



GAME EXCHANGE PROGRAMMER'S GUIDE



CORPORATE OFFICE:

12555 W. Jefferson Blvd., #285
Los Angeles, CA 90066
(310) 577-0500

Game Exchange Programmer's Guide

December 1999

Copyright © 1999 by Nichimen Graphics Incorporated. All rights reserved. No part of this work covered by the copyright herein may be reproduced or copied in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping or information and retrieval systems—without the written permission of the publisher.

All sample source code included in this manual is licensed by Nichimen Graphics.

Game Exchange, N-World, Mirai, and Nendo are trademarks of Nichimen Graphics Incorporated. All other trademarks are used for identification purposes only, and are acknowledged as the property of their respective owners.

Printed in the United States of America.

TABLE OF CONTENTS

TABLE OF CONTENTS	3
PREFACE	7
SCOPE OF THIS MANUAL	8
SYSTEM REQUIREMENTS	8
SUPPORTING DOCUMENTS	8
REQUIRED KNOWLEDGE	9
STYLE CONVENTIONS	9
NOTES	9
CODE SAMPLES	9
SPECIFIC C++ ELEMENTS	9
CONTACTING SUPPORT	10

Chapter 1

GAME EXCHANGE: A BRIEF OVERVIEW	11
CHAPTER OVERVIEW	12
ABOUT GAME EXCHANGE	12
CREATING CONTENT	14
EXPORTING CONTENT	14
USING THE GAME EXCHANGE LIBRARY TO CONVERT DATA TO A PLATFORM SPECIFIC FORMAT	14
WHAT'S NEW IN GAME EXCHANGE 2.1?	14

Chapter 2

GAME EXCHANGE FILES	17
CHAPTER OVERVIEW	18

GAME EXCHANGE FILE TYPES	18
TEMPLATE FILES	18
CONTENT FILES	19
FILE RELATIONSHIPS	20
A NOTE ON EMBEDDED FILES	23
GAME EXCHANGE FILE FORMAT	23
TYPE NAMES	24
VARIABLE NAMES AND VALUES	24
A NOTE ON RESERVED KEYWORDS	25
DECLARATIONS	25
INSTANTIATIONS	28
THE RELATIONSHIP BETWEEN DECLARATIONS AND INSTANTIATIONS	28
GAME EXCHANGE FILE STRUCTURE	29
HEADER SECTION	30
INCLUDE SECTION	31
TEMPLATE DECLARATION SECTION	32
TEMPLATE INSTANTIATION SECTION	32
EXAMPLES OF FILE STRUCTURE	32

Chapter 3

PARSING35
CHAPTER OVERVIEW	36
WHAT IS PARSING?	36
TYPE NODES	36
COMPOSITE NODES	37
AN EXAMPLE OF PARSING	37
TYPE CHECKING	40
PARSING TABLES	40

Chapter 4

USING THE GAME EXCHANGE LIBRARY CLASSES.41
CHAPTER OVERVIEW	42
GAME EXCHANGE LIBRARY CLASSES	42
THE GEABSTRACTNODE CLASS	43
THE GECOMPOSITENODE CLASS	44
THE GETYPENODE CLASS	45
THE GEGAPI CLASS	46

THE GELIST CLASS	48
THE GEVECTOR CLASS	52
THE GESTRING CLASS	55
THE GEABSTRACTITERATOR CLASS	56
THE GELISTITERATOR CLASS.	57
THE GEHASH CLASS	58

Chapter 5

ANATOMY OF A WRITER63
CHAPTER OVERVIEW	64
CODING WRITERS	64
WORKING WITH SIMPLE WRITER'S FILES.	64
BUILDING SIMPLE WRITER	64
ON THE NT PLATFORM	65
ON THE IRIX PLATFORM	65
RUNNING SIMPLE WRITER	65
ON THE NT PLATFORM	65
ANALYZING THE OUTPUT DATA	66
DECONSTRUCTING THE WRITER	70
THE SIMPLE.H FILE	71
THE SIMPLEMAIN.CPP FILE	71
THE SIMPLE.CPP FILE	71
MODIFYING SIMPLE WRITER	74

Chapter 6

CONCLUSION79
REVIEW	80
GETTING STARTED ON YOUR OWN WRITER	80

Appendix A

SIMPLE WRITER CODE83
SIMPLE WRITER CODE	84
SIMPLE.H	85
SIMPLE.CPP	87
SIMPLEMAIN.CPP	95

Appendix B

ADVANCED TOPICS	.97
USING CUSTOM PROPERTIES WITH GAME EXCHANGE	98
USING VERSIONING INFORMATION	100
USING GRAMMAR VERSION INFORMATION	100
USING TEMPLATE VERSION INFORMATION	101
USING LIBRARY VERSION INFORMATION	101

Appendix C

GLOSSARY OF TERMS	103
INDEX	107



PREFACE

This preface discusses the scope and conventions of this manual.

SCOPE OF THIS MANUAL

The term *Game Exchange* refers both to an intermediate input/output file format and to a C++ library. Your source application (Mirai, Nendo, Alias|Wavefront's Maya, etc.) handles the procedures for writing out data in the Game Exchange file format. These procedures are explained in the documentation accompanying the source application, so this manual will not repeat them. Instead, this manual will focus on the use of the Game Exchange C++ library. This library is designed to assist you in creating writers for converting data in the Game Exchange file format into a format readable by your target application. The tools of the library are flexible enough to permit you to convert content into a format that is readable by virtually any target application. Toward that goal, this manual will describe the format of the various Game Exchange file types. Thereafter, it will instruct you on how to create your own writers using the C++ classes included in the Game Exchange library. Each class is briefly defined, then its uses illustrated with practical examples, including sample source code.

As you read this manual, keep in mind that import requirements of game engines and other target applications vary tremendously. While the Game Exchange library gives you the flexibility to write out data in a format that suits your target, this manual cannot describe the exact process that you will undertake in creating your writer. However, the examples provided should give you a thorough enough understanding of how the classes function.

SYSTEM REQUIREMENTS

The Game Exchange library is designed to run on the following platforms:

- Microsoft NT™ (version 4.0, Service Pack 3 and greater)
- Irix™ (version 6.3 and greater)

The Game Exchange library supports the following C++ compilers:

- NT Visual C++ (version 5.0 and greater)
- Windows 98 Visual C++ 6.0
- Irix MipsPro 7.2.1 for MIPS-2 (o32)
- Irix MipsPro 7.2.1 for MIPS-2 (o32)

SUPPORTING DOCUMENTS

If you have not already installed Game Exchange, refer to the *readme* file included on the Game Exchange installation CD. For a concise overview of the Game Exchange file format and library, you can read the *Game Ex-*

change White Paper. This manual is also meant to be read in conjunction with the *Game Exchange Reference Guide*, which provides a comprehensive description of the library's C++ classes and methods.

REQUIRED KNOWLEDGE

This manual assumes a working knowledge of C++. Concepts which are of particular relevance to Game Exchange include templates, classes, and hierarchies. You should also be familiar with tree and list data structures.

STYLE CONVENTIONS

This manual contains a number of style conventions designed to make it easier to use. Different paragraph and character styles designate various user actions and interface elements. The following sections describe the style conventions used.

NOTES

Notes of special interest are indented:

NOTE: *Notes provide information that is important enough to demand special attention. The information may be relevant to the current discussion but awkward to include without disrupting the flow of information.*

CODE SAMPLES

Lines or a blocks of sample code are indented and displayed in a courier font:

```
geTypeNode<geVector<float> >*           typeNode = NULL;
typeNode = (geTypeNode<geVector<float> >*) normal;
vec =                                     typeNode->getValue();
```

SPECIFIC C++ ELEMENTS

When discussed in a paragraph of regular text, C++ elements such as variables, commands, and methods appear in an italicized avant garde font. For example:

- The *geVector* constructor creates a vector called *vec* of type *float* with a length of three.

CONTACTING SUPPORT

If you have any difficulty using the Game Exchange library, send e-mail to the following address:

gex-support@nichimen.com



CHAPTER 1

GAME EXCHANGE: A BRIEF OVERVIEW

This chapter describes what's new in Game Exchange 2.1 and provides a brief overview of the process for using the Game Exchange format and library.

CHAPTER OVERVIEW

In this brief chapter, you'll learn about:

- The overall process for using the Game Exchange format and library.
- The enhancements made to Game Exchange for the 2.1 release.

ABOUT GAME EXCHANGE

Game Exchange provides a mechanism for converting content created in N-World, Mirai, Nendo, or one of several third-party applications into a file format readable by your target application. For example, you can use Game Exchange to convert a Mirai *scene* (including all objects, materials, skeletons, and animation data) into a format readable by a proprietary game engine. Game Exchange can also be used to convert 3rd party data into formats readable by Mirai.

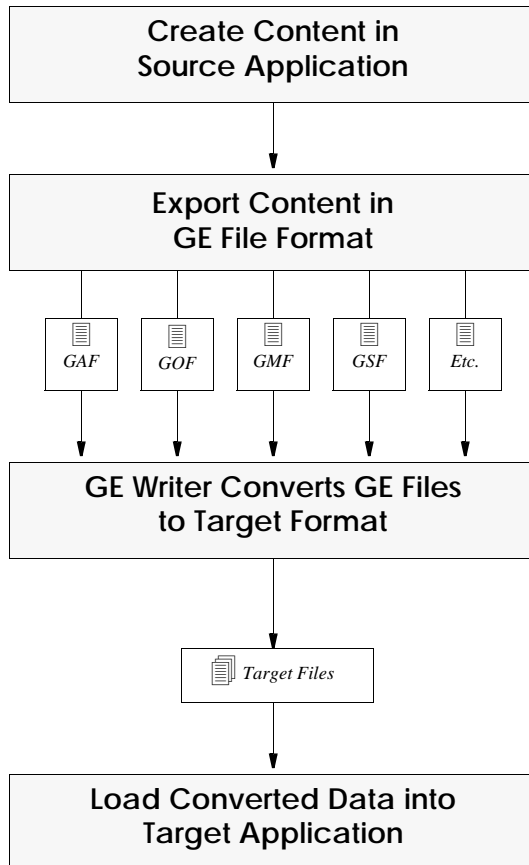
NOTE: *In Mirai, a group of related entities (objects, skeletons, materials, and lights) can be loaded as a single unit called a scene. Your source application may or may not support such a concept; however, this manual uses the word "scene" to describe any block of data that is being processed through Game Exchange.*

The process for using Game Exchange consists of the following phases:

- 1 Create content in the source application.
- 2 Export content in the Game Exchange format, the output of which is a set of ASCII files. (These files include GAF, GEF, GOF, GBF, GMF, GLF, GCF, and GSF files, all of which are explained in Chapter 2).
- 3 Use a Game Exchange *writer* to convert the Game Exchange output files to a set of files which are readable by your target platform.
- 4 Load the converted data into your target application.

The figure below illustrates this process:

Figure 1.1 The data conversion process



If you are using Game Exchange to convert content between 3D packages, this diagram applies in both directions. The following sections describe each of the phases of using Game Exchange in more detail.

CREATING CONTENT

The following Nichimen applications export 3D content directly into Game Exchange format:

- N-World (version 3.2 and greater). If you want to export data in Game Exchange 2.1 format from N-World 3.2, you'll need to obtain an additional plug-in. Contact Nichimen Graphics Support at support@nichimen.com.
- Mirai (version 1.0.4.3 and greater)
- Nendo (version 1.0 and greater)

NOTE: *Other third-party 3D applications also export content to Game Exchange formats. Follow the Game Exchange link on the Nichimen website (<http://www.nichimen.com>) for complete information on these applications.*

EXPORTING CONTENT

To prepare the source content for eventual conversion to your target application, you use the source application's output commands to export the data in the Game Exchange file format.

If you are uncertain how to export content in the Game Exchange format from your source application consult its documentation. If you are exporting from Mirai or Nendo, you can search the accompanying online help for "export." If you are exporting from N-World, look for "export" in the indices to the *N-Geometry Reference Guide* and the *N-Dynamics Reference Guide* printed documentation.

USING THE GAME EXCHANGE LIBRARY TO CONVERT DATA TO A PLATFORM SPECIFIC FORMAT

Once you have exported the source content in Game Exchange format, you're ready to use a Game Exchange writer to convert the output file set to a format that is readable by your target platform. The Game Exchange library assists you in creating writers by parsing Game Exchange output files into C++ data structures and providing functions to traverse these data structures and write files for your target application. The process of creating writers will be the focus of this manual.

WHAT'S NEW IN GAME EXCHANGE 2.1?

If you've worked with Game Exchange 2.0 in the past, you may be wondering what enhancements have been made for this latest release. This section briefly overviews these enhancements.

● VERSIONING

The header sections within all Game Exchange 2.1 files now include detailed versioning information. For example:

```
filetype gx;
GrammarVersion 2.1.0.0;
TemplateVersion 2.1.0.0;
LibraryVersion 2.1.0.0;
HostName "Greenghost";
UserName "Poky Russell";
TimeStamp "1/2/99 2:34 pm";
OSName "Windows NT";
OSVersion "4.0 service pack 3";
ApplicationName "Mirai";
ApplicationVersion "1.0.4.3";
```

See *Header Section* on page 30 for more information.

● SPEED IMPROVEMENTS

The classes that control traversals of data trees have been improved. These make it faster to search for desired data after parsing a set of Game Exchange files.

● DECREASED MEMORY USAGE

The parsing process has been modified to eliminate redundant data.

● GRAMMAR FIXES

Several grammar problems from the Game Exchange Library API have been corrected.

● NEW TEMPLATES

Substantial portions of the template files have been rewritten in order to improve consistency in the Game Exchange format. See the *Game Exchange 2.1 Reference Guide* for a full accounting of these changes.

● GREATER PATHNAME FLEXIBILITY FOR TEMPLATES AND PARSING TABLES

In previous versions, parsing tables and templates had to reside in the same directory as the content files in order for parsing to take place. We added methods to the library that allow you to specify pathnames to these files, so that they may now reside anywhere on your network.

See *Parsing Tables* on page 40 for more information.

● NEW DOCUMENTATION

We extensively revised the Game Exchange documentation. Based on a customer feedback, we created this manual to aid programmers in developing of their own writers. We also completely updated the *Game Exchange 2.1 Reference Guide* to reflect all the latest changes to Game Exchange 2.1.

● INCLUSION OF A SAMPLE WRITER

The installation CD now includes a fully-operational sample writer, which developers can use as a foundation for designing their own writers. See “*Anatomy of a Writer;*” on page 63 for a full explanation of how to use this writer.



CHAPTER 2

GAME EXCHANGE FILES

*This chapter gives detailed information on the format of
Game Exchange files.*

CHAPTER OVERVIEW

In this chapter, you'll learn:

- About the various export file types supported by Game Exchange.
- How Game Exchange template files define the format for exported data.
- How Game Exchange content files describe the actual exported scene's contents.
- How Game Exchange files either cross-reference each other or are embedded within each other to allow for the parsing of an entire scene's data.
- About the internal structure of Game Exchange files, including the various code elements which compose them.

GAME EXCHANGE FILE TYPES

When you export content in the Game Exchange format, your source application converts the content into a set of intermediate ASCII files. These files can be broadly divided into two categories, *template files* and *content files*. Both content and template files share the same general format and structure, but, as you'll read below, they differ in function.

TEMPLATE FILES

Template files don't contain any actual exported data, but rather act as schema for what data may be exported from a source application and how that data should be organized and formatted. Template files are application specific, so you will see variances in template files depending on the application from which you receive your data. The template files that you see as output from Mirai or Nendo represent the export standards that Nichimen has defined for these products.

Template files serve two main functions:

- They give the Game Exchange format the flexibility to accommodate new forms of data as source applications evolve new features.
- They provide a means for verifying the integrity of exported data.

As Mirai and Nendo evolve new features, new options for the export of data emerge. Template files give the Game Exchange format the fluidity to accommodate data from these new features. With each new release of Mirai or Nendo, Nichimen can modify the Mirai or Nendo template files in order to give export support to all the latest features.

Most of the changes that Nichimen implements in the template files are simply extensions to previous templates. As with any API, you should review your code when new versions of Game Exchange are released in the event that any extensions have been added that you might want to take advantage of.

Template files also offer Nichimen a means of verify the integrity of exported data. As you'll read in *An Example of Parsing* on page 37, part of the process of parsing Game Exchange file includes comparing template files and content files (the files containing the actual exported data) against each other to check for data errors. This process, called *type checking*, can reveal errors in the content data.

Template files best serve you the programmer as road maps to the data that you want your writer to extract and convert. After reading this chapter, you'll have a much better understanding the structure of template files, and how to make use of them in creating your writer.

