませんが、ファイルの中身はプラグインでは処理されません; 代わりにプラグインは単に四面体のペアを含んだシーン グラフを作るだけです。このチュートリアルは、最初のチュートリアルを完了しており `CMakeLists.txt` と `FindKICAD.cmake` スクリプト ファイルが既に作られている状況を想定して書かれています。

前のチュートリアルのソース ファイルと同じディレクトリに新しいソース ファイルを置き、このチュートリアルをビルドするため前のチュートリアルの `CMakeLists.txt` ファイルを拡張しましょう。このプラグインは KiCad のシーン グラフを作るので、KiCad のシーン グラフ ライブラリ `kicad_3dsd` へのリンクが必要となります。KiCad のシーン グラフ ライブラリはシーン グラフ オブジェクトをビルドするために使われるクラスのセットを提供します。シーン グラフ オブジェクトは 3D キャッシュ マネージャで使われる可視化フォーマットの間データです。モデルの可視化をサポートする全てのプラグインは、このライブラリ経由でモデルデータをシーン グラフへと変換しなければなりません。

最初のステップは、このチュートリアル プロジェクトをビルドできるように `CMakeLists.txt` を拡張することです:

```
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
```

ではソース ディレクトリへと移動してソース ファイルを作成しましょう:

```
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"

// このプラグインのバージョン情報
#define PLUGIN_3D_DEMO2_MAJOR 1
#define PLUGIN_3D_DEMO2_MINOR 0
#define PLUGIN_3D_DEMO2_PATCH 0
#define PLUGIN_3D_DEMO2_REVNO 0

// このプラグインの名前を提供する
const char* GetKicadPluginName( void )
{
    return "PLUGIN_3D_DEMO2";
}

// このプラグインのバージョンを提供する
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;
}

// サポートされている拡張子の数
#ifdef _WIN32
#define NEXTS 1
#else
#define NEXTS 2
#endif

// サポートされているフィルタ セットの数
#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
```

# アプリケーションプログラミングインタフェース (API)

プラグインはアプリケーション プログラミング インタフェース (API) の実装により開発されます。各プラグイン クラスは固有の API を持っており、3D プラグイン チュートリアルでは `3d_plugin.h` ヘッダで宣言された 3D プラグイン API の実装例を見てきました。プラグインはまた KiCad ソース ツリーで定義された別の API にも依存しています; 3D プラグインの例では、モデルの可視化をサポートする全てのプラグインは `ifsg_all.h` ヘッダとそれ自身が包んでいるヘッダとで宣言された Scene Graph API と密接に関わらなければなりません。

この章では利用可能なプラグイン クラス API とプラグイン クラスの実装に必要と思われる別の KiCad API の詳細について説明します。

## プラグイン クラス API

今のところ、KiCad で宣言されているプラグイン クラスは次の一つだけです: 3D プラグイン クラス。全ての KiCad プラグイン クラスは `kicad_plugin.h` で宣言されている基本的な関数のセットを実装しなければなりません; これらの宣言はベース KiCad プラグイン クラスとして参照されます。ベース KiCad プラグイン クラスの実装は存在していません; ヘッダ ファイルはプラグイン開発者が各プラグインでこれらの定義済み関数を確実に実装するためだけに存在しています。

KiCadでは、プラグイン ローダーの各インスタンスはプラグインが公開する API を提供します。プラグイン ローダーはプラグインのサービスを提供するクラスのようなものです。これはプラグインで実装されたものと同様な関数名を含んだパブリック インターフェイスをプラグイン ローダー クラスが提供することで実現されています; 引数リストは、例えばプラグインが読み込まれていないというような予見される問題をユーザーに知らせる必要に応じて、適宜変更しても構いません。内部的には、プラグイン ローダーはユーザーに代わって各関数を呼び出すために各 API 関数へのストアド ポインタを使用します。

