

Плагины KiCad

Table of Contents

Введение в систему плагинов KiCad	2
Классы плагинов	2
Примеры: Класс 3D-плагинов	4
Простой 3D-плагин	4
Сложный 3D-плагин	9
Интерфейс программирования приложений (API)	12
API класса плагинов	12
API калсса графа сцены	14

Система плагинов KiCad

Авторские права

Авторские права © 2016 на данный документ принадлежит его разработчикам (соавторам), перечисленным ниже. Документ можно распространять и/или изменять в соответствии с правилами лицензии GNU General Public License (<http://www.gnu.org/licenses/gpl.html>), версии 3 или более поздней, или лицензии типа Creative Commons Attribution License (<http://creativecommons.org/licenses/by/3.0/>), версии 3.0 или более поздней.

Все торговые знаки этого руководства принадлежат его владельцам.

Соавторы

Cirilo Bernardo

Перевод

Барановский Константин <baranovskiykonstantin@gmail.com>, 2016-2019

Обратная связь

Оставить свои комментарии или замечания можно на следующих ресурсах:

- О документации KiCad: <https://gitlab.com/kicad/services/kicad-doc/issues>
- О программном обеспечении KiCad: <https://gitlab.com/kicad/code/kicad/issues>
- О переводе программного обеспечения KiCad: <https://gitlab.com/kicad/code/kicad-i18n/issues>

Дата публикации

29 января 2016 года

Введение в систему плагинов KiCad

Система плагинов KiCad - это специальный механизм для расширения возможностей KiCad, использующий динамические библиотеки. Одно из основных преимуществ использования плагинов — это отсутствие необходимости заново собирать весь проект KiCad в процессе разработки плагина. На деле, плагины можно построить с применением очень малого набора заголовочных файлов из всего дерева исходного кода KiCad. Освобождение от необходимости сборки KiCad в процессе разработки, здорово увеличивает продуктивность благодаря тому, что разработчик компилирует только тот код, который непосредственно относится к проектируемому плагину, что, в свою очередь, уменьшает время на каждую сборку в процессе тестирования.

Изначально, система плагинов была разработана для реализации предварительного просмотра 3D-моделей и обеспечения поддержки большего количества форматов 3D-моделей, без необходимости вносить серьезные изменения в исходный код KiCad для каждого нового поддерживаемого формата. Механизм плагинов со временем был обобщен и, таким образом, в будущем разработчики смогут создавать плагины различных классов. На данный момент в KiCad реализованы только 3D-плагины, но планируется добавить класс плагинов для печатных плат, который позволит пользователям реализовать импорт и экспорт данных.

Классы плагинов

Плагины делятся на классы, так как каждый из них решает проблемы определённой области и, поэтому, требует отдельного интерфейса к данной области. Например, плагины 3D-моделей загружают трёхмерные данные из файлов и преобразуют их в формат, который может быть показан в программе 3D-просмотра, в то время как плагин импорта/экспорта печатных плат должен принимать данные о печатных платах и экспортировать их в другой формат электрических или механических данных для KiCad. На данный момент разработан только класс 3D-плагинов и именно на нём будет сосредоточено внимание в этом документе.

Для реализации нового класса плагина необходимо добавить код в дерево исходного кода KiCad, который будет управлять загрузкой плагина. В файле `plugins/ldr/pluginldr.h`, из исходного кода KiCad, определён базовый класс для всех загрузчиков плагинов. В этом классе определены общие функции, которые должны присутствовать в любом из плагинов KiCad (шаблонный код), а их реализация будет выполнять основные проверки на совместимость версий между загрузчиком и доступными плагинами. Заголовочный файл `plugins/ldr/3d/pluginldr3d.h` определяет загрузчик для класса 3D-плагинов. Загрузчик отвечает за загрузку полученного плагина и делает его функции доступными для KiCad. Каждый экземпляр загрузчика плагинов предоставляет реализацию конкретного плагина и выступает в качестве прозрачного моста между `kicad` и функциями плагина. Для поддержки плагинов нужно не только добавить код загрузчика в исходный код KiCad, ещё нужен код для обнаружения плагинов и код для вызова функций плагина через загрузчик. В случае с 3D-плагином, обнаружение и вызов функций, вместе, реализовано в классе `S3D_CACHE`.

