OPENFLIGHT API 15

DEVELOPER GUIDE, VOLUME 2

LEVELS 3 AND 4 : EXTENSIONS/TOOLS

CONTENT CREATION

PRESAGIS

Contents

OpenFlight API Developer Guide Volume 2

Welcome to the Developer Guide, Volume 2 for OpenFlight API 15.

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 2 of the *OpenFlight API Developer Guide* and describes levels 3 and 4, the Extensions/Tools APIs. Please refer to the *OpenFlight API Reference* for supplementary information on the API.

To use levels 3 and 4 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 Extensions/Tools API

The **Extensions API** lets you define your own constructs in OpenFlight. You can extend the definition of existing OpenFlight node and palette types as well as create new node types specific to your application. Your custom data will be saved in a valid OpenFlight database file that you can give to others.

The **Tools API** lets you extend the functionality of the Presagis modeling products, including Creator, without having to modify the main program code. Within Creator, the extended functionality will appear to the user as supplemental capabilities.

The constructs you define using the **Extensions API** are referred to as *Data Extensions,* while the capabilities you define using the **Tools API** are referred to as *Tools*.

There are two types of Data Extensions you can define:

- *Tag-along:* Attributes to extend an existing node type (e.g., a `fltGroup` node).
- *Stand-alone:* New node types with their own attributes.

There are six types of Tools you can define:

- *Database Importer:* tools to read database files from unsupported file formats into Creator.
- *Database Exporter:* tools to write an existing database or portions of an existing database to disk formats not directly supported by Creator.
- *Image Importer:* tools to read image/texture files from unsupported file formats into Creator.
- *Viewer:* tools to passively interact with a database. Viewers can be used to implement any action that does not modify the contents of the database.
- *Editor:* tools to modify the contents of an existing database.
- *Input Device:* tools to convert foreign input device input to that which is understandable by Editor tools in Creator.

These data extension and tool types are described in detail in subsequent chapters of this guide.

Program Environments

In general, your goal as a developer using the **Extensions/Tools API**, is to build data extensions and/or tools that can be accessed by end users, including both modelers and programmers. The **Extensions/Tools API** is the mechanism by which these end users gain access to the components you develop.

The components you develop will be accessed by the end user in two different program (runtime) environments. The first is the Creator modeling system. The second is the stand-alone program developed using the OpenFlight API.

Using the Extensions/Tools API, you will create separately compiled library modules that *export* the data extensions and/or tools you develop. At runtime, these modules *plug in* to the program environments described above, thereby making the data extensions and/or tools they contain available. The library modules you develop using the Extensions/Tools API are referred to as *plug-in modules,* or more simply, *plug-ins*. Data extensions and tools that are contained in *plug-ins* are referred to, collectively, as *plug-in objects*.

Data extensions defined in your plug-in can be accessed within both of the program environments. In the Creator environment, end users access your constructs just as they access any other OpenFlight data construct. Extension attributes are viewed in *Attribute Pages* and new node types you define are created, selected and manipulated just like other existing OpenFlight node types. In the stand-alone program environment, end users use the Read/Write API to access your constructs in the same way OpenFlight constructs are accessed.

Unlike data extensions that are accessible in both program environments, tools you define in your plug-in can only be accessed by the end user within Creator. This is because tools require runtime components only available within Creator, such as Graphical User Interface components and other model time components that are not provided in the stand-alone program environment.

The mechanisms for loading a plug-in into these two different program environments are identical in terms of how you write your plug-in module. For this reason, these two runtime environments will be referred to, collectively, as the *Plug-in Runtime Environment.*

Creating and Using a Plug-in

This guide describes how to develop a plug-in module and integrate it into the plug-in runtime environment. It shows how to create both data extensions and tools within your plug-in and provides an overview of the Extensions/Tools API functions that help you in doing so.

The following is an outline of the process for creating a plug-in:

1. Obtain a **U**niversally **U**nique **Id**entifier (**UUID**) for your plug-in module. **UUID's** are character strings that provide a means of unique identification for your module. Utilities exist on different computer platforms to generate these identifiers.
2. Create **C** language text file(s) containing source code for the plug-in module. For an overview of your **C** source code, see "Plug-in Source Code Overview" on page 19. You may modify a sample file, tailoring it to your specific needs, or you may create a new file.
3. (Optional) If any of the tools you are creating require Graphical User Interfaces (GUIs), create platform specific resource files containing GUI dialog layout and other resources required by your tool. See "GUI" on page 137.
4. Compile and link your plug-in module that contains your plug-in objects. (See "Building a Plug-in Module" on page 253)
5. Place your module in the plug-in runtime directory so that it will load when the plug-in runtime environment starts. (See "Plug-ins in the Runtime Environment" on page 259)
6. Run the plug-in runtime environment (Creator or stand-alone program) to test and debug the plug-in objects defined in your plug-in.

Plug-in Source Code Overview

Plug-ins are library modules, native to a specific operating system (Windows or Linux), on which the plug-in runtime environment (Creator and/or stand-alone program) runs. The *Extensions/Tools API* is designed to provide the maximum flexibility while being functionally equivalent for both platforms.

Each plug-in module can define one or more plug-in objects. Remember, a plug-in object is either a data extension or a tool. Since a plug-in module can contain one or more plug-in objects, the developer can decide how objects are packaged within modules. To achieve maximum modularity, each plug-in object could be packaged in its own module. Alternatively, an entire suite of similar objects could be packaged in a single module to allow each object to share common code and thereby save code space.

This chapter describes a high-level framework you will use to develop your plug-in module. Subsequent chapters provide the details needed to build the specific plug-in objects you want to define.

Symbols You Define

When the plug-in runtime environment initializes, it searches the *Plug-in Runtime Directory* for plug-in modules to load. The location of this directory is described in "Plug-ins in the Runtime Environment" on page 259. When the runtime environment finds a module in this directory, it further verifies that the module is a compatible plug-in by inspecting the module's symbol table. Certain pre-determined symbols are expected to be exported by your plug-in module so that it can be recognized and loaded by the plug-in runtime environment.

This section describes the symbols your plug-in module must declare. The API provides macros to use when declaring these symbols to ensure the symbols are exported properly on both operating system platforms. You are encouraged to use these macros so your source code is ported more easily between platforms.

Important: These symbols must have the names described in this section. If you do not use these names, your plug-in will not be recognized by the plug-in runtime environment and will not be loaded properly.

To identify your plug-in module, you must declare **Vendor, Name, UUID,** and **API Compatibility Version** symbols. A macro is provided by the API for you to use to declare these symbols. It is strongly recommended that you use this macro rather than declaring the symbols manually. For this reason, the symbols themselves are not described here, only the macro.

mgDeclarePlugin(vendor,name,uuid)

`vendor` - is a character string that represents the vendor or company name that developed the plug-in module. There is no limit on the length of this string.

`name` - is a character string that represents the name of the plug-in module. There is no limit on the length of this string.

`uuid` - is a character string that is the Universally Unique Identifier of your plug-in module. On Windows, use the **`uuidgen`** utility to generate a UUID.

Note: The **`uuid`** for each plug-in module must be unique. The plug-in runtime environment cannot load two plug-in modules with the same **`uuid.`** If you copy/paste one of the sample plug-ins, make sure that you generate a new **`uuid`** for your plug-in.

The **API Compatibility Version** symbol, also defined by this macro is built into the macro and gets created automatically when you *compile* your module. The plug-in runtime environment uses this symbol internally to determine whether your module is compatible with the current version of the API libraries used by the plug-in runtime environment. You should not try to manually define this symbol. Instead, let the macro do it for you.

You must instantiate this macro at the outermost scope of your source code as follows:

```
#include "mgapiall.h"

mgDeclarePlugin (
      "Plugin Vendor",
      "Plugin Name",
      "Plugin UUID"
```

```
    );
```

mgpInit

This is the plug-in initialization function. It is required and is called by the plug-in runtime environment at start-up. In this function, you are responsible for initializing your plug-in module. Within this function, you will declare the data extensions and/or tools contained in your plug-in module. You can also do any other initialization your module requires here.

This function returns a boolean value indicating whether or not the plug-in module was successfully initialized. If your function returns **`MG_TRUE`**, the plug-in module will remain loaded for the duration of the modeling session. Otherwise it will be unloaded.

You can also use this return value to control whether your plug-in is to be loaded and, therefore, be available to the user. You might do this, for example, if you develop a commercial plug-in and only want it to be accessible to users who have purchased it. In this case, your plug-in initialization function might check for the existence of a license and return **`MG_TRUE`** only if the license was found and is valid. See "Licensing a Plug-in Module" on page 247 for more information.

You must declare this function using the **`MGPIDECLARE`** macro at the outer most scope of your source code as follows:

```
MGPIDECLARE(mgbool)
mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgbool initOk;
   // register plug-in objects and do other initialization
   initOk = InitializeMyPlugin ( plugin );
   return initOk;
}
```

`plugin` - is an opaque object that uniquely identifies your plug-in module. You will need to pass this to certain API functions that require a plug-in module identifier.

`argc` - is a pointer to an integer value that is the number of arguments in **`argv`** that follow.

`argv` - is a pointer to an array of strings that are the command line arguments passed to the program (Creator or stand-alone program) in which the plug-in is being loaded.

mgpExit

This is the plug-in termination function. This function is required and is called by the plug-in runtime environment at termination. Within this function, you are responsible for cleaning up after your module. For example, if your plug-in allocated any memory, you can free it here.

You must declare this function using the **`MGPIDECLARE`** macro at the outermost scope of your source code as follows:

```
MGPIDECLARE(void)
mgpExit ( mgplugin plugin )
{
   // unregister plug-in objects and do other exit processing
}
```

`plugin` - is an opaque object that uniquely identifies your plug-in module.

API Header Files

There are many 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 plug-in requires, it is recommended that you simply include **`mgapiall.h`**.

When you define and/or access a data extension, you will also need to include the header files that are auto-generated by the **`ddbuild`** utility. These include files are described in later chapters.

Additionally, if your plug-in module uses a resource file to define GUI elements, you will also need to include your *resource header file*. Resource header files are discussed in "Resource Files" on page 138.

Variable Argument Style

Many functions in the Extensions/Tools API use *variable argument style* parameters. These functions are recognizable by the ellipses **`(...)`** used to denote the last parameter of the parameter list. When you see this style parameter list in a function declaration, is means that a list of optional parameters may be passed in place of the ellipses. When calling variable argument style functions in the API, you must always remember to terminate the parameter list with the **`MG_NULL`** symbol. This is required to properly terminate the parameter list you are passing to the function.

Data Extensions Overview

OpenFlight is a rich file format that contains information to describe your visual scene. In addition to the data supported directly by the OpenFlight file format, the OpenFlight API allows you to add your custom data to the OpenFlight file. You can extend the definition of existing OpenFlight node and palette types as well as create new node types specific to your application. Your custom data will be saved in a valid OpenFlight database file that you can give to others.

There are two very different methods for creating and maintaining Data Extensions in OpenFlight:

- **Easy OpenFlight Extensions** - With this method, you use very simple API functions to create, examine and modify your extension data. See "Easy OpenFlight Extensions" on page 25 for more information.
- **Extension Plugins** - With this method, you define a formal data dictionary and create an extension plug-in that manages your extension data. See "Extension Plugins" on page 28 for more information.

Easy OpenFlight Extensions

The API provides simple functions to create, examine and modify your *Easy OpenFlight Extension* data. An overview of the steps are provided in this section.

Before you can put extension data on a node in your OpenFlight database, you must first describe the data to the API. This will allow the API to recognize and interact with your data correctly. Extension data is grouped by "extension sites". An extension "site" is typically an organization or some other entity that "owns" and defines the extension data. Within an extension site is a list of "extension fields". An extension "field" is the data itself. This is the data that will be attached to nodes in your scene. For each extension field, you will specify the OpenFlight node type to which the data can be attached (group, object, polygon, etc) as well as its data type. You can choose from the following data types:

- **Boolean** (true or false)
- **Integer** (a whole number)
- **Float** (single precision real number)
- **Double** (double precision real number)
- **Unformatted String** (variable length ASCII character sequence)
- **XML String** (variable length ASCII character sequence interpreted as XML format)

Note: Both Extension Sites and Fields have **U**niversally **U**nique **Id**entifiers (**UUID**) assigned to them. These UUIDs help the API differentiate between your data and extension data defined by others. On Windows, use the `uuidgen` utility to generate these UUIDs when needed.

To Create your Easy Extension Site

1. Create an Extension Site using `mgExtensionSiteAdd`. As mentioned above, this site will contain all the extension fields you define. You will provide a new UUID for your site.
2. Use `mgExtensionSiteSetName` to give your site a name. This is typically the name of your organization but can be any text string you want to use to identify your site.

To Create an Easy Extension Field

1. Create an Extension Field using `mgExtensionFieldAdd`. You will provide the UUID assigned to your extension site and a new UUID for your extension field.
2. Use `mgExtensionFieldSetName` to give your field a name. This is typically descriptive of your data and can be any text string you want to use to identify your field.
3. Use `mgExtensionFieldSetAttach` to define the kind of OpenFlight node to which you want your data attached. For example, if you want to add your extension field to polygon nodes in the OpenFlight scene, specify `fltPolygon.`
4. Use `mgExtensionFieldSetType` to define the data type for your extension field. The data types you can use are listed in above in "Easy OpenFlight Extensions" on page 25.

To Modify Extension Data in the OpenFlight Scene

1. To attach or modify existing extension data to a node in the OpenFlight scene, use one of the following functions according to the data type of the extension field:
   `mgExtensionFieldSetInteger` - assign an integer value.
   `mgExtensionFieldSetFloat` - assign a single precision floating point value.
   `mgExtensionFieldSetDouble` - assign a double precision floating point value.
   `mgExtensionFieldSetBool` - assign a boolean value.
   `mgExtensionFieldSetString` - assign an ASCII string value.
   `mgExtensionFieldSetXMLString` - assign an XML string value.

   For each of these functions, you will specify the node to which you want the extension attached and the UUID of the extension field you want to attach. If the extension field is not yet attached to the node, these functions will create and attach the new extension field. If the extension field is already attached to the node, these functions will modify the value of the data.

2. To read extension data from a node in the OpenFlight scene, use one of the following functions according to the data type of the extension field:
   `mgExtensionFieldGetInteger` - retrieves an integer value.
   `mgExtensionFieldGetFloat` - retrieve a single precision floating point value.
   `mgExtensionFieldGetDouble` - retrieve a double precision floating point value.
   `mgExtensionFieldGetBool` - retrieve a boolean value.
   `mgExtensionFieldGetString` - retrieve an ASCII string value.
   `mgExtensionFieldGetXMLString` - retrieve an XML string value.

   For each of these functions, you will specify the node from which you want to retrieve extension data and the UUID of the extension field you want to retrieve.

3. To remove extension data from a node in the OpenFlight scene, use **`mgExtensionFieldDelete`**. You will specify the node from which you want to delete the extension data and the UUID of the extension field you want to delete.

4. To determine if extension data is attached to a node in the OpenFlight scene, use **`mgExtensionFieldDefined.`** You will specify the node and UUID of the extension field you want to query.

Extension Plugins

There are two types of Data Extensions you can define in your plug-in:

- *Tag-along*: attributes to extend an existing node type (e.g., a `fltGroup` node).
- *Stand-alone*: new node types with their own attributes.

The information that describes your extensions to the database is called the *schema*. You define your extensions with a *data dictionary*, which is a physical representation of your schema.

Chapters 3 - 6 describe how to create a data extension. An overview of the process is as follows:

1. Define a Site ID that you will use to identify your data extension. (See "Your Site ID" on page 28)
2. Define your extensions in an ASCII *data dictionary* file. (See "Creating a Data Dictionary" on page 31)
3. Run the `ddbuild` parser on your data dictionary file to create the header files that define your data extension. (See "Running the ddbuild Parser" on page 201). If you plan to use your extension data in a stand-alone OpenFlight Script, you will also use the `ddbuild` parser to create the Python module you will need to import in your script. This module contains the definitions of your extension fields and records.
4. Write C code that declares your data extension using the header files generated by the `ddbuild` parser. (See "Writing a Data Extension" on page 39)
5. Compile and link your plug-in module that contains your data extension. (See "Building a Plug-in Module" on page 253)
6. Place your plug-in module in the plug-in runtime directory so that it loads when the plug-in runtime environment starts. (See "Plug-ins in the Runtime Environment" on page 259)

Your Site ID

To identify your extensions within an OpenFlight file, the API tags them with an identifier, or *site ID*, that is unique to you or your organization. In general, each organization needs only one site ID. However, there is no restriction on the number of site IDs that you can have. If you are developing two or more significantly different real-time applications, you may want to keep your groups of extensions under separate site IDs.

Site IDs are text strings of up to 7 characters (8 including a null terminating character). They may contain any special characters (e.g., “&” or “-”) but may not contain white space characters (e.g., space, tab, etc.). Your site ID provides the means for “handshaking” between your extensions library and OpenFlight files. If Creator finds a library to match the site ID stored in an OpenFlight file, it tries to process those records. Otherwise it treats them as unknown records.

Use the site ID in your library to tell Creator and the API how to identify your records.

Creator and the API also use the site ID. For example:

- The API tags every extension record (nodes and node attributes) in the OpenFlight file with your site ID to identify which data extension is required to interpret the record properly.
- Creator checks the site ID of the data extension you register in your plug-in when it is loaded, to determine which OpenFlight records it can interpret.

You can define your own site *prefix*, to use in naming your data dictionary and header files. For example, the Acme software company might choose a site ID of *ACMESOFT* and a site prefix called “acme”. They would name their data dictionary **`acme.dd`**, and their header files **`acmecode.h`** and **`acmetable_.h`**. Your site prefix is not controlled in any way; it is for your own convenience.

Obtaining a Site ID

Because many people share OpenFlight files between organizations, it is important that each data extension contained in any OpenFlight file be uniquely identifiable via its site ID. To help ensure that the site IDs of each data extension is unique, Presagis maintains a list of site IDs in use. You should register your site ID with Presagis to keep your extensions from becoming confused with those of other organizations. Note that you are only logging an 7-character identifier string, not your extensions themselves. The contents of your data extensions remain private until you distribute your plug-in library file.

Note: If you do not plan on distributing your data extension or OpenFlight databases containing your data extensions, and you never plan on reading OpenFlight database files that contain other organization's data extensions, your site ID need not be registered with Presagis.

How Many New Definitions Can I Create?

You can create up to 4096 unique record and field types. The following section, "Creating a Data Dictionary" on page 31, describes how to define records and fields.

Creating a Data Dictionary

The data dictionary is an ASCII file in which to define nodes and/or node attributes. To begin creating a data dictionary, open a new file in any text editor. Save your data dictionary with the name **`<siteprefix>.dd`**. Following this convention prevents naming conflicts with other data extensions.

There are two basic types of elements in the data dictionary: *fields*, which represent single values or strings, and *records*, which are groups of fields.

There are seven keywords that you use to define entries in the data dictionary: **`dataDef, recordDef, struct, describe, parent, child`**, and **`alias`**. This chapter describes how to use these keywords to define your data dictionary. This chapter also discusses guidelines to follow to ensure backward compatibility when modifying a previous version of your data dictionary to create a new version.

Defining Fields and Records

To define fields and records, you use the keywords **`dataDef, recordDef`**, and **`struct.`**

Use **`dataDef`** to define a new field. The following example defines three simple integer fields called **`myInt1, myInt2`**, and **`myInt3`**.

```
dataDef myInt1 { MTYPE_INT }
dataDef myInt2 { MTYPE_INT }
dataDef myInt3 { MTYPE_INT, LABEL="My Second Integer" }
```

Use **`recordDef`** to define a new record. The following example defines a record type called **`myInts,`** which extends the node and will appear in a pane labeled "My Integers" in the attribute page.

```
recordDef myInts { MTYPE_REC, myInts,
          LABEL="My Integers",
          XCODE=
}
```

Use **`struct`** to associate fields with a record type. The following example defines a record that contains all three integer fields and attaches it as an extension attribute to the node type.

```
struct myInts {
          myInt1;
          myInt2;
          myInt3;
}
```

`dataDef` and **`recordDef`** entries can have a variable-length argument list. The first parameter between the braces must always be the data type **(`MTYPE_INT`, `MTYPE_DOUBLE`**, and so forth). See "Data Types" on page 271 for a list of valid data types.

For a **`recordDef`**, the data type must always be an **`MTYPE_BEAD`** (to specify a new node type) or an **`MTYPE_REC`** (for all other record types). **`recordDef`** entries also require a second parameter - the name of the **`struct`** definition that lists the associated fields.

`dataDef` and **`recordDef`** entries can have any number of options. The example above used two: **`LABEL`**, which defines a text string for the Creator attributes pages and **Extensions menu** to display, and **`XCODE`**, which defines which OpenFlight record you want to extend. There is a complete list of supported options in "Data Dictionary Keywords" on page 265.

Defining a New Node Type

```
/******************** Create my own node type *******************/
/******* The node will have an inline rec and a ptr rec ********/
dataDef myInlineInt { MTYPE_INT }
dataDef myInlineFloat { MTYPE_FLOAT }
dataDef myInlineText { MTYPE_TEXT }
recordDef myInlineRec { MTYPE_REC, myInlineRec}
struct myInlineRec {
          myInlineInt;
          myInlineFloat;
          myInlineText;
}

dataDef myPtrRecInt { MTYPE_INT }
dataDef myPtrRecFloat { MTYPE_FLOAT }
dataDef myPtrRecText { MTYPE_TEXT }
recordDef myPtrRec { MTYPE_REC, myPtrRec}
struct myPtrRec {
          myPtrRecInt;
          myPtrRecFloat;
          myPtrRecText;
}

dataDef myInt { MTYPE_INT }
dataDef myFloat { MTYPE_FLOAT }
dataDef myText { MTYPE_TEXT, LEN=16 } //important: need to specify text LEN
recordDef myNodeType { MTYPE_BEAD, myNodeType,
          LABEL=”Extension Node”,
          PREFIX=my
}
struct myNodeType {
          myInt;
          myFloat;
          myText;
          myInlineRec;
          myPtrRec, PTR;
}
```

Making Your Data Dictionary User-Friendly

To make your data dictionary more understandable to others, you can add explanatory text.

- You can include C or C++ style comments in your data dictionary. The **`ddbuild`** parser will ignore them.
  `/* comment */`
  `// comment`
- You can also use **`describe`** to define text to describe any field, then use **`ddbuild`** to auto-generate a document from your data dictionary. (“Running the ddbuild Parser” on page 201 describes how to use **`ddbuild`**.) The resulting document will have a list of all valid fields and records in the data dictionary (showing the hierarchy), and it will be annotated with any text you define with **`describe`**.

The following example attaches some text to explain the function of the integer fields and the record we defined earlier. There isn’t a **`describe`** line for **`myInt2`**, but it will still appear in the document; it just won’t have any notes after it:

```
describe myInt1 { some text that explains myInt1 }
describe myInt3 { some text that explains myInt3
          It has several lines
          because it is a very interesting field.
}
describe myInts { A record with several integer fields }
```

You cannot insert comments inside a **`describe`** entry; the comment would be treated as part of the field or record’s description.

Attachment Rules

Use **`parent`** and **`child`** to define rules for attachment between nodes.

By default, a new node type you define has no attachment restrictions. Any node type can be the new node's child, and any node type can be its parent. You can change this behavior by defining lists of valid children and/or parent types. The following example defines an attachment list where only a **`fltGroup`** can be a parent and only **`fltObject`** or nodes can be children:

```
// Only have group as a parent
parent myNode_Parent {
   fltGroup;
}

// Only have Object and Polygon as children
child myNode_Child {
   fltObject;
   ;
}
```

This example simply *defines* attachment rules in a generic sense; it doesn't *apply* them to any node types. The following example changes the definition of myNodeType on page 32 to use these attachment rules:

```
recordDef myNodeType { MTYPE_BEAD, myNodeType,
          LABEL="Extension Node",
          PREFIX=my,
          PARENT=myNode_Parent,
          CHILD=myNode_Child
}
```

Aliases: Defining a Generic Pointer

An **`alias`** lets you hide the definition of a data structure and simply use a dummy pointer, usually declared as a **`(void*)`**. The alias list provides a cross reference between the record you want to cast, and the list of valid data types to cast to.

The following example defines a record for a dummy pointer to store transformation data. At runtime, the dummy pointer could be cast to any of the types listed in the alias declaration, depending on the transformation type. Alias definitions by themselves do not have a way of identifying which of the record types in the alias list to use. You must have a way of determining this for yourself, for example by checking the value of an integer field:

```
recordDef fltXformPtr { MTYPE_REC, fltRecDummy, ALIAS=Alias_fltXformPtr }

alias Alias_fltXformPtr {
          fltXmTranslate;
          fltXmScale;
          fltXmRotate;
          fltXmPut;
          fltXmGeneral;
          fltXmRotateEdge;
          fltXmScaleToPoint;
}
```

Enumeration Types

Use enumeration types when you want to select one of a group of options. Since each of the options only has two values (on and off) use fields of type **`MTYPE_FLAG`** and tag them with the ENUM option. The following example creates a record with five different enumerated values:

```
/******************** Special Flags for Polygon ****************/
dataDef Monday { MTYPE_FLAG, LABEL="Monday", DEF=0, ENUM }
dataDef Tuesday { MTYPE_FLAG, LABEL="Tuesday", DEF=0, ENUM }
dataDef Wednesday { MTYPE_FLAG, LABEL="Wednesday", DEF=0, ENUM }
dataDef Thursday { MTYPE_FLAG, LABEL="Thursday", DEF=1, ENUM }
dataDef Friday { MTYPE_FLAG, LABEL="Friday", DEF=0, ENUM }

recordDef Days { MTYPE_REC, DayList,
          LABEL="Days of the week",
          XCODE=
}
struct DayList {
          Monday;
          Tuesday;
          Wednesday;
          Thursday;
          Friday;
          xxxpad, PAD=5;
}
```

The **`DEF`** option specifies the default value of each flag. Note that since, by definition, only one enumerated value can be selected at a time, all of the defaults must be 0 except one **`(Thursday)`**.

Padding

The Extensions API supports record padding in the definition of your file format. *Padding* lets you reserve space by inserting unused bytes into a

struct definition. By replacing your previously defined padding with real fields, you can add values to the list later, without affecting the layout of the record. In previous versions of the API, this was required if you ever wanted to add fields to the end of your record in subsequent versions of your data extension. This is no longer the case. See "Version Updates - Backward Compatibility" on page 36 for more information on how to preserve backward compatibility between versions of your data extension.

Even though you are no longer required to use padding to reserve space for future updates, you may still want to use padding to hide private fields from users.

You specify padding using the **PAD** option to **struct**, providing a dummy variable name. The dummy name is ignored, so you can use any name (even the same one each time). You do not need to declare the name with **dataDef**.

The following example has padding in two places. The first pad is 4 bytes after **field1_int** to hide the original **field2_int**. The second pad is 16 bytes at the end of the list, to reserve space for future fields (this is no longer required but shown for the sake of example):

```
struct myRecord {
          field1_int;
          foo, PAD=4;       // hide the original field2_int
          field3_int;
          field4_int;
          field5_short;
          foo, PAD=16;      // reserved for future use
}
```

Note: Padding of **MTYPE_FLAG** fields, including enumeration flags, is unsupported.

Version Updates - Backward Compatibility

When updating your data dictionary from one version to the next, it is important that the new version of your data extension be able to read databases containing older versions of your extension data. This section discusses explicit guidelines, that when followed, will ensure that subsequent versions of your data extension remain backward compatible with previous versions of your data extension.

If you comply with the guidelines listed below when making version updates to your data dictionary, the API will be able to automatically convert database files containing older versions of your extension data to your new format.

1 ***Do*** add new fields and records (`datadef` and `recorddef` declarations) to your data dictionary file after all `datadef` and `recorddef` declarations of previous versions. When your data dictionary is parsed, each field and record is assigned a unique integer code based on the order it appears in your data dictionary file. These integer codes are written in the headers of data extension records in a database file to identify the data extension records. If you intersperse `datadef` and `recorddef` declarations for a newer version into those of previous versions, you will alter the integer codes assigned to the fields and records of your previous versions. This will cause older extension records contained in existing database files to become unrecognizable.

2 ***Do Not*** remove a `datadef` or `recorddef` declaration from previous versions of your data dictionary. If you do so, you will alter the integer codes assigned to your existing fields and records. As shown in (1) above, this will cause older extension records contained in existing database files to become unrecognizable.

3 ***Do Not*** remove a field from a `struct` definition. When your data dictionary is parsed, the physical layouts of your records are calculated according to the order and size of the members of your `struct` definition. If you alter the physical layout of a record, you will cause databases containing older versions of your extensions to become unreadable by your new data extension. If you want to "delete" an existing field, you can replace it by padding of the same size to hide it and effectively achieve the same result.

4 ***Do*** add new fields to a `struct` definition after all fields of previous versions. If you intersperse `struct` fields for a newer version into those of previous versions, you will alter the physical layout of the record and will cause databases containing older versions of your extensions to become unreadable by your new data extension.

5 ***Do Not*** change the data type of a `datadef` declaration such that its size changes. If you do so you will alter the physical layout of the record and will cause databases containing older versions of your extensions to become unreadable by your new data extension.

6 ***Use Caution*** if you change the data type of a `datadef` declaration to a type that has the same size as that of the previous version. For example, if you change a `datadef` declaration from `MTYPE_INT` to `MTYPE_FLOAT`, the value of the field when read in from a database containing the older

version of your extension will be garbled. In this example, the 32 bit integer in the database file will be interpreted as a 32 bit floating point number when it is read in using the new data extension definition. This will yield unpredictable results but, strictly speaking, will not corrupt databases containing older versions of your extensions.

The API, when it finds an extension record in a database file that is shorter than the current corresponding `struct` definition, automatically expands the extension record (when it is read from disk into memory) to accommodate the new fields at the end of the new `struct` definition. These values for these new fields are set to 0. Of course, this assumes you have followed (4) above and have added your new fields after all fields of previous versions.

Writing a Data Extension

This chapter describes the **C** code you will write to define your data extension.

Declaring a Data Extension

Declare a data extension in your plug-in module's initialization function by calling **`mgRegisterSite`** as shown below. You must include the following header files generated by **`ddbuild`**:

`prefixcode.h` - it contains **`PrefixRecordCodeMap`** and **`PrefixDataCodeMap`**

`prefixtable_.h` - it contains **`RecordDefTable`** and **`DataDefTable`**

Prior to including **`prefixcode.h`**, you must declare the following macro to correctly export the symbols **`PrefixRecordCodeMap`** and **`PrefixDataCodeMap`**:

`#define _prefixDATA_`

Note: ***`prefix`*** is your site prefix.

In the following example, the site prefix is **`my.`** The plug-in initialization function declares the data extension:

```
#define _MYDATA_                // important!

#include "mgapiall.h"
#include "mycode.h"
#include "mytable_.h"           // generated by ddbuild

mgDeclarePlugin (
      "My Company",
      "MyData Data Dictionary",
      "5e2c23f0-1afe-11e1-9c7c-00a0241b9c3a" );

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgpluginsite pluginSite = mgRegisterSite (
      plugin, "MYDATA",
      &RecordDefTable[0], MyRecordCodeMap,
      MYOP_RECORD_OFFSET, MY_RECORD_MAX,
      &DataDefTable[0], MyDataCodeMap,
      MYOP_DATA_OFFSET, MY_DATA_MAX,
      MTA_VERSION, "1.0", MG_NULL );

   if ( pluginSite ) {
      InitActions ( pluginSite );
      InitHelpers ( pluginSite );
      InitDrawFunctions ( pluginSite );
   }
   return (pluginSite ? MG_TRUE : MG_FALSE);
}
```

The functions **`InitActions`**, **`InitHelpers`** and **`InitDrawFunctions`** are described in the following sections. You can find sample code similar to this in the sample extension directory included with the API software.

Action Functions

For each field you define in your data dictionary, you can register two *optional* action functions, a *pre-edit* function and a *post-edit* function. For each new node type you define in your data dictionary, you can also register an *optional* create function. This section describes these three action functions.

Edit Action Functions

The pre-edit function is called before a modification made to the field value is committed. The pre-edit function returns a Boolean value indicating whether or not the modification should be allowed. The post-edit function is called after a modification made to the field value is committed. Since these functions are optional, you may register one or both or neither.

Use the functions **`mgRegisterPreEdit`** and **`mgRegisterPostEdit`** to register pre-edit and post-edit functions, respectively, for a data extension field. These action functions must be registered immediately after you register your data extension in your plug-in module's initialization function as shown in the following:

```
static mgbool PreEditFunc ( mgrec* rec, mgcode code, void* val )
{
   int* iAddr = (int*) val;
   int ival = *iAddr;

   if ( ival >= 1 )
      return MG_TRUE;    // allow modification
   else
      return MG_FALSE;   // do NOT allow modification
}

static void PostEditFunc ( mgrec* rec, mgcode code, void* val )
{
   // do any post editing processing here
}

void InitActions ( mgpluginsite pluginSite )
{
   mgRegisterPreEdit ( pluginSite, myCode, PreEditFunc );
   mgRegisterPostEdit ( pluginSite, myCode, PostEditFunc );
}
```

In this example the value for field **`myCode`** is not allowed to be less than 1. This is enforced by the pre-edit callback assigned to the field.

Both the pre-edit and post-edit functions take three parameters. The first (**`mgrec* rec`**) is a pointer to the database node to which the edit applies. The second (**`mgcode code`**) is the code of the field to which the edit applies. For pre-edit functions, the third parameter (**`void* val`**) is the address of the value that is *about to be committed*. For post-edit functions, the third parameter is the address of the value that *has been committed*. The third parameter is declared as **`void*`**, but at runtime the value passed will actually point to different types of data depending on the type of the field. For example, for fields declared to be of type **`MTYPE_INT`**, **`val`** will actually be the address of an integer value (as shown in the example above). The following shows how to correctly extract some other kinds of data types from the **`val`** parameter. Numeric types are fairly obvious, but take careful note of the difference between **`MTYPE_CHAR`** and **`MTYPE_TEXT`**:

```
// for fields of type MTYPE_DOUBLE, use:
//
double* dAddr = (double*) val;
double dval = *dAddr;

// for fields of type MTYPE_CHAR, use:
//
char* cAddr = (char*) val;
char cval = *cAddr;

// for fields of type MTYPE_TEXT, use:
//
char** tAddr = (char**) val;
char* tVal = *tAddr;
```

Your pre-edit and post-edit functions *should not* attempt to modify the value pointed to by the **`val`** parameter.

Create Action Functions

If you assign a create function for an extension node type you have defined in your data dictionary, it will be called after a node of that type is created within Creator by the user either through the Extensions Menu or the Create Tool. The create action function is not called in stand-alone programs, nor is it called by `mgNewRec.`

Use the function `mgRegisterCreate` to register a create action function for an extension node type defined in your data dictionary. This action function must be registered immediately after you register your data extension in your plug-in module's initialization function as shown in the following:

```
static mgstatus CreateFunc ( mgrec* db, mgrec* rec )
{
   // can assign default attributes to rec or whatever...
}

void InitActions ( mgpluginsite pluginSite )
{
   mgRegisterCreate ( pluginSite, myCode, CreateFunc );
}
```

In this example the function `CreateFunc` will be called after the user creates a node of type `myCode` as described above.

Note: You can only register create action functions for node types you have defined in your data dictionary. Specifically, you cannot register a create action function for an OpenFlight node type.

Extension Node Helper Functions

When any node type is created in an OpenFlight database, it is assigned a unique id. For example, the first created in an OpenFlight database is assigned the id `p1`. The second is assigned `p2` and so on. When the node created is an extension node type, ensuring the uniqueness of the id assigned is impossible without the help of the data extension developer that defined that extension node type.

When you define a new node type in your data dictionary, you provide a prefix for the node type that is used to form unique ids for new nodes of that type. For example, if you define the prefix of your extension node to be `mybead`, the first node created of this type will be assigned the id `mybead1` and so forth. In addition to providing a unique prefix for each of your new node types, the following steps must be taken to ensure that the indices appended to your prefix to form ids for newly created nodes will be unique.

1. For each new node type you define, you must register a tag-along field to the OpenFlight header node used to store the index of the last node of this type added to the OpenFlight file.
2. You must register a set of *Extension Node Helper Functions* that help correlate the tag-along fields you attached to the OpenFlight header node to the new node types you define in your data dictionary.

The extension node helper functions are called when an extension node is being created within either Creator or the stand-alone program:

***Get Max Id Function*:** called before the id is assigned to the newly created node. Your function is responsible for returning the next index to assign for the specified node type. You retrieves this index from the tag-along field on the OpenFlight header node corresponding to the specified node type.

***Set Max Id Function*:** called after the id is assigned to the newly created node. Your function is passed the index just assigned. You should store this index in the tag-along field on the OpenFlight header node corresponding to the specified node type.

***Get Header Code Function*:** called to get the record code of the record containing the header tag-along fields described above.

Use the functions `mgRegisterGetMaxId`, `mgRegisterSetMaxId`, and `mgRegisterGetHeaderCode` to register Get Max Id, Set Max Id, and Get Header Code functions, respectively, for new node types you define in your data dictionary. These action functions should be registered immediately after you register your data extension in your plug-in module's initialization function as shown in the following:

static mgcode Rcode2Idcode (mgcode rcode)

```
{
  mgcode fcode;
  if ( rcode == myBead1 )
    fcode = myBdIndex1;
  else if ( rcode == myBead2 )
    fcode = myBdIndex2;
  else
   fcode = 0;
  return fcode;
}

static mgbool GetMaxIdFunc ( mgrec* rec, mgcode rcode, int* no )
{
  mgcode fcode;
  int n = 0;

  if ( fcode = Rcode2Idcode ( rcode ) )
    n = mgGetAttList ( toprec, fcode, no, MG_NULL );
  else
    *no = 0;
  return ( n ? MG_TRUE : MG_FALSE );
}

static mgbool SetMaxIdFunc ( mgrec* rec, mgcode rcode, int no )
{
  mgcode fcode;
  int n = 0;

  if ( fcode = Rcode2Idcode ( rcode ) )
    n = mgSetAttList ( toprec, fcode, no, MG_NULL );
  return ( n ? MG_TRUE : MG_FALSE );
}

static mgbool GetHeaderCodeFunc ( mgrec* rec, mgcode* rcode )
{
  *rcode = myHeader;
  return MG_TRUE;
}

void InitHelpers ( mgpluginsite pluginSite )
{
  mgRegisterGetMaxId ( pluginSite, GetMaxIdFunc );
  mgRegisterSetMaxId ( pluginSite, SetMaxIdFunc );
  mgRegisterGetHeaderCode ( pluginSite, GetHeaderCodeFunc );
}
```

Extension Node Draw Functions

When you define a new extension node type, you can register *optional* draw functions to render nodes of your extension type in the Creator graphics and hierarchy view displays. If you do so, Creator will call these functions automatically when it encounters one of your node types when drawing in the graphics and hierarchy views.

The following code fragment shows how a graphic draw function might be assigned to an extension node type.

```
static mgstatus DrawNodeGraphics ( mgrec* db,
                             mgrec* rec,
                             mggraphicviewdata* viewData )
{
   mggraphicdrawmode drawMode = viewData->drawMode;
   mgbool lightingEnabled = viewData->lightingEnabled;
   mgbool textureEnabled = viewData->textureEnabled;

   switch ( drawMode )
   {
      case MGDM_WIREFRAME:
         break;
      case MGDM_SOLID:
         break;
      case MGDM_WIRESOLID:
         break;
      case MGDM_SELECT:
         break;
   }

   // tell Creator NOT to draw any geometry below the extension node
   return MGD_DONTDRAWBELOW;
}

void InitDrawFuntions ( mgpluginsite pluginSite )
{
   mgRegisterGraphicDraw ( pluginSite, myCode, DrawNodeGraphics );
}
```

In this example the function **`DrawNodeGraphics`** is assigned as the graphic draw function for extension nodes of type **`myCode`**. When Creator draws any graphics view of a database containing a node of this type, it will call this function to draw the node. The view data passed to the graphic draw function contains parameters that specify the current viewing parameters associated with the graphics view being drawn. This includes the draw mode and whether or not texture and lighting is enabled. The value that the graphic draw function returns controls whether or not nodes below this node will be drawn by Creator. If your graphic draw function returns **`MSTAT_OK`**, nodes below your extension node will be drawn normally. If your graphic draw function returns **`MGD_DONTDRAWBELOW`** (as shown in the example above), nodes below your extension node will not be drawn by Creator.

When Creator renders a graphic view, it keeps track of the current color, material and texture (among other things) as it traverses and draws individual nodes in the database. If your graphic draw function needs to change the current, material or texture, it must use one of the API provided functions listed below. Doing so will insure that Creator can minimize state changes when rendering the graphic view.

`mgGLColor3`	sets the current color for GL rendering using 3 color components, red, green and blue.
`mgGLColor4`	sets the current color for GL rendering using 4 color components, red, green, blue and alpha.
`mgGLColorIndex`	sets the current color for GL rendering using a database color index and intensity.
`mgGLMaterialIndex`	sets the current material for GL rendering using a database material index.
`mgGLTextureIndex`	sets the current texture for GL rendering using a database texture index.

The following code fragment shows how a hierarchy draw function might be assigned to an extension node type:

```
static mgstatus DrawNodeHierarchy ( mgrec* db,
                              mgrec* rec,
                              mghierarchyviewdata* viewData )
{
   unsigned int left = viewData->left;
   unsigned int bottom = viewData->bottom;
   unsigned int right = viewData->right;
   unsigned int top = viewData->top;

   glColor3f ( 1.0, 1.0, 1.0 );
   glBegin ( GL_POLYGON );
      glVertex2i ( left, bottom );
      glVertex2i ( right, bottom );
      glVertex2i ( right, top );
      glVertex2i ( left, top );
   glEnd ();
   return (MSTAT_OK);
}

void InitDrawFuntions ( mgpluginsite pluginSite )
{
   mgRegisterHierarchyDraw ( pluginSite, myCode, DrawNodeHierarchy );
}
```

In this example the function **`DrawNodeHierarchy`** is assigned as the hierarchy draw function for extension nodes of type **`myCode`**. When Creator draws any hierarchy view of a database containing a node of this type, it will call this function to draw the node. The hierarchy of a database is drawn in the form of a two dimensional grid - each node occupying a different spot on that grid. The view data passed to the hierarchy draw function contains parameters that specify the coordinates of the rectangle where this extension node is to be drawn on the grid. Your hierarchy draw function must restrict the drawing it does to be within the bounds of this rectangle. If you do not, undefined rendering may result in the hierarchy view. Your hierarchy draw function should only draw the shape of the node. The node name and the select outline will be drawn automatically by Creator.