CONTENT FILES

Content files contain the actual exported data in the format specified by the template files. These files describe the various entities of an exported scene (for example, the 3D coordinates of an object's vertices or the HSV values for a material's diffuse color).

Content files can include the following file types:

CONTENT FILE TYPE	FILE EXTENSION	FUNCTION
Game Animation Format	.gaf	Describes animation data for light objects, cameras, and 3D objects. The locations of these entities are described frame-by-frame. If an object is composed of subobjects, the location of the parent object is described first and is followed by the location of the subobjects. Depending on the options you choose at export, the locations of subobjects are described with respect to the global origin or relative to the location of the parent object.
Game Object Format	.gof	Describes the transformation data for a single object. This object may be a terminal object or a hierarchical object with several subobjects.
Game Body Format	.gbf	Describes the topological data for a single object.
Game Environment File	.gef	Describes the environmental properties for an exported scene (for example, fog color, fog position, etc.).
Game Material Format	.gmf	Describes the properties of materials and locally assigned object attributes.
Game Skeleton Format	.gsf	Describes a single skeleton including its hierarchy, name, bones, joints, descriptions of all base states, descriptions of each pose, and listings of all hard and soft skin parts associated with each bone. Any skins associated with the skeleton are exported as <i>objects</i> and are described in associated GOF files.
Scene File List	.grp	Lists files which must be parsed together.
Game Light File	.glf	Describes properties for all lights.
Game Camera File	.gcf	Describes camera properties.

FILE RELATIONSHIPS

When content is exported in the Game Exchange format, the source application adds the appropriate references between all content and template files to ensure that all the data is parsed together (you’ll read more about parsing in Chapter 3). The figure below summarizes the default references for Mirai. Keep in mind that if you’re exporting from another application, your files may not have this exact structure. (See the section “A Note on Embedded Files,” on page 23).

Figure 2.1. Default GE file references for Mirai

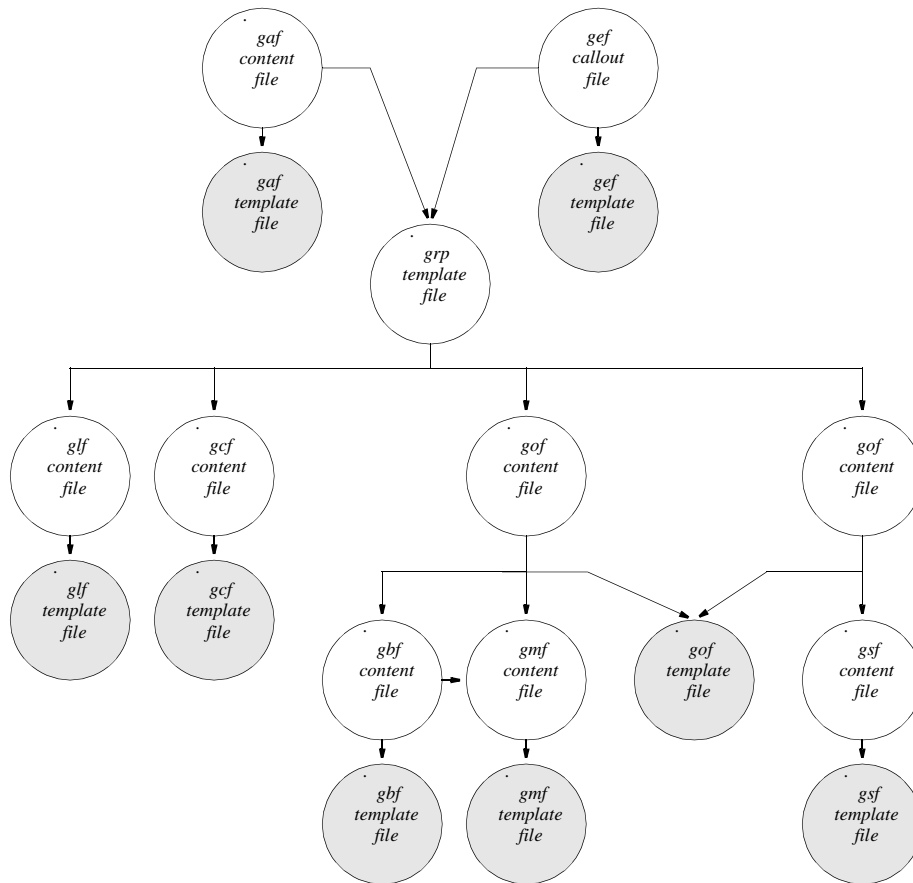


Figure 2.1 shows a GAF file (containing the exported scene’s animation data) and a GEF file (containing the exported scene’s environmental data), both of which reference the GRP file. The GRP file, in turn, references each

of the exported scene's data files (the GLF, GCF, and GOF content files). Note that there can be multiple content files, depending on the content of the exported scene.

If a GOF file describes an object with an applied material, that GOF file references a GMF file. The GMF file includes data for *all* materials in the exported scene. A GOF file can also reference a GBF file, which contains the topological (body) data for an object, and a GSF file, which describes a skeleton. The GBF file can in turn reference the GMF file, if applicable. Multiple GOF and GBF files may share the same GMF file.

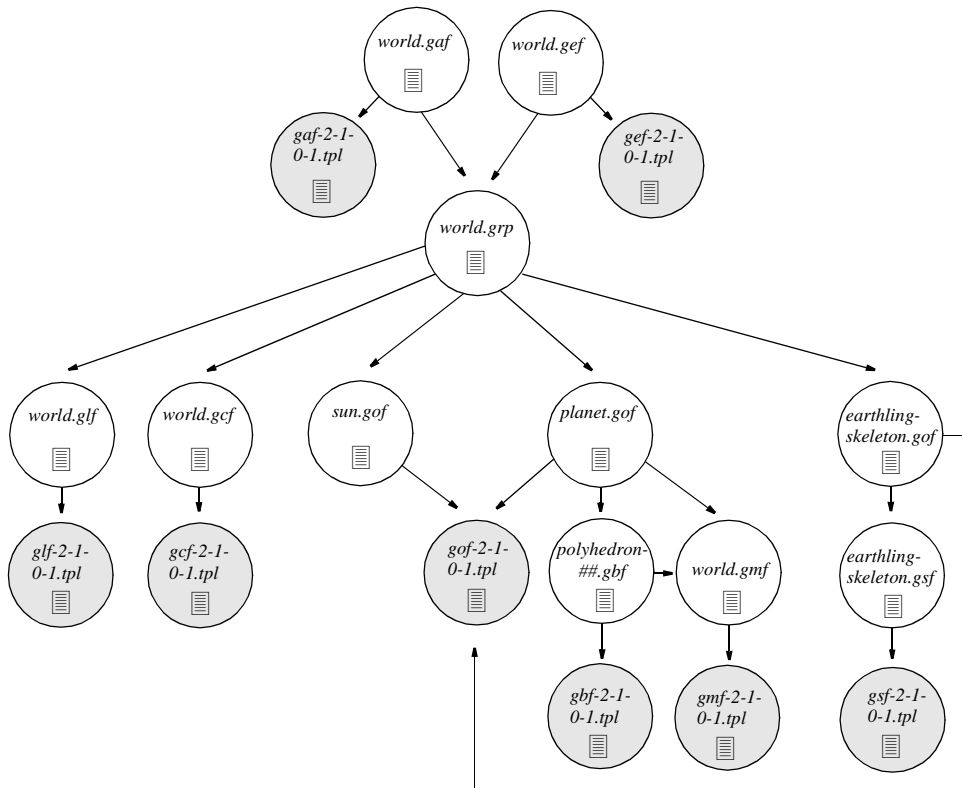
NOTE: *Each GAF, GEF, GLF, GCF, GOF, GBF, GSF and GMF file will have associated TPL (template) files.*

Let's look at an example. Imagine that you are exporting a scene from Mirai called "world" with the following elements:

- A sphere called "planet" with an applied material called "terra."
- A skeleton called "earthling-skeleton."
- An animation script that animates the skeleton, sphere, and camera.
- A point light called "sun."

The output of the Game Exchange files from Mirai would resemble the diagram below:

Figure 2.2. Sample GE file output from Mirai.



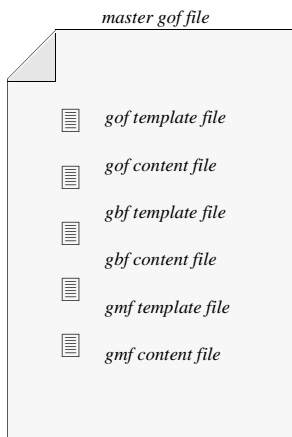
Notice that three GOF files are written out—one for the “planet” 3D object, one for the “sun” light object, and one for the “earthling-skeleton” skeleton. All of these GOF files reference the same TPL file. The “planet” GOF also references a GMF file, which describes the object’s applied materials, and a GBF file, which describes its transformation (body) data. The “sun” light references nothing other than the TPL file, since lights exported from Mirai never have a body or a material. The “earthling skeleton” GOF references the GSF file, which describes the skeleton’s hierarchy, base states, poses, etc.

NOTE: *Data for any light exported from Mirai will reside in two files: a GOF file, which describes the light’s transformation matrix, and a GLF file, which describes the properties of the light (for example, its color, brightness, etc.)*

A NOTE ON EMBEDDED FILES

While Mirai writes out each of the Game Exchange files separately (with appropriate references between them), *your* source application may not. For example, Nendo writes out only *one* GOF file per export. All other files, including content and template files, are embedded within this GOF file:

Figure 2.3. Nendo's embedded file structure.



This structure precludes the need for references between Game Exchange files.

GAME EXCHANGE FILE FORMAT

Both content and template files adhere to the same general format, which this section will explore in detail. The discussion will explain the file format from the bottom up. You'll first learn about the basic building blocks of a Game Exchange file—*type names*, *variable names*, and *values*. Then you'll learn how these blocks can be combined into larger structures, namely *declarations* and *instantiations*.

Three basic elements make up the data inside Game Exchange files:

- Type names, which specify a type or kind of data (for example, an integer or string).
- Variable names, which assign a name to the data.
- Values, which represent the data itself in the form of numbers, strings, booleans, etc.

The following sections discuss the usage and syntax of these elements in greater detail.

TYPE NAMES

Type names are either built-in or custom (that is, defined by you). There are ten built-in type names, which are listed in the table below, along with their default values and ranges, if applicable.

NOTE: Type names are case sensitive.

BUILT-IN TYPE NAMES	DEFAULT VALUE	RANGE
float	0.0;	10^{-38} - 10^{+38}
integer	0;	-2^{31} - $2^{31}+1$
string	(empty)	N/A
boolean	false;	True or False
vec2i	0 0;	N/A
vec3i	0 0 0;	N/A
vec4i	0 0 0 0;	N/A
vec2f	0.0 0.0;	N/A
vec3f	0.0 0.0 0.0;	N/A
vec4f	0.0 0.0 0.0 0.0;	N/A

NOTE: The float values of NAN (not a number) and +/- infinity are not represented in Game Exchange files. Attempting to parse these types of float values will result in unexpected token error messages.

You can also create your own custom type names, though this is typically not necessary if you are exporting from N-World, Mirai, or Nendo. See Appendix B for more information on adding custom types.

VARIABLE NAMES AND VALUES

Variable names are typically written *variable name*. However, if you want to create a variable that will contain an array, the variable name is written *variable name[]*. For example, the variable definition below creates an integer-type variable called NOISE-LAYERS and assigns it a default value of one.

```
integer NOISE-LAYERS 1;
```

The sample below also defines an integer-type variable.

```
integer REFRACT-MAP-SAMPLING-LIMIT [] <32; 256; >;
```

But this time, the variable is set up as a one-dimensional array, as indicated by the two closed-brackets (“[]”) at the end of the variable name. This array is assigned the integer values 32 and 256. Notice that angle brackets (“<”>”) enclose the values of the array, and that a semi-colon follows each value of the array. As with all variable definitions, a semi-colon ends the line.

A NOTE ON RESERVED KEYWORDS

Before learning how to combine type name, variable names, and values into larger structures, you should be aware that certain keywords hold special meaning in Game Exchange and should not be used in the code you create. The table below summarizes these *reserved keywords*. You'll recognize many of these keywords from the previous discussion on type names.:

KEYWORD	FUNCTION
extends	Allows a template declaration to inherit attributes from another template declaration. (See Appendix B for more information on using this keyword.)
properties	Used in instantiations to specify additional properties for that instance. (See Appendix B for more information on using this keyword.)
template	Defines a template for a new composite type.
include	Precedes a list of files that will be parsed after the current file.
float	Built-in type name that specifies a floating point number for the value.
integer	Built-in type name that specifies an integer for the value.
string	Built-in type name that specifies the value is a character string.
boolean	Built-in type name that specifies a value of true or false be returned.
vec2i	Built-in type name that uses 2D vector integers and gives directional and scalar (quantity) information.
vec3i	Built-in type name that uses 3D vector integers and gives directional and scalar (quantity) information.
vec4i	Built-in type name that uses 4D vector integers and gives directional and scalar (quantity) information.
vec2f	Built-in type name that uses 2D vector floating point numbers and gives directional and scalar (quantity) information.
vec3f	Built-in type name that uses 3D vector floating point numbers and gives directional and scalar (quantity) information.
vec4f	Built-in type name that uses 4D vector floating point numbers and gives directional and scalar (quantity) information.
__root@@	Built-in type name for implicit declaration and instantiation of root nodes.

DECLARATIONS

Together, types, variables, and optional values form *declarations*. A declaration creates a new *data type* that is composed of one or more variable definitions. This data type doesn't contain any actual export values, only optional default values. This new type provides a mold for exported data to fill.

Declarations compose the core of all template files. A sample declaration is shown below:

```
template face-part (  
string name "default" ;  
)
```

The diagram uses L-shaped lines to point from text labels to specific parts of the code:

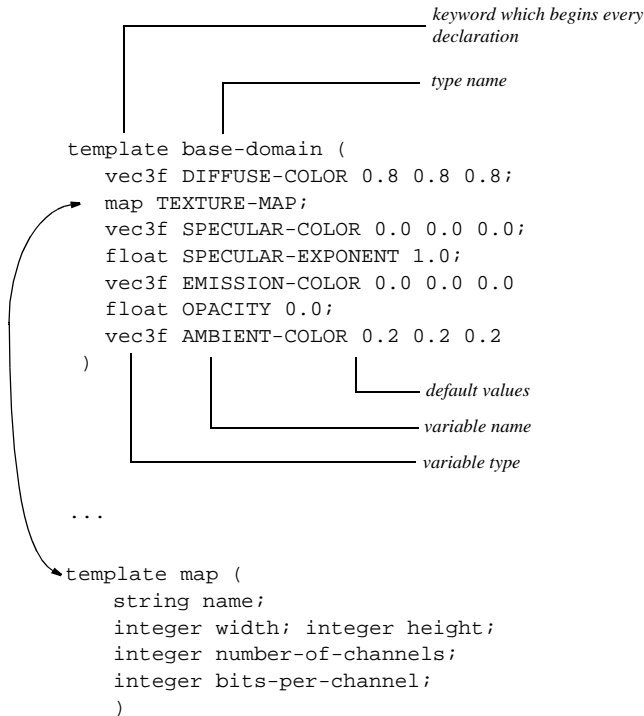
- keyword which begins every declaration* points to `template`.
- name of new type* points to `face-part`.
- default value* points to the string `"default"`.
- variable name* points to `name`.
- variable type* points to `string`.

This sample declaration creates a new type called *face-part* that is composed of a string-type variable with an assigned value of *default*.

Note these syntax rules for declarations:

- A declaration must begin with the keyword *template*.
- All variable definitions must be enclosed within parentheses (“()”). You can include as many variable definitions as you like, but *you cannot use duplicate variable names*.
- Each variable definition must end with a semi-colon.

Below is another sample of a declaration. This one is slightly more complex than the previous example:



This sample shows that it is possible to reference one declaration from another. In the example, the line in the *base-domain* declaration that begins with *map* references another declaration called *map*.

Creating such references between declarations has several advantages. The primary advantage is that it enables you to organize potentially large volumes of data hierarchically; that is, you can divide data into as many levels as necessary (for example, declaration a references declaration b, which in turn, references declaration c). You can readily imagine the difficulties, if you could not create hierarchical levels. For example, if you were to export a high resolution object that had thousands of faces and many displacements to a completely flat hierarchy, finding the coordinates to a particular vertex would be very difficult and time-consuming.

