OPENFLIGHT API 15

DEVELOPER GUIDE, VOLUME 1

LEVELS 1 AND 2 : READ/WRITE

CONTENT CREATION

PRESAGIS

Contents

1

OpenFlight API Developer Guide Volume 1

Welcome to the Developer Guide, Volume 1 for OpenFlight API 15.

1

Introduction

The Presagis OpenFlight® API is a set of C header files and libraries that provides a programming interface to the OpenFlight® database format as well as the Creator modeling system. The API provides functions to read and/or modify existing databases and to create new databases. Using the API, you can create:

- Translators to and from the OpenFlight format.
- Real-time simulators and games.
- Modeling applications.
- Plug-ins that extend the functionality of Creator.

Why Use the API?

The API provides portable access to OpenFlight databases, without having to write code to deal with the OpenFlight disk file format. With the API, you can:

- Quickly read a file and extract just the information you want.
- Guarantee that the files you create are valid OpenFlight format and are supported within Presagis compatible software.
- Remain current with OpenFlight format version changes easily, by installing updates to the API.
- Create portable programs that are isolated from platform specific file storage issues, like byte ordering.

Levels of the API

The API is composed of four levels:

1. **Read**: Lets you examine the contents of OpenFlight database files. You can traverse and query elements of a database.

2. **Write**: Lets you create or modify OpenFlight database files. You can create or modify node hierarchies and save them to disk in OpenFlight format.

3. **Extensions**: Lets you extend or customize the OpenFlight format. You can add new attributes to existing nodes as well as create new node types of your own. Creator and the API treats your extension data just like standard OpenFlight data.

4. **Tools**: Lets you create plug-in tools to extend the capabilities of Creator to suit your specific needs.

All levels of the API are available on the *Windows* platform. Levels 1, 2 and 3 are available on the *Linux* platform.

OpenFlight Script

OpenFlight Script is a cross-platform Python Language binding to the C Language OpenFlight API. Based on the Python scripting language, OpenFlight Script provides all the functionality of the OpenFlight API levels 1 and 2. If you understand the concepts of the OpenFlight API, you will find programming in either the C Language API or OpenFlight Script very similar.

You can use OpenFlight Script to do anything you could otherwise do with the C Language API. In general, OpenFlight Script applications run more slowly than comparable tools written using the C Language API, so if performance is an issue, consider using the C Language API.

The OpenFlight API Document Set

The functionality of the OpenFlight API is described in two document sets, the *OpenFlight API Developer Guide* and the *OpenFlight API Reference Set*. The *OpenFlight API Developer Guide* describes "how to" use the *OpenFlight API* to create stand-alone applications and plug-ins that can access and

manipulate the contents of an OpenFlight database. The *OpenFlight API Reference* gives detailed descriptions of the data types, functions and symbols contained in the API.

The *OpenFlight API Developer Guide* is divided into two volumes. The first volume describes levels 1 and 2 (Read/Write) while the second volume describes levels 3 and 4 (Extensions/Tools). Both volumes are provided in PDF (Portable Document Format) and can be viewed in Adobe Acrobat® or any suitable PDF viewer. On Windows, the *OpenFlight API Developer Guide* is also provided in CHM (Compiled HTML Help) format.

The *OpenFlight API Reference Set* is also composed of two parts, the *OpenFlight API Reference* and the *OpenFlight Data Dictionary*. The *OpenFlight API Reference* describes the data types, functions and symbols of all levels of the API while the *OpenFlight Data Dictionary* lists the record and field codes associated to elements and attributes of an OpenFlight database. The *OpenFlight API Reference* is distributed in CHM (Windows only) and HTML (both platforms) and can be viewed with most browsers.

The OpenFlight API document set also includes:

- *OpenFlight API Installation Guide* - describes how to install the OpenFlight API.
- *OpenFlight API Release Notes* - describes what's new and different in the current version of the OpenFlight API.
- *Amendment to Software License Agreement* - Fill out this form if you intend to redistribute the OpenFlight Dynamic Link Libraries with your application.

About This Document

This document is Volume 1 of the *OpenFlight API Developer Guide* and describes levels 1 and 2, the Read/Write APIs. Please refer to the *OpenFlight API Reference* for supplementary information on the API.

To use levels 1 and 2 of the OpenFlight API, and this document, you should be comfortable with C programming and have some understanding of 3D modeling. Familiarity with Creator and the OpenFlight format is helpful, but not required.

Overview of the Read/Write API

The Read/Write API lets you read, modify and then save your changes to OpenFlight database files. It also lets you create new database files.

Program Environments

The end products generated by developers using the Read/Write API can be classified in one of two forms, stand-alone applications and plug-ins. Stand-alone applications execute in their own program environment. Plug-ins can execute in either the Creator program or the stand-alone program environment. The fact that plug-ins can execute in either of these program environments is very important to understand when using the Read/Write API.

All levels of the OpenFlight API are designed so that any API function can be called in either program environment, stand-alone application and Creator, alike. Even though all functions of the API can be called within either program environment, some functions may behave differently or even be disabled depending on the program environment in which they are invoked. This is better explained by the following examples.

Consider for example, the Read/Write API function **`mgSendMessage`** which reports a message to the user. It behaves differently depending on the program environment in which it was invoked. In both environments, it *delivers* a message to the user, but it does so in a way that is appropriate to the environment in which it was called. If invoked within the stand-alone program environment (by a plug-in or by the user application code itself), the message reported is printed to the console window. If called within the Creator environment (by a plug-in), the message reported is sent to the Creator Status Log. For functions like this, the semantics are the same in both environments, but the implementations may differ.

Also consider the function **`mgInit`** which initializes the OpenFlight API runtime for stand-alone applications. This function only makes sense if called within the stand-alone program environment. For this reason, it and functions like it that do not make sense in the Creator environment are disabled if called within Creator.

Unless specifically noted, all functions of the Read/Write API behave the same in both program environments. Those that do not, or are disabled in the Hardware and Software Prerequisites.

Installation

The *OpenFlight API Installation Guide* describes the installation procedure for both Windows and Linux platforms.

Installed Directories

On both Windows and Linux platforms, the OpenFlight API is installed into the Presagis Common Directory Organization. The OpenFlight API files are distributed into several subfolders under the root of this organization. This root folder is different depending on the platform but the subfolder structure below this root is the same for all platforms.

Windows Installation Root

The default location for the API installation directory on Windows is:

`C:\Presagis\Suite<suite>\OpenFlight_API`

where **`<suite>`** is the suite number of the API you have installed. For example, OpenFlight API 15 would be installed to:

`C:\Presagis\Suite15\OpenFlight_API`

On Windows, the OpenFlight API installer creates an environment variable, **`PRESAGIS_OPENFLIGHT_API,`** whose value is set to the "root" of this directory organization. This environment variable can be used in your Visual Studio Environment (or other IDE) accordingly.

Linux Installation Root

The default location for the API installation directory on Linux is:

`/usr/local/Presagis/Suite<suite>/OpenFlight_API`

where **`<suite>`** is the suite number of the API you have installed. For example, OpenFlight API 15 would be installed to:

/usr/local/Presagis/Suite15/OpenFlight_API

On Linux, the OpenFlight API installer includes a *tcsh* script file that you can execute to set the environment variable, **PRESAGIS_OPENFLIGHT_API,** to the "root" of this directory organization. This environment variable can then be used in your build environment accordingly. This script file is located at:

/usr/local/Presagis/Suite<suite>/OpenFlight_API/SOURCEME

and can be invoked as follows:

```
# cd /usr/local/Presagis/Suite<suite>/OpenFlight_API
# source SOURCEME
```

Note: You must run the **SOURCEME** script from the directory where it is located as it is dependent on the current directory.

This script file also sets the **LD_LIBRARY_PATH** environment variable to include the path to the OpenFlight API link libraries. This is necessary when you run applications you create with the API.

On both Windows and Linux, the subfolder structure below the installation root is as follows:

Subdirectory	Description
bin<platform>	Folders containing dynamic link libraries (DLLs).
bin<platform>/debug	Debug version of dynamic link libraries (DLLs)
bin<platform>/release	Release version of dynamic link libraries (DLLs)
docs	API documentation - see "Documentation" on page 13 for platform specific information on the documentation installed.
include	Header files containing definitions of API data types and functions
lib<platform>	API link library files
samples/apps	Sample programs

Subdirectory	Description
`samples/extensions`	Sample data extensions
`samples/plugins`	Sample Creator plug-ins **NOTE:** This folder is not included in the Linux distribution.
`samples/scripts`	Sample OpenFlight Scripts
`tools`	API binary tools `(ddbuild)`

Depending on which platform and compiler/architecture you choose to install, `<platform>` will be:

<platform>	If you installed...
`<empty>`	`Win32 - MSVS 2013 (VC12 - Win32)`
`_x64`	`Win64 - MSVS 2013 (VC12 - x64)`
`_vc9`	`Win32 - MSVS 2008 (VC9 - Win32)`
`_vc9_x64`	`Win64 - MSVS 2008 (VC9 - x64)`
`_vc8`	`Win32 - MSVS 2005 (VC8 - Win32)`
`_vc8_x64`	`Win64 - MSVS 2005 (VC8 - x64)`

Documentation

The documents included with the OpenFlight API include:

- *OpenFlight API Installation Guide*
- *OpenFlight API Release Notes*
- *OpenFlight API Developer Guide*
  - *Volume 1 (Read/Write)*
  - *Volume 2 (Extensions/Tools)*

- *OpenFlight API Reference Set*
  - *OpenFlight API Reference*
  - *OpenFlight Data Dictionary*

These files are installed in different locations depending on the platform.

Windows

For the Windows platform, the documentation for the OpenFlight API is included in the *Presagis Documentation Library*. The *Presagis Documentation Library* is Compiled HTML Help (CHM) format and includes documentation for all the Presagis products you have installed.

The *Presagis Documentation Library* is located at:

`$(PRESAGIS_ROOT)\docs\Presagis_MS.chm`

where **`PRESAGIS_ROOT`** is the root folder where your Presagis products are installed. To view the OpenFlight documentation, as well as all Presagis product documentation, open this file and browse to the OpenFlight API section in the viewer that is displayed. You can also access the *Presagis Documentation Library* via the Windows Start Menu.

In addition to the CHM versions of these documents, some documents are provided in alternative formats:

OpenFlight API Installation Guide and *OpenFlight API Release Notes* (PDF) - both located in:

`$(PRESAGIS_ROOT)\docs`

OpenFlight API Developer Guide (PDF) - located in:

`$(PRESAGIS_OPENFLIGHT_API)\docs\developerguide`

OpenFlight API Reference Set (HTML) - located in:

`$(PRESAGIS_OPENFLIGHT_API)\docs\reference\OpenFlight_API_Reference_Set.htm`

Linux

For the Linux platform, the documentation is located in:

`$(PRESAGIS_OPENFLIGHT_API)/docs`

The Installation Guide and Release Notes are located in the root (**docs**) folder.

The *OpenFlight API Developer Guide* (PDF format) is located in the folder:

$(PRESAGIS_OPENFLIGHT_API)/docs/developerguide

The *OpenFlight API Reference Set* (HTML format) is located in:

$(PRESAGIS_OPENFLIGHT_API)/docs/reference/OpenFlight_API_Reference_Set.htm

Overview of an OpenFlight File

Each OpenFlight file has a database node hierarchy that organizes the visual database into logical groupings, facilitating real-time functions such as field-of-view culling, LOD switching, and instancing. Each node type has data attributes specific to its function in the database.

The API provides a way for you to load an OpenFlight database into memory where it can be examined or modified. When in memory, the OpenFlight database is stored in a hierarchical "graph" structure. This in-memory graph structure is commonly referred to as the *OpenFlight Scene Graph*. At the root of this graph is the database header node. At the bottom are vertex nodes. In between are different organizational nodes, like group, object, level of detail, degree or freedom, polygon, mesh and others.

Each OpenFlight database also contains palettes comprised of entries for color, texture, material and other visual elements. An item contained in a palette is identified by a unique index and is referenced by a node in the hierarchy by that index. For example, a polygon node specifies which material is applied to it through its *material index* attribute. This material index attribute designates the specific item in the material palette applied to the polygon.

In memory, the API defines a generic (polymorphic) record called **mgrec** to store nodes in the OpenFlight Scene Graph, items in the palettes and several other OpenFlight constructs. Each **mgrec** has a record code assigned. This code allows you to identify "what kind of data" is stored in any **mgrec**. In the API this code is represented by the type **mgcode**. The **mgcode** for a database header node is **fltHeader** while a material palette item is identified by the

mgcode fltFMaterial. To query the **mgcode** assigned to a record, use **mgGetCode**. Note that once a record is created, its code is fixed and cannot be changed, hence there is no function to set the **mgcode** of a record.

For technical reasons in the API, when a record is created, it is permanently bound to a specific database. This implies that the node can only be attached to the specific database to which it is bound. To query the database to which a record is bound, use **mgRec2Db**.

The following section introduces the main node record types in the OpenFlight Scene Graph. Node records are discussed in more detail in "Node Records" on page 72. See "Palette Records" on page 102 for more information on palettes.

Node Record Types

The principal node types are as follows:

Database Header: Identified by the **fltHeader** record code, the database header node is always the first node in the file and represents the top of the database hierarchy and tree structure. There is one database header node per file. Common ancillary records include: color, material, texture, and vertex palette records.

Group: Identified by the **fltGroup** record code, a group node represents a logical subset of the database. Group nodes can be transformed (translated, rotated, scaled, etc.). The transformation applies to all its children. Groups can have child nodes and sibling nodes of any type except database header or vertex nodes. Common ancillary records include: comment, long ID, and transformation records.

Object: Identified by the **fltOjbect** record code, an object node contains a logical collection of polygons. It is effectively a low-level group node that offers some attributes distinct from the group node.

Polygon: Identified by the **fltPolygon** record code, a polygon node represents geometry. Its children are limited to a set of vertices that describe a polygon, line, or point. The front side of the polygon is viewed from a counterclockwise traversal of the vertices. Polygon attributes include color, texture, material, and transparency.

Nested Polygon: A nested polygon is a polygon node that is coplanar to and drawn on top of its nested parent. Nested polygons can themselves be nested parent node. This construct resolves the display of coplanar faces. Like polygons, nested polygons are also identified by the `fltPolygon` record code.

Mesh: Identified by the `fltMesh` record code, a mesh defines a set of related polygons, each sharing common attributes and vertices. Using a mesh, related polygons can be represented in a more compact format than would be required to represent each of the polygons separately. That is because only one copy of the polygon attributes is stored per mesh, regardless of how many actual polygons are represented in the mesh. A mesh is defined by a set of "polygon" attributes (color, material, texture, etc.), a common "vertex pool" and one or more "geometric primitives" that use the shared attributes and vertices. Each geometric primitive of a mesh represents either a triangle strip, triangle fan, quadrilateral strip or indexed polygon.

Curve: Identified by the `fltCurve` record code, a curve node can represent different types of geometric curves or curve segments using control points. Its children are limited to a set of vertices that are the control points of the curve.

Light source: Identified by the `fltLightSource` record code, a light source node is similar to a group, but also stores an index into the light source palette, as well as the position and direction of the light source. The light source position and direction are transformed by the transformations above it in the tree (if any).

Light point: Identified by the `fltLightPoint` record code, a light point node is used to represent light points or strings. Its children are limited to a set of vertices that define the position and color of the light(s). A light point may have one or more vertex children, each one representing a different point of light. A light string has one vertex child, a replication count (attribute ***`fltRepCnt`***) and transformation records. The vertex of a light string defines the position of the first light in the string. The replication count is the number of additional lights in the string while the transformation records represent the matrix to apply to get the positions of successive points in the string as measured from the previous light in the string.

Sound: Identified by the `fltSound` record code, a sound node is similar to a group, but serves as a location for a sound emitter. The emitter position is

the sound offset transformed by the transformations above it in the tree (if any).

Text: Identified by the `fltText` record code, a text node draws text with a specified font, without inserting the actual geometry into the database as polygon nodes. This is a leaf node and therefore cannot have any children.

Road: Identified by the `fltRoad` record code, a road node is the primary record of a road segment. The children of the road node represent the geometry and paths of the road segment.

Path: Identified by the `fltPath` record code, a path node is a child of a road node and describes a lane of the parent road node. By convention, the first path child of a road node is the center lane.

Vertex: Identified by the `fltVertex` record code, a vertex node represents a double precision 3D coordinate. Vertex attributes include x, y, z and optionally include color, alpha, normal and texture mapping information. Vertex nodes are the children of polygon nodes.

Morph vertex: A morph vertex is attached to a normal vertex in a level of detail such that when that level of detail switches in, the visual appearance of the normal vertex *morphs* (using interpolation) between the attributes of the morph vertex and those of the normal vertex. Using morph vertices in conjunction with level of detail transition distances, real-time systems can morph vertex attributes (position, normal, color, texture uv's, etc.) to provide smooth visual transitions between successive levels of detail. Like normal vertices, morph vertices are also identified by the `fltVertex` record code.

Degree of freedom: Identified by the `fltDof` record code, a degree-of-freedom (DOF) node is similar to a group with a predefined set of internal transformations. It specifies the articulation of parts in the database and set limits on the motion of those parts.

Level of detail: Identified by the `fltLod` record code, a level-of-detail (LOD) node is similar to a group, but serves as a switch to turn the display of everything below it on or off, based on the range from the viewer, according to its switch-in, switch-out distance and center location.

Switch: Identified by the `fltSwitch` record code, a switch node is a more general case of an LOD node. It allows the selection of zero or more children by invoking a selector mask. Any combination of children can be selected per mask and the number of definable masks is unlimited.

Clip: Identified by the `fltClip` record code, a clip node is composed of 4 points in space that define a clipping rectangle to apply to geometry below the clip node. Using this node, you can "cull" out geometry.

External reference: Identified by the `fltXref` record code, an external reference node is similar to a group node, but serves to reference an entire database file. The `fltXrefFilename` attribute specifies the database referenced by an external reference node.

Design of the API

File I/O

The API works by loading an OpenFlight file into in memory and allowing you to query or modify it while it is in memory. Changes you make to the database are saved to disk only when you explicitly write the file.

For more details on how to load and save files, see "File I/O" on page 41.

New Data Types

Most API parameters are described by standard C types: `char`, `short`, `int`, `float`, and `double`. The API defines several new data types, the most common of which are described here:

`mgbool`	A simple type for boolean values. Values are `MG_TRUE` and `MG_FALSE`.
`mgstatus`	An integer type to indicate error status values. Embedded in these values *will be* information identifying the kind of error, the severity of error, the subsystem that sent the error, etc. This embedded information is not currently implemented but will be in future releases of the API. In the current version, non-zero values indicate error, zero values success. To minimize the impact of future enhancements on your code, it is highly recommended that you use the macro `MSTAT_ISOK` to check values of this type for success rather than using equality operators.

mgrec	A generic type that can be used to access most types of records defined by the API. Primarily, objects of type **mgrec** represent nodes in the database. API users cannot access the fields of this structure directly; simply pass the structure into API functions instead. In this document, variables of type **mgrec** are often referred to as *records*.
mgcode	An integer type used to uniquely identify objects of type **mgrec**. Each **mgrec** has an associated **mgcode** that indicates which kind of record it is.

Access to Data

In memory, elements of an OpenFlight database are represented in the API by a generic record of type **mgrec**. There are three classes of records:

- *Node records* form the hierarchy of an OpenFlight database by linking to other node records.
- *Palette records* form the palettes referenced by attributes.
- *Attribute records* define the properties of node, palette or other (nested) attribute records.

The API provides access functions that let you store retrieve data in records. See "Storing and Retrieving Attributes" on page 65 for more information.

API Header Files

There are numerous header (include) files in the API. Each header file contains the declarations for different functional groupings of the API. For simplicity, the API provides one header file, **mgapiall.h**, that includes all others. Although you may choose to include just those header files that your application requires, it is recommended that you simply include **mgapiall.h**.

A Basic API Program

A stand-alone application that uses the API must call **mgInit** before opening a database or performing any other API operation. There is one exception to

this rule that is described below. A stand-alone application must call **mgExit** to clean up when it is through using API calls.

The only OpenFlight API function that your application can call before **mgInit** is **mgSetMessagesEnabled**. If you call any other API function before **mgInit**, it may not operate properly. Normally, **mgInit** issues several output messages that you might want to hide from the users of your application. Calling **mgSetMessagesEnabled** allows the stand-alone application to disable these messages.

The following program opens two databases: one that already exists, and one new database. This program also adds a node to the new database and writes both files to disk.

Sample: egio.c

```
/*****************************************************************************

  Sample file: EGIO.C

  Objective: Shows the structure of an OpenFlight program.  Shows how
     to open, create, close, and write OpenFlight database files.

  Program functions: Opens an OpenFlight database file specified on
     the command line.
     Opens a new OpenFlight database file named "new.flt".
     Closes and writes both database files.

  API functions used:
     mgInit(), mgExit(), mgGetLastError(), mgSetNewOverwriteFlag(),
     mgOpenDb(), mgNewDb(), mgCloseDb(), mgWriteDb(),
     mgAttach(), mgNewRec()

 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <openflightapi/mgapiall.h>

void main (int argc, char* argv[])
{
   mgrec *db;     /* top record of database file specified on command line */
   mgrec *newdb;  /* top record of new database file */
   mgrec *grec;   /* group record created for new database file */
   char new_fname[80];

   /* Always call mgInit() before any other OpenFlight API calls */
   mgInit (&argc, argv);

   /* open the database file with the name specified on the command line */
```

```
    if (!(db = mgOpenDb (argv[1]))) {
        char msgbuf [1024];
        mgGetLastError (msgbuf, 1024);
        printf ("%s\n", msgbuf);
        mgExit ();
        exit (1);
    }

    /* create a new database */
    strcpy (new_fname, "newfile.flt");
    mgSetNewOverwriteFlag (MG_TRUE);
    if (!(newdb = mgNewDb (new_fname))) {
        char msgbuf [1024];
        mgGetLastError (msgbuf, 1024);
        printf ("%s\n", msgbuf);
        mgExit ();
        exit (1);
    }
    /* create a group record and attach it to the */
    /* new database so it isn't empty */
    grec = mgNewRec (fltGroup);
    mgAttach (newdb, grec);

    /* write both database files */
    mgWriteDb (db);
    mgWriteDb (newdb);

    /* close both database files */
    mgCloseDb (db);
    mgCloseDb (newdb);

    /* always call mgExit() after all OpenFlight API calls */
    mgExit ();
}
```

Compiling and Linking Your Program

The OpenFlight API is distributed in three binary formats for the Windows platform; Visual Studio VC8, VC9 and VC12. In addition all versions are distributed for both Win32 (32 bit) and x64 (64 bit) architectures. If you are developing plug-ins for Creator version 15, you must use VC12 and the architecture that matches that of the version of Creator you are using (32 or 64 bit) so your plug-ins are compatible with Creator. If you are developing stand-alone applications, you can choose VC8, VC9 or VC12.

Once you have created the C/C++ source files that contain your stand-alone application, you are ready to compile the code and create an executable program.

There are a few things you have to do to set up your build environment. They are:

- Define **Preprocessor Definitions** for your compiler:
  On Windows (Win32): **WIN32** and **API_LEV4**
  On Windows (x64): **WIN64** and **API_LEV4**
  On Linux: **API_LEV4**
- Define **Include Directory** for your compiler:
  On Windows (Win32 and x64):
  $(PRESAGIS_OPENFLIGHT_API)\include
  On Linux: **$(PRESAGIS_OPENFLIGHT_API)/include**
  When you do this, you will include API header files in your source files as follows:
  #include "mgapiall.h"
- Define **Import Libraries** for your linker:
  On Windows (Win32 and x64): **mgapilib.lib** and **mgdd.lib**
  On Linux: **libmgapilib.so** and **libmgdd.so**
- Define **Library Path** for your linker:
  On Windows (Win32 and x64):
  $(PRESAGIS_OPENFLIGHT_API)\lib<platform>
  On Linux: **$(PRESAGIS_OPENFLIGHT_API)/lib**

In the lines above, **<platform>** is defined as described in "Installed Directories" on page 11. Also, **PRESAGIS_OPENFLIGHT_API** is an environment variable whose value is the directory where the API was installed.

On Windows: The default location for the API installation directory is:

C:\Presagis\Suite<suite>\OpenFlight_API

where **<suite>** is the suite number of the API you have installed. For example, OpenFlight API 15 would be installed to:

C:\Presagis\Suite15\OpenFlight_API

The API installer creates the environment variable **PRESAGIS_OPENFLIGHT_API** automatically.

On Linux: The default location for the API installation directory is:

/usr/local/Presagis/OpenFlight_API_<version>

where <version> is the version of the API you have installed. For example, version 15 of the API would be installed to:

/usr/local/Presagis/OpenFlight_API_15_0

You have to set the environment variable **PRESAGIS_OPENFLIGHT_API** manually in your shell environment. See the *tcsh* script file included with the distribution:

/usr/local/Presagis/OpenFlight_API_<version>/SOURCEME

for more information on setting **PRESAGIS_OPENFLIGHT_API.**

Windows

The following section provides detailed instructions for setting up your build environment on Windows. These instructions are specific to Visual Studio 2008. Instructions for previous or subsequent versions of Visual Studio are similar but may not be identical.

You may also find it useful to examine the sample read-write programs included with the OpenFlight API distribution. These samples include source code as well as Visual Studio workspace and project files showing you how to set the compiler and link settings described in this section.

If you are using a different compiler, see the documentation included with your **C/C++** compiler.

Define Preprocessor Definitions

The settings described in this section are slightly different on Win32 and x64 architectures.

Windows 32-bit Systems (Win32)

In Visual Studio, open the **Project Properties** window and select **Configuration Properties** > **C/C++** > **Preprocessor.** In **Preprocessor Definitions**, make sure **WIN32** and **API_LEV4** are set. Do this for all configurations as necessary.

Windows 64-bit Systems (x64)

In Visual Studio, open the **Project Properties** window and select **Configuration Properties** > **C/C++** > **Preprocessor**. In **Preprocessor Definitions,** make sure **WIN64** and **API_LEV4** are set. Do this for all configurations as necessary.

Define Include Directory

The settings described in this section are the same on Win32 and x64 architectures.

You must specify the location of the API header files. In Visual Studio, open the **Project Properties** window and select **Configuration Properties > C/C++ > General**. In **Additional Include Directories**, make sure the following path is included:

```
$(PRESAGIS_OPENFLIGHT_API)\include
```

Do this for all configurations as necessary.

Define Import Libraries

The settings described in this section are the same on Win32 and x64 architectures.

You must tell the linker to link with the necessary API import library files. In Visual Studio, open the **Project Properties** window and select **Configuration Properties > Linker > Input**. In **Additional Dependencies**, make sure the following libraries are included:

`mgapilib.lib` and `mgdd.lib`

Do this for all configurations as necessary.

NOTE: In early versions of the OpenFlight API, the import library `fltdata.lib` was also needed. But starting with API version 2.3, it is no longer required for your program to link against `fltdata.lib.` The corresponding dynamic link library file, `fltdata.dll,` is still required by the runtime environment but the import library file is no longer required at link time.

Define Library Path

You must tell the linker where to find the API import library files. The settings described in this section are slightly different on Win32 and x64 architectures. And on Win32, the procedures may be slightly different for the different versions of Visual Studio you might be using.

Windows 32-bit Systems

In Visual Studio, open the **Project Properties** window and select **Configuration Properties** > **Linker** > **General**. In **Additional Library Directories**, make sure the following path is included:

```
$(PRESAGIS_OPENFLIGHT_API)\lib<platform>
```

where **<platform>** is defined as described in **"Installed Directories"** on page 11. For example if you are using **VC8-Win32** you will use:

```
$(PRESAGIS_OPENFLIGHT_API)\lib_vc8
```

Do this for all configurations as necessary.

Windows 64-bit Systems (x64)

In Visual Studio, open the **Project Properties** window and select **Configuration Properties** > **C/C++** > **Preprocessor**. In **Preprocessor Definitions**, make sure **WIN64** and **API_LEV4** are set. Do this for all configurations as necessary.

In Visual Studio, open the **Project Properties** window and select **Configuration Properties** > **Linker** > **General**. In **Additional Library Directories**, make sure the following path is included:

```
$(PRESAGIS_OPENFLIGHT_API)\lib<platform>
```

where **<platform>** is defined as described in **"Installed Directories"** on page 11. For example if you are using **VC8-x64** you will use:

```
$(PRESAGIS_OPENFLIGHT_API)\lib_vc8_x64
```

Do this for all configurations as necessary.

Microsoft CRT Dependencies

Both of the VC9 versions of the OpenFlight API DLLs are dependent on VC90 CRT version **9.0.21022.8**. The OpenFlight API installer automatically installs the proper Microsoft Visual Studio 2008 (release) redistributable package if your computer does not already have it. The debug version of the CRT should be installed on your computer when you apply the proper patches/service packs to Visual Studio 2008.

Both of the VC8 versions of the OpenFlight API DLLs are dependent on VC80 CRT version **`8.0.50727.4053`**. The OpenFlight API installer automatically installs the proper Microsoft Visual Studio 2005 (release) redistributable package if your computer does not already have it. The debug version of the CRT should be installed on your computer when you apply the proper patches/service packs to Visual Studio. 2005

Linux

The following section provides detailed instructions for setting up your build environment on Linux.

You may also find it useful to examine the sample read-write programs included with the OpenFlight API distribution. These samples include source code as well as *makefiles* showing you how to set the compiler and link settings described in this section. These samples assume you are using the *gcc* compiler. If you are using a different compiler, see the documentation included with your **C/C++** compiler.

Define Preprocessor Definitions

Set the compiler option **`-DAPI_LEV4`** when you compile your **C/C++** source files.

Define Miscellaneous Compiler Options

As noted above, the OpenFlight API binary libraries are 32 bit format. To generate code compatible with the API libraries, specify the compiler option: `-m32`.

Define Include Directory

You specify the location of the API header files with the compiler option `-I` when you compile your **C/C++** source files. The default directory where the header files are located is:

`/usr/local/Presagis/OpenFlight_API_<version>/include`

If you use the **`PRESAGIS_OPENFLIGHT_API`** environment variable you can specify the compiler option **`-I$(PRESAGIS_OPENFLIGHT_API)/include.`** If you do not use the **`PRESAGIS_OPENFLIGHT_API`** environment variable, set the **`-I`** compiler option accordingly.