Разработчикам плагина не нужно разбираться в деталях исходного кода KiCad для управления им, если новый класс плагинов уже разработан. Для реализации плагина нужно лишь определить функции, объявленные в соответствующем классе плагинов.

Заголовочный файл `include/plugins/kicad_plugin.h` объявляет основные функции, обязательные для всех плагинов KiCad. Эти функции определяют имя класса плагина и имя данного плагина,

возвращают информацию о версии API класса, информацию о версии самого плагина и проверяют их на совместимость. Вкратце об этих функциях:

```
/* Возвращает имя класса плагина в виде строки UTF-8 */
char const* GetKicadPluginClass( void );

/* Возвращает информацию о версии API класса плагина */
void GetClassVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/*
    Возвращает истину, если реализованная проверка версий в плагине
    определила, что указанный API класса -- совместим.
*/
bool CheckClassVersion( unsigned char Major,
    unsigned char Minor, unsigned char Patch, unsigned char Revision );

/* Возвращает имя данного плагина, например, "PLUGIN_3D_VRML" */
const char* GetKicadPluginName( void );

/* Возвращает информацию о версии данного плагина */
void GetPluginVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );
```

Класс плагинов PLUGIN_3D

В заголовочном файле include/plugins/3d/3d_plugin.h объявляются функции, которые должны быть реализованы в во всех 3D-плагинах, а также указано несколько функций, которые пользователь не должен изменять. Следующие функции не должны реализовываться пользователем:

```
/* Возвращает имя класса плагина -- "PLUGIN_3D" */
char const* GetKicadPluginClass( void );

/* Возвращает информацию о версии API класса PLUGIN_3D */
void GetClassVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/*
    Выполняет обычную проверку версии, реализованную разработчиками
    загрузчика класса плагинов PLUGIN_3D, и возвращает истину, если
    проверка успешно пройдена
*/
bool CheckClassVersion( unsigned char Major, unsigned char Minor,
    unsigned char Patch, unsigned char Revision );
```

Следующие функции должны быть реализованы пользователем:

```

/*
    Возвращает количество строк с расширениями, которые поддерживаются
    плагином
*/
int GetNExtensions( void );

/*
    Возвращает запрошенную строку с расширением; доступны значения от 0 до
    GetNExtensions() - 1
*/
char const* GetModelExtension( int aIndex );

/*
    Возвращает общее количество фильтров типов файлов, которые
    поддерживаются плагином
*/
int GetNFilters( void );

/*
    Возвращает запрошенный фильтр типов файлов; доступны значения от 0 до
    GetNFilters() - 1
*/
char const* GetFileFilter( int aIndex );

/*
    Возвращает истину, если плагин может отобразить данный тип 3D-модели.
    В некоторых случаях, плагин не может предоставить визуальную модель
    и должен вернуть ложь.
*/
bool CanRender( void );

/*
    Загрузить указанную модель и вернуть указатель на данные её визуального
    представления
*/
SCENEGRAPH* Load( char const* aFileName );

```

Примеры: Класс 3D-плагинов

Этот раздел содержит описание двух очень простых плагинов из класса PLUGIN_3D и проведёт пользователя от настройки до сборки кода.

Простой 3D-плагин

Этот пример проведёт пользователя через весь процесс разработки очень простого 3D-плагина под именем "PLUGIN_3D_DEMO1". Цель этого примера — показать конструкцию элементарного 3D-плагина, который не делает ничего, кроме предоставления некоторых фильтров типов файлов, что позволит пользователям KiCad отфильтровать файлы в процессе выбора 3D-моделей. Показанный здесь код, является необходимым минимумом для любого 3D-плагина и может быть использован как шаблон для создания более функциональных плагинов.