Another advantage of cross-referencing declarations is that you can *share* a single declaration among multiple declarations. For example, imagine that you were creating a template in which several declarations included a common set of variable definitions. You could simply repeat this set of variable definitions in all of the declarations. However, a more elegant solution would be to create a declaration that contained the common set of variable definitions, then reference this declaration from every other declaration that required that set of variable definitions.

INSTANTIATIONS

Content files contain lists of *instantiations*. Earlier, we referred to declarations as molds. You can think of instantiations as the actual content for these molds. The *format* for an instantiation is derived from a declaration, but its *values* are derived from the content of an exported scene or entity.

A sample instantiation is shown below:

```

face-part (
  name "outer-wall";
)

```

The format of the this sample was derived from the sample declaration given on page page 25. It includes the same type name (*face-part*) and a string variable called *name*. Note, however, that its value is no longer “default.” Instead, the value *outer-wall*, which came from the content of the exported scene, has overwritten the default value.

Note that instantiations closely follow the syntax rules of declarations:

- An instantiation begins with the type name.
- All the variables and their values are enclosed within parentheses.
- Each variable line ends with a semi-colon.

THE RELATIONSHIP BETWEEN DECLARATIONS AND INSTANTIATIONS

The samples below will give you a better understanding of the relationship between declarations and instantiations. First, look at the following declaration, from a GLF template file:

```

template point-light (
  string name;
  float brightness 0.5;
  vec3f rgb 1.0 1.0 1.0;
  vec3f position 0.0 10.0 0.0;
  boolean use-attenuation TRUE;
  float radius 10.0;
)

```

As you can see, this declaration defines the format for exported point lights. *Point-light* is the type being created, which is composed of several variable definitions for light properties such as brightness, color, position etc. Default values are provided for most of the variables.

Now look at a corresponding instantiation, which is excerpted from the GLF content file:

```
point-light (  
    name "sun";  
    brightness 2.0;  
    rgb 1.0 0.6285714 0.6285714;  
    position 0.0 25.0 0.0;  
    use-attenuation TRUE;  
    radius 5.0;  
)
```

The instantiation reflects the format of the declaration, but contains actual values from the exported scene, which have overwritten the default values provided by the template. For example, the value for brightness is now 2.0 rather than 0.5.

Multiple instantiations can reference the same declaration. For example, if you were to export two lights from a scene, “sun” and “moon,” the resulting GLF content file would contain two instantiations for lights, both of which would reference the same declaration in the GLF template file.

```
light lights (  
    point-lights[] <  
        (name "sun";  
            brightness 2.0;  
            rgb 1.0 0.6285714 0.6285714;  
            position 0.0 25.0 0.0;  
            use-attenuation TRUE;  
            radius 5.0;)  
        (name "moon";  
            brightness 1.0;  
            rgb 1.0 0.3285714 0.3285714;  
            position 0.0 25.0 0.0;  
            use-attenuation TRUE;  
            radius 3.0;)>  
    )
```

Don’t worry for now, if the concepts of declaration and instantiation seem a little confusing. Their usage will become clearer as you continue on in this chapter. For now, take a closer look at the internal structure of Game Exchange files.

GAME EXCHANGE FILE STRUCTURE

By now, you’re better acquainted with Game Exchange file formats. This chapter will close by explaining how the main elements, such as declarations and instantiations, are arranged within Game Exchange files. Remember that both content and template files both have the same basic structure.

Game Exchange files can have up to four sections:

- Header (mandatory)
- Include (optional)

- Template Declaration (template files only)
- Template Instantiation (content files only)

The sections below clarify the structure and function of each section.

HEADER SECTION

Every Game Exchange file includes a *header*. A header section identifies a file as being a Game Exchange output file and includes versioning information that helps the writer determine how to parse the file.

Early versions of Nichimen applications write out header sections that look like this:

```
filetype gx;  
version 2.03;
```

As you can see, this header section makes it clear that the file is a Game Exchange (“gx”) file and that it was generated using Game Exchange 2.03.

Later versions Nichimen applications writer out more detailed versioning information to each header section. For example:

```
filetype gx;  
GrammarVersion 2.1.0.0;  
TemplateVersion 2.1.0.0;  
LibraryVersion 2.1.0.0;  
HostName "Greenghost";  
UserName "Poky Russell";  
TimeStamp "1/2/99 2:34 pm";  
OSName "Windows NT";  
OSVersion "4.0 service pack 3";  
ApplicationName "Mirai";  
ApplicationVersion "1.0.4.3";
```

NOTE: *The header elements that appear in quotes above may have a slightly different format in your output files depending on your source application.*

In this later format, version numbers have four parts:

- The first number increments major changes to a system (for example, a complete redesign).
- The second number increments minor changes to a system; these changes don’t affect the interface or core functionality of the system.
- The third number increments the addition of features to the system.
- The fourth number increments bug fixes.

Version numbers are reported for Game Exchange parsing tables, template files, and library files. Making use of this versioning information is fully explained in Appendix B.

Note that the header also supplies the following information:

- The name of the computer from which the content was exported (*Hostname*).
- The user who exported the content (*UserName*).
- The time at which the content was exported (*TimeStamp*).
- The operating system in use (*OSName*).
- The version of the operating system in use (*OSVersion*).
- The name of the software package from which the content was exported (*PackageName*).
- The version of the software package from which the content was exported (*PackageVersion*).

INCLUDE SECTION

As you read in *File Relationships* on page 20, certain Game Exchange files must be parsed together. For example, if you export an object that has an applied material, the GOF file that describes the object must be able to reference the GMF file that describes the material. Additionally, that same GOF file must be able to reference its TPL file in order for Game Exchange to verify the integrity of the data. The *include* section makes such references possible by listing the files that must be parsed after the current file. If no other files need be parsed with this file, then the include section is omitted.

You can see an example of an include section below, which was excerpted from a GOF file.

```
include "gof-2-03.tpl";  
include "world.gmf";
```

The above section references a TPL file and a GMF content file, both of which are stored in the same directory as the GOF file itself. These references ensure that the GOF file, its TPL file, and the GMF file will all be parsed together.

Typically, it's not necessary to modify the include section of a particular file as your source application creates the appropriate references by default. It is possible, though, to modify an include section to group together any desired set of files. For example, suppose that you wanted to parse the files from several exported scenes in one pass. You could easily modify the include section of one of the scene's GRP file so that it contained references to the GRP files for the other scenes. For example:

```
include "scene2.grp";  
include "scene3.grp";  
include "scene4.grp";
```

TEMPLATE DECLARATION SECTION

With the exception of the GRP file, all Game Exchange content files either have an embedded *template declaration* section or reference TPL files that include template declaration sections. Template declaration sections contain one or more declaration statements in the format described in *Declarations* on page 25. As mentioned earlier, each declaration acts as a template for how a particular collection of data elements can be exported.

TEMPLATE INSTANTIATION SECTION

Template instantiation sections contain lists of instantiations in the format described in *Instantiations* on page 28. Each of these instantiations contains exported data which conforms to the format defined in a corresponding declaration.

EXAMPLES OF FILE STRUCTURE

The figures below show you how the header, include, declaration, and instantiation sections appear in a couple of actual Game Exchange files. Figure 2.4 shows a GOF template file. Note that because the template file is merely the template for data export, it includes a declaration section but not an instantiation section. Figure 2.5 shows the

GOF content file. As it embodies the actual scene data in the format specified by the template file, it includes an instantiation section but not a declaration section.

Figure 2.4.. Sample GOF template file

```

header section  filetype gx;
                GrammarVersion 2.1.beta.5;
                TemplateVersion 2.1.beta.6;

declaration section
                template gobject (
                    string name;
                    vec4f matrix[] <1.0 0.0 0.0 0.0; 0.0 1.0 0.0 0.0; 0.0 0.0 1.0
0.0; 0.0 0.0 0.0 1.0;>
                    vec4f init-matrix[] <1.0 0.0 0.0 0.0; 0.0 1.0 0.0 0.0; 0.0 0.0
1.0 0.0; 0.0 0.0 0.0 1.0;>
                    vec3f bounding-box[];
                    string body;
                    gobject children[];
                    attached-bone attached-bone; # Attached Bone
                )

```

Figure 2.5. Sample GOF content file

```

header section  filetype gx;
                GrammarVersion 2.1.beta.5;
                TemplateVersion 2.1.beta.6;
                HostName "SNUFFY";
                UserName "jason";
                TimeStamp "Wed 27-Oct-99, 4:24 pm";
                OSName "Windows NT 4.00.1381";
                ApplicationName "Mirai";
                ApplicationVersion "1.0.26.0 5351";

include section
                include "gof-2-1-beta-6.tpl";
                include "unnamed.gmf";

instantiation section
                include "Polyhedron 31.gbf";
                gobject Cube (
                    name "Cube";
                    matrix[] <
                        1.0 0.0 0.0 0.0;
                        0.0 1.0 0.0 0.0;
                        0.0 0.0 1.0 0.0;
                        0.0 0.0 0.0 1.0; >
                    bounding-box[] <-10.0 -10.0 -10.0;10.0 10.0 10.0;>
                    body "Polyhedron-31";
                )

```




CHAPTER 3

PARSING

This chapter explains the process by which Game Exchange output files are parsed into data structures.

CHAPTER OVERVIEW

In this chapter, you'll learn:

- About parsing and data trees produced by parsing.
- About the general structure of data trees, including the kinds of nodes that compose them.

WHAT IS PARSING?

If you read the last chapter, you are now acquainted with the format of Game Exchange output files. Parsing is the first step toward converting these files into a format that is readable by your target application. In the context of Game Exchange, parsing means to convert one or more Game Exchange export files into a *tree*. The *geGapi* class, which you'll read about in *The geGapi Class* on page 46, handles the parsing of the files for you. Once the class has parsed the files into a data tree, your writer can traverse it and write out the appropriate data.

The tree generated by the *geGapi* class is composed of two types of nodes, *type nodes* and *composite nodes*.

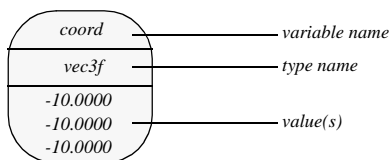
- Type nodes make up the leaves of the tree; they hold actual data values.
- Composite nodes make up the branches of the tree; they can connect with type nodes and other composite nodes.

The following sections explain these nodes in a little more detail.

TYPE NODES

Types nodes, the leaves of a tree, are actually instances of the *geTypeNode* class, which is fully explained in *The geTypeNode Class* on page 45. As the figure below shows, a type node consists of a variable name (see *Variable Names and Values* on page 24), a type name (see *Game Exchange File Structure* on page 29), and a single value (for example, an integer or bool):

Figure 3.1. A type node.

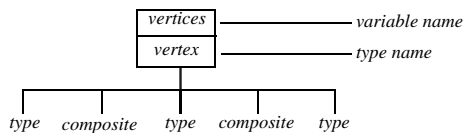


You'll design your writer to traverse down the tree to a desired type node and extract its values.

COMPOSITE NODES

Composite nodes are actually instances of the *geCompositeNode* class, which is fully explained in *The geCompositeNode Class* on page 44. The elements of custom types and arrays become composite nodes when a GE file is parsed. As the figure below shows, a composite node can consist of a variable name, type name, and links to multiple type nodes and/or other composite nodes. In later chapters, we'll refer to all of the composite and type nodes that are contained within a single composite node as a *composite node list*.

Figure 3.2. A composite node.



Composite nodes make up all of the nodes on a tree, except for the terminal nodes. You'll design your writer to iterate over the various composite node that compose a tree in search of the data you want to write out.

A special list called *root* makes up the topmost node on a tree. The root node is not parsed from a GE output file; rather, the parser inserts it into a tree in order to connect the top-level nodes of each GE output file. This allows for a single tree for all content data and a single tree for all template data.

AN EXAMPLE OF PARSING

To actually parse a collection of GE files, you invoke functions which are part of the *geGapi* class (see *The geString Class* on page 55). Parsing will produce two separate data trees, one based on the information within the content files and another based on the information within the template files. As you'll read in *Type Checking* on page 40, the template-based data tree is used to verify the integrity of the content-based data tree. You'll use the library's various functions to traverse the content data tree and write out the necessary data into a file or group of files that is readable by your target application.

To give you a better idea of how the parsing process works, a sample tree is depicted on the following pages. Figure 3.3 shows the content file which was parsed, a GOF file. Figure 3.4 shows a graphical representation of the content tree itself. A template data tree is not shown, but it is similar in structure, the main difference being that its type nodes hold default values rather than actual values derived from the exported data.

Figure 3.3. A sample GOF file .

```

filetype gx;
version 2.03;

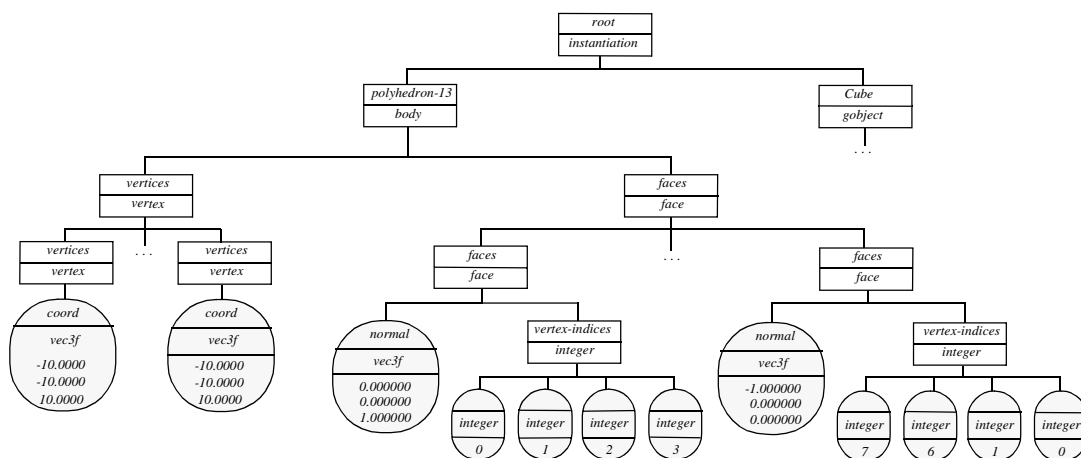
include "gof-2-03.tpl";
include "treehouse.gmf";

body Polyhedron-13 (
  vertices[] <
    (coord -10.0000 -10.0000 10.0000 ; )
    (coord -10.0000 10.0000 10.0000 ; )
    (coord 10.0000 10.0000 10.0000 ; )
    (coord 10.0000 -10.0000 10.0000 ; )
    (coord 10.0000 -10.0000 -10.0000 ; )
    (coord 10.0000 10.0000 -10.0000 ; )
    (coord -10.0000 10.0000 -10.0000 ; )
    (coord -10.0000 -10.0000 -10.0000 ; )
  >
  faces[] <
    (normal 0.000000 0.000000 1.000000 ;
     vertex-indices[] <0;1;2;3;>)
    (normal 0.000000 0.000000 -1.000000 ;
     vertex-indices[] <4;5;6;7;>)
    (normal 0.000000 1.000000 0.000000 ;
     vertex-indices[] <1;6;5;2;>)
    (normal 0.000000 -1.000000 0.000000 ;
     vertex-indices[] <7;0;3;4;>)
    (normal 1.000000 0.000000 0.000000 ;
     vertex-indices[] <3;2;5;4;>)
    (normal -1.000000 0.000000 0.000000 ;
     vertex-indices[] <7;6;1;0;>)
  >
)

gobject Cube (
  name "Cube";
  matrix[] <
    1.0 0.0 0.0 0.0;
    0.0 1.0 0.0 0.0;
    0.0 0.0 1.0 0.0;
    0.0 0.0 0.0 1.0;
  >
  bounding-box[] <
    -10.0 -10.0 -10.0;
    10.0 10.0 10.0;
  >
  body "Polyhedron-13";
)

```

Figure 3.4. A sample data tree.



While an actual data tree for the content of an exported scene will have the same hierarchy as the figure above, keep in mind that it can be considerably more complex. An actual content data tree will include the data from all GE content files. For example, the first levels of nodes under the root will include not just a body and gobject node, but also nodes related to materials, skeletons, lights, etc.

An actual content data tree will also incorporate default values from the template files when no actual values exist in the exported content. (For example, if the exported content did not include values for the specular or emission colors of a material, the content data tree would use the template data tree's default values for those properties.)

TYPE CHECKING

You'll recall that the GRP file for an exported scene contains references to all the other content files. This ensures that all the content files are parsed together, resulting in a single tree for all the content data. You'll also recall that parsing produces a second tree that is based on the template files. Part of the parsing process entails comparing the template and content trees against each other to check for data errors; this process is called *type checking*. Type checking reveals errors in content data such as duplicate variable names or unexpected values, such as a string where an integer should appear. Any problems discovered during type checking are reported in error messages. (See the *Game Exchange Reference Guide* for a full listing of possible error messages.)