The value returned by the hierarchy draw function is currently ignored but is reserved for future enhancement. In this version of the API, your hierarchy draw function should always return the value **`MSTAT_OK.`**

Tools Overview

There are six classes of tools you can define in your plug-in:

- *Database Importer*: These tools read database files from unsupported file formats into Creator. (See "Writing a Database Importer" on page 57)
- *Database Exporter*: These tools write an existing database or portions of an existing database to disk formats not directly supported by Creator. (See "Writing a Database Exporter" on page 61)
- *Image Importer*: These tools read image/texture files from unsupported file formats into Creator. (See "Writing an Image Importer" on page 65)
- *Viewer*: These tools passively interact with a database. Viewers can be used to implement any action that does not modify the contents of the database. (See "Writing a Viewer" on page 75)
- *Editor*: These tools modify the contents of an existing database. (See "Writing an Editor" on page 79)
- *Input Device*: These tools convert input data from foreign input devices such as 2-D or 3-D digitizers to a format that *Editor* tools can use during the Creator modeling session. (See "Writing an Input Device" on page 129)

This chapter and those that follow describe how to create your tool. The process is as follows:

1. Write C (or C++) code that declares your specific class of tool.
2. (Optional) If any of your tools require a graphical user interface (GUI), create platform-specific resource files containing GUI dialog layout and other resources required by your tool. (See "GUI" on page 137)
3. Compile and link your plug-in module that contains your tool. (See "Building a Plug-in Module" on page 253)
4. Place your plug-in module in the plug-in runtime directory so that it is loaded when the plug-in runtime environment starts. (See "Plug-ins in the Runtime Environment" on page 259)

Declaring Tools

Declare all plug-in tools in your plug-in initialization function by calling the appropriate API tool registration function. There is a tool registration function for each of the plug-in tool classes. All of these functions have similar forms. When you register a tool using one of these functions, specify attributes that are applicable for your tool. In these functions, there are two kinds of attributes to supply, those that are required and those that are optional. Required attribute parameters appear explicitly in the parameter list of the function declaration. Optional parameters are passed using the variable argument style as described on page 22. When specifying optional tool attributes to these functions, always pass them in pairs. The first item of the pair is the *tool attribute name* while the second is the *tool attribute value*. The order in which you pass each pair is not important, only the ordering within the pair. And remember, as with all variable argument style parameter lists, you must always terminate the parameter list with the **`MG_NULL`** symbol.

All tool classes share one common required attribute:

Name: the name you have assigned to your tool.

Database importers and exporters, viewers, and editors share the following required attributes:

Start Function: function that is called when the tool is launched within Creator. This function performs different operations depending on the class of the tool.

Start Data: user data that will be passed to your Start Function when it is invoked (may be null).

Image importers and input devices each require several different callback functions described in the appropriate sections of this guide.

The optional tool attributes vary between the different classes of tools. The names and symbols of the optional tool attributes are listed here. In addition, the names of the optional attributes that are applicable for a particular tool class are listed in the section that describes that tool class:

Version: **`MTA_VERSION`**

Tool Tip: **`MTA_TOOLTIP`**

Help Context: `MTA_HELPCONTEXT`

Palette Location: `MTA_PALETTELOCATION`

Palette Icon: `MTA_PALETTEICON`

Menu Location: `MTA_MENULOCATION`

Menu Position: `MTA_MENUPOSITION`

Menu Submenu: `MTA_MENUSUBMENU`

Menu Label: `MTA_MENULABEL`

Filter: `MTA_FILTER`

File Type: `MTA_FILETYPE`

Tools in the Program Environment

The way in which a tool is invoked or launched in Creator differs for each class of tool. This section describes how each class of tool is accessed by the user within Creator.

Database importers and exporters and image importers are launched from within file selection dialogs that Creator provides. Each class of importer and exporter has a corresponding file selection dialog from which these tools are launched.

Viewer and editor tools are launched from either a menu or a toolbox in Creator. When you register a viewer or editor, you specify which menu or toolbox. The menus and toolboxes from which you can select are enumerated by symbols in the API.

The menus and corresponding symbols from which you can select are:

File menu: `MMENU_FILE`

Edit Menu: `MMENU_EDIT`

View Menu: `MMENU_VIEW`

Info Menu: `MMENU_INFO`

Select Menu: `MMENU_SELECT`

Attributes Menu: `MMENU_ATTRIBUTES`

LOD Menu: `MMENU_LOD`

Local-DOF Menu: `MMENU_LOCALDOF`

Palettes Menu: `MMENU_PALETTES`

Terrain Menu: `MMENU_TERRAIN`

Road Menu: `MMENU_ROAD`

GeoFeature Menu: `MMENU_GEOFEATURE`

Sound Menu: `MMENU_SOUND`

Instruments Menu: `MMENU_INSTRUMENTS`

BSP Menu: `MMENU_BSP`

Help Menu: `MMENU_HELP`

The toolboxes and corresponding symbols from which you can select are:

Face Tools: `MPAL_FACETOOLS`

Geometry Tools: `MPAL_GEOMETRYTOOLS`

Maneuver Tools: `MPAL_MANEUVERTOOLS`

Duplicate Tools: `MPAL_DUPLICATE`

Modify Geometry Tools: `MPAL_MODIFYGEOMETRY`

Modify Face Tools: `MPAL_MODIFYFACE`

Modify Vertex Tools: `MPAL_MODIFYVERTEX`

Properties Tools: `MPAL_PROPERTIES`

Map Texture Tools: `MPAL_MAPTEXTURE`

Modify Texture Tools: `MPAL_MODIFYTEXTURE`

Construction Tools: `MPAL_CONSTRUCTIONTOOLS`

Boolean Tools: `MPAL_BOOLEANTOOLS`

Deform Tools: `MPAL_DEFORMTOOLS`

Wizard Tools: `MPAL_WIZARDTOOLS`

Create Tools: `MPAL_CREATETOOLS`

Hierarchy Tools: `MPAL_HIERARCHYTOOLS`

Input device tools are launched in conjunction with editor tools that request 2-D or 3-D point input. Only when an editor tool is active will the data provided by an input device tool be accessible.

Tool Instances

When a tool is launched by the user, a logical invocation of the tool is created. The logical invocation of a tool is referred to as a *tool instance*. Since different classes of tools are designed to implement different kinds of database interactions, Creator imposes different protocols upon the instances of each tool class.

It is important to be aware of these protocols when you are designing your plug-in tool. It is even more important to understand the reasons why these protocols exist so that you can design your plug-in tool to uphold the spirit of the protocol and therefore better co-exist in Creator.

The protocols for database importer, database exporter, and image importer tools are all similar. The graphical user interface for accessing these kinds of tools is modal. This is because the act of selecting a file and the corresponding action of importing or exporting must be performed atomically. Given this, only one importer or exporter instance will ever be active at any time during the modeling session. Therefore an importer or exporter tool can have at most one active instance.

Creator enforces that only one editor tool instance be active at any time per database. In other words, when an editor tool instance is active for a given database, it is guaranteed to be the only editor tool instance active for that database. This is important for two reasons. First, since editor tools can modify the contents of a database, this protocol provides a semaphore mechanism for protecting the modification of database elements. Second, since editor tools can control how mouse pointer input is interpreted within database views, this protocol provides arbitration for decoding mouse pointer input. For these reasons, editor tools can have at most one instance active per open database. For example, if there are two databases open within the Creator, there may be, at most, two separate editor tool instances active at that time, one for each database. These two editor tool instances may be for the same editor tool or for two different editor tools.

Creator imposes very little protocol on viewer tools. That is because viewer tools, by definition, provide “read only” access to database elements. During the modeling session, one or more viewer tool instances may be active at any one time. This may include one or more instances of the same viewer tool. The lack of protocol enforced by Creator for viewer tool instances provides more flexibility to the tool but at the same time puts more burden on the tool to perform its actions sensibly.

Creator imposes a very strict protocol on input device tools. This protocol is very much coupled with input requests made by active editor tool instances.

The protocols for each class of tool are described further in the respective section that describes that tool class.

Activation Database

All tool instances, except those for input devices, are launched in the context of a particular database open within Creator. This database, referred to as the *activation database*, is the top database (e.g., database that has focus) within Creator when the tool instance was launched. In general, while a tool instance is active, it is limited to interacting only with its associated activation database. Viewer tool instances are allowed to deviate from this limitation. Since input device tool instances must operate independent of which database is open, they too are exempt from this limitation.This section discusses the relationship between tool instances and activation databases.

While one is active, a tool instance for a database importer, database exporter, image importer, or editor tool must interact only with its associated activation database. This is not only what the user expects but also a presumption upon which much API and Creator functionality is based. The undo mechanism in the modeling system, for example, depends on tool instances to follow this rule. Adhering to this convention helps maintain data and system integrity during the modeling session.

When a viewer tool instance is first launched, it is initially expected to interact with its associated activation database (this is most likely what the user intended when launching the tool). But since a viewer tool instance may remain active indefinitely, it must be allowed to interact with other databases when that becomes appropriate. For example, you may design a viewer tool that continuously displays and updates bounding volume dimensions for the top database within Creator. When the user opens a new database or switches focus to another database, the viewer tool instance needs to update the dimensions displayed to reflect the new top database. If a viewer tool instance is intended to stay active like this, it is a good practice and will be more clear to the user if it is designed to always reflect the state of the top database.

Writing a Database Importer

Database importer tools are accessed within Creator in the **Import Database File Dialog.** All database importers that are loaded are accessed in this dialog. Here the user selects the importer tool to use and specifies the file to be imported. The database importer tool converts the contents of the file to OpenFlight constructs and attaches them to the current database.

Declaring a Database Importer

You declare a database importer tool in your plug-in module's initialization function by calling `mgRegisterImporter`. The following code fragment shows how a database importer tool might be registered:

```
static mgstatus ImporterStartFunc ( mgplugintool pluginTool,
                              void* userData,
                              void* callData )
{
   mgimportercallbackrec* importerData = (mgimportercallbackrec*) callData;
   char* fileName = importerData->fileName;
   mgrec* topDb = mgGetActivationDb ( importerData->toolActivation );
   mgrec* parent = mgGetModelingParent ( topDb );

   // import the contents of file named fileName into database topDb
   return (MSTAT_OK);
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;

   pluginTool = mgRegisterImporter (
      plugin, "My Database Importer",
      ImporterStartFunc, MG_NULL,
      MTA_VERSION, "1.0",
      MTA_FILTER, "*.ext",
      MTA_FILETYPE, "My Kind of Files",
      MG_NULL
      );
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

You specify the following *required* tool attributes:

Name - the name you have assigned to your importer.

Start Function - function called when the user selects a file to import in the Import Database File Dialog. This function is responsible for converting the contents of the specified file to valid OpenFlight constructs and attaching them to the current database.

Start Data - user data passed to your Start Function when it is invoked (may be null).

In addition to these required tool attributes, you can specify any of the following *optional* tool attributes:

Version - text version you have assigned to your importer.

Filter - text used as the filter pattern in the Import Database File Dialog.

File Type - text used as the filter description in the Import Database File Dialog.

Help Context - text that identifies a context sensitive help topic for your importer. See "Context Sensitive Help" on page 242 for a description of this attribute.

In the example listed above, the name of the tool is **"My Database Importer"**. The start function is **ImporterStartFunc**, while the start data is **MG_NULL**. This tool has declared itself as version **"1.0"**. The filter pattern for file matching is **"*.ext"** while the file type is **"My Kind of Files"**.

The value returned by the start function should indicate whether or not the import succeeded or failed. The start function should therefore return **MSTAT_OK** if the file was imported successfully. If the file did not import successfully, the start function should return any non-zero value. It is also recommended that the start function report an error message indicating the nature of the failure.

Database Importers in the Program Environment

After the user selects a file from the Import Database File Dialog, the appropriate database importer start function is called to extract database elements from the selected file, convert those elements to OpenFlight format, and finally attach those elements into the current database. The current database is specified in the parameters passed to your start function as shown in the example above. Your importer should not attach database elements to any other database. The elements you import can include geometry, palette items (color, material, texture, etc.), eyepoints, etc.

The whole of the database importing process must occur entirely within the database importer **start** function. That is, when the start function returns, the importing must be complete. This means that if your database importer must acquire additional user input after your start function is called, you must be able to gather and process that input before your start function returns. Using a modeless dialog for this is not acceptable as this would require your start function to return before the dialog contents are processed. Using a modal dialog, on the other hand, would allow your start function to gather user input synchronously. Modal dialogs are discussed in "User Defined Modal Dialogs" on page 152.

The geometry elements that the database importer creates may be attached to the current modeling parent node or any other node of the current database. As shown in the example above, the function **`mgGetModelingParent`** returns the current modeling parent for the specified database. To attach to another node of the current database, your database importer may traverse down the current database node to find a suitable attach point.

Your database importer can also create non-geometric database elements such as palette items and/or eyepoints for the current database using the appropriate Read/Write API functions.

When a database importer is launched from within Creator, the user can specify whether the file is to imported into a new database window or into the current database window. If the file is imported into a new database window and the start function fails (returns any value other than **`MSTAT_OK)`**, the new database window will not be created on the desktop. If the file is imported into the current database window and the start function fails, no such adverse action is taken.

Writing a Database Exporter

Database exporter tools are accessed within Creator in the *Export Database File Dialog*. All database exporters that are loaded are accessed in this dialog. Here the user selects the exporter tool to use and specifies the file to which the contents of the current database are to be exported. The database exporter tool converts the contents of the current database to the file format expected in the output file.

Declaring a Database Exporter

You declare a database exporter tool in your plug-in module's initialization function by calling **`mgRegisterExporter`**. The following code fragment shows how a database importer tool might be registered:

```
static mgstatus ExporterStartFunc ( mgplugintool pluginTool,
                              void* userData,
                              void* callData )
{
   mgexportercallbackrec* exporterData = (mgexportercallbackrec*) callData;
   mgrec* topdb = mgGetActivationDb ( exporterData->toolActivation );
   char* fileName = exporterData->fileName;

   // export the contents of database topDb to file named fileName
   return (MSTAT_OK);
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;

   pluginTool = mgRegisterExporter (
      plugin, "My Database Exporter",
      ExporterStartFunc, MG_NULL,
      MTA_VERSION, "1.0",
      MTA_FILTER, "*.ext",
      MTA_FILETYPE, "My Kind of Files",
      MG_NULL
      );
   return (exporter ? MG_TRUE : MG_FALSE);
}
```

You specify the following *required* tool attributes:

Name - the name you have assigned to your exporter.

Start Function - function called when the user enters the name of the file to which the current database is to be exported in the Export Database File Dialog. This function is responsible for converting the contents of the current database to the correct file format.

Start Data - user data passed to your Start Function when it is invoked (may be null).

In addition to these required tool attributes, you can specify any of the following *optional* tool attributes:

Version - text version you have assigned to your exporter.

Filter - text used as the filter pattern in the Export Database File Dialog.

File Type - text used as the filter description in the Export Database File Dialog.

Help Context - text that identifies a context sensitive help topic for your exporter. See "Context Sensitive Help" on page 242 for a description of this attribute.

In the example listed above, the name of the tool is `"My Database Exporter"`. The start function is `ExporterStartFunc`, while the start data is `MG_NULL`. This tool has declared itself as version `"1.0"`. The filter pattern for file matching is `"*.ext"` while the file type is `"My Kind of Files"`.

Note: The value returned by the database exporter start function is currently ignored but is reserved for future enhancement. In this version of the API, your start function should always return the value `MSTAT_OK.`

Database Exporters in the Program Environment

After the user selects a file name from the Export Database File Dialog, the appropriate database exporter start function is called to extract database elements from the current database, convert those elements to the exported database format and write those elements to the specified file. The current database is specified in the parameters passed to your start function as shown in the example above. Your exporter should not export elements from any other database. The elements you export can include geometry, palette items (color, material, texture, etc.), eyepoints, etc.

The whole of the database exporting process must occur entirely within the database exporter start function. That is, when the start function returns, the exporting must be complete. This means that if your database exporter must acquire additional user input after your start function is called, you must be able to gather and process that input before your start function returns. Using a modal dialog would allow your start function to gather user input synchronously. Modal dialogs are discussed in "User Defined Modal Dialogs" on page 152. Modeless dialogs would not be suitable to use by database exporters as they would require your start function to return before the dialog contents are processed.

Your database exporter may process all or parts of the current database. For example, it may be written to export only the nodes selected in the current database. Alternatively, it may be designed to export the entire database or possibly, only nodes meeting a certain criteria (such as polygons that have exactly three vertices). Obtaining the list of currently selected nodes from a database is discussed in "The Modeling Context" on page 205. Designing your own database traversal/criteria matching function can be done using the appropriate Read/Write API traversal and attribute query functions.

Your database exporter can also extract non-geometric database elements such as palette items and/or eyepoints from the current database using the appropriate Read/Write API functions.

Writing an Image Importer

Image importer tools are accessed within Creator in the *Read Pattern Dialog*. All image importers that are loaded are accessed in this dialog. Here the user selects the importer tool to use and specifies the file to be imported. The image importer tool converts the contents of the file to a format that Creator understands so that the image may be displayed in the Texture palette and applied to geometry within the modeling environment.

The interaction between Creator and image importer tools differs significantly from that of database importer tools. For database importer tools, the process of importing a database file is atomic. In other words, when a database file is to be imported, Creator makes a single call to the database importer start function to perform the whole of the import processing.

The process of importing an image file, on the other hand, is carried out in a series of smaller transactions between Creator and the image importer tool. The sequence of transactions that take place when an image file is imported consist of the following:

1. `Open` - The importer is asked to open an image file.
2. `Get Info` - The importer is asked to return information (type, width, height) for the open image file.
3. `Get Geo Info` - The importer is asked to return georeferencing information (if present) for the open image file. Most image file formats do not support georeferencing data so this is optional.
4. `Get Texels` - The importer is asked to return the texels of the open image file.
5. `Close` - The importer is asked to close the open image file.

Note: The interactions between Creator and image importer tools have been defined in a modular fashion to allow for planned enhancements to image importer plug-in capabilities. Future capabilities will be designed to incorporate and leverage these fundamental transactions. Currently, the capabilities provided by an image importer tool are only used to import full resolution image texels. Future releases of the API will support multiple resolutions, tiling, and paging of image files. As the API expands to support these capabilities, the capabilities provided by image importer tools will grow as well.

Declaring an Image Importer

Instead of supplying one single start function like database importers, an image importer tool will provide several callback functions, each corresponding to one of the transactions listed above. Collectively, these image importer callback functions are referred to as *image importer functions*.

Declare an image importer tool in your plug-in module's initialization function by calling **`mgRegisterImageImporter`**. You specify the following *required* tool attributes:

Name - the name you have assigned to your importer.

Import Flags - a bit mask specifying what kinds of capabilities your image importer provides. Currently only *whole image importing* is supported.

Open Image Function - called when the API needs your tool to open an image file.

Close Image Function - called when the API needs your tool to close an image file.

Get Image Info Function - called when the API needs information (width, height, etc.) regarding an open image.

Get Image Texels Function - called when the API needs the texels for an open image.

User Data - user data passed to the image importer functions when they are invoked (may be null).

In addition to these required tool attributes, you can specify any of the following *optional* tool attributes:

Version - text version you have assigned to your importer.

Filter - text used as the filter pattern in the Read Pattern Dialog.

File Type - text used as the filter description in the Read Pattern Dialog.

Help Context - text that identifies a context sensitive help topic for your importer. See "Context Sensitive Help" on page 242 for a description of this attribute.

The following code fragment shows how an image importer tool might be registered.

```
static mgstatus ImageOpenFunc ( mgplugintool pluginTool,
                  void* userData,
                  const char* filename,
                  const char* filemode,
                  int* imageId )
{
   // open image file and assign unique identifier to imageId
   return MIMG_NO_ERROR;
}

static mgstatus ImageCloseFunc ( mgplugintool pluginTool,
                  void* userData,
                  int imageId )
{
   // close image file corresponding to imageId
   return MIMG_NO_ERROR;
}

static mgstatus ImageGetInfoFunc ( mgplugintool pluginTool,
                  void* userData,
                  int imageId,
                  mgimageinfo textureInfo )
{
   int width;
   int height;
   int type;

   // determine type and calculate width and height for image
   // corresponding to imageId, assign values to textureInfo

   mgSetTextureWidth ( textureInfo, width );
   mgSetTextureHeight ( textureInfo, height );
   mgSetTextureType ( textureInfo, type );
   mgSetTextureSampleSize ( textureInfo, 8 );
   mgSetTextureTiledFlag ( textureInfo, MG_FALSE );

   return MIMG_NO_ERROR;
}

static mgstatus ImageGetTexelsFunc ( mgplugintool pluginTool,
                  void* userData,
                  int imageId,
                  int resolution,
                  unsigned char* texels )
{
   // load texels with those extracted from image
   // corresponding to imageId
   return MIMG_NO_ERROR;
}
MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;

   pluginTool = mgRegisterImageImporter (
      plugin, “My Image Importer”,
      MGP_IMAGEWHOLE,
      ImageOpenFunc, ImageCloseFunc,
      ImageGetInfoFunc, ImageGetTexelsFunc,
      MG_NULL,
      MTA_VERSION, “1.0”,
      MTA_FILTER, “*.img”,
      MTA_FILETYPE, “My Image Files”,
      MG_NULL );
   return (pluginTool ? MG_TRUE : MG_FALSE);
}

static mgstatus ImageGetTexelsFunc ( mgplugintool pluginTool,
                  void* userData,
                  int imageId,
                  int resolution,
                  unsigned char* texels )
{
   // load texels with those extracted from image
   // corresponding to imageId
   return MIMG_NO_ERROR;
}
```

```
MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;

   pluginTool = mgRegisterImageImporter (
      plugin, “My Image Importer”,
      MGP_IMAGEWHOLE,
      ImageOpenFunc, ImageCloseFunc,
      ImageGetInfoFunc, ImageGetTexelsFunc,
      MG_NULL,
      MTA_VERSION, “1.0”,
      MTA_FILTER, “*.img”,
      MTA_FILETYPE, “My Image Files”,
      MG_NULL );
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

In this example, the name of the tool is **`“My Image Importer”`**. The importer capability flag must be **`MGP_IMAGEWHOLE`** as whole image importing is the only image importing capability supported by the API. Future releases of the API may provide for reduced resolution importing, etc. The image importer functions are **`ImageOpenFunc`** (open image), **`ImageCloseFunc`** (close image), **`ImageGetInfoFunc`** (get image info) and **`ImageGetTexelsFunc`** (get image texels). The user data that is passed to the image importer functions is **`MG_NULL`**. This tool has declared itself as version **`"1.0"`**. The filter pattern for file matching is **`"*.img"`**.

Image Importers in the Program Environment

After the user selects a file from the Read Pattern Dialog, the appropriate open image function is called to open the selected image file. If successful, the open function returns a unique *image identifier* that will be used in future image importer transactions to identify the open image. The image importer tool should leave this image file open until the close image function is called. After an image file is successfully open, the get image info function is called to return size and type information for the image. Using the type and size information returned by this function, the API creates a texel buffer large enough to hold the texels of the image. This buffer is passed to the get image texels function where it is filled in by the tool with the texels of the image. Finally, after the texels are received for the image, the API calls the close image function to close the file associated with the image.

Get Image Info Function

The get image info function is responsible for providing important information to the API about how an open image is represented during the modeling session. This information, referred to as *image info*, includes such attributes as image type, width, and height.

When the API calls the get image info function, it passes an opaque object of type **`mgimageinfo`** to the tool. This object is called an *image object*. Before returning, the get image info function must assign values for each field of the image object using the following functions:

`mgSetTextureType`	sets the type attribute for an image object.
`mgSetTextureWidth`	sets the width attribute for an image object.
`mgSetTextureHeight`	sets the height attribute for an image object.
`mgSetTextureSampleSize`	sets the sample size attribute for an image object, valid values are 8 and 16.

`mgSetTextureTiledFlag`	sets the tiled flag attribute for an image object.
`mgSetTextureMinMax`	sets the minimum and maximum texel values for an image object.
`mgSetTextureTransparentValue`	sets the transparent texel value attribute for an image object.
mgSetTextureSignedFlag	sets the signed flag attribute for an image object.

Note: Tiled flag is reserved for future enhancement. Your image importer tool should always set the value for this fields as follows:

```
mgSetTextureTiledFlag ( imageInfo, MG_FALSE );
```

The values assigned for image type, width, and height determine the memory format of the image texels and are described in more detail in the following section.

Get Image Texels Function

The get image texels function of an image importer tool is responsible for providing texel data for an open image. When this function is called by the API, the tool is expected to fill in a block of memory with the texels extracted from the image. This block of memory is pre-allocated by the API. The size of this block is based on the type, dimensions and sample size of the image returned by the get image info function. This memory is completely managed by the API. Your image importer should only fill in the texel values. This section describes the texel memory format filled in by the image importer tool.

Note: The texel memory format defined for image importer tools is NOT the same as that defined for image read functions in API levels 1 and 2.

There are four types of images your image importer tool can return: *Intensity*, *Intensity-Alpha*, *RGB*, and *RGBA*. A texel is represented by 1 (Intensity), 2 (Intensity-Alpha), 3 (RGB), or 4 (RGBA) components. Each component is one 8-bit byte or one 16-bit word (depending on the sample size) representing the value of the component. The values of these components have different meanings for each kind of component as described in the table below.

Each image type is represented by an interleaved, row oriented, 4-byte boundary aligned memory format. This means:

- All components of a texel are stored contiguously (e.g., RGBRGB... for type RGB)
- Rows of texels are stored contiguously (e.g., r1c1 r1c2 ... r2c1 r2c2 ...) in which r1 is row 1 and c2 is column 2, etc. The texel in the lower left corner of the image is at row 1, column 1.
- Padding is added at the end of each row such that the number of bytes allocated for that row is always a multiple of 4.

Each type has a different memory format as shown in the following table.

Intensity	**`MIMG_INT`**	These are single band grayscale images. A texel value of 0 is black and a value of 255 (8-bit) or 65535 (16-bit) is white.
Intensity-Alpha	**`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 (8-bit) or 65535 (16-bit) is white. In the alpha channel, a texel value of 0 is transparent and a value of 255 (8-bit) or 65535 (16-bit) is opaque.
RGB	**`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 (8-bit) or 65535 (16-bit) is full intensity.
RGBA	**`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 (8-bit) or 65535 (16-bit) is full intensity. In the alpha channel, a texel value of 0 is transparent and a value of 255 (8-bit) or 65535 (16-bit) is opaque.

Georeferenced Images

The API supports *optional* georeferencing information attached to images imported by image importer plug-in tools. If your image importer wants to provide georeferencing information for images that it imports, it must register a **`Get Geo Info`** function after the importer is registered in the plug-in module's initialization function. The purpose of the get geo info function is described in the next section.

The example from above has been modified to register a get geo info function. Only new or modified functions are shown here:

```
static mgstatus GetGeoInfoFunc ( mgplugintool pluginTool,
                  void* userData,
                  int imageId,
                  mgimagegeoinfo geoInfo )
{
   int projection, earthModel;
   int utmZone, utmHemisphere;
   int imageOrigin;
   int numControlPoints, i;
   double imageX, imageY;
   double projX, projY;

   // determine projection, earth model, UTM info, etc for
   // image corresponding to imageId, assign values to geoInfo

   mgSetTextureGeoType ( geoInfo, MIMG_GEOTYPE_CTRLPT );
   mgSetTextureGeoProjection ( geoInfo, projection );
   mgSetTextureGeoEarthModel ( geoInfo, earthModel );
   mgSetTextureGeoUTMZone ( geoInfo, utmZone );
   mgSetTextureGeoUTMHemisphere ( geoInfo, utmHemisphere );
   mgSetTextureGeoImageOrigin ( geoInfo, imageOrigin );
   mgSetTextureGeoNumCtlPts ( geoInfo, numControlPoints );
   for ( i = 0; i < numControlPoints; i++ )
   {
      // calculate ith control point
      mgSetTextureGeoCtlPt ( geoInfo, i, imageX, imageY, projX, projY );
   }
   return MIMG_NO_ERROR;
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;

   pluginTool = mgRegisterImageImporter (
      plugin, "My Image Importer",
      MGP_IMAGEWHOLE,
      ImageOpenFunc, ImageCloseFunc,
      ImageGetInfoFunc, ImageGetTexelsFunc,
      MG_NULL,
      MTA_VERSION, "1.0",
      MTA_FILTER, "*.img",
      MTA_FILETYPE, "My Image Files",
      MG_NULL
      );

   if ( pluginTool )
      mgSetReadImageGeoInfoFunc (
         plugin, pluginTool,
         GetGeoInfoFunc, MG_NULL
         );

   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

Get Geo Info Function

The get geo info function is responsible for providing georeferencing information about an open image if such information is present in the image. This information, referred to as *geo image info*, includes such attributes as image geo type, projection, earth model, UTM info, origin and geographic control points.

When the API calls the get geo info function, it passes an opaque object of type **`mgimagegeoinfo`** to the tool. This object is called an *georeference info object*. Before returning, the get geo info function must assign values for each applicable field of the georeference info object using the following functions:

`mgSetTextureGeoType`	sets the geographic control type attribute for a georeference info object.
`mgSetTextureGeoProjection`	sets the map projection attribute for a georeference info object.
`mgSetTextureGeoEarthModel`	sets the Earth model (Ellipsoid) attribute for a georeference info object.
`mgSetTextureGeoUTMZone`	sets the UTM zone attribute for a georeference info object.
`mgSetTextureGeoUTMHemisphere`	sets the hemisphere attribute for a georeference info object for the UTM ptojection.
`mgSetTextureGeoImageOrigin`	sets the image origin attribute for a georeference info object.
`mgSetTextureGeoNumCtlPts`	sets the number of control points attribute (used to geo reference the image) for a georeference info object.

`mgSetTextureGeoCtlPt`	sets the control points used to geo reference an image for a georeference info object.

If the get geo info function returns **`MIMG_NO_ERROR`**, the georeferencing information contained in the georeference info object is assumed to describe the georeferencing information for the open image. These attributes will be added to open image and become accessible for the image.

Writing a Viewer

Viewer tools are accessed from either a menu or a toolbox in Creator. If you want your viewer to be launched from a menu, specify the menu and the caption to appear on the menu item. Similarly, if you want your viewer to be launched from a toolbox, specify which toolbox, the icon to be displayed on the toolbox button and an optional tool tip caption to be displayed when the user positions the mouse over the button in the toolbox.

Declaring a Viewer

Declare a viewer tool in your plug-in module's initialization function by calling **`mgRegisterViewer`**.

The following code fragment shows how a viewer tool might be registered:

```
static mgstatus ViewerStartFunc ( mgplugintool pluginTool,
                           void* userData,
                           void* callData )
{
   mgviewercallbackrec* viewerData = (mgviewercallbackrec*) callData;
   mgrec* topDb = mgGetActivationDb ( viewerData->toolActivation );

   // begin processing database topDb
   return (MSTAT_OK);
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;

   pluginTool = mgRegisterViewer (
      plugin, "My Viewer",
      ViewerStartFunc, MG_NULL,
      MTA_VERSION, "1.0",
      MTA_MENULOCATION, MMENU_INFO,
      MTA_MENUPOSITION, "Statistics",
      MTA_MENUSUBMENU, "My Submenu",
      MTA_MENULABEL, "My &Viewer...",
      MTA_HELPCONTEXT, "My Viewer Topic",
      MG_NULL
      );
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

You specify the following *required* tool attributes:

Name - the name you have assigned to your viewer.

Start Function - function called when the user launches your viewer.

Start Data - user data passed to your Start Function when it is invoked (may be null).

In addition to these required tool attributes, you can specify any of the following *optional* tool attributes:

Version - character string version you have assigned to your viewer.

Palette Location - the toolbox where the button for your tool will be located.

Palette Icon - the platform-specific image to display on your toolbox button.

Tool Tip - the tool tip text to display over your toolbox button.

Menu Location - the menu where the menu item for your tool will be located.

Menu Label - the text to display on your menu item.

Menu Position - the item in the menu before or after which your menu item will be placed.

Menu Submenu - the submenu in the menu where your menu item will be located.

Help Context - the text that identifies a context sensitive help topic for your viewer. See "Context Sensitive Help" on page 242 for a description of this attribute.

In the example listed above, the name of the tool is **`"My Viewer"`**. The start function is **`ViewerStartFunc`**, while the start data is **`MG_NULL`**. This tool has declared itself as version **`"1.0".`** The tool will be launched from a menu item that will be placed in a submenu created in the Info menu of Creator. The new submenu created will have the caption **`"My Submenu".`** It will be created after the existing **`"Statistics"`** menu item. The caption of the new menu item will be **`"My Viewer"`**.

When the user requests context sensitive help for this tool, the topic **`"My Viewer Topic"`** of the plug-in help file will be displayed in the Creator online help window. Valid menu and palette locations are listed in “Tools in the Program Environment” on page 51.

Note: If you specify either Palette Location or Palette Icon, you must specify both. Similarly, if you specify either Menu Location or Menu Label, you must specify both. Also, although you are allowed to specify that your tool be accessed from both a menu item and a toolbox, it is recommended that you choose one or the other (not both).

The value returned by the viewer start function is currently ignored but is reserved for future enhancement. In this version of the API, the start function should always return the value **`MSTAT_OK.`**

Viewers in the Program Environment

When the user launches a viewer tool, the start function is called to notify the tool that it has been invoked. At this point, the viewer tool may choose to activate a new tool instance or re-activate an existing one. Unlike database importer and exporter tool instances that must perform all processing synchronously within their start functions, viewer tool instances may remain active after their start function returns. Of course, a viewer tool instance can perform all of its processing within its start function if that is appropriate, but this is not required. It is acceptable for a viewer tool instance to stay active indefinitely during the modeling session.

If a viewer tool acts on the select list, it can obtain the list of currently selected nodes from a database. See “The Select List” on page 206 for more information.

Also, if your viewer is to remain active indefinitely, you may select model-time events to be reported to your tool using the event notification mechanism discussed in “Event Notification” on page 215.

Writing an Editor

Editor tools are accessed from either a menu or a toolbox in Creator. If you want your tool to be launched from a menu, specify which menu and the text that is to appear on the menu item. Similarly, if you want your tool to be launched from a toolbox, you will specify which toolbox, the icon that will be displayed on your toolbox button and an optional tool tip caption that will be displayed when the user positions the mouse over your button in the toolbox.

Declaring an Editor

Declare an editor tool in your plug-in module's initialization function by calling **`mgRegisterEditor.`** The following code fragment shows how an editor tool might be registered.

Note: This example will be referenced and expanded in subsequent sections as more editor tool functionality is discussed:

```
static mgstatus EditorStartFunc ( mgplugintool pluginTool,
                            void* userData,
                            void* callData )
{
   mgeditorcallbackrec* editorData = (mgeditorcallbackrec*) callData;
   mgrec* topDb = mgGetActivationDb ( editorData->toolActivation );

   // begin processing database topDb
   return (MSTAT_OK);
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;
   mgmodulehandle moduleHandle = mgGetModuleHandle ( plugin );
   mgresource resource = mgLoadResource ( moduleHandle );
   mgpixmap pixmap = mgResourceGetPixmap ( resource, MY_BITMAP );

   pluginTool = mgRegisterEditor (
      plugin, "My Editor",
      EditorStartFunc, MG_NULL,
      MTA_VERSION, "1.0",
      MTA_PALETTELOCATION, MPAL_FACETOOLS,
      MTA_PALETTEICON, pixmap,
      MG_NULL
      );
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

You specify the following *required* tool attributes:

Name - the name assigned to your editor.

Start Function - the function called when the user launches your editor.

Start Data - user data passed to your Start Function when it is invoked (may be null).

In addition to these required tool attributes, you can specify any of the following *optional* tool attributes:

Version - character string version you have assigned to your editor.

Palette Location - the toolbox where the button for your tool will be located.

Palette Icon - the platform-specific image to display on your toolbox button.

Tool Tip - the tool tip text to display over your toolbox button.

Menu Location - the menu where the menu item for your tool will be located.

Menu Label - the text to display on your menu item.

Menu Position - the item in the menu before or after which your menu item will be placed.

Menu Submenu - the submenu in the menu where your menu item will be located.

Help Context - the text that identifies a context sensitive help topic for your editor. See "Context Sensitive Help" on page 242 for a description of this attribute.

Scriptable - a boolean flag specifying whether or not your tool is available in Creator Script. By default editor tools are not scriptable. You must do certain things when you write your editor tool if you want it to be available in Creator Script. See "Making your Editor Tool Scriptable" on page 115 for more information.

In the example listed above, the name of the tool is **`"My Editor"`**. The start function is **`EditorStartFunc`**, and the start data is **`MG_NULL`**. This tool has declared itself as version **`"1.0"`**. The tool will be launched from a tool icon whose pixmap identifier is **`MY_BITMAP`** within the Face toolbox in

Creator. Valid palette and menu locations are listed in “Tools in the Program Environment” on page 51. Since no tool tip was specified, the name of the tool, `“My Editor”` is used as the tool tip text for the tool icon. When the user requests context sensitive help for this tool, the plug-in help file will be displayed but no topic will be searched for since no help context was specified.

Note: If you specify either Palette Location or Palette Icon, you must specify both. Similarly, if you specify either Menu Location or Menu Label, you must specify both. Also, although you are allowed to specify that your tool be accessed from both a menu item and a toolbox, it is recommended that you choose one or the other (not both).

The value returned by the editor start function is currently ignored but is reserved for future enhancement. In this version of the API, your start function should always return the value `MSTAT_OK`.

Editors in the Program Environment

When an editor tool is launched, the tool’s start function is called to notify the tool that it has been invoked. Editor tools can be invoked to run interactively in Creator or can be invoked in the context of Creator Script.

Note: If you have not defined your editor tool as “scriptable” (see “Declaring an Editor Tool as Scriptable” on page 116), your tool can only be invoked in Creator to run interactively. If your editor tool is “scriptable”, it can be invoked to run interactively or in Creator Script.

When your editor tool is invoked, information about “how” your tool is being used is passed to your start function in the form of a *tool activation*. This tool activation includes information such as the active database, whether or not the tool is running interactively or in Creator Script, and if the tool is running in Creator Script, a pointer to the parameter block for the tool. For more information on the tool activation, see “Editor Tool Activation” on page 82.

Unlike viewer tools that can choose to activate either a new or existing tool instance when they are invoked, the editor tool must activate a new tool instance in the context of the associated activation database. An editor tool instance can perform all of its processing within its start function if that is appropriate, or it can choose to display a dialog to solicit additional user input. If invoked in the context of Creator Script, the editor tool must perform its processing wholly within its start function as dialogs should not be displayed in Creator Script (this is explained in more detail in “Making the Start Function for a Scriptable Editor Tool” on page 123). In any case, while it remains active, the editor tool instance must interact only with its associated activation database.

If an editor tool acts on the select list, it can obtain the list of currently selected nodes from a database. See “The Select List” on page 206 for more information.

Also, you may select model-time events to be reported to your editor tool using the event notification mechanism discussed in “Event Notification” on page 215.

It is very likely that your editor tool instance will require additional user input before it can complete its processing task. User input can come in many forms. The most common form is that which is contained in a dialog that your editor tool instance will display. Additionally, the API allows an editor tool instance to receive user input via the mouse pointer device. Using the mouse, the user can point and click in the database graphics view to enter 2D point or 3D vertex coordinates, or pick geometry nodes. Editor tools are the only class of plug-in tools that can receive mouse input in this way.

If your editor tool instance requires additional user input in any form, there is a prescribed protocol your tool instance must follow to ensure correct operation within Creator. The remainder of this chapter describes this protocol as well as some recommended guidelines for your editor tool instance.

Editor Tool Activation

When the user launches an editor tool, Creator invokes the corresponding editor tool start function. A tool activation (**`mgtoolactivation`**) is passed to the start function in the **`callData`** parameter. This tool activation record contains information about the context in which the tool is expected to operate, including the active database. The tool activation also specifies whether the tool is being invoked interactively or in Creator Script.

The following code fragment shows how an editor tool start function might examine the tool activation record passed in:

```
static mgstatus EditorStartFunc ( mgplugintool pluginTool,
                         void* userData,
                         void* callData )
{
  mgeditorcallbackrec* editorData = (mgeditorcallbackrec*) callData;
  mgtoolactivation toolActivation = editorData->toolActivation;
  mgrec* topDb = mgGetActivationDb ( toolActivation );
  mgtoolactivationtype activationType = mgGetActivationType (
toolActivation );

  // activationType is MTAT_NORMAL if running interatively
  // activationType is MTAT_SCRIPT if running in Creator Script