В процессе сборки демонстрационного проекта понадобится следующее:

- [CMake](#)
- Заголовочные файлы плагина KiCad
- Библиотека графа сцены KiCad (kicad_3dsg)

Для автоматического обнаружения заголовочных файлов KiCad и библиотеки нужно воспользоваться скриптом FindPackage на CMake. Скрипт, приведённый в этом примере, должен работать в Linux и MSWindows, если соответствующие заголовочные файлы установлены в `${KICAD_ROOT_DIR}/kicad`, а библиотека графа сцены — в `${KICAD_ROOT_DIR}/lib`.

Для начала создайте каталог для проекта и скрипт FindPackage:

```

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" )

    # попытка извлечь информацию о версии из файла 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" )

        # извлечение строки "#define KICADSG_VERSION*"
        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()

        # привести неуказанные чатси версии к нулю
        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 и его заголовочные файлы для плагина должны быть установлены. Если они установлены в пользовательский каталог или в `/opt` в Linux, или используется Windows, то нужно определить переменную среды `KICAD_ROOT_DIR`, которая будет указывать на каталог `kicad`, содержащий каталоги `include` и `lib`. Для OSX, показанный здесь скрипт `FindPackage`, возможно, придётся немного подкорректировать.

Для настройки и сборки кода примера будет использоваться CMake, создайте файл скрипта `CMakeLists.txt`:

```
cd ${DEMO_ROOT}
cat > CMakeLists.txt << _EOF
# указать имя проекта
project( PLUGIN_DEMO )

# проверить, установлена ли нужная версия CMake со всеми нужными свойствами
cmake_minimum_required( VERSION 2.8.12 FATAL_ERROR )

# указать CMake место для поиска скрипта FindKICAD
set( CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/CMakeModules )

# попытка найти установленные заголовочные файлы и библиотеку KiCad
# и определить переменные:
#     KICAD_INCLUDE_DIR
#     KICAD_LIBRARY
find_package( KICAD 1.0 REQUIRED )

# добавить каталог заголовочных файлов kicad к путям поиска компилятора
include_directories( ${KICAD_INCLUDE_DIR}/kicad )

# создать плагин с именем s3d_plugin_demo1
add_library( s3d_plugin_demo1 MODULE
    src/s3d_plugin_demo1.cpp
)

_EOF
```

Первый демонстрационный проект очень прост. Он состоит из единственного файла без каких-либо внешних зависимостей (помимо зависимостей компилятора). Начнём с создания каталога для исходного кода:

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

Теперь создайте файл исходного кода для самого плагина:

s3d_plugin_demo1.cpp

```
#include <iostream>

// в заголовочном файле 3d_plugin.h объявлены функции, обязательные для
// 3D-плагинов
#include "plugins/3d/3d_plugin.h"

// укажите информацию о версии данного плагина; не путайте это с
// версией класса плагина, которая указана в 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

// реализуйте функцию, которая предоставляет пользователям имя плагина
const char* GetKicadPluginName( void )
{
    return "PLUGIN_3D_DEMO1";
}

// реализуйте функцию, которая предоставляет пользователям версию плагина
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;
}

// количество поддерживаемых расширений; на системах *NIX расширения
// указываются дважды - одно в нижнем регистре, второе - в верхнем
#ifdef _WIN32
    #define NEXTS 7
#else
    #define NEXTS 14
#endif