PARSING TABLES

In order to properly parse export files, Game Exchange requires access to two tables. The first required table is contained within a file called `<version info>.dfa` (for example, `ge2_1_0_0.dfa`), and the second is contained within a file called `<version info>.llr` (for example, `ge2_1_0_0.llr`).

Both of these files are included with the Game Exchange library. For parsing to occur, you must tell your writer where to find the tables. There are two means of doing so:

- 1 Place these files in the same directory as the content files to be parsed.
- or-
- 2 Using the `setTablePath` method of the `geGapi` class, explicitly point the writer to the files by providing a path-name to the director in which they are contained (see *The geGapi Class* on page 46).



CHAPTER 4

USING THE GAME EXCHANGE LIBRARY CLASSES

This chapter briefly overviews the C++ classes used in the Game Exchange library. Code samples illustrate the primary uses for each class.

CHAPTER OVERVIEW

In this chapter, you'll learn:

- Which classes are contained in the Game Exchange library.
- What each class is used for when creating a writer.

GAME EXCHANGE LIBRARY CLASSES

The Game Exchange Library contains several classes, each with a robust set of methods. These methods provide you with most of the functionality you need to create your own writer.

The following preprocessor directive in the program file includes all the class definitions in the Game Exchange library:

```
#include "gegapi.h"
```

The table below lists and briefly describes each of the classes that are included in the Game Exchange library:

CLASS NAME	MAIN USES
geAbstractNode	Referencing geComposite and geTypenode classes (this is the base class to these classes). Note that this class is <i>not</i> intended to be used directly—use geCompos- iteNode or geTypeNode.
geCompositeNode	Forming composite nodes on data trees.
geTypeNode	Forming type nodes on data trees.
geGapi	Parsing Game Exchange output files.
geList	Representing data as lists.
geVector	Representing data as vectors.
geString	Representing data as strings.
geHash	Adding and retrieving items from hash tables.
geAbstractIterator	Traversing nodes on data trees. Note that this class is <i>not</i> intended to be used directly—use geListIterator or geHashIterator.
geListIterator	Traversing lists.
geHashIterator	Traversing hash tables.

The basic process for writing data involves first iterating to the right node on a data tree, then making a call to the appropriate function or method to extract the data. You use the `CompositeNode` class and the `TypeNode` class to do this. As you begin to code your writer, keep in mind that instances of these two classes will probably constitute its bulk:

- The `CompositeNode` Class

An instance of the `CompositeNode` class gets you to the right node by creating and implementing iterators.

- The `TypeNode` Class

An instance of the `TypeNode` class checks a node by name, casts down to retrieve its value and writes out the value to file.

This chapter provides an overview of all classes, and presents you with code samples that call methods from each of the classes. These code samples address tasks which are common to most writers. Some of these samples are excerpted from the sample writer that appears at the end of this manual (see Appendix A). Carefully analyzing them will help you create your own writer.

This chapter does not discuss *all* of the methods associated with the library's classes. It merely discusses those that are most commonly used. For a full description of each class and all of its associated methods, refer to the *Game Exchange Reference Guide*.

THE `GEABSTRACTNODE` CLASS

You'll recall from the discussion in Chapter 3 that Game Exchange files are parsed into data trees composed of type and composite nodes. Remember that type nodes are instances of the `geTypeNode` class and composite nodes are instances of the `geCompositeNode` class. The `geTypeNode` and the `geCompositeNode` both publicly inherit from the `geAbstractNode` class. This common base allows both `geTypeNodes` and `geCompositeNodes` to coexist in the same list of `geAbstractNode` pointers. This also provides several common methods for every node in a tree. For example, the `getName` function is defined at the `geAbstractNode` level, so you can call this function on any node.

USES

The `geAbstractNode` is a virtual class. The primary use of this class is to assist in traversing data trees with `geAbstractNode` pointers. The `AbstractNode` contains two methods, `getName` and `getType`, that are vital for traversing data trees. You can call these methods on any node for which you have an `AbstractNode` pointer. Depending on the value that the method returns, you can determine what action the writer should perform next. For example, if the `getType` method returns "skeleton," then you can send this node to the appropriate section of code in the writer for dealing with skeletons. To use another example, if `getType` returns "vec3f" and `getName` returns "diffuse color," then you might want to have the writer write out the color values.

Another usage for `geAbstractNode` is worth mentioning: `geAbstractNode` pointers are very useful for holding locations in the data tree. For example, if your writer is iterating over a body's nodes and it encounters a material which it will need to make use of later, you can have the writer store the material node's location as a `geCompositeNode` pointer. The sample below, excerpted from the sample writer in Appendix A, shows the `geCompositeNode` class being used in this manner. This example stores pointers with one of three variables depending on the name returned by the `getName` method.

```

nodeName = (*first)->getName();
if (nodeName == "vertex-indices") {
    vertInd = (*first);
} else if (nodeName == "vertex-normal-indices") {
    normInd = (*first);
} else if (nodeName == "uv-indices") {
    uvsInd = (*first);
}

```

getName method; returns the name of the current node

NOTE: *This code sample is part of a larger function, which uses an iterator to sequentially access a whole series of nodes (see “The `geListIterator` Class,” on page 57 for more information).*

The following events occur in this sample.

- 1 The `nodeName` variable is set equal to the name of the current node, then `getName` method is called to return the name this node.
- 2 The remaining lines check the name of the current node (which is stored as `nodeName` variable's value) against a series of names. If a match is found, the pointer to the current node is stored with a particular variable (for example, if a node named “vertex-indices” is found, a pointer to it is stored in the `vertInd` variable).

Once the locations to the desired nodes have been stored as pointers, other parts of the code can perform actions on the nodes to which the pointers point.

For a sample usage of the `getType` method, refer to the last code sample in “The `geList` Class,” on page 48.

THE `GECOMPOSITENODE` CLASS

Instances of the `geCompositeNode` class form the composite nodes on data trees. This class inherits its base functionality from the `geAbstractNode` class.

USES

The `geCompositeNode` class includes methods for adding and removing nodes to data trees which are useful when constructing a reader. In constructing a writer, you'll typically use the `geCompositeNode` class to group elements.

In order to extract data from a tree, you will frequently need to navigate across a composite node. To do so, you'll use methods defined in the `geListIterator` class (which is referenced in the sample below as the `childIterator`).

The sample below, excerpted from the sample writer in Appendix A, shows an example of navigating through a composite node called *normals* in order to write out a set of normals to a file:

```
void simpleWriter::writeNormals(geCompositeNode* normals)
{
    geCompositeNode::childIterator first(*normals);
    geCompositeNode::childIterator last (*normals);
    last.end();

    vbOutFile << "Normal " << normals->getChildNodeCount() << endl;

    for (; first<last; first.increment()) writeNormal(*first);
}
```

Some of the methods employed in the this sample are defined in the `geListIterator` class, but we'll discuss them in this section because we are using them to traverse an instance of the `geCompositeNode` class. The following events occur in this sample:

- 1 The *geCompositeNode* constructor creates two list iterators called *first* and *last*. (For now, just think of a *list iterator* as a means for traversing across the nodes in a list; you'll read more about iterators in *The geListIterator Class* on page 57.) By default, these list iterators point to the first node in the list.
- 2 The *end* method points the *last* iterator to the end of the list of nodes.
- 3 The *getChildNodeCount* method determines number of nodes in the list. This number is then written to a file.
- 4 Finally the *increment* method advances through the list of nodes until it reaches the final node. A function called *writeNormal* (defined elsewhere in the sample writer) writes out the data encountered at each node to a separate file.

THE GETypeNode CLASS

Instances of the `geTypeNode` class form the leaf nodes on data trees. The `geTypeNodes` hold the actual data from your content files. For example, if you would use a `geTypeNode` to store the name of a particular body. This class inherits its base functionality from the `geAbstract` node class. It is a templated class, meaning that you can use it to work with values of any type (for example, floats, strings, integers, etc.).

USES

The `geTreeNode` class contains methods for extracting these values from a tree. The sample below, excerpted from the sample writer in Appendix A, shows a process for extracting values from a type node. You'll notice that casting must take place before values can be extracted.

```
void simpleWriter::writeNormal (geAbstractNode* normal)
{
    geVector<float> vec(3);

    geTreeNode<geVector<float> >* typeNode = NULL;
    typeNode = (geTreeNode<geVector<float> >*)normal;
    vec = typeNode->getValue();
    vbOutFile << vec.getAt(0) << " , "
               << vec.getAt(1) << " , "
               << vec.getAt(2) << endl;
}
```

getValue method; returns the values in the form of a geVector (see "The geVector Class," on page 52)

The following events occur in this sample:

- 1 The first line passes a *geAbstractNode* pointer called *normal* into a function called *writeNormal*.
- 2 Next, the *geVector* constructor creates a vector called *vec* of type *float* with a length of three.

The next lines perform the necessary casting so that data can be pulled from the type node at the abstract node level.

- 3 First, a *geTreeNode* pointer of the type *geVector* is created and set to null.
- 4 The *normal* pointer is then cast to the *geTreeNode* pointer, which is templated for values of the type *geVector<float>*.
- 5 With casting to the proper pointer complete, the sample uses the *getValue* method to store values from the type node into the vector *vec*.
- 6 Finally, the sample prints the results. These are extracted from the vector using the *getAt* method (see *The geVector Class* on page 52).

THE GEGAPI CLASS

You primarily use the *geGapi* class to parse a set of Game Exchange export files into a data tree, then return the root node from this tree. As all other Game Exchange classes perform operations on a parsed tree, your writer will usually use the methods from the *geGapi* class before those of the other Game Exchange classes. Typically, you use the methods of the class to set the appropriate parsing parameters. Next, you invoke the method that actually

parses the tree. Finally, you invoke the method that returns the tree's root node. After this point, you can begin to use the other Game Exchange classes to traverse the tree and access the appropriate data.

The sample below, excerpted from the sample writer shown in Appendix B, shows you how to use the methods of the `geGapi` class to open and parse a set of Game Exchange files. This code passes in the pathname (as a string) to a file for parsing. To understand the what's happening, you should realize that the sample writer defines a class called *simpleWriter*. The *gapi* method below is an instance of the `geGapi` class that has been declared elsewhere as a private member of the *simpleWriter* class:

```
bool simpleWriter::openGX2File(geString geFile)
{
    gapi.ignoreWarnings(true);
    gapi.setTablesPath(geString("g:\\tmp"));
    return (gapi.parse(geFile));
}
```

The following events occur in this sample:

- 1 The first line defines a method called *openGX2File* that belongs to the *simpleWriter* class. This method will return a value of *true* if the files parse successfully and *false* if they don't.
- 2 Next, *ignoreWarnings*, a method of the `geGapi` class, is set to *true*, meaning that no warning messages will be printed to the screen.
- 3 Then the *setTablesPath* method, another method of the `geGapi` class, defines the pathname to the tables required for parsing. As discussed in Chapter 3, you'll need to include this method in your writer unless you place the DFA and LLR files in the same directory as the as the content files which you are parsing.
- 4 Finally, *gapi.parse*, another method of the `geGapi` class, parses the files, returning a value of *true* or *false* depending on the outcome.

Parsing with the `geGapi` class only returns true on success and false on failure. Your console will display more information on parsing as it occurs, including information on what files have been parsed and what errors, if any, were encountered. For example, parsing a batch of files might produce the following output in your console.

```
Game Exchange Library Version 2.1.0.0
Using tables located in: g:\tmp/
Parsing cube.grp

Includes
Parsing material1-cube.gof

Includes
Parsing gof-2-1-beta-6.tpl
Parsing cube.gmf

Includes
Parsing gmf-2-1-beta-6.tpl
Parsing Polyhedron 105.gbf

Includes
Parsing gbf-2-1-beta-6.tpl
The file cube.gmf has already been parsed
Finished
```

THE GEList CLASS

The `geList` class is a templated class that includes various methods for building lists and extracting values from them. Because it is a templated class, you can use `geList` to work with lists that are comprised of values of any type.

USES

Internally, the Game Exchange library uses the `geList` class to build the nodes of data trees during parsing. However, there are numerous tasks for which you may want to employ the `geList` class. For example, you might use `geList` to list the coordinates of all of an object's vertices, or you might use it to list all the names of all of the materials in a scene. A few code samples will illustrate some of the most commonly used methods of `geList`.

MAKING LISTS AND ADDING/REMOVING ELEMENTS

The code sample below highlights the methods for adding elements to and removing elements from the beginning of a list. It also shows how to declare and print the contents of a list.


```

void example()
{
    geList<int>    integerList;
    integerList.addHead(1);
    integerList.printList();
    integerList.addHead(2);
    integerList.addHead(3);
    integerList.printList();
    integerList.removeHead();
    integerList.printList();
}

```

geList declaration; defines a new list to be composed of integers
addHead method; adds the integer "1" to the beginning of the list
printList method; prints the current contents of the list to screen
removeHead method; removes the first integer in the list

output

```

1
3 2 1
2 1

```

The following events occur in this sample:

- 1 The *geList* constructor creates a new list, one composed of integer values.
- 2 The *addHead* method adds elements to the beginning of the list. (If you wanted to add an element to the end of the list, you could employ the *addTail* method.)
- 3 The *printList* method prints the current contents of the list to the screen (this method is most useful for debugging purposes).
- 4 After two more integers are added to the beginning of the list, the *removeHead* method removes the first element from the list. (If you wanted to remove the last element in the list, you could use the *removeTail* method.)

Now suppose that you need to insert an element not just at the beginning or end of a list but after or before a specific element in the list. The sample below illustrates how to do so by using an index number.

```
void example()
{
    geList<geString>  stringList;

    stringList.addHead(geString("wood"));
    stringList.addHead(geString("leaf"));
    stringList.addHead(geString("bark"));
    stringList.printList();

    stringList.insertAfter( 1, geString("stem"));—— insertAfter method; inserts "stem" string
    stringList.printList();                        after the second element in the list

    stringList.insertBefore(1, geString("flower"));—— insertBefore method; inserts the "flower"
    stringList.printList();                        string before the second element in the list

    int index = stringList.getIndex(geString("leaf"));
    stringList.removeAt(index);—— removeAt method; deletes the "leaf"
    stringList.printList();                        string from the list
}
```

output

```
bark leaf wood
bark leaf stem wood
bark flower leaf stem wood
bark flower stem wood
```

The following events occur in this sample:

- 1 The *geList* constructor creates a new list, which the *addHead* method populates with a series of strings.
- 2 After printing the contents of the list, the sample makes use of the *insertAfter* method to insert the *stem* string after a specific element in the list. When using this method, you indicate the insertion point for the new element by specifying the index number of the element that should precede the new element. In this example the index is set to one. This means that the string “stem” should be inserted after the element with the index number of one.

NOTE: Remember that C++ indexes begin at zero. Therefore, if you want to insert an element after the second element in a list you must specify “1” as the index number, as the first element of the list is actually indexed as “0.”

- 3 The *printList* method prints the current contents of the list to the screen.
- 4 The *insertBefore* method shown in the sample works like *insertAfter*, except that it inserts the string *flower* after the index number “1.”

- 5 The *printList* method prints the current contents of the list to the screen.
- 6 Finally, the sample uses the *removeAt* method to remove the *leaf* string from the list. When using this method, you simply indicate the index number that corresponds to the element that you want to remove.

EXTRACTING ELEMENTS FROM LISTS

You've learned how to print the entire contents of a list using the *printList* command. The next code sample for the *geList* class will show you how to extract specific elements from a list.

```
void example()
{
    geList<float> floatList;

    floatList.addHead(1.2f);
    floatList.addHead(2.3f);
    floatList.addHead(3.4f);
    floatList.addHead(4.5f);
    floatList.printList();

    cout << floatList.getHead() << endl;
    cout << floatList.getTail() << endl;
    cout << floatList.getAt(1) << endl;
}
```

getHead method; extracts the first float value from the list

getTail method; extracts the last float value from the list

getAt method; extracts the value with index number "1" from the list

output

```
4.5 3.4 2.3 1.2
4.5
1.2
3.4
```

The following events occur in this sample:

- 1 The *geList* constructor creates a new list, which the *addHead* method populates with a series of floats.
- 2 The *printList* method prints the list's entire contents.
- 3 The *getHead* and *getTail* methods extract the first and last values from the list.
- 4 Finally, the *getAt* method extracts the value in the list which is indexed as "1." Again, because C++ indexes the list starting at zero, this results in the output of the second value in the list.