  // begin processing database topDb
  return (MSTAT_OK);
}
```

In this example, you will notice that **`mgGetActivationType`** is used to determine whether the tool is being invoked interactively or in Creator Script. If the tool is being invoked interactively, the tool may create a dialog to solicit additional data from the user. This is described in "Editor Tool Dialogs" on page 84. If the tool is "scriptable" and is being invoked in Creator Script, the tool must perform all its processing before the start function returns. See "Making the Start Function for a Scriptable Editor Tool" on page 123 for more information.

Here are the functions you can use to extract information from the tool activation:

`mgGetActivationDb`	retrieves the database from a tool activation object.

`mgGetActivationParamBlock`	retrieves the activation parameter block from a tool activation object.
`mgGetActivationType`	retrieves the activation type from a tool activation object.

Editor Tool Dialogs

If your editor tool is running interactively and requires user input when it is launched, you must provide a dialog in which the user interaction takes place. The first part of this section describes some conventions to follow when you are designing your editor tool dialog. Some conventions described here are required and some are provided merely as recommendations. Adhering to the recommendations as well as the requirements, however, provides a more consistent user interface among editor tools in Creator.

The following are **required** layout conventions:

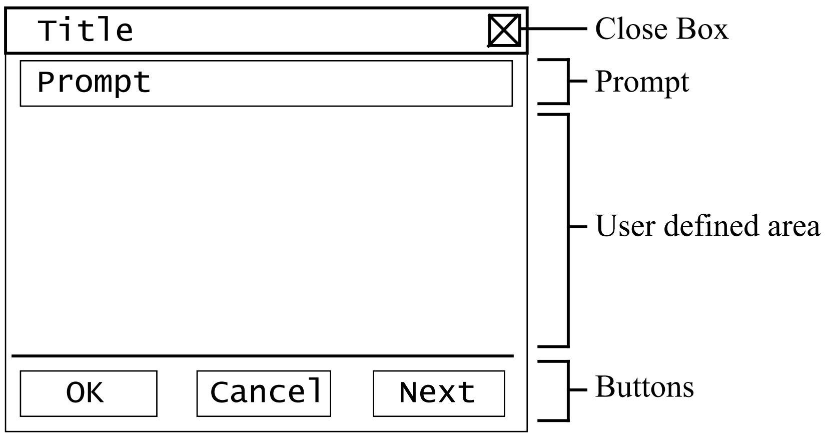
- Editor tool dialogs shall have at least two push button controls; **OK** and **Cancel**.
- The **OK** push button control shall have identifier `MGID_OK` (1).
- The **Cancel** push button control shall have identifier `MGID_CANCEL` (2).
- The **Next** push button control (if present) shall have identifier `MGID_NEXT` (3).
- The **Prompt** text control (if present) shall have identifier `MGID_PROMPT` (5).

The following are **recommended** layout conventions:

- The **OK**, **Cancel**, and **Next** (if present) push button controls should appear along the bottom of the dialog in the order shown in the following diagram. If the **Next** push button control is not present, the other two buttons should be positioned in the lower right corner of the dialog (**Cancel** on the right, **OK** to its left).
- The **Prompt** text control (if present) should appear along the top of the dialog as shown in the following diagram.
- The minimum width for editor tool dialogs is 182.

- The **Help** push button control (if present) should have the identifier **`MGID_HELP`** (4).
- The **Undo** push button control (if present) should have the identifier **`MGID_UNDO`** (6).
- The **Redo** push button control (if present) should have the identifier **`MGID_REDO`** (7).
- If the editor tool dialog has 3 or fewer buttons (**OK**, **Cancel** and **Next**), the width of each button should be 54.

The recommended layout for editor tool dialogs is shown in the following diagram. Note that the **Prompt** text control, **Next** push button control, and horizontal separator are all optional.



Remember that Creator enforces that at most one editor tool instance is active at a time per database. To help manage this, Creator takes over much of the dialog management responsibilities for editor tools.

Note: You should not use `mgSetTitle` to set the title for an editor tool dialog. Titles for editor tool dialogs are managed by the API and display the name of the editor tool as well as the name of the database for which the editor was launched.

There are several steps you must take if your editor tool displays a dialog. First, create a dialog template describing the layout of your dialog. Second, register a *create dialog function* for your tool. Third, indicate during the editor tool start function whether or not the dialog is needed. Finally, register a *termination function* for your tool. The first step, creating a dialog template, is discussed in "GUI" on page 137. The other steps are described in the remainder of this section.

As previously stated, Creator performs most of the dialog management tasks for your editor tool dialog. Over the lifetime of an editor tool instance, the corresponding tool dialog will be created, displayed, hidden, and destroyed automatically for you by Creator. The only part of the dialog management that you must do is to register a create dialog function for your editor tool using the function **`mgEditorSetCreateDialogFunc.`** The create dialog function you register will be called by the API to create a dialog instance when your editor tool is launched as described below. You can register the create dialog function for your editor tool after you register the editor tool itself, typically in the plug-in initialization function.

Again, you should not use any of the dialog management functions **`mgShowDialog`**, **`mgHideDialog`** or **`mgDestroyDialog`** on editor tool dialogs as these functions are performed automatically by Creator at appropriate times during the lifetime of your editor tool instance.

In the body of the editor tool start function, indicate as part of the return parameters whether or not the dialog is needed for the tool instance. Use this mechanism to allow or disallow the editor tool instance from continuing. Set the **`dialogRequired`** field of the **`mgeditorcallbackrec`** record passed to your start function to **`MG_TRUE`** if you want the dialog to be created, **`MG_FALSE`** otherwise. By default, this field is initialized to **`MG_FALSE,`** so if you do nothing, no dialog is created. You can also fill in the optional field, **`toolData`** of this record to an arbitrary **`void*`** value. This value, referred to as *tool instance data,* is then passed to subsequent editor tool functions as described in later sections of this chapter.

As described above, editor tool dialogs may contain as many as three pre-defined push button controls. The first two, **OK** and **Cancel**, are required while the third, **Next** is optional. A fourth button, **Help** is defined but not yet implemented; it is reserved for future enhancement. As a convenience, you can also register a button function for your editor tool that will be called by the API when the user presses either of these three push button controls in your tool dialog. Use the function **`mgEditorSetButtonFunc`** to register a button function for your editor tool. For the button function to work correctly, the **OK**, **Cancel**, and **Next** push button controls must have the identifiers **`MGID_OK`**, **`MGID_CANCEL`** and **`MGID_NEXT`**, respectively.

You must also register a termination function for your editor tool using the function **`mgEditorSetTerminateFunc`**. The function you register will be called by the API when your tool instance terminates for any reason. The termination reason is specified as a parameter (of type **`mgtoolterminationreason`**) to your termination function.

The following is a list of actions that can cause an editor tool instance to be terminated followed by the associated **`mgtoolterminationreason`** value.

- User presses the **OK** button in the editor tool dialog - **`MTRM_DONE.`**
- User presses the **Cancel** button in the editor tool dialog - **`MTRM_CANCEL.`**
- User clicks on the close icon in the editor tool dialog - **`MTRM_SYSTEM.`**
- User closes the activation database associated with the tool instance - **`MTRM_SYSTEM.`**
- User launches another editor tool for the same activation database as that associated with the tool instance - **`MTRM_SYSTEM.`**
- User exits the modeling session - **`MTRM_SYSTEM.`**
- Tool instance terminates itself by calling **`mgEditorTerminateTool`** - **`MTRM_SELF.`**

Your termination function may need to perform different actions depending on the termination reason.

If your editor tool dialog includes a **Close Box** the title bar and the user clicks this, the default behavior of your editor tool should be to cancel the tool instance. Most tools in Creator follow this convention. By default, Creator makes this happen automatically by simply calling your editor tool button function with **`MBT_CANCEL`** when the user clicks the **Close Box**. In this way, it will appear to your code that the user simply clicked the **Cancel** button. Put another way, your code cannot distinguish between the user clicking the **Close Box** and the user clicking the **Cancel** button.

You can change this default behavior by defining a close dialog function for your editor tool using **`mgEditorSetCloseDialogFunc`**. If you do this and the user clicks the **Close Box**, Creator will call your close dialog function and let it decide how to proceed. Your close dialog function can specify which button event to send to your button function (**`MBT_OK`** or **`MBT_CANCEL`**) in response to the **Close Box** click or if the **Close Box** click is to be ignored completely. Your close dialog function might prompt the user what they want to do or it might use some other criteria to decide how to proceed.

In the following code fragment, our example has been expanded to show how a plug-in would register and implement a create dialog function, a termination function, and a button function for the editor tool:

```
// typedef for tool instance data
typedef struct mytoolrec_t {
   mgrec*     db;
   mgresource resource;
   mggui      dialog;
} mytoolrec;

static mgstatus EditorStartFunc ( mgplugintool pluginTool,
                            void* userData,
                            void* callData )
{
   mgresource resource = (mgresource) userData;
   mgeditorcallbackrec* editorData = (mgeditorcallbackrec*) callData;
   mgrec* topDb = mgGetActivationDb ( editorData->toolActivation );

   if ( OkToStartTool ( topDb ) ) {
      // tool instance can continue
      mytoolrec* instanceData = mgMalloc ( sizeof(mytoolrec) );
      instanceData->resource = resource;
      instanceData->db = topDb;
      instanceData->dialog = MG_NULL;

      // tell API dialog is needed
      editorData->dialogRequired = MG_TRUE;
      editorData->toolData = instanceData;
   }
   else
      // tool instance cannot continue, dialog not needed
      editorData->dialogRequired = MG_FALSE;

   return (MSTAT_OK);
}

static mgstatus DialogProc ( mggui dialog, mgdialogid dialogId,
                        mgguicallbackreason callbackReason,
                        void *userData, void *callData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;

   switch ( callbackReason ) {
      case MGCB_INIT:
         instanceData->dialog = dialog;
         break;
      case MGCB_DESTROY:
         break;
   }
   return (MSTAT_OK);
}

static mggui CreateDialogFunc ( mgplugintool pluginTool, void* toolData )
{
   mggui dialog;
   mytoolrec* instanceData = (mytoolrec*) toolData;

   dialog = mgResourceGetDialog ( MG_NULL, instanceData->resource,
                              MYEDITORDIALOG,
                              MGCB_INIT|MGCB_DESTROY,
                              DialogProc, instanceData );
   return (dialog);
}

static void ButtonFunc ( mgeditorcontext editorContext,
                     int whichButton,
                     void* toolData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;

   switch ( whichButton ) {
      case MBT_CANCEL:
         break;
      case MBT_DONE:
         break;
      case MBT_NEXT:
         break;
      case MBT_HELP:  // reserved for future enhancement
         break;
   }
}

static void TerminateFunc ( mgeditorcontext editorContext,
                      mgtoolterminationreason terminateReason,
                      void* toolData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;

   switch ( terminateReason ) {
      case MTRM_DONE:
         break;
      case MTRM_CANCEL:
         break;
      case MTRM_SELF:
         break;
      case MTRM_SYSTEM:
         break;
   }
   mgFree ( instanceData );
}

static mgbool CloseDialogFunc ( mgplugintool pluginTool,
                      mgclosedialogcallbackrec* callData,
                      void* toolData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;

   // ask the user if they really want to cancel
   int answer = mgMessageDialog (instanceData->dialog, "Confirm",
      "Are you sure you want to cancel without saving changes?",
      MMBX_YESNOCANCEL);
   if (answer == 1) {
      // user answered Yes, they want to cancel but not save changes
      // tell Creator to send the Cancel button event
      callData->buttonEvent = MBT_CANCEL;
   }
   // if you return MG_TRUE, the buttonEvent you specify in the
   // callData will be sent to your button function, the dialog
   // is closed and the tool instance is terminated
   // if you return MG_FALSE, your button function is not called,
   // the dialog is not closed and the tool instance is not terminated
   return answer == 1 ? MG_TRUE : MG_FALSE;
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;
   mgmodulehandle moduleHandle = mgGetModuleHandle ( plugin );
   mgresource resource = mgLoadResource ( moduleHandle );
   mgpixmap pixmap = mgResourceGetPixmap ( resource, MY_BITMAP );

   pluginTool = mgRegisterEditor (
      plugin, "My Editor",
      EditorStartFunc, resource,
      MTA_VERSION, "1.0",
      MTA_PALETTELOCATION, MPAL_FACETOOLS,
      MTA_PALETTEICON, pixmap,
      MG_NULL
      );
   if ( pluginTool ) {
      mgEditorSetCreateDialogFunc ( pluginTool, CreateDialogFunc );
      mgEditorSetTerminateFunc ( pluginTool, TerminateFunc );
      mgEditorSetButtonFunc ( pluginTool, ButtonFunc );
      mgEditorSetCloseDialogFunc ( pluginTool, CloseDialogFunc );
   }
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

In this example, when the user launches **`"My Editor"`**, the specified tool start function, **`EditorStartFunc`**, is called by the API. Since **`resource`** has been specified as the start data for the editor tool, it will be passed as **`userData`** to the start function. In the start function, a user provided function, **`OkToStartTool`** is invoked to determine if the editor tool instance can continue. If the tool instance can continue, the field **`dialogRequired`** is set to **`MG_TRUE`** and the field **`toolData`** is filled in with user defined tool instance data. When the start function returns, the API checks the value of the **`dialogRequired`** field to see if the editor tool dialog is needed. If so, the API will invoke the create dialog function registered, **`CreateDialogFunc`** to create a new dialog to display in conjunction with this instance of the editor tool. Since the API completely manages the dialog created by this function, it is very important that the dialog returned by the create dialog function is a new dialog instance.

If the user presses one of the pre-defined push button controls, **OK**, **Cancel**, or **Next** while the tool dialog is displayed, the specified button function, **`ButtonFunc`** is called indicating which button was pressed.

When the tool instance is terminated, the specified termination function, **`TerminateFunc`** is called indicating how the tool instance was terminated.

Editor Context

When your editor tool instance is active, the operations it performs are non-modal in nature. As discussed in this chapter, you will register several callbacks that the API will call to notify your editor tool instance of significant user actions. To help manage these non-modal operations and to better keep track of your editor tool instance while it is active, Creator maintains an object called an *editor context* that encapsulates the state of your tool instance. The editor context data structure is opaque to your tool but contains valuable information used by Creator while your tool instance remains active Throughout this chapter, the terms *editor context* and *editor tool instance* are used interchangeably.

Many of the callbacks you register for your editor tool will receive an editor context as a parameter. In the example in the previous section, both the button function and the termination function receive an object of type **`mgeditorcontext`** which is the editor context that identifies the editor tool instance. In turn, many of the API functions your callbacks are likely to use may require you to pass that editor context to them to identify your editor tool instance.

The following are API functions that extract information from an editor context or perform actions on behalf of an editor context:

`mgEditorGetDbRec`	returns the activation database associated to an editor context.
`mgEditorGetDialog`	returns the dialog associated to an editor context.
`mgEditorGetContext`	returns the editor context associated to a dialog.

`mgEditorGetToolData`	returns the tool instance data associated to an editor context (value specified for **`toolData`** field of **`mgeditorcallbackrec`** record passed to editor start function).
`mgEditorSetPrompt`	loads a string into the prompt text control of the tool dialog associated to an editor context. The text control must have identifier **`MGID_PROMPT.`**
`mgEditorTerminateTool`	terminates the editor tool instance associated to an editor context with the termination reason **`MTRM_SELF`**.
`mgEditorCancelTool`	terminates the editor tool instance associated to an editor context with the termination reason **`MTRM_CANCEL`**.
`mgEditorRefreshScene`	notifies Creator that the activation database associated to an editor context has changed and needs to be redrawn.

While the editor tool instance is active, the tool instance data you provided in the editor tool start function, typically, should contain all the data your editor tool instance needs. If, for some reason, you need additional data to be associated with your tool instance, you can attach additional plug-in defined data to the corresponding editor context. See "Editor Context Properties" on page 227 for more information.

Device Input

While an editor tool instance is active, it can accept input from external input devices. Device input data is presented to editor tool instances in one of following forms:

- Two dimensional (2D) coordinates (point)
- Three dimensional (3D) coordinates (vertex)
- Geometry node selection (pick)
- Other device specific data (device input)

Creator provides built in support for the mouse pointer and allows other input devices to be registered as input device plug-in tools. Using the mouse pointer, the user can point and click directly in the database graphics view to enter 2D or 3D coordinates or to select geometry nodes. Using other registered input plug-in devices, the user can enter 2D or 3D coordinates or device specific data using whatever metaphor is appropriate for the plug-in device. Input device plug-in tools are discussed in "Writing an Input Device" on page 129.

Regardless of which device (mouse or plug-in device) reports the input data, the data is always delivered to editor tool instances in a consistent form. Because Creator converts raw device data to known forms automatically, editor tools need not worry about which input device is reporting the data. In this way, editor tools that accept device input will always be compatible with any plug-in input device. To achieve this, Creator automatically packages all input device data (regardless of which device actually reports the data) to appear as if it is being delivered from a generic **`mouse device.`** Throughout the remainder of this chapter, all input device data reported to editor tool instances will be referred to as **`mouse input.`**

If your editor tool needs to receive user input via input devices in any of these ways, you must register a mouse input function for each kind of device input it can accept. You can register these functions after you register the editor tool itself, typically in the plug-in initialization function. Then while your editor tool instance is active, you can select which mouse input function to make active at different stages of the tool instance.

To register mouse input functions, use the following:

`mgEditorSetVertexFunc`	registers a mouse input function to receive 3D coordinates from the mouse device.
`mgEditorSetPointFunc`	registers a mouse input function to receive 2D coordinates the mouse device.
`mgEditorSetPickFunc`	registers a mouse input function to receive geometry picking from the mouse device.
`mgEditorSetDeviceInputFunc`	registers a mouse input function to receive device specific data from registered plug-in input devices.

While your editor tool can register a mouse input function for each type of mouse input, only one mouse input function can be active at one time while your editor tool instance is active. Use the function

`mgEditorSelectMouseInput` to select which mouse input function is active.

Note: By default, no mouse function is active when an editor tool instance starts. No mouse input will be reported to your tool instance until you call the function `mgEditorSelectMouseInput` specifying which mouse input function to make active.

In the following code fragment, the example has been expanded to show how a plug-in would register and implement mouse input functions for the editor tool. Only new or modified functions are shown:

```
static int VertexFunc ( mgeditorcontext editorContext,
                        mgvertexinputdata* vertexInputData,
                        void* toolData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;
   int updateRef = 0;

   mgmousestate mouseEvent = vertexInputData->mouseEvent;
   unsigned int keyboardFlags = vertexInputData->keyboardFlags;
   unsigned int buttonFlags = vertexInputData->buttonFlags;
   mgcoord3d* thisPoint = vertexInputData->thisPoint;
   mgcoord3d* firstPoint = vertexInputData->firstPoint;

   switch ( mouseEvent )
   {
      case MMSS_START:
         break;
      case MMSS_CONTINUE:
         break;
      case MMSS_STOP:
         break;
   }
   return (updateRef);
}

static void PointFunc ( mgeditorcontext editorContext,
                        mgpointinputdata* pointInputData,
                        void* toolData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;
   mgmousestate mouseEvent = pointInputData->mouseEvent;
   unsigned int keyboardFlags = pointInputData->keyboardFlags;
   unsigned int buttonFlags = pointInputData->buttonFlags;
   mgcoord2i* thisPoint = pointInputData->thisPoint;
   mgcoord2i* firstPoint = pointInputData->firstPoint;

   switch ( mouseEvent )
   {
      case MMSS_START:
         break;
      case MMSS_CONTINUE:
         break;
      case MMSS_STOP:
         break;
   }
}

static void PickFunc ( mgeditorcontext editorContext,
                   mgpickinputdata* pickInputData,
                   void* toolData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;
   unsigned int keyboardFlags = pickInputData->keyboardFlags;
   unsigned int buttonFlags = pickInputData->buttonFlags;
   mgselectlist selectList = pickInputData->pickList;
}

static void DeviceInputFunc ( mgeditorcontext editorContext,
                         mgdeviceinputdata* deviceInputData,
                         void* toolData )
{
   mytoolrec* instanceData = (mytoolrec*) toolData;
   mginputdevice inputDevice = deviceInputData->inputDevice;
   void* deviceData = deviceInputData->deviceData;
   int button = deviceInputData->button;
   mgmousestate mouseEvent = deviceInputData->mouseEvent;
   char* inputDeviceName = deviceInputData->toolName;

   switch ( mouseEvent )
   {
      case MMSS_START:
         break;
      case MMSS_CONTINUE:
         break;
      case MMSS_STOP:
         break;
   }
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;
   mgmodulehandle moduleHandle = mgGetModuleHandle ( plugin );
   mgresource resource = mgLoadResource ( moduleHandle );
   mgpixmap pixmap = mgResourceGetPixmap ( resource, MY_BITMAP );

   pluginTool = mgRegisterEditor (
      plugin, "My Editor",
      EditorStartFunc, resource,
      MTA_VERSION, "1.0",
      MTA_PALETTELOCATION, MPAL_FACETOOLS,
      MTA_PALETTEICON, pixmap,
      MG_NULL
      );

   if ( pluginTool ) {
      mgEditorSetCreateDialogFunc ( pluginTool, CreateDialogFunc );
      mgEditorSetTerminateFunc ( pluginTool, TerminateFunc );
      mgEditorSetButtonFunc ( pluginTool, ButtonFunc );
      // register mouse input functions
      mgEditorSetVertexFunc ( pluginTool, VertexFunc );
      mgEditorSetPointFunc ( pluginTool, PointFunc );
      mgEditorSetPickFunc ( pluginTool, PickFunc );
      mgEditorSetDeviceInputFunc ( pluginTool, DeviceInputFunc );
   }
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

In this updated example, four mouse input functions have been registered to accept input from the user via the mouse device. Depending on which mouse input function is active, **`VertexFunc`** will be called to report 3D point input, **`PointFunc`** to report 2D point input, **`PickFunc`** to report geometry pick input, and **`DeviceInputFunc`** to report any device specific data generated.

While the editor tool instance is active, the tool can select which mouse input function is active as follows:

```
// Select 3D vertex input function to be active
mgEditorSelectMouseInput ( editorContext, MMSI_VERTEXINPUT );

// Select 2D point input function to be active
mgEditorSelectMouseInput ( editorContext, MMSI_POINTINPUT );

// Select geometry pick input function to be active
mgEditorSelectMouseInput ( editorContext, MMSI_PICKINPUT );

// Select device specific input function to be active
mgEditorSelectMouseInput ( editorContext, MMSI_DEVICEINPUT );

// Select no mouse input function to be active
mgEditorSelectMouseInput ( editorContext, MMSI_NOINPUT );
```

Coordinate Input Techniques

For 2D point input, the coordinates of the point reported are measured in pixels from the lower left corner of the database graphics view. The lower left corner is `(0,0)`.

For 3D point input, the coordinates of the point reported are measured in database units. The origin of the database is `(0.0,0.0,0.0)`. 3D coordinate input is interpreted from the 2D mouse pointer position using the same coordinate entry techniques employed within Creator (tracking plane, face, vertex, and edge reference, etc.). Refer to the on-line help in Creator for more information.

In addition to the default 3D coordinate entry techniques provided within Creator, an editor tool instance can further control how the 3D coordinate is derived from the mouse pointer input.

An editor tool instance can specify a tracking plane that will override the plane set by the user. Doing so will cause all 3D point input to be projected onto the specified plane instead of the current tracking plane in Creator. To install an override tracking plane, use the function **`mgEditorSetTrackPlane`** specifying the address of the plane you want to use. To revert to using the tracking plane defined in Creator, specify **`MG_NULL`** as the plane address.

Note: The grid drawn by Creator does not change when an override tracking plane is set.

An editor tool instance can request that all 3D point input be "snapped" to a specified line before being reported. This line is referred to as a *snap line*. A snap line has different effects on different coordinate entry techniques used by the user. For vertex or edge referenced 3D point input, the actual point reported is the point on the snap line that is closest to the referenced point. For normal 3D point input (input that would otherwise be projected onto the tracking plane), a snap line actually changes the way the mouse position is projected into database space. To install a snap line, use the function **`mgEditorSetSnapLine`** specifying the address of the line you want to use. To revert to normal 3D coordinate input, specify **`MG_NULL`** as the line address.

An editor tool instance can also request that all 3D point input be "snapped" to faces in the scene before being reported. When this is enabled, all 3D point input is projected onto the face being pointed to by the user in Creator. To enable or disable this, use the function **`mgEditorSetSnapFace.`**

Construction Geometry

An editor tool instance can create *construction geometry* in its activation database. Construction geometry can be created to represent vertices and/or edges and is not attached to any node hierarchies in the database. Construction vertices are single points in space represented by x, y, and z coordinates. Construction edges are composed of two construction vertices connected by a line segment. Both construction vertices and edges can be colored using predefined color values. The valid colors are:

Color	Symbol
Red	`MCCOLOR_RED`
Green	`MCCOLOR_GREEN`
Yellow	`MCCOLOR_YELLOW`
Blue	`MCCOLOR_BLUE`
Magenta	`MCCOLOR_MAGENTA`
Cyan	`MCCOLOR_CYAN`
White	`MCCOLOR_WHITE`

Creator draws construction geometry in the database graphic view automatically even though it is not attached to any node. The editor context object associated to the editor tool instance keeps track of all the construction geometry created by that tool instance. Because it keeps track of construction geometry in this way, the API can provide functions to traverse all the construction geometry nodes associated with an editor tool instance.

The table below describes the functions that manipulate construction vertices:

`mgNewConstructVertex`	creates a construction vertex node.
`mgDeleteConstructVertex`	seletes a construction vertex node.
`mgSetConstructVertexColor`	sets the color of a construction vertex node.
`mgSetConstructVertexCoords`	sets the coordinates of a construction vertex node.
`mgFirstConstructVertex`	gets the first construction vertex node associated with an active editor tool instance.

`mgNextConstructVertex`	gets the next construction vertex node associated with an active editor tool instance.

The table below describes the functions that manipulate construction edges:

`mgNewConstructEdge`	creates a construction edge node.
`mgDeleteConstructEdge`	deletes a construction edge node.
`mgSetConstructEdgeColor`	sets the color of a construction edge node.
`mgSetConstructEdgeCoords`	sets the coordinates of the endpoints of construction edge node.
`mgFirstConstructEdge`	gets the first construction edge node associated with an active editor tool instance.
`mgNextConstructEdge`	gets the next construction edge node associated with an active editor tool instance.

When an editor tool instance is terminated, it can choose to delete any or all of the construction geometry it has created. All construction geometry deleted will be removed from the scene. If the editor tool instance does not delete all of the construction geometry it has created, it must provide an undo function that will do so. See "Undo/Redo" on page 103. Any construction geometry not deleted remains in the scene.

To delete individual construction nodes, use **`mgDeleteConstructVertex`** and **`mgDeleteConstructEdge`** as described above. To delete every construction node that was created while your editor tool instance was active, use **`mgDeleteAllConstructs`**.

Note: Construction geometry is not saved with the OpenFlight database file.

The Focus Vertex

While an editor tool is active and requesting 3D vertex input, it may want to draw the user's attention to the location of a particular 3D vertex in the graphics view. The Polygon tool in Creator, for example, does this as the user enters successive vertices of the polygon being created. The current vertex being entered by the user draws with a special symbol in the graphics view in order to focus the user's attention on that vertex. Furthermore, after entering several vertices of the polygon, if the user wants to re-enter the coordinates of one of the previously entered vertices, he can press the **`CTRL`** key while clicking the vertex he wants to re-enter. This will cause the vertex that he selects to be drawn with the special symbol. Now the next 3D vertex coordinate the user enters will be applied to this selected vertex of the polygon. To return to entering new vertices for the polygon, the user can **`CTRL`** select anywhere in the graphics view that does not select a vertex contained in the polygon.

The API provides a mechanism called the *focus vertex* to support this kind of functionality. First some terms will be defined. The *focus vertex* is the vertex in the graphics view that is drawn using the special symbol described above. This symbol is called the *focus vertex icon*. There is at most one focus vertex in the graphics view at a time. When the user presses the **CTRL** key and clicks on a vertex in the graphics view to select a new focus vertex, this action is called *selecting a focus vertex* or *focus vertex selection*. When the user presses the **CTRL** key anywhere in the graphics view that does not select a vertex in the focus vertex list, this action is called *deselecting a focus vertex*. Creator maintains a list of database nodes called the *focus vertex list*. The nodes in this list are the nodes from which the user can select a focus vertex. Editor tools add and delete nodes from this list to control which nodes in the scene are candidates to become the focus vertex.

The API provides the following functions to manipulate the focus vertex and the focus vertex list.

`mgSetFocusVertex`	sets the current focus vertex.
`mgGetFocusVertex`	gets the current focus vertex.
`mgClearFocusVertex`	clears the current focus vertex.
`mgFocusVertexListAddItem`	adds a node to the focus vertex list. Vertices in this last are candidates for focus vertex selection by the user as described above.
`mgFocusVertexListDeleteAllItems`	deletes all nodes from the focus vertex list.

`mgFocusVertexListDeleteItem`	removes a particular node from the focus vertex list.

Your editor tool can use these functions anytime while its editor tool instance is active. If your editor tool will use the focus vertex functionality and wants to allow the user to be able to select a focus vertex from the focus vertex list as described above, your editor tool must register a focus vertex function using **`mgSetFocusVertexFunc.`** The focus vertex function you register will be called by Creator when the user selects or deselects a focus vertex from the current focus vertex list. You must register your focus vertex function for your editor tool after you register the editor tool itself, typically in the plug-in initialization function.

In the following code fragment, the example has been modified to show how a plug-in would register and implement focus vertex input functions for the editor tool. Only new or modified functions are shown:

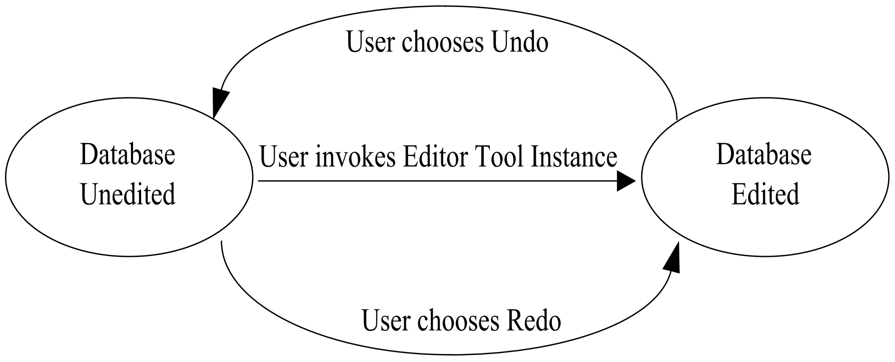
```
static int VertexFunc ( mgeditorcontext editorContext,
                    mgvertexinputdata* vertexInputData,
                    void* toolData )
{
   int updateRef = 0;
   mgmousestate mouseEvent = vertexInputData->mouseEvent;
   mgcoord3d* thisPoint = vertexInputData->thisPoint;
   mgrec* focusVertex = mgGetFocusVertex ( editorContext );

   switch ( mouseEvent )
   {
      case MMSS_START:
         if ( focusVertex )
            // user has selected a focus vertex using CTRL+select
            // ...just update its coords
            mgSetConstructVertexCoords ( focusVertex, thisPoint );
         else {
            // this is a new vertex, create a construction vertex,
            // make it the current focus vertex and add to focus
            // vertex list so user can use CTRL+select to select
            // it later
            mgrec* newVtx =
               mgNewConstructVertex ( editorContext, thisPoint );
            mgSetFocusVertex ( editorContext, newVtx );
            mgFocusVertexListAddItem ( editorContext, newVtx );
         }
         break;

      case MMSS_CONTINUE:
      case MMSS_STOP:
         if ( focusVertx )
            // modify the coords of focus vertex using thisPoint
            mgSetConstructVertexCoords ( focusVertx, thisPoint );
         break;
   }
   return (updateRef);
}

static void FocusVertexFunc ( mgeditorcontext editorContext,
                         mgrec* vertexRec,
                         void* toolData )
{
   if ( vertexRec )
      // vertexRec is guaranteed to be one of the construction
      // vertices our tool has created as those are the only
      // nodes in the focus vertex list eligible to be selected
      // by the user
      mgSetFocusVertex ( editorContext, vertexRec );
   else
      // user CTRL+selected anywhere in the graphic view other
      // than at a vertex in the focus vertex list.  This means
      // to clear the focus vertex.  Next 3D vertex entered will
      // now create a new construction vertex.
      mgClearFocusVertex ( editorContext );
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;
   mgmodulehandle moduleHandle = mgGetModuleHandle ( plugin );
   mgresource resource = mgLoadResource ( moduleHandle );
   mgpixmap pixmap = mgResourceGetPixmap ( resource, MY_BITMAP );

   pluginTool = mgRegisterEditor (
      plugin, "My Editor",
      EditorStartFunc, resource,
      MTA_VERSION, "1.0",
      MTA_PALETTELOCATION, MPAL_FACETOOLS,
      MTA_PALETTEICON, pixmap,
      MG_NULL
      );

   if ( pluginTool ) {
      mgEditorSetCreateDialogFunc ( pluginTool, CreateDialogFunc );
      mgEditorSetTerminateFunc ( pluginTool, TerminateFunc );
      mgEditorSetButtonFunc ( pluginTool, ButtonFunc );
      // register 3D vertex input function
      mgEditorSetVertexFunc ( pluginTool, VertexFunc );
      // register focus vertex function
      mgEditorSetFocusVertexFunc ( pluginTool, FocusVertexFunc );
   }
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

In this example, the editor tool creates a new construction vertex every time the user enters a 3D vertex coordinate. Each new construction vertex created is added to the focus vertex list and therefore becomes eligible for focus vertex selection by the user. If the user does select a focus vertex in this way, that vertex will be the vertex to which all subsequent 3D vertex coordinates entered will be applied. In this way, the user can go back and reposition construction vertices previously entered. To begin entering new construction vertices again, the user must deselect the current focus vertex.

Note: When the user selects or deselects a focus vertex, the focus vertex function is called but the current focus vertex is neither set nor cleared automatically. This is left to the focus vertex function. If a focus vertex is selected, it will be passed to the focus vertex function. If the focus vertex is deselected, `MG_NULL` will be passed to the focus vertex function.

Undo/Redo

Creator supports undo and redo for editing operations performed by the user. The figure below describes database state transitions as the user edits during the modeling session. The database states are represented by the ovals. The transitions between the database states (the user actions) are represented by the arrows between the states.



As the user performs edit operations in Creator on a database, those that can be undone are listed in the **Edit>Undo History** submenu. The most recent undo-able operation appears as the first item in this submenu and also explicitly as the **Edit>Undo *Tool Name*** menu item (where ***Tool Name*** is the name of the undo-able tool). Creator updates the **Edit>Undo *Tool Name*** menu item and adds menu items to the **Edit>Undo History** submenu automatically as new tools are subsequently invoked by the user.

Creator maintains the list of undo-able operations internally in a *stack*. There is one such stack per database. These stacks are implemented as Last-In-First-Out (LIFO) and each has a maximum length specified by the user via the **Max**

Undos Editing preference. Each new operation in a database is added to the top of the stack for that database, moving previous operations down. When the number of entries in the stack exceeds the maximum length of the stack, the entry at the bottom is removed. When an entry is removed from the stack, it can never be accessed again. When the user switches from one database to another in Creator, the **Edit>Undo History** and **Edit>Redo History** submenus are updated to reflect the undo/redo stacks of the corresponding database.

The user can undo the most recent operation by choosing the **Edit>Undo *Tool Name*** menu item or can undo a sequence of recent operations by choosing any item listed in the **Edit>Undo History** submenu. When the user chooses the **Edit>Undo *Tool Name*** menu item (the top operation in the stack), only that operation is undone. When the user chooses an item from the **Edit>Undo History** submenu (other than the first item which is equivalent to choosing the **Edit>Undo *Tool Name*** menu item), Creator will undo the selected operation as well as all operations above it in the menu. For example, if the user chooses the 3rd item in the **Edit>Undo History** submenu, the 1st, 2nd and 3rd items are all undone. Since all items in the **Edit>Undo History** submenu are dependent on the actions undone by previous items in the menu, the order in which these 3 items are undone is 1st, 2nd, then 3rd.

Just as operations are listed in the **Edit>Undo History** submenu as they are performed, they get listed in the **Edit>Redo History** as they are undone. This allows the user to perform an action, undo it and then to redo it as needed.

This section gives an overview of the Creator undo/redo mechanism, describes the interfaces provided by the API to access it and describes a set of Undo Helper Functions provided by the API that you can use to perform standard undo/redo processing on behalf of your editor tool.

Note: Not all editor tools are designed to modify the database. If an editor tool is not designed to modify the database, it is not required to implement undo or redo.

Undo

As described above, undo items are stored internally in a stack. The API provides access to this *undo stack* by allowing editor tools to add their operations to it. Each element in the undo stack is referred to as an *undo entry*.

*Undo menu item*s appear in the **Edit>Undo History** submenu of Creator. When an editor tool instance creates a new undo entry, the entry can be assigned a new undo menu item in the **Edit>Undo History** submenu or it can be appended onto a previous undo menu item. In this way, undo menu items can represent one or more undo entries. Appending undo entries onto an existing undo menu item allows an editor tool instance to make several smaller undo entries look like one undo menu item to the user. This allows you to segment the functionality performed by your editor tool as you choose.

When your editor tool instance creates an undo entry, it provides two callback functions. The first is referred to as the *undo function*. The second is referred to as the *undo cleanup function*. You can also specify an optional opaque data pointer (*user data*) that will be passed to the undo and undo cleanup functions when they are called.

The undo function is responsible for restoring the database to the state it was in prior to the actions performed by the editor tool instance. This is important for previous undo/redo actions to be able to correctly perform their processing. The undo cleanup function typically frees any dynamically allocated data (user data) created for the undo entry, but can perform any "cleanup" action needed. The undo function is also responsible for implementing redo for the actions it undoes. This is described in "Redo" on page 108.

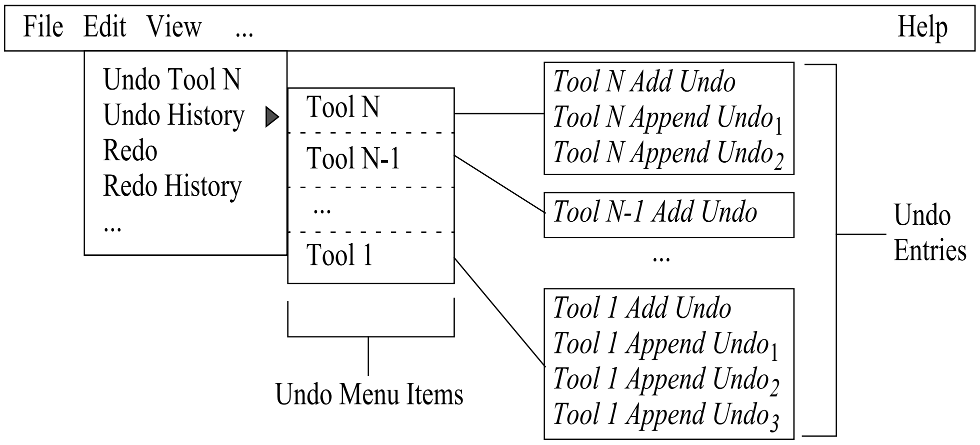
Typically, the undo function for an undo entry is called **before** the corresponding undo cleanup function is called. This happens when the undo menu item containing the undo entry is invoked by the user. Alternatively, the undo cleanup function may be called without the undo function having been called. This is the case whenever the undo menu item containing the undo entry in the **Edit>Undo History** submenu becomes unreachable. This can happen if the undo menu item is at the bottom of the stack when the number of undo menu items exceeds the **Max Undos** Editing preference or when a database is closed that still contains items in its **Edit>Undo History** submenu. When the number of undo menu items exceeds the limit, the menu item at the bottom of the **Edit>Undo History** submenu is removed. When a database is closed, all items in the **Edit>Undo History** submenu are removed. In either case, whenever an undo menu item is removed from the stack without having been undone, the undo cleanup function for each of the undo entries of that menu item is called and the undo function is not. For this reason, it is important to keep the **undo** actions in the *undo* function and the **cleanup** actions in the *undo cleanup* function. If, for example, you perform both the undo actions and the cleanup actions in your undo function (which may be tempting), you will likely introduce memory leaks into the system.

To add a new **`undo`** entry that is to be assigned its own menu item in the **Edit>Undo History** submenu, use the function **`mgEditorAddUndo.`** To append an undo entry onto an existing menu item, use the function **`mgEditorAppendUndo.`** Note that you can only append an undo entry onto a menu item that your editor tool instance created. As mentioned above, when you call either of these functions, you specify an undo function, an undo cleanup function and an optional user data pointer (that is passed to these functions) for the new undo entry. When you call **`mgEditorAddUndo,`** you also specify the text of the menu item that is to added to the **Edit>Undo History** submenu. This text is not required when you call **`mgEditorAppendUndo`** since the corresponding undo entry is appended onto an existing menu item and, therefore, is not visible to the user.

If the whole of the undo processing for your editor tool instance can be encapsulated by a single undo entry, call **`mgEditorAddUndo`** only. If it makes more sense or is more convenient for the undo processing for your tool instance to be segmented into multiple undo entries, call **`mgEditorAddUndo`** to add the first undo entry and **`mgEditorAppendUndo`** to add each successive undo entry. As a convenience, calling **`mgEditorAppendUndo`** without first calling **`mgEditorAddUndo`** is equivalent to calling **`mgEditorAddUndo.`** If you do this, the text of the corresponding menu item will be derived from the name of the editor tool.

It is acceptable for an editor tool instance to call **`mgEditorAddUndo`** multiple times while it is active. Remember, when you do this, you will create a new menu item in the **Edit>Undo History** submenu each time you call **`mgEditorAddUndo.`** This may be desirable, for example, if your editor tool instance performs multiple operations and wants the user to be able to undo each operation individually. Typically an editor tool dialog contains a "Next" push button to support this. When pressed, the tool finishes the current operation and begins the next. Many tools in Creator are implemented in this way, including those in the Face toolbox.

The following diagram shows the relationship between undo menu items in the Creator **Edit>Undo History** submenu and undo entries created by editor tool instances.



In this example, the user has invoked N editor tools, each of which has created one or more undo entries. Tool 1 has created four undo entries, Tool N-1 one undo entry and Tool N three undo entries. The order in which the undo entries are listed for each tool in the diagram above represents the order in which that tool created them. For example, Tool N *added* an undo entry, then *appended* two undo entries. Each undo entry has its own undo function and undo cleanup function assigned. When the user chooses to undo Tool N, the undo functions for the corresponding undo entries are called in order *Tool N Append Undo*$_2$, *Tool N Append Undo*$_1$ and finally *Tool N Add Undo*.

When an editor tool instance modifies the contents of a database, it must create at least one undo entry or notify Creator that its operation cannot be undone. If an editor tool informs Creator that its operation cannot be undone, all previous undo entries in the stack will be removed.

Note: If an editor tool instance modifies the contents of a database and neither adds an undo entry nor notifies Creator that its operation cannot be undone, the existing entries in the undo stack may become corrupt. This may lead to undefined results.

As mentioned above, there may be cases when an editor tool instance may not be able to undo the edit operation it performs. When this is the case, the tool instance can notify Creator of this by calling `mgEditorResetUndo.` Doing so causes the undo stack to be cleared of all entries. This is necessary because previous undo entries may depend on the database being in a particular state.

When an editor tool instance disrupts that state, those undo entries cannot be reliably performed. For this reason, it is highly recommended that every editor tool instance register undo actions diligently.

Redo

After the user chooses to undo an operation they perform in Creator, they can then redo that operation. Furthermore, after redo-ing an operation, the user can then choose to undo the redo, and so forth and so on.

In the previous section, it was described how editor tool instances implement undo by creating (adding or appending) undo entries for each undo-able action. Undo entries are created by an editor tool instance as it performs its action(s) while it is active. Similarly, an editor tool implements redo by creating a *redo entry* for each redo-able undo action. One redo entry is created for each undo entry that is invoked. In this way, there is a strict one to one correspondence between undo entries invoked (via their undo function) and redo entries created. As noted in the previous section, there is not necessarily a one to one correspondence between editor tool instances invoked and undo entries created. One editor tool instance can create one or more undo entries. Each undo entry (in its undo function) creates exactly one redo entry.

The undo function creates a redo entry using the function **`mgEditorAddRedo.`** Since it is required that the undo function call this function exactly once, there is no notion of *appending* a redo entry, only *adding*. Also, since there is a one to one correspondence between undo and redo entries, menu items in the **`Edit>Redo History`** submenu can be created automatically by Creator based on the order of the undo entries created. The text used for the redo menu items is also derived from the corresponding undo entries.

When the undo function adds a redo entry, it will provide two callback functions. The first is referred to as the *redo function*. The second is referred to as the *redo cleanup function*. You can also specify an optional opaque data pointer (*user data*) that will be passed to the redo and redo cleanup functions when they are called.

The redo function is responsible for restoring the database to the state it was in prior to the actions performed by the undo function. This is important for previous undo/redo actions to be able to correctly perform their processing. The redo cleanup function typically frees any dynamically allocated data (user data) created for the redo entry, but can perform any "cleanup" action needed. The redo function is also responsible for implementing undo for the actions it redoes. This is described in "Undo for Redo" on page 110.

Typically, the redo function for a redo entry is called *before* the corresponding redo cleanup function is called. This happens when the redo menu item containing the redo entry is invoked by the user. Alternatively, the redo cleanup function may be called without the redo function having been called. This is the case whenever the redo menu item containing the redo entry in the **Edit>Redo History** submenu becomes unreachable. This occurs when there are items in the **Edit>Redo History** submenu of a database and any of the following things happen:

- The user performs a new edit operation on the database.
- The user invokes an item in the **Edit>Undo History** submenu and any of the corresponding undo functions do not call **`mgEditorAddRedo`**.
- The database is closed.