Define Import Libraries

You tell the linker to link with the necessary API import library (shared object) files with the link option `-l`. The API import libraries required are `libmgapilib.so` and `libmgdd.so`. To link with both these libraries include the link options `-lmgapilib` and `-lmgdd`. The API itself is dependent on several system libraries. As a consequence, you will need to link your application with them:

`libc.so` (**C** Runtime library)
`libstdc++.so` (**C++** Runtime library, if your application is **C++**)
`libm.so` (Math library)
`libpthread.so` (Thread library)
`libuuid.so` (Universally Unique Identifier library)

Define Library Path

You tell the linker where to find the API import library files with the link option `-L`. The default directory where these import libraries are located is:

```
/usr/local/Presagis/OpenFlight_API_<version>/lib
```

If you use the `PRESAGIS_OPENFLIGHT_API` environment variable you can specify the link option `-L$(PRESAGIS_OPENFLIGHT_API)/lib`. If you do not use the `PRESAGIS_OPENFLIGHT_API` environment variable, set the `-L` link option accordingly.

Running Your Program

The API libraries are dynamically linked (or shared). They are loaded at runtime, rather than encapsulated in your application. Therefore, you must make sure that your application can find the shared libraries at runtime.

Windows

On Windows, the OpenFlight API Dynamic Link Libraries (DLLs) are required when you run your stand-alone application. When you run your application, Windows searches for DLLs (including the OpenFlight API DLLs) in the following sequence:

1. The directory where your stand-alone executable is located.
2. The current directory.

3 The Windows system directory. The Windows function **`GetSystemDirectory`** function retrieves the path of this directory.

4 The Windows directory. The Windows function **`GetWindowsDirectory`** function retrieves the path of this directory.

5 The directories listed in the **`PATH`** environment variable.

It is recommended that you set up your runtime environment to find the API DLLs in either (1) or (5). Using (2), (3) or (4) can lead to potential versioning conflicts when running Creator or other OpenFlight API-based applications. To avoid confusion, you should choose either (1) or (5) but not both, depending on your requirements.

To implement (1), simply place your application in the **`\bin<platform>`** folder where the OpenFlight API Dynamic Link Libraries were installed. By default, this is one of the following, depending on the platform and configuration you are using:

\bin folder	Platform / Configuration
`$(PRESAGIS_OPENFLIGHT_API)\bin\debug`	`VC12-Win32/debug`
`$(PRESAGIS_OPENFLIGHT_API)\bin\release`	`VC12-Win32/release`
`$(PRESAGIS_OPENFLIGHT_API)\bin_x64\debug`	`VC12-x64/debug`
`$(PRESAGIS_OPENFLIGHT_API)\bin_x64\release`	`VC12-x64/release`
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc9\debug`	`VC9-Win32/debug`
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc9\release`	`VC9-Win32/release`
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc9_x64\debug`	`VC9-x64/debug`
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc9_x64\release`	`VC9-x64/release`
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc8\debug`	`VC8-Win32/debug`
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc8\release`	`VC8-Win32/release`
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc8_x64\debug`	`VC8-x64/debug`

\bin folder	Platform / Configuration
`$(PRESAGIS_OPENFLIGHT_API)\bin_vc8_x64\release`	`VC8-x64/release`

To implement (5), add the directory where the OpenFlight API DLLs are located to the **PATH** environment variable. By default, this is one of the `bin` folders listed above, depending on the platform and configuration you are using.

NOTE: The following assumes the OpenFlight API DLLs are located in the default location (as listed above). If you installed the API somewhere else, make the necessary adjustments.

On Windows 7, environment variables are displayed and may be set in the System Control Panel. Click **System** in the **Windows Control Panel**. Click the **Advanced system settings** link and then the **Environment Variables** button there. Locate the **PATH** variable in the **User Variable** list. If the **PATH** variable is not defined in this list, create it by clicking **New**. In the dialog that is displayed, enter **PATH** in the **Variable** field. Then enter the **\bin** folder (listed above), corresponding to the platform and configuration you are using, in the **Variable Value** field and click **OK**.

If the **PATH** variable is already defined, highlight it and click **Edit**. In the dialog that is displayed, add the **\bin** folder (listed above), corresponding to the platform and configuration you are using, at the end of the **Variable Value** field, separated from the previous directory listed in this field with a semicolon. After entering the value, click **OK**.

Linux

On Linux, the OpenFlight API Shared Objects (SOs) are required when you run your stand-alone application. When you run your application, Linux searches for SOs (including the OpenFlight API SOs) in the following sequence:

1. The directory where your stand-alone executable is located
2. The directories listed in the **LD_LIBRARY_PATH** environment variable. See the *tcsh* script file included with the distribution: **/usr/local/Presagis/OpenFlight_API_<version>/SOURCEME** for more information on setting **LD_LIBRARY_PATH**.

Using OpenFlight Script

As noted above, OpenFlight Script is a cross-platform Python Language binding to the **C** Language OpenFlight API. Distributed as a Python 'module', OpenFlight Script gives you access to the OpenFlight API functions in your Python runtime environment.

You will need to install a Python environment (Python 2.7.x is required) on your computer before you can use OpenFlight Script in a stand alone program environment.

NOTE: You can run OpenFlight scripts in Creator directly since Creator includes a built-in Python environment.

There are many Python distributions available. A good place to start is

`http://www.python.org`

A general knowledge of Python will help you get started quickly using OpenFlight Script.

After installing a Python environment on your computer, you must tell Python where to locate the OpenFlight Script module. The OpenFlight Script module is comprised of the OpenFlight API dynamic link libraries (including **`fltdata.dll`**) and two additional Python specific files included in the OpenFlight API distribution. The additional Python files are:

- **`mgapilib.pyd`**
- **`mgapilib.py`**

NOTE: The Python binding files (**`.pyd`**) are only available with the VC9 (Release) versions of the installed binaries.

Both Python files (and the dynamic link library files) are located in the `bin` folder corresponding to the platform and configuration you are using:

\bin folder	Platform / Configuration
`$(PRESAGIS_OPENFLIGHT_API)\bin\release`	`VC12-Win32/release`
`$(PRESAGIS_OPENFLIGHT_API)\bin_x64\release`	`VC12-x64/release`

Here are the steps to make the OpenFlight Script module available in Python.

NOTE: The steps listed here are required only if you want to run your OpenFlight scripts in a stand-alone program environment. You do not need to do this if you are running your scripts in Creator.

- Set the **PYTHONPATH** environment variable to include the folder where the OpenFlight Script module files (dynamic link library files, **mgapilib.pyd** and **mgapilib.py**) are located (see list above for platform and configuration you are using).
- Set the **PRESAGIS_OPENFLIGHT_SCRIPT** environment variable to be the folder where the OpenFlight Script module files are located.

Running Your Script in a Stand-Alone Program Environment

After you configure your Python environment to include the OpenFlight Script module, you are ready to create and run OpenFlight Scripts. Perhaps the easiest way to get started is to study the sample scripts included with the OpenFlight API distribution. These sample scripts are located at:

```
$(PRESAGIS_OPENFLIGHT_API)\samples\scripts
```

An OpenFlight script run in the Python environment must first import the OpenFlight Script module. You do this with the following Python statement:

```
from mgapilib import *
```

This will make available to your script all the OpenFlight Script functions.

If your script needs to use other modules you must "import" them as well. A common module to import is **sys** which gives you access to the arguments passed from the Python environment to your script when it is run.

Your script must also call **mgInit(None, None)** before opening a database or performing any other API operation. There is one exception to this rule that is described below. A script must call **mgExit** to clean up when it is through using API calls.

The only OpenFlight API function that your script can call before **mgInit** is **mgSetMessagesEnabled**. If you call any other OpenFlight Script function before **mgInit**, it may not operate properly. Normally, **mgInit** issues several

output messages that you might want to hide from your script's users. Calling **mgSetMessagesEnabled** allows your script to disable these messages.

The following sample script opens two databases: one that already exists, and one new database. This script also adds a node to the new database and writes both files to disk:

```
##
##  Sample file: EGIO.PY
##
##  Objective: Shows the structure of an OpenFlight program.  Shows how
##     to open, create, close, and write OpenFlight database files.
##
##  Program functions:Opens an OpenFlight database file specified on
##     the command line.
##          Creates a new OpenFlight database file named "newfile.flt".
##          Closes and writes both database files.
##
##  API functions used:
##     mgInit(), mgGetLastError(), mgSetNewOverwriteFlag(),
##          mgNewRec(), mgAttach(),
##          mgOpenDb(), mgNewDb(), mgCloseDb(),
##          mgWriteDb(), mgExit().
##

# need access to argc and argv
import sys

# include all OpenFlight API headers
from mgapilib import *

NEWFILE = "newfile.flt"

def main ():

   # check for correct number of arguments
   if len(sys.argv) < 2:
      print "\nUsage: %s <input_db_filename>\n" % (sys.argv[0])
      print "   Reads/Writes database: <input_db_filename>\n"
      print "   Creates a new OpenFlight database file named %s\n" % (NEWFILE)
      print "   Creates a new group in the new file\n"
      print "\n"
      return

   # initialize the OpenFlight API
   # always call mgInit BEFORE any other OpenFlight API calls
   #
   mgInit (None, None)

   # open database
   print "\nOpening database: ", sys.argv[1], " \n"
   db = mgOpenDb (sys.argv[1])
   if db == None:
      msgbuf = mgGetLastError()
      print msgbuf, "\n"
      mgExit()
```

```
      return

   # create a new database (newdb)
   mgSetNewOverwriteFlag (MG_TRUE)
   print "\nCreating database: %s\n" % (NEWFILE)
   newdb = mgNewDb (NEWFILE)
   if newdb == None:
      msgbuf = mgGetLastError()
      print msgbuf, "\n"
      mgExit()
      return

   # create a group record and attach it to the new database so it isn't empty
   group = mgNewRec (fltGroup)
   if group == None:
      print "Creating top group: Failed\n"
   else:
      print "Creating top group: Ok\n"

   ok = mgAttach (newdb, group)
   if ok == MG_TRUE:
      print "Attaching top group: Ok\n"
   else:
      print "Attaching top group: Failed\n"

   # write both database files
   ok = mgWriteDb (db)
   if ok == MG_FALSE:
      print "Error writing database\n"

   ok = mgWriteDb (newdb)
   if ok == MG_FALSE:
      print "Error writing new database\n"

   # close both database files

   # close the database
   ok = mgCloseDb (db)
   if ok == MG_FALSE:
      print "Error closing input database\n"

   # close the database
   ok = mgCloseDb (newdb)
   if ok == MG_FALSE:
      print "Error closing new database\n"

   # always call mgExit() AFTER all OpenFlight API calls
   mgExit()

main()
```

Running Your Script in Creator

As noted above, you do not need to install Python on your system if you only want to run your OpenFlight scripts in Creator. For more information on how to run your scripts in Creator, see the Creator help for the *OpenFlight Script Editor*.

NOTE: The sample stand-alone scripts included with the OpenFlight API SDK are not suitable for use in Creator. They include calls to **`mgExit`** and in some cases **`sys.exit`**. If you are running an OpenFlight script in Creator you should not call either of these functions.

OpenFlight Script for OpenFlight C Language API Programmers

This section describes those features of OpenFlight Script which may be unfamiliar to OpenFlight **C** Language API programmers.

Function Signatures in Python vs C Language API

Most of the functions in OpenFlight Script are identical in calling method and functionality to that of the corresponding **C** language specification. There are a few exceptions due to the inherent differences between the **C** and Python languages. Most of these exceptions exist when a C language function has one or more output parameters. Output parameters are those that get modified in the function and hence returned to the caller. Since C functions have only a single return value, output parameters are commonly used for a function to return multiple values simultaneously.

In general Python does not support output parameters for functions. To compensate for this Python does support multiple return values for functions in the form of "lists" or "tuples". For **C** language functions that do have output parameters, there will be a different function signature for the Python equivalent. This is best explained by example. Consider the OpenFlight function **`mgGetVtxCoord`**. In **C**, its function signature is**:**

```
mgbool mgGetVtxCoord (mgrec* vtx,
      double* x, double* y, double* z);
```

This **C** function returns the x, y, z coordinates of a vertex in the output parameters `x`, `y` and `z`, respectively. In Python, remember, output parameters

are not supported. For that reason, the Python signature for this same function is different:

```
mgbool, x, y, z mgGetVtxCoord (mgrec* vtx)
```

In Python, this notation indicates that **mgGetVtxCoord** return 4 four values as a list. The syntax for calling this function in both **C** and Python is shown here.

C Language

```
double x,y,z;
mgbool status;
status = mgGetVtxCoord (vtx, &x, &y, &z);
if (status == MG_TRUE) {
   // x, y and z contain valid values
}
```

Python

```
status,x,y,z = mgGetVtxCoord (vtx)
if (status == MG_TRUE):
   // x, y and z contain valid values
```

When the **C** and Python signatures for an OpenFlight function differ, each signature is listed in the *OpenFlight API Reference*. When there is no difference between the **C** and Python signature for a function, a single signature is listed.

2

General Topics

This chapter describes some general tasks that most programs need to perform. These tasks include:

- Initialization
- Message Reporting
- Memory Management
- File I/O
- Database Management

Initialization

In the stand-alone program environment, the OpenFlight API must be initialized before it can be used, and must be terminated when it is no longer needed. In the Creator program environment, the API is initialized and terminated automatically by Creator.

A stand-alone program can initialize the API library using the function **`mgInit`**. Generally, your program must call this function before calling other API functions. An exception to this rule is for the API function **`mgSetMessagesEnabled`**, which you may call prior to calling **`mgInit`**. This is useful because **`mgInit`** typically displays many messages itself and you may want to suppress those messages. Your stand-alone program can terminate the API

library using the function **mgExit**. The API library is not designed to be initialized and terminated more than one time in your program. Therefore, you should only call **mgInit** and **mgExit** one time each over the lifetime of your program.

Note: Because the API is initialized and terminated automatically by Creator, **mgInit** and **mgExit** are disabled if called within the Creator program environment.

For initialization and cleanup functions that apply to individual databases, see **mgNewDb**, **mgOpenDb**, **mgWriteDb**, **mgSaveAsDb**, **mgExportDb** and **mgCloseDb**.

Message Reporting

Messages can be reported by the API as well as by plug-ins and stand-alone applications. Messages are classified by severity levels, status, warning, or error. When a message is reported, it is displayed in a way that is appropriate for the program environment in which it was reported. For example, when a message is reported in the stand-alone program environment, it is displayed to **stdout**. When a message is reported in the Creator program environment, it is displayed in the Creator Status Log window.

In a stand-alone program, the volume of messages reported may become so great that it may be desirable to "turn off" message display for one or more severity levels. The function **mgSetMessagesEnabled** allows you to do just that. To determine whether message display is enabled or disabled for a given severity level, use the function **mgGetMessagesEnabled**.

Even when message display is disabled for a particular severity level, messages of that severity will continue to get reported, they just will not be displayed. Whether message display is enabled or not, the API saves a copy of the last message reported. The text of the last message reported can be retrieved using the function **mgGetLastError**.

The following code fragment shows how a stand-alone program might disable the display of all messages and then an important error occurs, get the text of the error and display it in its own way:

```
mgSetMessagesEnabled (MMSG_ERROR, MG_FALSE);
mgSetMessagesEnabled (MMSG_WARNING, MG_FALSE);
```

```
mgSetMessagesEnabled (MMSG_STATUS, MG_FALSE);

if (!(db = mgOpenDb (argv[1])))
{
   char msgbuf [1024];
   mgGetLastError (msgbuf, 1024);
   printf ("%s\n", msgbuf);
}
```

As mentioned above, Creator always displays all messages reported within its program environment to the Status Log window. Message display cannot be disabled in the Creator environment. Therefore, the function **mgSetMessagesEnabled** is disabled and the function **mgGetMessagesEnabled** will always return **MG_TRUE** if called from within the Creator program environment. And since messages are always available to the user in the Status Log window, the function **mgGetLastError** is also disabled in this environment.

Plug-ins and stand-alone applications can report **printf** style formatted messages of any severity level using the function **mgSendMessage**. The following code fragment shows how this function might be called to display some numeric values:

```
int x = 10;
float y = 20.0f;
double z = 30.0;

mgSendMessage ("x=%d, y=%f, z=%lf\n", x, y, z);
```

Memory Management

The API provides its own memory management functions, **mgMalloc** and **mgFree**. You are encouraged to use these functions (instead of **malloc** and **free)** when allocating or freeing memory for use by the API. Whatever memory management functions you choose to use, be careful not to "mix" them. For example, use only **mgFree** for memory blocks allocated by **mgMalloc**, do not use **free**. Conversely, do not use **mgFree** for memory blocks allocated by **malloc**. If you do mix alloc/free calls, you will experience severe problems in your application.

Some C language API functions allocate space for the data they return. In these cases, you must free the memory when it is no longer needed, or you will cause a memory leak. You must use **mgFree** to dispose of this memory

properly. You do not free memory in this way when using OpenFlight script. The Python language implements its own form of "garbage collection" to free memory when no longer needed.

File I/O

Most programs start by opening an OpenFlight file. You can create a new database with **mgNewDb**, or open an existing one with **mgOpenDb**. Usually it is an error to call **mgOpenDb** on a file that does not exist, or **mgNewDb** on an existing file. However, you can change this behavior.

You can change how **mgNewDb** behaves when it is called to create a file that already exists. If you call the function **mgSetNewOverwriteFlag** and specify **MG_TRUE**, subsequent calls to **mgNewDb** or **mgSaveAsDb** will overwrite the file specified if one by that name already exists.

Similarly, you can change how **mgOpenDb** behaves when it is called to open a file that does not exist. If you call the function **mgSetOpenCreateFlag** and specify **MG_TRUE**, subsequent calls to **mgOpenDb** will create the file specified if one by that name does not exist.

If you open a database using **mgOpenDb** that contains one or more external references, you can specify whether or not you want the API to load the contents of these external reference files. If your program does not need to access the contents of externally referenced files, it is much more efficient to *NOT* load external references. Use the function **mgSetReadExtFlag** to specify whether or not you want the API to load externally referenced files it finds contained in the file being opened when you call **mgOpenDb**.

Just as you can control how external references are loaded by **mgOpenDb**, you can do the same with respect to textures. Use the function **mgSetReadTexturesFlag** to specify whether or not you want the API to load textures and their attributes into memory. Note that reading textures can consume both processing and memory resources. Unless your application explicitly needs to access the texels or attributes of a texture, it is much more efficient to set this preference to **MG_FALSE**.

Both **mgNewDb** and **mgOpenDb** return a database node record, an **mgrec** structure. You will need this record to access information that applies to this database, such as palettes and graphics context.

The API performs most of its work in memory and requires explicit instructions before saving any changes to disk. You call **mgWriteDb** or **mgSaveAsDb** to save any database changes, including those made to new files. Both **mgWriteDb** and **mgSaveAsDb** write databases in the OpenFlight format version corresponding to the API running. To write databases to other OpenFlight format versions, use **mgExportDb**. If you exit your application or close a database without explicitly writing it, the corresponding file on disk is not affected and any changes you might have made are discarded.

You can, however, make changes to other types of files without writing the database file. You might want to do this, for example, to edit palette files, texture files, or texture attributes, without modifying the database. Like the database file, these files must also be saved explicitly.

For more information, see the *OpenFlight API Reference* for:

mgWriteLightSourceFile
mgWriteLightPointFile
mgWriteMaterialFile
mgWriteDefaultColorPalette
mgWriteColorPalette
mgWriteImage
mgWriteImageAttributes
mgWriteTexture
mgWriteTexturePalette

When you are finished with a database, always call **mgCloseDb** to delete the hierarchy and free the associated memory.

In the Creator modeling environment, the user normally opens and closes files explicitly on the Creator desktop. There are instances in which your plug-in or script might want to process a file in Creator. To enable this, your plug-in or script can use **mgNewDb**, **mgOpenDb**, **mgWriteDb** and **mgSaveAsDb** in Creator. When you do this the file you open is not opened into an active window on the Creator desktop. Note that you cannot open, save or close a file that is open in an active window on the Creator desktop.

Important: **mgCloseDb** does not write the database to disk.

Database Management

An application will typically work on a single OpenFlight database at a time. Such an application might load an OpenFlight database, modify parts of it and then save and close the database. The application may do this over and over for multiple databases but will have only one database loaded at any one time. While this is typical, it is not required. Your application may have more than one database open simultaneously. This section describes the concepts in the API to support this.

The API maintains what is referred to as the *current database*. When defined, the current database is the **`fltHeader`** node (**`mgrec`**) of that database. When you open a database, it becomes the current database. When you open a subsequent database (without closing the first), the newly opened database becomes the current, and so on. When you close a database in your stand-alone program, (and there are still more open) the current database is undefined. In this situation, your application must specify which of the remaining open databases are to become the new current database. In the Creator modeling environment, the current database is managed completely by the user and Creator. Your plug-in or script running in Creator has no control over which is the current database.

Many API functions operate in the context of a specific database. Most of these functions work in the context of an *explicit* database - one that you specify as a parameter to the function. Whenever a function accepts an **`mgrec`** record (database node or other record) as a parameter, that function works in the context of the database that contains that record. Consider the API function **`mgWalk`**. When you pass a node record to this function, it traverses the nodes "below" this node and as it does, calls one or more traversal callbacks you specify. In this way, **`mgWalk`** works in the context of the database that contains the node record you passed to it. Some functions work in the context of a database but do not accept records as parameters to specify *which* database. Such functions work in the context of an *implicit* database, the *current database*. One such function is **`mgNewRec`**. This function creates a new node record in the context of the current database. When a node is created by **`mgNewRec`** it is bound to the current database and can only be attached to hierarchy contained in the current database. If you are working with more than one open database, you must make sure that the current database is set properly before you call **`mgNewRec`**. If the current database was not set to the database where you plan to attach the new node

created by **mgNewRec**, the new node will fail to attach. If you want to create a new node in the context of a database that is not the current database, you can use **mgNewRecDb**.

Your application uses the function **mgGetCurrentDb** to obtain the current database and **mgSetCurrentDb** to set it. In Creator, the user controls the current database on the Creator desktop so you should not call **mgSetCurrentDb** from a plug-in or script inside the Creator modeling environment. In the stand-alone program environment, some API functions, like **mgOpenDb** and **mgNewDb**, set the current database automatically for you. There are instances, however, in which your application may need to call **mgSetCurrentDb** explicitly. One such instance was mentioned above. If your application has more than one database open and closes one, you must call **mgSetCurrentDb** to *choose* which of the remaining open databases is to become the new current database. In addition, if your application is ever unsure of which open database is current, it is a better to call **mgSetCurrentDb** too many times (even if doing so is redundant) than not enough.

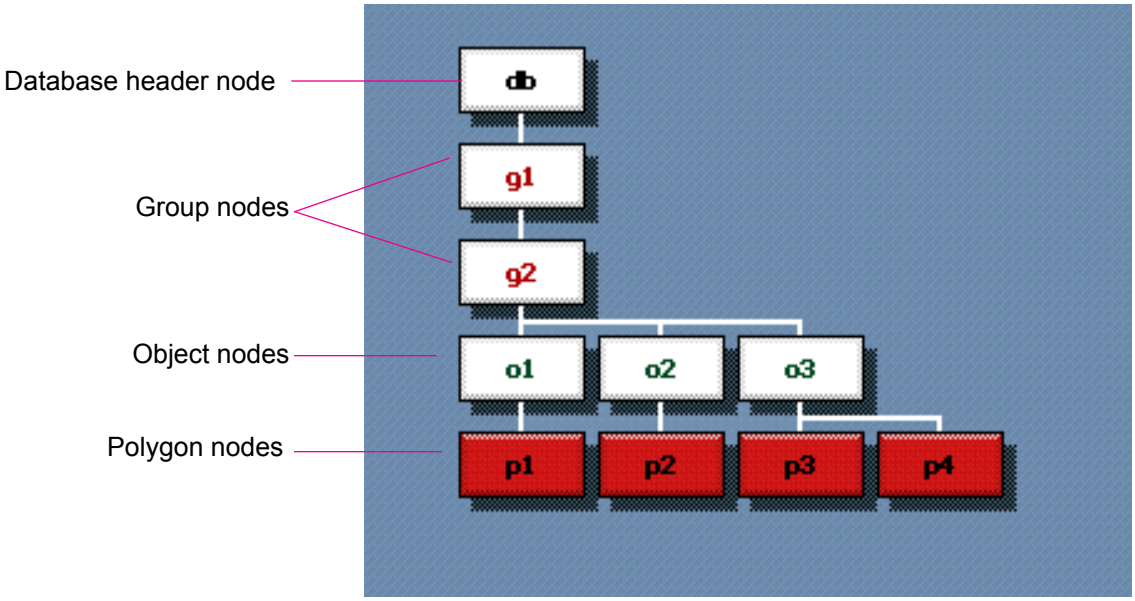
As suggested above, some API functions require you to pass a database node to specify the database in which the function is to operate. Your application can obtain the database node in one of several ways:

- If your application opens exactly one database at a time, you can store the database header node returned by **mgNewDb** or **mgOpenDb** in a global static variable.
- If you don't have access to the database node but do have access to another node contained in the database, you can use **mgRec2Db** to find the database node that contains that other node.
- Use **mgGetCurrentDb** to simply return the current database node as maintained by the API.

You can determine the file name of any node's database by calling **mgRec2Filename**.

The OpenFlight Hierarchy

OpenFlight uses a multilevel hierarchy (graph) to define the organization of items in the database. At the top or root level is the database header node (*node* is a generic term for any record in the hierarchy). At the bottom are object nodes comprised of polygon nodes, which are, in turn, made up of vertex nodes. Between these two levels are a number of different types of organizational nodes that are attached to each other to organize the database.



Simple Relationships

OpenFlight supports a variety of relationships between nodes in the hierarchy. The simplest of these relationships is between parent node, child node, and sibling node. This is a direct hierarchical relationship in which control flows from the top down. Children inherit transformations and display state from parents. Attach and detach operations treat a node and all its descendants as a unit.

Use **`mgGetChild`** to access the first child of a node; use **`mgGetParent`** from any child to access the parent. Use **`mgGetNext`** or **`mgGetPrevious`** to walk through the list of siblings. When the last node in the list is reached, these routines return **`MG_NULL`**.

In the figure above,

- **`mgGetChild(g2)`** returns **`o1`**
- **`mgGetParent(o1)`**, **`mgGetParent`**(o2), and **`mgGetParent`**(o3) all return **`g2`**.
- **`mgGetNext(o1)`** returns **`o2`**
- **`mgGetPrevious(o2)`** returns **`o1`**
- **`mgGetPrevious(o1)`**, **`mgGetNext(o3)`**, **`mgGetChild(p1)`**, and **`mgGetParent(db)`** all return **`MG_NULL`**.

Nesting

Nesting provides a mechanism to form simple hierarchical relationships between nodes that are semantically different from standard child/parent relationships. In this way, a node may be hierarchically related to more than just its child and its parent. The API uses nesting to represent two different modeling constructs, subfaces and morph vertices. These constructs are described in this section.

Subfaces

When two polygons are coplanar, it is helpful to structure them as a nested child and nested parent. Creator and real-time applications use this

convention to determine the drawing order of coplanar polygons, when drawing with a z-buffer. Both the nested child and the nested parent must be **fltPolygon** nodes; they are equivalent to Creator's subface/superface nodes.

Use **mgAttach**, **mgAppend**, or **mgInsert** to create nested nodes from coplanar polygons. Use **mgGetNestedChild** and **mgGetNestedParent** to access a nested node or its nested parent, respectively.

A **fltPolygon** can have many nested children. The relationship between these children is just like any other siblings, and you use **mgGetNext** and **mgGetPrevious** to traverse the list.

A nested child can, in turn, have nested children of its own.

Morph Vertices

A special form of nesting is *morph vertices*. Morph vertices help provide a smooth visual transition between levels of detail. A morph vertex can be attached to a normal vertex in a level of detail such that when that level of detail switches in, the visual appearance of the normal vertex *morphs* (using interpolation) between the attributes of the morph vertex and those of the normal vertex. Using morph vertices in conjunction with level of detail transition distances, real-time systems can morph vertex attributes (position, normal, color, texture uv's, etc.) to provide smooth visual transitions between successive levels of detail.

Both the normal and morph vertex must be **fltVertex** nodes. To assign a morph vertex to a normal vertex, use the function **mgAttach**. The vertex that you assign to be a morph vertex cannot be attached to any other node hierarchies. To return the morph vertex of a normal vertex, use **mgGetMorphVertex**. A morph vertex cannot have a morph vertex of its own.

The following code fragment assigns a morph vertex to a vertex, **vtx:**

```
mgrec *morphVtx;

morphVtx = mgNewRec (fltVertex);
mgAttach (vtx, morphVtx);
```

Instancing

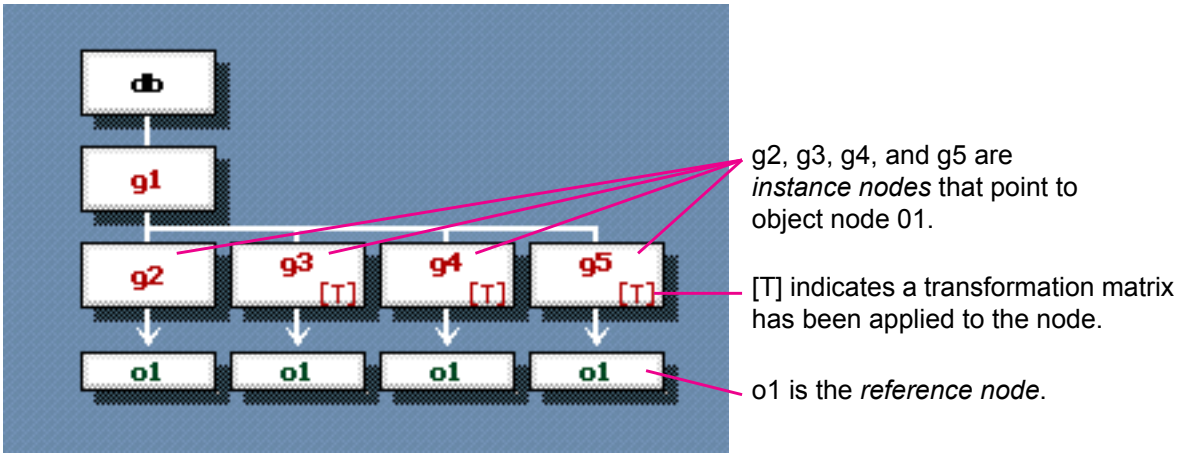
Instancing is the ability to describe a portion of geometry one time, then display it one or more times.