Before leaving the `geList` class, take a look at one final code sample, this one taken from the sample writer in Appendix A. This example iterates over a composite node list. When it encounters a node of type *gobject*, it stores a pointer to that node in a list. After it constructs the list, it loops over the list, writing out its contents.

```

geString nodeType;
for (; first < last; first.increment()) {
    nodeType = (*first)->getType();

    if (nodeType == "gobject") {
        gObjectList.addTail(*first);
    }
    .
    .
    for (int i = 0; i < gObjectList.getCount(); i++) {
        writeGObject(gObjectList.getAt(i));
    }
}

```

test for whether node is of the gobject type
addTail method; appends pointers to "gobject" nodes to end of the list
getAt method; sequentially extracts the values from the list.

The following events occur in this sample:

- 1 The first lines set up a means of looping over the current composite node list. You'll learn more about this in *The geAbstractIterator Class* on page 56).
- 2 The next lines determine whether a node is of the *gobject* type.
- 3 The *addTail* method, which you saw used in the previous examples, appends the pointers to those nodes that are of the type *gobject* to the end of the list.
- 4 Once the list is compiled, the *getAt* method extracts each pointer from the list and calls the *writeGObject* method, which is part of a class defined earlier in the writer. This method will write out the each element of the list.

THE GEVECTOR CLASS

The `geVector` class is a templated class that includes methods for building *vectors* and extracting values from them. For Game Exchange purpose, vectors will typically store only a small set of data elements, usually a maximum of four, all of which are of a common type. Because `geVector` is a templated class, you can use it to work with vectors that are comprised of values of any type.

USES

In graphics applications, a vector is ideal for uses such as storing and writing the Cartesian coordinates of a vertex, the normal information for a face, or the transformational matrix data for a object. The following code samples show some common uses for `geVector`.

MAKING A VECTOR

The code sample below highlights methods for adding elements to a vector.

```
void example()
{
    geVector<float> bbMax;
    geVector<float> bbMin;

    bbMax.setLength(3);
    bbMin.setLength(3);

    bbMax.setAt(0, 10.0);
    bbMax.setAt(1, 10.0);
    bbMax.setAt(2, 10.0);

    bbMin.setAt(0, -10.0);
    bbMin.setAt(1, -10.0);
    bbMin.setAt(2, -10.0);

    bbMax.print(cout);
    cout << endl;
    bbMin.print(cout);
    cout << endl;
}

output
10 10 10
-10 -10 -10
```

The following events occur in this sample:

- 1 The *geVector* constructor creates two new vectors, both composed of float values.
- 2 Next, the *setLength* constructor is used to set the length of each vector to three, meaning that each can hold three values.
- 3 Next, the *setAt* method is used to fill each vector with values; each is placed in an indexed slot.
- 4 Finally, the *print* method is used to print out the contents of each vector to the screen.

EXTRACTING DATA FROM A VECTOR

As the following example demonstrates, the *geVector* class also includes methods for extracting values from vectors. The example shows the construction of a vector, then the retrieval and printing of the vector's values. In designing your writer, you'll only have to concern yourself with the retrieval of vector values, as the vectors themselves will be produced during parsing.

```

void runExample()
{
    geTypeNode<geVector<float> > aVector; geTypeNode constructor; creates new type node
    geVector<float> vec(3);

    vec.setAt(0, 1.0);
    vec.setAt(1, 2.0);
    vec.setAt(2, 3.0);

    aVector.setValue(vec);

    example(&aVector);
}
.
.
.
void example(geCompositeNode* node)
{
    geVector<float> vec;

    geTypeNode<geVector<float> >* typeNode = NULL;
    typeNode = (geTypeNode<geVector<float> >*)node;
    vec = typeNode->getValue();

    for (unsigned int i = 0; i < vec.getLength(); i++)
        cout << vec.getAt(i) << " "; getAt method; extracts values from the vector according to index number
    cout << endl;
}

```

output

```
1 2 3
```

The following events occur in the *runExample* function:

- 1 First, the *geTypeNode* constructor creates a type node called *aVector* of the type *geVector*.
- 2 Next, the *geVector* constructor creates a three-element vector called *vec* of the type *float*.
- 3 The next lines use the *setAt* method (described above) to fill *vec* with the values 1.0, 2.0, and 3.0.
- 4 Then *setValue*, a method of the *geTypeNode* class, sets the value of the node to equal that of the vector *vec*.

With the vector created and values assigned, the remaining portion of the sample extracts the values by calling the *example* function.

- 5 The *geVector* constructor creates a vector called *vec* of the type *float*.

The next lines perform the necessary casting to extract the data from the type node and place it in a vector. You saw a similar example in *The geTypeNode Class* on page 45.

- 6 The *geTypeNode* constructor creates a *geTypeNode* pointer of the type *geVector* and sets it to null.
- 7 The *node* pointer is then cast to the *geTypeNode* pointer, which is templated for values of the type *geVector<float>*.
- 8 With casting to the proper pointer complete, the next line uses the *getAt* method to load values from the type node into the *vec* vector.
- 9 The final lines set up a loop from 0 to length of vector. As the loop proceeds, the *getAt* method extracts each value from the vector, which is then output to the screen.

THE GESTRING CLASS

The *geString* class provides you with methods for working with information that is stored as strings, such as object names, pathnames, and so forth.

USES

With these methods you can format strings (such as converting a string to UPPERCASE) and perform decision-making operations (such as determining whether a variable is equal to a particular string value). In doing so, you can compare instances of the *geString* class against C strings or against other instances of the *geString* class. The sample below, taken from the sample writer in Appendix A, shows a similar operation, which is very common in the design of a writer. This portion of the writer iterates over a composite node list. As it hits on a node that matches a given name, it stores a pointer to that node in a variable. (You may recall seeing a very similar example in *The geAbstractNode Class* on page 43.)

```
geString nodeName;
for (; first<last; first.increment()) {
    nodeName = (*first)->getName();

    if (nodeName == "vertices") {
        vertices = (*first);
    } else if (nodeName == "normals") {
        normals = (*first);
    } else if (nodeName == "uv-array") {
        uvs = (*first);
    } else if (nodeName == "faces") {
        faces = (*first);
    } else if (nodeName == "material") {
        bodymat = (*first);
    }
}
```

The following events occur in this sample:

- 1 The first line declares an instance of the *geString* class called *nodeName*.
- 2 The next line iterates over a composite node (see *The geListIterator Class* on page 57 for more information on iterators).
- 3 For each node encountered, the next line sets *nodeName* equal to the name of the node.
- 4 The remaining lines make comparisons between *nodeName* and a series of C strings. If a match is found, then a pointer to the matching node is stored in a corresponding variable. For example, if a node called *normal* is found, then a pointer to it is stored in a variable called *normals*.

THE GEABSTRACTITERATOR CLASS

You use iterators to sequentially access specific data in a structure (for example, a list). The *geAbstractIterator* contains all of the base methods for iteration. In addition, the Game Exchange library contains three additional classes related to iteration, all of which inherit their base functionality from the *geAbstractIterator* class:

- The *geListIterator* class contains the methods for iterating over instances of the *geList* class (see *The geListIterator Class* on page 57).
- The *geHashIterator* class contains the methods for iterating over Hash tables (see *The geHash Class* on page 58 for more information).
- The *geNullIterator* contains the methods for iterating over the empty set of a type node.

Of these three classes, the *geListIterator* class is most frequently employed when creating a writer, so it will be the focus of the following discussion on iterators. Once you understand how this class works, you'll find it very easy to employ the methods of the *geHashIterator* class.

The *geNullIterator* only exists for cases when your writer is on a type node, but doesn't yet recognize it as a type node and attempts to iterate over it. Since type nodes cannot have child nodes, it's impossible to iterate over a type node, and the class simply exits your writer from the iteration loop.

You should also be aware that there are other means of traversing trees in addition to iterators. You can, for example, use the *geList* class to traverse both regular lists (with the *getAt* method) and composite node lists (with the *getChildNode* method). Recognize, however, that it will take longer to traverse a structure with a method of the *geList* class than it would with an iterator. This is due to the fact that lists are implemented as linked lists, so in order to get to a particular position, the writer must start from the beginning of the structure each time (*geList* methods have no memory). Iterators, on the other hand, keep track of their position, allowing for sequential access to list members or child nodes.

THE GEListIterator CLASS

The `geListIterator` class inherits from the `geAbstractIterator` class and contains methods designed for traversing lists constructed using the `geList` class. For most traversals, whether of a list or a hash table, you create two iterators, the first of which marks the beginning of the traversal sequence and the second of which marks the end.

TRAVERSING LISTS CREATED FROM GEList

The sample below shows the construction and traversal of an instance of `geList`.

```
void example()
{
    geList<int> myList;

    myList.addHead(1);
    myList.addHead(2);
    myList.addHead(3);
    myList.addHead(4);
    myList.addHead(5);

    geListIterator<int> first(&myList);
    geListIterator<int> last(&myList);

    first.begin();
    last.end();

    for (; first < last; first.increment()) {
        cout << (*first) << " ";
    }
}
```

geListIterator constructor; creates an iterator called "first" of the type "int."

output

5 4 3 2 1

The following events occur in this sample:

- 1 First, the `geList` constructor creates a list called `myList` that is designed to hold integers.
- 2 Next, a series of `addHead` methods, part of the `geList` class, build the list by appending a sequence of integers to its head.
- 3 Then the `geListIterator` constructor creates two iterators called `first` and `last`.
- 4 The `begin` and `end` methods, part of the `geAbstractIterator` class, set the `first` iterator to the first element in the list and the `last` iterator to one past the last element.

- 5 The last line de-references the *first* iterator, sequentially returning each item in the list.

TRAVERSING COMPOSITE NODES

The main use of iterators is to traverse composite node lists. Game Exchange provides some help for doing so in the form of child and parent iterators. Child iterators are instances of the `geAbstractNodeChildIterator`, and parent iterators are instances of the `geAbstractNodeParentIterator`. These are both templated classes that inherit their functionality from the `geListIterator`. You use them by passing them a pointer to a composite node, and they respond by creating an iterator for traversing the children of that node.

The following declaration from the Game Exchange library makes the use of child iterators over composite nodes possible:

```
typedef geAbstractNodeChildIterator<geAbstractNode*> childIterator;
```

Since this declaration has already been made for you, you need only add a line like the following to your writer in order to make an iterator for a composite node list:

```
geCompositeNode::childIterator first(*vertices);
```

In this case, the result would be an iterator called *first* that iterates over all of the children of the composite node *vertices*. You could then call all of the base methods of the `geAbstractIterator` class on this iterator (for example, *begin*, *end*, *increment*, etc.)

You can also make child and parent iterators for type nodes, but as discussed on page 56 this will simply result in a `geNullIterator` because type nodes have no child nodes over which to iterate.

THE GEHASH CLASS

The `geHash` Class provides you with the functionality for building and extracting from *hash tables*. Hash tables allow you to store a large number of elements, then rapidly retrieve any number of those elements. An entry in a hash table is composed of two elements:

- A key, which is a value of any type (for example, an integer or string) that uniquely identifies the entry.
- An item, which is the data element which you wish to store (for example, an integer, a string, or a pointer).

When building a hash table, you supply both a key and an item for each entry. When accessing an entry, you typically supply the key and the item is returned.

The great advantage to using hash tables is the speed with which they can return items. Because each item has a unique identifier, data retrieval from a hash table is typically much faster than data retrieval from an array or list.

Another advantage of using hash tables is that they spare you the chore of writing look-up code. When using a list, for example, you have to write code the code responsible for traversing a list in search of a certain item. With a hash table this is not necessary; you simply extract an item by looking up its key.

The following example shows you how to build and retrieve items from a hash table.

```
void example()
{
    geHash<geString, geString>    matImages;
    geString                      hashValue;

    matImages.setItem(geString("wood"), geString("/bark1.bmp"));
    matImages.setItem(geString("leaf"), geString("/maple1.bmp"));

    matImages.lookup(geString("leaf"), hashValue);
    cout << "The bitmap used for leaf is "
          << hashValue << endl;
    matImages.getAt(0, hashValue);
    cout << "The first bitmap is "
          << hashValue << endl;
}
```

geHash constructor; creates a hash table called "matImages" with keys and items of the type "geString."

setItem method; makes an entry in the hash table composed of a "wood" key and a "/bark1.bmp" item.

lookup method; the variable "hashValue" is set to the value from the hash table with the "leaf" key.

geHash method; the variable "hashValue" is set to the value from the hash table with the index number "0".

output

```
The bitmap used for leaf is /maple1.bmp
The first bitmap is /bark1.bmp
```

The following events occur in this sample:

- 1 The first line creates a hash table called *matImages*; the keys and items for this table will both be strings.
- 2 The second line creates an instance of the *geString* class called *hashValue*.
- 3 The next two lines use the *setItem* method of the *geHashIterator* class to create entries for the hash table. The first entry has the string *wood* as its key and the string */bark1.bmp* as its item. The second entry has the string *leaf* as its key and the string */maple1.bmp* as its item.

Next, the sample retrieves the entries.

- 4 The *lookup* method retrieves the item with the *leaf* key and sets the variable *hashValue* equal to this item.
- 5 The next lines output this item.
- 6 The *getAt* method shows you an alternate method for retrieving an entry. With this method, an index number is supplied (in this case, "0") and the corresponding value is returned (in this case, the first entry in the hash table.)
- 7 The last lines output the item.

Before leaving the `geHash` class, take a look at the following code sample from the sample writer in Appendix A.

```

geHash< geString, geCompositeNode* > bodyHash;
geHash< geString, geCompositeNode* > materialHash;
.
.
.
    if (nodeType == "gobject") {
        gObjectList.addTail(*first);
    } else if (nodeType == "body") {
        bodyHash.setItem((*first)->getName(), (*first));
    } else if (nodeType == "material") {
        materialHash.setItem((*first)->getName(), (*first));
    }
.
.
.
geCompositeNode*    body = NULL;
if (!bodyHash.lookup(bodyName, body)) exit(-2);
writeBody(body);

```

The following events occur in this sample:

- 1 The first two lines create two hash tables, the types for which are strings and the items for which are pointers.
- 2 The next lines store elements encountered in a composite node differently based on type.
 - If the node is of the type *gobject*, a pointer to the node is appended to a list.
 - If the node is of the type *body*, a pointer to it is stored as a hash table item, the key to which is the string name of the node.
 - If the node is of the type *material*, a pointer to it is stored as a hash table item, the key to which is the string name of the node.

NOTE: *In the sample writer, only pointers to materials and polyhedral bodies are stored in hash tables. You'll find that it is most convenient to hash any items which you want to quickly be able to reference later. For example, if you know that you will eventually need to retrieve a material called "red," use a hash table. On the other hand, if you know that you will eventually want to retrieve an entire collection of items (for example, all "gobjects"), use a list instead.*

- 3 The next line creates an `CompositeNode` pointer that will eventually hold the location of a polyhedral body. For now, it is set to null.
- 4 The next line uses the `lookup` method to find the item in the `bodyHash` hash table with the key `bodyName`. The variable `body` is set equal to the result. If the operation fails, the program exits.

- 5 The last line calls the *writeBody* function (defined elsewhere in the writer) which will go out and write the body information to a file.



CHAPTER 5

ANATOMY OF A WRITER

This chapter describes the anatomy of a basic writer, using a sample writer called Simple Writer as an example.

CHAPTER OVERVIEW

In this chapter, you'll learn how to get started on the process of creating your own writer by performing the following tasks:

- Building the sample writer (called Simple Writer) which is included on the Game Exchange installation CD.
- Running the Simple Writer on a sample set of data files (also included on the installation CD).
- Analyzing the results of Simple Writer's output.
- Modifying Simple Writer's code in order to introduce some new functionality.
- Looking carefully at the construction and functionality of the Simple Writer.

CODING WRITERS

There is no uniform process for coding a writer. The functionality of your writer will vary tremendously depending on your target application. For this reason, we cannot provide you with step-by-step instructions for coding your writer. Instead, we've taken the approach of providing you with a sample writer called Simple Writer. We'll use this writer in discussing how to build a writer, but it can also serve you as design model for your own writer. You may even find that you can use Simple Writer's code as the foundation for your own writer, which you can freely modify and add functionality to in order to suit your needs.

WORKING WITH SIMPLE WRITER'S FILES

You can see a printout of the code for the Simple Writer in Appendix A. You can find the actual files for Simple Writer on the Game Exchange installation CD inside the *samplewriter/build/* directory.

Inside the */samples* directory you'll also find some sample export data from Mirai.

This chapter will ask you to load and work with these files. First, you'll build the writer for your platform, then you'll run it in order to convert some sample data and analyze the results. After performing a simple modification to Simple Writer, you'll tear into its innards in order to learn more about its construction.