## API: ベース KiCad プラグイン クラス

ベース KiCad プラグイン クラスはヘッダ ファイル `kicad_plugin.h` で定義されます。このヘッダは全ての異なるプラグイン クラスの宣言に含まれなければなりません; 例としてヘッダ ファイル `3d_plugin.h` で宣言された 3D プラグイン クラスを見てみましょう。これらの関数のプロトタイプは [Plugin Classes](#)内で簡潔に記述されています。API は `pluginldr.cpp` で定義されているようにベース プラグイン ローダーによって提供されます。

ベース KiCad プラグイン ヘッダで要求される関数を理解するには、ベース プラグイン ローダー クラスで何が起るのか調べる必要があります。プラグイン ローダー クラスは読み込まれるプラグインのフルパスを引数とする virtual 関数 `Open()` を宣言します。特定のプラグイン クラス ローダーでの `Open()` 関数の実装では、ベース プラグイン ローダーの protected 関数 `open()` を呼び出します; このベース `open()` 関数は要求された基本的なプラグインの関数それぞれのアドレスを見つけようとします; 各関数のアドレスが取得されると、いくつかのチェックが実行されます:

1. プラグイン `GetKicadPluginClass()` が呼び出されると、プラグイン ローダーが提供するプラグイン クラス文字列との比較が行われます; もしこれらの文字列が一致しなければ、開かれたプラグインはこのプラグイン ローダー インスタンス向けのものではありません。
2. プラグイン `GetClassVersion()` はプラグインが実装しているプラグイン クラス API Version を取得するために呼び出されます。
3. プラグイン ローダー virtual 関数 `GetLoaderVersion()` はローダーが実装しているプラグイン クラス API Version を取得するために呼び出されます。

4. プラグインとローダーが報告するプラグイン クラス API Version は同じメジャー バージョン番号を持っている必要があります。もし違っていれば互換性はないと考えられます。これは最も基本的なバージョンのテストで、ベース プラグイン ロードерによって強制的に実行されます。
5. プラグイン `CheckClassVersion()` はプラグイン ロードーのプラグイン クラス API Version information を呼び出します; もしプラグインが与えられたバージョンをサポートしていれば、成功を意味する `true` が返ります。成功した場合、ローダーは `GetKicadPluginName()` と `GetPluginVersion()` の結果を基に `PluginInfo` 文字列を作成し、plugin loading procedure がプラグイン ロードーの `Open()` 実装部の中で続けて実行されます。

## API: 3D プラグイン クラス

3D プラグイン クラスはヘッダ ファイル `3d_plugin.h` で宣言されており、[Plugin Class: PLUGIN\\_3D](#) 内で記述されるような必要とされるプラグイン関数を拡張します。対応するプラグイン ロードーは `pluginldr3D.cpp` 内で定義され、ローダーは要求された API 関数に加えて public 関数を提供します。

```
/* フル パス "aFullFileName" で指定されたプラグインを開く */
bool Open( const wxString& aFullFileName );

/* 現在開かれているプラグインを閉じる */
void Close( void );

/* このプラグイン ロードーが提供する プラグイン クラス API Version を取得する */
void GetLoaderVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Revision, unsigned char* Patch ) const;
```

要求された 3D プラグイン クラス 関数は、以下の関数を経由して公開されます:

```

/* プラグイン クラスを返す。プラグインが読み込まれていなければ NULL */
char const* GetKicadPluginClass( void );

/* プラグインが読み込まれていなければ false を返す */
bool GetClassVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/* クラスのバージョン チェックが失敗またはプラグインが読み込まれていなければ、false を返す */
bool CheckClassVersion( unsigned char Major, unsigned char Minor,
    unsigned char Patch, unsigned char Revision );

/* プラグイン名を返す。プラグインが読み込まれていなければ NULL */
const char* GetKicadPluginName( void );

/*
    プラグインが読み込まれていない場合、false を返す。これ以外は、
    引数は GetPluginVersion() の戻り値に含まれる
*/
bool GetVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/*
    プラグインが読み込まれていない場合、空文字列を aPluginInfo にセットする。
    これ以外は、下記フォームの文字列が aPluginInfo にセットされる：
    [NAME]:[MAJOR].[MINOR].[PATCH].[REVISION] ここで
    NAME = name provided by GetKicadPluginClass()
    MAJOR, MINOR, PATCH, REVISION = version information from
    GetPluginVersion()
*/
void GetPluginInfo( std::string& aPluginInfo );