- An *instance* is an occurrence of a portion of the geometry in a database.
- A *reference* is a node that occurs more than once in a database. This node is where the geometry and attributes being instanced are defined. Changes to the reference affect all the instances of that node.

You apply a transformation to position each instance and distinguish it from the others. (See "Transformations" on page 67.) Instancing is most commonly used with simple, repetitive models, such as trees in a forest.

Instancing allows you to use a portion of a database many times without reloading the geometry, which saves computer memory and modeling time. OpenFlight supports instancing of objects, groups, and group-like nodes.

Instance nodes are nodes that point to the *reference node*. There can be many instances of a single reference node, but only one reference node for each instance. The following illustration shows four instances of the single reference node o1.

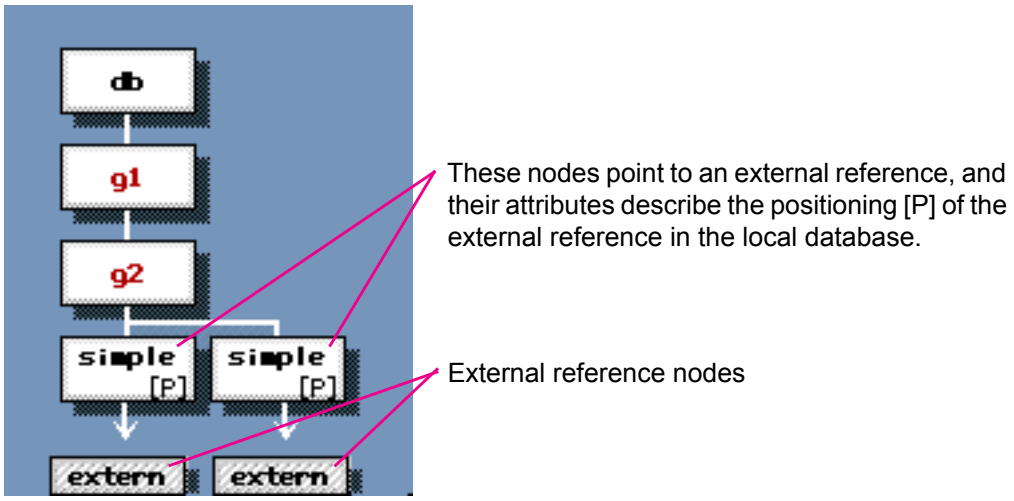


- To access the reference node from one of its instances (for example, to change the geometry or color), use **`mgGetReference`**. Remember that changes to the reference node affect all the instances.
- To modify a single instance (by adding a transformation, for example), operate on the instance itself.

- To walk through a reference node's list of instances, use **`mgGetFirstInstance`** and **`mgGetNextInstance`**.
- To add an instance to a reference node's instance list, call **`mgReference`**.
- To remove an instance from the list, call **`mgDeReference`**.

External References

An *external reference* is a reference to another database. External referencing allows you to include the contents of one database in other databases without duplicating the geometry, which saves disk space and streamlines the assembly of complicated databases. Unlike instances, external references are *read only*; you can position, orient, and scale an external reference, but you cannot edit its contents without opening the external database file itself.



You create external reference nodes just like you create any other node. External reference nodes are of type **`fltXref`**. (See "Creating New Nodes" on page 57 for more information.)

Traversing the Hierarchy

There are two ways to traverse the database hierarchy: *automatic* and *manual*. With automatic traversal, you make a single call to **`mgWalk`** or **`mgWalkEx`** passing in functions to be called when visiting each node in the tree. With manual traversal, you explicitly step through the branches of the tree,

deciding which branches to traverse, and in what order. Automatic traversal is so flexible that you will rarely need to use manual traversal unless you have special traversal requirements.

Automatic Traversal

With automatic traversal, you provide some general instructions up front. You specify flags that define what kinds of relationships to traverse (for example, whether to traverse one instance or all of them). You also provide function pointers for actions to be taken before and after visiting every node in the tree.

There are two functions to perform automatic traversal, **`mgWalk`** and **`mgWalkEx.`** They are very similar. They both recursively traverse a node's children performing a set of callbacks at each node in the hierarchy visited. They both, optionally, can accumulate a matrix stack during the traversal. This matrix stack accumulated keeps track of the transformations applied in the hierarchy and is accessible inside the callback functions by calling **`mgWalkGetMatrix.`** The matrix stack accumulated is formed by multiplying together all the matrices of the nodes contained in the path from the root of the traversal to the current node of the traversal. The main difference between **`mgWalk`** and **`mgWalkEx`** is the starting matrix they each use during matrix accumulation. The function **`mgWalkEx`** allows you to specify an arbitrary starting matrix while **`mgWalk`** always starts with the identity matrix:

```
/*****************************************************************************

  Sample file: EGWALK1.C

  Objective: Shows how to traverse an OpenFlight database in several ways.

  Program functions:
    Open a database from command line.
    Traverse the database without visiting vertices, or references.
    Traverse the database, visiting vertices and the first instance of each
    referenced node.
    Traverse the database, visiting vertices and every instance of each
    referenced node.
    Traverse the database, printing the accumulated matrix stack at the nodes.
    Cycle through the database's levels of detail, traverse and visit only
    the records associated with the current level of detail.

  API functions used:
    mgInit(), mgExit(), mgGetLastError(),
    mgGetPrevious(), mgGetName(), mgFree(),
    mgWalk(), mgWalkEx(), mgWalkGetMatrix(),
```

```
    mgMoreDetail(), mgOpenDb(), mgCloseDb()

 ****************************************************************************/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

static mgbool PrintRecName (mgrec* db, mgrec* parent, mgrec* rec, void *id)
{
   char *name;
   char *cid = id;

   if (!mgGetPrevious (rec))
      printf ("\n");
   if (name = mgGetName (rec)) {
      printf ("%s\t", name);
      strcpy (cid, name); /* save name in user data... for whatever reason */
      mgFree (name); /* mgGetName allocs, user must dealloc */
   }

   return (MG_TRUE);
}

static mgbool PrintMatrix (mgrec* db, mgrec* parent, mgrec* rec, void *id)
{
   mgmatrix matrix;
   char formatString[50];
   int fieldWidth = 8;

   /* get accumulated matrix at this point of the traversal,
      must send id to identify ourselves */
   mgWalkGetMatrix (id, &matrix);

   sprintf (formatString, "%%%d.3lf %%%d.3lf %%%d.3lf %%%d.3lf",
            fieldWidth, fieldWidth, fieldWidth, fieldWidth);

   /* print out the elements of the 4x4 matrix */
   printf (formatString, matrix[0], matrix[1], matrix[2], matrix[3]);
   printf (formatString, matrix[4], matrix[5], matrix[6], matrix[7]);
   printf (formatString, matrix[8], matrix[9], matrix[10], matrix[11]);
   printf (formatString, matrix[12], matrix[13], matrix[14], matrix[15]);
   return (MG_TRUE);
}

void main (int argc, char* argv[])
{
   mgrec* db;
   char idname[80];
   char *id = idname;

   /* check for correct number of arguments */
   if (argc < 2) {
      printf ("Usage: %s <input_db_filename>\n", argv[0]);
```

```
      exit (EXIT_FAILURE);
   }

   /* always call mgInit() before any other OpenFlight API calls */
   mgInit (&argc, argv);

   /* open database */
   if (!(db = mgOpenDb (argv[1]))) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      mgExit ();
      exit (EXIT_FAILURE);
   }

   /* traverse the database without visiting vertices, or references */
   printf ("\nMWALK_NORDONLY\n");
   mgWalk (db, PrintRecName, MG_NULL, id, MWALK_NORDONLY);
   printf ("\n");

   /* traverse the database, visiting vertices and the first instance of each
      referenced node */
   printf ("\nMWALK_NORDONLY + MWALK_VERTEX + MWALK_MASTER\n");
   mgWalk (db, MG_NULL, PrintRecName, id, MWALK_NORDONLY
                                        + MWALK_VERTEX
                                        + MWALK_MASTER);
   printf ("\n");

   /* traverse the database, visiting vertices and every instance of each
      referenced node */
   printf ("\nMWALK_NORDONLY + MWALK_MASTER + MWALK_MASTERALL\n");
   mgWalk (db, PrintRecName, MG_NULL, id, MWALK_NORDONLY
                                        + MWALK_VERTEX
                                        + MWALK_MASTER
                                        + MWALK_MASTERALL);
   printf ("\n");

   /* traverse the database, printing the accumulated matrix stack at the nodes */
   printf ("\nMWALK_MATRIXSTACK\n");
   mgWalkEx (db, MG_NULL, PrintMatrix, MG_NULL, id, MWALK_MATRIXSTACK);
   printf ("\n");

   /* cycle through the database's levels of detail, traverse and visit
      only the records associated with the current level of detail */
   while (mgMoreDetail (db)) {
      printf ("\n New Level Of Detail: MWALK_NORDONLY + MWALK_ON\n");
      mgWalk (db, PrintRecName, MG_NULL, id, MWALK_NORDONLY
                                        + MWALK_ON);
      printf ("\n");
   }

   /* close the database */
   mgCloseDb (db);

   /* always call mgExit() after all OpenFlight API calls */
   mgExit ();
}
```

Manual Traversal

Manual traversal gives you the most control over the traversal. Because you control the traversal, you can perform different action functions for different types of nodes. However, manual traversal requires a greater understanding of factors such as the database hierarchy, instancing, and so forth, to perform traversal and action functions without introducing errors.

There are many ways to do manual traversal. This section illustrates a few.

Traversal Example 1

This example uses **mgGetChild** and **mgGetNext** and traverses only simple relationships. Instances, external references, and nested children are ignored:

```
static void SimpleTraverse (mgrec *node)
{
   /* traverse a simple database with no instances,
      external references, or subfaces */

   int i;
   static int num_tabs = 0;

   if (!node)
      return;

      /* indent and print name */
   for (i = 0; i < num_tabs; i++)
      printf ("\t");
   printf ("%s\n", mgGetName (node));

      /* traverse down */
   num_tabs++;
   if (mgGetChild (node))
      SimpleTraverse (mgGetChild (node));
   num_tabs--;

      /* traverse right */
   if (mgGetNext (node))
      SimpleTraverse (mgGetNext (node));
}
```

Traversal Example 2

This example adds additional detail to the traversal, checking for more complex relationships:

```
static void Traverse1 (mgrec *node)
{
   /* traverse a general database, checking for
```

```
      instances, external references, and subfaces */

   int nodetype;
   register int i;
   static int num_tabs = 0;

   if (!node)
      return;

   /* indent and print name */
   for (i = 0; i < num_tabs; i++)
      printf ("\t");
   printf ("%s\n", mgGetName (node));

   nodetype = mgGetCode (node);

   /* traverse down to the vertex level;
      don't try to traverse external references  */
   if (nodetype != fltVertex)
   {
      num_tabs++;
      if (mgGetChild (node))          /* children */
         Traverse1 (mgGetChild (node));
      if (mgGetNestedChild(node))   /* subfaces */
         Traverse1 (mgGetNestedChild (node));
      if (mgGetReference (node)          /* this is an instanced node */
         && mgIsFirstInstance (node))   /* only step into 1st instance */
         Traverse1 (mgGetReference (node));
      num_tabs--;
   }

   /* traverse right */
   if (mgGetNext (node))
      Traverse1 (mgGetNext (node));
}
```

Traversal Example 3

This example illustrates how to descend into instances and external references:

```
static void ListInstances (mgrec *node)
{
   mgrec *thisnode = node;

   printf ("listing instances of %s \n", mgGetName (node));

   thisnode = mgGetFirstInstance (node);
   while (thisnode)
   {
      printf ("\tinstance of %s \n", mgGetName (thisnode));
      thisnode = mgGetNextInstance (thisnode);
   }
}

static void Traverse2 (mgrec *node)
```

```
{

      /* traverse down to the vertex level,
         going into external references */

   int nodetype;
   register int i;
   static int num_tabs = 0;
   mgrec *thisnode = node;

   while (thisnode)
   {
      /* indent and print name */
      for (i = 0; i < num_tabs; i++)
         printf ("\t");
      printf ("%s\n", mgGetName (thisnode));

      nodetype = mgGetCode (thisnode);

      switch (nodetype)
      {
         case eFltVertex:
            break;
         case eFltXref:
         {
            char filename[100];
            mgGetAttBuf (node, fltXrefFilename, filename);
            newDb = mgOpenDb (filename);
            Traverse2 (newDb);
            break;
         }
         default:
         {
            num_tabs++;
            if (mgGetChild (thisnode))        /* children */
               Traverse2 (mgGetChild (thisnode));
            if (mgGetNestedChild (thisnode))  /* subfaces */
               Traverse2 (mgGetNestedChild (thisnode));
            if (mgIsInstance (thisnode)          /* instanced node-- only */
               && mgIsFirstInstance (thisnode))  /* traverse one instance */
               ListInstances (thisnode);
            num_tabs--;
         }
      }

      /* traverse right */
      thisnode = mgGetNext (thisnode);
   }
}
```

Manual Traversal Using Structure Functions

The API provides two other structure management functions: **`mgCountChild`** and **`mgGetRecByName.`** These routines are convenience

functions that do the traversal for you; their performance is equivalent to the other traversal methods.

Note that if you already obtained a node ID using another function, it is more efficient to store a node id for reuse than to do repeated traversals.

Creating New Nodes

You can create a new node by calling **mgNewRec**. The type of node that is created is specified by the **mgcode** parameter you pass. The node created by **mgNewRec** is permanently bound to the current database and can only be inserted into hierarchy contained in the current database. If you want to create a new node that can be inserted into a specific database hierarchy (other than the current database), use **mgNewRecDb**.

New node records are considered orphan (not attached to any other node) until you insert them into the hierarchy by calling **mgAttach**, **mgAppend**, **mgInsert**, or **mgReference**. **mgWriteDb** does not save orphan nodes to the OpenFlight file; these nodes are lost when you call **mgCloseDb**.

```
void OpenFile (const char *filename)
{
   mgrec *db, *grec1, *grec2;

   db = mgNewDb (filename);

   grec1 = mgNewRec (fltGroup);
   mgAttach (db, grec1);

   grec2 = mgNewRec (fltGroup);
   mgAttach (grec1, grec2);
}
```

The functions **mgAttach**, **mgAppend**, and **mgInsert** all attach a node (and its descendents) into a hierarchy. **mgAttach** attaches a node to a parent as the *first* child, **mgAppend** attaches a node to a parent as the *last* child and **mgInsert** attaches a child to a parent *after* a given sibling. There is a significant computational difference between **mgAttach**, **mgInsert** and **mgAppend** in terms of how fast each function executes. **mgAttach** and **mgInsert** are both of constant order while **mgAppend** is of order(N) where N is the number of children currently attached to the parent. In other words, **mgAppend** can take significantly longer to execute when called for a parent that already contains many children. That is because **mgAppend** must find the end of the parent's child list before attaching the new node.

Here are some things to consider when attaching new nodes in the hierarchy:

- When it does not matter if your new node goes at the beginning or end of a parent's child list, use **mgAttach**.
- If your new node must go at the end of a parent's child list and you cannot avoid doing so, use **mgAppend**.

If you must attach several new nodes to the end of a parent's child list, attach the first using **mgAppend** and the subsequent nodes using **mgInsert** as shown in the following code fragment:

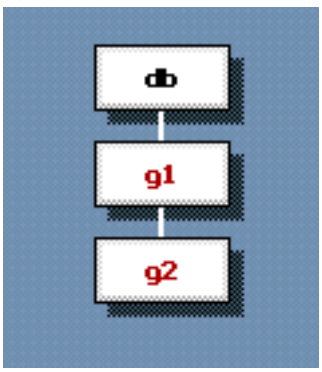
```
void CreateObjects (mgrec* group)
{
   // create 3 new objects attached to the end
   // of group's child list
   mgrec* newObject;
   mgrec* insertAfter;

   newObject = fltNewRec (fltObject);
   // have no choice but to use mgAppend
   // for the first object
   mgAppend (group, newObject);
   // save this object to insert after
   insertAfter = newObject;

   newObject = fltNewRec (fltObject);
   // use mgInsert, it is faster than mgAppend
   mgInsert (insertAfter, newObject);
   insertAfter = newObject;

   newObject = fltNewRec (fltObject);
   // use mgInsert, it is faster than mgAppend
   mgInsert (insertAfter, newObject);
   insertAfter = newObject;
}
```

When you create a new database, you should start out with a minimum hierarchy that looks like this:



Historically, OpenFlight databases have had at least two group nodes below the header. Some real-time applications have grown to depend on this convention, so it is safest for you to follow it as well.

You can create the minimum hierarchy using this code:

```
mgrec *newdb = mgNewDb ("foo.flt");
mgrec *grec1 = mgNewRec (fltGroup);
mgrec *grec2 = mgNewRec (fltGroup);

mgAttach (newdb, grec1);
mgAttach (grec1, grec2);
```

To add an instance, call **mgReference**. To add an external reference, use **mgNewRec** just as you would to create any type of node. Then to specify the name of the external file to reference, set the **fltXrefFilename** attribute as follows:

```
mgrec *newdb = mgNewDb ("foo.flt");
mgrec *grec1 = mgNewRec (fltGroup);
mgrec *grec2 = mgNewRec (fltGroup);
mgrec *xRef = mgNewRec (fltXref);

mgSetAttList (xRef, fltXrefFilename, "bar.flt", MG_NULL);

mgAttach (newdb, grec1);
mgAttach (grec1, grec2);
mgAttach (grec2, xRef);
```

This example creates a database that externally references file **bar.flt**.

Sample: egstruct2.c

The following example creates several types of nodes with a variety of relationships:

```
/*****************************************************************************

  Sample file: EGSTRUCT2.C

  Objective: Show how to create, delete, duplicate, attach, and move records
     around in an OpenFlight database.

  Program functions: Create new database with filename from command line.
     Create a group with 3 child groups.
     Create object, polygon, and vertex records with
     certain attributes.  Duplicate the object record and attach
     it under a group.  Share the object with the other group.
     Write the file.

  API functions used:
     mgNewRec(), mgDuplicate(), mgAttach(), mgAppend(),
     mgInsert(), mgReference(), mgSetVtxCoord()
```

```
 ******************************************************************************/

#include <stdio.h>
#include <stdlib.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

static void AddVertex (mgrec *db, mgrec *prec, double x, double y, double z)
/* add a vertex to a polygon */
{
   mgrec *vrec;
   vrec = mgNewRec (fltVertex);
   mgAppend (prec, vrec);
   mgSetCoord3d (vrec, fltCoord3d, x, y, z);
}

static mgrec* MakePoly (mgrec *db, unsigned int color, double offsetx, double offsety,
double offsetz)
/* creates a new polygon record with 4 vertices, */
/* returns ptr to new polygon record */
{
   mgrec* prec;
   double ic0[3] = {0., 0., 0.};
   double ic1[3] = {100., 0., 0.};
   double ic2[3] = {100., 100., 0.};
   double ic3[3] = {0., 100., 0.};

   /* make polygon, attach to object */
   prec = mgNewRec (fltPolygon);

   /* make vertices, attach to polygon */
   AddVertex (db, prec, ic0[0]+offsetx, ic0[1]+offsety, ic0[2]+offsetz);
   AddVertex (db, prec, ic1[0]+offsetx, ic1[1]+offsety, ic1[2]+offsetz);
   AddVertex (db, prec, ic2[0]+offsetx, ic2[1]+offsety, ic2[2]+offsetz);
   AddVertex (db, prec, ic3[0]+offsetx, ic3[1]+offsety, ic3[2]+offsetz);

   /* set color */
   mgSetAttList (prec, fltPolyPrimeColor, color, MG_NULL);

   return prec;
}

void main(int argc, char* argv[])
{
   mgrec *db;     /* top record of database file specified on command line */
   mgrec *grec1;     /* group record created for new database file */
   mgrec *grec2;     /* group record created for new database file */
   mgrec *grec3;     /* group record created for new database file */
   mgrec *grec4;     /* group record created for new database file */
   mgrec *orec1;     /* object record created for new database file */
   mgrec *orec2;     /* object record created for new database file */
   mgrec *prec1;     /* polygon record created for new database file */
   mgrec *prec2;     /* nested polygon record created for new database file */
   unsigned int blue, red;    /* color indices */
   float inten;      /* color intensity */
```

```
/* check for correct number of arguments */
if (argc < 2) {
   printf ("Usage: %s <create_db_filename>\n", argv[0]);
   exit (EXIT_FAILURE);
}

/* Always call mgInit() before any other OpenFlight API calls */
mgInit (&argc, argv);

/* open the database file with the name specified on the command line */
/* store the top record ptr in db */
mgSetNewOverwriteFlag (MG_TRUE);
if (!(db = mgNewDb (argv[1]))){
   char msgbuf [1024];
   mgGetLastError (msgbuf, 1024);
   printf ("%s\n", msgbuf);
   exit (EXIT_FAILURE);
}

/* create a group record and attach it to the */
/* new database so it isn't empty */
grec1 = mgNewRec (fltGroup);
mgAttach (db, grec1);

/* create 3 child groups under the first group */
grec2 = mgNewRec (fltGroup);
mgAttach (grec1, grec2);
grec3 = mgNewRec (fltGroup);
mgAttach (grec1, grec3);
grec4 = mgNewRec (fltGroup);
mgAttach (grec1, grec4);

/* now create an object which is not attached to the database */
orec1 = mgNewRec (fltObject);

/* get color indices for blue and red */
mgRGB2Index (db, 0, 0, 255, &blue, &inten);
mgRGB2Index (db, 255, 0, 0, &red, &inten);

/* create a polygon with nested polygon, attach to object */
prec1 = MakePoly (db, blue, 0., 0., 0.);
prec2 = MakePoly (db, red, 50., 50., 0.);
mgAttach (prec1, prec2);
mgAttach (orec1, prec1);

/* now share the object between the second and fourth groups */
mgReference (grec2, orec1);
mgReference (grec4, orec1);

/* now duplicate the object  */
/* and attach under the third group */
orec2 = mgDuplicate (orec1);
mgAttach (grec3, orec2);

/* write database file */
mgWriteDb (db);
```

```
   /* close database */
   mgCloseDb (db);

   /* always call mgExit() after all OpenFlight API calls */
   mgExit ();
}
```

Deleting Nodes

To delete a node entirely, call **`mgDelete`**. When you do, the node is detached and deleted. To remove a node from the hierarchy without deleting it, call **`mgDetach`**.

To delete an instance, call **`mgDeReference`**. It is an error to call **`mgDelete`** on a reference node without deleting all the instances first.

Note: Detach, delete, reference, and dereference operations treat a node and all of its descendants as a unit.

4

Attribute Records

The previous chapter discussed how to navigate the database hierarchy. This chapter describes how to work with the data itself.

Most information in the database is stored as attributes. This chapter describes the types of attributes and how to work with them.

Record Codes

Every type of record in the database is identified by its record code. There are record codes that describe node types, node attributes, database palettes, palette entries, and so forth. You can retrieve an **`mgrec`** for every record code, but since, in many cases, the attribute being stored has a familiar C data type like **`int`** or **`float`**, it is often easier to bypass the **`mgrec`** and work with the attribute's value directly.

The easiest way to determine an attribute's data type is to look up its record code in the *OpenFlight API Reference*. However, you can also find the data type of an existing **`mgrec`** programmatically, with **`mgGetType`**.

To view the complete list of record codes and attribute types, see the *OpenFlight API Reference*. This list also identifies each attribute's data type. Record codes listed with "record" types are nested attributes.

Storing and Retrieving Attributes

The data type of an attribute determines how the attribute is stored and retrieved. Many attributes have *convenience functions* to make access easier. For example, you can find out a node's type by retrieving the **`fltCode`** attribute, or you can simply call **`mgGetCode`**. There are convenience functions for features such as:

- node type
- node name
- database traversal
- palette management
- transformations

Technically, it is possible to write almost any API program using nothing but **`mgGetAttList`** and **`mgSetAttList`**. However, it is important to use the convenience functions when they are available, because they often perform necessary pre- or post- processing actions on the data.

The following sections describe the access methods for most attributes.

Simple Attributes

Most attributes can be represented by a single number, letter, or flag. For these simple attributes, the API provides a generic access method. The syntax is as follows:

```
mgGetAttList (rec,
              att1code, &val1,
              att2code, &val2,
              ...
              attncode, &valn,
              MG_NULL);

mgSetAttList (rec,
              att1code, val1,
              att2code, val2,
              ...
              attncode, valn,
              MG_NULL);
```

`attncode` is one of the record codes listed in the *OpenFlight API Reference*. Declare **`valn`** with the corresponding type listed in this document.

Nested Attributes

Attribute records are arranged hierarchically. For example, the **fltPolygon** record code describes a record's node type; within **fltPolygon** there are record codes such as **fltPolyColor** and **fltPolyMaterial**, which describe drawing attributes of the polygon.

Some attributes have further nesting; for example, the **fltGroup** node type has a **fltBoundingBox** record, which in turn has attributes of type **fltCoord3d** and **fltBoundingSphere**. You can access nested attributes in the same way as you access simple ones; the nesting is transparent to you. You need to specify only the node record and the desired attribute's record code. For example:

```
mgGetAttList (groupNode,
    fltGrpPrio,   &prio,
    fltBCRadius,  &radius
    fltBCHeight,  &height,
    fltBYaw,      &yaw,
    MG_NULL);
```

The same holds true for record creation. When you specify the values for a nested attribute, the API creates the nested records for you:

```
mgSetAttList (groupNode,
    fltGrpPrio,   prio,
    fltBCRadius,  radius
    fltBCHeight,  height,
    fltBYaw,      yaw,
    MG_NULL);
```

Text Strings

Several common text string attributes have convenience functions:

mgGetName
mgSetName
mgGetComment
mgSetComment
mgDeleteComment

In general, attributes that are marked as type **char* (N)** where **N** is the maximum length of the string, are retrieved in a special way. The following shows how to get a general text attribute, in this case the **fltHdrLastDate** attribute of the **fltHeader** record:

```
char* dateString = MG_NULL;
```

```
// mgGetAttList allocates a string, guaranteed
// to be large enough and fills in the string
// with the attribute value

mgGetAttList (header, fltHdrLastDate, &dateString, MG_NULL);

// perform operations with dateString

// free it when done

mgFree (dateString);
```

Setting the value of a text attribute is much more straightforward. The following shows how to set a general text attribute, again using the **fltHdrLastDate** attribute of the **fltHeader record**:

```
mgSetAttList (header, fltHdrLastDate, "Mon Apr 21
16:03:17 2003", MG_NULL);
```

Coordinates and Vectors

Some nested attributes, like coordinates, normals, and colors, appear so frequently in the attribute hierarchy that it is easiest to think of them like simple attributes. These attributes have helper functions named according to the type they access, of the form **mgGet<Type>** and **mgSet<Type>**.

```
mgGetCoord3f / mgSetCoord3f
mgGetCoord3d / mgSetCoord3d
mgGetCoord2i / mgSetCoord2i
mgGetVector / mgSetVector
mgGetColorRGBA / mgSetColorRGBA
mgGetPolyColorRGB / mgSetPolyColorRGB
mgGetPolyAltColorRGB / mgSetPolyAltColorRGB
mgGetVtxColorRGB / mgSetVtxColorRGB
mgGetVtxColorRGBA / mgSetVtxColorRGBA
mgGetNormColor / mgSetNormColor
```

Transformations

A transformation matrix allows you to position one or more nodes in the database without modifying the nodes' vertex coordinates. The transformation is applied to all the descendants of the node containing the matrix. Transformation matrices are especially useful with instance node and external reference, where you want to position a single object in several

places. You can apply a transformation matrix to any node type except **fltPolygon**, **fltVertex**, and **fltDof**.

A transformation matrix is an efficient way to apply several transformations at once. However, editing a transformation matrix is not intuitive. It is usually more convenient to describe the specific transformation element you want to apply (translate, scale, and so forth).

OpenFlight supports several transformation types:
fltXmTranslate
fltXmRotate
fltXmRotateEdge
fltXmScale
fltXmScaleToPoint
fltXmPut
fltXmGeneral

These are explained in the *OpenFlight API Reference*. You can attach several transformation types to a node, and the API computes the resultant matrix for you.

Transformation records do not appear in a standard hierarchy traversal. Instead, you must query each node to find a transformation:

mgHasXform determines whether a node has a transformation list

mgGetXform returns the first transformation on a node

mgGetXformType determines the type of transformation

You can access the transformation fields the same way you get other attributes:

mgGetAttList retrieves a given transformation field

mgSetAttList sets a given transformation field

You create transformation list elements the same way you create nodes:

mgNewRec creates a record with a specific transformation type

mgAttach attaches a transformation as the first entry in a node's transformation list

mgAppend attaches a transformation as the last entry in a node's transformation list

mgInsert inserts a transformation at any point in a node's transformation list

mgDetach detaches a transformation from a node's transformation list

mgDelete deletes a transformation from a node's transformation list

mgGetNext gets subsequent entries in the list (NULL-terminated)

The following example adds a rotation element to a node.

AddXForms

```
static void AddXForms (mgrec *node)
{
   xformrec = mgNewRec (fltXmRotate);

   center.x = 1.0;
   center.y = 2.0;
   center.z = 3.0;

   axis.i = 4.0f;
   axis.j = 5.0f;
   axis.k = 6.0f;

   angle = 7.0f;

   if (!mgSetCoord3d (xformrec, fltXmRotateCenter,
         center.x, center.y, center.z))
      printf ("Couldn't set rotate center\n");

   mgSetAttList (xformrec,
         fltVectorI, axis.i,
         fltVectorJ, axis.j,
         fltVectorK, axis.k,
         mgNULL);

   mgGetAttList (xformrec,
         fltVectorI, &a,
         fltVectorJ, &b,
         fltVectorK, &c,
         mgNULL);

   mgSetAttList (xformrec, fltXmRotateAngle, angle, mgNULL);
   mgAttach (grec, xformrec);
}
```

The following example examines the transformations on a node and prints out translate and rotate elements.