BUILDING SIMPLE WRITER

Follow the instructions below to build Simple Writer on the platform of your choice. These instructions assume that you've already installed the Game Exchange library. If you haven't yet, refer to the *readme* file on the accompanying installation CD.

ON THE NT PLATFORM

To build Simple Writer using Visual C++ (version 5.0 or higher) on the NT platform:

- 1 Open up the sample writer workspace. To do so:
 - In Visual C++, click on *File>Open Workspace*, navigate to *samplewriter/build/simple.dsw*, and click the *Open* button. The sample writer's files appear in the workspace.
- 2 Click on *Build>Build* or press F7 to create an executable. You'll run this executable later in this chapter.

ON THE IRIX PLATFORM

To build Simple Writer using SGI C++ (version 6.2 or higher) on the Irix platform:

- 1 Navigate to *samplewriter/build*.
- 2 Now make the executable.
 - For an o32 compilation, type *make -f makefile -o32*.
 - For an n32 compilation, type *make*.
- 3 The resulting executable will appear in the *samplewriter/bin* directory. You'll run this executable later in this chapter.

NOTE: *For new builds, you can type `make -f makefile -o32 clean` (for o32) or `make clean` (for n32) to remove all object files.*

RUNNING SIMPLE WRITER

Now you're ready to run Simple Writer.

ON THE NT PLATFORM

To run the writer using Visual C++ (version 5.0 or higher):

- 1 Click on *Project>Settings*.
- 2 On the left-side of the dialog box that appears, click once on the *simple* directory (this should be the top-level directory).
- 3 Click on the *Debug* tab. You'll see two fields called *Working directory* and *Program arguments*.

- The first field contains the pathname to the parent directory of the export file that you want to convert (for example, `../././samples`). By default, this field displays the pathname to the `samples` directory, which contains Game Exchange's sample data. If for some reason this field does not display the `samples` directory, type in the correct pathname to this directory relative to your current location.
- The second field lets you specify the file that you want to convert (the input file) and the file name to which you wish to direct the converted data (the output file). Type `cube.gof test.sim` into this file. This specifies `cube.gof` as the input file and `test.sim` as the output file.

NOTE: *If you wanted to parse the data for the entire scene, you would specify the GRP file (for example, `cube.grp`).*

- 4 Click the Ok button to close the Project Settings dialog box.
- 5 Click on *Build>Execute* to run Simple Writer.

ON THE IRIX PLATFORM

To build the writer using SGI C++ (version 6.2 or higher):

- 1 Navigate to `sampleWriter/bin/sgi63`. You'll see the executable for the n32 or the o32 build that you created above.
- 2 Type `simpleWriter .././././samples/cube/cube.gof test.sim` to the to run the sample writer.
 - The first pathname in this command specifies the `cube.gof` file as the file which you wish to convert (the input file).
 - The second pathname specifies `test.sim` as the file to which you wish to direct the converted data (the output file).

NOTE: *If you wanted to parse the data for the entire scene, you would specify the GRP file (for example, `cube.grp`).*

- 3 Press RETURN to run the writer.

NOTE: *If you experience any problems building or running Simple Writer, please contact Nichimen Support by calling 310-577-0500 or sending e-mail to support@nichimen.com.*

ANALYZING THE OUTPUT DATA

Now you're ready to compare the input files to the output generated by Simple Writer.

- In the editor of your choice, open up `/samples/polyhedron_14.gbf` and `/samples/test.sim`.

NOTE: *You may be wondering why we are asking you to open up Polyhedron_14.gbf rather than cube.gof, which was the file you specified as the input file above. Remember that the GOF file references the GBF file, so in running the Simple Writer above you converted the data from both files. We'll work with the GBF file in this section because it contains the topological data for the object, which is of the most interest for comparing input to output.*

Figure 5.1 and Figure 5.2 below show you the contents of these two files. (Note that we added line numbers to the printouts for easy referencing.)

Figure 5.1 . The input data from /samples.

```

1 filetype gx;
2 GrammarVersion 2.1.beta.4;
3 TemplateVersion 2.1.beta.4;
4 HostName "SNAGGLEPUSS";
5 UserName "lsmith";
6 TimeStamp "Wed 29-Sep-99, 1:50 pm";
7 OSName "Windows NT 4.00.1381";
8 ApplicationName "Mirai";
9 ApplicationVersion "1-0-23-0";
10
11 include "gbf-2-1-beta-4.tpl";
12 include "Cube.gmf";
13
14 body Polyhedron-14 (
15   vertices[] < (coord -10.0000 -10.0000 10.0000 ; )
16               (coord -10.0000 10.0000 10.0000 ; )
17               (coord 10.0000 10.0000 10.0000 ; )
18               (coord 10.0000 -10.0000 10.0000 ; )
19               (coord 10.0000 -10.0000 -10.0000 ; )
20               (coord 10.0000 10.0000 -10.0000 ; )
21               (coord -10.0000 10.0000 -10.0000 ; )
22               (coord -10.0000 -10.0000 -10.0000 ; )>
23   faces[] < (normal 0.000000 0.000000 1.000000 ;
24             vertex-indices[] <0;2;3;>
25             vertex-normal-indices[] <0;0;0;>
26             uv-indices[] <0;1;2;> )
27             (normal 0.000000 0.000000 -1.000000 ;
28             vertex-indices[] <4;6;7;>
29             vertex-normal-indices[] <1;1;1;>
30             uv-indices[] <2;3;0;> )
31             (normal 0.000000 1.000000 0.000000 ;
32             vertex-indices[] <1;5;2;>
33             vertex-normal-indices[] <2;2;2;>
34             uv-indices[] <3;1;1;> )
35             (normal 0.000000 -1.000000 0.000000 ;
36             vertex-indices[] <7;0;3;>
37             vertex-normal-indices[] <1;3;0;>
38             uv-indices[] <3;3;1;> )
39             (normal 1.000000 0.000000 0.000000 ;

```

```

40         vertex-indices[] <3;5;4;>
41         vertex-normal-indices[] <0;4;4;>
42         uv-indices[] <0;3;0;> )
43     (normal -1.000000 0.000000 0.000000 ;
44     vertex-indices[] <7;6;1;>
45     vertex-normal-indices[] <1;5;5;>
46     uv-indices[] <0;3;3;> )
47     (normal -1.000000 0.000000 0.000000 ;
48     vertex-indices[] <7;1;0;>
49     vertex-normal-indices[] <1;5;6;>
50     uv-indices[] <0;3;0;> )
51     (normal 1.000000 0.000000 0.000000 ;
52     vertex-indices[] <5;3;2;>
53     vertex-normal-indices[] <7;0;2;>
54     uv-indices[] <3;0;3;> )
55     (normal 0.000000 -1.000000 0.000000 ;
56     vertex-indices[] <7;3;4;>
57     vertex-normal-indices[] <1;0;4;>
58     uv-indices[] <3;1;1;> )
59     (normal 0.000000 1.000000 0.000000 ;
60     vertex-indices[] <5;1;6;>
61     vertex-normal-indices[] <8;5;8;>
62     uv-indices[] <1;3;3;> )
63     (normal 0.000000 0.000000 -1.000000 ;
64     vertex-indices[] <6;4;5;>
65     vertex-normal-indices[] <8;4;9;>
66     uv-indices[] <3;2;1;> )
67     (normal 0.000000 0.000000 1.000000 ;
68     vertex-indices[] <2;0;1;>
69     vertex-normal-indices[] <2;6;5;>
70     uv-indices[] <1;0;3;> ) >
71 normals[] < 0.577350 -0.577350 0.577350 ;
72             -0.577350 -0.577350 -0.577350 ;
73             0.577350 0.577350 0.577350 ;
74             0.577350 -0.577350 0.577350 ;
75             0.577350 -0.577350 -0.577350 ;
76             -0.577350 0.577350 0.577350 ;
77             -0.577350 -0.577350 0.577350 ;
78             0.577350 0.577350 0.577350 ;
79             -0.577350 0.577350 -0.577350 ;
80             0.577350 0.577350 -0.577350 ; >
81 uv-array[] < (uv[] < 0.000000;1.000000;> )
82              (uv[] < 1.000000;0.000000;> )
83              (uv[] < 1.000000;1.000000;> )
84              (uv[] < 0.000000;0.000000;> ) >
85 material "Cube_local";
86 )

```

Figure 5.2. The output data (polyhedron_14.gbf).

```

1 Simple format test.sim
2 version 0.9
3

```

```

4 Start Polyhedron-14
5 Diffuse Color 0.224853, 0.224853, 0.638095
6 Vertex 8
7 -10,-10,10
8 -10,10,10
9 10,10,10
10 10,-10,10
11 10,-10,-10
12 10,10,-10
13 -10,10,-10
14 -10,-10,-10
15 Normal 10
16 0.57735,-0.57735,0.57735
17 -0.57735,-0.57735,-0.57735
18 0.57735,0.57735,0.57735
19 0.57735,-0.57735,0.57735
20 0.57735,-0.57735,-0.57735
21 -0.57735,0.57735,0.57735
22 -0.57735,-0.57735,0.57735
23 0.57735,0.57735,0.57735
24 -0.57735,0.57735,-0.57735
25 0.57735,0.57735,-0.57735
26 UV 4
27 0,1
28 1,0
29 1,1
30 0,0
31 Face 12
32 0,2,3,0,0,0,0,1,2
33 4,6,7,1,1,1,2,3,0
34 1,5,2,2,2,2,3,1,1
35 7,0,3,1,3,0,3,3,1
36 3,5,4,0,4,4,0,3,0
37 7,6,1,1,5,5,0,3,3
38 7,1,0,1,5,6,0,3,0
39 5,3,2,7,0,2,3,0,3
40 7,3,4,1,0,4,3,1,1
41 5,1,6,8,5,8,1,3,3
42 6,4,5,8,4,9,3,2,1
43 2,0,1,2,6,5,1,0,3
44 End Polyhedron-14

```

Before comparing these files in detail, keep in mind that Simple Writer is *not* a fully-functional writer. It is simply meant to provide you with a framework for creating your own writer. It writes out a fictitious format that only supports limited data for materials (diffuse color only), vertices, normals, UVs, and faces. Simple Writer also requires that exported objects be composed exclusively of triangulated faces, and it offers very limited error messaging.

You can see the data for materials, vertices, normal, UVs and faces in the output file:

- Line 5 shows the data for materials (in the form of RGB values).
- Lines 6 through 14 show the data for vertices (in the form of Cartesian coordinates for each vertex).

- Lines 14 through 24 show the data for normals (in the form of a normalized vectors).
- Lines 26 through 30 show the data for UVs.
- Lines 31 through 43 show the data for faces. The first three numbers of each line represent indices to a face's vertices. The middle three numbers of each line represent indices to the vertex normals of the face. The last three number represent indices to the UVS of each vertex of the face.

You can easily see the correspondences between the input and output data. For example, look at lines 15 through 22 of the input file. You can see that each of these lines is an array containing the Cartesian coordinates for each vertex on the object. When Simple Writer parsed the file, it wrote these values to the appropriate section of the output file.

DECONSTRUCTING THE WRITER

Now that you've built and run Simple Writer, you're ready to take a closer look at its code. As you delve into the code in this section, remember again that Simple Writer has very limited functionality. As mentioned above, it only supports limited data for materials, vertices, normals, UVs, and faces. Although it will suffice as a foundation, your own writer is likely to evolve into something of considerably more complexity.

Most of the functionality for Simple Writer is contained within a class called *simpleWriter*. The *simpleWriter* class includes just two public methods, one for parsing GE files and one for writing out data to a file.

The class also contains numerous private methods which are designed to iterate over nodes of a specific type and write out the data contained therein. For example, the class includes a private method called *writeVertices* which is designed to iterate over nodes of the type *vertex*. This method then calls another method called *writeVertex* which actually writes out the vertex data. You'll explore these methods in greater detail in *The Simple.cpp File* on page 71.

Three separate files contain the code for the Simple Writer

- A file called *simple.h* defines the *simpleWriter* class.
- A file called *simple.cpp* includes the implementation of the *simpleWriter* class.
- A file called *simpleMain.cpp*, a very small main program, makes an instance of the *simpleWriter* class and calls the class' two methods.

The following sections discuss the contents of each of these files in greater detail. In Appendix A, you'll find a printout of each of these files. Each line of the printout is numbered, so that you can refer to specific sections of the code as you read the discussion below.

THE SIMPLE.H FILE

First, let's look at the *simple.h* file. We've already stated that most of the Simple Writer's functionality is contained within the *simpleWriter* class. The *simple.h* file declares the members of this class. (As you'll read below these members' functionality is defined in the *simple.cpp* file.)

Look at the printout of this file on page 85. You can see that lines 35 through 48 declare the public methods for the class. These include:

- The *openGX2File* method, which parse a Game Exchange file.
- The *write* method, which writes out the converted data to a file.

Lines 50 through 59 declare the private methods of the class. The functionality of these classes is described in detail below.

Notice that line 51 declares an instance of the *geGapi* class. Elsewhere, the code will use this instance to parse the input file and get the root node.

Finally, look at lines 56 through 58. Line 56 defines an instance of the *geList* class, which will be used to store pointers to nodes at the object level of the tree. Lines 57 and 58 define instances of the *geHash* class. These two hash tables will store pointers to nodes at the body and material levels of the tree. (Remember earlier we said that you typically use hash tables to store data to which you require ready reference later. For the Simple Writer, these two node types are the only ones that warrant using a hash table.) The *bodyHash* instance is indexed according to the name of the body, and the *materialHash* instance is indexed according to the name of the part to which a material is applied.

THE SIMPLEMAIN.CPP FILE

This is a very simple main program. As you can see from lines 37 through 38, the program requires that a user provide two arguments when executing Simple Writer. The first argument is a pathname to the input file to be converted; the second is a pathname for the output file. If the user fails to provide the two arguments, the program generates an error message. Otherwise, the program proceeds by calling the two public methods of the *simpleWriter* class.

THE SIMPLE.CPP FILE

The *simple.cpp* file contains the meat of the Simple Writer code. Herein, all of the methods of the *simpleWriter* class are defined. For example, the lines beginning 40 define the *openGX2File* method which is designed to actually parse the input file.

If you jump to line 366, you can see the definition of the other public method, *write*. Look over this code to gain a better understanding of this method. In general terms, this method is defined to iterate over the first level of inferi-

or nodes and store the nodes of the type *gobject* in a list and nodes of the type *body* and *material* in separate hash tables. Finally, the method leaves off by calling *writeObject*, one of the private methods.

Rather than dissecting the functionality of each of these private methods, let's discuss them in general terms. For each category of data that is supported (for example, vertices or UVs), two methods are defined. The first of these methods is responsible for iterating across the nodes of a list; the second is responsible for actually writing out the data. In the case of *faces*, for example, a method called *writeFaces* iterates across the nodes of the *face* type, while a method called *writeFace* actually writes out the data to a file. These private methods call each other in succession in order to write out all categories of data that the Simple Writer supports. The figure below show the order in which these methods call each other.