In any of these situations, all items in the **Edit>Redo History** submenu are removed. Whenever a redo menu item is removed from the stack without having been redone, the redo cleanup function for each of the redo entries of that menu item is called and the redo function is not. For this reason, it is important to keep the "redo" actions in the redo function and the "cleanup" actions in the redo cleanup function. If, for example, you perform both the redo actions and the cleanup actions in your redo function, you will likely introduce memory leaks into the system.

As mentioned in the previous section, when an editor tool instance cannot undo the operation it performs, it is expected to call **`mgEditorResetUndo`**. This function is expected to be called explicitly because not all editor tool instances modify the database. There is no similar function you must call when your undo function cannot redo the operation it performs. This is because every undo function *is expected* to call **`mgEditorAddRedo.`** If your undo function does not call **`mgEditorAddRedo,`** the API detects this and assumes your undo function does not support redo. When this is the case, all previous redo entries will be removed from the **Edit>Redo History** submenu.

Undo for Redo

After the user chooses to redo an operation that was previously undone, the database should return to the state it was in just after the original editor tool instance completed. Given this, if the user then chooses to undo after redoing, the database should return to the state it was in just after the original undo was chosen.

Even though undoing a redo results in the database returning to the same state as undoing the original editor tool instance, the mechanism by which the editor tool instance implements undo in these distinct situations is slightly different.

In the previous section, it was described how undo functions implement redo by adding redo entries for each redo-able undo action. Redo entries are created by the undo function. In a similar fashion, an editor tool implements undo after redo by creating an *undo entry* for each undo-able redo action. One undo entry is created for each redo entry that is invoked. In this way, there is a strict one to one correspondence between redo entries invoked (via their redo function) and undo entries created. This correspondence is similar to that which exists between redo entries created and undo entries invoked. Each undo entry (in its undo function) creates exactly one redo entry. Similarly, each redo entry (in its redo function) creates exactly one undo entry.

The redo function creates an undo entry using the function **`mgEditorAddUndoForRedo`**. Since it is required that the redo function call this function exactly once, there is no notion of *appending* an undo entry after redo, only *adding*. Also, since there is a one to one correspondence between redo and undo entries, menu items in the **Edit>Undo History** submenu can be created automatically by Creator based on the order of the redo entries created. The text used for the undo menu items is also derived from the corresponding redo entries.

When the redo function adds an undo entry, it will provide two callback functions just as the editor tool instance does when it creates an undo entry. Those callbacks are the *undo function* and the *undo cleanup function*. You can also specify an optional opaque data pointer (*user data*) that will be passed to the undo and undo cleanup functions when they are called.

The undo function is responsible for restoring the database to the state it was in prior to the actions performed by the redo function. This is important for previous undo/redo actions to be able to correctly perform their processing. The redo cleanup function typically frees any dynamically allocated data (user data) create for the redo entry, but can perform any "cleanup" action needed. The undo function is also responsible for implementing redo for the actions it undoes.

In almost every case, you should pass the same undo and undo cleanup functions to **`mgEditorAddUndoForRedo`** that you passed to **`mgEditorAddUndo`** or **`mgEditorAppendUndo`** when the original undo entry was created by the editor tool instance. If you use the same undo and undo cleanup functions, and these functions operate correctly after the original undo, there is an excellent chance they will function correctly when they are invoked after redo. Implementing redo is described in "Redo" on page 108.

Sharing User Data in Undo/Redo

There is often a close relationship between the undo and redo functionality implemented by an editor tool instance. This relationship usually manifests itself in the user data specified for the undo and redo entries created. In many cases, the user data specified for a specific undo entry and its corresponding redo entry is the same data. When the user data specified to **`mgEditorAddUndoForRedo`**, **`mgEditorAddRedo`** and **`mgEditorAddUndo`** or **`mgEditorAppendUndo`** is the same address, the undo and redo cleanup functions are automatically deferred by Creator until this 'shared' data is really not needed anymore. When an undo or redo cleanup function is called, it is always safe (and required) to free the associated memory.

Undo Helper Functions

The kinds of operations that can be performed by an editor tool on a database vary widely. A common class of operations performed by an editor tool are those that modify the hierarchy of the database in some way. These operations include:

- Creating new nodes and attaching them in the hierarchy

- Deleting existing nodes from the hierarchy
- Moving existing nodes from one location in the hierarchy to another location

If your editor tool performs any of these common hierarchy operations, you can use the Undo Helper functions provided by the API to implement undo/redo functionality (as well as the associated undo/redo cleanup) for your tool. This can greatly reduce the amount of undo/redo support code you must write for your editor tool.

If your editor tool performs *only* these common hierarchy operations, you will not have to implement any undo/redo functions for your tool. Instead, you must simply call the appropriate undo helper function as described in this section.

The undo helper functions provided by the API are described in the following sections.

Undo Helpers for Creating Nodes

When an editor tool creates new geometry and attaches that new geometry into the scene, it can implement the undo/redo processing for this action using either of the undo helper functions **`mgEditorAddUndoForCreate`** or **`mgEditorAppendUndoForCreate`**. When calling either of these functions, you provide the node that *was created and attached* in the scene. If you call **`mgEditorAddUndoForCreate`**, you also specify the text of the menu item that is to be added to the **Edit>Undo History** submenu. If you create and attach multiple nodes in the scene, you call the corresponding undo helper function once for each node you create.

Note: You only have to call these functions for each "root" node you create. For example, if you create an object node with several child polygons, call the undo helper function only for the "root" object. These functions operate on the specified node and all their children automatically.

For **`mgEditorAddUndoForCreate`** and **`mgEditorAppendUndoForCreate`** to work correctly for a *node*, they must be called **after** *node* (and all its children) is created and attached into the scene. This is required so the information required by the undo/redo can be captured correctly.

After calling **`mgEditorAddUndoForCreate`** or **`mgEditorAppendUndoForCreate`** for a *node*, an undo entry is created. If you call **`mgEditorAddUndoForCreate`**, the undo entry is added in the **`Edit>Undo History`** submenu. If you call **`mgEditorAppendUndoForCreate,`** the undo entry is appended to the existing undo entry for the editor tool instance. In either case, when the corresponding undo entry is selected by the user, the specified *node* is un-created and removed from the scene and a redo entry is created in the **Edit>Redo History** submenu with the same label as that used for the undo entry. When this redo entry is selected, *node* will be re-created and reattached back into the scene, at which point another undo entry will be created. As long as the user selects undo/redo of this action, the proper actions will be automatically performed by the API with no further action required of the editor tool. As noted above, the undo/redo functions un-create and re-create *node* and all its children.

Both **`mgEditorAddUndoForCreate`** and **`mgEditorAppendUndoForCreate`** can be used in combination with other undo helper functions as well as **`mgEditorAddUndo`** and **`mgEditorAppendUndo`**.

Undo Helpers for Deleting Nodes

When an editor tool wants to delete geometry from the scene, it can implement the delete as well as the undo/redo processing for this action using either of the undo helper functions **`mgEditorAddUndoForDelete`** or **`mgEditorAppendUndoForDelete`**. When calling either of these functions, you provide the node that *is to be deleted* from the scene. If you call **`mgEditorAddUndoForDelete`**, you also specify the text of the menu item that is to be added to the **Edit>Undo History** submenu. If you want to delete multiple nodes in the scene, you call the corresponding undo helper function once for each node you want to delete.

Note: You only have to call these functions for each “root” node you want to delete. For example, if you want to delete an object node that has several child polygons, call the undo helper function only for the “root” object. These functions operate on the specified node and all their children automatically.

For **`mgEditorAddUndoForDelete`** and **`mgEditorAppendUndoForDelete`** to work correctly for a *node*, they must be called **instead of** deleting *node* from the scene. Again, these functions actually delete *node* and perform all functionality required for undo/redo automatically. This is different than the undo helper functions for node creation which require you to create and attach *node* before calling **`mgEditorAddUndoForCreate`** or **`mgEditorAppendUndoForCreate.`**

After calling **`mgEditorAddUndoForDelete`** or **`mgEditorAppendUndoForDelete`** for a *node*, *node* is deleted and an undo entry is created. If you call **`mgEditorAddUndoForDelete`**, the undo entry is added in the **Edit>Undo History** submenu. If you call **`mgEditorAppendUndoForDelete,`** the undo entry is appended to the existing undo entry for the editor tool instance. In either case, when the corresponding undo entry is selected by the user, the specified *node* is un-deleted and reattached in the scene and a redo entry is created in the **Edit>Redo History** submenu with the same label as that used for the undo entry. When this redo entry is selected, *node* will be re-deleted from the scene, at which point another undo entry will be created. As long as the user selects undo/redo of this action, the proper actions will be automatically performed by the API with no further action required of the editor tool. As noted above, the undo/redo functions un-delete and re-delete *node* and all its children.

Both **`mgEditorAddUndoForDelete`** and **`mgEditorAppendUndoForDelete`** can be used in combination with other undo helper functions as well as **`mgEditorAddUndo`** and **`mgEditorAppendUndo.`**

Undo Helpers for Moving Nodes

When an editor tool wants to move geometry from one attach point (parent) in the scene to another, it can implement the undo/redo processing for this action using either of the undo helper functions **`mgEditorAddUndoForMove`** or **`mgEditorAppendUndoForMove.`** When calling either of these functions, you provide the node that *will be moved* in the scene. If you call **`mgEditorAddUndoForMove`**, you also specify the text of the menu item that is to be added to the **Edit>Undo History** submenu. If you move multiple nodes in the scene, you call the corresponding undo helper function once for each node you move.

Note: You only have to call these functions for each “root” node you want to move. For example, if you move an object node that has several child polygons, call the undo helper function only for the “root” object. These functions operate on the specified node and all their children automatically.

For **`mgEditorAddUndoForMove`** and **`mgEditorAppendUndoForMove`** to work correctly for a *node*, they must be called **before** moving *node* in the scene. This is required so the information required by the undo/redo can be captured correctly. This is different than the undo helper functions for node creation and node deletion. For node creation, you call the helper functions after creating and attaching *node* . For node deletion, you call the helper functions instead of deleting *node*.

After calling **`mgEditorAddUndoForMove`** or **`mgEditorAppendUndoForMove`** for a *node*, an undo entry is created. If you call **`mgEditorAddUndoForMove`**, the undo entry is added in the **Edit>Undo History** submenu. If you call **`mgEditorAppendUndoForMove`**, the undo entry is appended to the existing undo entry for the editor tool instance. In either case, when the corresponding undo entry is selected by the user, the specified *node* is un-moved in the scene and a redo entry is created in the **Edit>Redo History** submenu with the same label as that used for the undo entry. When this redo entry is selected, *node* will be re-moved in the scene, at which point another undo entry will be created. As long as the user selects undo/redo of this action, the proper actions will be automatically performed by the API with no further action required of the editor tool. As noted above, the undo/redo functions un-move and re-move *node* and all its children.

Both **`mgEditorAddUndoForMove`** and **`mgEditorAppendUndoForMove`** can be used in combination with other undo helper functions as well as **`mgEditorAddUndo`** and **`mgEditorAppendUndo.`**

Making your Editor Tool Scriptable

Creator Script provides access to editor tools via OpenFlight Script. Editor tools that are "scriptable" are automatically available to the Creator user via Creator Script. If you want your editor tool to be available in Creator Script, you must follow the guidelines described in this section. Here is a brief outline of the steps you will take to make your editor tool scriptable. More details are included in the subsections that follow.

- Declare your tool as being *scriptable*. By default, editor tools are not scriptable. You must identify to Creator that your editor tool can be called from Creator Script. See "Declaring an Editor Tool as Scriptable" on page 116.

- Declare the data your tool needs to operate. If your tool normally displays a dialog to gather additional information from the user when it is invoked, you must describe that data to Creator so it can be collected as a “parameter block” from the user in Creator Script. See “Declaring the Parameter Block for a Scriptable Editor Tool” on page 117.
- Modify the start function of your tool to run *silently*. In Creator Script, dialogs should not be displayed..Instead, parameters to your tool are delivered via a *parameter block* passed to your start function. See “Making the Start Function for a Scriptable Editor Tool” on page 123.

Declaring an Editor Tool as Scriptable

By default, editor tools are not automatically accessible by Creator Script. You must explicitly declare your editor tool as being scriptable. You do this by specifying **`MG_TRUE`** for the **`MTA_SCRIPTABLE`** tool attribute when you register your tool with **`mgRegisterEditor.`** The following code fragment shows how a scriptable editor tool might be registered:

```
MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;
   mgmodulehandle moduleHandle = mgGetModuleHandle ( plugin );
   mgresource resource = mgLoadResource ( moduleHandle );
   mgpixmap pixmap = mgResourceGetPixmap ( resource, MY_BITMAP );

   pluginTool = mgRegisterEditor (
      plugin, "My Editor",
      EditorStartFunc, MG_NULL,
      MTA_VERSION, "1.0",
      MTA_PALETTELOCATION, MPAL_FACETOOLS,
      MTA_PALETTEICON, pixmap,
      MTA_SCRIPTABLE, MG_TRUE,   // identify this tool as "Scriptable"
      MG_NULL
      );
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

After you have identified your editor tool as being *scriptable*, you must tell Creator about the data your tool expects to receive. This is described in the next section Declaring the Parameter Block for a Scriptable Editor Tool below.

Declaring the Parameter Block for a Scriptable Editor Tool

As mentioned above, dialogs should not be displayed in Creator Script. If your editor tool normally displays a dialog to gather additional information from the user when it is invoked interactively, you must change how your tool behaves when invoked by Creator Script. Your tool must get its data in a different form. This section describes that mechanism.

When the Creator user invokes your editor tool in Creator Script, the user programmatically constructs an object known as a *parameter block* and passes that parameter block to your tool using **`mgExecute`.** The parameter block contains one or more *parameters*. Each parameter contains a different value intended for your tool.

In order for the user to know how to construct this parameter block in Creator Script, you must describe (to Creator) the data your tool expects. You do this when you register your tool. To do this, you will construct a default parameter block using **`mgNewParamBlock`** which will contain one or more parameters you define for your tool. You will define individual parameters in the parameter block using the following functions:

`mgParamAddInteger`	Adds a parameter of type **`integer`**
`mgParamAddFloat`	Adds a parameter of type **`float`**
`mgParamAddDouble`	Adds a parameter of type **`double`**
`mgParamAddBool`	Adds a parameter of type **`mgbool`** (boolean)
`mgParamAddString`	Adds a parameter of type **`char*`** (string)
`mgParamAddDouble2`	Adds a parameter of type **`double[2]`** (array)
`mgParamAddDouble3`	Adds a parameter of type **`double[3]`** (array)

Each of these functions adds a different "type" of parameter to the parameter block. To add an **`integer`** parameter, for example, use **`mgParamAddInteger`.** When you call any of these functions, you will specify the *name* of the parameter and a *default value* for that parameter. The name of the parameter is very important. It is a string that will be used to identify and access the parameter in Creator Script.

If you define a numeric parameter **`(integer, float or double)`**, you can optionally define minimum and maximum constraints for that parameter using the functions listed below. In this way, you can limit the range of values the user can set for these parameters. For example, if you define a parameter representing an “angle” (measured in degrees), you might find it useful to constrain its value to numbers between 0 and 359.

If your parameter is of type **`integer,`** use these functions to define constraints:

`mgParamSetIntegerMaxLE`	Defines the maximum value (inclusive) for an **`integer`** parameter - the parameter must be less than or equal to the value you specify
`mgParamSetIntegerMaxLT`	Defines the maximum value (exclusive) for an **`integer`** parameter - the parameter must be strictly less than the value you specify
`mgParamSetIntegerMinGE`	Defines the minimum value (inclusive) for an **`integer`** parameter - the parameter must be greater than or equal to the value you specify
`mgParamSetIntegerMinGT`	Defines the minimum value (exclusive) for an **`integer`** parameter - the parameter must be strictly greater than the value you specify

If your parameter is of type **`float`** or **`double`**, use these functions to define constraints:

`mgParamSetDoubleMaxLE`	Defines the maximum value (inclusive) for an **`float`** or **`double`** parameter - the parameter must be less than or equal to the value you specify

`mgParamSetDoubleMaxLT`	Defines the maximum value (exclusive) for a **`float`** or **`double`** parameter - the parameter must be strictly less than the value you specify
`mgParamSetDoubleMinGE`	Defines the minimum value (inclusive) for a **`float`** or **`double`** parameter - the parameter must be greater than or equal to the value you specify
`mgParamSetDoubleMinGT`	Defines the minimum value (exclusive) for a **`float`** or **`double`** parameter - the parameter must be strictly greater than the value you specify

Note: You can apply one or more of these functions simultaneously to your parameter. For example, to constrain an `integer` parameter to values greater than `0` and less than or equal to `10`, use the following:

```
mgParamSetIntegerMinGT (param, 0);
mgParamSetIntegerMaxGE (param, 10);
```

You can use an **`integer`** parameter to represent *numeric* (the default) or *enumerated* values. Enumerated **`integer`** parameters represent a set (may be non-contiguous) of discrete values, while numeric **`integer`** parameters represent a range of whole numbers. If your **`integer`** parameter is enumerated, use **`mgParamSetEnumerant`** to define each enumerant. An enumerant is comprised of a string value and its corresponding integer value. When a user accesses an enumerated **`integer`** parameter in Creator Script, they can specify the parameter value by string or by integer. Using the string value may be more readable in the user's script, while using an integer value may be more succinct. Both are equivalent.

By default, a parameter contains a single value. When a parameter is of type **`double2 or double3,`** however, that single value is actually comprised of two or three elements, respectively. These types are useful for 2 or 3 dimensional coordinate values. You might find **`double2`** useful for UV texture coordinates (u,v). Similarly, **`double3`** is useful for 3D positions (x,y,z) or vectors (i,j,k).

You can also define a parameter to contain an array of values. You do this by calling **`mgParamSetDimension.`** If your parameter has a fixed maximum dimension, pass that number to this function. For example, to define a parameter to be an array of at most 4 integer values, you would use:

```
param = mgParamAddInteger (block, "Name",
defValue);
mgParamSetDimension (param, 4);
```

If your parameter can have a variable number of values, pass `0` to **`mgParamSetDimension.`**

After you define the default parameter block for your editor tool, you give it to Creator using the function **`mgPluginToolSetDefaultParamBlock.`** When you do this, you specify the exact structure of the parameter block your editor tool expects to receive when it is invoked in Creator Script. Creator parses the data contained in this parameter block and uses it to validate parameters to your tool in Creator Script. Creator also adds this data to the **Help on Creator Script** window. This window is accessible to the user when writing Creator Script in the **OpenFlight Script Editor** window in Creator. The data displayed in this window helps the user set up the parameters to your tool.

The following code fragment shows how a scriptable editor tool might declare its default parameter block:

```
static void RegisterParamBlock ( mgplugintool pluginTool )

{

  mgparamblock paramBlock;

  mgparam param;


  // create the default param block for our editor tool

  paramBlock = mgNewParamBlock();


  // our tool has four parameters


  // first parameter is an enumerated integer with 3 possible values
```

```
   param = mgParamAddInteger ( paramBlock, “Options”, 2 );

   // ... possible values are these enumerants:
   mgParamSetEnumerant ( param, 1, “Option 1” );
   mgParamSetEnumerant ( param, 2, “Option 2” );
   mgParamSetEnumerant ( param, 3, “Option 3” );

   // second parameter is a boolean value
   param = mgParamAddBool ( paramBlock, “Checkbox”, MG_TRUE );

   // third parameter is a double value (angle between 0 and 90)
   param = mgParamAddDouble ( paramBlock, “Angle”, 45.0 );

   // ... angle must be greater than or equal to 0.0 ...
   mgParamSetDoubleMinGE ( param, 0.0 );
   // ... and less than 90.0
   mgParamSetDoubleMaxLT ( param, 90.0 );

   // final parameter is an array of 3D coords
   param = mgParamAddDouble3 ( paramBlock, “Coords”, 0.0, 0.0, 0.0 );
   mgParamSetMaxDimension ( param, 0 );

   // assign this param block as the default for this tool
   mgPluginToolSetDefaultParamBlock ( pluginTool, paramBlock );
}
```

```
MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv []
)
{
  mgplugintool pluginTool;
  mgmodulehandle moduleHandle = mgGetModuleHandle ( plugin );
  mgresource resource = mgLoadResource ( moduleHandle );
  mgpixmap pixmap = mgResourceGetPixmap ( resource, MY_BITMAP );

  pluginTool = mgRegisterEditor (
    plugin, "My Editor",
    EditorStartFunc, MG_NULL,
    MTA_VERSION, "1.0",
    MTA_PALETTELOCATION, MPAL_FACETOOLS,
    MTA_PALETTEICON, pixmap,
    MTA_SCRIPTABLE, MG_TRUE,   // identify this tool as "Scriptable"
    MG_NULL
    );

  // describe the data needed by Creator Script for this tool
  RegisterParamBlock ( pluginTool );

  return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

In this example listed above, the parameter block for our editor tool is defined to contain four parameters. The first parameter is an enumerated **`integer`** parameter named **`Options`** with three possible values. Its default value is **`2,`** which corresponds to the string **`Option 2.`** The second parameter is **boolean.** It is named **`Checkbox`** and is **`MG_TRUE`** by default. The third parameter is **`double`** and named **`Angle.`** It is constrained to be greater than or equal to **`0`** and less than **`90.`** Its default value is **`45`**. The final parameter is an array of 3D coordinates named **`Coords`**. The default value of each coordinate is **`(0.0, 0.0, 0.0)`**. The array parameter can contain an unlimited number of coordinates. Finally, when the default parameter block is complete, it is assigned to the editor tool using **`mgPluginToolSetDefaultParamBlock.`**

After you define this parameter block and assign it to your tool, the final step in making your editor tool ready for Creator Script is to modify its start function to operate correctly when invoked in Creator Script. This is described in the next section Making the Start Function for a Scriptable Editor Tool below.

Making the Start Function for a Scriptable Editor Tool

If your editor tool is *scriptable* (as described in the sections above) it may be invoked in Creator interactively or in Creator Script (non-interactively). The editor tool start function for a "scriptable" tool should be written in such a way to detect which mode it is being run and take appropriate action for each mode.

When your start function detects that your tool is being run interactively, your tool can display a dialog to solicit additional information from the user. See "Editor Tool Dialogs" on page 84 for more information.

When your start function detects that your tool is being run in Creator Script, your tool should not display a dialog. Instead, you should collect user parameters from the parameter block passed to your tool in the tool activation. This parameter block is built for your tool from the parameters set by the user in Creator Script. Your start function will read the data from the parameter block instead of displaying a dialog to gather the data. After reading the parameters in this way, your start function must do all the processing required for your tool before returning. This includes creating undo/redo entries for this invocation as described in "Undo/Redo" on page 103.

The following code fragment shows how a scriptable editor tool might detect how it is being invoked (interactively or in Creator Script) and take appropriate action accordingly:

```
static mgstatus EditorStartFunc ( mgplugintool pluginTool,
                    void* userData,
                    void* callData )
{
  mgeditorcallbackrec* editorData = (mgeditorcallbackrec*) callData;
  mgtoolactivation toolActivation = editorData->toolActivation;
  mgrec* topDb = mgGetActivationDb ( toolActivation );
  mgtoolactivationtype activationType = mgGetActivationType (
toolActivation );

  if ( OkToStartTool ( topDb ) ) {
    // tool instance can continue
    mytoolrec* instanceData = mgMalloc ( sizeof(mytoolrec) );
    instanceData->resource = resource;

    if ( activationType == MTAT_NORMAL ) {
      // tool is to run interactively

      // tell API dialog is needed to get parameters from user
      editorData->dialogRequired = MG_TRUE;
      editorData->toolData = instanceData;
    }
    else if ( activationType == MTAT_SCRIPT ) {
      // tool is to run in Creator Script
      // user parameters are passed in as parameter block from Creator Script


      // get the parameter block containing user-defined parameters for tool
      mgparamblock paramBlock = mgGetActivationParamBlock (
toolActivation );


      // get the parameter values from the parameter block
      // note that mytoolrec structure has fields for each of the parameters


      mgParamGetInteger ( paramBlock, “Options”, &instanceData->option, 2
);
      mgParamGetBool ( paramBlock, “Checkbox”, &instanceData-
>checkBox, MG_TRUE );
      mgParamGetDouble ( paramBlock, “Angle”, &instanceData->angle, 45.0
);


      // “Coords” is an array type parameter, get each item in the array


      // first, determine how many items are in the array parameter
      instanceData->numCoords = mgParamGetSize ( paramBlock, “Coords”
);


      // allocate space for this many coords
      coords = mgMalloc ( instanceData->numCoords * sizeof(mgcoord3d) );


      // get each coord, loading each into a slot in our allocated array
```

```
      for ( i = 1; i <= instanceData->numCoords; ++i )
      {
        mgParamGetDouble3Nth ( paramBlock, "Coords", i,
              &instanceData->coords[i].x,
              &instanceData->coords[i].y,
              &instanceData->coords[i].z,
              0.0, 0.0, 0.0 );
      }

      // now we have all the parameters we need from the user script
      // the data has been loaded into instanceData

      // DoProcessing does ALL the processing for this invocation here
      DoProcessing ( instanceData );

      // we don't need instanceData anymore, free it
      mgFree ( instanceData->coords );
      mgFree ( instanceData );

      // tell API dialog is NOT needed
      editorData->dialogRequired = MG_FALSE;
      editorData->toolData = MG_NULL;
    }
  }
  else {
      // tool instance cannot continue, dialog not needed
      editorData->dialogRequired = MG_FALSE;
   }

   return (MSTAT_OK);
}
```

In this modified example (the original example is in "Editor Tool Dialogs" on page 84) when the user launches "**`My Editor`**", the specified tool start function, **`EditorStartFunc,`** is called by the API. In the start function, a user provided function, **`OkToStartTool`** is invoked first to determine if the editor tool instance can continue. If the tool instance can continue, the start function then detects whether it is being invoked interactively or in Creator Script. If the start function detects that it is being run interactively, you will notice that it performs the same action as that of the original example. If the tool detects it is being invoked in Creator Script, you can see how it gathers user parameters from the parameter block provided and then does all the processing necessary in the start function before it returns.

Writing an Input Device

An input device plug-in tool is used to provide 3D vertex and 2D point input to tools in Creator. The most typical use of such a plug-in would be to support custom input devices such as the Microscribe 3D digitizer or any 2D digitizing tablet. When an input device plug-in tool is loaded into Creator, all existing editor tools in Creator will be able to receive data from the attached device transparently. This chapter describes how to write an input device plug-in tool and the protocols for reporting device data in such a way that any Creator tool can use the device data.

Declaring an Input Device

Declare an editor tool in your plug-in module's initialization function by calling **`mgRegisterInputDevice:`**

```
static mgstatus InputDeviceStartFunc ( mginputdevice inputDevice,
                                int inputFlags,
                                void* userData )
{
   // Start the input device based on the requested input flags
   return (MSTAT_OK);
}

static mgstatus InputDeviceStopFunc ( mginputdevice inputDevice,
                               void* userData )
{
   // Stop the input device
   return (MSTAT_OK);
}

static mgstatus InputDeviceSetInputTypeFunc ( mginputdevice inputDevice,
                                      int inputFlags,
                                      void* userData )
{
   // Notify input device of the type of input requested now
   return (MSTAT_OK);
}

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgplugintool pluginTool;

   pluginTool = mgRegisterInputDevice (
      plugin, "My Input Device",
      MMSI_VERTEXINPUT|MMSI_POINTINPUT|MMSI_DEVICEINPUT,
      InputDeviceStartFunc,
      InputDeviceStopFunc,
      InputDeviceSetInputTypeFunc,
      MG_NULL,
      MTA_VERSION, "1.0",
      MG_NULL
      );
   return (pluginTool ? MG_TRUE : MG_FALSE);
}
```

You specify the following *required* tool attributes:

Name - the name assigned to your input device.

Input Flags - input device capability flags.

Start Function - function called to notify your input device that input is desired.

Stop Function - function called to notify your input device that input is no longer desired

Input Type Function - function called to notify your device which type of input is desired.

User Data - user data that will be passed to your Start, Stop, and Input Type functions.

In addition to this required tool attributes, you can specify any of the following *optional* tool attributes:

Version - the character string version assigned to your input device.

Help Context - text that identifies a context sensitive help topic for your input device. See "Context Sensitive Help" on page 242 for a description of this attribute.

In the example listed above, the name of input device plug-in the tool is **`"My Input Device"`**. Its version is **`"1.0"`** and is capable of providing 3D vertex, 2D point and device specific data. **`InputDeviceStartFunc`** will be called when an editor tool in Creator requests any of these kinds of input. **`InputDeviceStartFunc`** will be called when the editor no longer needs device input. **`InputDeviceStartFunc`** *may* be called while the editor is active if the editor switches from requesting one type of input to requesting a different type. Since **`MG_NULL`** has been specified as the user data, no special user data will be passed to the callback functions.

The start, stop and input type functions return **`MSTAT_OK`** upon success. They should return non-zero device specific values to report errors depending on the nature of the device. This example will be referenced and expanded in subsequent sections as more input device tool functionality is discussed.

Input Devices in the Program Environment

Input device plug-in tools are activated (started) when editor tools request device input and are deactivated (stopped) when editor tools no longer require device input. Editor tools can request input from devices in the form of 3D vertex coordinates, 2D point coordinates, geometry selection and device specific data. Input device plug-in tools can provide some but not all of these input types. They can provide 3D vertex, 2D point and device specific data but cannot provide geometry selection input. For more details on how editor tools request and receive device input, see "Device Input" on page 92.

Editor tools are the only consumers of plug-in device input. For that reason, the way in which plug-in devices report input is very similar to the way in which editor tools receive it. For both 3D and 2D coordinate input, plug-in devices must report *start-continue-stop* event sequences, where one start event is reported, followed by zero or more continue events and then one stop event. In terms of a mouse style device, the start event would map to a button on the mouse being pressed at a given position. The continue event(s) would map to the device being moved while the button remains pressed. The stop event would map to the button being released. Each event in the sequence will include the current 3D or 2D position of the device at the time the event is reported.

The API will start a plug-in device by calling its start function. When calling this function, the API will specify which type of input (3D vertex or 2D point) is desired by the active editor tool. The API will only start those plug-in devices that have specified that they can provide the type of input desired. Devices that cannot provide the desired input type will not be started. If, while it is active, a plug-in device reports any type of data other than that requested, that data is ignored.

While a plug-in device is active, it is expected to report start-continue-stop event sequences as the user interacts with the physical device. Event reporting is described in "Sending Events" on page 133. While the editor tool is active, it may, at any time, request a different type of input. When this happens, the API *notifies* the plug-in devices as follows:

- For all active plug-in devices that support the new input type, the API will call the device's input type function to indicate that a new type of input is being requested.
- For all active plug-in devices that do not support the new input type, the API will call the device's stop function.
- For all inactive plug-in devices that support the new input type, the API will call the device's start function.

Each time the active editor tool requests a different input type, the steps outlined above are followed to ensure that only the proper plug-in devices are active. Finally when the active editor tool terminates, all active plug-in devices are also stopped. The API stops a plug-in device by calling its stop function.

Plug-in Device Handles

To uniquely identify input device plug-in tools registered, each input device plug-in tool is assigned a unique *input device handle.* An object of type **`mginputdevice`** is used to store the input device handle for each input device plug-in tool registered.

Since events are reported by plug-in devices in start-continue-stop event sequences, an arbitration strategy is enforced by Creator when multiple input devices try to report event sequences simultaneously. While a plug-in device is active, it must grab *focus* before it can begin sending an event sequence. After it sends the stop event of the sequence, the plug-in device automatically releases focus. The API keeps track of which input device handle has focus at any given time. As mentioned above, many input device handles are assigned (one per plug-in device) but only one ever has focus at any given time.

To grab focus the plug-in device must call **`mgInputDeviceGetHandle.`** If another device has focus when this function is called, it will return **`MG_NULL.`** This indicates that the plug-in device cannot send an event sequence now. If your plug-in device can grab focus successfully, **`mgInputDeviceGetHandle`** will return a pointer to your input device handle and you can begin sending an event sequence. Again, after you send the stop event of the sequence, your plug-in device automatically loses focus.

Button Devices

The API recognizes 128 virtual buttons for plug-in devices, numbered 0 through 127. When reporting 3D vertex or 2D point input, certain button numbers have special meanings:

Button	3D Vertex Input	2D Point Input
0	Maps to left mouse button	Maps to left mouse button
1	Maps to **`NEXT`** button (if present) on editor dialog	Maps to middle mouse button
2	No mapping	Maps to right mouse button

General editor tools are written to handle only the button mappings shown here. However, custom editor tools intended to be used with specific input devices can be written to take advantage of the other button device numbers. See "Device Specific Data" on page 135 for more information on how an editor tool can receive button device and other device specific information from a plug-in device.

Sending Events

When your input device plug-in tool grabs focus as described above, it can send an event sequence consisting of one start event, followed by zero or more continue events and terminated by one stop event. Each event in the sequence can be loaded (set) with 3D or 2D position data, button states and/or device specific data. To send an event, you first set the input device handle acquired by **`mgInputDeviceGetHandle`** with all relevant position, button state and device specific data and then send it. The following are the API functions that let you set various input data on the input device handle.

`mgInputDeviceSetVertex`	sets a 3D vertex with coordinates *x*, *y*, *z* on the input device.
`mgInputDeviceSetPoint`	sets a 2D point with coordinates *x*, *y* on the input device.

`mgInputDeviceSetButtonStatus`	sets the status of a button on the input device.
`mgInputDeviceSetDeviceData`	sets device specific data on the input device.

After all relevant data has been set on the input device handle, the event can be sent by calling **`mgInputDeviceSendEvent.`** The following code fragment extracted from a timer callback used to poll an input device shows how an input device plug-in tool might send events:

```
mginputdevice InputDevice = mgInputDeviceGetHandle ( pluginTool );
double x, y, z;
int i, j;
void* InputDeviceData;

// don't bother if another device has focus
if ( !InputDevice )
   return;

x = DeviceState ( X );
y = DeviceState ( Y );
z = DeviceState ( Z );

i = DeviceState ( I );
j = DeviceState ( J );

InputDeviceData = DeviceData ();

// set 3D vertex coordinates from the input device
mgInputDeviceSetVertex ( InputDevice, x, y, z );

// set 2D point coordinates from the input device
mgInputDeviceSetPoint ( InputDevice, i, j );

// set device specific data
mgInputDeviceSetDeviceData ( InputDevice, InputDeviceData );

// set the button status for button 0
switch ( Event ( LEFT ) )
{
   case DOWN:
      mgInputDeviceSetButtonStatus ( InputDevice, 0, MMSS_START );
      break;
   case ON:
      mgInputDeviceSetButtonStatus ( InputDevice, 0, MMSS_CONTINUE );
      break;
   case UP:
      mgInputDeviceSetButtonStatus ( InputDevice, 0, MMSS_STOP );
      break;
}
```

```
// set the button status for button 1
switch ( Event ( RIGHT ) )
{
   case DOWN:
      mgInputDeviceSetButtonStatus ( InputDevice, 1, MMSS_START );
      break;
   case ON:
      mgInputDeviceSetButtonStatus ( InputDevice, 1, MMSS_CONTINUE );
      break;
   case UP:
      mgInputDeviceSetButtonStatus ( InputDevice, 1, MMSS_STOP );
      break;
}

// send the event
mgInputDeviceSendEvent ( InputDevice );
```

n this example, the input device plug-in tool supports 3D vertex data, 2D point data and device specific data. Only if it can grab focus does it continue. Once it does grab focus, it queries the input device for the current 3D and 2D positions and device specific data and sets all this data on the input device handle. Even though it sets both 3D vertex and 2D point data on the input device handle, only the input type currently requested will be accepted. For example, if the current input type requested is 2D point input, the 3D vertex data set will be ignored. This example also queries the states of the two device specific buttons it supports, **`LEFT`** and **`RIGHT`** and maps those device specific buttons to devices 0 and 1 respectively. If the current input type requested is 2D point input, device 0 is mapped to the left mouse button and device 1 is mapped to the right mouse button when the event is reported to the active editor tool instance. If the current input type requested is 3D vertex input, device 0 is mapped to the left mouse button and device 1 is mapped to the **`NEXT`** button on the editor tool dialog (if present).

Device Specific Data

Custom editor tools can obtain device specific data using the functions **`mgInputDeviceGetButtonStatus`** and **`mgInputDeviceGetDeviceData`**. When an editor tool requests device specific input using **`mgEditorSetDeviceInputFunc`**, an object of type **`mgdeviceinputdata`** is passed which encapsulates the state of the input device. Useful information in **`mgdeviceinputdata`** includes the activating button and its state, device specific data and the name of the input device. These functions are discussed in more detail in “Device Input” on page 92.

GUI

The Tools API provides mechanisms for your plug-in tools to create and display Graphical User Interface (GUI) items in the context of Creator. Some GUI items are built automatically for your tool by Creator. Other GUI items you create and manage as needed during the course of the modeling session.

The GUI items that are created for you automatically include the Import and Export Database File Dialogs for database importer and exporter tools, the Read Pattern File Dialog for image importer tools, and menu items and toolbox buttons for viewer and editor tools. These GUI items are created by Creator based on the tool attribute parameters you specified when you registered the tool in your plug-in module's initialization function. During the modeling session, these GUI items are completely managed by Creator with no additional programming required of your plug-in.

The GUI items that your plug-in tool creates and manages during the modeling session include *dialogs*, *menu bars*, *controls, pixmaps, cursors*, and *string definitions*. A *dialog* is a window your plug-in tool creates to display and/or retrieve information from the user. A dialog can contain a *menu bar* which contains one or more submenus. Submenus contain individual menu items or nested submenus. A dialog may also contain one or more *controls* with which the user enters text, chooses options, or directs the action of the tool. Controls may display text (defined by *string definitions*) or images (defined by *pixmaps*). Note that menu items are also controls. A *cursor* is a picture that you can assign to track the movement of the mouse pointing device while your tool is active.

There is a simple hierarchical parent/child relationship between the GUI items your tool creates. Dialogs can parent other dialogs as well as controls. Controls cannot parent anything and are siblings to other controls parented by the same dialog. This hierarchical relationship does not apply to pixmaps, cursors or string definitions.

When you create a dialog, you must specify another dialog that is to be your dialog's parent. In most cases, you will specify **`MG_NULL`** as the parent. This will result in your new dialog being attached properly to the main Creator window. If you must create a nested dialog, that is a dialog that is parented by another dialog you have already created, you can do so, but you may not find it necessary.

This chapter describes how your plug-in tools can create, display, and interact with GUI elements in Creator.

Resource Files

You will define your GUI items in Standard Windows Resource Files. The API contains functions that know how to extract the definitions of your GUI items from these resource files and make them accessible programmatically to your plug-in. Once they have been extracted from the resource file, GUI items are represented within the API using an abstraction layer that is common to all platforms on which Creator runs.

You also need to supply a *resource header file* that contains the identifiers of the GUI items contained in the associated resource file. This header file must be included by any of your plug-in module's source code files that contain calls to API functions that access the GUI item identifiers contained therein. Even though this file can be named anything, it will be referred to as **`resource.h`** in the sample code fragments throughout this document.

On Windows, the API recognizes the standard Windows Resource File (.res) format. You will use a resource editor to create a resource definition file (.rc) which will be compiled and linked directly into your plug-in module.

On Windows: The program development environment you are using (e.g., Visual Studio) will likely include a resource editing utility and will handle the compilation of .rc files into your module.

Before your plug-in can access any of the GUI items contained in your resource file, you must load the resource file. And when finished with the resource file, you must unload it. Typically, you will load the resource file in the plug-in module initialization function and unload it in the module termination function as shown in the following code fragments:

```
#include "mgapiall.h"
#include "resource.h"

static mgresource Resource = MG_NULL;

MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgmodulehandle moduleHandle = mgGetModuleHandle ( plugin );
   ...
   Resource = mgLoadResource ( moduleHandle );
}

MGPIDECLARE(void) mgpExit ( mgplugin plugin )
{
   ...
   mgUnloadResource ( Resource );
}
```

After the resource file is loaded as shown above, your plug-in tools can use other API functions to extract GUI items from the resource file and display them as needed. How to extract and display GUI items is discussed in subsequent sections in this chapter.

GUI Abstraction Layer

When you layout your dialogs, menu bars and controls, you define them in your Windows Resource File. Remember, you define a Windows Resource File for either Windows or Linux platforms. When these items are extracted from the resource file, they are converted in two stages.

First, they are converted from the Windows format defined in the resource file to the native format defined by the platform on which you are executing. On Windows, this conversion step is not necessary since the Windows Resource format is the same as the Windows runtime format.

Next, they are converted from their native format to a common format defined by the Tools API GUI Abstraction Layer. In this way, GUI items can be accessed via a single abstraction layer independent of computer platform. The Tools API GUI abstraction layer defines a set of GUI item styles and behaviors common to all computer platforms on which Creator runs.

This abstraction strategy is known as a *common denominator* strategy. The advantage of this strategy is that it provides a portable set of functions to access your GUI items regardless of the computer platform. This allows you to write plug-ins that are easily ported from one platform to the other. The disadvantage of this strategy is that it cannot provide access to all capabilities of the GUI items in their native formats. This is due to the inherent differences in the native formats themselves. To address this weakness, the GUI abstraction layer does provide a non-portable mechanism to access the native formats of your GUI items. This mechanism is described later in this section.

This section introduces the GUI Abstraction Layer provided by the Tools API. The foundation of this layer is the set of abstract types used to represent GUI items.

```
typedef struct mggui_t* mggui;

typedef struct mgpixmap_t* mgpixmap;

typedef struct mgcursor_t* mgcursor;
```

The type **`mggui`** is used to represent both dialogs and controls, **`mgpixmap`** to represent pixmaps, and **`mgcursor`** to represent cursors. There is no abstract type defined for string definitions. They are represented simply as **`char*`**.

Use **`mgGetGuiHandle`** to access the native format of a **`mggui`** object.

On Windows: **`mgGetGuiHandle`** returns an object of type **`HWND`**.

Use **`mgGetPixmapHandle`** to access the native format of a **`mgpixmap`** object.

On Windows: **`mgGetPixmapHandle`** returns an object of type **`HBITMAP`** or **`HICON`** depending on the underlying format of the pixmap in the resource file.

Use **`mgGetCursorHandle`** to access the native format of a **`mgcursor`** object.

On Windows: **`mgGetCursorHandle`** returns an object of type **`HCURSOR`**.

GUI Identifiers

There are several API functions that provide access to GUI items by *identifier*. That is, you specify the identifier of the GUI item you want, and the function returns an object in terms of the GUI abstraction layer; either an **`mggui`**, **`mgpixmap`**, **`mgcursor`** or **`char*`** (as described in the previous section). For example, when you extract a dialog, pixmap, cursor or string definition from a resource file, specify the identifier of the item you want to extract. Similarly, when accessing a particular control within a dialog, specify the identifier of that control.

On Windows, GUI identifiers are integer values. Identifiers for dialog, pixmap, cursor, and string definition GUI items contained in a single resource file must be unique. That is, you cannot have two dialog definitions in the same resource file with the same identifier. This is enforced by most GUI layout tools that you will be using. GUI identifiers for controls in dialogs must also be unique per dialog. That is you cannot have two controls in the same dialog with the same identifier. You can, however, have two controls, each contained in a different dialog, with the same identifier. Most GUI layout tools enforce this rule but some do not. The Tools API depends on control identifiers to be unique per dialog. When the tool you are using does not enforce this uniqueness, you must take extra care to ensure that control identifiers are unique per dialog.