PrintXForms

```
static void PrintXforms (mgrec *node)
{
```

```
   if (mgHasXform (node))
   {
      xrec = mgGetXform (node);
      xtype = mgGetXformType (xrec);

      mgGetAttList (xrec, fltXmLimitMax, &maxlimit, mgNULL);
      printf ("XFORM Limit Max = %f\n", maxlimit);

      switch (xtype)
      {
         case XLL_TRANSLATE:
         {
            double x, y, z;

            mgGetCoord3d (xrec, fltXmTranslateFrom, &x, &y, &z);
            printf ("TranslateFrom xyz:  %f, %f, %f\n", x, y, z);
            break;
         }
         case XLL_ROTPT:
         {
            float angle;
            double x, y, z;
            float i, j, k;

            mgGetCoord3d (xrec, fltXmRotateCenter, &x, &y, &z);
            printf ("RotateCenter: %f, %f, %f\n", x, y, z);
            mgGetAttList (xrec,
               fltVectorI, &i,
               fltVectorJ, &j,
               fltVectorK, &k,
               mgNULL);
            printf ("RotateAxis: %f, %f, %f\n", i, j, k);
            mgGetAttList (xrec, fltXmRotateAngle, &angle, mgNULL);
            printf ("RotateAngle: %f\n", angle);
            break;
         }
      }
   }
}
```

The OpenFlight API computes a matrix for each transformation on a node and multiplies them together to compute the node's overall matrix. If you are reading a database and are only interested in this resultant matrix, you can call **`mgGetMatrix`** rather than traverse the transformation list. You can also call **`mgGetMatrix`** on any of the individual transformations in the list.

There are no functions to set these matrices directly, since the OpenFlight API recomputes it (and overwrites it). To attach a simple matrix to a node, attach a **`fltXmGeneral`** transformation to the node. Remember that you can apply more than one transformation element to a node; to change a matrix once you've attached it, you must either edit the existing one or delete it and add a new one.

5

Node Records

"The OpenFlight Hierarchy" on page 46 described the relationships between node records in the hierarchy. This chapter discusses individual nodes in more detail.

In memory, the API defines a generic (polymorphic) record called **mgrec** to represent nodes in the OpenFlight Scene Graph hierarchy. Each node record has a permanent record code assigned. This code (**mgcode**) identifies the "type" of node represented by the record. The node type (group, object, polygon, etc.) defines the node's role in the hierarchy. To query a node's type, use **mgGetCode.** To query whether a node is a specific type, use **mgIsCode.**

When a node is created, it is permanently bound to a specific database. This means that the node can only be attached to the hierarchy of the specific database to which it is bound. To query the database to which a node is bound, use **mgRec2Db**.

Each type of node has a distinctive set of attributes applicable to that node type. Node attributes are data (flags, numerical values, and text) saved with each node in the database hierarchy. They are used to represent node-specific information such as name or color.

It is important to know a node's type before you try to access any attributes. Some node attributes are common to all node types, but most are unique to a specific node type. An attempt to access attributes that are not valid for a given node type returns an error.

This chapter describes the different kinds of nodes and their attributes.

Generic Node Attributes

Some attributes are common to most node types. These are described below.

Name

Every node has a unique name. Use **`mgGetName`** or **`mgSetName`** to query or set a node's name.

Note: **`mgGetName`** allocates storage for the name string; the user is responsible for calling **`mgFree`** to deallocate this memory.

Display State

Most nodes have a flag that indicates whether or not they are enabled for display (turned on). This flag is useful for performing LOD switching. You can use the LOD switching functions **`mgMoreDetail`**, **`mgLessDetail`**, **`mgMostDetail`**, and **`mgLeastDetail`** to turn nodes on and off. You can also use the function **`mgIsFlagOn`** to determine whether a node is on or off. Also, when you use **`mgWalk`** to traverse database elements, you can specify whether nodes that are off should be traversed.

In the Creator program environment, a node that is off will not be drawn in the graphic view of its database. A plug-in can turn individual nodes on and off by setting the **`fltIOn`** attribute.

Note: The **`fltIOn`** attribute of a node is a runtime attribute only and does not apply to nodes of type **`fltHeader`** and **`fltVertex`**. It is not saved with the OpenFlight database.

User Data

When writing programs that use the API, you may want to store your own data with a node. The function **`mgSetUserData`** lets you attach a **`void*`** pointer to any node. This can be a pointer to any data that you want to associate with the node. To retrieve a pointer to this data later on, use the

function **mgGetUserData**. User data attached to a node is only valid while your program is running. It is not stored in the OpenFlight database file.

Note: These functions are disabled if called in the Creator modeling environment. This is to prevent user data assigned by multiple plug-ins from colliding. Plug-ins running in Creator can use *Database Node Properties* to attach user data to nodes in the Creator modeling environment. Nodes maintain a separate property list for each plug-in to prevent collision. Refer to the *API Developer Guide Level 3/4* for more information.

Comment

All node types except vertex have a comment attribute. The comment is an ASCII string that can contain any text. Comments are optional attributes. Use **mgSetComment** to add a comment to a node or replace an existing comment. Use **mgGetComment** to retrieve a comment from a node, if one exists. Use **mgDeleteComment** to delete a comment from a node.

Bounds

Nodes that contain geometry have an implied "bounding box" that contains this geometry. Use **mgGetBounds** to retrieve the bounding box of a node and its children.

Node Types and Their Attributes

Each node type has a uniquely defined set of attributes. The following sections provide an overview of the various node types (returned by **mgGetCode**) and some of the attributes unique to them. For a comprehensive list of the node types and their associated attributes, see the *OpenFlight API Reference*.

fltHeader Nodes

The database header node is at the top of the hierarchy. It is created automatically for each new file. The database header node's attributes include things like the palettes, the database units, and the file's OpenFlight version number. They also include preference controls that apply to individual databases, such as whether or not to save normals with the database.

Related Functions:
mgRec2Filename

Related Record Codes:
fltHeader

fltVertex Nodes

Vertices are coordinate points in the database. The unit of the database coordinate system is determined by the **fltHdrUnits** attribute of the database header node.

While morphing between LODs, each vertex in an LOD is allocated a morph vertex - usually the closest vertex in the next lower LOD. Use **mgAttach** to assign a morph vertex to a vertex in the database. Use **mgGetMorphVertex** to retrieve the morph vertex assigned to a vertex (not **mgGetChild**).

Related Functions:
mgGetVtxColorRGB
mgSetVtxColorRGB
mgGetVtxColorRGBA
mgSetVtxColorRGBA
mgGetVtxBackColorRGB
mgSetVtxBackColorRGB
mgGetVtxColorName
mgSetVtxColorName
mgGetVtxCoord
mgSetVtxCoord
mgGetVtxNormal
mgSetVtxNormal
mgGetMorphVertex

Related Record Codes:
fltVertex

fltPolygon Nodes

A polygon, or face, is a collection of ordered, coplanar vertices describing a surface. The order of the vertices in the polygon define which side of the polygon is front and which is back. When looking at the *front* of a polygon, the vertices will be ordered in a counter-clockwise direction. The polygon normal will point towards you if you are looking down on the front of the

polygon. If you are looking at the back of the polygon, the normal will point away from you.

In general, a **fltPolygon** with three or more vertices is treated as a polygon, a **fltPolygon** with two vertices is treated as a line, and a **fltPolygon** with only one vertex is treated as a fixed-size point. However, you can set **fltPolygon** attributes that force the interpretation of the polygon as wireframe. For optimal rendering performance, use convex polygons (rather than concave polygons) where possible.

In the database hierarchy, one **fltPolygon** attached to another is called a *nested polygon*. Nested polygons provide a useful means of specifying drawing order on z-buffered systems, because in z-buffered drawing mode, every nested child draws on top of its parent. Each nested polygon should lie within, and on the same plane as, its parent. Nested polygons can, in turn, be the parents of other nested polygons. You use **mgAttach**, **mgAppend** and **mgInsert** to attach nested polygons to polygons just like other children.

Related Functions:

mgGetNestedParent
mgGetNestedChild
mgGetPolyColorRGB
mgSetPolyColorRGB
mgGetPolyAltColorRGB
mgSetPolyAltColorRGB
mgGetPolyColorName
mgSetPolyColorName
mgGetPolyNormal
mgIsPolyConcave

Related Record Codes:

fltPolygon

fltMesh Nodes

A mesh defines a set of related polygons, each sharing common attributes and vertices. Using a mesh, related polygons can be represented in a more compact format than would be required to represent each of the polygons separately. That is because only one copy of the polygon attributes is stored per mesh, regardless of how many actual polygons are represented in the mesh.

A mesh is defined by a set of 'polygon' attributes (color, material, texture, etc.), a common 'vertex pool' and one or more 'geometric primitives' that use the shared attributes and vertices. Each geometric primitive (or simply mesh primitive) of a mesh represents either a triangle strip, triangle fan, quadrilateral strip or indexed polygon.

The attributes of a mesh node (**`fltMesh`**) are identical to those of a normal polygon (**`fltPolygon`**) node. In fact, the attribute names used for **`fltMesh`** nodes are the exact same names used by **`fltPolygon`** nodes. For example, the texture index of a **`fltMesh`** node is accessed by attribute **`fltPolyTexture`** just as is the texture index of a **`fltPolygon.`** This unifies access to these two node types.

There are two additional attributes that are part of the **`fltMesh`** node that are not applicable to **`fltPolygon`** nodes. They are:

`fltMeshNumVtx` - the number of vertices in the vertex pool of the mesh.

`fltMeshNumPrimitives` - the number of primitives in the mesh.

Vertex Pool

The vertex pool of a mesh represents the vertices that are used by the primitives of the mesh. Each vertex in the pool includes one or more of the following kinds of data:

- Coordinate position - The x, y and z coordinates of the vertex.
- Color - The color of the vertex. This attribute may be represented by either a color index and intensity pair or RGBA color components.
- Normal - The i, j and k components of the vertex normal.
- Texture coordinates - The u and v texture coordinates for each applicable texture layer.

Each vertex in the pool must have at minimum, a coordinate position. Then, depending on whether the primitives of the mesh are shaded and/or textured, the vertices may have additional attributes including color, normal and/or texture coordinates. Each mesh has a Vertex Mask whose bits define which kinds of data the vertices in the vertex pool have. Use the function **`mgMeshGetVtxMask`** to retrieve the vertex mask for a mesh. You can then check the mask to see if a particular bit is **ON** or **OFF**, indicating whether a

particular kind of data is defined for the vertices in the vertex pool. The bits in the vertex mask are identified by the following constants:

- **MMESH_VTXCOORD** - If this bit is **ON** in the vertex mask, all of the vertices in the vertex pool have x, y, and z coordinate positions. Use functions **mgMeshGetVtxCoord** or **mgMeshSetVtxCoord** to retrieve or set the position of a vertex in the pool.
- **MMESH_VTXCOLOR** - If this bit is **ON** in the vertex mask, all of the vertices in the vertex pool have color values identified by index and intensity values. Use functions **mgMeshGetVtxColor** or **mgMeshSetVtxColor** to retrieve or set the color index and intensity of a vertex in the pool. Use **mgMeshGetVtxColorAlpha** or **mgMeshSetVtxColorAlpha** to retrieve or set the alpha color component of a vertex in the pool.
- **MMESH_VTXCOLORRGB** - If this bit is **ON** in the vertex mask, all of the vertices in the vertex pool have color values identified by RGBA (red, green, blue and alpha) components. Use functions **mgMeshGetVtxColorRGB, mgMeshGetVtxColorRGBA,** or **mgMeshGetVtxColorAlpha** to retrieve RGB, RGBA or alpha color components, respectively, of a vertex in the pool. Use **mgMeshSetVtxColorRGB, mgMeshSetVtxColorRGBA** or **mgMeshSetVtxColorAlpha** to set the RGB, RGBA or alpha color components, respectively, of a vertex in the pool.
- **MMESH_VTXNORMAL** - If this bit is **ON** in the vertex mask, all of the vertices in the vertex pool have normals. Use functions **mgMeshGetVtxNormal** or **mgMeshSetVtxNormal** to retrieve the normal of a vertex in the pool.
- **MMESH_VTXUV[0-7]** - If any of these bits are **ON** in the vertex mask, all of the vertices in the vertex pool have u and v texture coordinates for the corresponding texture layer. Use functions **mgMeshGetVtxUV** or **mgMeshSetVtxUV** to retrieve or set the texture coordinates for the corresponding layer of a vertex in the pool.

As mentioned above, a vertex that has color may have either color index and intensity or RGB components but not both. If a vertex has color, you can call any of the functions **mgMeshGetVtxColorRGB, mgMeshGetVtxColorRGBA** or **mgMeshGetVtxColor** and the API will convert the color of the vertex to the desired format. For example, if the vertices in the vertex pool have color index/intensity values and you call **mgMeshGetVtxColorRGB,** it will look up the index/intensity values and automatically convert them to their RGB component values and return the RGB components to you.

To achieve efficiency in a mesh, there is one important restriction on all the vertices contained in the vertex pool. That is that each vertex in the pool must have the same kind of data as every other vertex in the pool. So if you define the vertex pool for a mesh to contain coordinate positions and normals only, each vertex in the pool must have a coordinate position and normal (and only a coordinate position and normal).

The order in which the vertices appear in the vertex pool is not important with respect to the pool itself. The position of a vertex in the pool is only significant with respect to the primitives of the mesh. That is because the primitives defined in the mesh reference vertices in the pool by position (or index). The index of a vertex in the vertex pool is the position of the vertex in the pool. The index of the first vertex in the pool is 0.

The index of the last vertex in the pool is the value of the mesh attribute **`fltMeshNumVtx`** minus one.

A vertex in the pool can be referenced by more than one primitive. In fact, this is what helps make the mesh more efficient than normal polygons. There is no limit to the number of primitives that reference the same vertex in the pool. And although it is not very common, the same primitive can reference the same vertex in the pool more than one time.

Mesh Primitives

While the vertex pool defines the position and attributes of the vertices in a mesh, mesh primitives define how vertices in the pool are connected to form triangles and/or other polygons. A mesh contains one or more mesh primitives. A single mesh primitive can define one or more related polygons. Each polygon defined by a mesh primitive inherits the "polygon" attributes of the mesh node. Each mesh primitive is defined by its type and its vertex index array. A mesh primitive can be one of the following types:

- Triangle Strip - **`MPRIM_TRI_STRIP`**
- Triangle Fan - **`MPRIM_TRI_FAN`**
- Quadrilateral Strip - **`MPRIM_QUAD_STRIP`**
- Indexed Polygon - **`MPRIM_INDEXED_POLY`**

The vertex index array is an ordered sequence of vertex indices defining the connectivity of the vertices in the primitive and refer to vertices in the vertex pool of the mesh.

Triangle Strip

This type of mesh primitive defines a connected group of triangles. The vertex index array of a triangle strip mesh primitive contains at least 3 vertices. One triangle is defined for each vertex in the index array after the first two. For *n* (starting at 0), the *n*th triangle is defined as follows. For even *n* (0, 2, 4, etc), the *n*th triangle is defined by vertices *n*, *n+1* and *n+2*. For odd n (1, 3, 5, etc), the *nth* triangle is defined by vertices *n+1, n* and *n+2*.

If a triangle strip mesh primitive has *N* vertices, *N-2* triangles are defined.

Triangle Fan

Like the Triangle Strip, this type of mesh primitive also defines a connected group of triangles. The vertex index array of a triangle fan mesh primitive contains at least 3 vertices. One triangle is defined for each vertex in the index array after the first two. For any *n* (starting at 0), the *n*th triangle is defined by vertices 0, *n+1* and *n+2*.

If a triangle fan mesh primitive has *N* vertices, *N-2* triangles are defined.

Quadrilateral Strip

This type of mesh primitive defines a connected group of quadrilaterals (4-sided polygons). The vertex index array of a quadrilateral strip mesh primitive contains an even number of vertices (at least 4). One quadrilateral is defined for each pair of vertices presented after the first pair. For any *n* (starting at 0), the *n*th quadrilateral is defined by vertices *2n, 2n+1, 2n+3* and *2n+2*. If a quadrilateral strip mesh primitive has *N* vertices, *(N/2)-1* quadrilaterals are defined

Indexed Polygon

This type of mesh primitive defines a single polygon. If an indexed polygon mesh primitive has *N* vertices, *1 N*-sided closed polygon or *1 (N-1)*-sided unclosed polygon.

Creating a Mesh

Creating a mesh is slightly different from creating other hierarchical nodes in a database. Mesh nodes do not have children. All of the data for a mesh node, including its vertex pool and primitives is fully contained in the mesh node itself.

To create a mesh, you begin by calling **`mgNewRec`** specifying **`fltMesh.`** This creates a mesh node ready to define and attach in the database. To define a mesh node, you will do the following:

1. Allocate a vertex pool for the mesh using function **`mgCreateVtxPool.`** This function allocates and attaches a vertex pool to a specific mesh node. At the time you call this function, you must know how many vertices are going to be allocated in the pool and what kinds of data each vertex in the pool is going to have. You will specify as parameters to **`mgCreateVtxPool`** the number of vertices in the pool and the vertex mask defining which kinds of data you want for each vertex. The actual size of the vertex pool allocated is dependent on these two values. After you create a mesh, you can change the vertex mask using **`mgMeshSetVtxMask.`**

2. Allocate the primitives for the mesh using function **`mgCreatePrimitive.`** This function allocates and attaches the specified number of primitives to the mesh node. At the time you call this function, you must only know how many primitives are going to be allocated for the mesh. At this time, you need not define what kind of primitive (triangle strip, triangle fan, quad strip, or indexed polygon) each is. Nor must you define how many vertices each primitive is going to have. You can define these later.

3. Define the data for each vertex in the vertex pool you allocated/attached with **`mgCreateVtxPool.`** You will do this using the functions **`mgMeshSetVtxCoord, mgMeshSetVtxColor, mgMeshSetVtxColorRGB, mgMeshSetVtxColorRGBA, mgMeshSetVtxColorAlpha, mgMeshSetVtxNormal,`** and **`mgMeshSetVtxUV`** to define coordinate positions, color index/intensity, RGB colors, RGBA colors, alpha color, normals and texture coordinates respectively. Note that it is an error if you try to define data for a mesh if the vertex mask you used to create the mesh does not include that particular kind of data. For example, creating a mesh with a vertex mask specifying **`MMESH_VTXCOORD, MMESH_VTXCOLOR,`** and **`MMESH_VTXUV0`** calling **`mgMeshSetVtxUV`** for layer 1 would fail since the allocated vertex pool only contains uv data for layer 0. If you want to "add" uv data for layer 1 or any other vertex data not specified when you

create the mesh, use **mgMeshSetVtxMask** to first change the vertex mask of a mesh, then set the additional data.

4. Define the type and vertex index array for each primitive you allocated/attached with **mgCreatePrimitive.** You specify the type for a mesh primitive using the function **mgMeshPrimitiveSetType.** You define the vertex index array of a mesh primitive using the function **mgMeshPrimitiveSetVtxIndexArray.**

5. Attach the completed mesh node into the database hierarchy using **mgAttach** or **mgAppend.**

Accessing a Mesh

As stated above, a mesh node is slightly different from other hierarchical nodes in a database in that its data is fully self-contained. Remember, mesh nodes do not have children so you cannot use **mgGetChild** or **mgWalk** to retrieve data 'below' a mesh.

Given a **fltMesh** node, you can query its data as described below:

- Number of vertices in the vertex pool - Query the **fltMesh** attribute **fltMeshNumVtx** using **mgGetAttList.**
- Number of primitives in the mesh - Query the **fltMesh** attribute **fltMeshNumPrimitives** using **mgGetAttList.**
- Type of a mesh primitive - Use **mgMeshPrimitiveGetType.**
- Vertex Index Array of a mesh primitive - Use **mgMeshPrimitiveGetVtxIndexArray.**
- Data for a vertex in the vertex pool - Use **mgMeshGetVtxCoord, mgMeshGetVtxColor, mgMeshGetVtxColorRGB, mgMeshGetVtxColorRGBA, mgMeshGetVtxColorAlpha, mgMeshGetVtxNormal,** and **mgMeshGetVtxUV** to retrieve coordinate positions, color index/intensity, RGB colors, RGBA colors, alpha color, normals and texture coordinates respectively.

Changing a Mesh

The data stored in the vertex pool of a mesh is very compact. Remember that when you create a mesh, you specify the number of vertices in the vertex pool and the vertex mask defining which kinds of data you want for each vertex. When you do this, a fixed size block of memory is allocated for the vertex pool - just enough to fit the data for the number of vertices you specified. If

you need to change either the number of vertices in a mesh or the kinds of vertex data represented by the mesh, you need to follow the instructions presented in this section.

To change the number of vertices allocated in the vertex pool of a mesh, you really have no option other than to recreate the mesh from scratch. There is no mechanism to expand or contract the number of vertices in the pool. If you need to do this, and you want to preserve some of the data assigned to the original vertices, it is recommended that you make a copy of the original mesh so that you can copy data from it into the new mesh you will need to create.

To change the kind of vertex data represented by the mesh, use **mgMeshSetVtxMask.** The following paragraphs describe how the new vertex pool is initialized under different circumstances when you call **mgMeshSetVtxMask:**

For each kind of vertex data that was specified by both the original vertex mask and the new vertex mask, the corresponding vertex data from the original vertex pool is copied to the correct locations in the new vertex pool. In this way, existing vertex data is preserved in the mesh.

For each kind of vertex data that was specified by the original vertex mask but not by the new vertex mask, the corresponding vertex data from the original vertex pool is discarded.

For each kind of vertex data that was specified by the new vertex mask but not by the original vertex mask, the corresponding vertex data in the new vertex pool is left uninitialized. It is the caller's responsibility to assign values for this new vertex data in the vertex pool.

Mesh Advanced Topics

The vertex pool is a packed sequence of values representing the vertex data for a mesh. In most cases, you need not be concerned how this data is packed in the vertex pool. To this end, the API provides convenient access functions to the vertex pool so you do not have to know how the data is packed. These functions (described above) include:
mgMeshGetVtxCoord, mgMeshGetVtxColor, mgMeshGetVtxColorRGB, mgMeshGetVtxNormal, and **mgMeshGetVtxUV f**or reading a mesh node as well as **mgMeshSetVtxCoord, mgMeshSetVtxColor,**

mgMeshSetVtxColorRGB, mgMeshSetVtxNormal, and **mgMeshSetVtxUV** for writing to a mesh node.

However, you may find it useful in certain situations to use the packed vertex pool rather than extracting the data for each vertex individually. For example, the vertex pool may be used as an argument to OpenGL vertex array functions. This section describes how the data is packed in the vertex pool and describes some additional functions available for extracting the packed data more easily.

The data for vertices in the vertex pool is packed contiguously starting with data for vertex at index 0, followed by data for vertex at index 1, and so forth. For each vertex, the data that is packed depends on the vertex mask of the mesh. The following describes the data packed for each vertex corresponding to each of the bits in the vertex mask:

- **MMESH_VTXCOORD** - If this bit is **ON** in the vertex mask, each vertex contains 3 8-byte double precision floating point values (24 bytes total) representing the coordinate position of the vertex. The x position is first, then y followed by z.
- **MMESH_VTXCOLOR** - If this bit is **ON** in the vertex mask, each vertex contains 1 4-byte integer value representing the color index (integer), intensity (single precision floating point) and alpha color of the vertex. The color index and intensity of the vertex is encoded in the lower 3 bytes of this integer value *N* as follows. The color index is the result of the integer division operation *N*/128. This will be an integer value in the range 0..1023. The intensity is the result of the floating point operation (N mod 128.0f)/128.0f. This will be a floating point number in the range 0.0f..1.0f. The alpha color is encoded in the upper byte of this integer value and is an integer value in the range 0..255, where 0 represents fully transparent and 255 represents fully opaque.
- **MMESH_VTXCOLORRGB** - If this bit is **ON** in the vertex mask, each vertex contains 4 1-byte unsigned integer values representing the RGBA color of the vertex. The red component is first, then green followed by blue and finally alpha. Each color component is a value in the range 0..255. For the alpha color component, 0 represents fully transparent and 255 represents fully opaque.
- **MMESH_VTXNORMAL** - If this bit is **ON** in the vertex mask, each vertex contains three 4-byte single precision floating point values (12 bytes total) representing the normal of the vertex. The i component is first, then j followed by k.

- **`MMESH_VTXUV[0-7]`** - For each of these bits that are **ON** in the vertex mask, each vertex contains two 4-byte single precision floating point values (8 bytes total) representing the texture coordinates for the corresponding texture layer of the vertex. Layer 0, if present is first, followed by layer 1 (if present) and so forth. For each layer, the u component appears first followed by v.

Within the data for a single vertex, the coordinate position values appear first and should always be present. The coordinate position values are followed by vertex color values or vertex color RGB values (if present), then vertex normal values (if present), then finally texture coordinate values (if present).

To retrieve a pointer to the start of the vertex pool of a mesh, use **`mgMeshGetVtxPool.`** The address returned by this function is the address of the first byte of data for the first vertex in the pool. You do not have to worry about the endian format of the data packed in the vertex pool. The API returns the vertex pool to you packed in the correct endian format for the platform on which your application (or plug-in) is running.

The total number of bytes allocated per vertex (which depends on the data included for each vertex) is called the vertex stride. Use **`mgMeshGetVtxStride`** to retrieve the vertex stride of a mesh. Note the vertex stride can also be calculated by adding up the number of bytes required for each kind of data indicated by the vertex mask. For example, if a meshes vertex mask includes bits **`MMESH_VTXCOORD, MMESH_VTXNORMAL`** and **`MMESH_VTXUV0`**, the vertex stride would be:

*(3*8) + (3*4) + (2*4) = 24 + 12 + 8 = 44*

Since the vertex stride is the number of bytes between successive vertices packed in the vertex pool, the start of the data for vertex *N+1* can be calculated by adding the vertex stride to the address of the start of vertex data *N*.

It has been shown that by using the vertex pool and the vertex stride, you can locate the start of the data for any vertex in the vertex pool. Once you find the start of the data for a particular vertex, you still must extract the individual data elements for that vertex. The coordinate positions are easy, they are simply the first three double precision floating point values packed one after the other. Using the vertex mask, you can figure out what comes next -- or you can use the function **`mgMeshGetVtxOffset.`**

You pass **mgMeshGetVtxOffset** the vertex mask bit indicating which kind of data you are interested in and it returns the byte offset to the start of that data (relative to the start of the vertex data). If the data corresponding to the vertex mask bit you specify is not present in the vertex pool, **mgMeshGetVtxOffset** returns `-1`. In the example above where a meshes vertex mask includes bits **MMESH_VTXCOORD**, **MMESH_VTXNORMAL** and **MMESH_VTXUV0**, the vertex offset for **MMESH_VTXCOORD** would be 0, 24 for **MMESH_VTXNORMAL**, 36 for **MMESH_VTXUV0** and -1 for all other vertex mask bits. This means that the vertex normal data for any vertex (3 4-byte single precision floating point values) is located 24 bytes from the start of the data for that vertex.

Remember, you probably will not have to access vertex pool data using the techniques described in this section. In most cases, you can use the convenience functions instead. And if you have a choice, certainly the convenience functions are recommended and are much simpler to user.

Related Functions:

mgMeshGetVtxPool
mgMeshGetVtxMask
mgMeshGetVtxStride
mgMeshGetVtxOffset
mgMeshSetVtxMask
mgMeshPrimitiveGetType
mgMeshPrimitiveGetNumVtx
mgMeshPrimitiveGetVtxIndexArray
mgMeshGetVtxCoord
mgMeshGetVtxColor
mgMeshGetVtxColorRGB
mgMeshGetVtxColorRGBA
mgMeshGetVtxColorAlpha
mgMeshGetVtxNormal
mgMeshGetVtxUV
mgMeshCreateVtxPool
mgMeshCreatePrimitives
mgMeshPrimitiveSetType
mgMeshPrimitiveSetVtxIndexArray
mgMeshSetVtxCoord
mgMeshSetVtxColor
mgMeshSetVtxColorRGB
mgMeshSetVtxColorRGBA
mgMeshSetVtxColorAlpha
mgMeshSetVtxNormal
mgMeshSetVtxUV

Related Record Codes:
`fltMesh`

fltLightPoint Nodes

A light point node is used to represent light points or strings. Its children are limited to a set of vertices that define the position and color of the light(s). A light point may have one or more vertex children, each one representing a different point of light. A light string will have one vertex child, a replication count (attribute `fltRepCnt`) and transformation records. The vertex of a light string defines the position of the first light in the string. The replication count is the number of additional lights in the string while the transformation records represent the matrix to apply to get the positions of successive points in the string as measured from the previous light in the string.

The attributes of a light point node are defined in the Light Point Appearance and Light Point Animation palettes. The light point appearance index and light point animation index attributes specify the appearance and animation palette entries assigned to the light point, respectively. See "Light Point Palette" on page 124 for more information on these palettes.