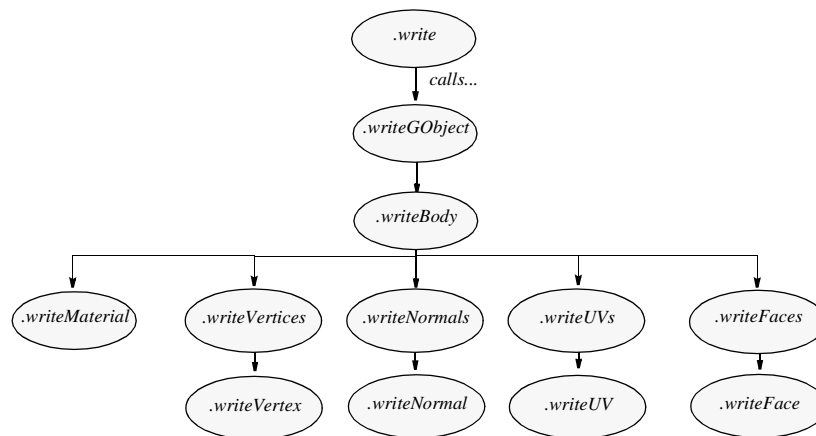
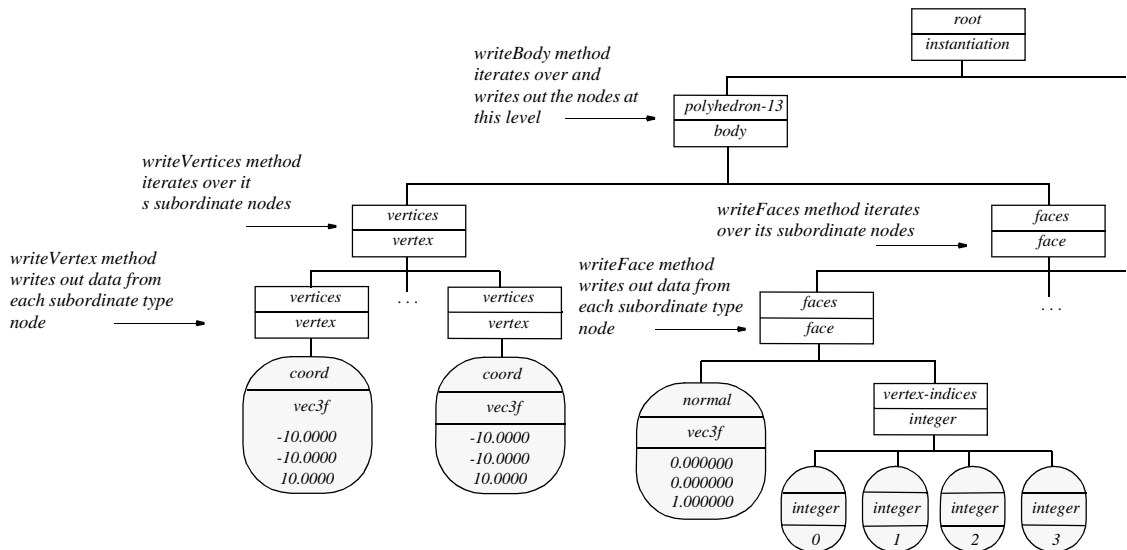


Figure 5.3 Sequence by which methods are called in Simple Writer

This series of method calls steps down the tree node level by node level. At each level, the relevant method iterates across the node while its partner method writes out the data. This process may be easier to grasp if you look at the figure below, which is a simpler version of the tree you saw back in Chapter 3.

Figure 5.4. Methods which operate on the various levels of a data tree.



The figure above shows a portion of the general process by which Simple Writer traverses the tree and writes out data. The *write* method iterates over and writes out the data for the nodes directly under the root node. It then calls *writeVertices* and *writeFaces* (as well as *writeMaterials*, *writeNormals*, and *writeUVs*, which are not shown on the figure). Each of these methods then iterate over their subordinate type nodes. For each node, they call a partner method (for example, *writeFaces* calls *writeFace*) to handle the output of the data.

This system of partner methods works well for extracting data from relatively simple structures, but in coding your writer, you'll have to consider how you will handle extraction from other structures as well. For example, you may need to design functions to extract data from:

- single type nodes
- single composite nodes
- entire branches of a tree

MODIFYING SIMPLE WRITER

You'll recall from the discussion above that Simple Writer only supports one attribute for materials (diffuse color). Suppose that you wanted to modify SimpleWriter to provide more support for materials. For example, say that you wanted to add the necessary functionality for writing out all pathnames to the texture maps that have been assigned in the base domain. This section shows you how you would do so. By following this example carefully, you'll have a clearer idea of how to begin modifying Simple Writer to support your target format.

- 1 The first thing that you need to do is look in the GMF template to find out where the pathnames to texture maps are stored. Scanning over this template, you see the following snippet of code:

```
template Base-domain (
    vec3f DIFFUSE-COLOR 0.8 0.8 0.8;
    map TEXTURE-MAP;
    vec3f SPECULAR-COLOR 0.0 0.0 0.0;
    float SPECULAR-EXPONENT 1.0;
    vec3f EMISSION-COLOR 0.0 0.0 0.0;
    float OPACITY 1.0;
    vec3f AMBIENT-COLOR 0.2 0.2 0.2;
)
```

If you want, you can look this section up for yourself by opening the GMF template file that is included in the *samples* directory of the installation CD.

When this file is parsed, the variable declaration for *TEXTURE-MAP* will become a node called *TEXTURE-MAP*. This tells you that you that the writer will eventually need to find a node in the data tree named *TEXTURE-MAP*.

- 2 In the same GMF file, skip to the template declaration for the *map* type. You'll see the following:

```
template map (
    string name;
    integer width; integer height;
)
```

Remember when we talked back in *Template Declaration Section* on page 32 about structuring template declarations hierarchically? This is just such an example. After parsing, this declaration will form a child node to the *TEXTURE-MAP* node. So, now you know that once the writer has arrived at the *TEXTURE-MAP* node, it will have to traverse downward to find the *name* node and extract the pathname to the texture map file from there.

- 3** You can more clearly see the manner in which the data is structured by looking in the sample content file (*samples/Cube.gmf*).

```
Base (  
    DIFFUSE-COLOR 0.22485265 0.2248526 0.63809526;  
    SPECULAR-COLOR 0.0 0.0 0.0;  
    EMISSION-COLOR 0.0 0.0 0.0;)  
    TEXTURE-MAP (  
        name "image_1.tif";  
        width 256;  
        height 256;))
```

*data element that you
want to extract*

From the *Base* node level, the writer needs to find the *TEXTURE-MAP* node, and from the *TEXTURE-MAP* node it needs to find the *name* node. Once there, the writer can extract and write out the data.

Now that you've determined where the data resides, you're ready to make the necessary code modifications to Simple Writer.

- 4** First you'll need to modify the function that writes the base domain. (You could also incorporate this new functionality into its own function, but for the sake of simplicity, just add it to the existing function.) You can

find the relevant portion of the code in the *simple.cpp* file (which is located in *simplewriter/src* directory of the installation CD). It looks like this:

```
*****
// writeMaterial
// purpose: This function will write out the name of the texture file.
void simpleWriter::writeMaterial (geCompositeNode* bodyMaterial)
{
    geCompositeNode::childIterator first(*bodyMaterial);
    geCompositeNode::childIterator last (*bodyMaterial);
    last.end();

    for (; first < last; first.increment()) {
        if ((*first)->getName() == "Base") {
            geCompositeNode::childIterator firstBase(**first);
            geCompositeNode::childIterator lastBase (**first);
            lastBase.end();

            for (; firstBase < lastBase; firstBase.increment()) {
                if ((*firstBase)->getName() == "DIFFUSE-COLOR") {
                    geVector<float> > diffuse(3);
                    diffuse = ((geTypeNode<geVector<float> >
>*)(*firstBase))->getValue();
                    simpleOutFile << "Diffuse Color "
                                << diffuse.getAt(0) << ", "
                                << diffuse.getAt(1) << ", "
                                << diffuse.getAt(2) << endl;
                }
            }
        }
    }
}
```

← insert new code here

The code shows that an iterator already exists for traversing the *Base* nodes child nodes in search of the *DIFFUSE-COLOR* node. You can easily modify this code to search for the *TEXTURE-MAP* node by adding the following lines at the position in the code designated above:

```
if ((*firstBase)->getName() == "TEXTURE-MAP") {
}
```

- 5 The lines above take care of the search, but once the writer has *found* the *TEXTURE-MAP* node, it needs the capability to iterate over the children of the node. Just below the new lines you added above, insert the following highlighted code to create the iterator:

```
if ((*firstBase)->getName() == "TEXTURE-MAP") {
    geCompositeNode::childIterator firstTextureMap(**firstBase);
    geCompositeNode::childIterator lastTextureMap (**firstBase);
    lastTextureMap.end();

    for (; firstTextureMap < lastTextureMap; firstTextureMap.increment()) {
    }
}
```

- 6 During the iteration, the writer needs to be able to search for the name node. Appending the following highlighted line of code will take care of this functionality.

```
if ((*firstBase)->getName() == "TEXTURE-MAP") {
    geCompositeNode::childIterator firstTextureMap(**firstBase);
    geCompositeNode::childIterator lastTextureMap (**firstBase);
    lastTextureMap.end();

    for (; firstTextureMap < lastTextureMap;
        firstTextureMap.increment()) {
        if ((*firstTextureMap)->getName() == "name") {
        }
    }
}
```

- 7 When the writer finds the right node, it needs to cast down to a type node that holds a string in order to extract its value. The following highlighted code lines will handle the casting:

```
if ((*firstBase)->getName() == "TEXTURE-MAP") {
    geCompositeNode::childIterator firstTextureMap(**firstBase);
    geCompositeNode::childIterator lastTextureMap (**firstBase);
    lastTextureMap.end();

    for (; firstTextureMap < lastTextureMap;
        firstTextureMap.increment()) {
        if ((*firstTextureMap)->getName() == "name") {
            geTypeNode<geString>* holdTexture;
            holdTexture = (geTypeNode<geString>*)(*firstTextureMap);
        }
    }
} }
```

- 8** Now with just a few last modifications, the writer can easily get the value, verify it is not empty, and write out the pathname:

```

if ((*firstBase)->getName() == "TEXTURE-MAP") {
    geCompositeNode::childIterator firstTextureMap(**firstBase);
    geCompositeNode::childIterator lastTextureMap (**firstBase);
    lastTextureMap.end();

    for (; firstTextureMap < lastTextureMap;
        firstTextureMap.increment()) {
        if ((*firstTextureMap)->getName() == "name") {
            geTypeNode<geString>* holdTexture;
            holdTexture = (geTypeNode<geString>*)(*firstTextureMap);
            geString textureMap = holdTexture->getValue();
            if (textureMap != "") {
                simpleOutFile << "Texture Map " << textureMap <<
endl;
            }
        }
    }
}

```

You've seen from this exercise that it is relatively simple to add support materials. By the same general process, you could add support for other data elements.



CHAPTER 6

CONCLUSION

This chapter concludes this manual by reviewing what you've learned and providing you with some suggestions for getting started on your own writer.

REVIEW

This manual has provided you with a good overview of the workings of Game Exchange:

- You're now familiar with how the Game Exchange format and library fit into a data conversion pipeline.
- You now have a detailed knowledge of the Game Exchange format—the files it produces and their internal structure.
- You understand how Game Exchange files are parsed into data trees.
- You possess a basic knowledge of the classes contained within the Game Exchange library and how their methods can be employed to search for, reformat, and write out desired data.
- Finally, you're acquainted with the design behind a sample writer, which can serve as a foundation for your own writer.

GETTING STARTED ON YOUR OWN WRITER

You're now ready to start working with Game Exchange at ground level. This manual will conclude with a few suggestions for getting started on your own Game Exchange writer.

● BROWSE THROUGH THE GAME EXCHANGE REFERENCE GUIDE

You've been introduced in this manual to some of the most commonly used methods for each Game Exchange library class. However, there are many more methods which we did not discuss. These are all thoroughly catalogued in the *Game Exchange Reference Guide*. Become as familiar as possible with all the methods of each class to avoid needlessly coding functionality that already exists.

● CAREFULLY ANALYZE THE TEMPLATES

When you feel that you have an adequate knowledge of the Game Exchange library's classes, you're ready to begin the process of coding your own writer. You may find it helpful to start by exporting a sample scene from your source application in the Game Exchange format, then printing out all of the template files. Next, mark up the data that requires conversion into your target format. Marking each data element will make it much easier to build into your writer the appropriate searches for node names on the parsed data tree.

● NOTE ALL REQUIRED CONVERSIONS

As you isolate data elements for conversion, pay special attention to the export conventions for each data element. For example, Mirai writes all angles in degree units, but your target application may require radians. You'll need to code the appropriate conversions into your writer.

● INTEGRATE THE COMMONLY USED CLASSES

As learned in Chapter 4, the basic process for writing data involves first iterating to the right node on a data tree, then making a call to the appropriate function or method to extract the data. You use the `CompositeNode` class and the `TypeNode` class to do this. As you begin to code your writer, keep in mind that instances of these two classes will probably constitute its bulk:

- The `CompositeNode` Class

An instance of the `CompositeNode` class gets you to the right node by creating and implementing iterators. You saw an instance of the `CompositeNode` class back in *The `geCompositeNode` Class* on page 44.

- The `TypeNode` Class

An instance of the `TypeNode` class checks a node by name, casts down to retrieve its value and writes out the value to file. You saw an example of the `TypeNode` class back in *The `geTypeNode` Class* on page 45.

● USE HASH TABLES FOR STORING DATA SETS FROM WHICH YOU REQUIRE RAPID RETRIEVAL

Using hash tables in your writer can greatly speed up data extraction. Therefore, you'll want to use them to store any data items which can easily be mapped to a key (such as, skeletal joints, polyhedral bodies, or materials).

● WHEN TESTING, OUTPUT THE RESULT OF EACH STOP OF AN ITERATION

As you're testing your writer, you may find it helpful to print out the current location of your writer in the data tree. Use the `getType` and `getName` methods in the manner described in *The `geAbstractNode` Class* on page 43 to find out the name or type of the current node, then output the result to the screen.



APPENDIX A

SIMPLE WRITER CODE

This appendix contains the code for the Simple Writer.

SIMPLE WRITER CODE

Three separate files contain the code for the Simple Writer

- A file called *simple.h* defines a class called *simpleWriter*.
- A file called *simple.cpp* includes the implementation of the class *simpleWriter*. (The *simpleWriter* class encompasses most of the functionality of the Simple Writer writer.)
- A file called *simpleMain.cpp*, a very small main program, makes an instance of the *simpleWriter* class and calls the classes two methods.

You can find *simple.h* on the Game Exchange installation in the *simplewriter/include* directory, and *simple.cpp* and *simpleMain.cpp* in the *simplewriter/src* directory.

This appendix contains a printout of each of these files. For your convenience, each line of code in these printouts is numbered.

SIMPLE.H

```
//
// *****
// ** (c) Copyright 1999 Nichimen Corporation. All rights reserved.
//
// The software, data, and information contained herein are proprietary
// to, and comprise valuable trade secrets of, Nichimen Corporation,
// which intends to keep such software, data, and information confidential
// and to preserve them as trade secrets. They are given in confidence by
// Nichimen Corporation pursuant to a written license agreement, and may
// be used, copied, transmitted, and stored only in accordance with the
// terms of such license.
//
// RESTRICTED RIGHTS LEGEND
// Use, duplication, and disclosure by the Government are subject to
// restrictions as set forth in subdivision (c)(1)(ii) of the Rights in
// Trademark Data and Computer Software Clause at FAR 52.227-7013.
//
// Nichimen Corporation
// Nihonbashi 3-11-1,
// Chuo-ku, Tokyo
// Japan
//
// *****
//

#ifdef _SIMPLEWRITER_H
#define _SIMPLEWRITER_H

#include "gegapi.h"
#include <fstream.h>

class simpleWriter
{
public:
    bool openGX2File(geString geFile);
    void write      (geString simpleFile);
private:
    void writeHeader(geString filename);
    void writeBody   (geCompositeNode* body);
    void writeVertice (geCompositeNode* vertice);
    void writeVertex  (geCompositeNode* vertex);
    void writeNormals (geCompositeNode* normals);
    void writeNormal  (geCompositeNode* normal);
    void writeUVs     (geCompositeNode* uvs);
    void writeUV      (geCompositeNode* uv);
    void writeFaces   (geCompositeNode* faces);
    void writeFace    (geCompositeNode* face);
    void writeMaterial(geCompositeNode* material);
    void writeGObject(geCompositeNode* gObject);
};
```

```
private:
    geGapi    gapi;
    geString  inFile;
    geString  outFile;
    ofstream  simpleOutFile;

    geList< geCompositeNode* >      gObjectList;
    geHash< geString, geCompositeNode* > bodyHash;
    geHash< geString, geCompositeNode* > materialHash;
};

#endif // _SIMPLEWRITER_H
```

SIMPLE.CPP

```

//
// *****
// ** (c) Copyright 1999 Nichimen Corporation. All rights reserved.
//
// The software, data, and information contained herein are proprietary
// to, and comprise valuable trade secrets of, Nichimen Corporation,
// which intends to keep such software, data, and information confidential
// and to preserve them as trade secrets. They are given in confidence by
// Nichimen Corporation pursuant to a written license agreement, and may
// be used, copied, transmitted, and stored only in accordance with the
// terms of such license.
//
// RESTRICTED RIGHTS LEGEND
// Use, duplication, and disclosure by the Government are subject to
// restrictions as set forth in subdivision (c)(1)(ii) of the Rights in
// Trademark Data and Computer Software Clause at FAR 52.227-7013.
//
// Nichimen Corporation
// Nihonbashi 3-11-1,
// Chuo-ku, Tokyo
// Japan
//
// *****
//
// SimpleWriter.cpp
// Purpose: SimpleWriter was created to provide a simple example of writing
// files to a target format from the Game Exchange format using
// the Game Exchange library.
//
// Date: 8/26/99
// By: lsmith
// Notes:
// The format written to is an imaginary simple(sim) format.
//
#include "simple.h"
#include <stdlib.h>
// *****
// openGX2File
// purpose: To open and parse the Game Exchange files. These are the
// standard calls used to open a Game Exchange file.
bool simpleWriter::openGX2File(geString geFile)
{
    gapi.ignoreWarnings(true);
// gapi.setTablesPath(geString("g:\\tmp"));
    return (gapi.parse((char *)geFile.asChar()));
}