Dialogs Overview

Most plug-in tools use dialogs to prompt for additional information from the user. For example, an editor tool that creates *N-sided* polygons may use a dialog to retrieve the value of `N` from the user. Plug-in tools will also use dialogs to display information or options while the user works in another window. For example, a viewer tool that displays the bounding volume of the currently selected nodes in a database may display the bounding volume dimensions in a dialog during the modeling session. While the user works in the database window, the dialog remains on the screen and can be updated dynamically by the viewer tool as the selection changes. A plug-in tool that uses a dialog in this way will typically create it when the tool is first launched and continue to display it for the duration of the modeling session or until the user explicitly closes the dialog.

To support the different ways plug-in tools can use dialogs, the API provides two types of dialogs: modal and modeless. A modal dialog requires the user to supply information or cancel the dialog before allowing the application to continue. A modeless dialog allows the user to supply information and return to the previous task without closing the dialog.

A plug-in tool uses modal dialogs in conjunction with tools that require additional information before they can proceed. Modal dialogs are simpler to manage than modeless dialogs because they are created, perform their task, and are destroyed by calling a single API function. Though they are simpler, the use of modal dialogs should be limited to situations that absolutely require modality.

Modal dialogs are classified into two categories: *Convenience* and *User Defined*. Convenience dialogs are provided by the API to perform very simple modal user interaction and hide the dialog management tasks. User defined modal dialogs are dialogs whose contents are defined, laid out, and completely managed by the plug-in tool. They can contain the same kinds of controls that a modeless dialog can contain.

For all modeless and user-defined modal dialogs, you must define the content, layout and style of your dialog in your resource file. The definition of a dialog in this form is called a *dialog template*. The Tools API contains functions that read a dialog template from a resource file and create a *dialog instance* based on the definition contained in the template. The dialog instance is a new copy of the dialog. In this way, you can create one or more dialog instances from a single dialog template. Each dialog instance will be a separate copy of the same dialog template.

For these kinds of dialogs that are created from dialog templates, you must also supply a *dialog function* that the API calls when it has input for the dialog or tasks for the dialog to carry out. Convenience dialogs, because they are completely managed by the API, do not require you to create dialog templates in your resource file. Each style of dialog is described in the following sections.

Convenience Dialogs

Convenience dialogs come in two styles, *Message Dialogs* and *Prompt Dialogs*. Message dialogs contain a message and title, plus any combination of predefined icons and push buttons. They can be used to pose simple yes/no/cancel type questions. They can also be used to display important messages that require user attention before processing can continue.

Prompt dialogs also come in two styles, *Simple Prompt Dialogs* and *System Prompt Dialogs*. Simple Prompt Dialogs contain a message, a text field, and two push buttons **OK** and **Cancel**. They are used to prompt the user to enter a numeric or text value. System Prompt Dialogs are used to prompt the user to select a file, folder or color.

Message Dialogs

To display a message dialog, use **`mgMessageDialog`.** The following example displays a message dialog parented by the main Creator window. The title of the dialog is *Alert*. The message displayed within the dialog is “Do you want to continue?”. The dialog has three buttons, **Yes**, **No**, and **Cancel**. A question icon is also displayed in the dialog. The value returned by **`mgMessageDialog`** indicates which of the buttons was pressed by the user to dismiss the dialog. A return value of *1* indicates the **Yes** button, *2* means **No**, and *3* **Cancel**.

```
int response;
response = mgMessageDialog (
            MG_NULL, “Alert”,
            “Do you want to continue?”,
            MMBX_YESNOCANCEL | MMBX_QUESTION );
```

Simple Prompt Dialogs

Simple Prompt Dialogs prompt the user to enter a simple numeric or string value.

`mgPromptDialogInteger`	prompts the user to enter an integer value.
`mgPromptDialogFloat`	prompts the user to enter a float value.
`mgPromptDialogDouble`	prompts the user to enter a double value.
`mgPromptDialogString`	prompts the user to enter a string value.

The numeric prompt functions (**`mgPromptDialogInteger`**, **`mgPromptDialogFloat`** and **`mgPromptDialogDouble`**) all have the same form as shown in the following example:

```
mgstatus stat;
double val = 10.0;

stat = mgPromptDialogDouble ( MG_NULL, “Enter a value:”, &val );
if ( MSTAT_ISOK ( stat ) )
```

```
   // Ok button pressed, do something with val
else
   // Cancel button pressed, do something else
```

This example displays a prompt dialog parented by the main Creator window. The message text is 'Enter a value:'; there are two buttons, **OK** and **Cancel**. A text field is displayed into which the user may enter a double value. The initial value displayed in the text field is 10.0. If the user presses **OK**, the function returns `MSTAT_OK` and `val` contains the value entered by the user. If the user presses **Cancel**, the value in `val` is not defined. To prompt for integer or float numeric values, use the appropriate prompt function and substitute the corresponding type `int` or `float` for `double` in this example.

The text prompt function (`mgPromptDialogString`) has a slightly different form as shown in the following example:

```
mgstatus stat;
char string[40];

stat = mgPromptDialogString ( MG_NULL, 20, "Enter text:", string, 40 );
if ( MSTAT_ISOK ( stat ) )
   // Ok button pressed, str contains string entered, do something with it
else
   // Cancel button pressed, str not defined, do something else
```

This example displays a prompt dialog parented by the main Creator window. The message text is **Enter text:**; there are two buttons, **OK** and **Cancel**. A text field is displayed into which the user may enter a text value. The default text displayed in the text field is **Default**. The text field is wide enough to display approximately 20 characters. If the user presses **OK**, the function returns `MSTAT_OK` and `str` contains the value that the user entered. If the user presses **Cancel**, the value in `str` is not defined.

System Prompt Dialogs

System Prompt Dialogs allow the user to select a file, folder or color from a standard browser dialog.

`mgPromptDialogFile`	prompts the user to select one or more files.
`mgPromptDialogFolder`	prompts the user to select a folder.
`mgPromptDialogColor`	prompts the user to enter a color (RGB) value.

`mgPromptDialogTexture`	prompts the user to select one or more texture files.

The following code fragment demonstrates how you might use **`mgPromptDialogFile`** to prompt the user to select one or more files.

```
int numFilesSelected = 0;
mgstringlist filenames = MG_NULL;
mgstatus stat;

stat = mgPromptDialogFile (
   parent, MPFM_OPEN, &numFilesSelected, &filenames,
   MPFA_FLAGS, MPFF_MULTISELECT|MPFF_FILEMUSTEXIST,
   MPFA_PATTERN, "OpenFlight Files|*.flt || Text Files|*.txt",
   MPFA_TITLE, "OpenFlight Files or Text Files",
   MPFA_DIRECTORY, "C:/Presagis/Creator/gallery/models",
   MPFA_FILENAME, "a10s.flt",
   MG_NULL );

if ( MSTAT_ISOK ( stat ) ) {
   printf ( "Successful operation" );
   // free allocated memory when done with file names
   mgFreeStringList ( filenames );
}
```

The function **`mgPromptDialogFile`** uses the variable argument style as described earlier. The following parameters are *required*:

Parent - the dialog to parent the file browser dialog. If you specify **`MG_NULL`**, the main Creator window will be used as parent.

Mode - specifies whether the file selected will be used for the "open" or "save" operation.

Number of Files Selected - output parameter indicating how many files were selected by the user.

File List - output parameter containing the list of files selected by the user.

In addition to these required parameters, you can specify any of the following *optional* parameters that control the behavior or initial state of the file browser dialog:

Title - character string title for the file browser dialog.

Flags - integer mask specifying special behavior of the dialog.

Directory - character string specifying the initial directory to display in the dialog.

Filename - character string specifying the initial file selected in the dialog.

Full Filename - character string specifying the full path of the initial file selected in the dialog (this includes directory and file name).

Pattern - character string specifying the filter(s) to display in the dialog.

Note: If you specify Full Filename, you should not specify Directory or Filename. Doing so is ambiguous and the results are not defined.

You can also customize the dialog that is displayed by the file browser using the following *optional* parameters:

Dialog Id - GUI identifier of the dialog template to use for the file browser.

Resource - resource that contains the dialog template identified by Dialog Id.

Dialog Function - dialog function called by the API to notify your plug-in of significant dialog events.

Dialog Event Mask - specifies which dialog events to report to the Dialog Function.

User Data - user data passed to Dialog Function when it is called.

Custom dialogs are further described in subsequent sections of this chapter.

The example above displays a standard file browser dialog parented by the dialog `parent`. The dialog title is "`Open Flight Files or Text Files`", and the dialog displays two filters, "`*.flt`" and "`*.txt`". Each of these filters appear in an option menu that the user will select to "filter" for these types of files. The initial folder displayed in the dialog is "`C:/Presagis/Creator/gallery/models`" and the initial file selected is `a10s.flt.` The dialog allows more than one file to be selected. The user is only allowed to select files that do exist.

If the user selects a file (or files) and presses **OK**, the function returns `MSTAT_OK, numFilesSelected` contains the number of files selected, and `filenames` contains a list of all the files selected. When you are done using the file list, you must deallocate it using `mgFreeStringList.` If the user presses **Cancel**, the values in `numFilesSelected` and `filenames` are not defined.

The following code fragment demonstrates how you might use **`mgPromptDialogFolder`** to prompt the user to select a folder from the file system on the computer:

```
char* selectedFolder = MG_NULL;
mgstatus stat;

stat = mgPromptDialogFolder (
   parent, "Select a Folder",
   "C:/Presagis/Creator/gallery/models"
   &selectedFolder );

if ( MSTAT_ISOK ( stat ) ) {
   printf ( "Successful operation" );
   // free allocated memory when done with folder
   mgFree ( selectedFolder );
}
```

This example displays a standard folder browser dialog parented by the dialog **`parent.`** It has a title of **Select a Folder.** The initial folder displayed in the dialog is **`C:/Presagis/Creator/gallery/models`**. If the user selects a folder and presses **OK**, the function returns **`MSTAT_OK`** and **`selectedFolder`** contains a list of all the filenames selected. When you are done using the folder name, you must deallocate it using **`mgFree.`** If the user presses **Cancel**, the value in **`selectedFolder`** is not defined.

The following code fragment demonstrates how you might use **`mgPromptDialogColor`** to prompt the user to select a color:

```
mgstatus stat;
float r, g, b;

stat = mgPromptDialogColor ( parent, 255, 128, 0, &r, &g, &b );

if ( MSTAT_ISOK ( stat ) ) {
   printf ( "Successful operation" );
   // values in r, g, and b are valid
}
```

This example displays a standard color browser dialog parented by the dialog **`parent.`** The initial color displayed in the dialog has a red component of **`255,`** green **`128`** and blue **`0.`** If the user selects a color and presses **OK**, the function returns **`MSTAT_OK`** and **`r, g,`** and **`b`** contain the red, green and blue components of the selected color. If the user presses **Cancel**, the values in **`r, g,`** and **`b`** are not defined.

The function **`mgPromptDialogTexture`** is very similar to **`mgPromptDialogFile.`** The main difference is the file browser that is displayed. The file browser displayed by **`mgPromptDialogTexture`** includes a texture preview area. When the user selects a texture file in the file browser, the texture preview area displays a thumbnail view of the texture and lists some useful information (dimensions) about the texture selected.

Unlike **`mgPromptDialogFile,`** **`mgPromptDialogTexture`** does not allow you to supply a custom dialog. For that reason, the following optional input parameters are not supported for **`mgPromptDialogTexture:`**

`Dialog Id`

`Resource`

`Dialog Function`

`Dialog Event Mask`

`User Data`

If you specify any of these optional parameters they will be ignored. If you do not specify the optional parameter **`Pattern,`** **`mgPromptDialogTexture`** will automatically add the standard list of texture pattern extensions supported by Creator. If you do specify **`Pattern,`** you can override this default filter list with your own. In this case, the list you specify will be the only filters displayed in the file browser.

The Dialog Function

Both user-defined modal and modeless dialogs require dialog templates in your resource file to define their content. They also require a dialog function to be supplied that is called by the API to notify your plug-in tool of significant dialog events. Dialog events correspond to different stages in the life-cycle of a dialog. They include *initialize*, *show*, *refresh*, *size*, *hide*, and *destroy*. Dialogs are initialized and destroyed only once but may be shown, refreshed, and/or hidden multiple times over their life-cycles. You can specify which of these dialog events your dialog function is to be called for. In this way, you can choose to ignore some or all of the dialog events. Dialog events are fully described in "Dialog Events" on page 149. The following example shows the form of the dialog function:

```
static mgstatus DialogProc ( mggui dialog, mgdialogid dialogId,
                        mgguicallbackreason callbackReason,
                        void* userData, void* callData )
{
   switch ( callbackReason )
   {
      case MGCB_INIT:
         break;
      case MGCB_SHOW:
         break;
      case MGCB_REFRESH:
         break;
      case MGCB_SIZE:
         break;
      case MGCB_HIDE:
         break;
      case MGCB_DESTROY:
         break;
   }
   return (MSTAT_OK);
}
```

The parameters to the dialog function include the dialog itself, the dialog identifier, the callback reason (or dialog event) that triggered the callback, user data specified when you created the dialog, and call-specific data.

Depending on the callback reason, the object pointed to by the call-specific **`callData`** parameter varies. You can always safely cast this parameter to a pointer to an object of type **`mgguicallbackrec`** to determine the type of call data this object really points to. The only callback reason that sends meaningful data using this parameter is the size event, **`MGCB_SIZE.`** The call-specific data for each of the dialog events is described in "Dialog Events" on page 149.

The value returned by the dialog function is currently ignored but is reserved for future enhancement. In this version of the API, your dialog function should always return the value **`MSTAT_OK.`**

Dialog Events

As mentioned above, the life-cycle of a dialog is marked by dialog events. Dialog events are used to notify the plug-in that the state of the dialog has changed. The API *sends* dialog events by calling the dialog function for which the event is intended, specifying the corresponding dialog event (or callback reason). This section describes dialog states and events and explains the correlation between modeling session events, dialog events, and dialog states.

Dialog states are described below:

- Start: A dialog in this state has not yet been created.
- Hidden: A dialog in this state has been created but is not displayed on the screen.
- Displayed: A dialog in this state has been created and is displayed on the screen. It may be either minimized or maximized; the API does not distinguish between the two.
- End: A dialog in this state has been destroyed and is no longer accessible.

Dialog events are described below:

- *Initialize* **`[MGCB_INIT]:`** This dialog event is sent one time only when your dialog is created. This event is always the first event sent and is sent when the state of your dialog changes from *Start* to *Hidden*.

  For all dialogs, this event is sent by the API function that was used to create the dialog. For modal dialogs, the creation function is **`mgResourceModalDialog.`** For modeless dialogs, the creation function is **`mgResourceGetDialog.`** The dialog creation function sends this event before it returns.

- *Show* **`[MGCB_SHOW]`**: This dialog event is sent whenever the state of your dialog changes from *Hidden* to *Displayed*.

  For modal dialogs, this event is sent automatically (one time only) by **`mgResourceModalDialog.`**

  For modeless dialogs associated with editor tools, this event is sent automatically when the tool is launched. For all other modeless dialogs, this event is sent explicitly by the API function **`mgShowDialog.`**

  Note: The state of a dialog does not change when it is minimized or maximized. Nor does it change when a dialog becomes partially or fully occluded by another dialog or window.

- *Hide* **`[MGCB_HIDE]`**: This dialog event is sent whenever the state of your dialog changes from *Displayed* to *Hidden*.

  For modal dialogs, this event is sent automatically (one time only) by **`mgResourceModalDialog`** when the user dismisses the dialog.

  For modeless dialogs associated with editor tools, this event is sent automatically (one time only) when the tool is terminated. For all other modeless dialogs, this event is sent explicitly by the API function **`mgHideDialog.`** This event may also be sent automatically to a modeless dialog if the modeling session ends while the dialog is displayed or if the dialog's parent is explicitly hidden.

Note: The state of a dialog does not change when it is minimized or maximized. Nor does it change when a dialog becomes partially or fully occluded by another dialog or window.

- *Size* **`[MGCB_SIZE]`**: This dialog event is sent whenever the user changes the size of your dialog. Users can change the size of your dialog by dragging the resize handles of your dialog or by clicking either the minimize or maximize box on your dialog's title bar. This event does not mark a state change for your dialog.

  The call-specific data parameter passed to the dialog function when this event is sent points to an object of type **`mgdialogsizecallbackrec.`** This record contains a flag that indicates whether or not the dialog is being minimized. If the dialog is not being minimized, you can determine the new size of the dialog using the functions described in "Sizing and Positioning Gui Items" on page 165.

  Resizable dialogs are discussed in more detail in "Resizable Dialogs" on page 154.

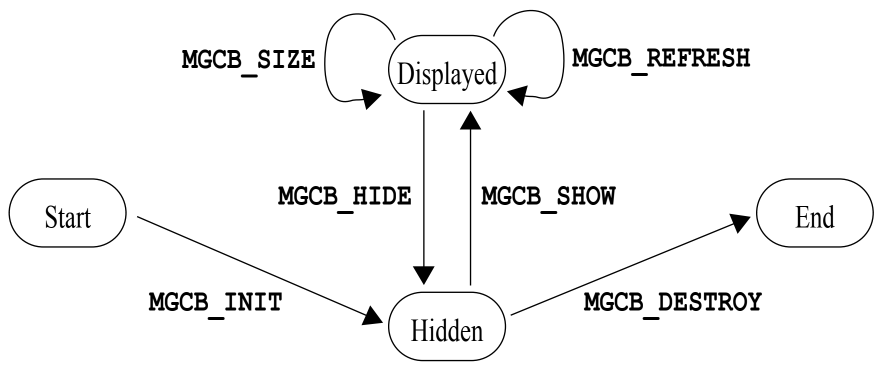
- *Refresh* **`[MGCB_REFRESH]`**: This dialog event is sent whenever the contents of the dialog need to be updated. This event does not mark a state change for your dialog. This event is sent automatically when the state of a dialog changes from *Hidden* to *Displayed* (that is immediately following a **`MGCB_SHOW`** event). It also occurs when your dialog is explicitly refreshed using **`mgRefreshDialog.`** This event will be discussed in more detail in "Control Events" on page 160.

- *Destroy* **`[MGCB_DESTROY] :`** This dialog event is sent one time only when your dialog is destroyed. This event is always the last event sent for a dialog.

  For modal dialogs, this event is sent automatically by **`mgResourceModalDialog`** before it returns (after the user dismisses the dialog).

  For modeless dialogs associated with editor tools, this event is sent automatically when the tool terminates. For all other modeless dialogs, this event is sent explicitly by the API function **`mgDestroyDialog.`** This event may also be sent automatically to a modeless dialog if the modeling session ends while the dialog is displayed or if the dialog's parent is explicitly destroyed.

The figure below describes dialog state transitions. The dialog states are represented by the ovals. The transitions between the dialog states (the dialog events) are represented by the arrows between the states.



User Defined Modal Dialogs

To display a user defined modal dialog, use **mgResourceModalDialog** as shown in the following example:

```
static mgbool GetAnswer ( mgresource resource )
{
   int whichButton;

   whichButton = mgResourceModalDialog (
                    MG_NULL, resource, DIALOGID,
                    MGCB_INIT|MGCB_SHOW|MGCB_REFRESH|
                    MGCB_HIDE|MGCB_DESTROY,
                    DialogProc, MG_NULL );

   return (whichButton == 1 ? MG_TRUE : MG_FALSE);
}
```

In this example, **mgResourceModalDialog** creates and displays a modal dialog instance defined by a dialog template extracted from the loaded resource file identified by **resource.** The identifier of the dialog template within the resource that defines the contents of the dialog is **DIALOGID.** Since the dialog parent has been specified as **MG_NULL,** the parent of the dialog is the main Creator window. The dialog function, **DialogProc** is called for all dialog events and is not passed any user data.

The dialog remains active and all other interactions with the application are disallowed until the user dismisses the dialog. The user can only dismiss the modal dialog by pressing one of two predefined buttons or clicking the close icon on the dialogs title bar. Your modal dialog must include at least one of these items so that the user can successfully dismiss the dialog. If it does not, and the user cannot dismiss the dialog, the application will hang. These two buttons are expected to have identifiers **`MGID_OK and MGID_CANCEL.`** These identifiers are defined correctly for both platforms in the include file **`mgapires.h.`**

The function **`mgResourceModalDialog`** returns a value of **`1`** if the user dismisses the dialog by pressing the button with the identifier **`MGID_OK.`** A value of **`2`** is returned if the button with the identifier **`MGID_CANCEL`** is pressed or if the user clicked the close icon on the title bar. In the example shown above, **`GetAnswer`** returns **`MG_TRUE`** if the user presses the **`MGID_OK`** button.

Modeless Dialogs

To display a modeless dialog, use **`mgResourceGetDialog`** and **`mgShowDialog`** as shown in the following example:

```
static mgbool StartDialog ( mgresource resource )
{
   mggui dialog;

   dialog = mgResourceGetDialog (
                  MG_NULL, resource, DIALOGID,
                  MGCB_INIT|MGCB_DESTROY,
                  DialogProc, MG_NULL );

   mgShowDialog ( dialog );
   return (dialog ? MG_TRUE : MG_FALSE);
}
```

In this example, **`mgResourceGetDialog`** creates a modeless dialog instance defined by a dialog template extracted from the loaded resource file identified by **`resource. DIALOGID`** is the identifier of the dialog template within the resource that defines the contents of the dialog. Since the dialog parent has been specified as **`MG_NULL,`** the parent of the dialog is the main Creator window. The dialog function, **`DialogProc`** is called only for **`MGCB_INIT and MGCB_DESTROY`** dialog events and will not be passed any user data. The call to **`mgShowDialog`** causes the dialog to be displayed on the screen.

Dialog Titles

Dialogs can display a title string that appears within the top border of the dialog. The function, **`mgSetTitle,`** sets the title to a specified text string value. To retrieve the current title of a dialog, use **`mgGetTitle.`** On Windows, the initial title for a dialog is specified in the resource editor.

Note: You should not use **`mgSetTitle`** to set the title for an editor tool dialog. Titles for editor tool dialogs are managed by the API automatically to display the name of the editor tool as well as the name of the database for which the editor was launched.

Resizable Dialogs

When a dialog instance is created its initial size and position is defined in the resource file. In most cases, the size of a dialog will not need to change during its life-cycle. However, there may be times when you will want to allow the user to change the size of your dialog to suit his or her needs. For example, you may implement a viewer tool that renders a portion of the database in a GL control contained in your dialog. Since rendering speed is often affected by viewport size, you may want to allow the user to grow or shrink the dialog depending on the complexity of the scene being drawn.

To allow your dialog to be resizable, you must define the proper platform specific window decorations for your dialog in the resource file. For Windows, the window styles **`WS_THICKFRAME`**, **`WS_MINIMIZEBOX`** and **`WS_MAXIMIZEBOX`** applied to a dialog allow the user to resize, minimize and maximize the dialog, respectively.

Though you can define a dialog to be resizable in the resource file, not all resource file formats allow you to define minimum and maximum dimensions for your dialog. For that reason, the API provides a function that lets you specify just how small and how large the user can size the dialog. After a dialog instance has been created, the function **`mgDialogSetAttribute`** can be used to define limits on the dimensions your dialog can be sized. This function accepts two parameters. The first parameter is of enumerated type **`mgdialogattribute`** and indicates which dimension (or dialog attribute) you are specifying in the second parameter. The second parameter is of type **`int`** and specifies the attribute value you want to set. The following list defines the dialog attributes whose values can be set by this function.

- *Minimum Width* **`[MDA_MINWIDTH]`**: The value of this integer attribute defines the minimum width, in screen pixels, that a dialog can be sized.
- *Maximum Width* **`[MDA_MAXWIDTH]`**: The value of this integer attribute defines the maximum width, in screen pixels, that a dialog can be sized.
- *Minimum Height* **`[MDA_MINHEIGHT]`**: The value of this integer attribute defines the minimum height, in screen pixels, that a dialog can be sized.
- *Maximum Height* **`[MDA_MAXHEIGHT]`**: The value of this integer attribute defines the maximum height, in screen pixels, that a dialog can be sized.
- *Resize Width* **`[MDA_RESIZEWIDTH]`**: The value of this boolean attribute defines whether or not the width of a dialog is resizable. A value of **`1`** allows the dialog width to be resized.
- *Resize Height* **`[MDA_RESIZEHEIGHT]`**: The value of this boolean attribute defines whether or not the height of a dialog is resizable. A value of **`1`** allows the dialog height to be resized.

Note: If a dialog is not defined as resizable in the resource file, setting these attributes will have no effect.

Functions to set the size and position of a dialog are described in "Sizing and Positioning Gui Items" on page 165.

Controls

Controls are specific user interface elements within a dialog with which the user interacts. When a dialog instance is created, the controls it contains become accessible to your plug-in via the GUI abstraction layer introduced in "GUI Abstraction Layer" on page 139. This layer defines and provides portable access to a set of GUI control classes.

Following are the GUI control classes defined by the Tools API GUI Abstraction Layer:

- *Text*: A text control lets the user view and/or edit text.

- *Push Button*: A push button is a rectangle containing a caption or a pixmap that indicates what the button does when the user selects it.
- *Toggle Button*: A toggle button can be either checkbox or radio button style. Both styles can have a caption or pixmap indicating a choice the user can make by selecting the button. Radio buttons are displayed in groups that permit the user to choose from a set of related mutually exclusive options.
- *List*: These controls display a list from which the user can select one or more items.
- *Option Menu*: These controls display a drop down list from which the user can select a single item.
- *Scale*: A scale control contains a slider whose position represents a discrete value within a defined range.
- *Progress*: A progress control is used to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled, from left to right, with a colored bar as an operation progresses. Progress controls do not allow user interaction.
- *GL*: A GL control is a rectangular region that is used to perform application specific "drawing" using OpenGL commands. GL controls can also be configured to receive mouse input from the user.
- *Spin*: A spin control consists of an up/down arrow pair that the user can press and hold to get continuous push button like behavior.
- *Menu Item*: A menu item control is very similar to a push button or toggle button control. But unlike the push button and toggle button controls which are typically located in the main part of your dialog, menu item controls are contained in a menu bar or submenu on your dialog.

It is important to understand the relationship between native control classes and the abstract control classes defined by the Tools API. The native control classes are those defined by the platform-specific resource file format you are using to layout your dialog. How you define a control in your resource file dictates which abstract control class it maps to when it is created.

When you define your dialog in your resource file, In theWindows Resource Model (which is used by the OpenFlight API on Windows), the native GUI item is called a *window*. A *window* has a *class* name and one or more *style* attributes. Following is a table showing the mapping between abstract control classes and *window class/style* attributes for Windows. In order for your control to be recognized and accessible as one of the abstract control classes, it must be defined using the *window class* and *style* listed in the table for the corresponding abstract class. If it is not, the Tools API will not recognize the control within the dialog.

Control Class	Window Class	Window Style	Remarks
Text	"Edit"	N/A	For editable text fields
	"Static"	N/A	For static labels
Push Button	"Button"	BS_PUSHBUTTON	For non-default push button
	"Button"	BS_DEFPUSHBUTTON	For default push button
Toggle Button	"Button"	BS_CHECKBOX	For check boxes
	"Button"	BS_AUTORADIOBUTTON	For radio buttons
List	"ListBox"	N/A	For single Select lists
	"ListBox"	LBS_EXTENDEDSEL	For multi-select lists
Option Menu	"ComboBox"	CBS_DROPDOWNLIST	For all option menus
Scale	"msctls_trackbar32"	N/A	For all scales
Progress	"msctls_progress32"	N/A	For all progress controls
GL	"GLControl"	N/A	For all GL controls
Spin	"msctls_updown32"	UDS_ARROWKEYS	For all spin buddies
Menu Item	N/A	N/A	Menu item controls are created as part of the menu bar you define in your Windows Resource File.

After you create a dialog instance that contains one or more controls, you can access a specific control within that dialog using the function `mgFindGuiById` as shown in the following code fragment:

```
static mgbool StartDialog ( mgresource resource )
{
   mggui dialog;
   mggui control;

   dialog = mgResourceGetDialog (
                  MG_NULL, resource, DIALOGID,
                  MGCB_INIT|MGCB_DESTROY,
                  DialogProc, MG_NULL );

   control = mgFindGuiById ( dialog, CONTROLID );

   mgShowDialog ( dialog );
   return (dialog ? MG_TRUE : MG_FALSE);
}
```

In this example, **`mgFindGuiById`** returns the control contained in **`dialog`** whose identifier is **`CONTROLID.`**

Buddy Controls

Usually, controls act independently within a dialog. In some cases, however, a control can be *attached* to another control in the dialog to modify or enhance the behavior of the control to which it is attached. For example, a text control could be attached to a scale control. This would allow the user to type in explicit values for the scale in addition to dragging the slider to change its value. Controls that are attached to other controls in this way are called *buddy* controls. Controls that have buddy controls attached to them are referred to as *main* controls. When one or more buddy controls are attached to a main control, the group of controls behave as one single control.

Of course you can implement this kind of behavior to connect two or more controls in your plug-in dialog code programmatically, but to make it much easier for you, the API does provide several convenience functions. The API contains functions that allow you to easily connect buddy controls to either text or scale controls. The API supports spin buddy controls attached to text controls. It also supports spin or text buddy controls attached to scale controls. The functions used to connect buddy controls to text controls are discussed in "Text Buddy Controls" on page 173. The functions that connect buddy controls to scale controls are discussed in "Scale Buddy Controls" on page 183.

The Control Callback Function

After you create a dialog instance that contains one or more controls, you then define the behavior of the controls. For example, you may define the action that is to be taken when a user presses a push button control or enters a value into a text control. In addition, you may specify how values are displayed in your controls. To do this, supply a *control callback function* that is called by the API to notify the plug-in tool of significant control events. The contents of this callback function define the behavior of your control.

Just as dialog events mark significant events for dialogs, control events are used to notify the plug-in tool when something significant has happened to a control. Control events include *activate*, *refresh,* and *draw*. Not all control events are sent to all control classes. For example, the GL control only receives the draw control event. It does not receive either the activate or refresh control event. Buddy controls do not receive control events. All user interaction with buddy controls generate control events for the control to which the buddy control is attached. Specifics like these will be discussed in the sections that describe the individual control classes later in this chapter.

Controls may receive control events multiple times over their life-cycles. You can specify which of the control events your control callback function is to be called for. In this way, you can choose to ignore some or all of the control events. Control events are described in "Control Events" on page 160. The following shows the form of the control callback function:

```
static mgstatus ControlCallback ( mggui gui, mgcontrolid controlId,
                            mgguicallbackreason callbackReason,
                            void *userData, void *callData )
{
   switch ( callbackReason )
   {
      case MGCB_ACTIVATE:
         break;
      case MGCB_REFRESH:
         break;
      case MGCB_DRAW:      // GL Controls only
         break;
   }
   return (MSTAT_OK);
}
```

The parameters to the control callback function include the control itself, the control identifier, the callback reason (or control event) that triggered the callback, user data specified when you assigned the callback, and call-specific data. Depending on the callback reason, the object pointed to by the call-specific **`callData`** parameter varies. You can always safely cast this parameter to a pointer to an object of type **`mgguicallbackrec`** to determine the type of call data this object really points to. Several control events send event specific data using this parameter. The contents of this data is discussed in the sections that describe the individual control classes later in this chapter.

Note: The value returned by the control callback function is currently ignored but is reserved for future enhancement. In this version of the API, your control callback function should always return the value **`MSTAT_OK.`**

Control Events

Control events are described below:

- *Activate* **`[MGCB_ACTIVATE]:`** This control event is sent whenever the user *activates* the control. This event maps to different user actions for each of the abstract control classes. See below for more details on when this event is sent to the different control classes.
- *Refresh* **`[MGCB_REFRESH]:`** This control event is sent whenever the contents of the control need to be updated. This event is sent automatically when the state of the dialog that contains the control changes from *Hidden* to *Displayed*. It also occurs when the dialog that contains the control is explicitly refreshed using **`mgRefreshDialog`** or when the control itself is explicitly refreshed using **`mgRefreshControl.`** This control event is sent to menu item controls when the user "opens" the submenu containing the menu item. The GL control does not receive this event.
- *Draw* **`[MGCB_DRAW]:`** This control event is sent only to GL controls. It is sent when the GL control needs to be redrawn. This event is sent automatically by the window system whenever the contents of the GL control has become invalid. This includes when the state of the dialog that contains the control changes from *Hidden* to *Displayed* and when any portion of the GL changes from being occluded to being visible. You can also explicitly cause a GL control to be redrawn by calling **`mgDrawControl.`**

To assign a control callback function to a control, use the function **`mgSetGuiCallback`** as shown in the following code fragment:

```
static void InitializeControlCallbacks ( mggui dialog )
```

```
{
   mggui gui;

   gui = mgFindGuiById ( dialog, CONTROLID );
   mgSetGuiCallback ( gui, MGCB_ACTIVATE|MGCB_REFRESH,
                              ControlCallback, MG_NULL );
}
```

In this example, the control whose identifier is **`CONTROLID`** is assigned the control callback function **`ControlCallback.`** The callback function is called for the control events, **`MGCB_ACTIVATE`** and **`MGCB_REFRESH,`** and is not passed any user data.

The following describes when the activate control event is sent for each of the control classes supported by the API.

- *Push Button* - Sent when the user clicks the button.
- *Text* - If the text control is not a buddy control attached to another control, the activate control event is sent to it when the user enters a new value into the text control and presses the enter or tab key or after the text control loses focus. If the text control has a spin buddy control attached to it, it will also receive an activate control event when the user activates the spin control.
  If the text control is a buddy control attached to a scale control, the activate control event is actually sent to the scale control, not to the text control.
- *Toggle Button* - Sent when the user clicks the control.
  Note: For radio button style controls, the activate control event is only sent to the control in the group that changed from *clear* to *set*.
- *List* - Sent when the user selects an item in the list.
- *Option Menu* - Sent when the user selects an item in the option menu.
- *Scale* - Sent when the user changes the value of the control by dragging the slider or manipulating any of the text or spin buddy controls attached to the scale.
- Menu Item - Sent when the user selects an item in a submenu of a menu bar.
- Progress - The activate control event is not sent to a progress control.
- GL - The activate control event is not sent to a progress control.
- Spin - The activate control event is not sent to a spin control. Since spin controls can only exist as buddy controls attached to other

controls, the activate control event is sent to the control to which the spin buddy control is attached.

A couple of important things stated above warrant reemphasis here. First, the activate control event is not applicable for GL or progress controls. Second, for buddy controls that are attached to other controls, the activate control event triggered by the interaction with the buddy control is always sent to the control to which the buddy is attached (not to the buddy control).

Built-in Control Callback Functions

The API provides several built-in control callback functions to perform common callback operations for controls in your dialog. Built-in control callback functions are functions defined in the API that you can assign to controls in your dialogs. Since they are defined in the API, they can save you from having to create your own functions. They are declared as functions of type **`mgguifunc,`** so you can pass them directly to **`mgSetGuiCallback.`** Consider the common situation in which a dialog contains a **Close** push button control that when activated simply hides the dialog. To do this, you would normally define a control callback function and assign it the "Close" push button as follows:

```
static mgstatus CloseCallback ( mggui gui, mgcontrolid controlId,
    mgguicallbackreason callbackReason,
    void* userData, void* callData)
{
   if ( callbackReason == MGCB_ACTIVATE )
      mgHideDialog (gui);
   return (MSTAT_OK);
}

static void InitializeControls ( mggui dialog )
{
   mggui gui = mgFindGuiById ( dialog, IDC_CLOSE );
   mgSetGuiCallback ( gui, MGCB_ACTIVATE, CloseCallback, MG_NULL );
}
```

In this example, an explicit control callback function, **`CloseCallback`** is defined that performs the very simple operation of hiding the dialog that contains the **Close** push button control. To make this simpler to implement, the API provides a built-in control callback function, **`mgHideDialogCallback,`** designed to do exactly the same function as **`CloseCallback.`** Using it, the code shown above could be simplified as follows:

```
static void InitializeControls ( mggui dialog )
{
   mggui gui = mgFindGuiById ( dialog, IDC_CLOSE );
   // Note: You must select the MGCB_ACTIVATE event.
   //       If you select any other events, they are ignored.
   //       You must pass MG_NULL as the user data
   mgSetGuiCallback ( gui, MGCB_ACTIVATE, mgHideDialogCallback, MG_NULL );
}
```

In this simplified example, the built-in control callback function, **`mgHideDialogCallback`** is assigned to the **Close** push button control. Doing so saves the developer from defining a separate function to perform this simple operation.

The API provides the following built-in control callback functions:

`mgHideDialogCallback`	When the activate control event is sent, calls **`mgHideDialog`**. Ignores all other control events.
`mgDestroyDialogCallback`	When the activate control event is sent, calls **`mgDestroyDialog`**. Ignores all other control events.
`mgRefreshDialogCallback`	When the activate control event is sent, calls **`mgRefreshDialog`**. Ignores all other control events.

Note: You must select the activate control event when assigning built-in control callback functions to your controls. If you do not, the activate control event will not be sent to the control, and as a result, the control callback function will not be called.

General Control Functions

As mentioned earlier, the API provides a cross-platform abstraction layer to access and manipulate GUI elements, such as dialogs and controls. This and subsequent sections in this chapter focus on the functions in this layer that provide access to controls. The functions in this layer are categorized as *general* or *specific*. General functions are those that apply to more than one class of control while specific functions pertain to a particular control class.

An example of a general control function would be **`mgSetVisible`** which hides or displays any kind of control (except menu item controls). A specific function is one like **`mgListAddItem`** which adds a string to a list control.

This section describes the general control functions of the API. Unless specifically noted, general control functions apply to all control classes.

Controls in dialogs can exist in different states. They may be *visible* or *hidden* as well as *enabled* or *disabled*. A control that is visible is displayed while one that is hidden is not. Hidden controls are still there; they just cannot be seen. A control that is enabled accepts user input; while one that is disabled will not. Typically, disabled controls are displayed in such a way that indicates to the user the control is not sensitive to user input.

The initial state of a control is specified in the resource editor. Within the properties page for a particular control, you can specify whether the control is initially visible or hidden, enabled or disabled.

Programmatically, you can manipulate the state of a control using the following functions:

`mgSetVisible`	displays or hides a control (except menu item controls).
`mgSetEnabled`	enables or disables a control.
`mgIsVisible`	determines if a control is visible.
`mgIsEnabled`	determines if a control is enabled.
`mgSetFocus`	sets the keyboard focus to the specified control.

Note: You cannot change the visibility state of menu item controls.

Push button and toggle button controls can be configured to display either a caption or a pixmap. Static text and menu item controls can only display a caption. A caption is a text string and a pixmap is an image. For push button and toggle button controls, captions and pixmaps are used within the control to convey the action that will be performed when the control is activated. For static text controls, captions are used to convey the state of a particular user interface component or simply to add aesthetics to a dialog. You can use window class `STATIC` with static style `SS_BITMAP or SS_ICON` to get the same effect as a static pixmap. For menu item controls, captions are used within the control to convey the action that will be performed when the control is activated.

Note: The initial caption for a control is specified in the resource file. The button style `BS_BITMAP` or `BS_ICON` must be set for the push button or toggle button for which a pixmap is to be assigned.

The following functions are provided to access the control captions and pixmaps:

`mgSetCaption`	sets the caption of a control to a specified text value.
`mgClearCaption`	clears the caption of a control.
`mgGetCaption`	returns the caption of a control.
`mgSetPixmap`	assigns a pixmap to be displayed for a control (except menu item controls).

Most controls can be configured to display tool tips when the user positions the mouse pointer over a control in a dialog. See "Tool Tips" on page 244 for more information.

Just as you can send the refresh control event to all controls contained in a dialog using the function `mgRefreshDialog,` you can send the refresh control event to an individual control using the function `mgRefreshControl.`

Note: You cannot assign a pixmap to a menu item control.

Sizing and Positioning Gui Items

When a dialog instance is created its initial size and position (as well as the size and position of all the controls it contains) is defined in the resource file. After the dialog instance has been created, its size and position can be changed using API functions. These same functions can be used to size and position controls within dialogs as well (with the exception of menu item controls which are always positioned relative to the menu bars containing them).

This section describes the API functions used to size and position GUI items. Before these functions can be described, the coordinate system in which the dimensions and positions of GUI items are expressed must be defined.

All positions and dimensions are expressed in screen pixels. The screen origin is the upper left corner of the screen and is at position (0, 0). A dialog has two dimensions, an outer dimension and an inner (view) dimension. The outer dimension defines the screen area occupied by the dialog and includes any margins or title bar areas of the dialog. The view dimension of a dialog defines the view area of the dialog and includes only the area that is occupied by controls. The view area does not include margin or title bar areas of the dialog. The position of a dialog is the position of the upper left corner of the screen area of the dialog and is always measured relative to the screen origin.

Just like dialogs, controls also have two dimensions, an outer dimension and an inner (view) dimension. For most classes of controls, however, the outer dimension is identical to the view dimension. The position of a control is the position of the upper left corner of the outer dimension of the control and is always measured relative to the upper left corner of the dialog's view area. A control in the upper left corner of a dialog's view area is at position (0,0).

You can get and set the size and position of GUI items using the following functions:

`mgGuiGetSize`	returns the outer dimensions of a dialog or control.
`mgGuiSetSize`	sets the outer dimensions of a dialog or control.
`mgGuiGetViewSize`	returns the view dimensions of a dialog or control.
`mgGuiGetPos`	returns the position of a dialog or control.
`mgGuiSetPos`	sets the position of a dialog or control.

Note: You cannot set the view area dimensions for dialogs or controls, nor can you set the position or dimensions for menu item controls.

Changing the Font of Gui Items

The font used for a dialog is specified in the resource file. When you choose a font for a dialog in this way, the font is applied to all the controls contained in the dialog. The default font is a variable width font. Within a variable width font, the width of each character glyph varies according to the character (e.g., the glyph used for the “i” character is narrower than that used for the “m” character etc.).

Sometimes you may want to use a fixed width font (one in which all character glyphs are the same width) in one or more controls in the dialog. For example, you may have a list control in which you will display items that contain distinct “fields” that you want to align in a columnar fashion. The only way to get a fixed width font on your list control via the resource file is to apply the fixed width font to the entire dialog. If you do this, however, all the other controls in the dialog will have the fixed width font as well - and that is probably not what you want. The API provides a function that you can use to apply a fixed width font to a specific control. To apply a fixed width font to a control, use the function **`mgGuiSetFixedFont.`** The API chooses a fixed width font suitable for the current system settings.

Note: You cannot change the font of menu item controls.

Menus in Dialogs

A plug-in dialog can contain a menu bar which contains submenus and individual menu item controls. Adding menus to your plug-in dialog is simple if you follow these steps:

1. Define your menu bar in the resource file. Here you can arrange the hierarchy of submenus and individual menu items. When defining your menu bar, keep in mind:

   - The API supports two kinds of menu item controls, **checked** and **unchecked**. A checked menu item control behaves like a toggle button control while one that is unchecked behaves like a push button control. You must define the type for each menu item control in the resource file and cannot change a menu control’s type at runtime. These two kinds of menu item controls are described later in this section.