Related Functions:
`mgGetMatrix`

Related Record Codes:
`fltLightPoint`
`fltLpAnimationPalette`
`fltLpAppearancePalette`

Sample: eglightpoint.c

The following example shows how to use replication count and transformations to build different kinds of light point nodes:

```
/******************************************************************************

  Sample file: EGLIGHTPOINT.C

  Objective: Shows how to create light point node records.
     Shows how to create simple light points which are light point
     nodes with N vertices, each representing a unique light point.
     Shows how to create light "strings" which are light point nodes
     with a single vertex that is "replicated" N times using
     transformation linked lists.
```

```
  Program functions: Create a new database containing different kind of
     light points.

  API functions used:
     mgInit(), mgSetNewOverwriteFlag(), mgNewDb(), mgGetLastError(),
     mgRGB2Index(), mgAttach(), mgAppend(), mgSetAttList(),
     mgSetCoord3d(), mgNewRec(), mgSetVtxNormal(), mgWriteDb(),
     mgCloseDb(), mgExit()

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

#define RUNWAY_WIDTH    100.0
#define RUNWAY_LENGTH   600.0
#define NUM_LIGHTS      20

void main (int argc, char* argv[])
{
   mgrec* db;
   mgrec *grec, *orec, *prec, *vrec, *xrec;
   mgrec *lightPoint, *lightPointAppearance;
   int lpaIndex[2];
   unsigned int grayColor, yellowColor, redColor;
   float grayIntensity, yellowIntensity, redIntensity;
   double lightSpacing;
   int i;

   /* check for proper arguments */
   if (argc < 2) {
      printf ("Usage: %s <create_db_filename>\n", argv[0]);
      exit (EXIT_FAILURE);
   }

   /* always call mgInit before any other API calls */
   mgInit (&argc,argv);

   /* start a new OpenFlight database, overwrite if exists */
   mgSetNewOverwriteFlag (MG_TRUE);
   if (!(db = mgNewDb(argv[1]))) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      mgExit ();
      exit (EXIT_FAILURE);
   }

   /* get a good "gray" color */
   mgRGB2Index (db, 127, 127, 127, &grayColor, &grayIntensity);

   /* get a good "yellow" color */
   mgRGB2Index (db, 255, 255, 0, &yellowColor, &yellowIntensity);
```

```
   /* get a good "red" color */
   mgRGB2Index (db, 255, 0, 0, &redColor, &redIntensity);


/*********************************************************************
   make group/object/polygon for runway
*********************************************************************/

   grec = mgNewRec (fltGroup);
   mgAttach (db, grec);

   orec = mgNewRec (fltObject);
   mgAttach (grec, orec);

   prec = mgNewRec (fltPolygon);
   mgSetAttList (prec,
               fltPolyPrimeColor, grayColor,
               fltPolyPrimeIntensity, grayIntensity,
               MG_NULL);
   mgAttach (orec, prec);


   vrec = mgNewRec (fltVertex);
   mgSetCoord3d (vrec, fltCoord3d, 0.0, 0.0, 0.0);
   mgAttach (prec, vrec);
   mgSetAttList (vrec,
               fltVColor, grayColor,
               fltVIntensity, grayIntensity,
               MG_NULL);

   vrec = mgNewRec (fltVertex);
   mgSetCoord3d (vrec, fltCoord3d, RUNWAY_WIDTH, 0.0, 0.0);
   mgAppend (prec, vrec);

   vrec = mgNewRec (fltVertex);
   mgSetCoord3d (vrec, fltCoord3d, RUNWAY_WIDTH, RUNWAY_LENGTH, 0.0);
   mgAppend (prec, vrec);

   vrec = mgNewRec (fltVertex);
   mgSetCoord3d (vrec, fltCoord3d, 0.0, RUNWAY_LENGTH, 0.0);
   mgAppend (prec, vrec);

/*********************************************************************
   make light point appearance records for our lights
*********************************************************************/

   lightPointAppearance = mgNewLightPointAppearance (db, "Bi-Directional",
                              &lpaIndex[0]);
   mgSetAttList (lightPointAppearance,
                  fltLpDirectionalityType, 2,      /* Bi-directional */
                  fltLpBackColor, redColor,
                  fltLpBackColorIntensity, redIntensity,
                  fltLpHorizLobeAngle, 60.0,
                  fltLpVertLobeAngle, 30.0,
                  MG_NULL);
```

```
   lightPointAppearance = mgNewLightPointAppearance (db, "Uni-Directional",
                              &lpaIndex[1]);
   mgSetAttList (lightPointAppearance,
                  fltLpDirectionalityType, 1,      /* Uni-directional */
                  fltLpBackColor, redColor,
                  fltLpBackColorIntensity, redIntensity,
                  fltLpHorizLobeAngle, 45.0,
                  fltLpVertLobeAngle, 45.0,
                  MG_NULL);

/*********************************************************************
   make light points for left hand side or runway
*********************************************************************/

   /* make simple light point node composed of NUM_LIGHTS vertices */

   lightPoint = mgNewRec (fltLightPoint);
   mgSetAttList (lightPoint,
                  fltLpAppearanceIndex, lpaIndex[0],     /* Bi-directional */
                  MG_NULL);
   mgAppend (orec, lightPoint);

   lightSpacing = RUNWAY_LENGTH / (double) NUM_LIGHTS;
   for (i = 0; i < NUM_LIGHTS; i++) {
      vrec = mgNewRec (fltVertex);
      mgSetAttList (vrec,
                  fltVColor, yellowColor,
                  fltVIntensity, yellowIntensity,
                  MG_NULL);
      mgSetCoord3d (vrec, fltCoord3d,
                  RUNWAY_WIDTH/10.0,
                  lightSpacing*(double)i,
                  0.0);

      /* vertex normal important for bi-directional light points */
      mgSetVtxNormal (vrec, 0.0, -1.0, 0.0);
      mgAppend (lightPoint, vrec);
   }

/*********************************************************************
   make light point "string" for right hand side or runway
*********************************************************************/

   /* make light point node composed of 1 vertex, replicated
      NUM_LIGHTS - 1 times (vertex is drawn and then replicated
      fltRepCnt number of times - that's why NUM_LIGHTS - 1) */

   lightPoint = mgNewRec (fltLightPoint);
   mgSetAttList (lightPoint,
                  fltRepCnt, NUM_LIGHTS-1,
                  fltLpAppearanceIndex, lpaIndex[0],     /* Bi-directional */
                  MG_NULL);
   mgAppend (orec, lightPoint);

   /* for each replicated vertex in light point node, apply this
      translation matrix to get "light string" effect - other
      transformation linked list elements will apply also */
```

```
   xrec = mgNewRec (fltXmTranslate);
   mgSetCoord3d (xrec, fltXmTranslateFrom, 0.0, 0.0, 0.0);
   mgSetCoord3d (xrec, fltXmTranslateDelta, 0.0, lightSpacing, 0.0);
   mgAppend (lightPoint, xrec);

   /* add the single vertex that is replicated/transformed */
   vrec = mgNewRec (fltVertex);
   mgSetAttList (vrec,
               fltVColor, yellowColor,
               fltVIntensity, yellowIntensity,
               MG_NULL);
   mgSetCoord3d (vrec, fltCoord3d,
               RUNWAY_WIDTH - (RUNWAY_WIDTH/10.0),
               0.0, 0.0);
   /* vertex normal important for bi-directional light points */
   mgSetVtxNormal (vrec, 0.0, -1.0, 0.0);
   mgAttach (lightPoint, vrec);


/*********************************************************************
   make light point "string" for center of runway
*********************************************************************/

   /* make light point node composed of 1 vertex, replicated
      NUM_LIGHTS - 1 times down the right side of the runway
      (vertex is drawn and then replicated fltRepCnt number of
      times - that's why NUM_LIGHTS - 1) */
   lightPoint = mgNewRec (fltLightPoint);
   mgSetAttList (lightPoint,
                  fltRepCnt, (NUM_LIGHTS-1)/2,
                  fltLpAppearanceIndex, lpaIndex[1],     /* Uni-directional */
                  MG_NULL);
   mgAppend (orec, lightPoint);

   /* for each replicated vertex in light point node, apply this
      translation matrix to get "light string" effect - other
      transformation linked list elements will apply also */
   xrec = mgNewRec (fltXmTranslate);
   mgSetCoord3d (xrec, fltXmTranslateFrom, 0.0, 0.0, 0.0);
   mgSetCoord3d (xrec, fltXmTranslateDelta, 0.0, lightSpacing*2.0, 0.0);
   mgAppend (lightPoint, xrec);

   /* add the single vertex that is replicated/transformed */
   vrec = mgNewRec (fltVertex);
   mgSetAttList (vrec,
               fltVColor, yellowColor,
               fltVIntensity, yellowIntensity,
               MG_NULL);
   mgSetCoord3d (vrec, fltCoord3d, RUNWAY_WIDTH/2.0, 0.0, 0.0);

   /* vertex normal important for uni-directional light points */
   mgSetVtxNormal (vrec, 0.0, -1.0, 0.0);
   mgAttach (lightPoint, vrec);


   /* write and close the database */
```

```
   mgWriteDb (db);
   mgCloseDb (db);

   /* always call mgExit() after all OpenFlight API calls */
   mgExit ();
}
```

fltObject Nodes

An object node is a collection of polygons. It is good practice to keep objects simple and to use convex polygons.

Some **fltObject** attributes affect the object's descendants:

- **fltObjTransparency** is inherited by **fltPolygon** nodes and their nested children.
- **fltObjFlagNoIllum** prevents the object's vertex normals or color from being saved.
- **fltObjFlagNoShade** forces the object to use polygon normals instead of vertex normals.

Related Record Codes:
fltObject

fltGroup Nodes

Collecting sets of logically connected objects intogroups makes them easier to manipulate in the database. You can group any type of node except **fltHeader** and **fltVertex**. While legal, it is uncommon to group **fltPolygon** nodes; they are usually children of a **fltObject** node instead.

fltGroup nodes can have a bounding volume attached. OpenFlight supports a variety of bounding volume types, including box, sphere, and cylinder. Each **fltGroup** node can have its own bounding volume.

The API automatically recalculates bounding volumes each time **mgWriteDb** is called, unless the **fltGrpFlagFreezeBox** flag is set. If this flag is set, you can specify your own bounding volume by setting this flag and setting the bounding volume parameters.

Related Record Codes:
fltGroup
fltBoundingBox

fltBoundingSphere
fltBoundingCylinder
fltBoundingOrientation
fltBoundingCenter
fltBox

fltBsp Nodes

A **fltBsp** node identifies the top of a binary separating plane tree.

Binary separation is a modeling device that allows rendering (drawing) software to sort a database's hierarchy to determine the relative visual priority of items in the database. When the database is displayed, the portion that lies on the same side of each separating plane as the eyepoint has visual priority over the portion that lies on the opposite side.

A database can only use binary separating planes if it is *separable*. A database is separable if (and only if):

- A plane can be inserted between each pair of nodes selected for separation, and this plane does not intersect the geometry attached to either node.
- None of the convex hulls around the items being separated are interpenetrating. A convex hull is the smallest convex region that completely encloses all the vertices of each item selected for separation. Testing convex hulls is the polygonal equivalent of "shrink wrapping" each item and making sure none of the shrink wraps intersect one another.

Once the above conditions are met and the database is separated, it can be rendered quickly and correctly from all viewing angles. In contrast, a fixed listed database can be drawn quickly but there may not be a hierarchy that guarantees that the database draws correctly, while a z-buffered database displays correctly, but requires intensive computation and/or expensive graphics hardware. The performance gains in rendering make the extra effort required to separate a database worthwhile.

A *binary separating plane tree* consists of a left subtree, a polygon representing the binary separating plane, and a right subtree, all attached to a **fltBsp** node. The *binary separating plane* is the polygon that separates the left and right subtree of a **fltBsp** node. The **fltBsp** node is simply an attach point for these three items.

The right subtree (the last child) of a **fltBsp** node is sometimes referred to as the *true* side of the binary separating plane and is the side from which the plane is front-facing. In this context, the terms true and false are synonyms for frontfacing and backfacing. To draw correctly, each binary separating plane must face its right (last) sibling in the hierarchy.

The attachment of a binary separating plane to a BSP node distinguishes it from other polygons in the database. This polygon is for visualizing the separating plane and usually is not drawn in a simulation. In all other respects, it is identical to a normal polygon.

Related Record Codes:
fltBsp
fltDPlane

fltLod Nodes

Levels of detail are sets of models that represent the same item in the database at different levels of complexity. Levels of detail play an important part in building realistic databases that are quickly drawn by the real-time program. Each model is displayed within a specified range of distances from the eyepoint. When the eyepoint is far away from the item, a simple (low-resolution) level of detail is displayed. As the eyepoint approaches the item, lower resolution models *switch out* and increasingly complex (higher resolution) models *switch in*.

fltLod nodes function as groups in the hierarchy and can be attached to any group-like node. The **fltLodSwitchIn** and **fltLodSwitchOut** attributes of this node type determine the range in which the level of detail is displayed. The level of detail is displayed when the distance from the eyepoint to the level of detail falls within this range. **fltLodSwitchIn** must be greater than **fltLodSwitchOut** for the level of detail to be displayed.

The center of a level of detail is represented by the value of the **fltLodCenterPoint** attribute. The real-time software uses this value to calculate the distance from the eyepoint to the node (sometimes referred to as the *slant range*); it is not necessarily the geometric center of the model. The default value of the center is 0,0,0. When the database is written to disk, this value is recalculated and is set to the geometric center of the level of detail. If you set the center point manually, you should disable the automatic recalculation by setting the **fltLodFlagFreezeCenter** flag.

Level-of-Detail Hierarchy

As mentioned above, a level of detail is selected for display when the distance from the eyepoint to the center of the level of detail falls within the range of its switch in and switch out attributes. However, if a level of detail is switched out (not displayed), none of its descendants can be switched in. This saves drawing time because the real-time software does not process anything attached to a level-of-detail node that is switched out.

Setting up a hierarchical level of detail tree by nesting **`fltLod`** nodes can allow more efficient display in real time. The most common nesting scheme for level of detail nodes is a binary tree. Because a level of detail cannot switch in before its parent, the **`fltLodSwitchIn`** values of a **`fltLod`** node child can be any value greater than or equal to the **`fltLodSwitchIn`** value of its parent. Likewise, the **`fltLodSwitchOut`** values of a **`fltLod`** node child can be any value greater than or equal to the **`fltLodSwitchOut`** value of its parent.

Because each level of detail is selected for display on the basis of its own **`fltLodSwitchIn`** and **`fltLodSwitchOut`** attributes, the display of various levels of detail does not need to be mutually exclusive. Additive level-of-detail nodes can be used to accumulate the pieces of a visual scene. For example, a radio tower can be modeled using two level-of-detail nodes: a tower body that is always displayed, and an antenna that switches in or out depending on the distance from the eyepoint to the tower.

The API provides four convenience functions to perform LOD switching: **`mgMoreDetail`**, **`mgLessDetail`**, **`mgMostDetail`**, and **`mgLeastDetail`**. These functions turn LOD nodes on and off according to their **`fltLodSwitchIn`** and **`fltLodSwitchOut`** attributes.

Related Functions:

`mgMoreDetail`
`mgLessDetail`
`mgMostDetail`
`mgLeastDetail`

Related Record Codes:

`fltLod`

fltCurve Nodes

A curve node can represent different kinds of geometric curves or curve segments using control points. Its vertex children are the control points of the curve. The curve type can be set using the **`fltCurveType`** attribute.

Related Record Codes:
`fltCurve`

fltDof Nodes

Degrees of freedom allow you to place points of articulation in the database. Degree-of-freedom attributes define a range of translation, rotation, and/or scale for the geometry attached to the **`fltDof`** node. For example, a degree of freedom can specify the angle through which a door swings or the range over which a window slides.

The range of motion in a degree of freedom is specified relative to a local coordinate system and the transformations it contains are applied to all the descendants of the degree of freedom node. This ensures that the effects of multiple degrees of freedom accumulate naturally in the database hierarchy. Consider the example of a hand attached to an arm. The wrist of the hand bends and rotates relative to the position of the arm because the degree-of-freedom node that specifies the range of motion for the wrist is a descendant of the degree-of-freedom node that specifies the range of motion for the arm.

A **`fltDof`** node is the only node type with built-in transformation information. The built-in "current" translation, rotation, and scale attributes are used to compute the node's overall transformation matrix. You can query this matrix using **`mgGetMatrix`** just as you can for any other node type. However, you cannot add or delete transformations from a **`fltDof`** node's transformation list. As with any node, you cannot set the overall matrix directly.

Degree of Freedom Axes

All movement in a degree of freedom is relative to the axes that define its origin. Initially, degree of freedom axes are placed at the transformed local origin of the **`fltDof`** node, meaning that if the degree of freedom is a descendant of a node containing a transformation matrix, the **`fltDof`** axes

are also transformed. To position and orient a set of **fltDof** axes yourself, set the **fltDofPut<field>** attributes.

Related Functions:
mgGetMatrix

Related Record Codes:
fltDof

fltSound Nodes

Sound nodes function as group nodes in the database hierarchy. They are attached to groups, levels of detail, switch nodes, degrees of freedom, and so forth, and can have any type node as a child.

Sound nodes are hierarchical references to files listed in the *Sound palette*. A sound node's position is the coordinate location from which the sound is emitted. When a sound node is created, it is automatically positioned at the database origin until you set the **fltSndOffset** attribute. Creator supports *aiff* sound files; to play other formats, you must install a third-party sound player.

The position of a sound node is the coordinate location from which the sound is emitted. When a sound node is created, it is automatically positioned at the database origin.

Related Record Codes:
fltSound
fltSoundPalette

fltLightSource Nodes

A Light Source node is a hierarchical node in the database that emits light but is itself invisible. Light source nodes are hierarchical instances of entries in the Light Source palette, with a few extra attributes. The position and direction stored in the light source node attributes override those stored in the palette.

A light source node can be flagged as local or global. Local light sources only shine on their descendants in the hierarchy; global light sources shine on all geometry in the database. You can add global lights to a database without inserting them into the hierarchy: simply flag them as modeling lights in the

light source palette by setting the **fltLspModeling** flag in the palette entry record. Modeling lights are useful when you only want static, global lights in the database. You might want to include lights in the hierarchy, however, to move them or turn them on and off dynamically-- for example, with LOD nodes or animation.

Note: The number of concurrent light sources that can be rendered at any one time is platform dependent.

Related Functions:

mgGetLightSource
mgIndexOfLightSource
mgNameOfLightSource
mgGetLightSourceCount
mgGetFirstLightSource
mgGetNextLightSource
mgNewLightSource
mgWriteLightSourceFile

Related Record Codes:

fltLightSource
fltLightSourcePalette

fltSwitch Nodes

A switch node is a group-like node that contains masks that control the display of its children. *Switching masks* are attributes of each switch node. When a switch node is created, it holds a single mask (Mask 0) containing one bit for each child of the switch bead, with the first bit representing the first child in the hierarchy. The value of the *n*th bit within a mask signals that a child should be turned on or off. Use **mgGetSwitchBit** and **mgSetSwitchBit** to get or set a bit. Each switch mask is identified by its index and an optional name.

Use **mgGetSwitchMaskCount** to find out how many masks a switch node has, and **mgGetSwitchMaskNo** to retrieve any mask in the list. Use **mgAddSwitchMask** and **mgDeleteSwitchMask** to add and remove masks.

The attribute **fltSwCurMask** specifies the index of the current mask on the switch node. If you set this attribute, the API turns on and off the switch children according to the bits of the mask index you specify.

Use **mgInitSwitchMask** to set all the bits of a mask to a given value (in other words, to turn all the children on or off for a given mask).

Related Functions:

mgAddSwitchMask
mgDeleteSwitchMask
mgGetSwitchBit
mgGetSwitchMaskCount
mgGetSwitchMaskNo
mgGetSwitchMaskName
mgInitSwitchMask
mgSetSwitchBit
mgSetSwitchMaskName

Related Record Codes:

fltSwitch

fltText Nodes

A text node draws a string of data using a specified font. Use **mgGetTextString** and **mgSetTextString** to retrieve and set the actual string for the text. Various visual characteristics of the text and formatting information can be specified by the attributes in the record.

Related Functions:

mgGetTextString
mgSetTextString

Related Record Codes:

fltText

fltRoad Nodes

A road node is the primary record of a road segment. It stores the attributes used to create and modify a road segment. The children of the road node represent the geometry and paths of the road and should not be manually edited. Since the attributes of a road node are closely linked to the geometry below the road node, modifying the attributes of a road node will invalidate the road segment. For this reason, the attributes of a road node should be treated as 'read only'.

Related Record Codes:

fltRoad

fltPath

fltPath Nodes

A road path node is a child of a road node. It describes a lane of the parent road node. The child of a road path bead is a polygon node whose vertices provide the coordinates of the center of the lane. By convention, the first path child of a road node is the center lane. Since the road tools create path nodes automatically, any modifications you make to these nodes may be overridden by the road tools. For this reason, the contents and of a path node and the nodes below should be treated as 'read only'.

Related Record Codes:
fltRoad
fltPolygon

Instances

Almost any node type can be instanced, so there is no attribute code to identify a local instance. To go back and forth between instances and their references, use the following:
mgGetFirstInstance
mgGetNextInstance
mgGetReference

To create an instance to a node, use **mgReference**; to delete an instance, use **mgDeReference**. If you remove all references to a node, make sure you delete the node to avoid leaving orphan nodes in the database.

For more information about instances, see The OpenFlight Hierarchy.

fltXref Nodes

External reference nodes cannot have any children in the local tree. They simply reference an external file to include, and have attributes that specify whether to use the palettes from the local or the external database. The attribute **fltXrefFilename** defines the name of the OpenFlight database referenced.

Normally the attribute **fltXrefFilename**, specifies a simple OpenFlight file name as in:

`filename.flt`

In this way, the entire database contained in **`filename.flt`** is referenced.

Alternatively, you can reference a single node in an OpenFlight file using the following syntax for the **`fltXrefFilename`** attribute:

`filename.flt<node_name>`

In this way, the node whose name is **`node_name`** found in **`filename.flt`** is referenced.

For more information about external references, see The OpenFlight Hierarchy.

Related Record Codes:
`fltXref`

6

Palette Records

OpenFlight defines several types of palettes for colors, materials, textures, etc. The individual items contained in a palette are uniquely identified by index. Each database contains at most one of each palette type. Palettes contained in a database are implicitly shared by all elements in that database.

Certain nodes of a database have attributes that reference items in palettes. Nodes of type **`fltPolygon`** for example, have the attribute **`fltPolyTexture`** which represents the index of the texture pattern that is applied to the polygon. The value of such an attribute specifies the index of the item in the palette that is being referenced. In this way, changing the definition of an item in a palette has the effect of changing all nodes in the database that have attributes that reference that palette item. By convention, if such an attribute has a value of -1, the attribute is unassigned. In the case of the attribute, **`fltPolyTexture`**, a value of -1 would indicate that the associated polygon has no texture applied.

External reference nodes have palette override attributes. These are flags that specify whether the external file should use its own palettes or those of the database that references it.

Color Palettes

Each database has a color palette containing colors that can be applied to polygons, meshes, light points and vertices. This palette contains 1024 color entries that can be edited independently. Each editable color entry is defined by

red, green, and blue values ranging from 0 to 255. Every entry is further divided into a band of 128 shades, or *intensities*, ranging from 0 to 127.

Node Attributes

Every **fltPolygon**, **fltMesh** and **fltVertex** record has color attributes that determine its graphical appearance. The node attributes that control color include:

fltPolyPrimeColor
fltPolyPrimeIntensity
fltPolyAltColor
fltPolyAltIntensity
fltVColor
fltVIntensity

Note that **fltPolygon** and **fltMesh** share the exact same set of attributes. Since color can be assigned to a database at the **fltVertex, fltPolygon**, and **fltMesh** levels equally, OpenFlight provides the **fltGcLightMode** attribute to resolve any conflicts. Vertex color is used if the **fltGcLightMode** is set to 1 (Gouraud) or 3 (Dynamic Gouraud) for either polygons or meshes. The color of the polygon or mesh is used if **fltGcLightMode** is set to 0 (None) on the polygon or mesh.

Color Palette Index

A color palette index is a number that uniquely identifies each *full-intensity* color in the database. To uniquely define any color, you must assign both an index and an intensity. Valid indices range from 0 to 1023; valid intensities range from 0 to 127. To assign RGB values to an index, use **mgSetColorIndex**. To convert between a color palette index and its RGB values, use **mgIndex2RGB** and **mgRGB2Index**.

To load the default color palette, call **mgReadDefaultColorPalette**. Call **mgWriteDefaultColorPalette** to save a database's color palette as the default color palette.

Default Color Palette

New databases inherit the default color palette stored in the file **flt1.color.** The API loads this file from the directory path specified in the

environment variable **PRESAGIS_CREATOR_RESOURCEPATH.** If this file is not found, the API creates a hue/lightness/saturation (HLS) color palette. This palette has 1024 primary colors, with ramps of 128 intensities of each.

Custom Color Palette

If you develop a custom color palette, you can save it in a file that can be loaded by other databases. This simplifies the use of similar color palettes in similar databases. For example, a group of terrain databases might all use a color palette with a large selection of earth tones.

To save a database's color palette to a file, call **mgWriteColorPalette**; to read a color palette from a file, call **mgReadColorPalette**.

Color Names

Symbolic names can be assigned to colors to make colors easier to manage. Only full-intensity colors can be named; individual shades of intensity cannot be named.

All color names in the palette must be unique, but an entry in the color palette can have more than one name. To add or delete a name from an index, call **mgNewColorName** or **mgDeleteColorName**.

You can select a *current color name*, which will be assigned to nodes whenever the corresponding color index is assigned. Use **mgSetCurrentColorName**.

To access and traverse the name list, use **mgGetCurrentColorName** and **mgGetNextColorName**. To find a color's palette index from its name, use **mgGetColorIndexByName**.

Sample: egcolor1.c

```
/*****************************************************************************

  Sample file: EGCOLOR1.C

  Objective:Shows how to access color values from a color palette and
     from polygon and vertex node records.
     Shows how to convert color index values to red, green and blue values.
     Shows how to set color values in a color palette and polygon and
     vertex node records.

  Program functions: Read database given on command line.
```

```
     Prints the index, RGB values, and names of each
     color in the database color palette.
     Search the database and count how many polygons
     and vertices are using each color index.
     Increases the red component of all polygon colors.
     Increases the blue component of all vertex colors.
     Makes all palette colors max intensity.

  API functions used:
     mgInit(), mgExit(),
     mgGetAttList(), mgIndex2RGB(), mgGetNextColorName(),
     mgWalk(), mgIsPolygon(), mgIsVertex(),
     mgOpenDb(), mgCloseDb().

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#include <openflightapi/mgapiall.h>

int pcolorcount[1024];
int vcolorcount[1024];

static void PrintPolyColors (void)
{
   int i;

   printf ("\n\nNumber of Polygons with Each Color\n");
   for (i=0 ; i<1024 ; i++) {
      if (pcolorcount[i])
         printf ("\n%d: %d", i, pcolorcount[i]);
   }
   printf ("\n");
}

static void PrintVtxColors (void)
{
   int i;

   printf ("\n\nNumber of Vertices with Each Color\n");
   for (i=0 ; i<1024 ; i++) {
      if (vcolorcount[i])
         printf ("\n%d: %d", i, vcolorcount[i]);
   }
   printf ("\n");
}

static mgbool CountPolyColor (mgrec *db, mgrec *par, mgrec *rec, void *info)
{
   int status, pcolor;
   float pintens;

   if (mgIsCode (rec, fltPolygon)) { /* only count polygons */
      status = mgGetAttList (rec, fltPolyPrimeColor, &pcolor,
                             fltPolyPrimeIntensity, &pintens, MG_NULL);
      if (status)
```

```
         pcolorcount[pcolor]++;
      else
         printf ("Error trying to get color/intensity from polygon\n");
   }
   return (MG_TRUE);
}

static mgbool CountVtxColor (mgrec *db, mgrec *par, mgrec *rec, void *info)
{
   int status, vcolor;

   if (mgIsCode (rec, fltVertex)) { /* only count vertices */
      status = mgGetAttList (rec, fltVColor, &vcolor, MG_NULL);
      if (status)
         vcolorcount[vcolor]++;
      else
         printf ("Error trying to get color from vertex\n");
   }
   return (MG_TRUE);
}

void main (int argc, char* argv[])

/* This program must be given an OpenFlight database file
   as input. The database must contain at least one polygon */
{
   mgrec*         db;
   unsigned int   i;
   short          r, g, b;
   void           *namelistptr=MG_NULL;
   char           *name=MG_NULL;

   /* check for correct number of arguments */
   if (argc < 2) {
      printf ("Usage: %s file_name\n" argv[0]);
      exit (1);
   }

   /* always call mgInit before any other API calls */
   mgInit (&argc, argv);

   /* open database */
   if (!(db = mgOpenDb (argv[1]))) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n" msgbuf);
      exit (1);
   }

   /* print the index, RGB, and color names */
   /* for each entry in the color palette */
   printf ("\nColor Palette Values\n");
   for (i = 0; i < 1024; i ++)
   {
      mgIndex2RGB (db, i, 1.f, &r, &g, &b);
      printf ("\n%d: %d, %d, %d" i, r, g, b);
      while (name = mgGetNextColorName (db, i, &namelistptr))
```

```
            printf ("  %s  name);
         pcolorcount[i] = 0;
         vcolorcount[i] = 0;
         mgFree (name);
      }

      /* count how many polygons are using each palette color */
      mgWalk (db, CountPolyColor, MG_NULL, MG_NULL, MWALK_NORDONLY + MWALK_MASTER);
      PrintPolyColors ();

      /* count how many vertices are using each palette color */
      mgWalk (db, CountVtxColor, MG_NULL, MG_NULL, MWALK_NORDONLY
                                                 + MWALK_MASTER + MWALK_VERTEX);
      PrintVtxColors ();

      /* close the database */
      mgCloseDb (db);

      /* always call mgExit() after all OpenFlight API calls */
      mgExit ();
}
```

Material Palette

Materials provide special effects that simulate the light-reflecting characteristics of substances like wood, plastic, and metal. Material properties are defined in a material palette and can be selectively applied to polygons and meshes in the database using the **fltPolyMaterial** attribute code. Set the **fltGcLightMode** for the polygon or mesh to 2 (Dynamic) or 3 (Dynamic Gouraud) to use the **fltPolyMaterial** index of the polygon or mesh for rendering.

Every OpenFlight database has its own material palette containing the set of material definitions that can be applied to polygons and meshes. To traverse a database's material palette, use **mgGetFirstMaterial**, **mgGetNextMaterial**, and **mgGetMaterialCount**. To get a specific material in the palette when given the index, use **mgGetMaterial.**

New databases inherit the default material palette stored in the file **flt1.material**. The API loads this file from the directory path specified in the environment variable **PRESAGIS_CREATOR_RESOURCEPATH**. If this file is not found, the API will create a new one for the database containing common materials.

Like the color palette, elements in the material palette have names that can be used to identify them. Unlike the color palette, there is only one name per

material. A material's name can be used to uniquely identify a material index. The reverse is also true. Use **`mgIndexOfMaterial`** and **`mgNameOfMaterial`** to go back and forth between a material's name and its index.

There are two types of materials that can be in the palette; the *standard* material and the *extended* material. Both types have an optional name. The attribute **`fltMatType`** of a material entry determines which type of material the entry is. The components of each type are described in the sections that follow.