// количество поддерживаемых фильтров типов файлов
#define NFILS 5

// определите строки с расширениями и фильтрами, которые поддерживает
// данный плагин
static char ext0[] = "wr1";
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
```


Данный файл исходного кода содержит минимальный набор всех необходимых элементов для реализации 3D-плагина. Этот плагин не производит никаких данных для рендеринга моделей, но может дополнить KiCad списком поддерживаемых расширений файлов моделей и фильтров типов файлов в диалоговом окне выбора 3D-моделей. К тому же, в KiCad строка расширения используется для выбора плагинов, с помощью которых можно загрузить выбранные модели. Например, если выбрано расширение `wrl`, то KiCad будет вызывать каждый плагин, который объявил о поддержке этого расширения, до тех пор, пока один из них не вернёт данные визуализации. Фильтры файлов, предоставленные каждым из плагинов, передаются в диалоговое окно выбора 3D-моделей, чтобы улучшить процесс поиска.

Для сборки плагина:

```
cd ${DEMO_ROOT}
# экспортируйте KICAD_ROOT_DIR, если понадобится
mkdir build && cd build
cmake .. && make
```

Плагин будет построен, но не установлен. Можно скопировать его в каталог, в котором хранятся плагины, установленные вместе с `kicad`, если желаете, чтобы он был загружен.

Сложный 3D-плагин

Этот пример проведёт пользователя через весь процесс разработки 3D-плагина под именем "PLUGIN_3D_DEMO2". Цель этого примера — показать конструкцию элементарного графа сцены, который `kicad` сможет отобразить. Плагин должен поддерживать тип файлов `txt`. Кроме этого, данный файл должен существовать, чтобы менеджер кэша смог запустить плагин. Содержимое файла не обрабатывается плагином, вместо этого, он просто создаёт граф сцены, содержащий пару тетраэдров. В данном примере предполагается, что первый пример был завершен и файлы скриптов `CMakeLists.txt` и `FindKICAD.cmake` были созданы.

Поместите новый файл исходного кода в тот же каталог, в котором находится файл исходного кода из предыдущего примера, и дальше будет дополнен уже имеющийся файл `CMakeLists.txt`, чтобы построить этот пример. Так как данный плагин будет создавать граф сцены для KiCad, нужно подключить библиотеку графов сцены из KiCad — `kicad_3dsg`. Эта библиотека предоставляет набор классов, которые можно использовать для построения объекта графа сцены. Объект графа сцены — это вспомогательный формат данных визуализации, который используется менеджером кэша трехмерных данных (3D Cache Manager). Все плагины, поддерживающие модель визуализации должны преобразовывать данные моделей в граф сцены с помощью библиотеки.

Первым делом нужно дополнить `CMakeLists.txt` для сборки примера проекта:

```
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
```

Теперь перейдите в каталог с исходным кодом и создайте новый файл:

```
cd ${DEMO_SRC}
```

s3d_plugin_demo2.cpp

```
#include <cmath>
// объявления из класса 3D-плагинов
#include "plugins/3d/3d_plugin.h"
// интерфейс для работы с библиотекой графа сцены из KiCad
#include "plugins/3dapi/ifsg_all.h"

// информация о версии для данного плагина
#define PLUGIN_3D_DEMO2_MAJOR 1
#define PLUGIN_3D_DEMO2_MINOR 0
#define PLUGIN_3D_DEMO2_PATCH 0
#define PLUGIN_3D_DEMO2_REVNO 0

// предоставляет имя данного плагина
const char* GetKicadPluginName( void )
{
    return "PLUGIN_3D_DEMO2";
}

// предоставляет версию данного плагина
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;
}

// количество поддерживаемых расширений
#ifdef _WIN32
#define NEXTS 1
#else
#define NEXTS 2
#endif