- Assign each menu item a unique control identifier. Just like other controls, this identifier is needed to access the individual menu item controls in your dialog.
- You can define any *practical* level of nesting in your submenu structure. The API does not restrict the hierarchy of submenus you create as long as each menu item has a unique identifier.

2 In the resource file, associate the menu bar to your dialog. When you do this, the menu bar will automatically be created and attached to the dialog when the dialog is created.

3 Just like you do other controls in your dialog, assign control callback functions to individual menu item controls in your menu bar using **`mgSetGuiCallback.`** In this way, you define the actions taken when users select your menu items.

As stated above, there are two kinds of menu item controls, checked and unchecked. Checked menu item controls behave like toggle button controls, unchecked menu item controls like push button controls. The API displays a **check mark** icon next to checked menu items to indicate their checked state (on or off). You can use **`mgMenuSetState`** and **`mgMenuGetState`** to set and query the state of checked menu item controls. The following code fragment shows how to set up menu item controls in your dialog:

```
static int State = 0;

static mgstatus MenuButtonCallback ( mggui gui, mgcontrolid controlId,
                              mgguicallbackreason callbackReason,
                              void *userData, void *callData )
{
   if ( callbackReason == MGCB_ACTIVATE ) {
      // perform action when user selects menu item
   }
   else if ( callbackReason == MGCB_REFRESH ) {
      // this happens just before the menu is displayed,
      // you can call mgSetEnabled to gray out menu
      // item if not selectable at this time
   }
   return (MSTAT_OK);
}

static mgstatus MenuCheckCallback ( mggui gui, mgcontrolid controlId,
                              mgguicallbackreason callbackReason,
                              void *userData, void *callData )
{
   if ( callbackReason == MGCB_ACTIVATE )
      // save the state of the checked menu item
      State = mgMenuGetState ( gui );
   else if ( callbackReason == MGCB_REFRESH )
      // set the state of the checked menu item
      mgMenuSetState ( gui, State );
   return (MSTAT_OK);
}

static void InitializeControlCallbacks ( mggui dialog )
{
   mggui menuCheck;
   mggui menuButton;

   menuCheck = mgFindGuiById ( dialog, MENUCHECKID );
   menuButton = mgFindGuiById ( dialog, MENUBUTTONID );

   mgSetGuiCallback ( menuCheck, MGCB_ACTIVATE|MGCB_REFRESH,
                      MenuCheckCallback, MG_NULL );
   mgSetGuiCallback ( menuButton, MGCB_ACTIVATE|MGCB_REFRESH,
                      MenuButtonCallback, MG_NULL );
}
```

In this example, the user has defined two menu item controls in the menu bar. The first is a checked menu item whose identifier is **`MENUCHECKID`** and the second is an unchecked menu item whose identifier is **`MENUBUTTONID.`**

The checked menu item control is assigned the control callback function **`MenuCheckCallback.`** When this menu control is activated, the state of the checked menu item is saved in a global variable. Just before the menu item is displayed (when the refresh control event is sent to the control), the state of this global variable is used to set the checked state of the menu item control.

The unchecked menu item control is assigned the control callback function **`MenuButtonCallback.`** When this menu control is activated, the callback can perform the action appropriate for when the user selects this menu item.

In certain situations, you may want to disallow users from selecting a menu item. Just like other controls, you can use the function **`mgSetEnabled`** to do this. A good time to do this is in response to the refresh control event for the menu item control. The refresh control event is sent to a menu item control just before it is displayed by the user.

For more sample code on how to add a menu to a plug-in dialog, see the sample plug-in **`menutest`** included in the sample plug-in folder in the OpenFlight API SDK.

Note: While menu item controls are very similar to other controls in your dialogs, they are slightly different. In particular, there are some general control functions that are not applicable to menu item controls. These functions are listed here:

`mgSetVisible`	menu item controls cannot be hidden.
`mgIsVisible`	menu item controls are always visible.
`mgSetPixmap`	menu item controls can only have text captions.
`mgGuiGetSize`	menu item controls cannot be resized.
`mgGuiSetSize`	menu item controls cannot be resized.
`mgGuiGetViewSize`	menu item controls cannot be resized.
`mgGuiGetPos`	menu item controls cannot be resized.
`mgGuiSetPos`	menu item controls cannot be resized.

Spin Controls

As mentioned earlier, spin controls cannot act independently within a dialog. A spin control can only function when it is attached as a buddy control to another control in the dialog. The API supports spin controls as buddy controls to both text and scale controls. This section explains how a spin control behaves when it is attached as a buddy control. It behaves similarly whether it is attached to a text or to a scale control.

A spin control can be pressed and held down by the user to produce a continuous sequence of activate control events sent to the text or scale control to which it is attached. If the user presses and releases the spin control fast enough, a single event is sent. If the user presses and holds down the spin control longer, multiple events are sent. In this case, each event in the sequence is "marked" to indicate whether the event is the first event in the sequence, the last event in the sequence or one of the potentially many events in the middle of the sequence. The control callback that handles these events may choose to perform different actions depending on where in the sequence a particular activate control event falls. To see how this may be useful, consider the following example.

Consider a viewer tool that displays a dialog containing a GL control into which a complex scene is rendered. This dialog also contains a text control whose numerical value specifies the field of view of the scene drawn in the GL control. Attached to the text control is a spin buddy control that allows the user to continuously increment or decrement the field of view value. As the field of view value changes, the scene in the GL control is updated. If the scene takes a bit of time to render, the viewer tool may choose to draw a less complex representation of the scene (say without texture and/or depth testing) while the user holds down the spin control. This will help keep the scene drawn "up to date" with the value of the text control and will provide a better experience for the user. When the user finally releases the spin control, the viewer tool will draw the more complex representation of the scene.

When the activate control event is delivered to the text or scale control, the call specific data parameter passed to the control function indicates where in the sequence the event falls. This is discussed in more detail in "Text Buddy Controls" on page 173 and "Scale Buddy Controls" on page 183.

Text Controls

Text controls can be either *static* or *editable*. Static text controls are used to display text values to the user. Editable text controls are also used to display text values to the user but can also accept text values entered by the user. You can specify whether a text control is static or editable using the function **`mgTextSetEditable.`**

Note: A static text control can be created in two different ways. The first way is using a window of class **`STATIC.`** The second way is using a window of class **`EDIT`** with edit style **`ES_READONLY.`** The difference between the two methods is that the control created using the first technique can never become editable using the function **`mgTextSetEditable.`** The control created using the second technique can change between static and editable using this function.

There are several functions that you can use to load a text string into a text control. The simplest is **`mgTextSetString`** which copies a given text string into the control. You can also clear the text displayed in a text control using the function **`mgTextClear.`** There are also functions that load textual representations of numeric values into text controls. The functions, **`mgTextSetInteger,`** **`mgTextSetFloat,`** and **`mgTextSetDouble`** will copy textual representations of numeric values of type **`int`**, **`float`**, and **`double,`** respectively, into a text control. The third parameter to these functions is an optional **`printf`** style formatting string used to convert the numeric values to text strings before they are displayed in the text control. The following code fragment shows how to use each of these functions:

```
int iVal = 10;
float fVal = 10.0f;
double dVal = 10.0;
char string[20];
mgstatus stat;

sprintf ( string, "%d", iVal );
stat = mgTextSetString ( editControl, string );
stat = mgTextSetInteger ( editControl, iVal, MG_NULL );
stat = mgTextSetFloat ( editControl, fVal, "%.0f" );
stat = mgTextSetDouble ( editControl, dVal, "%.2lf" );
```

In this example, the calls to **`mgTextSetString, mgTextSetInteger,`** and **`mgTextSetFloat`** all display the identical text string, **`"10"`**, in the edit control. The call to **`mgTextSetDouble`** displays the text string **`"10.00"`**.

When you specify **`MG_NULL`** as the formatting string, default formatting strings are used. The default formatting strings for **`mgTextSetInteger, mgTextSetFloat,`** and **`mgTextSetDouble`** are **`"%d"`**, **`"%f"`**, and **`"%lf"`** respectively.

When a text control is not long enough to display a string, typically the string is truncated on the right. This can be a problem when the text being truncated is a file name that includes a long path and the "name" part of the file name cannot be seen. There is a special function to compensate for this called **`mgTextSetFilename.`** When the file name string loaded into a text control using this function is too long to "fit" in the control, the string is truncated on the left, ensuring that the "name" part is visible within the control.

There are several functions that you can use to retrieve a text string from a text control. The simplest is **`mgTextGetString`** which retrieves a text string, as is, from the control. There are also functions that retrieve numeric values from text controls. The functions, **`mgTextGetInteger, mgTextGetFloat,`** and **`mgTextGetDouble`** attempt to convert the text contained in a control to the corresponding numeric type, **`int, float,`** and **`double,`** respectively. These functions fail if the text contained in the control cannot be interpreted as the proper numeric type. The following code fragment shows how to use each of these functions:

```
int iVal;
float fVal;
double dVal;
char string[20];
mgstatus stat;

stat = mgTextGetString ( editControl, string, 20 );
if ( MSTAT_ISOK ( stat ) )
   sscanf ( string, “%d”, &iVal );

stat = mgTextGetInteger ( editControl, &iVal );
if ( MSTAT_ISOK ( stat ) ) {
   // iVal is valid and contains an integer value
}
stat = mgTextGetFloat ( editControl, &fVal );
if ( MSTAT_ISOK ( stat ) ) {
   // fVal is valid and contains a float value
}
stat = mgTextGetDouble ( editControl, &dVal );
if ( MSTAT_ISOK ( stat ) ) {
   // dVal is valid and contains a double value
}
```

In this example, the call to `sscanf` after `mgTextGetString,` result in the same integer value being loaded into `iVal` as the call to `mgTextGetInteger.`

All the text control functions described above that retrieve text values from controls consider the whole of the text entered in the control. There is another function that retrieves just the text in the control that is currently selected. This function is `mgTextGetSelection.` You can also use the functions `mgTextSelectAll` and `mgTextSelectRange` to select all or portions of the text contained in text controls. In the following example, the call to `mgTextGetSelection` returns the string `“Selected”:`

```
char string[20];
mgstatus stat;

stat = mgTextSetString ( editControl, “Some Selected Text” );
stat = mgTextSelectAll ( editControl );
stat = mgTextSelectRange ( editControl, 5, 12 );
stat = mgTextGetSelection ( editControl, string, 20 );
```

Text Buddy Controls

The behavior of a text control can be enhanced by attaching a spin buddy control to it. When a spin buddy is attached to a text control, the user can press and hold the spin control to continuously increment or decrement the value displayed in the text control. The following code fragment shows how a spin buddy control might be attached to a text control.

```
static mgstatus TextCallback ( mggui gui, mgcontrolid controlId,
                          mgguicallbackreason callbackReason,
                          void *userData, void *callData )
{
   if ( callbackReason == MGCB_ACTIVATE ) {
      mgtextactivatecallbackrec* cbData =
                     (mgtextactivatecallbackrec*) callData;
      mgmousestate mouseState = cbData->mouseState;
      switch ( mouseState )
      {
         case MMSS_NONE:
            // user typed into text field or clicked spin buddy
            // fast enough such that only one event was sent
            break;
         case MMSS_START:
            // first event when user holds down spin buddy
            break;
         case MMSS_CONTINUE:
            // one of N events while user holds down spin buddy
            break;
         case MMSS_STOP:
            // last event when user releases spin buddy
            break;
      }
   }
   else if ( callbackReason == MGCB_REFRESH )
      . . .
   return (MSTAT_OK);
}
```

In this example, `mgTextSetSpinBuddy` is called to attach a spin buddy control to a text control. The identifier of the spin control is `SPINID` and the identifier of the text control is `TEXTID.` The function `mgTextSetSpinIncrement` sets the *spin increment* to `0.01.` This is the amount by which the value displayed in the text control will be incremented or decremented when the user presses the up or down arrow of the spin control. The default spin increment is `1.0.` When the user presses the spin control, the activate control event is sent to the text control, not the spin. Also when the user presses the spin control, the updated value is automatically written back into the text control as a double precision floating point value, or `double`, using the *text format string* specified by `mgTextSetTextFormat.` The default text format string is `"%lf"`. The format string you specify must be compatible with double. In this example the format string `"%.0lf"` will make the value in the text control look like an `integer.`

Note: When calling `mgTextSetTextFormat`, it is important to specify a format string similar to the one used when using any of the `mgTextSetXXX` functions (in this example, `mgTextSetInteger`) to write string values into the text control. If you do not use format strings that yield identical representations of your numeric value, the text control will display inconsistent formats of the value. It will display one format when the user presses the spin control and then will display another when the user types a value into the text control.

Text Activate Call Specific Data

When a text control receives the activate control event via its control callback function, the call specific data parameter passed to the control callback points to an object of type **`mgtextactivatecallbackrec.`** This record contains a field which indicates the sequencing of this activate control event. This information is most useful when the text control has a spin buddy control attached. See “Spin Controls” on page 170 for a description of how activate control events are sequenced when they are triggered by spin buddy controls. The following code fragment shows how to extract this call specific data when the activate control event is received by the text control callback:

```
#define DFORMAT “%.01f”
#define IFORMAT “%d”

static int Value = 0;

static mgstatus TextCallback ( mggui gui, mgcontrolid controlId,
                          mgguicallbackreason callbackReason,
                          void *userData, void *callData )
{
   if ( callbackReason == MGCB_ACTIVATE ) {
      int iVal;
      if ( MSTAT_ISOK ( mgTextGetInteger ( gui, &iVal ) ) )
         Value = iVal;
   }
   else if ( callbackReason == MGCB_REFRESH )
      mgTextSetInteger ( gui, Value, IFORMAT );
   return (MSTAT_OK);
}

static void InitializeControlCallbacks ( mggui dialog )
{
   mggui text, spin;

   text = mgFindGuiById ( dialog, TEXTID );
   spin = mgFindGuiById ( dialog, SPINID );
   mgTextSetSpinBuddy ( text, spin );
   mgTextSetSpinIncrement ( text, 0.01 );
   mgTextSetTextFormat ( text, DFORMAT );

   mgSetGuiCallback ( text, MGCB_ACTIVATE|MGCB_REFRESH,
                      TextCallback, MG_NULL );
}
```

Toggle Button Controls

Toggle button controls can be used to implement checkbox or radio button style user interfaces. In either case, a toggle button control can be in either of two states: *set* or *clear*. For a checkbox style toggle button control, its state is independent of all other controls in the dialog. Radio button style controls are usually part of a group that represents a set of related but mutually exclusive options. In this way, only one toggle button control in a radio button group is set at one time; all others are clear.

You can specify that a toggle button control is set or clear using the function `mgToggleButtonSetState.` To query whether a toggle button control is set or clear, use the function `mgToggleButtonGetState.` Currently, the API supports only two state toggle button controls but may introduce multi-state controls in the future. Therefore, the state of a toggle button control is represented to these functions by an `int` type rather than an `mgbool.` For now, the integer value `1` corresponds to the set state, and `0` (zero) corresponds to the clear state.

List Controls

List controls contain a list of items from which the user can choose. List controls can be configured as *single* or *multi-select*. Single select list controls allow at most one item from the list to be selected. Multi-select list controls allow selection of any number of items. In either case, when the number of items in a list control exceeds the number of items that can be simultaneously displayed in the visible part of the control, the list control can include scroll bars that allow the user to view different chunks of the list items. User defined data can be associated to individual items in a list control.

You specify whether a list control is single or multi-select in the resource file. The API does not provide a function to change this property of a list control after it is created.

The API contains list control functions to add, delete, select, query, and position items in a list. In general, these functions allow for list items to be identified by string or by position. The first item in a list control is at position `1.`

There are several functions to add items to a list control. The first is

`mgListAddItem`	adds an item to a list at a specified position.
`mgListAddItemData`	adds an item to a list at a specified position and associates user defined data to that item.
`mgListAppendItem`	adds an item to the end of a list.
`mgListAppendItemData`	adds an item to the end of a and associates user defined data to that item.
`mgListReplaceItem`	replaces the first item in the list that matches a specified string with another string.
`mgListReplaceItemAtPos`	replaces the item at a specified position with another string.

`mgListAddItem.` When you use **`mgListAddItem,`** **`mgListAddItemData,`** **`mgListAppendItem,`** or **`mgListAppendItemData`** you can also specify whether or not the item is to be initially selected.

Given a list control that is initially empty, the following code fragment yields a list that contains the items ***`Item 1`***, ***`New Item 2,`*** and ***`Item 3 with Data`***. ***`Item 1`*** is selected while the other items are not:

mgstatus stat;

```
stat = mgListAddItem (listControl, “Item 1”, 1, MG_TRUE);

stat = mgListAppendItem (listControl, “Item 2”, MG_FALSE);

stat = mgListReplaceItem (listControl, “Item 2”, “New Item 2”);


char buf[100];

strcpy (buf, “Item 3 has data”);

char* itemText = mgMalloc (strlen(buf)+1);

strcpy (itemText, buf);


stat = mgListAppendItemData (listControl, “Item 3 with Data”, itemText,
MG_FALSE);
```

The following functions set or retrieve user defined data associated to items in a list control. Note that if you attach dynamically allocated data to items in a list control, it is your responsibility to deallocate the memory at the appropriate time. The data associated to list items is not deallocated or otherwise disposed of when a list control is destroyed.

`mgListSetItemDataAtPos`	associates user defined data with an item in a list control.
`mgListGetItemDataAtPos`	retrieves user defined data associated with an item in a list control.

The following functions delete items from a list control:

`mgListDeleteItem`	deletes the item that matches a specified string.
`mgListDeleteItemAtPos`	deletes the item at a specified position.
`mgListDeleteAllItems`	deletes all items from a list.

`mgListDeleteSelectedItems`	deletes all the items in a list that are selected.

The following functions select or deselect items in a list control:

`mgListSelectItem`	selects the item that matches a specified string.
`mgListSelectItemAtPos`	selects the item at a specified position.
`mgListSelectAllItems`	selects all items in a list.
`mgListDeselectAllItems`	deselects all the items in a list.
`mgListDeselectItemAtPos`	deselects the item at a specified position.

To query the number of items or number of selected items in a list, use the following:

`mgListGetItemCount`	returns the number of items in a list.
`mgListGetSelectedItemCount`	returns the number of items selected in a list.
`mgListIsItemInList`	determines if a string is contained in a list.
`mgListGetSelectedItemString`	returns the string of the selected item in a list. For multi-select lists, returns the string of the first selected item.
`mgListGetSelectedItemPos`	returns the position of the selected item in a list. For multi-select lists, returns the position of the first selected item.
`mgListIsItemAtPosSelected`	determines if an item at a specified position is selected in a list.

There are several functions that retrieve the actual strings contained in a list:

mgListGetItemStringAtPos	returns the string of the item in a list at a specified position.
mgListGetStrings	returns the text of all the items in a list.
mgListGetSelectedStrings	returns the text of all the selected items in a list.

Both of these functions return null-terminated arrays of character strings that have been allocated by the API. When you are done accessing the arrays returned, use the function **mgFreeStringList** to dispose of the associated allocated memory. The following code fragment prints out the text of all the selected items in the list control:

```
char **strings;

if ( strings = mgListGetSelectedStrings ( listControl ) ) {
   char **thisString = strings;
   int i = 1;
   while ( thisString && *thisString ) {
      printf ( "Selected item %d : %s\n", i, *thisString );
      thisString++;
      i++;
   }
   mgFreeStringList ( strings );
}
```

When a list control is in a scrolling state (e.g., it contains more items than can be displayed within the visible portion of the control), there are two pairs of functions that can be used to position list items within the visible part of the control. The first pair, **mgListSetTopItem** and **mgListSetTopPos,** attempt to make an item that matches a specified string or is at a specified position, respectively, the *first* visible item in the list. Calling these functions only ensures that the specified item is visible in a scrolling list control. Depending on the configuration of the items in the list, when either of these functions is called, it may not be possible to make the specified item the first item visible, but it is guaranteed to be visible. Similar to **mgListSetTopItem** and **mgListSetTopPos, mgListSetBottomItem** and **mgListSetBottomPos** attempt to make an item that matches a specified string or is at a specified position, respectively, the *last* visible item in the list. You can also use the function, **mgListGetTopPos** to return the position of the first visible item in a list control.

Option Menu Controls

Like single select list controls, option menu controls display a list (or menu) of items from which the user can choose at most one item. Option menu controls statically display only the text of the selected item. The entire menu list is displayed only when the user clicks on the control. While the menu is displayed, the user can select an item from the list. After an item is selected, the menu is hidden again. In this way, option menu controls are more compact than list controls.

The API contains option menu control functions to add, delete, select, and query items in an option menu. Unlike list controls, these functions allow for menu items to be identified by position only. The first item in an option menu is at position 1.

The following functions manipulate option menu controls:

`mgOptionMenuAddItem`	adds an item to the end of an option menu.
`mgOptionMenuDeleteAllItems`	deletes all items from an option menu.
`mgOptionMenuSelectAtPos`	selects an item in an option menu at a specified position.
`mgOptionMenuGetItemCount`	returns the number of items in an option menu.
`mgOptionMenuGetSelectedItemString`	returns the string of the selected item in an option menu.
`mgOptionMenuGetSelectedItemPos`	returns the position of the selected item in an option menu.

Note: The API only supports option menus whose Window Style is **`CBS_DROPDOWNLIST.`** Also, be aware when you set the **`CBS_SORT`** property of an option menu control in the resource file. When you do, the option menu automatically sorts the items alphabetically as you add them to it. Given this behavior, when you add an item to the option menu using **`mgOptionMenuAddItem,`** the item you add may not actually get added to the end of the option menu list as you intend. Instead, the item will get added in the position according to where it fits alphabetically with the other items in the list. Therefore, if your code depends on the option menu items appearing in the same order in which you added them, you may get incorrect results if the **`CBS_SORT`** property is set.

Scale Controls

Scale controls allow the user to select a value from within a finite range of values using a slider. The user can drag the slider of the scale control to increase or decrease the value of the control. A scale control has a range and a current position (value). The range represents the minimum and maximum valid values of the scale, and the current position represents the current value of the scale. You can define the range of a scale control as well as its position. The default minimum and maximum values of a scale control are 0 and 100 respectively. To afford the maximum resolution for scale controls, the API stores scale control values in double precision floating point or **`double.`**

The following functions manipulate scale controls:

`mgScaleSetValue`	sets the value of a scale control to a specified **`double`** value.
`mgScaleGetValue`	returns the value of a scale control.
`mgScaleSetMinMax`	sets the minimum and maximum values a scale control can display.
`mgScaleGetMinMax`	returns the minimum and maximum values a scale control can display.

Scale Control Behavior

When the user presses and drags a scale control, a continuous sequence of activate control events is sent to the scale control. Each event in the sequence is “marked” to indicate whether the event is the first event in the sequence, the last event in the sequence or one of the potentially many events in the middle of the sequence. The control callback that handles these events may choose to perform different actions depending on where in the sequence a particular activate control event falls.

Scale Buddy Controls

The behavior of a scale control can be enhanced by attaching either a spin buddy control or a text buddy control (or both) to it. When a spin buddy control is attached, the user can press and hold the spin control to continuously increment or decrement the value displayed in the scale control. When a text buddy control is attached, the user can type values for the scale control into the text control. The attached text buddy will also display the current value of the scale control automatically as it changes.

When the user activates a spin or text buddy control attached to a scale, the corresponding activate control event is sent to the scale control, not the spin or text. The following code fragment shows how a spin and a text buddy control might be attached to a scale control:

```
#define FORMAT "%.01f"

static double Value = 0.0;

static mgstatus ScaleCallback ( mggui gui, mgcontrolid controlId,
                          mgguicallbackreason callbackReason,
                          void *userData, void *callData )
{
   if ( callbackReason == MGCB_ACTIVATE ) {
      double dVal;
      if ( MSTAT_ISOK ( mgScaleGetValue ( gui, &dVal ) ) )
         Value = dVal;
   }
   else if ( callbackReason == MGCB_REFRESH )
      mgScaleSetValue ( gui, Value );
   return (MSTAT_OK);
}

static void InitializeControlCallbacks ( mggui dialog )
{
   mggui scale, text, spin;

   scale = mgFindGuiById ( dialog, SCALEID );
   text = mgFindGuiById ( dialog, TEXTID );
   spin = mgFindGuiById ( dialog, SPINID );

   mgScaleSetSpinBuddy ( scale, spin );
   mgScaleSetTextBuddy ( scale, text );
   mgScaleSetSpinIncrement ( scale, 0.01 );
   mgScaleSetTextFormat ( scale, FORMAT );

   mgSetGuiCallback ( scale, MGCB_ACTIVATE|MGCB_REFRESH,
                          ScaleCallback, MG_NULL );
}
```

In this example, **`mgScaleSetSpinBuddy`** and **`mgScaleSetTextBuddy`** are called to attach a spin buddy control (**`SPINID)`** and a text buddy control (**`TEXTID`**), respectively, to a scale control (**`SCALEID`**). The function **`mgScaleSetSpinIncrement`** sets the *spin increment* to **`0.01.`** This is the amount by which the value displayed in the scale control will be incremented or decremented when the user presses the up or down arrow of the spin control. The default spin increment is **`1.0.`** When the value of the scale control changes, its new value is automatically written back into the text buddy control as a double precision floating point value, or **`double,`** using the *text format string* specified by **`mgScaleSetTextFormat.`** The default text format string is **`"%lf"`**. The format string you specify must be compatible with **`double.`** In this example the format string **`"%.0lf"`** will make the value in the text control look like an **`integer.`**

Scale Activate Call Specific Data

As mentioned in "Scale Control Behavior" on page 182, user interaction with a scale control produces a continuous flow of activate control events while the scale is being dragged. A spin buddy control attached to a scale control also produces a similar flow of events to the scale control. Both of these event flows are packaged in *start-continue-stop* sequences. These event sequences always begin with exactly one *start* event, followed by zero or more *continue* events, and terminate with exactly one *stop* event. Where a particular activate control event falls in this event sequence is called the *mouse state* of the event.

When a scale control receives the activate control event via its control callback function, the call specific data parameter passed to the control callback points to an object of type **`mgscaleactivatecallbackrec.`** This record contains a field which indicates the mouse state of the activate control event being reported. For scale controls, the mouse state of an activate control event does not indicate whether the scale is being dragged or the attached spin buddy control is being pressed. Both of these user interactions produce the same sequence of events and the control callback need not know the difference:

The following code fragment shows how to extract this call specific data when the activate control event is received by the text control callback.

```
static mgstatus ScaleCallback ( mggui gui, mgcontrolid controlId,
                          mgguicallbackreason callbackReason,
                          void *userData, void *callData )
{
   if ( callbackReason == MGCB_ACTIVATE ) {
      mgscaleactivatecallbackrec* cbData =
                     (mgscaleactivatecallbackrec*) callData;
      mgmousestate mouseState = cbData->mouseState;
      switch ( mouseState )
      {
         case MMSS_NONE:
            // user entered value into text buddy or clicked spin
            // buddy fast enough such that only one event was sent
            break;
         case MMSS_START:
            // first event when user begins dragging slider
            // or holds down spin buddy
            break;
         case MMSS_CONTINUE:
            // one of N events while user continues dragging
            // slider or holds down spin buddy
            break;
         case MMSS_STOP:
            // last event when user releases slider or spin buddy
            break;
      }
   }
   else if ( callbackReason == MGCB_REFRESH )
      . . .
   return (MSTAT_OK);
}
```

Progress Controls

A progress control can be used to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled, from left to right, with a colored bar as an operation progresses. A progress control has a range and a current position (value). The range represents the entire duration of the operation, and the current position represents the progress that the application has made toward completing the operation. You can define the range of a progress control. The default minimum and maximum values of a progress control are 0 and 100 respectively.

You can set the position of a progress control in one of two ways. You can *set* the value or *step* the value. To set the value, call **`mgProgressSetValue.`** To step the value, call **`mgProgressSetStepIncrement`** to set a step increment. Then call **`mgProgressStepValue`** to increment the value of the control. The default value of the step increment is 10.

To afford the maximum resolution for progress controls, the API stores progress control values in double precision floating point or **`double.`**

The following functions manipulate progress controls:

`mgProgressSetValue`	sets the value of a progress control to a specified **`double`** value.
`mgProgressGetValue`	returns the value of a progress control.
`mgProgressSetMinMax`	sets the minimum and maximum values a progress control can display.
`mgProgressGetMinMax`	returns the minimum and maximum values a progress control can display.
`mgProgressSetStepIncrement`	sets the step increment value for a progress control - this is the value by which the progress control is incremented by the function **`mgProgressStepValue`**.
`mgProgressStepValue`	increments the value of a progress control by the current step increment value.

GL Controls

A GL control can be used to perform OpenGL rendering. It can receive and process mouse pointer input from the user. It can be placed at any position and given any dimensions within a dialog. Any OpenGL rendering calls can be made to draw in the control.

Control callbacks are assigned to GL controls just like any other control class as shown in the following code fragment:

```
#ifdef _WIN32
#include <windows.h>
#endif
#include <GL/gl.h>
```

```
static mgstatus DrawCallback ( mggui gui, mgcontrolid controlId,
                          mgguicallbackreason callbackReason,
                          void *userData, void *callData )
{
   mggldrawcallbackrec* cbData = (mggldrawcallbackrec*) callData;
   mgglcontext glContext = cbData->glContext;
   int width = cbData->width;
   int height = cbData->height;

   // perform plug-in specific OpenGL rendering
   return (MSTAT_OK);
}

static void InitializeControlCallbacks ( mggui dialog )
{
   mggui gl = mgFindGuiById ( dialog, GLID );
   mgSetGuiCallback ( gl, MGCB_DRAW, DrawCallback, MG_NULL );
}
```

In this example, the function **`DrawCallback`** is assigned as the control callback for the GL control whose identifier is **`GLID.`** Only the draw control event has been selected. Whenever the contents of the GL control needs to be redrawn, the draw control event will be sent to this function. When a GL control receives the draw control event via its control callback function, the call specific data parameter passed points to an object of type **`mggldrawcallbackrec.`** This record contains fields which specify the GL control's width and height. It also contains an opaque object of type **`mgglcontext`** which is required when making calls to API provided OpenGL convenience functions described in "GL Control Borders" on page 189.

When invoked to handle the draw control event, the GL control callback can perform any OpenGL rendering to draw the contents of the control. The example above lists the required header files your plug-in code will have to include to use OpenGL functions and symbols on Windows. Before the draw control event is sent, the API does all the necessary GL control setup to enable drawing in the GL control. The next section describes this setup and provides guidelines for writing a control callback for a GL control.

Note: GL controls only receive the draw control event. The activate and refresh control events are not applicable for GL controls. Therefore, if you select **`MGCB_ACTIVATE or MGCB_REFRESH`** when calling **`mgSetGuiCallback`** for a GL control, they will be ignored.

For more sample code on how to add GL controls to a plug-in dialog, see the sample plug-in **`guitest`** included in the sample plug-in folder in the OpenFlight API SDK.

GL Draw Control Callback

As mentioned above, before the draw control event is sent to a GL control via its control callback, the API sets up the GL control to enable drawing within it. This setup includes attaching the proper GL rendering context, initializing a default 2D orthographic projection by calling **`glViewport`** and **`glOrtho`** with the proper dimensions of the GL control, and loading the identify matrix onto the model view matrix stack by calling **`glMatrixMode (GL_MODELVIEW)`** and **`glLoadIdentity.`** Other than the initial state described here, the GL control callback can make no other assumptions about the state of the GL rendering context when the draw control event is sent.

All GL controls in Creator share a common GL rendering context. This includes the GL controls that Creator uses in its windows and dialogs as well as those GL controls created within plug-in dialogs. This strategy has some advantages as well as some disadvantages. The primary advantage of a shared GL rendering context is that all GL controls can render using the same texture, material, lighting and other attribute definitions. This makes context switching between GL controls very efficient and centralizes the chore of context management in Creator. Unfortunately, this strategy imposes some rules on the developer when writing a GL draw control callback. The rule of thumb when writing a GL draw control callback is that you must restore any rendering state changes you make in your callback before your callback returns. This includes the state of all matrix stacks and rendering capabilities such as texture, lighting, color, material, depth testing, etc.

The draw control event is sent to a GL control whenever the window system determines that the contents of the control need to be redrawn. This happens automatically via the interaction of Creator with the underlying window system - when the GL control becomes visible, is moved, etc. Your plug-in may also send the draw control event to a GL control to force a re-draw by calling **`mgDrawControl.`** If your plug-in needs the GL control to be redrawn, you must call this function instead of calling the control callback function directly. This is important because if you do not go through **`mgDrawControl,`** the GL rendering context of your GL control will not be attached and setup correctly. This will lead to undefined rendering results.

GL Control Borders

It is often desirable to put a *frame* or *border* around your GL control. This gives your plug-in dialogs a more “finished” look. You can do this in a number of ways:

- Create a static picture control around your GL control and assign appropriate window styles to the picture control to get different kinds of borders. So the picture control does not obscure the GL control, define the tab order of the controls so the GL control is “in front” of the picture control or assign the “transparent” style to the picture control.
- Add OpenGL code to the end of your GL Draw Control Callback to draw a border around the outside of your GL control (after your code has drawn the contents of the GL control).
- Let the API do it for you automatically. This section describes how to do just that!

The API provides a convenient mechanism to draw specific *borders* automatically around your GL control. By assigning a border style attribute to your GL control using the function **`mgControlSetAttribute,`** you can specify which type of border will be drawn around your GL control. After assigning a border style attribute to a GL control in this way, the API will automatically draw the corresponding border around the control - immediately after calling the GL Draw Control Callback for the GL control:

```
static void InitializeControlCallbacks ( mggui dialog )
{
   mggui gl = mgFindGuiById ( dialog, GLID );
   mgSetGuiCallback ( gl, MGCB_DRAW, DrawCallback, MG_NULL );
   mgControlSetAttribute ( gl, MCA_GLBORDERSTYLE, MGLBS_SUNKEN );
}
```

In this example, the function **`mgControlSetAttribute`** is called to assign the **`MCA_GLBORDERSTYLE`** attribute for the GL control whose identifier is **`GLID.`** The value for this attribute is assigned **`MGLBS_SUNKEN.`** The next time this GL control is drawn (immediately after the GL Draw Control Callback, **`DrawCallback`** is called), a sunken border will be drawn automatically around the control. The valid values for the **`MCA_GLBORDERSTYLE`** are listed here:

- *GL Border Style None* **`[MGLBS_NONE]`**: If you specify this value for **`MCA_GLBORDERSTYLE,`** no border will be drawn around the GL control. This is the default.

- *GL Border Style Sunken* **[MGLBS_SUNKEN]:** If you specify this value for **MCA_GLBORDERSTYLE,** a sunken frame will be drawn around the GL control. This will make the GL control appear inset in the dialog.
- *GL Border Style Raised* **[MGLBS_RAISED]**: If you specify this value for **MCA_GLBORDERSTYLE,** a raised frame will be drawn around the GL control. This will make the GL control appear raised above the dialog.
- *GL Border Style Solid* **[MGLBS_SOLID]**: If you specify this value for **MCA_GLBORDERSTYLE,** a solid black frame will be drawn around the GL control.

Note: All frames (except **MGLBS_NONE)** are drawn with a width of 2 screen pixels just inside the dimensions of your GL control. Since the frame the API draws is *inside* your control, it will obscure whatever you draw within that 2 pixel-wide border. So whenever you choose a border style in this way, be careful to account for this border when implementing the rendering code in your GL Draw Control Callback. Remember, if you draw anything within the 2 pixel-wide border, it will be covered by the border drawn by the API.

GL Control Rendering Using Database Palettes

When drawing into a GL control using OpenGL rendering functions, you should not define your own material, lighting or texture definitions. If you do so, you may overwrite existing definitions maintained by Creator for databases that are open on the Creator desktop. This is because all GL controls share a common rendering context. If you define your own material, lighting or texture definitions, this will lead to undefined rendering results, not only in your GL control but also in the graphics views displayed on the Creator desktop.

When you are drawing into a GL control, the Tools API does allow you to "bind" to color, material and texture definitions defined in palettes of database files open on the Creator desktop. In most cases, these are the color, material and texture definitions you will want to render with anyway. For the most part, you will not need to define your own material, lights or textures - just use one of the existing definitions from a database palette. Your GL control callback can use the following functions to "bind" to database color, material and texture definitions.

`mgGLColorIndex`	specifies a color for GL rendering using a database color index and intensity.
`mgGLMaterialIndex`	specifies a material for GL rendering using a database material index.
`mgGLTextureIndex`	specifies a texture for GL rendering using a database texture index.

GL Control Mouse Input

The Tools API provides a mechanism for GL controls to receive mouse pointer input from the user. This input is packaged and is sent to the GL mouse function assigned to the GL control by **`mgGLSetMouseFunc`** in the form of GL mouse events. Several kinds of GL mouse events can be processed by GL controls and sent to GL mouse functions; *motion* events, *button* events, *wheel* events and *double-click* events.

The GL mouse events, motion and button, are very similar. They both are used to report the current position of the mouse within the GL control. Motion events, however, are only sent when *no mouse button is pressed.* Button events, on the other hand, are only sent when *one or more mouse buttons are pressed.*

Motion Events

Motion events sent to GL controls arrive as a sequence of one or more events and are used to report that the mouse is positioned within the bounds of the GL control. When a motion event is sent to a GL control via its GL mouse function, the call specific data parameter passed to the mouse function points to an object of type **`mgglmousemotiondatarec.`** This record contains fields which indicate the state of the special keyboard keys, **SHIFT, ALT** and **CTRL** as well as the **x, y** position of the motion event. This position is measured in screen pixels relative to the lower left corner of the control. The lower left corner of the GL control is at position `(0,0)`. Note that this is a different coordinate system than that which is used to specify positions of controls and dialogs - where the upper left corner of the screen is at `(0,0).`

Button Events

Button events sent to GL controls arrive as a sequence of events in the form *start-continue-stop*. Button events are sent when the user presses and releases any mouse button within the bounds of the GL control. This sequence is sent whether the user drags the mouse button while it is down or releases it without moving. Button event sequences always begin with exactly one *start* event, followed by zero or more *continue* events, and terminate with exactly one *stop* event. Where a particular mouse event falls in this event sequence is called the *mouse state* of the event. When a button event is sent to a GL control via its GL mouse function, the call specific data parameter passed to the mouse function points to an object of type **`mgglmousebuttondatarec.`** This record contains fields which indicate the mouse state of the button event, the state of the mouse buttons, the state of the special keyboard keys, **SHIFT, ALT** and **CTRL**as well as the **x, y** position of the mouse event. Like the position of the motion event, this position is also measured in screen pixels relative to the lower left corner of the control. The lower left corner of the GL control is at position `(0,0).`

Wheel Events

Wheel events sent to GL controls are single events sent when the user rotates the mouse wheel within the bounds of the GL control. When a wheel event is sent to a GL control via its GL mouse function, the call specific data parameter passed to the mouse function points to an object of type **`mgglmousewheeldatarec.`** This record contains fields which indicate the state of the state of the mouse buttons, the special keyboard keys, **SHIFT, ALT** and **CTRL,** the amount the mouse wheel has rotated, as well as the x, y position of the wheel event. Like the position of the motion and button events, this position is also measured in screen pixels relative to the lower left corner of the control. The lower left corner of the GL control is at position `(0,0).`

Note: Wheel events are only sent when the user's mouse is equipped with a mouse wheel.

Double Click Events

Double-click events sent to GL controls are single events sent when the user double-clicks a mouse button within the bounds of the GL control. When a double-click event is sent to a GL control via its GL mouse function, the call specific data parameter passed to the mouse function points to an object of type **`mgglmousedoubleclickdatarec.`** This record contains fields which indicate the state of the special keyboard keys, **SHIFT, ALT** and **CTRL** as well as the **x, y** position of the double-click event. Like the position of the motion, button and wheel events, this position is also measured in screen pixels relative to the lower left corner of the control. The lower left corner of the GL control is at position `(0,0)`.

GL Control Mouse Example

The following code fragment shows how a GL mouse function might be assigned to a GL control:

```
static mgstatus MouseCallback ( mggui gui, mgcontrolid controlId,
                          mgglmouseaction mouseAction,
                          void *userData, void *callData )
{
   if ( mouseAction == MGMA_BUTTON ) {
      mgglmousebuttondatarec* buttonData =
                  (mgglmousebuttondatarec*) callData;
      mgmousestate mouseEvent = buttonData->mouseEvent;
      unsigned int keyboardFlags = buttonData->keyboardFlags;
      unsigned int buttonFlags = buttonData->buttonFlags;
      mgcoord3d* thisPoint = buttonData->thisPoint;
      mgcoord3d* prevPoint = buttonData->prevPoint;
      mgcoord3d* firstPoint = buttonData->firstPoint;

      switch ( mouseEvent )
      {
         case MMSS_START:
            break;
         case MMSS_CONTINUE:
            break;
         case MMSS_STOP:
            break;
      }
   }
   else if ( mouseAction == MGMA_MOTION ) {
      mgglmousemotiondatarec* motionData =
                  (mgglmousemotiondatarec*) callData;
      unsigned int keyboardFlags = motionData->keyboardFlags;
      mgcoord3d* thisPoint = motionData->thisPoint;
   }
   else if ( mouseAction == MGMA_DOUBLECLICK ) {
      mgglmousedoubleclickdatarec* doubleClickData =
                  (mgglmousedoubleclickdatarec*) callData;
      unsigned int keyboardFlags = doubleClickData->keyboardFlags;
      unsigned int buttonFlags = doubleClickData->buttonFlags;
      mgcoord3d* thisPoint = doubleClickData->thisPoint;
   }
   return (MSTAT_OK);
}
```

```
static void InitializeControlCallbacks ( mggui dialog )
{
   mggui gl = mgFindGuiById ( dialog, GLID );
   mgGLSetMouseFunc ( gl,
                  MGMA_MOTION|MGMA_BUTTON|MGMA_DOUBLECLICK,
                  MouseCallback, MG_NULL );
}
```

In this example, the function **`MouseCallback`** is assigned as the GL mouse function to receive mouse input events directed to the GL control whose identifier is **`GLID.`** The GL mouse function will be called for motion, button and double-click mouse events and will not be passed any user data.

Tool Actions

To allow keyboard shortcuts to be used in your plug-in tool dialog, you use *tool actions.* A *tool action* is an atomic executable action that you define and bind to one or more controls in your plug-in tool dialog. You can, optionally, assign default shortcut keys to your tool actions. Users, inside Creator, can re-assign the keyboard shortcuts for the tool actions you have defined. Those keyboard shortcuts assigned will persist between modeling sessions.

You define tool actions for a plug-in tool in your plug-in initialization function. You can define as many tool actions as you want. You define each tool action using the function **`mgPluginToolNewAction`**. You provide a unique name for each tool action you define. This is the name users will see in the Keyboard Shortcuts window in Creator when they assign a shortcut to your tool action. If you want to assign a default keyboard shortcut for a tool action, use the function **`mgToolActionSetShortCut`**.