Standard Material

A *standard* material is composed of the properties described in this section. Ambient, Diffuse, Specular, and Emissive properties for *standard* materials are defined in terms of color. Shininess and Alpha properties for *standard* materials are scalar values. When a polygon or mesh references a standard material, the geometry color and transparency combine with that of the material as described in the OpenFlight specification.

`fltAmbient`	Defines the amount and color of light the material reflects from other objects in the scene. This affects surfaces that are not illuminated directly. Polygon/mesh color interacts with the ambient lighting effect.
`fltDiffuse`	Defines the amount and color of light reflected in all directions, regardless of the eyepoint position. This results in a flat reflection that is brightest where light strikes the surface at a 90 degree angle. Polygon/mesh color interacts with the diffuse lighting effect.
`fltSpecular`	Defines the brightness and color of the highlight, or glare, which is brightest when the eyepoint is located at the light's angle of reflection. Metal often imparts a reflected color to the spectral highlight. Specular light is directly affected by the shininess property.
`fltEmissive`	Defines the amount and color of light the material produces.

fltShininess	Defines the shininess of the material, which is indicated by the size of the highlight on the material ball. A high shininess value produces a small highlight, giving the material a metallic appearance.
fltMatAlpha	Defines the alpha component (transparency) of the material. A value of 0 defines a transparent material; a value of 1 defines an opaque material. Note: Pay careful attention to database traversal order when using material transparency. A transparent polygon/mesh blends with the polygon/mesh behind it in the order of traversal.

Use **mgGetMaterialElem** or **mgGetAttList** to get these values from a *standard* material entry. Use **mgSetAttList** to set properties for a *standard* material entry.

Extended Material

An *extended* material is composed of the *optional* properties described in this section. When a polygon or mesh references an *extended* material, the extended material completely describes the visual appearance of the geometry.

fltAmbientEx	Defines the amount and color of light the material reflects from other objects in the scene. This affects surfaces that are not illuminated directly.
fltDiffuseEx	Defines the amount and color of light reflected in all directions, regardless of the eyepoint position. This results in a flat reflection that is brightest where light strikes the surface at a 90 degree angle.

fltSpecularEx	Defines the brightness and color of the highlight, or glare, which is brightest when the eyepoint is located at the light's angle of reflection. Metal often imparts a reflected color to the spectral highlight. Specular light is directly affected by the shininess property.
fltEmissiveEx	Defines the amount and color of light the material produces.
fltMatAlphaEx	Defines the alpha component (transparency) of the material.
fltBumpMapEx	Defines the bump map of the material.
fltNormalMapEx	Defines the normal map of the material.
fltLightMapEx	Defines the light map of the material.
fltShadowMapEx	Defines the shadow map of the material.
fltReflectionMapEx	Defines the reflection map of the material.
fltPhysicalMaterialMapEx	Defines the physical material map of the material.

To create and save your own material palette, use **mgNewMaterial** and **mgWriteMaterialFile.** Note that with the introduction of the extended material in API version 3.4, the format of the material palette file saved by **mgWriteMaterialFile** is different from previous versions. If you need to write "old" versions of the material palette file, use **mgWriteMaterialFileVersion**. Similary, when saving a material palette file in Creator, you can choose to write the "old" or "new" format.

To read the contents of a material palette file into an existing database, use **mgReadMaterialFile**. Note that **mgReadMaterialFile** will read either the "old" or "new" material palette file format. To delete entries from the palette, use **mgDeleteMaterial** and **mgDeleteMaterialByName**.

Sample: egmaterial1.c

```
/*****************************************************************************

  Sample file: EGMATERIAL1.C

  Objective: Show how to access entries in the material palette.

  Program functions: Open a database file with a material palette.
     Get the first material in the palette.  Change some of
     the first material's properties. Print properties of all
     the materials in the palette.
     Write the material palette as a file.

  API functions used:
     mgInit(), mgExit(), mgGetLastError(),
     mgNameOfMaterial(), mgGetMaterialCount(), mgGetAttList(),
     mgGetFirstMaterial(), mgGetNextMaterial(), mgFree(),
     mgGetNormColor(), mgSetNormColor(),
     mgOpenDb(), mgCloseDb()

 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include all OpenFlight API headers */
#include <mgapiall.h>

#define MATERIALFILE"materials.mat"

static void PrintMaterialTexture (mgrec* mat, mgcode recCode, const char* label)
{
   int texture, layer;
   mgrec* nestedRec = mgGetAttRec (mat, recCode, MG_NULL);
   if (mgGetAttList (nestedRec, fltMatTexture, &texture, MG_NULL) == 1)
      printf ("%s Material Texture Index: %d\n", label, texture);
   else
      printf ("Error getting %s material texture index\n", label);

   if (mgGetAttList (nestedRec, fltMatLayer, &layer, MG_NULL) == 1)
      printf ("%s Material Texture Layer: %d\n", label, layer);
   else
      printf ("Error getting %s material texture layer\n", label);
}

static mgbool MaterialHasComponent (mgrec* mat, mgcode recCode)
{
   return mgHasAtt (mat, recCode);
}

static void PrintNormColor (mgrec* mat, mgcode recCode, const char* label)
{
   float r, g, b;
   if (mgGetNormColor (mat, recCode, &r, &g, &b) == MG_TRUE)
      printf ("%s: %5.3f %5.3f %5.3f \n", label, r, g, b);
```

```
   else
      printf ("Error getting %s color attributes\n", label);
}

#define PRINT_MATERIALTEXTURE(_mat,_code) PrintMaterialTexture(_mat,_code,#_code)
#define PRINT_NCOLOR(_mat,_code) PrintNormColor(_mat,_code,#_code)

static void PrintExtendedMaterial (mgrec* db, mgrec* mat, int index)
{
   if (MaterialHasComponent (mat, fltAmbientEx)) {
      printf ("fltAmbientEx\n");
      PRINT_NCOLOR (mat, fltAmbientExColor);
      PRINT_MATERIALTEXTURE (mat, fltAmbientExLayer1);
      PRINT_MATERIALTEXTURE (mat, fltAmbientExLayer2);
      PRINT_MATERIALTEXTURE (mat, fltAmbientExLayer3);
      PRINT_MATERIALTEXTURE (mat, fltAmbientExLayer4);
   }
   else {
      printf ("No fltAmbientEx Component\n");
   }

   if (MaterialHasComponent (mat, fltDiffuseEx)) {
      printf ("fltDiffuseEx\n");
      PRINT_NCOLOR (mat, fltDiffuseExColor);
      PRINT_MATERIALTEXTURE (mat, fltDiffuseExLayer1);
      PRINT_MATERIALTEXTURE (mat, fltDiffuseExLayer2);
      PRINT_MATERIALTEXTURE (mat, fltDiffuseExLayer3);
      PRINT_MATERIALTEXTURE (mat, fltDiffuseExLayer4);
   }
   else {
      printf ("No fltDiffuseEx Component\n");
   }

   if (MaterialHasComponent (mat, fltSpecularEx)) {
      float matshin;
      printf ("fltSpecularEx\n");

      if (mgGetAttList (mat, fltSpecularExShininess, &matshin, MG_NULL) == 1)
         printf ("Shininess: %5.3f\n", matshin);
      else
         printf ("Error getting extended specular shininess attributes\n");

      PRINT_NCOLOR (mat, fltSpecularExColor);
      PRINT_MATERIALTEXTURE (mat, fltSpecularExLayer1);
      PRINT_MATERIALTEXTURE (mat, fltSpecularExLayer2);
      PRINT_MATERIALTEXTURE (mat, fltSpecularExLayer3);
      PRINT_MATERIALTEXTURE (mat, fltSpecularExLayer4);
   }
   else {
      printf ("No fltSpecularEx Component\n");
   }

   if (MaterialHasComponent (mat, fltEmissiveEx)) {
      printf ("fltEmissiveEx\n");
      PRINT_NCOLOR (mat, fltEmissiveExColor);
      PRINT_MATERIALTEXTURE (mat, fltEmissiveExLayer1);
      PRINT_MATERIALTEXTURE (mat, fltEmissiveExLayer2);
```

```
   PRINT_MATERIALTEXTURE (mat, fltEmissiveExLayer3);
   PRINT_MATERIALTEXTURE (mat, fltEmissiveExLayer4);
}
else {
   printf ("No fltDiffuseEx Component\n");
}

if (MaterialHasComponent (mat, fltAlphaEx)) {
   int quality;
   float alpha;
   printf ("fltAlphaEx\n");

   if (mgGetAttList (mat, fltAlphaExQuality, &quality,
                     fltAlphaExAlpha, &alpha, MG_NULL) == 2)
      printf ("Quality: %d, Alpha: %5.3f\n", quality, alpha);
   else
      printf ("Error getting extended alpha quality/alpha attributes\n");

   PRINT_MATERIALTEXTURE (mat, fltAlphaExLayer1);
   PRINT_MATERIALTEXTURE (mat, fltAlphaExLayer2);
   PRINT_MATERIALTEXTURE (mat, fltAlphaExLayer3);
   PRINT_MATERIALTEXTURE (mat, fltAlphaExLayer4);
}
else {
   printf ("No fltAlphaEx Component\n");
}

if (MaterialHasComponent (mat, fltLightMapEx)) {
   float intensity;
   printf ("fltLightMapEx\n");

   if (mgGetAttList (mat, fltLightMapExMaxIntensity, &intensity, MG_NULL) == 1)
      printf ("Intensity: %5.3f\n", intensity);
   else
      printf ("Error getting extended light map intensity attribute\n");

   PRINT_MATERIALTEXTURE (mat, fltLightMapExTexture);
}
else {
   printf ("No fltLightMapEx Component\n");
}

if (MaterialHasComponent (mat, fltNormalMapEx)) {
   printf ("fltNormalMapEx\n");
   PRINT_MATERIALTEXTURE (mat, fltNormalMapExTexture);
}
else {
   printf ("No fltNormalMapEx Component\n");
}

if (MaterialHasComponent (mat, fltBumpMapEx)) {
   int tangentLayer, binormalLayer;
   printf ("fltBumpMapEx\n");

   if (mgGetAttList (mat, fltBumpMapExTangentLayer, &tangentLayer,
               fltBumpMapExBinormalLayer, &binormalLayer, MG_NULL) == 2)
```

```
         printf ("Tangent Layer: %d, Binormal Layer: %d\n", tangentLayer,
binormalLayer);
      else
        printf ("Error getting extended bump map tangent/binormal layer attributes\n");

      PRINT_MATERIALTEXTURE (mat, fltBumpMapExTexture);
   }
   else {
      printf ("No fltBumpMapEx Component\n");
   }

   if (MaterialHasComponent (mat, fltShadowMapEx)) {
      float intensity;

      printf ("fltShadowMapEx\n");

      if (mgGetAttList (mat, fltBumpMapExTangentLayer, &intensity, MG_NULL) == 1)
         printf ("Intensity: %5.3f\n", intensity);
      else
         printf ("Error getting extended shadow map intensity attribute\n");

      PRINT_MATERIALTEXTURE (mat, fltShadowMapExTexture);
   }
   else {
      printf ("No fltShadowMapEx Component\n");
   }

   if (MaterialHasComponent (mat, fltReflectionMapEx)) {
      printf ("fltReflectionMapEx\n");
      PRINT_NCOLOR (mat, fltReflectionMapExColor);
      PRINT_MATERIALTEXTURE (mat, fltReflectionMapExReflectionTexture);
      PRINT_MATERIALTEXTURE (mat, fltReflectionMapExEnvironmentTexture);
   }
   else {
      printf ("No fltReflectionMapEx Component\n");
   }
}

static void PrintMaterial (mgrec* db, mgrec* mat, int index)
/* prints the attributes of a given material */
{
   float matshin, matalpha;
   char* matname;
   int numAttr;

   matname = mgNameOfMaterial (db, index);
   printf ("Material: Index=%d Name=%s\n", index, matname ? matname : "None");
   PRINT_NCOLOR (mat, fltAmbient);
   PRINT_NCOLOR (mat, fltDiffuse);
   PRINT_NCOLOR (mat, fltSpecular);
   PRINT_NCOLOR (mat, fltEmissive);

   numAttr = mgGetAttList (mat, fltShininess, &matshin,
                                fltMatAlpha, &matalpha, MG_NULL);
   if (numAttr == 2)
      printf ("Shininess: %5.3f Alpha: %5.3f\n", matshin, matalpha);
   else
```

```
      printf ("Error getting shininess/alpha attributes\n");

   /* print any extended material components present */
   PrintExtendedMaterial (db, mat, index);

   mgFree (matname);  /* mgNameOfMaterial allocs, must dealloc */
}

static mgbool PrintMaterialPalette (mgrec* db)
/* print properties of all the database's materials */
{
   mgrec* mat;
   int index;

   mat = mgGetFirstMaterial (db, &index);
   if (mat != MG_NULL) {
      printf ("\nMaterial Palette\n");
      PrintMaterial (db, mat, index);
      while ((mat = mgGetNextMaterial (mat, &index)) != MG_NULL) {
         printf ("\n");
         PrintMaterial (db, mat, index);
      }
      return MG_TRUE;
   }
   return MG_FALSE;
}

int main (int argc, char* argv[])
{
   mgrec* db;
   mgrec* firstmat;
   int index, matCount;
   mgbool ok;
   char materialPaletteFile[1024];

   if (argc < 2) {
      printf ("\nUsage: %s <input_db_filename>\n", argv[0]);
      printf ("   Reads database: <input_db_filename>\n");
      printf ("   Prints the number of materials\n");
      printf ("   Prints the first material before and after making a change to it\n");
      printf ("   Prints all the materials\n");
      printf ("   Writes the material palette to a file: %s\n", MATERIALFILE);
      printf ("\n");
      exit (EXIT_FAILURE);
   }

   /* initialize the OpenFlight API
      always call mgInit BEFORE any other OpenFlight API calls
      */
   mgInit (&argc, argv);

   /* open database */
   printf ("\nOpening database: %s\n", argv[1]);
   db = mgOpenDb (argv[1]);
   if (db == MG_NULL) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
```

```
      printf ("%s\n", msgbuf);
      mgExit ();
      exit (EXIT_FAILURE);
   }

   /* check for light sources in this database */
   matCount = mgGetMaterialCount (db);
   if (matCount == 0) {
      printf ("No materials in database.\n");
      mgExit ();
      exit (EXIT_FAILURE);
   }

   printf ("\n");
   printf ("Total number of materials in database: %d\n\n", matCount);

   /* get the first material */
   firstmat = mgGetFirstMaterial (db, &index);
   if (firstmat == MG_NULL) {
      printf ("\nError getting first first material.\n");
      mgExit ();
      exit (EXIT_FAILURE);
   }

   /* get and print the attributes of the first material */
   printf ("First Material, before modifying:\n");
   PrintMaterial (db, firstmat, index);

   /* now change the specular color of the first material */
   ok = mgSetNormColor (firstmat, fltSpecular, 0.5f, 0.5f, 0.5f);
   printf ("Setting specular for first material: %s\n", ok==MG_TRUE ? "Ok" : "Failed");

   /* now print attributes of first material, notice that the */
   /* ambient value has changed */
   printf ("\nFirst Material, after modifying Specular component:\n");
   PrintMaterial (db, firstmat, index);

   /* now print the attributes of all the materials */
   PrintMaterialPalette (db);

   strcpy (materialPaletteFile, MATERIALFILE);
   ok = mgWriteMaterialFile (db, materialPaletteFile);
   if (ok == MG_TRUE)
      printf ("Material Palette saved to: %s\n", materialPaletteFile);
   else
      printf ("Error writing Material Palette file: %s\n", materialPaletteFile);

   /* close the database */
   ok = mgCloseDb (db);
   if (ok == MG_FALSE)
      printf ("Error closing database\n");

   /* always call mgExit() AFTER all OpenFlight API calls */
   mgExit ();

   exit (0);
}
```

Light Source Palette

A light source is a light that illuminates all or part of a database, producing the effect of shading. OpenFlight supports the following types of light sources:

- An Infinite light source shines along a direction vector. It is called an infinite light because it can be thought of as an infinitely wide wall of light, infinitely far away. Its attributes do not change according to distance; when an infinite light source illuminates a portion of a database, the same vector is applied to every affected vertex. Infinite light sources require the least computation to display. This type of light is sometimes called a directional light, since it has a direction attribute, but no position.
- A Local light source is located at a point in space and shines in all directions. When a local light source illuminates a portion of a database, a distinct vector is calculated from the light source position to each affected vertex. Local light sources produce a more natural lighting effect than infinite light sources, especially when you are lighting objects that have sharp angles between faces. However, they place a greater computational burden on the computer. This type of light is sometimes called a point light, since it originates from a single point.
- A Spotlight source is a local light with a cone of effect. Within the sharp cutoff of the cone is a soft drop-of area of lesser intensity. Spotlights are especially useful for modeling moving sources of light, such as automobile headlights.

To traverse a database's light source palette, use **`mgGetFirstLightSource`**, **`mgGetNextLightSource`**, and **`mgGetLightSourceCount`**.

Enabling a Light Source

A light source is turned on in one of two ways:

1. By directly enabling the light source in the light source palette. Such light sources are referred to as modeling lights. Modeling lights have a fixed position and/or direction and illuminate everything in the database.
2. By adding a light source node to the database hierarchy. Light source nodes allow you to vary the position and/or direction of a light source, and restrict the effect of the light source to specific portions of the hierarchy.

Just as a single texture pattern can be applied with many different mappings in a database, a single light source definition can be enabled as a modeling light and assigned to one or more light source nodes. The number of concurrent lights that can be enabled is platform-dependent. However, OpenFlight makes no restrictions on how many you can have in the database hierarchy or in the palette.

Light Source Attributes

Every OpenFlight database has its own light source palette containing the set of light sources that can be turned on.

To get (or set) the attributes of a light source palette entry, use **`mgGetLightSource`** to get the light source record from the palette; then use **`mgGetAttList`** (or **`mgSetAttList`**) on the individual attributes.

An entry in the light source palette can optionally have a name. Use **`mgIndexOfLightSource`** and **`mgNameOfLightSource`** to translate between name and index.

In addition to the name and index, a light source palette entry has the following attributes:

`fltLtspType`	Specifies whether the light is an infinite, local, or spot light.
`fltLtspAmbient`	Describes the intensity of the ambient light added to the database by this light source.
`fltLtspDiffuse`	Describes the color of the diffuse light a light source adds to the database. This attribute is, essentially, the color of the light.
`fltLtspSpecular`	Describes the color of the specular highlight produced on objects in the database by this light source.
`fltLtspSpotExp`	Controls the soft dropoff that occurs within the cone of effect defined for a spotlight. Increasing this value makes the light source more focused.

fltLtspSpotSpread	Controls the angle between the axis of the cone defining a spotlight and the sharp cutoff that occurs along the edge of the cone. This angle is defined in degrees.
fltLtspYaw **fltLtsppitch**	Specifies the direction vector used by infinite light and spotlights. This vector is ignored for point lights.
fltLtspConstAtten **fltLtspLinearAtten** **fltLtspQuadAtten**	Controls the decrease in light as the distance from the light source increases. This has no effect on infinite lights. The attenuation factor is defined by the equation: *Attenuation Factor = 1/(Constant + Linear * d + Quadratic * d^2)* where *d* is the distance from the light source position to a vertex.
fltLtspModeling	Specifies whether the light is a *modeling light*. Modeling lights are always enabled, even if they do not appear in the hierarchy as light source nodes.

Positioning a Light Source

Because an infinite light source is defined by a vector, it does not have a point of origin in the database.

Local and spotlight sources enabled through light source nodes are initially located at the origin, and can be repositioned and reoriented by modifying their **fltLtsPosition**, **fltLtsYaw**, or **fltLtsPitch** attributes. The direction attributes specified for a light source palette entry (**fltLtspYaw** and **fltLtspPitch**) are only used for modeling lights. Light source node attributes (**fltLtsYaw** and **fltLtsPitch**) override the values specified in the palette.

Local and spot modeling lights (lights enabled in the palette rather than by insertion into the hierarchy) are positioned at the eyepoint.

Saving and Loading a Light Source

To create a new light source in the palette, use **mgNewLightSource**, and to delete a light source from the palette, use **mgDeleteLightSource**. Use **mgWriteLightSourceFile** to save a palette to a file. Use **mgReadLightSourceFile** to read a light source palette from a file.

Related Functions:

mgGetLightSource
mgIndexOfLightSource
mgNameOfLightSource
mgGetLightSourceCount
mgGetFirstLightSource
mgGetNextLightSource
mgNewLightSource
mgDeleteLightSource
mgReadLightSourceFile
mgWriteLightSourceFile

Related Record Codes:

fltLightSource
fltLightSourcePalette
fltIcoord
fltColorRGBA

Sample: eglight1.c

```
/*****************************************************************************

  Sample file: EGLIGHT1.C

  Objective: Show how to access, modify, and create entries in the light
     source palette.

  Program functions: Read in a database file that has a light source palette
     Get the first light source entry and the light source count.
     Change some of the attributes of the light source.
     Step through all the light sources.
     Build and add a light source entry.
     Write the light source palette as a file.

  API functions used:
     mgInit(), mgExit(), mgFree(), mgGetLastError(),
     mgGetLightSource(), mgIndexOfLightSource(), mgSetColorRGBA(),
     mgNameOfLightSource(), mgGetColorRGBA(), mgGetAttList(),
     mgGetLightSourceCount(), mgGetFirstLightSource(),
     mgGetNextLightSource(), mgNewLightSource(),
     mgWriteLightSourceFile(), mgCloseDb()

 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

/* include all OpenFlight API headers */
#include <mgapiall.h>

#define LIGHTSOURCEFILE"lightsources.lts"

static void PrintLightSource (mgrec* db, mgrec* lightSource)
{
   int   lightSourceIndex, lightSourceType;
   char*  lightSourceName;
   float ambred, ambgreen, ambblue, ambalpha;
   float diffred, diffgreen, diffblue, diffalpha;
   float specred, specgreen, specblue, specalpha;
   mgbool ambok, diffok, specok, typeok, indexok;
   char lightSourceTypeString[100];

   if (mgGetAttList (lightSource, fltLtspPaletteId, &lightSourceIndex, MG_NULL) == 1)
      indexok = MG_TRUE;
   else
      indexok = MG_FALSE;

   lightSourceName = mgNameOfLightSource (db, lightSourceIndex);

   ambok = mgGetColorRGBA (lightSource,
         fltLtspAmbient, &ambred, &ambgreen, &ambblue, &ambalpha);

   diffok = mgGetColorRGBA (lightSource,
         fltLtspDiffuse, &diffred, &diffgreen, &diffblue, &diffalpha);

   specok = mgGetColorRGBA (lightSource,
         fltLtspSpecular, &specred, &specgreen, &specblue, &specalpha);

   if (mgGetAttList (lightSource, fltLtspType, &lightSourceType, MG_NULL) == 1)
   {
      typeok = MG_TRUE;
      /* from OpenFlight Data Dictionary, lightSourceType value should be:
            0 = INFINITE
            1 = LOCAL
            2 = SPOT
         */
      switch (lightSourceType)
      {
      case 0: strcpy (lightSourceTypeString, "INFINITE"); break;
      case 1: strcpy (lightSourceTypeString, "LOCAL");    break;
      case 2: strcpy (lightSourceTypeString, "SPOT");     break;
      default: strcpy (lightSourceTypeString, "Unknown"); break;
      }
   }
   else {
      typeok = MG_FALSE;
      strcpy (lightSourceTypeString, "Unknown");
   }

   printf ("\nLight Source:\n");
   if (indexok == MG_TRUE)
      printf ("Index:    %d\n", lightSourceIndex);
```

```
   else
      printf ("Error getting index of light source\n");

   printf ("Name:     %s\n", lightSourceName ? lightSourceName : "Unknown");

   if (typeok == MG_TRUE)
      printf ("Type:     %d (%s)\n", lightSourceType, lightSourceTypeString);
   else
      printf ("Error getting type of light source\n");

   if (ambok == MG_TRUE)
      printf ("Ambient:  %5.3f %5.3f %5.3f\n", ambred, ambgreen, ambblue);
   else
      printf ("Error getting ambient color of light source\n");

   if (diffok == MG_TRUE)
      printf ("Diffuse:  %5.3f %5.3f %5.3f\n", diffred, diffgreen, diffblue);
   else
      printf ("Error getting diffuse color of light source\n");

   if (specok == MG_TRUE)
      printf ("Specular: %5.3f %5.3f %5.3f\n", specred, specgreen, specblue);
   else
      printf ("Error getting specular color of light source\n");

   /* mgNameOfLightSource allocs lightSourceName, need to deallocate it */
   if (lightSourceName)
      mgFree (lightSourceName);
}

#define NEWLIGHTNAME "NewLight"

int main (int argc, char* argv[])
{
   mgrec* db;
   mgrec* lightSource;
   mgrec* newLight;
   mgbool ok;
   int index, lightSourceCount, newLightSourceIndex, searchIndex;
   char lightSourcePaletteFile[1024];

   if (argc < 2) {
      printf ("\nUsage: %s <input_db_filename>\n", argv[0]);
      printf ("   Reads database: <input_db_filename>\n");
      printf ("   Prints the number of light sources\n");
      printf ("   Prints the first light source before and after making a change to
it\n");
      printf ("   Prints all the light sources\n");
      printf ("   Build and add a light source entry\n");
      printf ("   Writes the light source palette to a file: %s\n", LIGHTSOURCEFILE);
      printf ("\n");
      exit (EXIT_FAILURE);
   }

   /* initialize the OpenFlight API
      always call mgInit BEFORE any other OpenFlight API calls
      */
```

```
mgInit (&argc, argv);

/* open database */
printf ("\nOpening database: %s\n", argv[1]);
db = mgOpenDb (argv[1]);
if (db == MG_NULL) {
   char msgbuf [1024];
   mgGetLastError (msgbuf, 1024);
   printf ("%s\n", msgbuf);
   exit (EXIT_FAILURE);
}

/* check for light sources in this database */
lightSourceCount = mgGetLightSourceCount (db);
if (lightSourceCount == 0) {
   printf ("No light sources in database.\n");
   mgCloseDb (db);
   exit (EXIT_FAILURE);
}

printf ("Total number of light sources in database:  %d\n", lightSourceCount);

/* get the first light source */
lightSource = mgGetFirstLightSource (db, &index);

/* since we know there is at least one light source in the file (see above)
   lightSource better be valid, check just in case
   */

if (lightSource != MG_NULL) {
   printf ("First Light before changing:\n");
   /* get and print the attributes of this light source */
   PrintLightSource (db, lightSource);
}

else {
   printf ("No light sources in database.\n");
   mgCloseDb (db);
   exit (EXIT_FAILURE);
}

/* now change the ambient color */
ok = mgSetColorRGBA (lightSource, fltLtspAmbient, 0.5f, 0.5f, 0.5f, 0.0f);
printf ("Setting ambient for light source: %s\n", ok==MG_TRUE ? "Ok" : "Failed");

/* now print attributes of first light source, notice that the
   ambient value has changed
   */
lightSource = mgGetFirstLightSource (db, &index);
printf ("First Light after changing:\n");
PrintLightSource (db, lightSource);

printf ("\n");
/* create a new light source palette entry */
newLight = mgNewLightSource (db, NEWLIGHTNAME, &newLightSourceIndex);
if (newLight != MG_NULL)
   printf ("New light source created at index: %d\n", newLightSourceIndex);
```

```
   else
      printf ("Error creating new light source:\n");

   /* search for a light source by name */
   searchIndex = mgIndexOfLightSource (db, NEWLIGHTNAME);
   if (searchIndex == -1)
      printf ("Light source named '%s' NOT found %d\n", NEWLIGHTNAME);
   else
      printf ("Light source named '%s' found at index: %d\n", NEWLIGHTNAME,
searchIndex);

   /* now print the attributes of all the light sources */
   printf ("All Light Sources in Palette:\n");
   lightSource = mgGetFirstLightSource (db, &index);
   while (lightSource != MG_NULL) {
      PrintLightSource (db, lightSource);
      lightSource = mgGetNextLightSource (lightSource, &index);
   }

   strcpy (lightSourcePaletteFile, LIGHTSOURCEFILE);
   ok = mgWriteLightSourceFile (db, lightSourcePaletteFile);
   if (ok == MG_TRUE)
      printf ("Light Source Palette saved to: %s\n", lightSourcePaletteFile);
   else
      printf ("Error writing Light Source Palette file: %s\n", lightSourcePaletteFile);

   /* close the database */
   ok = mgCloseDb (db);
   if (ok == MG_FALSE)
      printf ("Error closing database\n");

   /* always call mgExit() AFTER all OpenFlight API calls */
   mgExit ();

   exit (0);
}
```

Light Point Palette

There are two palettes associated with Light Points, the Light Point Appearance palette and the Light Point Animation palette. Both Light Point Appearance and Animation palette items are referred to by index. The Light Point palette window defines a current light point appearance as well as a current light point animation item.

Every OpenFlight database has its own light point appearance and animation palettes containing the set of light point definitions that can be applied to light points. To traverse a database's light point palettes, use **`mgGetFirstLightPointAppearance`**, **`mgGetNextLightPointAppearance`**, **`mgGetFirstLightPointAnimation`**,

mgGetNextLightPointAnimation, mgGetLightPointAppearanceCount, and **mgGetLightPointAnimationCount.** To get a specific light point appearance or animation in the palette when given the index, use **mgGetLightPointAppearance** or **mgGetLightPointAnimation,** respectively.

Like the other palettes, elements in the light point palettes have names that can be used to identify them. Use **mgNameOfLightPointAppearance** or **mgNameOfLightPointAnimation** to retrieve the name of a light point appearance or animation, respectively, given the item's index. Similarly, use **mgIndexOfLightPointAppearance** or **mgIndexOfLightPointAnimation** to retrieve the index of a light point appearance or animation, respectively, given an item's name.

Each light point appearance and animation definition is made up of several attributes. Use the functions **mgGetAttList** and **mgSetAttList** to get and set individual attributes of these palette definitions. Please refer to the *OpenFlight Data Dictionary* for the complete list of attributes.

To create and save your own light point palette, use **mgNewLightPointAppearance, mgNewLightPointAnimation** and **mgWriteLightPointFile**. To read the contents of a light point palette file into an existing database, use **mgReadLightPointFile.** To delete entries from the light point palettes, use **mgDeleteLightPointAppearance** and **mgDeleteLightPointAnimation.**

Related Record Codes:
fltLpAppearancePalette
fltLpAnimationPalette

Texture Palette

Texture palettes are described in "The Texture Palette" on page 136.