// количество поддерживаемых фильтров
#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
```

Интерфейс программирования приложений (API)

Плагины создаются путём реализации интерфейса программирования приложений (Application Programming Interface — API). Каждый класс плагинов имеет свой уникальный API и в приведённых примерах 3D-плагинов была показана реализация API для класса 3D-плагинов, согласно объявлениям из заголовочного файла `3d_plugin.h`. Кроме того, плагины могут использовать дополнительный API, объявленный в исходном коде KiCad. В случае с 3D-плагинами, все те плагины, что поддерживают визуализацию моделей, должны взаимодействовать используя API графов сцены, который объявлен в заголовочном файле `ifsg_all.h` и вложенных в него.

В этом разделе описываются детали API доступных классов плагинов и других API из KiCad, которые могут потребоваться для реализации новых классов.

API класса плагинов

На данный момент доступен только один класс плагинов для KiCad — это класс 3D-плагинов. Все классы плагинов для KiCad должны реализовывать основной набор функций, объявленный в заголовочном файле `kicad_plugin.h`. Эти объявления можно рассматривать как базовый класс плагинов KiCad. Но на самом деле, реализации базового класса плагинов для KiCad не существует, эти заголовочные файлы присутствуют только для того, чтобы убедиться в том, что разработчики реализуют данные функции в каждом новом плагине.

В самом KiCad, каждый экземпляр загрузчика плагина реализует тот же API, что и плагин, так как этот загрузчик предоставляет все возможности данного класса. Это достигается тем, что класс загрузчика плагинов предоставляет открытый интерфейс, содержащий такие же имена функций, что и в реализации самого плагина. Список параметров может отличаться, чтобы можно было уведомить пользователя о возникновении каких-либо проблем, например, о том, что плагин не удалось загрузить. В процессе работы, загрузчик использует сохранённые указатели на каждую из функций API для их дальнейшего вызова по требованию пользователя.

API: базовый класс плагинов KiCad

Базовый класс плагинов KiCad определён в заголовочном файле `kicad_plugin.h`. Этот заголовочный файл должен подключаться ко всем другим классам плагинов. Для примера, посмотрите на объявления в заголовочном файле `3d_plugin.h` для класса 3D-плагинов. Прототипы этих функций кратко описаны в разделе [Классы плагинов](#). В `pluginldr.cpp` показано как реализуется API базового загрузчика.

Чтобы понять назначение обязательных функций из заголовочного файла базового класса плагинов, нужно рассмотреть, что происходит при загрузке этих плагинов. В классе загрузчика объявляется виртуальная функция `Open()`, в которую передаётся полный путь к загружаемому плагину. В реализации функции `Open()` каждого конкретного класса загрузчика вызывается защищённая (protected) функция `open()` из базового загрузчика. Эта базовая функция `open()` пытается найти адреса каждой из обязательных функций базового плагина. Как только адреса для каждой из функций будут получены, начнётся выполнение следующих проверок:

1. Вызывается функция плагина `GetKicadPluginClass()` — возвращаемый результат сравнивается со значением из загрузчика для данного класса плагинов. Если значения не соответствуют,

значит этот плагин не предназначен для работы с данным загрузчиком.

2. Вызывается функция плагина `GetClassVersion()` — возвращается версия API класса плагина, реализованная данным плагином.
3. Вызывается функция загрузчика `GetLoaderVersion()` — возвращается версия API класса плагина, реализованная данным загрузчиком.
4. В версиях API, полученных от плагина и загрузчика, должен совпадать главный номер версии (Major Version number), иначе считается что плагин с загрузчиком не совместимы. Это самая простая проверка на соответствие версий и выполняется она базовым загрузчиком плагина.
5. Вызывается функция плагина `CheckClassVersion()` — в функцию передаётся версия API класса плагинов, полученная от загрузчика. Если плагин поддерживает указанную версию — возвращается истина (`true`), подтверждая совместимость. В таком случае загрузчик создаёт строку `PluginInfo` путём объединения результатов двух функций `GetKicadPluginName()` и `GetPluginVersion()`, и затем, процесс загрузки плагина продолжается с помощью функции `Open()` загрузчика.

API: класс 3D-плагинов

Класс 3D-плагинов объявлен в заголовочном файле `3d_plugin.h`. Помимо обязательных функций, в нем присутствуют дополнительные, их описание содержится в разделе [Класс плагинов: PLUGIN_3D](#). Загрузчик для этого класса плагинов определён в `pluginldr3d.cpp` и помимо обязательных функций API, реализует следующие дополнительные общедоступные функции:

```
/* Открыть плагин, указанный в виде полного пути "aFullFileName" */
bool Open( const wxString& aFullFileName );

/* Закрыть, открытый в данный момент, плагин.*/
void Close( void );

/* Получить версию API класса плагинов, реализованную данным загрузчиком */
void GetLoaderVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Revision, unsigned char* Patch ) const;
```

Необходимые функция из класса 3D-плагинов выявляются с помощью следующих функций:

```

/* возвращает имя класса плагинов или NULL, если плагин не загружен */
char const* GetKicadPluginClass( void );

/* возвращает ложь, если плагин не загружен */
bool GetClassVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/* возвращает ложь, если версия класса не совместима или плагин не загружен */
bool CheckClassVersion( unsigned char Major, unsigned char Minor,
    unsigned char Patch, unsigned char Revision );

/* возвращает имя плагина или NULL, если плагин не загружен */
const char* GetKicadPluginName( void );

/*
    возвращает ложь, если плагин не загружен, в противном случае
    в параметрах будет содержаться результат функции GetPluginInfo()
*/
bool GetVersion( unsigned char* Major, unsigned char* Minor,
    unsigned char* Patch, unsigned char* Revision );

/*
    если плагин не загружен, устанавливает значение переменной aPluginInfo
    в виде пустой строки; в противном случае, значение образуется следующим
    образом:
    [NAME]:[MAJOR].[MINOR].[PATCH].[REVISION]
    где:
    NAME = имя, полученное из GetKicadPluginClass()
    MAJOR, MINOR, PATCH, REVISION = информация о версии, полученная из
    GetPluginVersion()
*/
void GetPluginInfo( std::string& aPluginInfo );

```

В общем случае, пользователь должен выполнить следующее:

1. Создать объект класса `KICAD_PLUGIN_LDR_3D`.
2. Вызвать функцию `Open("/path/to/myplugin.so")`, чтобы открыть нужный плагин. Возвращаемое значение нужно проверять, чтобы убедиться в успешной загрузке плагина.
3. Вызвать любую функцию из класса 3D-плагинов, обнаруженную в `KICAD_PLUGIN_LDR_3D`.
4. Вызвать `Close()` чтобы закрыть (выгрузить) плагин.
5. Удалить объект класса `KICAD_PLUGIN_LDR_3D`.

API класса графа сцены

API класса графа сцены определён в заголовочном файле `ifsg_all.h` и вложенных в него. API содержит несколько дополнительных методов, объявленных в пространстве имён (namespace) `S3D` в файле `ifsg_api.h` и вспомогательных классов, объявленных в различных заголовочных файлах `ifsg_*.h`. Вспомогательные классы поддерживают основные форматы графов сцены, которые вместе образуют структуру графов, совместимую с VRML2.0. Заголовочные файлы, структуры, классы и их общедоступные функции рассмотрены далее:

sg_version.h

```
/*
    Определение информации о версии класса графа сцены.
    Все плагины, использующие класс графа сцены должны включать этот
    заголовочный файл и проверять версию каждый раз, используя результат
    функции S3D::GetLibVersion(), для подтверждения совместимости
*/

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

sg_types.h

```
/*
    Определение типов для класса графа сцены; эти типы
    максимально приближены к типам узлов VRML2.0.
*/

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
    };
};
```

Заголовочный файл `sg_base.h` состоит из объявлений основных типов данных, которые используются в классах графов сцены.

```

/*
    Эта модель RGB-цвета аналогична модели VRML2.0, где каждому
    цвету присваивается значение в диапазоне [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 );
};

/*
    SGVECTOR имеет 3 составляющие (x,y,z), подобно точке, но
    вектор содержит нормализованные значения и предотвращает
    их непосредственное изменение.
*/
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 );
};