```

典型的な状況では、ユーザーは以下のような使い方をするでしょう：

1. KICAD\_PLUGIN\_LDR\_3D のインスタンスを作成する。
2. 特定のプラグインを読み込むために `Open( "/path/to/myplugin.so" )` を呼ぶ。望み通りのプラグインが読み込まれたかどうか確かめるためには、戻り値をチェックする必要がある。
3. KICAD\_PLUGIN\_LDR\_3D で公開されているような、何れかの 3D プラグイン クラス コールを呼ぶ。
4. プラグインを閉じる (リンクを外す) ために `Close()` を呼ぶ。
5. KICAD\_PLUGIN\_LDR\_3D インスタンスを破棄する。

## シーングラフ クラス API

シーングラフ クラス API はヘッダ ファイル `ifsg_all.h` とそれに含まれるヘッダで定義されています。この API は、`ifsg_api.h` で定義されている名前空間 `S3D` にある幾つかのヘルパー ルーチンと `ifsg_*.h` ヘッダ 各種で定義されているラッパー クラスからなっています；ラッパーは、VRML2.0 のスタティック シーン グラフと互換があるシーン グラフ 構造体をまとめたものである、下層のシーン グラフ クラスをサポートします。ヘッダ、構造体、クラスおよびその public 関数は次の通りです：

## sg\_version.h

```
/*
 シーングラフ クラスのバージョン情報を定義する。
 シーングラフ クラスで使用される全てのクラスは本ヘッダを含む必要があり、
 また互換性を保証するため S3D::GetLibVersion() によって報告される
 バージョンに対してバージョン情報チェックを行わなければならない。
 */

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

## sg\_types.h

```
/*
 シーングラフ クラス タイプを定義する; これらのタイプは
 VRML2.0 ノード タイプと密接な関係がある。
 */

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
    };
};
```

`sg_base.h` ヘッダはシーングラフ クラスで使われる基本的なデータ型の宣言を含んでいる。

```

/*
    これは、各色が範囲 [0..1] の数を持つ
    VRML2.0 RGB モデルと同等な RGB
    色モデルである。
*/

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 );
};

/*
    SGVECTOR は点と同じく3つの成分 (x,y,z) を持っています; しかしながら
    ベクトルは保存された値が正規化されていることが保証されており、
    成分の値を直接操作できないようになっています。
*/
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 );
};

```



`IFSG_NODE` クラスは全てのシーングラフ ノードの基本クラスです。全てのシーングラフ オブジェクトはこのクラスの `public` 関数として実装されますが、いくつかのケースでは、ある関数は特定のクラスでは意味がないかも知れません。

```

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

    /**
     * Destroy 関数
     * このラッパーで保持されているシーングラフ オブジェクトを削除する
     */
    void Destroy( void );

    /**
     * Attach 関数
     * このラッパーに SGNODE* を関連付ける
     */
    virtual bool Attach( SGNODE* aNode ) = 0;

    /**
     * NewNode 関数
     * このラッパーに関連付ける新しいノードを作成する
     */
    virtual bool NewNode( SGNODE* aParent ) = 0;
    virtual bool NewNode( IFSG_NODE& aParent ) = 0;

    /**
     * GetRawPtr() 関数
     * 元々の内部 SGNODE ポインタを返す
     */
    SGNODE* GetRawPtr( void );