Note: For tool actions to be available in the Keyboard Shortcuts window and to otherwise function properly in Creator, you must define them in your plug-in initialization function at the same time you register your plug-in tools.

Tool actions become usable in Creator after they are “connected” (or bound) to controls in your plug-in tool dialog. When your plug-in tool dialog is first created (typically in response to the dialog initialization event sent to your dialog function) you bind a tool action to a control using the function **`mgSetGuiToolAction`**. After a tool action has been bound to a control, it is ready to be used. During the modeling session, when the user presses the shortcut key(s) assigned to a tool action bound to a control in your plug-in tool dialog, the API responds by sending the activate control event to the bound control.

In your plug-in termination function you should deallocate the tool actions you have defined using the function **`mgPluginToolFreeAction`**.

There are other functions you can use to query information about tool actions.These functions include:

`mgToolActionGetName`	gets the name assigned to a tool action.
`mgToolActionGetShortCut`	gets the shortcut key sequence assigned to a tool action, consisting of a key and a modifier key mask.
`mgToolActionGetShortCutString`	gets a readable string representation of the shortcut key sequence assigned to a tool action.
`mgPluginToolGetAction`	gets the tool action with a specified name associated with a plug-in tool.

Cursors

A cursor is a small picture whose location on the screen is controlled by the mouse. It is often useful to change the appearance of the cursor while certain operations are being performed. Doing this gives the user a clue as to what is going on. For example, it is very common to display a “watch” or “hourglass” cursor while lengthy operations are being performed.

The API provides several pre-defined cursors that a plug-in can display. The API also allows a plug-in to display custom cursors defined in its resource file. In this way, you can create very specific cursors that convey meaningful information to the user while your plug-in performs its processing. Before you can display a cursor, you must first get it from your resource file (or in the case of pre-defined cursors provided by the API, from Creator’s resource). Use the function **`mgResourceGetCursor`** to get a cursor from a resource. If you pass **`MG_NULL`** as the resource, you can get a cursor from Creator's resource.

Once you have a cursor object **`(mgcursor)`**, you can display it by calling **`mgSetCursor.`** To reset the cursor to the default arrow cursor, pass **`MG_NULL to mgSetCursor.`** Consider the following code fragment:

```
static mgbool ShowCursor (mgresource resource,
                    mgcursorid cursorId)
{
   mgcursor cursor = mgResourceGetCursor (resource, cursorId);
   mgSetCursor (cursor);
   return (cursor ? MG_TRUE : MG_FALSE);
}

static mgbool ResetCursor (void)
{
   mgSetCursor (MG_NULL);
}

// display the "Bucket" cursor provided by the API
ShowCursor (MG_NULL, MCURS_BUCKET);

// display a custom cursor identified by
// MYCURSOR contained in your resource file
ShowCursor (resource, MYCURSOR);

// revert back to the standard arrow cursor
ResetCursor ();
```

The first call to **`ShowCursor`** passes **`MG_NULL`** to **`mgResourceGetCursor.`** That means the specified cursor is to be located in Creator's resource file. In this case, it is the "bucket" cursor. The second call to **`ShowCursor`** passes the plug-in resource to **`mgResourceGetCursor.`** That means the specified cursor is to be located in the plug-in's resource file. In this case, it is a custom defined cursor identified in the resource by

MYCURSOR. Finally, passing **MG_NULL** to **mgSetCursor** is used to reset the cursor to the default arrow cursor.

MCURS_ARROW **MCURS_WAITMCURS_CROSS** **MCURS_NOMCURS_SIZEALL** **MCURS_SIZENESW** **MCURS_SIZENS** **MCURS_SIZENWSE** **MCURS_SIZEWE** **MCURS_UPARROW** **MCURS_ZOOMIN** **MCURS_ZOOMOUT** **MCURS_SPINMCURS_PEN** **MCURS_BUCKET** **MCURS_EYEDROPPER** **MCURS_SPLITH** **MCURS_SPLITV**	These are the identifiers of the predefined cursors provided by the API for your plug-in to use. See the **guitest** sample plug-in in the **sample_plugin** folder in the API SDK to see all these cursors in action.
mgResourceGetCursor	extracts a cursor from your resource file.
mgSetCursor	displays a cursor.
mgGetCursorHandle	gets the platform-specific handle for a cursor.

Predefined Pixmaps

As mentioned in this chapter, pixmaps are images that can be displayed by controls in your dialog. The API provides several pre-defined pixmaps that a plug-in can use. Following is a list of these pixmaps provided by the API:.

`MPIXMAP_NEWFILE`	These are the identifiers of the predefined pixmaps provided by the API for your plug-in to use. See the **`guitest`** sample plug-in in the **`sample_plugin`** folder in the API SDK to see all these pixmaps in action.
`MPIXMAP_OPENFILE`	
`MPIXMAP_SAVEFILE`	
`MPIXMAP_CUT`	
`MPIXMAP_COPY`	
`MPIXMAP_PASTE`	
`MPIXMAP_DELETE`	
`MPIXMAP_EDIT`	
`MPIXMAP_TEST`	
`MPIXMAP_UNDO`	
`MPIXMAP_REDO`	
`MPIXMAP_ROTATEVIEW`	
`MPIXMAP_PANVIEW`	
`MPIXMAP_ZOOMVIEW`	
`MPIXMAP_ARROW`	
`MPIXMAP_FENCE`	
`MPIXMAP_LASSO`	
`MPIXMAP_FITONFENCE`	

Remember, you use the function **`mgResourceGetPixmap`** to get a pixmap from a resource. Pass **`MG_NULL`** as the resource to get one of the pre-defined pixmaps from Creator's resource. This is shown in the following code fragment:

```
static void InitializeControlCallbacks (mggui dialog)
{
   mggui button;
   mgpixmap pixmap;
   button = mgFindGuiById (dialog, COPYBUTTONID);
   pixmap = mgResourceGetPixmap (MG_NULL, MPIXMAP_COPY);
   mgSetPixmap (button, pixmap);
   mgSetGuiCallback (button, MGCB_ACTIVATE|MGCB_REFRESH,
                        CopyCallback, MG_NULL);

   button = mgFindGuiById (dialog, PASTEBUTTONID);
   pixmap = mgResourceGetPixmap (MG_NULL, MPIXMAP_PASTE);
   mgSetPixmap (button, pixmap);
   mgSetGuiCallback (button, MGCB_ACTIVATE|MGCB_REFRESH,
                        PasteCallback, MG_NULL);
}
```

Running the *ddbuild* Parser

Use **`ddbuild`** to parse your data dictionary file and create two **C** header files, which you will ultimately build into your extensions library. To run **`ddbuild`**:

1 Open a DOS window (Windows) or Linux shell and type:

```
% ddbuild -h
```

This will display:

```
ddbuild - version 15.0
Usage: ddbuild [-h] [-s] [-f] [-p prefix -i siteid]

-h:
   Prints this help message.

-s:
   Sort the elements in the output headers and/or DD file
alphabetically.

-f:
   Add "html" formatting commands to the documentation.

-p prefix:
   Specify the prefix string for dictionary elements (default=FLT).

-i siteid:
   Specify the site id (7 character string).
```

2 To parse your data dictionary:

`ddbuild` has a simple ASCII interface. When you start the application, you are prompted to choose from four options as shown in the following example:

```
% ddbuild -p my -i mydata
ddbuild - version 15.0

Operations of "ddbuild":
               1: Read in a schema file.
               2: Write out data dictionary header files.
               3: Write out dictionary documentation.
               4: Write out OpenFlight Script Module
```

```
            0: Exit program
Enter number of desired operation: (1):
```

3 To read in your data dictionary file, enter `1` at the prompt (or press **`<return>`** to accept the default) and then enter the name of your file.

```
Enter number of desired operation: (1):<return>
Reading a schema file.
Name of file: (my.dd):my.dd
```

4 Write out the two header files. The default names are constructed from the prefix you provided. Unless you need special names, press **`<return>`** to accept the defaults.

```
Enter number of desired operation: (2):<return>
Writing data dictionary header files.
Name of definition file: (mycode.h):<return>
Name of table file: (mytable_.h):<return>
```

5 Enter `3` to write out the documentation file. Then type in the output file name, and enter `0`, `1`, or `2` to choose the access level.

In the following example, the output file name is **`my.doc`**, and entries are written out, public and private. (See the **`KEY`** option to find out more about public and private fields and records.)

```
Enter number of desired operation: (3):<return>
Writing data dictionary documentation.
Name of file: (mgapi.doc):my.doc
Enter key for access level (0=public, 1=ifmt,
2=private): (0):2
```

6 Enter `4` to write out a Python Script Module file. Then type in the output file name. This will create a Python module containing the record and field codes you will need to access your data from within OpenFlight Script.
In the following example, the output file name is **`my.py`**.

```
Enter number of desired operation: (4):<return>
Writing OpenFlight Script module.
Name of file: (MYDATA.py):my.py
```

To use this module file in your stand-alone OpenFlight Script, first copy the module file to the folder where the OpenFlight Script module files (dynamic link library files, **`_mgapilib.pyd`** and **`mgapilib.py`**) are located. Then, in your script, "import" it using the following syntax:

```
from my import *
```

Note: You do not need this module file if you want to access your extension data only from OpenFlight Script in Creator. Creator will automatically load the field and record codes into the OpenFlight Script environment when it loads the extension plug-in DLL.

7 Enter `0` to exit **`ddbuild`**.

```
Enter number of desired operation: (0):<return>
```

The Modeling Context

During the modeling session, the user depends on several constructs to perform modeling tasks. These include the *database*, the *select list*, the *modeling parent*, the *modeling mode* and the *graphics view*. The select list is a collection of nodes currently selected for operation by the user. The modeling parent is the current point of attachment in a database hierarchy. The modeling mode is the node level that determines how nodes are selected in the Graphics view. The graphics view defines different properties of the graphics rendering state of the scene on the Creator desktop. This chapter discusses these concepts.

The Database

The API provides functions to query certain aspects of the database. These functions include:

`mgIsDbModified`	checks if a database has been modified since the last "save" operation.
`mgIsDbUntitled`	checks if a database has a valid name. In Creator, when a user creates a new database, it does not have a valid name until it has been saved to disk. Prior to being saved, the name of the database (returned by **`mgRec2Filename`**) is a "temporary" one and should not be used for locating relative textures, external references, and shaders. Only after a database has been saved to an explicit file will its name be valid.
`mgRec2Filename`	gets the database name from any node of a database. If the database being queried has not yet been saved to disk, the name returned will be a "temporary" name. You can use the function **`mgIsDbUntitled`** to check this.
`mgIsFileOnDesktop`	checks if an OpenFlight file (given its name) is open on the Creator desktop.

The Select List

During the modeling session, the user selects nodes of a database before performing operations on them. Nodes that are selected in a database are collectively referred to as the *select list* of that database. The select list is the standard mechanism used by most tools in Creator to determine the nodes on which to perform actions.

The API provides a set of functions that can be used by a plug-in tool to enumerate the nodes (items) contained in a select list for a particular database. When a plug-in tool is invoked, it can obtain a copy of the select list for its activation database, and then traverse it to visit each item in the list. This *copy* of the select list is called a *select list object*. This object is opaque to a plug-in but can be accessed in many ways through the API.

The Select List Object

A select list object is a *snapshot* of the items that are selected when the select list object is created. The API provides functions to access the items of a select list object both by position and via iteration. To allow iteration of the select list object items, the select list object maintains an internal *traversal pointer* that specifies the *next* item of the select list object to be traversed. By default, when a select list object is created, its traversal pointer is set to the first item of the list.

Note: Even though the select list object is presented as a list containing items in an implied *order*, it is just a collection of items. The position of items in a select list object is arbitrary and does not reflect the order in which the items were selected by the user.

Each node of a select list object has an associated transformation matrix. This matrix represents all the transformations applied to the node above and including the node itself. This matrix is the same as the list that would result if you collected all transformations traversed from the database header node down to the node of the select list object. If you then traverse below this node, you can use this transformation matrix as a starting point to collect further transformations.

The following table describes functions that access a select list object of a database:

`mgGetSelectList`	creates and returns a select list object containing all the currently selected nodes of a database. You must call **`mgFreeRecList`** when you are done accessing the nodes in the select list object.
`mgGetRecListCount`	returns the number of selected items in a select list object.
`mgGetRecListLevel`	returns the node level of the first item in a select list object. Note: Subsequent items in the list are not guaranteed to be of this level.
`mgResetRecList`	resets the traversal pointer of a select list object to the first node in the list.
`mgGetNextRecInList`	returns the select list object node (and transformation matrix) that corresponds to the current position of the traversal pointer and then advances the traversal pointer to the next item in the list.
`mgGetNthRecInList`	returns the select list object node (and transformation matrix) that is at a specified position. The first node in a select list object is at position 1 (one).
`mgFreeRecList`	disposes of a select list object after it is done being used.

The following code fragment captures the currently selected nodes for a given database, **db**, in a select list object and shows two ways to traverse the nodes of that select list:

```
mgrec* rec;
mgmatrix* selectMatrix;
mgselectlist selectList;
int numSelected, i;

selectList = mgGetSelectList ( db );
numSelected = mgGetRecListCount ( selectList );

// iterate the items of select list object, first to last
rec = mgGetNextRecInList ( selectList, &selectMatrix );
while ( rec ) {
   // do something with rec
   rec = mgGetNextRecInList ( selectList, &selectMatrix );
}
```

```
// use direct access to visit items last to first
// ...first item is at position 1
for ( i = numSelected; i >= 1; i-- ) {
   rec = mgGetNthRecInList ( selectList, &selectMatrix, i );
   // do something with rec
}

// dispose of select list object when done with it
mgFreeRecList ( selectList );
```

Changing the Select List

In general, the user should be given complete control over which nodes are selected. The user explicitly selects nodes using standard node selection techniques such as picking and using the Select menu commands. Under certain circumstances, the user will tolerate the select list changing without their input. For example, some tools that create geometry such as **Rectangle** and **Circle** select the new nodes they create when they are finished. They do so under the assumption that the user is likely to perform some other action on these new nodes immediately following their creation. So instead of making the user explicitly select the new nodes after they are created, the tools select the new nodes automatically.

The API provides some functions to change which nodes are selected for a given database.

`mgSelectOne`	selects a database node - when node is a vertex, vertex is selected as vertex.
`mgSelectOneEdge`	selects a vertex node as an edge.
`mgSelectList`	selects a list of database nodes.
`mgDeselectOne`	deselects a database node.
`mgDeselectAll`	deselects all nodes of a database.
`mgIsSelected`	determines if a database node is selected - when node is a vertex, will return **`MG_TRUE`** if vertex is selected as vertex or as an edge.

`mgIsSelectedEdge`	determines if a vertex node is selected as an edge.

Selecting Vertices and Edges

You will notice there are two special functions in the section above for edges: **`mgSelectOneEdge`** and **`mgIsSelectedEdge`**. These are necessary because there is no OpenFlight node type for edge, only for vertex, yet Creator allows for both vertices or edges to be selected.

In the OpenFlight scene graph, there is no explicit construct for an "edge" of a polygon. Just as there is no explicit attribute for a polygon normal (the polygon normal is calculated based on the current positions of all its vertices), a polygon edge is "calculated" based on the position of a vertex of the polygon (the first vertex of the edge) and its *next* sibling vertex. For closed polygons, the last edge is a bit tricky. It is comprised of the *last* and *first* vertices of the polygon.

When you are working with vertices and their attributes in the scene graph, you do not generally need to worry about (nor do you have explicit access to) edges, only the vertices that comprise the edge. For example, if you reposition any *one* vertex of a closed polygon, you will implicitly change the position of *two* edges. This is because in a closed polygon, each vertex is part of two edges. Each vertex of a closed polygon is the *last* vertex of one edge and the *first* vertex of the *next* edge of that polygon. For unclosed polygons, the first and last vertices of the polygon are each part of only one edge, the first and last edge, respectively.

In Creator, however, the user can select vertices or edges. Since there is no construct in the OpenFlight scene graph for an "edge", only for vertices, Creator attaches a special attribute to a vertex when it is selected to indicate whether the vertex is selected *as a vertex* or is selected *as an edge*. The special select list functions for edges, **`mgSelectOneEdge`** and **`mgIsSelectedEdge`**, give you access to this vertex attribute so your plug-in can select a vertex as a vertex or as an edge and determine whether a vertex is selected as a vertex or as an edge.

Consider the following code snippet. It contains two functions. The first function **`SelectFirstVertexOfFace`** selects the first *vertex* of a polygon. The second function **`SelectFirstEdgeOfFace`** selects the first *edge* of a polygon. In both functions, the first child of the polygon is the vertex you want to select. To select this vertex as a vertex, use **`mgSelectOne`**. To select this vertex as an edge, use **`mgSelectOneEdge`**. Similarly use **`mgIsSelected`** and **`mgIsSelectedEdge`** when a vertex is selected to determine whether it is selected as a vertex or as an edge. **`mgIsSelected`** will return **`MG_TRUE`** regardless of whether the vertex is selected as a vertex or as an edge so that function by itself may not give you enough information. When a vertex is selected as a vertex, **`mgIsSelected`** will return **`MG_TRUE`** and **`mgIsSelectedEdge`** will return **`MG_FALSE`**. When a vertex is selected as an edge, both **`mgIsSelected`** and **`mgIsSelectedEdge`** will return **`MG_TRUE`**.

```
static void SelectFirstVertexOfFace ( mgrec* face )
{
   mgbool selected, selectedAsEdge;
   mgrec* firstVertex = mgGetChild ( face );

   mgSelectOne ( firstVertex );
   selected = mgIsSelected ( firstVertex );           // will be MG_TRUE
   selectedAsEdge = mgIsSelectedEdge ( firstVertex ); // will be MG_FALSE
}

static void SelectFirstEdgeOfFace ( mgrec* face )
{
   mgbool selected, selectedAsEdge;
   mgrec* firstVertex = mgGetChild ( face );

   mgSelectOneEdge ( firstVertex );
   selected = mgIsSelected ( firstVertex );           // will be MG_TRUE
   selectedAsEdge = mgIsSelectedEdge ( firstVertex ); // will be MG_TRUE
}
```

The Modeling Parent

During the modeling session, the user selects a particular node in a database that is to be the point of attachment in that database hierarchy for subsequent operations. This node is called the *database parent* or *modeling parent*. Creator maintains a separate modeling parent for each open database on the desktop.

Several tools that create geometry attach the geometry that they create beneath the modeling parent. A plug-in can access the modeling parent node of a particular database by calling **`mgGetModelingParent`** or **`mgGetDefaultModelingParent`**. A plug-in cannot programmatically set the modeling parent node.

The Modeling Mode

During the modeling session, the user sets a particular node level to control certain aspects of the modeling session. This node level is called the modeling mode. Creator maintains one modeling mode for all databases on the desktop.

Among other things, the modeling mode determines how nodes are selected in the Graphics view. Some tools may behave differently for different modeling modes. A plug-in can access the modeling mode by calling **`mgGetModelingMode.`** A plug-in can set the modeling mode by calling **`mgSetModelingMode.`**

Vertex Mode vs Edge Mode

As mentioned in "Selecting Vertices and Edges" on page 209, the OpenFlight scene graph does not explicitly distinguish between vertices and edges. Edges are implicitly defined by two adjacent vertices. However, since Creator allows the modeling mode to be vertex or edge, the API provides a special function, **`mgIsModelingModeEdge`** to help you distinguish between vertex and edge modeling modes.

Because there is no explicit record type in the OpenFlight scene graph for edge, **`mgGetModelingMode`**, by convention, returns **`fltVertex`** when the modeling mode is *either* vertex or edge. This implies that when **`mgGetModelingMode`** does return **`fltVertex,`** you cannot be sure if the modeling mode is vertex or edge. This is where **`mgIsModelingModeEdge`** can be used. When the modeling mode is vertex, **`mgGetModelingMode`** will return **`fltVertex`** and **`mgIsModelingModeEdge`** will return **`MG_FALSE`**. When the modeling mode is edge, **`mgGetModelingMode`** will return **`fltVertex`** and **`mgIsModelingModeEdge`** will return **`MG_TRUE`**.

The following code snippet defines two convenience functions you might call to determine if the modeling mode is vertex or edge:

```
static mgbool IsModelingModeVertex ( mgrec* db )
{
   if (mgGetModelingMode ( db ) == fltVertex) {
      if (mgIsModelingModeEdge ( db ) == MG_FALSE) {
         return MG_TRUE;
      }
   }
   return MG_FALSE;
}
```

```
static mgbool IsModelingModeEdge ( mgrec* db )
{
   // you don't really have to call mgGetModelingMode
   // here, mgIsModelingModeEdge only returns MG_TRUE
   // when the modeling mode is edge.
   return mgIsModelingModeEdge ( db );
}
```

The Graphics View

This section describes different aspects of graphics rendering in Creator.

Graphics View Settings

The graphiselectcs view settings describe the OpenFlight databases support 8 texture layers per polygon/vertex. During the modeling session, the user selects which texture layer is active for rendering and modeling. This allows the user to view/model individual texture layers or view texture layers blending together. Creator maintains one current texture layer setting for all databases on the desktop.

When a database is displayed in a graphic view on the Creator desktop, a plug-in can query and set different graphics view settings using the following functions:

`mgGraphicsViewSettingGetInteger`	returns an integer value from a named graphics view setting.
`mgGraphicsViewSettingGetDouble`	returns a double value from a named graphics view setting.
`mgGraphicsViewSettingSetInteger`	sets an integer view setting on one or more graphics views.
`mgGraphicsViewSettingSetDouble`	sets a double view setting on one or more graphics views.

Note: Currently only a small subset of the graphics view settings are accessible. See the enumeration type **`mgsettingname`** in the *OpenFlight API Reference* for a complete list of the accessible settings.

The Texture Layer

OpenFlight databases support eight texture layers per polygon/vertex. During the modeling session, the user selects which texture layer is active for rendering and modeling. This allows the user to view/model individual texture layers or view texture layers blending together. Creator maintains one current texture layer setting for all databases on the desktop.

The eight texture layers are numbered `0-7`. Blend is specified as `-1`. A plug-in can access the current texture layer setting by calling **`mgGetCurrentTextureLayer.`** A plug-in can set the current texture layer by calling **`mgSetCurrentTextureLayer.`**

Eyepoints

When a database is displayed in a graphic view on the Creator desktop, a plug-in can query and set the orientation of the eyepoint in the scene using the following functions:

`mgGetCurrentLookAt`	returns the current eyepoint expressed as the position of the eye, a point at which it is looking and an up vector.
`mgSetCurrentLookAt`	sets the current eyepoint expressed as the position of the eye, a point at which it is looking and an up vector (the up vector cannot be parallel to the line of sight from the eye position and the point at which it is looking).

Event Notification

During the modeling session, a plug-in tool may want to be notified when significant events occur. For example, consider a viewer tool that displays a dialog showing some attribute(s) of the currently selected nodes of the top database. When the user selects a different set of nodes or switches focus to a new database, the viewer tool will want to be notified so it can update the contents of its dialog to reflect the select list or database focus change.

The API provides a notification callback mechanism that allows plug-in tools to *subscribe* to certain model-time events. When subscribing to model-time events, the plug-in tool must specify an *action event*, an *action function,* and an *action database*. Some model-time events may require an *action node* as well. The action event is the event for which the plug-in tool is interested in receiving notification. The action function is the callback that the API calls to report the event. The action database is the database for which the event is to be reported. An action database may be a specific database that is open in Creator or may be **`MG_NULL.`** If a specific database is chosen as the action database, the model-time event will only be reported if that event happens to that database. If **`MG_NULL`** is specified as the action database, the model-time event will be reported that event happens to *any* database. The action node may further qualify the event before it is reported.

When a plug-in tool subscribes to a model-time event, the API creates a *notifier* on behalf of the tool that will report the occurrence of the action event to the tool. Subscribing to an action event is referred to as *registering a notifier.* A notifier is an opaque object to a plug-in tool but can be accessed through the API.

While a notifier is registered, it can be *enabled* or *disabled*. Enabled notifiers report events, disabled notifiers do not. A notifier is initially enabled when it is created. After it is registered, a notifier can be enabled, disabled, or unregistered explicitly by the plug-in tool.

Unregistering and disabling a notifier both stop the notifier from reporting events. When a notifier is unregistered, it is destroyed and permanently disabled - another notifier must be registered before events will be reported again. When a notifier is disabled, it is done so temporarily - the notifier must only be re-enabled before events will be reported again.

There are three different classes of model-time events, *Desktop* events, *Database* events and *Palette* events. Use **`mgRegisterNotifier`** to subscribe to desktop and database events and **`mgRegisterPaletteNotifier`** for palette events. The difference between these two registration functions is the signature (parameters) of the action function you provide for each. Creator reports different information for each class of event. These differences are reflected in the action function signatures.

Desktop events are those that happen in the context of the Creator desktop, not necessarily in the context of any particular database. For example, Creator can notify your plug-in just after it launches or just before it exits. Your plug-in may want to perform some sort of initialization or cleanup when it receives notification of these events.

Database events are reported in the context of specific databases on the Creator desktop. Examples of database events include when the select list changes or when a new database is opened. When database events are reported, the affected database and node (if applicable) will be reported to your action function

Palette events also happen in the context of a specific database but are a bit more specific. They report when something happens to a specific element contained in a palette. For example, Creator reports when the RGB values of a color index changes in the color palette and when the file associated to a specific texture index is changed in the texture palette. When palette events are reported, your action function will be notified of the database in which the palette was changed as well as the specific index of the affected palette item.

Following is a list of the model-time events sorted by class (Desktop, Database or Palette event) for which a plug-in tool can register a notifier. The API defines an enumerated type called **`mgnotifierevent`** that contains a member corresponding to each event. Listed with each event here is its member name and a description of when the event is reported.

Desktop Events

- *Desktop Initialized* **`[MNOTIFY_DESKTOPINIT]`**: is reported when the Creator desktop starts after all plug-ins are initialized.

- *Desktop Exit* **`[MNOTIFY_DESKTOPEXIT]`**: is reported when the Creator desktop exits.
- *Extension Changed* **`[MNOTIFY_EXTENSIONCHANGED]`**: is reported when the definition of any OpenFlight extension is changed in Creator.

Database Events

- *Select List Changed* **`[MNOTIFY_SELECTLISTCHANGED]`** - This is reported whenever the contents of the Select list changes for the action database.
- *New Top Database* **`[MNOTIFY_NEWTOPDATABASE]`** - This is reported whenever a new database is selected to be the top (focus) database in Creator.
- *Database Closed* **`[MNOTIFY_DATABASECLOSED]:`** is reported whenever a database is closed within Creator.
- *Current Primary Color Change* **`[MNOTIFY_CURRENTPRIMARYCOLORCHANGED]`**: is reported whenever the user selects a new primary modeling color in Creator.
- *Current Alternate Color Change* **`[MNOTIFY_CURRENTALTCOLORCHANGED]`**: is reported whenever the user selects a new alternate modeling color in Creator.
- *Current Material Change* **`[MNOTIFY_CURRENTMATERIALCHANGED]`**: is reported whenever the user selects a new modeling material in Creator.
- *Current Texture Change* **`[MNOTIFY_CURRENTTEXTURECHANGED]`**: is reported whenever the user selects a new modeling texture in Creator.
- *Current Texture Mapping Change* **`[MNOTIFY_CURRENTTEXTUREMAPPINGCHANGED]`**: is reported whenever the user selects a new modeling texture mapping in Creator.
- *Current Light Source Change* **`[MNOTIFY_CURRENTLIGHTSOURCECHANGED]`**: is reported whenever the user selects a new modeling light source in Creator.
- *Current Light Point Appearance Change* **`[MNOTIFY_CURRENTLIGHTPOINTAPPEARANCECHANGED]`**: is reported whenever the user selects a new modeling light point animation in Creator.
- *Current Light Point Animation Change* **`[MNOTIFY_CURRENTLIGHTPOINTANIMATIONCHANGED]`**: is reported

whenever the user selects a new modeling light point animation in Creator.

- *Current Sound Change* **`[MNOTIFY_CURRENTSOUNDCHANGED]`**: is reported whenever the user selects a new modeling sound in Creator.
- *Current Shader Change* **`[MNOTIFY_CURRENTSHADERCHANGED]`**: is reported whenever the user selects a new modeling shader in Creator.
- *Light Point Paletteized* **`[MNOTIFY_LIGHTPOINTPALETTEIZED]`**: is reported whenever an inline light point is read and converted to be a paletteized light point.
- *Database Opened* **`[MNOTIFY_DATABASEOPENED]`**: is reported when a database is opened in Creator.
- *Database Saved* **`[MNOTIFY_DATABASESAVED]`**: is reported when a database is saved in Creator.
- *Node Changed* **`[MNOTIFY_NODECHANGED]`**: is reported when a node attribute is changed in Creator.
- *Node Deleted* **`[MNOTIFY_NODEDELETED]`**: is reported when a node is deleted in Creator.
- *Node Created* **`[MNOTIFY_NODECREATED]`**: is reported when a node is created in Creator.
- *Node Re-Parented* **`[MNOTIFY_NODEREPARENTED]`**: is reported when a new node is re-parented in Creator.
- *Switch Distance Changed* **`[MNOTIFY_SWITCHDISTANCECHANGED]`**: is reported when the switch distance is changed in Creator, usually via the More Detail or Less Detail commands.
- *Time of Day Changed* **`[MNOTIFY_TIMEOFDAYCHANGED]`**: is reported when the sky color changes in Creator.
- *Texture Selected Changed* **`[MNOTIFY_TEXTURESELECTCHANGED]`**: is reported when the textures selected in the palette change in Creator.

Palette Events

- *Color Palette Changed* **`[MNOTIFY_COLORPALETTECHANGED]`**: is reported when a color palette entry is changed in Creator.
- *Material Palette Changed* **`[MNOTIFY_MATERIALPALETTECHANGED]`**: is reported when a material palette entry is changed in Creator.

- *Texture Palette Changed* **[MNOTIFY_TEXTUREPALETTECHANGED]**: is reported when a texture palette entry is changed in Creator.
- *Texture Mapping Palette Changed* **[MNOTIFY_TEXTUREMAPPINGPALETTECHANGED]**: is reported when a texture mapping palette entry is changed in Creator.
- *Light Point Appearance Palette Changed* **[MNOTIFY_LIGHTPOINTAPPEARANCEPALETTECHANGED]**: is reported when a light point appearance palette entry is changed in Creator.
- *Light Point Animation Palette Changed* **[MNOTIFY_LIGHTPOINTANIMATIONPALETTECHANGED]**: is reported when a light point animation palette entry is changed in Creator.
- *Shader Palette Changed* **[MNOTIFY_SHADERPALETTECHANGED]**: is reported when a shader palette entry is changed in Creator.
- *Sound Palette Changed* **[MNOTIFY_SOUNDPALETTECHANGED]**: is reported when a sound palette entry is changed in Creator.
- *Light Source Palette Changed* **[MNOTIFY_LIGHTSOURCEPALETTECHANGED]**: is reported when a light source palette entry is changed in Creator.

The table below describes the functions to create and manipulate notifiers.

mgRegisterNotifier	registers a notifier that will report a model-time event (desktop or database event) to a plug-in tool.
mgRegisterPaletteNotifier	registers a notifier that will report a model-time event (palette event) to a plug-in tool.
mgSetAllNotifiersEnabled	enables or disables all notifiers associated to a plug-in tool.
mgSetNotifierEnabled	enables or disables a notifier.
mgUnregisterAllNotifiers	unregisters all notifiers associated to a plug-in tool.
mgUnregisterNotifier	unregisters a notifier.

The class of a plug-in tool dictates the model-time events to which that tool can subscribe and may limit whether or not that tool can specify an action database of **`MG_NULL.`** Because viewer tools can be associated to any database at any given time, viewer tools can subscribe to any kind of event and may specify any action database, including **`MG_NULL.`** Because editor tool instances are associated to a particular activation database, they can subscribe only to a limited set of events. For the events to which they can subscribe, editor tools must specify a non-**`MG_NULL`** action database (the action database should be the activation database to which they are associated). Finally, due to their modal nature, database importer, database exporter and image importer tools cannot subscribe to any kind of event. The following table shows which kinds of tools can subscribe to which kinds of events.

Tool Class	Can Subscribe to Events	Action Database
Viewer	Any	Any - can be **`MG_NULL.`**
Editor	**`MNOTIFY_CURRENTPRIMARYCOLORCHANGED`** **`MNOTIFY_CURRENTALTCOLORCHANGED`** **`MNOTIFY_CURRENTMATERIALCHANGED`** **`MNOTIFY_CURRENTTEXTURECHANGED`** **`MNOTIFY_CURRENTTEXTUREMAPPINGCHANGED`** **`MNOTIFY_CURRENTLIGHTPOINTAPPEARANCECHANGED`** **`MNOTIFY_CURRENTLIGHTPOINTANIMATIONCHANGED`** **`MNOTIFY_CURRENTLIGHTSOURCECHANGED`** **`MNOTIFY_CURRENTSHADERCHANGED`** **`MNOTIFY_CURRENTSOUNDCHANGED`**	Must be the database in which the editor tool instance was started. Cannot be **`MG_NULL.`**
Database Importer	None	N/A
Database Exporter	None	N/A
Image Importer	None	N/A
Input Device	None	N/A

The following example shows how a plug-in tool, **`pluginTool`**, might set up notifiers:

```
mgrec* dbToReport;
mgnotifier notifier1, notifier2;

static void EventNotifier ( mgnotifier notifier, mgnotifierevent event,
                       mgrec *dbRec, mgrec *rec, void *userData )
{
   switch ( event ) {
      case MNOTIFY_SELECTLISTCHANGED:
         {
            mgrec* thisDb = (mgrec*) userData;
            // dbRec and thisDb will be dbToReport
         }
         break;
      case MNOTIFY_DATABASECLOSED:
         // dbRec may be dbToReport or any other database that was closed
         break;
   }
}

notifier1 = mgRegisterNotifier ( pluginTool, MNOTIFY_SELECTLISTCHANGED,
                           dbToReport, MG_NULL, EventNotifier, dbToReport );

notifier2 = mgRegisterNotifier ( pluginTool, MNOTIFY_DATABASECLOSED,
                           MG_NULL, MG_NULL, EventNotifier, MG_NULL );
```

In this example, two different notifiers are registered. The first reports when the Select list changes for database **`dbToReport.`** The second reports when *any* database is closed. Both events are reported to the same action function, **`EventNotifier.`** When the Select list changes in **`dbToReport,`** the API calls **`EventNotifier`** passing **`dbToReport`** as user data. This is redundant in this case because the database that caused the event to be reported is always passed to the action function.

Note: The fourth parameter to **`mgRegisterNotifer`** is the *action node.* This is ignored for most events. However, you can specify this for events such as **`MNOTIFY_NODECHANGED`**, **`MNOTIFY_NODEDELETED`** and **`MNOTIFY_NODEREPARENTED`**.

Properties and Preferences

This chapter describes two services provided by the API: *properties* and *preferences*.

Properties can be used to attach custom model-time data to database nodes, GUI elements, plug-ins and plug-in tools.

There are two kinds of preferences, modeling preferences and plug-in preferences. Modeling preferences are those defined within the Creator modeling system. Plug-in preferences are those defined by your plug-in to maintain user preference or other plug-in specific values between modeling sessions. These constructs are discussed in this chapter.

Properties

Properties are *plug-in defined data* that can be attached to database nodes, GUI elements, editor contexts, plug-ins or plug-in tools for model-time access. A property is defined by a name (key) and a value. The property name is a text string that uniquely identifies the property. The property value is a **`void*`** pointer that is the plug-in defined data attached to the database node, GUI element, plug-in or plug-in tool.

Properties can remain attached to database nodes as long as the database is open; they are not saved with the database file. Properties can remain attached to GUI elements until the corresponding GUI elements are destroyed. Properties can remain attached to an editor context only while the corresponding editor tool instance is active. Properties can remain attached to plug-ins or plug-in tools for the lifetime of the plug-in.

The API maintains separate name spaces per plug-in tool for properties attached to database nodes. In this way, different plug-in tools can each attach a property to the same database element using the same property name with no collision. The name space for properties attached to GUI elements is defined per GUI element. Since one plug-in cannot reference a GUI element created by another plug-in, separate name spaces are not required. Similarly, the name spaces for properties attached to plug-ins and plug-in tools are defined per plug-in and plug-in tool, respectively. This ensures minimal name space collisions.

Database Node Properties

Properties attached to database nodes provide similar functionality as that provided by *User Data* as discussed in volume 1 of the *OpenFlight API Developer Guide*. Both allow you to attach user-defined data to database nodes. However, since the implementation of properties does not have the contention problem inherent in that of user data (only one user data item per database node), using properties is the recommended way to attach user-defined data to database nodes for plug-ins. User data should only be used in stand-alone application code.

The functions used to manipulate properties on database nodes are listed in the following table:

mgRecPutProperty	attaches a property to a database node.
mgRecGetProperty	retrieves the value of a property attached to a database node.
mgRecDeleteProperty	removes a property from a database node.

To change the value of a property previously attached to a node via **mgRecPutProperty,** just call **mgRecPutProperty** again with the new property value. When you do, the previously attached property is simply replaced by the new one.

Note that when you call **mgRecDeleteProperty** to remove a property from a node, no action is taken on the value of the property. So if you attach a dynamically allocated property value to a node and then delete the property from the node, you are still responsible for disposing of the dynamic memory associated to the property value.

The following example illustrates how a plug-in tool might use properties to attach character string property values to database nodes:

```
mgrec* rec;
mgplugintool pluginTool;
```

```
#define MYPROPERTYNAME   "My Property Name"

...

static char* GetProperty ( mgplugintool pluginTool, mgrec* rec )
{
   return (char*) mgRecGetProperty ( pluginTool, rec, MYPROPERTYNAME );
}

static void SetProperty ( mgplugintool pluginTool,
                          mgrec* rec,
                          char* propValue )
{
   char* newPropValue = mgMalloc ( strlen(propValue) + 1 );
   char* oldPropValue = GetProperty ( pluginTool, rec );

   strcpy ( newPropValue, propValue );
   mgRecPutProperty ( pluginTool, rec, MYPROPERTYNAME, newPropValue );

   if ( oldPropValue )   // must dispose of previous value
      mgFree ( oldPropValue );
}

static void DeleteProperty ( mgplugintool pluginTool, mgrec* rec )
{
   char* propValue = GetProperty ( pluginTool, rec );
   mgRecDeleteProperty ( pluginTool, rec, MYPROPERTYNAME );
   if ( propValue )
      mgFree ( propValue );
}

...

char* propValue;

SetProperty ( pluginTool, rec, "Some Text" );
propValue = GetProperty ( pluginTool, rec );
printf ( "Property Value is : %s\n", propValue );

SetProperty ( pluginTool, rec, "Some Different Text" );
propValue = GetProperty ( pluginTool, rec );
printf ( "Property Value is now : %s\n", propValue );

DeleteProperty ( pluginTool, rec );
```

GUI Element Properties

Properties can be attached to dialogs and/or control GUI elements. The functions used to manipulate properties on GUI elements are listed in the following table:

`mgGuiPutProperty`	attaches a property to a dialog or control.

`mgGuiGetProperty`	retrieves the value of a property attached to a dialog or control.
`mgGuiDeleteProperty`	removes a property from a dialog or control.

To change the value of a property previously attached to a dialog or control via **`mgGuiPutProperty`**, just call **`mgGuiPutProperty`** again with the new property value. When you do, the previously attached property is simply replaced by the new one.

Note that when you call **`mgGuiDeleteProperty`** to remove a property from a dialog or control, no action is taken on the value of the property. So if you attach a dynamically allocated property value to a GUI element and then delete the property from the element, you are still responsible for disposing of the dynamic memory associated to the property value.

Plug-in and Plug-in Tool Properties

Properties can be attached to plug-ins and or plug-in tools. The functions used to manipulate properties on plug-ins and plug-in tools are listed in the following table:

`mgPluginPutProperty`	attaches a property to a plug-in.
`mgPluginGetProperty`	retrieves the value of a property attached to a plug-in.
`mgPluginDeleteProperty`	removes a property from a plug-in.
`mgPluginToolPutProperty`	attaches a property to a plug-in tool.
`mgPluginToolGetProperty`	retrieves the value of a property attached to a plug-in tool.
`mgPluginToolDeleteProperty`	removes a property from a plug-in tool.

To change the value of a property previously attached to a plug-in or plug-in tool via **`mgPlugin[Tool]PutProperty`**, just call **`mgPlugin[Tool]PutProperty`** again with the new property value. When you do, the previously attached property is simply replaced by the new one.

Note that when you call **`mgPlugin[Tool]DeleteProperty`** to remove a property, no action is taken on the value of the property. So if you attach a dynamically allocated property value and then delete the property, you are still responsible for disposing of the dynamic memory associated to the property value.

Editor Context Properties

As described in "Editor Context" on page 91, when an editor tool instance is active, Creator maintains an object called an *editor context* that encapsulates the state of your tool instance. An editor tool can attach properties to this editor context while the tool instance is active. The functions used to manipulate properties on an editor context are listed in the following table:

`mgEditorPutProperty`	attaches a property to an editor context.
`mgEditorGetProperty`	retrieves the value of a property attached to an editor context.
`mgEditorDeleteProperty`	removes a property from an editor context.

To change the value of a property previously attached to an editor context via **`mgEditorPutProperty,`** just call **`mgEditorPutProperty`** again with the new property value. When you do, the previously attached property is simply replaced by the new one.

Note that when you call **`mgEditorDeleteProperty`** to remove a property from an editor context, no action is taken on the value of the property. So if you attach a dynamically allocated property value to an editor context and then delete the property, you are still responsible for disposing of the dynamic memory associated to the property value.

Preferences

Modeling Preferences

Modeling preferences are values defined within the Creator modeling system that control how certain tools and functions in Creator behave. A modeling preference is defined by a name (key) and a value. The name is a text string that uniquely identifies the preference. The value is one of the pre-defined types, integer, double, or string. Depending on the type of the specific modeling preference, you will use one of the following functions to retrieve its value.

`mgModelingPrefGetInteger`	retrieves the value of an integer **`(int)`** modeling preference.
`mgModelingPrefGetDouble`	retrieves the value of a double precision floating point **`(double)`** modeling preference.
`mgModelingPrefGetString`	retrieves the value of a string **`(char*)`** modeling preference.

Each modeling preference is defined to be of a specific type, integer, double or string, and it is important to use the correct function to retrieve a preference value. The prefix of a modeling preference name indicates which type that preference value is. Integer preference values use the prefix **`MPREFI`**, double **`MPREFD`** and string **`MPREFS`**. For example, **`MPREFD_MATCHVERTEXTOLERANCE`** is a double precision floating point preference that specifies the maximum distance two vertices can be away from each other and still be considered coincident or "matching". Since this preference value is a double, **`mgModelingPrefGetDouble`** is the correct function to use.

Note: A limited number of modeling preferences are accessible. See the enumeration type **`mgmodelingprefname`** in the *OpenFlight API Reference* for a complete list of the accessible modeling preferences.