```


Класс `IFSG_NODE` — базовый класс для всех узлов графа сцены. Все объекты графа сцены реализуют общедоступные функции этого класса, хотя не все они используются некоторыми объектами.

```

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

    /**
     * Функция Destroy
     * удаляет данный объект графа сцены
     */
    void Destroy( void );

    /**
     * Функция Attach
     * связывает полученный SGNODE* с этим объектом
     */
    virtual bool Attach( SGNODE* aNode ) = 0;

    /**
     * Функция NewNode
     * создаёт новый узел и связывает его с этим объектом
     */
    virtual bool NewNode( SGNODE* aParent ) = 0;
    virtual bool NewNode( IFSG_NODE& aParent ) = 0;

    /**
     * Функция GetRawPtr()
     * возвращает указатель непосредственно на SGNODE
     */
    SGNODE* GetRawPtr( void );

    /**
     * Функция GetNodeType
     * возвращает тип узла данного объекта
     */
    S3D::SGTYPES GetNodeType( void ) const;

    /**
     * Функция GetParent
     * возвращает указатель на родительский SGNODE для этого объекта
     * или NULL, если объект не имеет родителей (т.е. является
     * объектом преобразования верхнего уровня) или, когда данный
     * объект не связан с SGNODE.
     */
    SGNODE* GetParent( void ) const;

    /**
     * Функция SetParent
     * присваивает родительский SGNODE для данного объекта.
     *
     * @param aParent [входящий] желаемый родитель узла
     * @return true если операция выполнена; false -
     *         полученный узел не может быть родителем для
     *         данного объекта.
     */
    bool SetParent( SGNODE* aParent );

    /**
     * Функция GetNodeTypename
     * возвращает тип узла в виде текста или NULL, если узел,

```

IFSG_TRANSFORM подобен узлу Transform из VRML2.0. Он может содержать любое количество дочерних или связанных узлов IFSG_SHAPE и IFSG_TRANSFORM. Корректный граф сцены должен иметь только один объект IFSG_TRANSFORM в качестве корневого.

ifsg_transform.h

```
/**
 * Класс IFSG_TRANSFORM
 * это оболочка для совместимости с блоком TRANSFORM из графа сцены VRML
 */

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 );

    /* прочие функции базового класса, которые здесь не рассматриваются */
};
```

IFSG_SHAPE подобен узлу Shape из VRML2.0. Он должен содержать единственный дочерний узел FACESET или ссылку на него. Также, может содержать дочерний узел APPEARANCE или ссылку на него.

ifsg_shape.h

```
/**
 * Класс IFSG_SHAPE
 * оболочка для класса SGSHAPE
 */

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

    /* прочие функции базового класса, которые здесь не рассматриваются */
};
```

IFSG_APPEARANCE подобен узлу Appearance из VRML2.0, но на данный момент, он реализован в соответствии с узлом Appearance, содержащим узел Material.

```

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 );

    /* прочие функции базового класса не показанные здесь */

    /* следующие функции не используются узлами Appearance
       и могут возвращать код ошибки

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

```

IFSG_FACESET подобен узлу Geometry из VRML2.0, который содержит узел IndexedFaceSet. Он должен состоять из одного дочернего узла COORDS или ссылки на него, одного дочернего узла COORDINDEX и одного дочернего узла NORMALS или ссылки на него. Дополнительно, он может содержать дочерний узел COLORS или ссылку на него. Элементарные функции операций над векторами предназначены помочь пользователям в связывании этих векторов с поверхностями. Далее указаны некоторые отличия от VRML2.0:

1. Векторы всегда относятся к вершинам.
2. Цвета всегда присваиваются вершинам.
3. Набор индексов координат должен описывать только треугольные грани.