    /**
     * GetNodeType 関数
     * このノード インスタンスのタイプを返す
     */
    S3D::SGTYPES GetNodeType( void ) const;

    /**
     * GetParent 関数
     * このオブジェクトの親 SGNODE へのポインタを返す。
     * もしオブジェクトが親を持っていない (例. トップ レベル transform) か、
     * ラッパーが現在の SGNODE と関連付けられていない場合は NULL
     */
    SGNODE* GetParent( void ) const;

    /**
     * SetParent 関数
     * このオブジェクトの親 SGNODE をセットする。
     *
     * @param aParent [in] はセットしたい親ノード
     * @return 関数が成功した場合は true; 与えられた
     * ノードが派生オブジェクトへのペアレントを許されて
     * いない場合は false
     */
    bool SetParent( SGNODE* aParent );

    /**
     * GetNodeTypeName 関数
     * ノード タイプを表すテキストを返す。
     * もし何故かノードが無効なタイプであれば NULL

```

IFSG\_TRANSFORM は VRML2.0 Transform ノードと同等です;これは、子ノード IFSG\_SHAPE と IFSG\_TRANSFORM および参照ノード IFSG\_SHAPE と IFSG\_TRANSFORM を大量に含んでいます。有効なシーングラフは、ルートに単一の IFSG\_TRANSFORM オブジェクトを持っている必要があります。

#### *ifsg\_transform.h*

```
/**
 * IFSG_TRANSFORM クラス
 * VRML 互換 TRANSFORM ブロック クラス SCENEGRAPH のラッパー
 */

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 );

    /* いくつかのベース クラス関数はここにありません */
};
```

IFSG\_SHAPE は VRML2.0 シェイプ ノードと同等です;これは、単独の子ノードあるいは参照ノード FACESET を含んでいる必要があります。また、単独の子ノードあるいは参照ノード APPEARANCE を含んでいても構いません。

#### *ifsg\_shape.h*

```
/**
 * IFSG_SHAPE クラス
 * SGSHAPE クラスのラッパー
 */

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

    /* いくつかのベース クラス関数はここにありません */
};
```

IFSG\_APPEARANCE は VRML2.0 Appearance ノードと同等です。しかしながら、今のところ Material ノードを含んだ Appearance ノードと同じ意味しか持っていません。

```

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 );

    /* いくつかのベース クラス関数はここにありません */

    /* 以下の関数は appearance ノードでは意味がありません。
       また、常に失敗を表すコードを返します。

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

```

IFSG\_FACESET は IndexedFaceSet ノードを含んだ VRML2.0 Geometry ノードと同等です。これは、単独の子ノードあるいは参照ノード COORDS、単独の子ノード COORDINDEX、単独の子ノードあるいは参照ノード NORMALS を含んでいる必要があります。さらに、単独の子ノードあるいは参照ノード COLORS を含んでいても構いません。ユーザーが面に法線を配置できるように、単純化された法線を計算する関数が用意されています。VRML2.0 同等と異なっている部分は次のとおりです:

1. 法線は常に頂点毎である。
2. 色は常に頂点毎である。
3. 座標値の集合は三角形の面だけを記述しなければならない。

## *ifsg\_faceset.h*

```
/**
 * IFSG_FACESET クラス
 * SGFACESET クラスのラッパー
 */

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

    bool CalcNormals( SGNODE** aPtr );

    /* いくつかのベース クラス関数はここにありません */
};
```

## *ifsg\_coords.h*

```
/**
 * IFSG_COORDS クラス
 * SGCOORDS のラッパー
 */

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 );

    /* いくつかのベース クラス関数はここにありません */

    /* 以下の関数は coords ノードでは意味がありません。
       また、常に失敗を表すコードを返します。

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

IFSG\_COORDINDEX は三角形の面のみを記述しなければならない点を除いて VRML2.0 coordIdx[] set と同等です。これは座標値の合計数が3で割り切れることを意味しています。