Plug-in Preferences

Plug-in preferences are text-based data stores that persist from one modeling session to the next. A plug-in can define any number of preferences that will be stored and retrieved automatically by the API. A preference is defined by a name (key) and a value. The name is a text string that uniquely identifies the preference. The value is one of several pre-defined types, integer, float (single or double precision), or string.

The API maintains separate name spaces for preferences for each plug-in module and for each plug-in tool. In this way, you can define plug-in module preferences that are global to all tools contained in your plug-in module, and at the same time, define plug-in tool preferences that are specific to individual tools contained in your plug-in module.

The API stores the preference values you define in a *preference file*. The format of this ASCII text file is similar to ".ini" style files as shown below:

```
; preferences for plug-in module
[My Plugin Module]
   Double Key = 11.000000
   Float Key = 22.000000
   Integer Key = -4
   String Key = Plugin Module String
;
; preferences for plug-in tool named My Plugin Tool 1
[My Plugin Module.My Plugin Tool 1]
   Double Key = 3.000000
   Float Key = 2.000000     ; this is an in-line comment
   Integer Key = 1
   String Key = Plugin Tool String 1
;
; preferences for plug-in tool named My Plugin Tool 2
[My Plugin Module.My Plugin Tool 2]
   Double Key = 10.000000
   Float Key = 23.000000
   Integer Key = 7
   String Key = Plugin Tool String 2
```

All characters appearing to the right of a semi-colon ("`;`") are considered comments and are ignored. This implies that preference names and string preference values cannot contain the semi-colon character. Each preference appears on its own line.

Blocks of preferences, either for a plug-in module or plug-in tool are preceded by lines bracketed by "[" and "]". For plug-in module preferences, the string contained between the brackets is the plug-in module name. For plug-in tool preferences, the string contained between the brackets is composed of the names of the plug-in module and plug-in tool separated by a dot ("."). The plug-in module name is the name specified when the plug-in it was declared using the `mgDeclarePlugin` macro. The plug-in tool name is the name specified when the tool was registered using one of the tool or data extension registration functions such as **`mgRegisterImporter, mgRegisterViewer, mgRegisterSite, etc.`**

The format of a preference line is:

```
        <key> = <value>
```

where `<key>` is the preference name and `<value>` is its value. Preference names cannot contain leading or trailing white space. Numeric preference values must adhere to formats appropriate for their type. If an integer preference value contains a fractional part, the value is truncated when it is read. String preference values are not enclosed in quotes and cannot contain leading or trailing white space.

The name of the preference file is `<library_filename>.ini` in which `<library_filename>` is the prefix of the plug-in library file containing your plug-in module. So for a plug-in module contained in `myplugin.dll`, the preference file will be named `myplugin.ini`. The preference file will be located in the same directory as your plug-in library file.

Manually editing the preference file is encouraged as long as the syntax rules are obeyed. This empowers plug-in developers to create "configurable" plug-in modules by allowing end users of Creator to "customize" the contents of these preference files. Furthermore, plug-in developers are encouraged to distribute preference files with their plug-in modules that contain default preference values.

The API reads the preference file corresponding to a plug-in module before calling the plug-in initialization function for the module. The API writes the preference file after calling the plug-in termination function for the module. The plug-in can get and set preference values at any time between initialization and termination.

When searching for the preference file corresponding to a plug-in to read, the API looks only in the directory where the plug-in library file was located. If the preference file is found, it is opened, its contents are read into internal data structures maintained by the API, and then it is closed. The API maintains the preference values throughout the modeling session automatically.

Note: The preference file does not remain open and is not re-read during the modeling session.

During the modeling session, a plug-in will get and set preference values by name. If a plug-in tries to retrieve a preference value that does not yet exist, the API will automatically create the preference and assign it the specified default value. There are two reasons that a preference may not exist when the plug-in attempts to retrieve it. First, a preference file may not have existed when the plug-in was loaded. This will probably be the case the very first time a plug-in is seen by Creator. Second, the preference may have been added to a newer version of the plug-in. This is likely to happen the first time a newer version of a plug-in is seen by Creator.

Plug-in Module Preferences

The following functions retrieve different types of preference values associated to a given plug-in module. Each of these functions will create the preference and return a specified default value if the preference does not yet exist.

`mgPluginPrefGetString`	retrieves the value of a string **`(char*)`** preference associated to a given plug-in module.
`mgPluginPrefGetInteger`	retrieves the value of an integer **`(int)`** preference associated to a given plug-in module.
`mgPluginPrefGetBool`	retrieves the value of an **`mgbool`** preference associated to a given plug-in module.
`mgPluginPrefGetFloat`	retrieves the value of a single precision floating point **`(float)`** preference associated to a given plug-in module.
`mgPluginPrefGetDouble`	retrieves the value of a double precision floating point **`(double)`** preference associated to a given plug-in module.

The following functions set different types of preference values associated to a given plug-in module. Each of these functions will create the preference if the preference does not yet exist.

`mgPluginPrefSetString`	sets the value of a string **`(char*)`** preference associated to a given plug-in module.
`mgPluginPrefSetInteger`	sets the value of an integer **`(int)`** preference associated to a given plug-in module.
`mgPluginPrefSetBool`	sets the value of an **`mgbool`** preference associated to a given plug-in module.
`mgPluginPrefSetFloat`	sets the value of a single precision floating point **`(float)`** preference associated to a given plug-in module.
`mgPluginPrefSetDouble`	sets the value of a double precision floating point **`(double)`** preference associated to a given plug-in module.

Plug-in Tool Preferences

The following functions retrieve different types of preference values associated to a given plug-in tool. Each of these functions will create the preference and return a specified default value if the preference does not yet exist.

`mgPluginToolPrefGetString`	retrieves the value of a string **`(char*)`** preference associated to a given plug-in tool.
`mgPluginToolPrefGetInteger`	retrieves the value of an integer **`(int)`** preference associated to a given plug-in tool.
`mgPluginToolPrefGetBool`	retrieves the value of an **`mgbool`** preference associated to a given plug-in tool.

`mgPluginToolPrefGetFloat`	retrieves the value of a single precision floating point **`(float)`** preference associated to a given plug-in tool.
`mgPluginToolPrefGetDouble`	retrieves the value of a double precision floating point **`(double)`** preference associated to a given plug-in tool.

The following functions set different types of preference values associated to a given plug-in module. Each of these functions will create the preference if the preference does not yet exist.

`mgPluginToolPrefSetString`	sets the value of a string **`(char*)`** preference associated to a given plug-in tool.
`mgPluginToolPrefSetInteger`	sets the value of an integer **`(int)`** preference associated to a given plug-in tool.
`mgPluginToolPrefSetBool`	sets the value of an **`mgbool`** preference associated to a given plug-in tool.
`mgPluginToolPrefSetFloat`	sets the value of a single precision floating point **`(float)`** preference associated to a given plug-in tool.
`mgPluginToolPrefSetDouble`	sets the value of a double precision floating point **`(double)`** preference associated to a given plug-in tool.

The following code fragment shows how a plug-in might access the preferences shown in the sample preference file at the beginning of this section. In this example, there are two plug-in tools contained in the plug-in module. The plug-in module name is **`"My Plugin Module".`** The plug-in tool names are **`"My Plugin Tool 1"`** and **`"My Plugin Tool 2":`**

```
mgplugin plugin;          // module name is “My Plugin Module”
mgplugintool pluginTool_1;  // tool name is “My Plugin Tool 1”
mgplugintool pluginTool_2;  // tool name is “My Plugin Tool 2”

...

char stringPrefValue [100];
int intPrefValue;
float floatPrefValue;
double doublePrefValue;

   // stringPrefValue will be: “Plugin Module String”
mgPluginPrefGetString ( plugin, “String Key”, ,
          sizeof(stringPrefValue), “Default String” );

   // intPrefValue will be: -4
mgPluginPrefGetInteger ( plugin, “Integer Key”, &intPrefValue, 10 );


   // intPrefValue will be: 1
mgPluginToolPrefGetInteger ( pluginTool_1, “Integer Key”, &intPrefValue, 0 );

   // floatPrefValue will be: 2.0
mgPluginToolPrefGetFloat ( pluginTool_1, “Float Key”, &floatPrefValue, 2.0f );


   // doublePrefValue will be: 10.0
mgPluginToolPrefGetDouble ( pluginTool_2, “Double Key”, &doublePrefValue, 1.0 );

   // floatPrefValue will be: 23.0
mgPluginToolPrefGetFloat ( pluginTool_2, “Float Key”, &floatPrefValue, 2.0f );

   // intPrefValue will be default value: 34
mgPluginToolPrefGetInteger ( pluginTool_2, “New Key”, &intPrefValue, 34 );
```

Palettes

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.

Creator displays graphical representations of the palettes belonging to the top database in palette windows. For example, the Color palette window in Creator displays the color palette of the top database.

The API provides a function to display any of the palettes listed below, **`mgShowPalette.`** It takes a single parameter specifying which palette you want to display. If the specified palette is not already displayed, it will be. If the palette is already displayed but occluded by another window, it will be brought to the top of the window order on the desktop.

Each of the palette windows has a notion of *current palette item*. The current palette item is the item in the palette that is currently selected by the user. It is always visually differentiated from the other items in the palette window in some way. The current texture, for example is drawn in the Texture palette window with a thick red outline. Some tools in Creator utilize current palette items when they perform their processing. The **Put Texture** command, for example, applies the current texture to the selected database nodes. The indices of the current palette items are not saved with the OpenFlight database file.

The API provides functions to get and set current palette items for the following database palettes:

- Color (both primary and alternate)
- Material
- Texture
- Texture Mapping
- Sound
- Light Source
- Light Points

- Shader

Color Palette

A database color 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. Therefore, color palette items are referred to by index *and* intensity.

The Color palette window defines a current primary color as well as a current alternate color. The table below describes the functions that get and set the current primary and alternate color for a database:

`mgGetCurrentColor`	returns the index and intensity of the current primary color.
`mgSetCurrentColor`	sets the current primary color to the item in the palette at the specified index and intensity.
`mgGetCurrentColorRGB`	Rrturns the RGB components of the current primary color.
`mgSetCurrentColorRGB`	sets the current primary color to an index and intensity that best matches the specified RGB values.
`mgGetCurrentAltColor`	returns the index and intensity of the current alternate color.
`mgSetCurrentAltColor`	sets the current alternate color to the item in the palette at the specified index and intensity.
`mgGetCurrentAltColorRGB`	returns the RGB components of the current alternate color
`mgSetCurrentAltColorRGB`	sets the current alternate color to an index and intensity that best matches the specified RGB values.

Material Palette

Material palette items are referred to by index. The Material palette window defines a current material item. The table below describes the functions that get and set the current material for a database:

`mgGetCurrentMaterial`	returns the index of the current material.
`mgSetCurrentMaterial`	sets the current material to the item in the palette at the specified index.

Texture Palette

Texture palette items are referred to by index. Subtextures within texture palette items are also referred to by index. The Texture palette window defines a current texture item as well as a current subtexture within that item (if one exists). The table below describes the functions that get and set the current texture and subtexture for a database.

`mgGetCurrentTexture`	returns the index of the current texture.
`mgSetCurrentTexture`	sets the current texture to the item in the palette at the specified index.
`mgGetCurrentSubTexture`	returns the index of the current subtexture of the current texture.
`mgSetCurrentSubTexture`	sets the index of the current subtexture of the current texture.

Texture Mapping Palette

Texture mapping palette items are referred to by index. The Texture Mapping palette window defines a current texture item. The table below describes the functions that get and set the current texture mapping for a database:

`mgGetCurrentTextureMapping`	returns the index of the current texture mapping.
`mgSetCurrentTextureMapping`	sets the current texture mapping to the item in the palette at the specified index.

Sound Palette

Sound palette items are referred to by index. The Sound palette window defines a current sound item. The table below describes the functions that get and set the current sound for a database:

`mgGetCurrentSound`	returns the index of the current sound.
`mgSetCurrentSound`	sets the current sound to the item in the palette at the specified index.

Light Source Palette

Light Source palette items are referred to by index. The Light Source palette window defines a current light source item. The following table describes the functions that get and set the current light source for a database:

`mgGetCurrentLightSource`	returns the index of the current light source.
`mgSetCurrentLightSource`	sets the current light source to the item in the palette at the specified index.

Light Points Palette

There are two palettes associated with Light Points, the Light Point Appearance palette and the Light Point Behavior palette. Both Light Point Appearance and Behavior 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 behavior item. The following table describes the functions that get and set the current light point appearance and behavior for a database:

`mgGetCurrentLightPointAppearance`	returns the index of the current light point appearance index.
`mgGetCurrentLightPointBehavior`	returns the index of the current light point behavior index.
`mgSetCurrentLightPointAppearance`	sets the current light point appearance to the item in the palette at the specified index.
`mgSetCurrentLightPointBehavior`	sets the current light point behavior to the item in the palette at the specified index.

Shader Palette

Shader palette items are referred to by index. The Shader palette window defines a current shader item. The table below describes the functions that get and set the current shader for a database:

`mgGetCurrentShader`	returns the index of the current shader.
`mgSetCurrentShader`	sets the current shader to the item in the palette at the specified index.

Online Help

Plug-in developers can integrate online help components into the Creator program environment for the modules they create. Using API functions, developers can add help files and tool tips for their plug-in tools. Help files added in this way are integrated directly into the Creator online help system. Similarly, tool tips added by a plug-in tool are accessed by the user just like those that exist for any internal Creator tool. This chapter describes how to add help files and tool tips for plug-in modules.

Help Files for Plug-ins

Creator provides an online help system through which the user can obtain useful information about how to use different components of the modeling system. Users can navigate this help system in many different ways. This section describes how developers can integrate the help files they create for their plug-ins into the Creator help system.

Following are the minimum steps necessary for adding a help file for a plug-in:

1. Create a help file. Your help file can be any format that has an application associated with it. On Windows, for example if you can 'open' the file in Windows Explorer, it will work as a Creator help file. The API provides additional support for two specific kinds of help files, Windows Help and HTML. You can name your help file any name you like but it will require less work on your part if you name it according to the following convention:

   ```
   <plug-in name>.<extension>
   ```

2. where **`<plug-in name>`** is the prefix of your plug-in module and **`<extension>`** is either **`<hlp>`** or **`<htm>`** (or whatever extension the file is) according to the format of its contents. For example if your plug-in module is named myplugin.dll and you create a Windows help file for it, you should name the help file myplugin.hlp.

3. Place the help file in the same directory where your plug-in module is installed and loaded into Creator. See "The Plug-in Runtime Directory"

on page 260 for more information on how Creator locates this directory at runtime.

If you name and install your help file according to the steps listed above and it is of type **`<hlp>`** or **`<htm>`** Creator will automatically locate and load your help file after your plug-in module is loaded. If your help file is not **`<hlp>`** or **`<htm>`**, you choose to name your help file some other name or install it a directory other than where your plug-in module is installed, you can use the function **`mgRegisterHelpFile`** to inform Creator where to locate your help file. If you explicitly specify the location of your help file using this function, you must do so within your plug-in initialization function **`mgpInit.`**

After your help file is loaded (whether Creator locates it in the default location or you have explicitly specified its location), it will become accessible within Creator in the Help On Plugins window during the modeling session.

Note: If Creator locates both a Windows help file and an HTML file when searching for your help file, the Windows help file will take precedence. For example if Creator finds both **`myplugin.hlp`** and **`myplugin.htm`** in the same directory as **`myplugin.dll`**, **`myplugin.hlp`** will be used.

To further integrate your help file into the modeling environment you can use API functions to set up more granular context sensitive associations between user interface items in Creator and topics contained in your help file. The following section provides more details on how to do this.

Context Sensitive Help

Creator users can access context sensitive help topics using the **Help** command **(F1)** clicking on the user interface element for which help is desired. In the remainder of this section, this technique will be referred to as **F1 click.** Using this technique, users click on any user interface element within the Creator application and expect to get some level of context sensitive help. When a user interface element is clicked in this way, Creator queries the user interface element to determine if a context sensitive help topic has been associated to that element. This section describes how plug-in developers can associate context sensitive topics within their help file to user interface elements.

The API refers to context sensitive help topics as *help contexts* and defines them using character strings. A help context is expected to uniquely identify a specific location within a help file. The following type definition is declared by the API to refer to help contexts.

```
typedef char* mghelpcontext;
```

The levels for which you can specify help contexts are listed below. With each level is a description of how the Creator user accesses the help contexts assigned to that level.

1 Plug-in tool

   F1 click on tool icon in toolbox corresponding to editor or viewer tool.

   F1 click on menu item corresponding to editor or viewer tool.

   Help On Plugins window - select plug-in tool from list and click **Help** button.

2 Plug-in tool dialog

   F1 click on control that does not have an explicit help context assigned itself.

3 Plug-in tool dialog control

   F1 click on control.

To associate a help context to an individual plug-in tool, you must specify a string value for the **Help Context** tool attribute **`MTA_HELPCONTEXT`** when the tool is registered. The string value you specify will be used to locate a specific location in your plug-ins help file when context sensitive help is requested for the plug-in tool.

The following code fragment shows how a help context might be assigned to an editor plug-in tool when the tool is registered:

```
pluginTool = mgRegisterEditor (
   plugin, “My Editor”,
   EditorStartFunc, MG_NULL,
   MTA_VERSION, “1.0”,
   MTA_PALETTELOCATION, MPAL_FACETOOLS,
   MTA_PALETTEICON, pixmap,
   MTA_HELPCONTEXT, “My Editor Topic”,
   MG_NULL
   );
```

In this example, a help context identified by the string **`“My Editor Topic”`** is associated to the editor tool. The help file location corresponding to this help context is displayed automatically when the user **F1 clicks** on the tool icon for this editor or clicks the **Help** button in the **Help On Plugins** window.

To associate a help context to a specific dialog or control GUI element, you can use the function **`mgGuiSetHelpContext`.** Again, the help context string you specify will be used to locate a specific location in your help file when context sensitive help is requested for the GUI element. You can define separate help contexts for any of the controls in your plug-in tool dialog or for the dialog itself. For all controls that do not have explicit help contexts assigned, the help context assigned to the parent dialog will be used.

Plug-ins can also display context sensitive help within the Creator help system by calling the function **`mgShowHelpContext`.** For example, you might include a push button on your plug-in tool dialog labeled **Help** that when pushed, causes the Creator help system to be invoked with a particular help context. The following code fragment shows how this might be implemented within the dialog corresponding to the editor tool shown in the previous example:

```
static mgstatus HelpCallback ( mggui gui, mgcontrolid controlId,
                          mgguicallbackreason callbackReason,
                          void *userData, void *callData )
{
   mgShowHelpContext ( Plugin, "My Editor Topic" );
   return (MSTAT_OK);
}

static void InitializeControlCallbacks ( mggui dialog )
{
   mggui gui = mgFindGuiById ( dialog, HELPBUTTON );
   mgSetGuiCallback ( gui, MGCB_ACTIVATE, HelpCallback, MG_NULL );
}
```

Tool Tips

Creator displays tool tips when the user positions the mouse pointer over a control and does not move the mouse for a certain amount of time. Plug-ins can assign tool tip strings for any control contained in a dialog they create. In addition, editor and viewer tools that are launched from a tool icon in a toolbox, can assign a tool tip string for their corresponding tool icon.

To assign a tool tip string to a control in a dialog, use the function **`mgGuiSetToolTip`** as shown in the following code fragment:

```
static void InitializeControlCallbacks ( mggui dialog )
{
   mggui gui = mgFindGuiById ( dialog, MYBUTTON );
   mgSetGuiCallback ( gui, MGCB_ACTIVATE, MyCallback, MG_NULL );
   mgGuiSetToolTip ( gui, “My Tool Tip” );
}
```

To assign a tool tip string to a tool icon corresponding to an editor or viewer tool, you must specify a string value for the **Tool Tip** tool attribute **`MTA_TOOLTIP`** when the tool is registered. The following code fragment shows how a tool tip string might be assigned to an editor plug-in tool when the tool is registered:

```
   pluginTool = mgRegisterEditor (
      plugin, “My Editor”,
      EditorStartFunc, MG_NULL,
      MTA_VERSION, “1.0”,
      MTA_PALETTELOCATION, MPAL_FACETOOLS,
      MTA_PALETTEICON, pixmap,
      MTA_TOOLTIP, “My Tool Tip”,
      MG_NULL
      );
```

Licensing a Plug-in Module

There may be situations in which you want to control access to the tools you provide in your plug-in. For example, if you develop a commercial plug-in, you may want to allow access to only those users that have purchased a license. The granularity at which a plug-in controls access to the tools it contains can vary with each plug-in. This chapter describes several common plug-in licensing scenarios and describes different mechanisms and strategies to support them.

Plug-in Access

In "Plug-in Source Code Overview" on page 19, it was noted that the value returned from the initialization function of a plug-in determines whether or not the plug-in is loaded. In other words, a plug-in returns **`MG_TRUE`** if it is ok to load, **`MG_FALSE`** otherwise. Using this mechanism, a plug-in can allow or deny access to itself (in whole) by simply returning the appropriate value from its plug-in initialization function. This is perhaps the most simple form of licensing and is illustrated here:

```
MGPIDECLARE(mgbool)
mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgbool licenseOk;
   // register plug-in objects and do other initialization
   InitializeMyPlugin ( plugin );

   // check if "licensed" to run
   licenseOk = CheckForLicense ( plugin );
   return licenseOk;
}
```

In this example, a plug-in defined function **`CheckForLicense`** is used to determine whether or not the functionality contained in the plug-in should be accessible. The return value of this function determines whether or not the plug-in is loaded.

Plug-in Tool Access

A plug-in that contains more than one plug-in tool may want to control access to each tool independently. You may want to allow the user to access one tool but not others. In this case, it is not enough to allow or deny access to the entire plug-in as shown in the previous section. More granularity is needed.

In the following example, a plug-in contains two plug-in tools. The first is an "introductory" version, which is freely available. The second is an "advanced" commercial version that must be purchased. The plug-in defines a function **`CheckForAdvancedLicense`** that checks to see if a license exists to access the commercial version of the tool. If so, the "advanced" version of the tool is loaded. Otherwise, the introductory (presumably limited) version of the tool is loaded. In either case, the plug-in initialization function returns **`MG_TRUE`** to allow access to whichever version of the tool was loaded.

```
MGPIDECLARE(mgbool)
mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgbool advancedLicenseOk;

   // check if "licensed" to run advanced tool
   advancedLicenseOk = CheckForAdvancedLicense ( plugin );

   // register the "advanced" version only if allowed
   if ( advancedLicenseOk )
      InitializeAdvancedTool ( plugin );
   else
      InitializeIntroductoryTool ( plugin );

   return MG_TRUE;
}
```

Functional Access

The example in the section above shows just one way in which access to different levels of functionality could be controlled by a plug-in. There are certainly many other mechanisms to do this, far too many to enumerate here. All such mechanisms involve some form of conditional access to the code in your plug-in.

Here are a few suggestions:

- Register different Start Functions when calling **`mgRegisterImporter`**, **`mgRegisterExporter`**, **`mgRegisterEditor`**, or

`mgRegisterViewer,` depending on the level of functionality you want to expose for a particular tool.

- Use the function **`mgSetVisible`** to hide specific controls in your dialog. For example, if a function is only accessible when a user presses a specific button control, use **`mgSetVisible`** to hide that button when you want to deny access to that functionality.
- Similar to **`mgSetVisible,`** use **`mgSetEnabled`** to disable specific controls in your tool dialogs without physically 'hiding' them.
- Bracket critical sections of your code using conditional statements depending on the level of functionality you want to expose.

Presagis Defined Licenses

The examples shown thus far in this chapter are all based on plug-in defined functions (e.g. **`CheckForLicense`** and **`CheckForAdvancedLicense`**) to determine what level of functionality to expose to the end user. In these examples, it is assumed that the plug-in developer completely manages the sale, distribution and support of any formal licensing mechanism directly with the end user and that these plug-in defined functions to check for valid licenses are tightly integrated with that licensing mechanism.

For plug-in developers that choose to manage the licensing mechanism themselves, the sections above offer some suggestions on how your code might be structured to support whatever licensing mechanism you choose to implement. The sections above do not, however, suggest any specific licensing mechanism. You are, of course, free to implement licensing using a commercial licensing package or develop your own scheme.

Presagis does offer a formal licensing system for qualified plug-in developers. Under this system, Presagis creates the licenses and distributes them directly to the end users on behalf of the plug-in developer. If you are interested in this licensing mechanism, please contact Presagis for more information.

Developers that employ this licensing system receive a unique "license name" (assigned by Presagis) for each individually licensable "entity" in their plug-in. A developer might want one license name for the entire plug-in or separate license names for each tool in the plug-in or some other combination of license names. Presagis maintains these license names to ensure they are unique for all registered plug-ins. As mentioned above, Presagis also distributes these licenses to end users and integrates these licenses into the Creator licensing mechanism.

Although Presagis manages nearly all aspects of this licensing mechanism, the plug-in developer is responsible for integrating the licensing API provided by the OpenFlight API into the plug-in source code.

The OpenFlight API represents 'licenses' using the **`mglicense object.`** A license object is created using the function **`mgNewLicense.`** When you create a license, you supply the 'license name' assigned to your licensable entity by Presagis. After a license is created, it can be 'checked out' and 'checked in' any number of times. To check out a license object, use the function **`mgCheckoutLicense.`** To check it in, use **`mgCheckinLicense.`** When a license is no longer needed, it is destroyed using **`mgFreeLicense.`** Note that **`mgFreeLicense`** does not check a license in if it is checked out so be sure to check it in before destroying it.

The following example shows how a plug-in might use this licensing API:

```
static mglicense MyLicense = MG_NULL;

// Plug-in initialization function
MGPIDECLARE(mgbool) mgpInit ( mgplugin plugin, int* argc, char* argv [] )
{
   mgbool isLicensed = MG_FALSE;
   mgstatus checkoutStatus;

   // create the license...
   MyLicense = mgNewLicense ( "myLicenseName" );

   // ...try to check it out
   checkoutStatus = mgCheckoutLicense ( MyLicense );
   isLicensed = MSTAT_ISOK ( checkoutStatus );

   if ( isLicensed ) {
      // register tools declared by this plugin
      RegisterMyTools ( plugin, argc, argv );
   }
   // return MG_TRUE to continue loading...
   // ...MG_FALSE to tell Creator to unload plugin
   return ( isLicensed );
}

// Plug-in termination function
MGPIDECLARE(void) mgpExit ( mgplugin plugin )
{
   mgUnregisterAllTools ( plugin );
   // done with license, check it in...
   mgCheckinLicense ( MyLicense );
   // ...and free it
   mgFreeLicense ( MyLicense );
   MyLicense = MG_NULL;
}
```

The plug-in initialization function creates and attempts to check out a license. In this case, the license name assigned by Presagis to this plug-in is 'myLicenseName.' If this license is installed and available on the user's system, **`mgCheckoutLicense`** will succeed, in which case the initialization function registers the plug-in tools and returns **`MG_TRUE.`** This allows the plug-in to get loaded. Note that the license object is stored in the static global object **`MyLicense.`**

The plug-in termination function checks the license in and destroys it.

Building a Plug-in Module

Once you have created the **`C/C++`** source files that contain the definitions of the plug-in objects you want to define in your plug-in module, you are ready to compile the code and create a library component.

Note: The OpenFlight API is distributed for different compilers and architectures on the Windows platform. You must use the VC8 Win32 version to build plug-ins compatible with Creator.

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`** and **`API_LEV4`**
  On Linux: `API_LEV4`
- Define **`Include Directory`** for your compiler:
  On Windows: **`$(PRESAGIS_OPENFLIGHT_API)\include`**
  On Linux: **`$(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: `mgapilib.lib` and `mgdd.lib`
  On Linux: **`libmgapilib.so`** and **`libmgdd.so`**
- Define **`Library Path`** for your linker:
  On Windows: **`$(PRESAGIS_OPENFLIGHT_API)\lib`**
  On Linux: **`$(PRESAGIS_OPENFLIGHT_API)/lib`**

In the lines above, **`PRESAGIS_OPENFLIGHT_API`** is an environmental variable whose value is the directory where the API was installed.

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

```
C:\Presagis\OpenFlightAPI_<version>
```

where **`<version>`** is the version of the API you have installed. For example, version 4.2 of the API would be installed to:

```
C:\Presagis\OpenFlightAPI_4_2
```

The API installer creates the environmental variable `PRESAGIS_OPENFLIGHT_API` automatically.

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

```
/usr/local/Presagis/OpenFlightAPI_<version>
```

where `<version>` is the version of the API you have installed. For example, version 4.2 of the API would be installed to:

```
/usr/local/Presagis/OpenFlightAPI_4_2
```

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

```
/usr/local/Presagis/OpenFlightAPI_<version>/SOURCEME
```

for more information on setting `PRESAGIS_OPENFLIGHT_API.`

Depending on the platform on which you are developing your plug-in, you will do the things listed here in slightly different ways. The following two sections provide detailed instructions for setting up your build environment on the two platforms, Windows and Linux.

Windows

As noted above, the OpenFlight API is distributed in several binary formats for the Windows platform; VC6, VC8 Win32 and VC8 x64. Only one can be installed at any one time on the same computer. If you are developing plug-ins for Creator, you must use VC8 Win32 so your plug-ins are compatible with Creator. If you are developing stand-alone applications, you can choose either VC6, VC8 Win32 or VC8 x64.

When you build your plug-in module on Windows, you will actually build two library files, a *dynamic link library* file (`.dll`) and an *import library* file (`.lib`). The dynamic link library contains exported symbols, or *exports*, that will be accessed by the plug-in runtime environments. Throughout this document, when referring to a plug-in module on Windows systems, the dynamic link library file is the file that is referred to. The import library file contains information about the exports contained in the dynamic link library file and is needed only at the time you link a stand-alone program (or other plug-in) that accesses data extensions or functions contained in your plug-in module. Once your program is built, the import library file is not needed in the runtime environment. This will be discussed in more detail in "Stand-Alone Programs" on page 261.

The following sections provide detailed instructions for setting up your build environment on Windows systems. These instructions are specific to Visual Studio 2005. Instructions for previous or subsequent versions of Visual Studio are similar but may not be identical.

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

You may also find it useful to examine the sample plug-ins and extensions 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 compilation and link settings described in this section.

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 previous versions of the 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

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

You must tell the linker where to find the API import library files. 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.
```

Do this for all Configurations as necessary.

Additional Setup

It is very important that your plug-in module be linked with the correct system run-time libraries. In Visual Studio, open the **Project Properties** window and select `Configuration Properties > C/C++ > Code Generation.` In **Runtime Library,** select `Multi-threaded Debug DLL` for the Win32 Debug configuration and `Multi-threaded DLL` for the Win32 Release configuration.

Linux

When you build your plug-in module on Linux, you will only build one library file, a *shared object* file (`.so`). The shared object file contains exported symbols, or *exports*, that will be accessed by the plug-in runtime environments. Throughout this document, when referring to a plug-in module on Linux systems, the shared object file is the file that is referred to. The shared object file also serves as the *import library* when you link a stand-alone program (or other plug-in) that accesses data extensions or functions contained in your plug-in module. This will be discussed in more detail in "Stand-Alone Programs" on page 261.

You may also find it useful to examine the sample extensions included with the OpenFlight API distribution. These samples include source code as well as **Makefiles** showing you how to set the compilation and link settings described in this section.

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 plugins 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/OpenFlightAPI_<version>/include

If you use the PRESAGIS_OPENFLIGHT_API environmental variable you can specify the compiler option **-I$(PRESAGIS_OPENFLIGHT_API)/include.** If you do not use the PRESAGIS_OPENFLIGHT_API environmental 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.**

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/OpenFlightAPI_<version>/lib

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

Plug-ins in the Runtime Environment

How Plug-ins are Loaded

Plug-ins are loaded into the Creator and stand-alone program runtime environments. In order for a plug-in to be available in either environment, it must be located in or below the *plug-in runtime directory*. This is a platform specific directory that is accessible at runtime. When the runtime environment system starts up, this directory and all directories below this directory will be searched for compatible plug-ins. All compatible plug-ins found will be loaded and will be accessible in the runtime environment.

Since plug-in modules may be distributed with a host of companion files (preference file, help file, additional library files, etc), it may be convenient to install each plug-in in its own subdirectory below the plug-in runtime directory. This is possible since the plug-in runtime directory is searched recursively.

Creator loads plug-ins automatically when it starts. Plug-ins are loaded by stand-alone programs when your program calls the API function **`mgInit.`**

Debugging Plug-ins in Creator

When debugging your plug-in initialization code on the Windows platform, you may find that the Creator splash screen occludes your workspace. To avoid this, you can disable the Creator splash screen by setting the environment variable **`PRESAGIS_CREATOR_NOSPLASH to 1, TRUE`** (or **`true)`** before running Creator. To re-enable the splash screen, un-set this environment variable or set it to **`0, FALSE`** (or **`false).`**

Note: After you change an environment variable on the Windows platform, you must exit and restart your development environment (Developer Studio, etc.) before the change you make takes effect in your environment.

The Plug-in Runtime Directory

Plug-ins are loaded in the stand-alone program environment when your program or script calls the API function **`mgInit`**. **`mgInit`** searches for plug-ins in locations listed below (in the order listed). As soon as a folder is found in one of these locations, **`mgInit`** stops searching for other locations and loads the plug-ins it finds in the first folder found.

Below is the directory search order for loading plug-in modules in the stand-alone program environment:

1. The directory (or directories) your program specifies when calling **`mgSetPluginFolder`**. If you call this function before **`mgInit`** to specify one or more directories, the folder(s) you specify will be searched for plug-ins. To specify more than one directory, pass a semi-colon delimited list of paths to **`mgSetPluginFolder.`**
2. The directory (or directories) specified by the environmental variable **`PRESAGIS_OPENFLIGHT_PLUGIN_DIR`**. This environmental variable may specify any accessible directory on your computer or may specify multiple directories using a semi-colon to delimit each path.
3. A directory named **`plugins`** immediately below the directory that contains the executable being run.

Creator loads plug-ins automatically when it starts. Creator searches for plug-ins in the locations listed below (in the order listed). As soon as a folder is found in one of these locations, Creator stops searching for other locations and loads the plug-ins it finds in the first folder found.

Below is the directory search order for loading plug-in modules in the Creator modeling environment:

1. The directory (or directories) specified by the environmental variable **`PRESAGIS_CREATOR_PLUGIN_DIR`**. This environmental variable may specify any accessible directory on your computer or may specify multiple directories using a semi-colon to delimit each path
2. A directory named plugins in the Creator configuration folder. This folder is located at:
   `PRESAGIS_ROOT/Creator/config/plugins`

Stand-Alone Programs

Volume 1 of the *OpenFlight API Developer Guide* describes how to set up your stand-alone program environment so that your application can locate the API libraries at runtime. The description provided there is sufficient if your application only accesses OpenFlight database elements and does not access data extension elements. This section describes the additional set up you must perform when building and running your stand-alone application so it can access data extensions declared in plug-ins.

Important: Stand-alone programs can only access data extensions contained in plug-ins. Any plug-in tools declared by a plug-in loaded into the stand-alone program runtime environment will be ignored.

Building a Stand-Alone Program that accesses Data Extensions

If your data extension plug-in only contains symbols defined by your data dictionary (i.e., does not export any functions or other symbols needed by your stand-alone program) you do not need to do any other setup before building your stand-alone program. Symbols defined in your data dictionary are dynamically loaded and become accessible automatically when your plug-in is loaded. This means that your stand-alone program does not have to explicitly link with your plug-in module(s). This mechanism for dynamic binding of data dictionary symbols is new for Creator and the OpenFlight API version 2.3 and makes the linking process much simpler. Previous versions required that your program explicitly link with your data extension plug-in.

However, if your plug-in defines functions or other symbols that your stand-alone program needs to access, then you must link your program with your plug-in just like any other library containing symbols your program uses. Most plug-ins do not do this but if yours does, follow the instructions contained in this section to link your stand-alone program explicitly with your data extension (or other) plug-in.

On Windows: In Visual Studio, open the **Project Settings** window, select the **Link** tab and then select the **Input** category. In the **`Object/library modules`** field, make sure the import library file corresponding to your plug-in is included: **`<myplugin>.lib,`** where **`<myplugin>`** is the name of your plug-in. In the `Additional library path` field, add the path to the library directory where your plug-in import library file is located. Do this for All Configurations (Win32 Debug and Win32 Release).

On Linux: Tell the linker to link with your plug-in import library (shared object) file with the link option `-l<myplugin>`, where `<myplugin>` is the name of your plug-in. Tell the linker where to find your plug-in import library with the link option `-L` option specifying where your plug-in import library file is located.

Running a Stand-Alone Program that accesses Data Extensions

Again, if your data extension plug-in only contains symbols defined by your data dictionary (i.e., does not export any functions or other symbols needed by your stand-alone program) you do not need to do any other setup before building your stand-alone program. Symbols defined in your data dictionary are dynamically loaded and become accessible automatically when your plug-in is loaded. This means that your stand-alone program does not have to explicitly link with your plug-in module(s). This mechanism for dynamic binding of data dictionary symbols is new for Creator and the OpenFlight API version 2.3 and makes the linking process much simpler. Previous versions required that your program explicitly link with your data extension plug-in.

Plug-ins will be loaded *explicitly* at runtime if it is located in the plug-in runtime directory, the data extension symbols it declares cannot be accessed directly by your stand-alone application unless the application can locate the plug-in *implicitly* at runtime. It is very important that the plug-in library file loaded explicitly by the runtime environment be the same plug-in library file that loads implicitly when your application runs.

This section describes the extra steps you must take before you run your stand-alone application so that it can correctly locate the external symbols declared by your plug-in at runtime.

On Windows: Add the location of your plug-in runtime directory to the `PATH` environment variable. Open the **System** properties window in the **Control Panel.** Click the **Environment** tab and select the `PATH` variable in the User Variable list. Type the location of the plug-in runtime directory in the Value field. If there is more than one path, each must be delimited by a semi-colon **`(;).`**

On Linux: Add the location of your plug-in runtime directory to the **`LD_LIBRARY_PATH`** environment variable. See the *tcsh* script file included with the API distribution:
`/usr/local/Presagis/OpenFlightAPI_<version>/SOURCEME`
for more information on setting **`LD_LIBRARY_PATH.`**

Important: If your stand-alone program implicitly loads one physical copy of your plug-in module and the runtime environment explicitly loads another, the data extension symbols referenced by your stand-alone program will not get initialized properly. This will render your data extensions inaccessible to your stand-alone application.

Data Dictionary Keywords

The data dictionary parser, **ddbuild,** supports a set of defined keywords that you use to define an entry in your data dictionary. They are described below.

alias	Defines a list of record types to which another record can be cast. Used to represent a `(void*).`
dataDef	Defines a new field. *Fields* are used to represent node attributes.

Options

Option	Definition
DEF=val	defines the default value for this field.
NODISP	gives the instructions "Do not display this field in the Creator attribute page."
NOEDIT	defines a read-only attribute in the attribute page.
ENUM	works with fields of type `MTYPE_FLAG` to identify that several fields work together as the different values of an enumerated type. Use DEF=1 to specify which flag is on by default; use DEF=0 for all the others.
KEY=n	allows you to define up to three different levels of access. **ddbuild** prompts you for the level you want to process and converts to header files or documentation. By convention, key 0 is public, key 1 is the next level, and key 2 includes all data in the dictionary. If no key is specified, a field or record is considered public.
LABEL="string"	is a quoted string used to generate the Creator user interface label (in the attribute page, for example). If you do not specify a label, Creator uses the field name by default.
LEN=n	is used to specify the length of a field, in bytes. In general, the length is auto-calculated. However, there are two cases in which you must specify a length by hand: 1) with text strings to specify the number of letters. If you don't use the `LEN` keyword to specify a text string's length, you will have a 0 length string (a 1 character string which is the null terminator). 2) with `ENUM` structures, to identify how many different enumerated values go together.
MAX=val	sets the maximum value that is valid for a field. For example, you might want to set a color field's maximum to 255.
MIN=val	sets the minimum value that is valid for a field. For example, you may want to set a color field's minimum to 0.

Option	Definition
NOSEARCH	disables attribute search for this field in Creator.
XCODE=code	identifies which type of node to attach the field or record to.

recordDef Defines a new record from a list of fields. A *record* is a group of fields. You can define a record that contains several attributes for a node (a logical grouping of *Tag-Along* extensions), or you can define a record type that is itself a node type (a *Stand-Alone* extension).

Options

Option	Definition
ALIAS=name	identifies the structure that lists the valid record types represented by a generic pointer. *Name* identifies an alias definition elsewhere in the data dictionary file.
CHILD=name	used with new node types, to define the name of the *child* attachment rules. The child list defines which node types can be the children of this node type. If you don't specify this option, any node type can be a child.
NODISP	gives the instructions "Do not display this record in the Creator attribute page."
NOEDIT	defines a read-only record in the attribute page.
KEY=n	allows you to define up to three different levels of access. `ddbuild` prompts you for the level you want to process and converts it to header files or documentation. By convention, key 0 is public, key 1 is the next level, and key 2 includes all data in the dictionary. If no key is specified, a field or record is considered public.
LABEL="string"	a quoted string used to generate the Creator user interface label (in the attribute page, for example). If you do not specify a label, Creator will use the field name by default.
PARENT=name	used with new node types, to define the name of the *parent* attachment rules. The parent list defines which node types can be the parent of this node type. If you don't specify this option, any node type can be a parent.
PREFIX=pref	a string used with new node types to set the prefix used for auto-generated node names. For example, the prefix for a fltGroup is g: new groups are named g1, g2, and so forth. **Note:** You do not need to enclose the prefix in quotes.
NOSEARCH	disables attribute search for this field in Creator.

Option	Definition
XCODE=code	identifies which type of node (as returned by *`mgCode()`*) to attach the field or record to.

struct Defines logical groupings of fields for creating *records*. Provides the name of the **`struct`** to **`recordDef`**.

Options

Option	Definition

PAD=n	lets you add padding to the internal representation of your data structures. Use this option inside a struct definition. Provide a dummy name followed by the number of padding bytes. For example:
	`struct myList {`
	`myField1,`
	`myField2,`
	`xxxpad, PAD=3,`
	`myField3,`
	`xxxpad, PAD=6`
	`}`

PTR	identifies a field that represents a pointer in memory.

parent Defines a list of node types that can be the parent of a specified node type.

child Defines a list of node types that can be the child of a specified node type.

describe Defines text strings for **ddbuild** to print when creating documentation for your data dictionary.

Data Types

When using **`dataDef`** and **`recordDef`**, you must declare all of your extension parameters as one of the following data types:

type	dataDef	recordDef	Represents
`MTYPE_CHAR`	x		char int
`MTYPE_UCHAR`	x		unsigned char int
`MTYPE_SHORT`	x		short
`MTYPE_USHORT`	x		unsigned short
`MTYPE_INT`	x		int
`MTYPE_UINT`	x		unsigned int
`MTYPE_FLOAT`	x		float
`MTYPE_DOUBLE`	x		double
`MTYPE_FLAG`	x		flag
`MTYPE_TEXT`	x		text string
`MTYPE_REC`	x	x	Generic record

Copyright

PRESAGIS