ifsg_faceset.h

```
/**
 * Класс IFSG_FACESET
 * это оболочка для класса SGFACESET
 */

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

    bool CalcNormals( SGNODE** aPtr );

    /* прочие функции базового класса, которые здесь не рассматриваются */
};
```

ifsg_coords.h

```
/**
 * Класс IFSG_COORDS
 * это оболочка для 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 );

    /* прочие функции базового класса не показанные здесь */

    /* следующие функции не имеют значения для узлов
       координат и всегда возвращают значение ошибки

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

IFSG_COORDINDEX подобен массиву coordIdx[] из VRML2.0, он он должен описывать только стороны треугольников и, таким образом, общее количество индексов должно быть кратным 3-м.

ifsg_coordindex.h

```
/**
 * Класс IFSG_COORDINDEX
 * это оболочка для 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 );

    /* прочие функции базового класса не показанные здесь */

    /* следующие функции не имеют значения для узла
       индексов координат и всегда возвращают значение ошибки

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

`IFSG_NORMALS` соответствует узлу Normals из VRML2.0.

ifsg_normals.h

```
/**
 * Класс IFSG_NORMALS
 * это оболочка для класса SGNORMALS
 */

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 );

    /* прочие функции базового класса не показанные здесь */

    /* следующие функции не имеют значения для узла
       векторов и всегда возвращают значение ошибки

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

IFSG_COLORS подобен массиву colors[] из VRML2.0.

ifsg_colors.h

```
/**
 * Класс IFSG_COLORS
 * это оболочка для 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 );

    /* прочие функции базового класса не показанные здесь */

    /* следующие функции не имеют значения для узла
       векторов и всегда возвращают значение ошибки

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

Остальные функции API определены в `ifsg_api.h` и показаны далее:


```

namespace S3D
{
    /**
     * Функция GetLibVersion возвращает информацию о версии
     * библиотеки kicad_3dsg
     */
    SGLIB_API void GetLibVersion( unsigned char* Major, unsigned char* Minor,
                                unsigned char* Patch, unsigned char* Revision );

    // функции для извлечения информации по указателям SGNODE
    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 );

    /**
     * Функция CalcTriNorm
     * возвращает нормальный вектор для треугольника, описанного вершинами p1, p2, p3
     */
    SGLIB_API SGVECTOR CalcTriNorm( const SGPOINT& p1, const SGPOINT& p2, const SGPOINT& p3
    );

    /**
     * Функция WriteCache
     * записывает дерево SGNODE в бинарный файл кэша
     *
     * @param aFileName - название файла для записи
     * @param overwrite - должен содержать истину, если нужно перезаписать существующий файл
     * @param aNode - любой узел из дерева, который нужно записать
     * @return true при успешном завершении
     */
    SGLIB_API bool WriteCache( const char* aFileName, bool overwrite, SGNODE* aNode,
                              const char* aPluginInfo );

    /**
     * Функция ReadCache
     * считывает бинарный файл кэша и создает дерево SGNODE
     *
     * @param aFileName - имя бинарного файла кэша для считывания
     * @return NULL при сбое, в случае успеха - возвращает указатель на
     * узел верхнего уровня SCENEGRAPH;
     * если понадобится, этот узел можно связать с оболочкой IFSG_TRANSFORM
     * с помощью функции IFSG_TRANSFORM::Attach().
     */
    SGLIB_API SGNODE* ReadCache( const char* aFileName, void* aPluginMgr,
                                bool (*aTagCheck)( const char*, void* ) );

    /**
     * Функция WriteVRML
     * записывает переданный узел и его дочерние узлы в файл VRML2
     *
     * @param filename - имя файла для записи
     * @param overwrite - должен быть установлен в истину, чтобы перезаписать
     * существующий файл VRML
     * @param aTopNode - указатель на объект SCENEGRAPH, представляющий сцену VRML
     * @param reuse - должен быть установлен в истину, для использования
     * свойств VRML DEF/USE
     * @return true при успешном завершении
     */
}

```

Примеры реального использования API графа сцены можно посмотреть в [примере 3D-плагина DEMO2](#) и в исходных кодах KiCad — 3D-плагины для работы с файлами в форматах VRML1, VRML2 и X3D.