## ifsg\_coordindex.h

```
/**
 * IFSG_COORDINDEX クラス
 * SGCORDINDEX のラッパー
 */

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 );

    /* いくつかのベース クラス関数はここにありません */

    /* 以下の関数は coordindex ノードでは意味がありません。
       また、常に失敗を表すコードを返します。

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

IFSG\_NORMALS は VRML2.0 Normals ノードと同等です。

## *ifsg\_normals.h*

```
/**
 * IFSG_NORMALS クラス
 * SGNORMALS クラスのラッパー
 */

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 );

    /* いくつかのベース クラス関数はここにありません */

    /* 以下の関数は normals ノードでは意味がありません。
       また、常に失敗を表すコードを返します。

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

IFSG\_COLORS は VRML2.0 colors[] set と同等です。

## ifsg\_colors.h

```
/**
 * IFSG_COLORS クラス
 * SGCOLORS のラッパー
 */

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 );

    /* いくつかのベース クラス関数はここにありません */

    /* 以下の関数は normals ノードでは意味がありません。
       また、常に失敗を表すコードを返します。

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

残りの API 関数は、以下のように `ifsg_api.h` で定義されています:



```

namespace S3D
{
    /**
     * GetLibVersion 関数は、kicad_3dsg library の
     * バージョン情報を取得する
     */
    SGLIB_API void GetLibVersion( unsigned char* Major, unsigned char* Minor,
                                unsigned char* Patch, unsigned char* Revision );

    // SGNODE ポインタから情報を得るための関数
    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 );

    /**
     * CalcTriNorm 関数
     * 頂点 p1, p2, p3 で表される三角形の法線ベクトルを返す
     */
    SGLIB_API SGVECTOR CalcTriNorm( const SGPOINT& p1, const SGPOINT& p2, const SGPOINT& p3
    );

    /**
     * WriteCache 関数
     * バイナリ キャッシュ ファイルへ SGNODE ツリーを書き込む
     *
     * @param aFileName 書き込まれるファイル名
     * @param overwrite 既存のファイルへ上書きするためには true をセットしなければならない
     * @param aNode 書き込まれるノード ツリー内にあるノードのいずれか
     * @return 成功した場合は true
     */
    SGLIB_API bool WriteCache( const char* aFileName, bool overwrite, SGNODE* aNode,
                              const char* aPluginInfo );

    /**
     * ReadCache 関数
     * バイナリ キャッシュ ファイルを読み込み、SGNODE ツリーを作成する
     *
     * @param aFileName 読み込まれるバイナリ キャッシュ ファイル名
     * @return 失敗した場合は NULL、成功した場合はトップ レベル SCENEGRAPH ノードへのポインタ;
     * 必要なら、このノードを IFSG_TRANSFORM::Attach() 関数経由で
     * IFSG_TRANSFORM ラッパーと関連付けることも可能
     */
    SGLIB_API SGNODE* ReadCache( const char* aFileName, void* aPluginMgr,
                                bool (*aTagCheck)( const char*, void* ) );

    /**
     * WriteVRML 関数
     * VRML2 ファイルへ与えられたノードとそのサブノードを書き込む
     *
     * @param filename 出力ファイル名
     * @param overwrite 既存の VRML ファイルへ上書きするためには true をセットしなければならない
     * @param aTopNode VRML シーンで表される SCENEGRAPH オブジェクトへのポインタ
     * @param reuse VRML DEF/USE 機能を使用する場合には true をセットしなければならない
     * @return 成功した場合は true
     */
    SGLIB_API bool WriteVRML( const char* filename, bool overwrite, SGNODE* aTopNode,
                              bool reuse, bool renameNodes );
}

```

シーングラフ API の実際の使用例は、[Advanced 3D Plugin tutorial](#) の記述と KiCad VRML1, VRML2, X3D パーサーを参照のこと。