Texture Mapping Palette

Texture mapping palettes are described in "Mapping Through the Texture Mapping Palette" on page 149.

Shader Palette

Shaders make it possible for you to control the shape, appearance and motion of geometry in the scene using programmable graphics hardware. Shader properties are defined in a shader palette and can be selectively applied to polygons and meshes in the database using the **`fltPolyShader`** attribute code.

Every OpenFlight database has its own shader palette containing the set of shader definitions that can be applied to a polygon or mesh. To traverse a database's shader palette, use **`mgGetFirstShader`**, **`mgGetNextShader`**, and **`mgGetShaderCount`**. To get a specific shader in the palette when given the index, use **`mgGetShader`**.

Like the other palettes, elements in the shader palette have names that can be used to identify them. Use **`mgNameOfShader`** to retrieve the name of a shader given the item's index. Similarly, use **`mgIndexOfShader`** to retrieve the index of a shader given an item's name.

Each shader definition is made up of several attributes. Use the functions **`mgGetAttList`** and **`mgSetAttList`** to get and set individual attributes of these palette definitions. Please refer to the *OpenFlight Data Dictionary* for the complete list of attributes.

To create a new entry in the shader palette, use **`mgNewShader.`** To delete entries from the shader palette, use **`mgDeleteShader`** or **`mgDeleteShaderByName.`**

Related Record Codes:
`fltShader`

Sound Palette

Creator audio software lets you build three-dimensional sounds into a database file. Every OpenFlight database has its own sound palette file containing a list of the sound files available for use in the database. Using the Sound palette, you can load audio files into a database and assign the sounds contained in the files to emitters (sound nodes) in the database scene.

To traverse a database's sound palette, use **`mgGetFirstSound`**, **`mgGetNextSound,`** and **`mgGetSoundCount`**.

Sound Attributes

To get (or set) the attributes of a sound palette entry, use **mgGetSound** to get the sound record from the palette; then use **mgGetAttList** (or **mgSetAttList**) on the individual attributes.

An entry in the sound palette can optionally have a name. Use **mgIndexOfSound** and **mgNameOfSound** to translate between name and index.

In addition to the name and index, a sound palette entry has the following attributes:

fltSndFilename	Full path of sound
fltSndPathname	Resolved path of sound

Saving and Loading a Sound Record

To create a new sound in the palette, use **mgNewSound**, and to delete a light source from the palette, use **mgDeleteSound**. Use **mgWriteSoundFile** to save a palette to a file as the default.

Related Functions:

mgGetSound
mgIndexOfSound
mgNameOfSound
mgGetSoundCount
mgGetFirstSound
mgGetNextSound
mgNewSound
mgDeleteSound
mgWriteSoundFile

Related Record Codes:

fltSound
fltSoundPalette

Eyepoints

Up to nine eyepoint positions can be stored in an OpenFlight Database. To get (or set) the attributes of an eyepoint record, use **`mgGetEyePoint`** with an index (ranging from 0 through 8) to get the eyepoint record from the database; then use **`mgGetAttList`** (or **`mgSetAttList`**) on the individual attributes. Then use **`mgSetEyePoint`** to store the eyepoint record into the database at a specified index (again, ranging from 0 through 8).

Related Functions:

`mgGetEyePoint`
`mgSetEyePoint`

Related Record Codes:

`fltEyePoint`

7

Textures

Texture patterns are two-dimensional images that are mapped onto polygons in the database to provide a photo-realistic appearance. They can be produced by scanning and editing photographs or manually creating them in a paintbox editor. A *texel* (texture element) is the unit of resolution that defines a texture pattern.

OpenFlight databases use two files for each texture: a *texture pattern file*, which contains the image, and a *texture attribute file*, which contains information about the texture pattern file, including how it is loaded into memory and displayed.

Texel Formats in Memory

There are four valid types of images: *Intensity*, *Intensity-Alpha*, *RGB*, and *RGBA*. The memory layout for each format is described in the following table.

Intensity Type = **`MIMG_INT`**	These are single-band grayscale images. A texel value of 0 is black and a value of 255 is white. The texels are stored in a contiguous array with the origin (texel 0, 0) in the lower-left corner of the image.

Intensity-Alpha Type = **MIMG_INTA**	These are two-band grayscale images with an alpha (transparency) channel. In the intensity channel, a texel value of 0 is black and a value of 255 is white. In the alpha channel, a texel value of 0 is transparent and a value of 255 is opaque. Texels are stored in a contiguous array with the origin (texel 0, 0) in the lower-left corner of the image. All the intensity values are stored contiguously, followed by all the alpha values.
RGB Type = **MIMG_RGB**	These are three-band color (Red, Green, Blue) images. The intensity of each color component for a texel value of 0 is black; a value of 255 is full intensity. Texels are stored in a contiguous array with the origin (texel 0, 0) in the lower-left corner of the image. All the red values are stored contiguously, followed by all the green values, followed by all the blue values.
RGBA Type = **MIMG_RGBA**	These are three-band color (Red, Green, Blue) images with an alpha (transparency) channel. The intensity of each color component for a texel value of 0 is black; 255 is full intensity. In the alpha channel, a texel value of 0 is transparent and a value of 255 is opaque. The texels are stored in a contiguous array with the origin (texel 0, 0) in the lower-left corner of the image. For this image type the components are interleaved (all four components of each texel are stored contiguously in RGBA order). For example, the first four texels in an image would be stored as RGBARGBARGBARGBA.

Supported Read and Write File Formats

The API can read many image file formats, including:

- SGI
- TIFF
- JPEG/JFIF
- BMP
- IFF/ILBM
- GIF
- PCX
- PNG
- PPM

- Alias Pix

In addition, the API can read any image file format for which an Image Importer plug-in is currently loaded. For example, Sun Raster, Direct Draw Surface (DDS), Targa, JPEG 2000, Clip Texture v1 and v2 as well as DED are all image file formats supported by plug-ins distributed by Presagis with the Creator modeling package.

Note that whenever an external image file format is read into the API, the texels are converted to one of the four memory texel formats described in "Texel Formats in Memory" on page 130.

The API writes the SGI image file formats (RGB, RGBA, INT and INTA) only.

Creating and Modifying Textures

While the API is not intended for full-feature texture editing, it does have support for some simple tasks. There are two components you must consider when creating and modifying textures: the attributes and the image.

Texture attributes for a texture file are stored on disk in a companion file whose suffix is **`.attr`**. This companion file is assumed to be located in the same folder as the texture file to which it is associated. All texture attributes are accessed through a texture attribute **`mgrec`** structure. To obtain this record for a palette index, call **`mgGetTextureAttributes`**. To store or retrieve individual attributes within the record, use **`mgSetAttList`** or **`mgGetAttList`**. For an example, see the sample program `egtexture1.c`.

When you define a new entry in the texture palette, you can attach an *attributes record* with **`mgSetTextureAttributes`**.

Changing texture attributes is possible with existing functions: you can set a texture attribute just like you set any other attribute in an OpenFlight database, using **`mgSetAttList`**. To save the texture attributes after making modifications, use **`mgWriteImageAttributes`**. This will save the texture attributes in the companion **`.attr`** file.

Additionally, the API provides several functions to make it easier to create and edit texture images. These functions simply read and write image and attribute files and are identified by the word "Image" in the function name. They are useful for converting image file formats or for performing simple

image manipulations (that is, read an image, filter the image, write the image). They include the following:

- **mgReadImage**
- **mgReadImageHeader**
- **mgReadImageAttributes**
- **mgFlipImage**
- **mgWriteImage**
- **mgWriteImageAttributes**

Call **mgNewTexture** to create a new texture, and **mgSetTextureTexels** and **mgSetTextureName** to initialize it. Call **mgDeleteTexture** to delete a texture.

The following example is a simple program to reverse the order of the channels of an image.

Sample: egimageio1.c

```
/*****************************************************************************

  Sample file: EGIMAGEIO1.C

  Objective: Show how to access and manipulate an image.

  Program functions: Reads an image from the command line.
     Swaps the channels of the image.
     Writes the swapped image.

  API functions used:
     mgInit(), mgExit(),
     mgReadImage(), mgWriteImage(),
     mgReadImageAttributes(), mgWriteImageAttributes()

 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <openflightapi/mgapiall.h>

static void PrintError (char *msg)
{
   if (msg)
      printf ("%s\n", msg);
}

static int SwapChannels (unsigned char *pixels, unsigned char *newpixels,
```

```
                int type, int width, int height)
{
   unsigned char *iptr1, *iptr2, *iptr3;
   unsigned char *optr1, *optr2, *optr3;
   int i, size;

   size = width * height;
   switch (type)
   {
      case MIMG_INT:     /* intensity - 1 channel - no need to swap */
         memcpy (newpixels, pixels, size * (type -1));
         return (MG_TRUE);

      case MIMG_INTA:     /* intensity alpha - 2 channels */
         iptr1 = pixels;
         iptr2 = pixels + size;
         optr1 = newpixels;
         optr2 = newpixels + size;
         i = size;
         while (i--)
         {
            *optr1++ = *iptr2++;
            *optr2++ = *iptr1++;
         }
         return (MG_TRUE);

      case MIMG_RGB:     /* RGB - 3 channesl - only need to swap channels 1 & 3 */
         iptr1 = pixels;
         iptr2 = pixels + size;
         iptr3 = pixels + 2*size;
         optr1 = newpixels;
         optr2 = newpixels + size;
         optr3 = newpixels + 2*size;
         i = size;
         while (i--)
         {
            *optr1++ = *iptr3++;
            *optr2++ = *iptr2++;
            *optr3++ = *iptr1++;
         }
         return (MG_TRUE);

      case MIMG_RGBA:     /* RGBA - 4 channels */
            /* Note the different way the channels are stored */
         iptr1 = pixels;
         optr1 = newpixels;
         i = size;
         while (i--)
         {
            *optr1 = *(iptr1+3);
            *(optr1+1) = *(iptr1+2);
            *(optr1+2) = *(iptr1+1);
            *(optr1+3) = *iptr1;
            iptr1 += 4;
            optr1 += 4;
         }
         return (MG_TRUE);
```

```
      default:
         return (MG_FALSE);
   }
}

void main (int argc, char **argv)
{
   int type, width, height, status;
   unsigned char *pixels, *newpixels;
   mgrec *attr_rec;

   /* check for proper arguments */
   if (argc < 3) {
      printf ("Usage: %s infile outfile\n", argv[0]);
      exit (0);
   }

   /* Read the input image, pixels is allocated, remember to mgFree */
   status = mgReadImage (argv[1], &pixels, &type, &width, &height);

   if (status != 0) {
      PrintError ("Could not read image");
      exit (EXIT_FAILURE);
   }

   /* Read the input image attributes */
   if (attr_rec = mgReadImageAttributes (argv[1])) {
      PrintError ("Could not read image attributes");
      exit (EXIT_FAILURE);
   }

   /* Allocate the swapped image */
   newpixels = (unsigned char *) mgMalloc (width * height * (type -1));
   if (!newpixels) {
      PrintError ("Could not allocate swapped image");
      exit (EXIT_FAILURE);
   }

   /* Swap the channels */
   if (!SwapChannels (pixels, newpixels, type, width, height)) {
      PrintError ("Could not swap the channels for this image");
      exit (EXIT_FAILURE);
   }

   /* Write the swapped image */
   status = mgWriteImage (argv[2], newpixels, type, width, height, MG_FALSE);
   if (status != 0) {
      PrintError ("Could not write image");
      exit (EXIT_FAILURE);
   }

   /* Write the swapped image attributes */
   if (!mgWriteImageAttributes (argv[2], attr_rec)) {
      PrintError ("Could not write image attributes");
      exit (EXIT_FAILURE);
   }
```

```
   mgFree (pixels);
   mgFree (newpixels);

   /* exit */
   mgExit ();
   exit (0);
}
```

The Texture Palette

Each OpenFlight database has its own texture palette. A texture must be listed in the texture palette before it can be used by a database.

Each texture in the palette has a unique name and index. Nodes reference textures by this index.

The texture palette is saved in the OpenFlight database. However, the palette is merely a list of the texture files assigned to the database. To conserve disk space, texture patterns are saved in individual files that can be shared among databases. The API uses the palette to reload the applicable patterns the next time you open the database file.

If a texture is used by several open databases, there is only one instance of the texels. If the texels or attributes are changed for one database, they will be changed for all of the databases.

Reading the Texture Palette

The API provides several functions to query a database's texture palette and to get information about specific textures.

Use **`mgGetFirstTexture`** and **`mgGetNextTexture`** to walk the texture palette. Provide records returned by these functions as input to **`mgGetTextureTexels`** and **`mgGetTextureAttributes`**, which return a texture's pattern and attributes.

To translate between a texture's name and index, use **`mgGetTextureIndex`** and **`mgGetTextureName`**.

The following programming example prints the index, name, and size of each texture in a database's texture palette.

Sample: egtexture2.c

```
/*****************************************************************************

  Sample file: EGTEXTURE2.C

  Objective: Show how to access information from textures in the texture
     palette.

  Program functions: Open a database from file name given on command line.
     Steps through all the textures in the database's
     texture palette and prints the texture attributes of
     each.

  API functions used:
     mgInit(), mgExit(),
     mgGetFirstTexture(), mgGetTextureAttributes(),
     mgGetAttList(), mgGetNextTexture().

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

void main (int argc, char **argv)
{
   int type, width, height;
   int patindex;
   mgrec *attr_rec, *db;
   char patname[1024];

   /* check for correct number of arguments */
   if (argc < 2) {
      printf ("Usage: %s <input_db_filename>\n", argv[0]);
      exit (EXIT_FAILURE);
   }

      /* Initialize the API */
   mgInit (&argc, argv);

      /* Load the database */
   if (!(db = mgOpenDb (argv[1]))) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      mgExit ();
      exit (EXIT_FAILURE);
   }

      /* Walk the texture palette & print the width, height & type of each texture */
   if (mgGetFirstTexture (db, &patindex, patname)) {
      do {
         if (attr_rec = mgGetTextureAttributes (db, patindex)) {
```

```
            mgGetAttList (attr_rec, fltImgWidth, &width,
                                    fltImgHeight, &height,
                                    fltImgType, &type,
                                    MG_NULL);

            printf ("Texture %d: %s: width = %d, height = %d, numChannels = %d\n",
                                          patindex, patname, width, height, (type-1));
         }
         else {
            printf ("Texture %d: %s: Error: cannot get attributes\n",
                                          patindex, patname);
         }
      } while (mgGetNextTexture (db, &patindex, patname));
   }

      /* Close the database and exit*/
   mgCloseDb (db);
   mgExit ();
}
```

Loading, Creating, and Modifying Textures

You can use the API to load or copy existing textures, or (to a limited extent) to create your own texture patterns.

The API provides several functions to load textures:

- Call **`mgIsTextureInPalette`** before loading a new texture, to make sure that it isn't loaded already.
- Use **`mgReadTexture`** or **`mgReadTextureAndAlpha`** if you want to specify the texture's index and position.
- Use **`mgInsertTexture`** or **`mgInsertTextureAndAlpha`** to let the API automatically assign the next available index and position.
- Use **`mgReplaceTexture`** to load a new texture that replaces an existing palette entry.
- Use **`mgNewTexture`** to create a new texture in the palette from the texels in memory.

You can copy a single texture or the entire palette between databases with **`mgCopyTexture`** and **`mgCopyTexturePalette`**.

To save a texture's image and attributes files to disk, use **`mgWriteTexture`**. You must specify the files' prefix; the API assigns suffixes to both files according to their image types.

The following programming example loads two databases, loads a texture into the first database, copies the texture into the second database, and then writes the two databases.

Sample: egtexture5.c

```
/*****************************************************************************

  Sample file: EGTEXTURE5.C

  Objective: Show how to add a texture to a texture palette.
     Show how to copy a texture from one database to another.

  Program functions: Opens two databases from filenames given on command line.
     Reads in a texture from command line.
     Adds the texture to the first database's texture palette.
     Copies the texture to the second database's texture palette.
     Writes out both databases.

  API functions used:
     mgInit(), mgExit(),
     mgInsertTexture(), mgCopyTexture().

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

static void PrintError (char *msg)
{
   if (msg)
      printf ("%s\n", msg);
}

void main (int argc, char **argv)
{
   int newindex;
   mgrec *db1, *db2;

   /* check for proper arguments */
   if (argc < 4) {
      printf ("Usage: %s <input_db_filename> <ouput_db_filename> <image_filename>\n",
argv[0]);
      exit (EXIT_FAILURE);
   }

   /* Initialize the API */
   mgInit (&argc, argv);

   /* Load the databases */
   if (!(db1 = mgOpenDb (argv[1]))) {
```

```
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      exit (EXIT_FAILURE);
   }
   if (!(db2 = mgOpenDb (argv[2]))) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      exit (EXIT_FAILURE);
   }

   /* Read a new texture into the first database's palette */
   newindex = mgInsertTexture (db1, argv[3]);

   printf ("Texture %s added to Database %s at index %d\n",
               argv[3], argv[1], newindex);

   /* Copy the new texture into the second database's palette */
   newindex = mgCopyTexture (db2, db1, argv[3], newindex);

   printf ("Texture %s added to Database %s at index %d\n",
               argv[3], argv[2], newindex);

   /* write the databases */
   if (!mgWriteDb (db1))
   {
      PrintError ("Database write failed");
      exit (EXIT_FAILURE);
   }
   if (!mgWriteDb (db2))
   {
      PrintError ("Database write failed");
      exit (EXIT_FAILURE);
   }

   /* Close the databases and exit */
   mgCloseDb (db1);
   mgCloseDb (db2);
   mgExit ();
}
```

Positioning Textures in the Palette

The texture palette has 128 *banks*; each bank can hold up to 256 textures. Textures with indices of 0-255 appear in bank 0, textures 256-511 appear in bank 1, and so on, up to a maximum index of 32767 in bank 127. You do not need to fill a bank before you can use the next one. For example, you can have a palette with three textures: index 0 and 1 in the first bank, and index 256 in the second.

You can also specify the *position* of a texture within a bank. The position of a texture specifies the texel location of the lower-left corner of the pattern relative to the lower-left corner of the bank as displayed in the Creator texture palette. Use the function **`mgSetTexturePosition`** to set the position of a texture. Use the function **`mgGetTexturePosition`** to retrieve the position of a texture. While the position of a texture is very important to a modeler trying to visually locate textures in the palette, it may not be as interesting to an API user (who typically keeps track of the texture palette entries programmatically). Unless you have a special need to manage the positions manually, you can let the API manage texture positions for you.

Texture Palette Statistics

The API maintains a few interesting statistics about textures. You should use these statistics to ensure that your database uses textures as efficiently as possible. To retrieve them call **`mgGetTextureCount`**, **`mgGetTextureSize`**, or **`mgGetTextureTotalSize`**.

The following programming example prints several statistics.

Sample: egtexture4.c

```
/*****************************************************************************

  Sample file: EGTEXTURE4.C

  Objective: Show how to get statistics about the texture palette.

  Program functions: Steps through all the textures in the texture palette and
     print the height, width, type and memory usage of each one.
     Prints the total number of textures and the total
     size of the textures in the texture palette.

  API functions used:
     mgInit(), mgExit(),
     mgGetFirstTexture(), mgGetTextureAttributes(),
     mgGetAttList(), mgGetNextTexture(), mgIsTextureDefault(),
    mgGetTextureCount(), mgGetTextureTotalSize().

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>
```

```
void main (int argc, char **argv)
{
   int type, width, height, mem_size, count;
   int patindex;
   mgrec *attr_rec, *db;
   char  patname[256];

   /* check for correct number of arguments */
   if (argc < 2) {
      printf ("Usage: %s <input_db_filename>\n", argv[0]);
      exit (EXIT_FAILURE);
   }

      /* Initialize the API */
   mgInit (&argc, argv);

      /* Load the database */
   if (!(db = mgOpenDb (argv[1]))) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      exit (EXIT_FAILURE);
   }

      /* Walk the texture palette & print the width, height, type &
         memory usage of each texture */
   if (mgGetFirstTexture (db, &patindex, patname)) {
      do {
         if (attr_rec = mgGetTextureAttributes (db, patindex)) {
            mgGetAttList (attr_rec, fltImgWidth, &width,
                                    fltImgHeight, &height,
                                    fltImgType, &type,
                                    MG_NULL);
            mem_size = mgGetTextureSize (db, patindex);

            if (mgIsTextureDefault (db, patindex))
               printf ("Texture %d (DEFAULT TEXTURE): %s: width = %d, height = %d, "
                       "numChannels = %d, size = %d bytes\n",
                        patindex, patname, width, height, (type-1), mem_size);
            else
               printf ("Texture %d: %s: width = %d, height = %d, "
                       "numChannels = %d, size = %d bytes\n",
                        patindex, patname, width, height, (type-1), mem_size);
         }
         else {
            printf ("Texture %d: %s: Error: cannot get attributes\n",
                        patindex, patname);
         }
      } while (mgGetNextTexture(db, &patindex, patname));
   }

      /* Get the total texture count & memory usage */
   count = mgGetTextureCount(db);
   mem_size = mgGetTextureTotalSize(db);

   printf ("\n");
   printf ("Total %d textures using %d bytes\n", count, mem_size);
```

```
      /* Close the database and exit */
   mgCloseDb (db);
   mgExit ();
}
```

The Default Texture

The API automatically loads a database's textures into the palette when the database is opened. If a texture referenced in the database cannot be found, a default texture (a 16 x 16 texel image of an ***X***) is loaded into the palette in its place.

To determine if a texture's texels have been replaced by the default texture, call **`mgIsTextureDefault`**.

Saving and Restoring Texture Palettes

The texture palette file is an ASCII file that contains a list of entries with the following format:

`<palnum> <patnum> <x> <y> <file name>`

where:

`<palnum>` (range 0 to 95)
specifies the number of the palette in which the pattern is loaded.

`<patnum>` (range 0 to 255)
specifies the local pattern ID on the palette specified by **`<palnum>`**.

`<x>,<y>` (range 0 to 2047)
specify the offset from the lower-left corner of the palette to the lower-left corner of the pattern.

`<file name>`
specifies the directory path and file name of the texture pattern.

To read or write a texture palette file, use **`mgReadTexturePalette`** or **`mgWriteTexturePalette`**.

Example

The following programming example loads a texture palette file into a database's texture palette, writes the new palette to a new texture palette file and saves the database with the new palette. Note that when textures in the new texture palette file have the same index as textures in the database's existing palette, the existing textures are replaced. Textures in the file with indices not in the palette will simply be added to the palette. Textures in the palette with indices not in the file are not affected. If you want to completely replace the contents of a database's palette with a texture palette file, you must first delete all textures in the database's palette. The "-r" option at the end of the command line of the sample program demonstrates this.

Sample: egtexture6.c

```
/*****************************************************************************

  Sample file: EGTEXTURE6.C

  Objective: Show how to load and save textures from a texture palette file.

  Program functions: Opens a database from the command line.
     Loads a texture palette file into the database's texture palette.
     Writes the new texture palette to a new texture palette file.
     Writes the database with the new texture palette.
     The "-r" option means that the database's palette will
     be completely replaced by the new one.

  API functions used:
     mgInit(), mgExit(),
     mgGetTextureCount(), mgGetFirstTexture(),
     mgGetNextTexture(), mgDeleteTexture(),
     mgReadTexturePalette(), mgWriteTexturePalette().

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

static void PrintError (char *msg)
{
   if (msg)
      printf ("%s\n", msg);
}

static int ClearPalette (mgrec *db)
{
```

```
   int numtxtrs, i;
   int *txtrindices;    /* Texture index list */
   char patname[1024];

      /* Get a count of the textures in the palette */
   numtxtrs = mgGetTextureCount (db);

      /* Allocate a texture index list */
   txtrindices = (int *) malloc(numtxtrs * sizeof(int));
   if (!txtrindices)
      return (MG_FALSE);

      /* Collect the texture indices */
   i = 0;
   mgGetFirstTexture (db, &txtrindices[i], patname);
   while (mgGetNextTexture (db, &txtrindices[i], patname))
      i++;

      /* Delete each texture by index */
   for (i = 0; i < numtxtrs; i++) {
      mgDeleteTexture (db, txtrindices[i]);
   }

      /* Free the index list */
   free (txtrindices);

   return (MG_TRUE);
}

void main (int argc, char **argv)
{
   mgrec *db;

   if (argc < 4) {
      printf ("Usage: %s <input_db_filename> <input_palette_filename> "
              "<output_palette_filename> [-r]\n", argv[0]);
      exit (EXIT_FAILURE);
   }

      /* Initialize the API */
   mgInit (&argc, argv);

      /* Load the database */
   if (!(db = mgOpenDb (argv[1]))) {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      mgExit ();
      exit (EXIT_FAILURE);
   }

      /* Check for the replace palette option */
   if ((argc > 4) && (strcmp(argv[4], "-r") == 0)) {
      if (!ClearPalette (db)) {
         PrintError ("Failed to clear Palette");
         mgExit ();
         exit (EXIT_FAILURE);
```

```
    }
  }

    /* Load the texture palette file */
  if (!mgReadTexturePalette (db, argv[2])) {
    PrintError ("Texture palette file read failed");
    mgExit ();
    exit (EXIT_FAILURE);
  }

    /* Write the database's texture palette to a file */
  if (!mgWriteTexturePalette (db, argv[3])) {
    PrintError ("Texture palette file write failed");
    mgExit ();
    exit (EXIT_FAILURE);
  }

    /* write the database */
  if (!mgWriteDb (db)) {
    PrintError ("Database write failed");
    mgExit ();
    exit (EXIT_FAILURE);
  }

    /* close the database and exit */
  mgCloseDb (db);
  mgExit ();
}
```

Mapping Texture to Geometry

The OpenFlight format supports 8 individual texture layers on geometry (polygons and meshes). For each layer, you define which texture is applied to that layer as well as how the texture coordinates are mapped to vertices in that layer.

To specify which texture is applied to your geometry in a particular layer, you assign a texture from the palette to that layer of the geometry. This is achieved by assigning the corresponding texture index attribute (on the polygon or mesh) for the layer in which you want the texture to appear:

`fltPolyTexture` - base layer 0
`fltLayerTexture1` - layer 1
`fltLayerTexture2` - layer 2
`fltLayerTexture3` - layer 3
`fltLayerTexture4` - layer 4
`fltLayerTexture5` - layer 5

fltLayerTexture6 - layer 6
fltLayerTexture7 - layer 7

The following assigns separate texture indices to both the base layer and to layer 1 of a polygon **poly**, in database **db**:

```
// add two patterns to the palette, one for the base layer
// and one for layer 1
short index[2];
index[0] = mgInsertTexture (db, "base.rgb");
index[1] = mgInsertTexture (db, "layer1.rgb");

// assign palette indices for base layer and layer 1 of poly
mgSetAttList (poly, fltPolyTexture, index[0],
                    fltLayerTexture1, index[1], MG_NULL);
```

Note that a texture index of -1 specifies that no texture is applied in that particular layer of geometry.

Polygons and meshes can also have an optional *detail texture*, which blends in when the face is viewed at close range. Specify a detail texture index by assigning the **fltPolyTexture1** attribute. A detail texture index of -1 means that the polygon or mesh is not mapped with a detail texture.

After you assign texture indices to the layers of the geometry, you must specify how you want to map the texture coordinates of the vertices of your geometry. Texture coordinates can be mapped to geometry in two ways:

- Direct assignment of texture u,v coordinates to vertices - in this way, u,v texture coordinates are explicitly assigned (per layer) to each vertex of the textured geometry.
- Reference to an entry in the texture mapping palette - in this way, u,v texture coordinates are automatically calculated at runtime given a particular mapping palette entry.

These two techniques for mapping texture coordinates are described in the following sections.

Mapping by Explicit Texture Coordinates

Texture u,v coordinates range from 0.0 to 1.0 for a single repetition of a texture pattern. By convention, (u,v) = (0.0, 0.0) is the lower left corner of the pattern while (u,v) = (1.0, 1.0) is the upper right. Assigning u,v coordinates greater than 1.0 or less than 0.0 are used to represent a repetition of the

texture. To map a texture to a polygon using explicit texture coordinates, you assign u,v texture coordinates at each vertex of the polygon. These coordinates at each vertex represent the 2 dimensional relationship (mapping) between the texture pattern and the planar polygon in space. For vertices of polygons, texture coordinates are assigned (per layer) using these attributes (at the vertex record level):

`fltVU,fltVV` - u,v coordinates for base layer 0
`fltLayerU1,fltLayerV1` - u,v coordinates for base layer 1
`fltLayerU2,fltLayerV2` - u,v coordinates for base layer 2
`fltLayerU3,fltLayerV3` - u,v coordinates for base layer 3
`fltLayerU4,fltLayerV4` - u,v coordinates for base layer 4
`fltLayerU5,fltLayerV5` - u,v coordinates for base layer 5
`fltLayerU6,fltLayerV6` - u,v coordinates for base layer 6
`fltLayerU7,fltLayerV7` - u,v coordinates for base layer 7

You store and retrieve these attributes just like any attribute, using **`mgGetAttList`** and **`mgSetAttList`**.

For meshes, use **`mgMeshGetVtxUV`** and **`mgMeshSetVtxUV`** to store and retrieve u,v texture coordinates on vertices of the mesh.

The following continues our previous example. In it, explicit texture coordinates are assigned for the vertices of **`poly`** (which we have already assigned texture indices for the base layer and layer 1). The polygon is a simple rectangle. We will map one repetition of the pattern in the base layer. We will map two repetitions of the pattern in layer 1:

```
// assume poly is a simple rectangle (4 vertices)
mgrec* v[4];

// get each vertex of the rectangle
v[0] = mgGetChild (poly);
v[1] = mgGetNext (v[0]);
v[2] = mgGetNext (v[1]);
v[3] = mgGetNext (v[2]);

// assign uv texture coordinates for both layers
mgSetAttList (v[0], fltVU, 0.0, fltVV, 0.0,
              fltLayerU1, 0.0, fltLayerV1, 0.0, MG_NULL);
mgSetAttList (v[1], fltVU, 1.0, fltVV, 0.0,
              fltLayerU1, 2.0, fltLayerV1, 0.0, MG_NULL);
mgSetAttList (v[2], fltVU, 1.0, fltVV, 1.0,
              fltLayerU1, 2.0, fltLayerV1, 2.0, MG_NULL);
mgSetAttList (v[3], fltVU, 0.0, fltVV, 1.0,
              fltLayerU1, 0.0, fltLayerV1, 2.0, MG_NULL);
```

To summarize this section, the steps needed to apply and map texture to geometry using explicit texture coordinates involve the following steps:

- Load the desired texture(s) into the palette of the database
- Assign texture index attribute on polygon or mesh in the desired layer
- Assign u,v texture coordinate attributes on each of the vertices in the desired layer

Mapping Through the Texture Mapping Palette

The OpenFlight format provides another mechanism for mapping texture to geometry that does not involve assigning explicit u,v texture coordinates to individual vertices. To explain this alternative method, we introduce the Texture Mapping Palette. This palette is a mechanism for storing parameters used by the texture applications tools found in the Presagis modeling software. This can greatly simplify the texture mapping process but may not provide the flexibility or precision required in certain situations.

Each entry in the texture mapping palette represents one of the following mapping types:

- 3 Point Put
- 4 Point Put
- Spherical Project
- Radial Project
- Environment

Note: The 3 Point Put and 4 Point Put mappings also contain a matrix that is suitable for use with the linear mappings provided by OpenGL's **`glTexGen`** function.

To map a texture to a polygon using an entry from the texture mapping palette, you simply assign a texture mapping index to the polygon or mesh. This is achieved by assigning the corresponding texture mapping index attribute (on the polygon or mesh) for the layer in which you want the texture to appear:

`fltPolyTexmap` - base layer 0
`fltLayerTexmap1` - layer 1

fltLayerTexmap2 - layer 2
fltLayerTexmap3 - layer 3
fltLayerTexmap4 - layer 4
fltLayerTexmap5 - layer 5
fltLayerTexmap6 - layer 6
fltLayerTexmap7 - layer 7

Each database has one texture mapping palette. Each texture mapping is stored in the palette with a unique index and an optional name. A user may name the mappings, but this is not required. Polygons and meshes within a database refer to texture mappings for each layer by their index. These attribute names are listed above for each layer.

Texture mapping palette functions are identified by the word 'TextureMapping' in the function name. These functions manage a database's texture mapping palette and allow you to query entries in the palette.

Functions are provided to query a database's texture mapping palette and to get information about specific texture mappings. Use **mgGetFirstTextureMapping** and **mgGetNextTextureMapping** to walk the texture palette.

You can retrieve three attributes from a texture mapping: *name*, *type*, and *mapping matrix*. Use **mgGetTextureMappingName**, **mgGetTextureMappingType**, and **mgGetTextureMappingMatrix** to retrieve them. Use **mgIsTextureMappingInPalette** to check the validity of a texture mapping index.

Note that when a texture mapping is assigned to a polygon or mesh, u,v texture coordinates for the vertices of the geometry are calculated automatically and are recomputed as the geometry is moved or scaled in the Creator. If explicit u,v texture coordinates are assigned to vertices in the geometry, they are ignored. This is expected of all OpenFlight runtime environments as well. If you want to revert to using explicit u.v texture coordinates for geometry, you must reset the polygon or mesh texture mapping index to -1.

The OpenFlight API and Creator calculate explicit u,v texture coordinates for each vertex contained in geometry that has a texture mapping applied, and assign the corresponding vertex attributes when a file is saved. In this way, runtime environments that do not support mapping textures via the texture

mapping palette, can simply use the explicit u,v coordinates stored on the vertices in the OpenFlight file instead.

To create and save your own texture mapping palette, use **mgNewTextureMapping** and **mgWriteTextureMappingFile**. To read the contents of a texture mapping palette file into an existing database, use **mgReadTextureMappingFile**. To delete entries from the palette, use **mgDeleteTextureMapping** and **mgDeleteTextureMappingByName**.

The following programming example prints the index, name, and type of each texture mapping in a database's texture mapping palette. If the mapping type is either *3 Point Put* or *4 point Put*, the mapping matrix is also printed.

Sample: egtexture3.c

```
/*****************************************************************************

  Sample file: EGTEXTURE3.C

  Objective: Show how to access information from the texture mapping palette.

  Program functions: Steps through all the texture mappings in the texture
     mapping palette and prints the name and type of each one.

  API functions used:
     mgInit(), mgExit(),
     mgGetFirstTextureMapping(), mgGetNextTextureMapping(),
     mgGetTextureMappingType(), mgGetTextureMappingMatrix()

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

static void PrintMatrix (mgmatrix matrix)
{
   printf ("\tmatrix = {\n");
   printf ("\t\t%f\t%f\t%f\t%f\n", matrix[0],  matrix[1],  matrix[2],  matrix[3]);
   printf ("\t\t%f\t%f\t%f\t%f\n", matrix[4],  matrix[5],  matrix[6],  matrix[7]);
   printf ("\t\t%f\t%f\t%f\t%f\n", matrix[8],  matrix[9],  matrix[10], matrix[11]);
   printf ("\t\t%f\t%f\t%f\t%f\n", matrix[12], matrix[13], matrix[14], matrix[15]);
   printf ("\t}\n");
}

void main (int argc, char **argv)
{
   int type;
   int mapindex;
```

```
mgrec *db;
char mapname[256];
mgmatrix matrix;

/* check for correct number of arguments */
if (argc < 2) {
   printf ("Usage: %s <input_db_filename>\n", argv[0]);
   exit (EXIT_FAILURE);
}

/* Initialize the API */
mgInit (&argc, argv);

/* Load the database */
if (!(db = mgOpenDb (argv[1]))) {
   char msgbuf [1024];
   mgGetLastError (msgbuf, 1024);
   printf ("%s\n", msgbuf);
   mgExit ();
   exit (EXIT_FAILURE);
}

/* Walk the mapping palette & print the name & type of each mapping */
if (mgGetFirstTextureMapping (db, &mapindex, mapname)) {
   do {
      type = mgGetTextureMappingType (db, mapindex);
      switch (type)
      {
      case 1:
         if (*mapname)
            printf ("Texture Mapping %d: %s: type = 3 Point Put\n",
                     mapindex, mapname);
         else
            printf ("Texture Mapping %d: (no name): type = 3 Point Put\n",
                     mapindex);
         if (mgGetTextureMappingMatrix (db, mapindex, &matrix))
            PrintMatrix (matrix);
         else
            printf ("ERROR - No Matrix\n");
         break;
      case 2:
         if (*mapname)
            printf ("Texture Mapping %d: %s: type = 4 Point Put\n",
                     mapindex, mapname);
         else
            printf ("Texture Mapping %d: (no name): type = 4 Point Put\n",
                     mapindex);
         if (mgGetTextureMappingMatrix (db, mapindex, &matrix))
            PrintMatrix (matrix);
         else
            printf ("ERROR - No Matrix\n");
         break;
      case 4:
         if (*mapname)
            printf ("Texture Mapping %d: %s: type = Spherical Project\n",
                     mapindex, mapname);
         else
```

```
                printf ("Texture Mapping %d: (no name): type = Spherical Project\n",
                         mapindex);
            break;
         case 5:
            if (*mapname)
               printf ("Texture Mapping %d: %s: type = Radial Project\n",
                         mapindex, mapname);
            else
               printf ("Texture Mapping %d: (no name): type = Radial Project\n",
                         mapindex);
            break;
         default:
            if (*mapname)
               printf ("Texture Mapping %d: %s: ERROR - Unknown Type\n",
                        mapindex, mapname);
            else
               printf ("Texture Mapping %d: (no name): ERROR - Unknown Type\n",
                        mapindex);
            break;
         }
      } while (mgGetNextTextureMapping (db, &mapindex, mapname));
   }

      /* Close the database and exit */
   mgCloseDb (db);
   mgExit ();
}
```

Putting It All Together

The following programming example ties the texture palette and the texture mapping palette to geometry. It traverses a database and prints the name and index of each polygon's texture and texture mapping. It also prints the type of the texture mapping, and then prints each vertex's texture u,v coordinate:

```
/*****************************************************************************

  Sample file: EGTEXTURE1.C

  Objective: Shows how to access texture information from polygon records.

  Program functions: Read database given on command line.
     Prints texture information (name, index, type, height, and width)
     for each textured polygon in the database.
     Prints texture mapping information for each
     polygon that has a texture mapping.
     Prints the UV information for each vertex of
     textured polygons.

  API functions used:
     mgInit(), mgExit(), mgGetLastError(),
     mgGetAttList(), mgIsCode(), mgGetTextureMappingType(),
```

```
     mgGetTextureName(), mgGetTextureAttributes(),
     mgGetTextureMappingName(), mgGetFirstTexture(), mgWalk(),
     mgOpenDb(), mgCloseDb().

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

/* include all API headers */
#include <openflightapi/mgapiall.h>

static mgbool Action (mgrec* db, mgrec* par, mgrec* rec, void *unused)
{
   short tindex = -1, tmindex = -1;
   char *tname, *tmname;
   mgrec *txtrec;
   int width, height, type;
   float u, v;
   static int hasTexture = MG_FALSE;

   if (!rec)
      return MG_FALSE;

      /* if rec is a polygon or mesh */
   if (mgIsCode (rec, fltPolygon) || mgIsCode (rec, mgMesh))
   {
         /* Get the polygon's texture index */
      mgGetAttList (rec, fltPolyTexture, &tindex,
                         fltPolyTexmap, &tmindex,
                         MG_NULL);

         /* If the polygon is textured */
      if (tindex > -1)
      {
         int numGeoCoords;
         hasTexture = MG_TRUE;

            /* Get some info about the texture */
         tname = mgGetTextureName (db, tindex);
         txtrec = mgGetTextureAttributes (db, tindex);
         mgGetAttList (txtrec, fltImgWidth, &width,
                               fltImgHeight, &height,
                               fltImgType, &type,
                               MG_NULL);

         printf ("Texture index = %d name = %s, type = %d\n", tindex, tname, type);
         printf ("width = %d height = %d\n", width, height);

         mgGetAttList (txtrec, fltTGNumCoords, &numGeoCoords, MG_NULL);
         if (numGeoCoords > 0) {
            mggeocoorddata gcData[10];
            mggeocoorddata addgcData[] = {8, 8, 8, 8, 9, 9, 9, 9, 10, 10, 10, 10};
            int num;
            const int numgcData = sizeof(gcData)/sizeof(mggeocoorddata);
            const int numaddgcData = sizeof(addgcData)/sizeof(mggeocoorddata);
            num = mgGeoCoordGet(txtrec, gcData, numgcData);
```

```
      if (num > 0)
         printf ("Num. Geo Coordinates: %d\n", numGeoCoords);
      while (--num >= 0)
         printf ("u: %lf, v: %lf, lat: %lf, lon: %lf\n",
                  gcData[num].u, gcData[num].v, gcData[num].lat, gcData[num].lon);
      printf ("Adding Geo Coordinate u: %lf, v: %lf, lat: %lf, lon: %lf\n",
                  8.0, 8.0, 8.0, 8.0);
      printf ("Adding Geo Coordinate u: %lf, v: %lf, lat: %lf, lon: %lf\n",
                  9.0, 9.0, 9.0, 9.0);
      printf ("Adding Geo Coordinate u: %lf, v: %lf, lat: %lf, lon: %lf\n",
                  10.0, 10.0, 10.0, 10.0);
      mgGeoCoordAdd (txtrec, addgcData, numaddgcData);
      printf ("Removing Geo Coordinate with indices 0 and 2\n");
      mgGeoCoordDelete (txtrec, 2);
      mgGeoCoordDelete (txtrec, 0);
      num = mgGeoCoordGet (txtrec, gcData, numgcData);
      while (--num >= 0) {
         printf ("u: %lf, v: %lf, lat: %lf, lon: %lf\n",
            gcData[num].u, gcData[num].v, gcData[num].lat, gcData[num].lon);
      }
   }
}
else
   hasTexture = MG_FALSE;

   /* If the polygon has a mapping */
if (tmindex > -1)
{
      /* Get some info about the mapping */
   tmname = mgGetTextureMappingName (db, tmindex);
   switch (type = mgGetTextureMappingType (db, tmindex))
   {
   case 1:
      if (tmname && *tmname)
         printf ("Texture Mapping %d: %s: type = 3 Point Put\n",
                  tmindex, tmname);
      else
         printf ("Texture Mapping %d: (no name): type = 3 Point Put\n",
                  tmindex);
      break;
   case 2:
      if (tmname && *tmname)
         printf ("Texture Mapping %d: %s: type = 4 Point Put\n",
                  tmindex, tmname);
      else
         printf ("Texture Mapping %d: (no name): type = 4 Point Put\n",
                  tmindex);
      break;
   case 4:
      if (tmname && *tmname)
         printf ("Texture Mapping %d: %s: type = Spherical Project\n",
                  tmindex, tmname);
      else
         printf ("Texture Mapping %d: (no name): type = Spherical Project\n",
                  tmindex);
      break;
   case 5:
```

```
            if (tmname && *tmname)
               printf ("Texture Mapping %d: %s: type = Radial Project\n",
                        tmindex, tmname);
            else
               printf ("Texture Mapping %d: (no name): type = Radial Project\n",
                        tmindex);
            break;
         default:
            if (tmname && *tmname)
               printf ("Texture Mapping %d: %s: ERROR - Unknown Type\n",
                        tmindex, tmname);
            else
               printf ("Texture Mapping %d: (no name): ERROR - Unknown Type\n",
                        tmindex);
            break;
         }
      }
   }

   /* if rec is a vertex and it's parent polygon has a texture assigned to it */
   else if ((mgIsCode(rec, fltVertex)) && hasTexture)
   {
      /* Get the texture u,v coordinates */
      mgGetAttList (rec, fltVU, &u, fltVV, &v, MG_NULL);
      printf ("u, v = %f, %f\n", u, v);
   }

   return (MG_TRUE);   /* If a FALSE is returned the walk will terminate */
}

void main(int argc, char* argv[])
{
   mgrec* db;
   int   status=0, patindex=0;
   char  patname[1024];

   /* check for correct number of arguments */
   if (argc < 2) {
      printf ("Usage: %s <input_db_filename>\n", argv[0]);
      exit (EXIT_FAILURE);
   }

      /* Initialize the API */
   mgInit (&argc, argv);

      /* Load the database */
   if (!(db = mgOpenDb (argv[1])))
   {
      char msgbuf [1024];
      mgGetLastError (msgbuf, 1024);
      printf ("%s\n", msgbuf);
      mgExit ();
      exit (EXIT_FAILURE);
   }

      /* Check for textures in the palette */
   status = mgGetFirstTexture (db, &patindex, patname);
```

```
   if (status) {
      printf ("First pattern (%d) name is %s\n", patindex, patname);
         /* Walk the database */
      mgWalk (db, Action, MG_NULL, MG_NULL, MWALK_VERTEX);

         /* Close the database */
   }
   else
      printf ("No textures\n");

   mgCloseDb (db);

   mgExit ();
}
```

8

Utilities

The OpenFlight API contains many utility functions common to real time graphics programs. These utilities include:

- Dynamic Arrays and Stacks
- Geometry Functions
- Matrix Functions

Dynamic Arrays and Stacks

The OpenFlight API supports the creation and processing of dynamic arrays and stacks. Such data structures are useful when you do not know the total number of items you will need to store prior to creating the structure. When items are added, removed or replaced in a dynamic array or stack, the internal data structure used to represent the array or stack is "resized" automatically. Dynamic arrays and stacks are very similar. The main difference is how they are accessed. Dynamic arrays allow random access to elements in the array. Stacks, on the other hand, only allow access to the "top" item.

Each element in the dynamic array or stack is of type `void*`. In this way, you can build dynamic data structures of any element type you like by storing the address of your elements in the array. These arrays are also referred to as *pointer arrays* since they are arrays whose elements are *pointers*. Similarly, these stacks are also referred to as *pointer stacks*.

Following is a list of functions used to create, manipulate and destroy pointer arrays:

mgNewPtrArray	Allocates a new pointer array.
mgFreePtrArray	Deallocates a pointer array.
mgPtrArrayAppend	Appends an item onto the end of a pointer array.
mgPtrArrayInsert	Inserts an item into a pointer array at a specific position.
mgPtrArrayGet	Gets an item from a pointer array at a specific position.
mgPtrArrayRemove	Removes an item from a pointer array at a specific position.
mgPtrArrayReplace	Replaces an item in a pointer array with a different item.
mgPtrArraySort	Sorts the items in a pointer array.
mgPtrArrayLength	Returns the number of items in a pointer array.
mgPtrArraySearch	Searches a pointer array for an item using a linear search.
mgPtrArrayBSearch	Searches a pointer array for an item using a binary search.

Items in a dynamic array are counted starting at 1. Be careful, this means the first item in the array is at index 1, not 0. Similarly, the last item in the array is at index length, where length is the value returned by the function mgPtrArrayLength.

The memory allocated for the dynamic array itself is managed by the API. The memory allocated for the array items is managed by the caller. It is the caller's responsibility, therefore, to dispose of the memory allocated for each item when a dynamic array is freed. The following example shows this:

```
mgptrarray myArray;
int i, len;
int* item;

/* create a dynamic array */
myArray = mgNewPtrArray ();

/* add 10 dynamically allocated integers */
for (i=1; i<=10; i++) {
   item = mgMalloc (sizeof(int));
   *item = i;
   mgPtrArrayAppend (myArray, item);
}

/* length should be 10 */
len = mgPtrArrayLength (myArray);
printf ("Length of array : %d\n", len);

/* print out each item in array */
for (i=1; i<=len; i++) {
   /* note that first item is index 1, not 0 */
   item = (int*) mgPtrArrayGet (myArray, i);
   printf ("item %d : %d\n", i, *item);
}

/* free each item we allocated */
for (i=1; i<=len; i++) {
   item = (int*) mgPtrArrayGet (myArray, i);
   mgFree (item);
}

/* free array when we're done */
mgFreePtrArray (myArray);
```

Following is a list of functions used to create, manipulate and destroy pointer stacks:

mgNewPtrStack	Allocates a new pointer stack.
mgFreePtrStack	Deallocates a pointer stack.
mgPtrStackPush	Pushes an item onto the top of a pointer stack.
mgPtrStackPop	Pops an item off the top of a pointer stack.
mgPtrStackTop	Returns the top item of a pointer stack.
mgPtrStackClear	Pops all items off of a pointer stack.

mgPtrStackLength	Returns the number of items in a pointer stack.

Geometry Functions

The OpenFlight API provides several functions to perform simple geometric operations. Several of these functions are highlighted in this section. For a complete list of Geometry Functions, see the *OpenFlight API Reference*.

These functions are used to manipulate 3D coordinates (**mgcoord3d**):

mgMakeCoord3d	Makes a 3D coordinate from 3 x, y, z values.
mgCoord3dZero	Returns the 3D coordinates (0.0, 0.0, 0.0)
mgCoord3dXAxis	Returns the 3D coordinates (1.0, 0.0, 0.0)
mgCoord3dYAxis	Returns the 3D coordinates (0.0, 1.0, 0.0)
mgCoord3dZAxis	Returns the 3D coordinates (0.0, 0.0, 1.0)
mgCoord3dNegativeXAxis	Returns the 3D coordinates (-1.0, 0.0, 0.0)
mgCoord3dNegativeYAxis	Returns the 3D coordinates (0.0, -1.0, 0.0)
mgCoord3dNegativeZAxis	Returns the 3D coordinates (0.0, 0.0, -1.0)
mgCoord3dAdd	Calculates the sum of two 3D coordinates.
mgCoord3dSubtract	Calculates the difference of two 3D coordinates.
mgCoord3dMultiply	Calculates the product of 3D coordinates and a scalar.
mgCoord3dDivide	Calculates the quotient of 3D coordinates and a scalar.

mgCoord3dCross	Calculates the cross product of two 3D coordinates.
mgCoord3dDot	Calculates the dot product of two 3D coordinates.
mgCoord3dLength	Calculates the length of 3D coordinates.
mgCoord3dLengthSquared	Calculates the squared length of 3D coordinates.
mgCoord3dMoveAlongVectord	Calculates the position of a 3D coordinate translated along a double precision vector a given distance.
mgCoord3dMoveAlongVectorf	Calculates the position of a 3D coordinate translated along a single precision vector a given distance.
mgCoord3dTransform	Transforms 3D coordinates using a specified matrix.
mgCoord3dEqual	Checks if two 3D coordinates are exactly.
mgCoord3dAlmostEqual	Checks if two 3D coordinates are equal within a tolerance.
mgCoord3dDistance	Calculates the distance between two 3D coordinates.
mgVectordToCoord3d	Converts a 3D vector to 3D coordinates.

Many of the functions listed above for 3D coordinates exist for 2D coordinates (**mgcoord2d**) as well. See the *OpenFlight API Reference* for more information on 2D coordinate functions.

These functions are used to manipulate 3D vectors (**mgvectord**):

mgMakeVectord	Makes a 3D vector between two 3D coordinates (not unitized).

mgMakeUnitVectord	Makes a unitized 3D vector between two 3D coordinates.
mgVectordFromLine	Makes a 3D vector from a line (not unitized).
mgVectordUnitize	Unitizes a 3D vector.
mgVectordCross	Calculates the cross product of two 3D vectors.
mgVectordDot	Calculates the dot product of two 3D vectors.
mgVectordFromLine	Makes a 3D vector from a line.
mgVectordXAxis	Returns the 3D vector (1.0, 0.0, 0.0)
mgVectordYAxis	Returns the 3D vector (0.0, 1.0, 0.0)
mgVectordZAxis	Returns the 3D vector (0.0, 0.0, 1.0)
mgVectordNegativeXAxis	Returns the 3D vector (-1.0, 0.0, 0.0)
mgVectordNegativeYAxis	Returns the 3D vector (0.0, -1.0, 0.0)
mgVectordNegativeZAxis	Returns the 3D vector (0.0, 0.0, -1.0)
mgVectordTransform	Transforms a 3D vector using a specified matrix.
mgCoord3dToVectord	Converts 3D coordinates to a 3D vector.
mgVectorfToVectord	Converts a single precision 3D vector to double precision.

These functions are used to manipulate 3D boxes (**mgboxd**):

mgMakeBox	Makes a box given two (min and max) 3D coordinates.
mgGetBounds	Returns the bounding box for a node record.

mgGetBoundsForRecList	Returns the bounding box for a list of nodes.
mgBoxExpandCoord3d	Expands a box so that it contans a specified 3D coordinate.
mgBoxExpandBox	Expands a box so that it contans another box.
mgBoxGetSizeX	Returns the X dimension of a box.
mgBoxGetSizeY	Returns the Y dimension of a box.
mgBoxGetSizeZ	Returns the Z dimension of a box.
mgBoxGetCenter	Returns the center of a box.
mgBoxGetCenterBottom	Returns the center bottom of a box.
mgBoxContainsCoord3d	Determines if a 3D coordinate is contained inside a box.
mgBoxContainsBox	Determines if a box is contained inside another box.
mgBoxIntersectsBox	Determines if a box intersects another box.

These functions provide miscellaneous utility for node records:

mgReversePoly	Reverses the order of the vertices in a polygon. This effectively "flips" the polygon normal, making it "face" the opposite way.
mgIsPolyCoplanar	Determines if the vertices of a polygon are all coplanar.
mgIsPolyConcave	Determines if a polygon is concave.

Matrix Functions

The OpenFlight API provides a suite of functions to perform common matrix operations. It also provides functions to create and manipulate matrix stacks. For a complete list of these functions, see Matrix Functions and Matrix Stack Functions in the *OpenFlight API Reference*.

Glossary

alpha

The transparency value of a polygon, mesh, vertex or *texel*. Valid alpha values range from 0 (completely transparent) to 255 (completely opaque).

API

The Presagis OpenFlight **A**pplication **P**rogramming **I**nterface, which is a set of C header files and libraries that provides a programming interface to Creator and the OpenFlight database format.

API level

One of four levels of the Presagis OpenFlight API. Level 1: *Read*, gives you access to existing OpenFlight files. Level 2: *Write*, creates OpenFlight node hierarchies and saves them in OpenFlight format. Level 3: *Extensions*, extends the kind of data represented in Creator and the OpenFlight format. These extensions then are treated as native data by Creator. Level 4: *Tools* allows you to define plug-ins to Creator. This release of the API includes Levels 1 and 2 (Read and Write) plus Levels 3 and 4 (Extensions and Tools).

attributes

Defining properties of a record. Database attributes include flags, and numeric and text data such as color, relative priority, and LOD switching distances.

attribute record

A record containing descriptive data (flags, numeric values, and text), which is associated with a node in the database hierarchy. Attribute records are accessed through record codes.

attribute value

The numeric value or setting of an attribute.

child node

A node attached below another node in the database tree. The child node inherits transformations and other properties from its parent node.

code

See *record code*.

color palette

The indexed set of available colors.

concave polygon

A polygon with two or more edges that form an inward indentation. If a line can be drawn between any two vertices of the polygon such that the line crosses outside the polygon, the polygon is concave.

current database

The OpenFlight database currently selected for editing.

database

The geometry and hierarchy that defines all models in an OpenFlight file.

database header

The attributes that apply to the entire database.

database header node

The top or root node of the database hierarchy. The database node is identified by the *fltHeader* record code.

data dictionary

A list of all the record types in an OpenFlight file. Using the Extensions API, you can define your own custom data dictionary.

default color palette

The color palette that is not associated with a particular database. It is used when the color palette is accessed but there is no current database.

DLL

Dynamic **L**ink **L**ibrary.

entry

One element of a palette. An entry is accessed in the palette either by its index or its name.

extension record

Node or node attribute records that are defined by an API user using the *Extensions API*.

external reference

A reference to another database. External referencing allows you to include the contents of one database in another database without pasting in the geometry, thereby saving disk space and streamlining the assembly of complicated databases. External references are *read only*. You can position, orient, and scale an external reference, but you cannot edit its contents without opening the external database file itself.

group

A node type used to organize other nodes in the hierarchy.

image

A two-dimensional matrix of texels.

image attributes

Parameters that describe an image, including its type, height, and width.

index (indices)

An integer value used to identify an entry in a palette.

instance node

A node that references a portion of the geometry in the database. The referenced geometry is called a *reference node*.

instancing

The ability to describe a group or object once, then display it one or more times with various transformations. OpenFlight supports instancing of objects, groups, and group-like nodes, using transformations such as rotate, translate, scale, and put.

level of detail

One of a set of models that represent the same item in the database at varying levels of complexity. When the eyepoint is far away from the item, a simple (low-resolution) level of detail is displayed. As the eyepoint approaches the item, lower resolution models switch out and increasingly complex (higher resolution) models switch in. This process is called *LOD switching*.

LOD

Level **O**f **D**etail

LOD switching

The process of changing from one level of detail to another as the eyepoint changes. See *LOD,* above.

light point

A visible, geometric representation of a light. Light points do not emit light nor do they affect the appearance of other geometry in the database.

light source

A database element that emits light. It is used to calculate the shading on the geometry in the database. OpenFlight supports two kinds of light sources: modeling lights, which are always turned on even if they are not part of the hierarchy; and light source nodes, which are hierarchical references to entries

in the light source palette. While light sources affect the appearance of the geometry in the database, they are not themselves visible.

light source palette

The indexed set of available light sources.

material

A surface property that simulates the light-reflecting characteristics of substances such as wood, plastic, or metal.

material palette

The indexed set of available materials.

nested node

A node representing a polygon attached to a coplanar polygon in the hierarchy. A nested child and nested parent are equivalent to Creator's subface and superface nodes, respectively. Nested nodes provide a useful means of specifying drawing order on z-buffered systems, because in z-buffered drawing mode, every nested child draws on top of its nested parent.

nested attributes

Attributes that are hierarchical within a single node. For example, a color attribute is composed of red, green, and blue attributes.

node

A generic term for any of the records that form the hierarchy of the OpenFlight database.

object

A node that has one or more polygons or meshes as its children.

orphan

A node that has no attach point in the hierarchy. Orphans are not saved with the database.

palette

An organized, indexed set of available properties, such as colors, materials, texture images, and light sources, that can be applied to certain elements of the database. Every entry in the palette is stored as a record, which can be accessed through the entry's index or name. A database usually contains one of each type of palette.

palette index

An integer that identifies an entry in a palette.

parent node

The attach point of any given node in the database.

pixel

The smallest unit of an image on a display screen.

polygon

A multi-sided, closed or unclosed face consisting of edges that connect vertices.

record

An organized collection of properties, attributes, and/or links that fully describes an element of an OpenFlight database and potentially its position in the database. There are three types of records: attribute records, palette records, and node records. Records can contain attributes that are other records, and attributes that are simple values.

record code

A unique integer that identifies each record type.

reference

A link to a shared node (reference node).

reference node

A node shared by more than one parent node. Each time a node is shared, it is said to be *instanced*.

replication

A technique for drawing a node more than once, adding a transformation each time the node is drawn. The *transformation matrix* is applied to the node once to draw the first replication, applied again to draw the second replication, and so on.

RGB

An image type in which colors are defined by red, green, and blue values.

RGBA

An image type in which colors are defined by red, green, blue, and alpha values.

shared node

See *reference node*.

sibling node

One of at least two child nodes attached to the same parent node in the database.

texel

A ***tex***ture ***el***ement, the basic unit of a texture. A texel can include a color value, a transparency (alpha) value, or both. When no magnification or minification filter is used, a texel is the same as a pixel.

texture

An image used to add the appearance of complexity to a polygon's surface.

texture attributes

Parameters that describe a texture and the way it is mapped to geometry.

texture palette

The indexed set of available textures. Each OpenFlight database has its own texture palette. A texture must be listed in the texture palette before it can be used by a database.

transformation

An operation that positions one or more nodes in the database without modifying their vertex coordinates. The transformation is applied to all the descendants of the node containing the transformation.

transformation matrix

An array of values that, when applied, position one or more nodes in the database without modifying their vertex coordinates. Transformation matrices are useful with instances and external references where you want to position a single object in several places, and in functions that reposition database objects and groups, such as translate and scale. You can apply a transformation matrix to any node type except *fltVertex*.

traversal

The act of walking through a database, visiting nodes in an order you define.

vertex (vertices)

One of the coordinate points that defines a polygon or mesh.

Copyright

February 4, 2016

PRESAGIS