```

```

// *****
// writeVertex
// purpose: Take a vertex node pointer list and write the vertex to the
//           output file. The first node holds the actual vertex.
void simpleWriter::writeVertex (geCompositeNode* vertex)
{
    geVector<float> vec(3);

    vec = ((geTypeNode<geVector<float> >*)(vertex->getChildNode(0)))->getValue();
    simpleOutFile << vec.getAt(0) << ", "
        << vec.getAt(1) << ", "
        << vec.getAt(2) << endl;
}

// *****
// writeVertice
// purpose: This function takes a pointer to a node which has the list of
//           vertice, then calls the writeVertex function on each vertex nodes.
void simpleWriter::writeVertice(geCompositeNode* vertice)
{
    geCompositeNode::childIterator first(*vertice);
    geCompositeNode::childIterator last (*vertice);
    last.end();

    // Write format information to the file
    simpleOutFile << "Vertex " << vertice->getChildNodeCount() << endl;

    // Iterate through the list and write each vertex
    for (; first<last; first.increment())
        writeVertex( (geCompositeNode*)(*first) );
}

// *****
// writeNormal
// purpose: This function takes a pointer to a normal node and write the
//           information from it to the file.
void simpleWriter::writeNormal (geCompositeNode* normal)
{
    geVector<float> vec(3);

    vec = ((geTypeNode<geVector<float> >*)normal)->getValue();
    simpleOutFile << vec.getAt(0) << ", "
        << vec.getAt(1) << ", "
        << vec.getAt(2) << endl;
}

// *****
// writeNormals
// purpose: Iterates over a list of normals calling the function that writes
//           each normal out.
void simpleWriter::writeNormals(geCompositeNode* normals)
{

```



```

geCompositeNode::childIterator first(*normals);
geCompositeNode::childIterator last (*normals);
last.end();

// write Normal header to file
simpleOutFile << "Normal " << normals->getChildNodeCount() << endl;

// iterate through the list and write each vertex
for (; first<last; first.increment())
    writeNormal( (geCompositeNode*)(*first) );
}

// *****
// writeUV
// purpose: Takes a UV node and writes a single UV to file.
void simpleWriter::writeUV(geCompositeNode* uv)
{
    geCompositeNode* theUV = (geCompositeNode*)uv->getChildNode(0);

    float u = ((geTypeNode<float>*)(theUV->getChildNode(0)))->getValue();
    float v = ((geTypeNode<float>*)(theUV->getChildNode(1)))->getValue();
    simpleOutFile << u << "," << v << endl;
}

// *****
// writeUVs
// purpose: Write the UV header and iterates over the list and calls
//          the function that writes the UV to the file.
void simpleWriter::writeUVs(geCompositeNode* uvs)
{
    geCompositeNode::childIterator first(*uvs);
    geCompositeNode::childIterator last (*uvs);
    last.end();

    // output UV header information
    simpleOutFile << "UV " << uvs->getChildNodeCount() << endl;

    // iterate through the list and write each vertex
    for (; first<last; first.increment())
        writeUV( (geCompositeNode*)(*first) );
}

// *****
// writeFace
// purpose: This function writes a face out. For this simple example, we
//          are assuming that all faces are triangles and then we are
//          writing out the vertice index, the normal index, and then the
//          UV index.
void simpleWriter::writeFace(geCompositeNode* face)
{
    geCompositeNode* vertInd = NULL;
    geCompositeNode* normInd = NULL;

```

```

geCompositeNode* uvsInd    = NULL;

// from the face level find the vertex Index, normal Index, uv Index
geCompositeNode::childIterator first(*face);
geCompositeNode::childIterator last (*face);
last.end();

// store all of the nodes
geString nodeName;
for (; first<last; first.increment()) {
    nodeName = (*first)->getName();

    if      (nodeName == "vertex-indices") {
        vertInd  = (geCompositeNode*)(*first);
    } else if (nodeName == "vertex-normal-indices") {
        normInd  = (geCompositeNode*)(*first);
    } else if (nodeName == "uv-indices") {
        uvsInd   = (geCompositeNode*)(*first);
    }
}

if ((vertInd != NULL) && (vertInd->getChildNodeCount() > 0)) {
    simpleOutFile << ((geTypeNode<integer>*)(vertInd->getChildNode(0)))->getValue()
        << ", "
        << ((geTypeNode<integer>*)(vertInd->getChildNode(1)))->getValue()
        << ", "
        << ((geTypeNode<integer>*)(vertInd->getChildNode(2)))->getValue();
}

if ((normInd != NULL) && (normInd->getChildNodeCount() > 0)) {
    simpleOutFile << ", "
        << ((geTypeNode<integer>*)(normInd->getChildNode(0)))->getValue()
        << ", "
        << ((geTypeNode<integer>*)(normInd->getChildNode(1)))->getValue()
        << ", "
        << ((geTypeNode<integer>*)(normInd->getChildNode(2)))->getValue();
}

if ((uvsInd != NULL) && (uvsInd->getChildNodeCount() > 0)) {
    simpleOutFile << ", "
        << ((geTypeNode<integer>*)(uvsInd->getChildNode(0)))->getValue()
        << ", "
        << ((geTypeNode<integer>*)(uvsInd->getChildNode(1)))->getValue()
        << ", "
        << ((geTypeNode<integer>*)(uvsInd->getChildNode(2)))->getValue();
}

simpleOutFile << endl;
}

// *****
// writeFaces

```

```

// purpose: Write face header then iterate over faces and call the function
//           that writes each one out.
void simpleWriter::writeFaces(geCompositeNode* faces)
{
    geCompositeNode::childIterator first(*faces);
    geCompositeNode::childIterator last (*faces);
    last.end();

    simpleOutFile << "Face " << faces->getChildNodeCount() << endl;

    // iterate through the list and write each vertex
    for (; first<last; first.increment())
        writeFace( (geCompositeNode*)(*first) );
}

// *****
// writeMaterial
// purpose: This function will write out the name of the texture file.
void simpleWriter::writeMaterial (geCompositeNode* bodyMaterial)
{
    geCompositeNode::childIterator first(*bodyMaterial);
    geCompositeNode::childIterator last (*bodyMaterial);
    last.end();

    for (; first < last; first.increment()) {
        if ((*first)->getName() == "Base") {
            geCompositeNode::childIterator firstBase(**first);
            geCompositeNode::childIterator lastBase (**first);
            lastBase.end();

            for (; firstBase < lastBase; firstBase.increment()) {
                if ((*firstBase)->getName() == "DIFFUSE-COLOR") {
                    geVector<float> > diffuse(3);
                    diffuse = ((geTypeNode<geVector<float> > >*)(*firstBase))->getValue();
                    simpleOutFile << "Diffuse Color "
                                << diffuse.getAt(0) << ", "
                                << diffuse.getAt(1) << ", "
                                << diffuse.getAt(2) << endl;
                }
            }
        }
    }
}

// *****
// writeBody
// purpose: This function will go through the body and (hash)hold the
//           important nodes (faces, vertices, normals, and uvs). Then it
//           will call the function that writes each out.
void simpleWriter::writeBody(geCompositeNode* body)
{
    geCompositeNode* vertice = NULL;

```

```

geCompositeNode* normals = NULL;
geCompositeNode* uvs     = NULL;
geCompositeNode* faces   = NULL;
geCompositeNode* bodymat = NULL;

// from the body level find the vertex, face, normal, and uv arrays. Also
// save the material
geCompositeNode::childIterator first(*body);
geCompositeNode::childIterator last (*body);
last.end();

// store all of the nodes
geString nodeName;
for (; first<last; first.increment()) {
    nodeName = (*first)->getName();

    if (nodeName == "vertices") {
        vertice = (geCompositeNode*)(*first);
    } else if (nodeName == "normals") {
        normals = (geCompositeNode*)(*first);
    } else if (nodeName == "uv-array") {
        uvs     = (geCompositeNode*)(*first);
    } else if (nodeName == "faces") {
        faces   = (geCompositeNode*)(*first);
    } else if (nodeName == "material") {
        bodymat = (geCompositeNode*)(*first);
    }
}

// Check for elements and if they are there, write them
if (bodymat != NULL) {
    geString matName = ((geTypeNode<geString>*)bodymat)->getValue();
    if (matName != "") {
        geCompositeNode* material;
        if (!materialHash.lookup(matName, material)) exit(-1);
        writeMaterial(material);
    }
}

if ((vertice != NULL) && (vertice->getChildNodeCount() != 0)) {
    writeVertice(vertice);
} else {
    cerr << "Warning, no vertex information found" << endl;
}

if ((normals != NULL) && (normals->getChildNodeCount() != 0)) {
    writeNormals(normals);
} else {
    cerr << "Warning, no normal information found" << endl;
}

if ((uvs != NULL) && (uvs->getChildNodeCount() != 0)) {

```

```

        writeUVs(uvs);
    } else {
        cerr << "Warning, no uv information found" << endl;
    }

    if ((faces != NULL) && (faces->getChildNodeCount() != 0)) {
        writeFaces(faces);
    } else {
        cerr << "Warning, no face information found" << endl;
    }
}

// *****
// writeGObject
// purpose: This function takes a pointer to a GObject and writes it.
void simpleWriter::writeGObject(geCompositeNode* gObject)
{
    geCompositeNode* gObjectBody      = NULL;

    // find body and material for object
    geCompositeNode::childIterator first(*gObject);
    geCompositeNode::childIterator last (*gObject);
    last.end();

    geString nodeType;
    // store all of the nodes that we need
    for (; first < last; first.increment()) {
        nodeType = (*first)->getName();

        if (nodeType == "body") {
            gObjectBody = (geCompositeNode*)(*first);
        }
    }

    // look up and write the body
    if (gObjectBody != NULL) {
        geString bodyName = ((geTypeNode<geString>*)gObjectBody)->getValue();
        if (bodyName != "") {
            geCompositeNode* body = NULL;
            if (!bodyHash.lookup(bodyName, body)) exit(-2);
            simpleOutFile << "Start " << bodyName << endl;
            writeBody(body);
            simpleOutFile << "End " << bodyName << endl;
        } else {
            // body name null
            exit(-3);
        }
    }
}

// *****
// writeHeader

```

```

// purpose: Writes the header information
void simpleWriter::writeHeader(geString filename)
{
    simpleOutFile << "Simple format " << filename << endl;
    simpleOutFile << "version 0.9" << endl << endl;
}

// *****
// write
// purpose: This function is designed to open the file for writing and then
//          collect all of the nodes from the highest level and call the
//          functions that will write them out.
void simpleWriter::write(geString simpleFile)
{
    // open the file
    simpleOutFile.open((simpleFile.asChar()), ios::out);
    if (simpleOutFile == NULL) {
        cerr << "Unable to open file for writing" << endl;
        exit(-4);
    }

    writeHeader(simpleFile);

    // get the root node to the tree
    geCompositeNode* root = (geCompositeNode*)gapi.getInstRootNode();

    // from the top level find the body, gObject, skeleton, and material
    geCompositeNode::childIterator first(*root);
    geCompositeNode::childIterator last (*root);
    last.end();

    // store all of the nodes that we need
    geString nodeType;
    for (; first < last; first.increment()) {
        nodeType = (*first)->getType();

        if (nodeType == "gobject") {
            gObjectList.addTail( (geCompositeNode*)(*first) );
        } else if (nodeType == "body") {
            bodyHash.setItem((*first)->getName(), (geCompositeNode*)(*first));
        } else if (nodeType == "material") {
            materialHash.setItem((*first)->getName(), (geCompositeNode*)(*first));
        }
    }

    // now go through the list and write all of the gObjects (Since there
    // should be relatively few objects, I will not iterate)
    for (int i = 0; i < gObjectList.getCount(); i++) {
        writeGObject(gObjectList.getAt(i));
    }

    simpleOutFile.close();
}

```

SIMPLEMAIN.CPP

```
//
// *****
// ** (c) Copyright 1999 Nichimen Corporation. All rights reserved.
//
// The software, data, and information contained herein are proprietary
// to, and comprise valuable trade secrets of, Nichimen Corporation,
// which intends to keep such software, data, and information confidential
// and to preserve them as trade secrets. They are given in confidence by
// Nichimen Corporation pursuant to a written license agreement, and may
// be used, copied, transmitted, and stored only in accordance with the
// terms of such license.
//
// RESTRICTED RIGHTS LEGEND
// Use, duplication, and disclosure by the Government are subject to
// restrictions as set forth in subdivision (c)(1)(ii) of the Rights in
// Trademark Data and Computer Software Clause at FAR 52.227-7013.
//
// Nichimen Corporation
// Nihonbashi 3-11-1,
// Chuo-ku, Tokyo
// Japan
//
// *****
//

#include "simple.h"
#include <stdlib.h>

void main(int argc, char** argv)
{
    if ((argc <= 2) || (argc > 3)) {
        cerr << "Incorrect number of arguments!" << endl;
        exit(-1);
    }

    simpleWriter myWriter;
    geString gxInFile(argv[1]);
    geString simpleOutFile(argv[2]);

    if (myWriter.openGX2File(gxInFile))
        myWriter.write(simpleOutFile);
}
```




APPENDIX B

ADVANCED TOPICS

This appendix discusses some advanced topics which you may have need to address as you construct your writer.

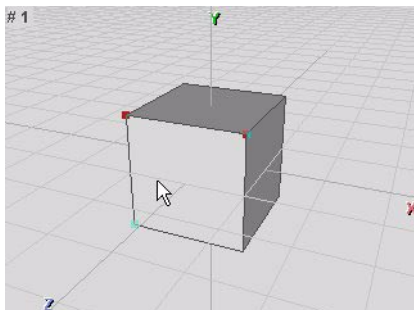
USING CUSTOM PROPERTIES WITH GAME EXCHANGE

If Mirai is your source application, you can make use of the Property Editor plug-in to add custom properties to different elements in your scene's content before you export to Game Exchange. This approach lets you write properties to the Game Exchange format that your target application understands but are not supported in the Mirai template files. For example, if your target application understands a property you've defined as "infrared," you can easily use the Property Editor to assign any vertex, edge, face, polyhedron, joint, bone, or skeleton in your scene the "infrared" property with the appropriate value. Then you can export that data to Game Exchange data format. As this property is not in the Mirai template files it will not undergo type checking (see *Type Checking* on page 40). Instead, the property and associated values will appear as instantiations in the appropriate content file.

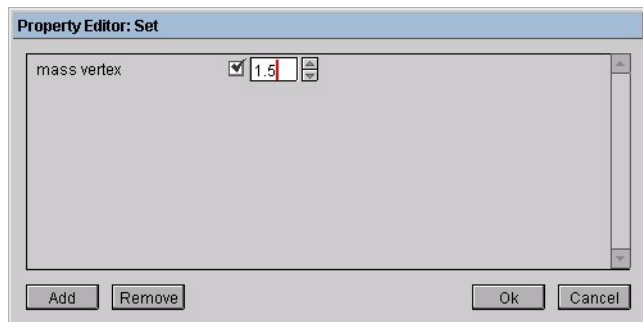
To better clarify the use of the Property Editor, this section will quickly walk through the procedures for adding a custom property to a scene element and writing out that element to Game Exchange. To take a very simple example, suppose you wanted to add a custom property called "mass-vertex" to several vertices on a cube. You would do the following:

NOTE: *The steps below are intended to give you just a cursory overview of how to use the Property Editor. For full instructions on using this plug-in, refer to the Mirai Support section of the Nichimen website (www.Nichimen.com).*

- 1 Define the "mass-vertex" property in a Property Editor template file (*not* to be confuse with the Game Exchange template file).
- 2 Use the *Applications>Plug-Ins>Load Property Template* command to load the template you created above into the Property Editor.
- 3 Select the vertices to which you wish to apply the custom property.



- 4 Use the *Set Property* command to assign the “mass-vertex” property and the appropriate value (say 1.5) to the collection of vertices.



- 5 Finally, use *File>Export>Game Exchange* to export the cube to the Game Exchange format.

These steps would write the following data to the the cube’s GBF content file:

```
body Polyhedron-13 (

    vertices[] < (
coord -10.0000 -10.0000 10.0000 ;
        properties (
            MASS-VERTEX 1.5;
        ) )
    (
coord -10.0000 10.0000 10.0000 ;
        properties (
            MASS-VERTEX 1.5;
        ) )
    (
coord 10.0000 10.0000 10.0000 ;
        properties (
            MASS-VERTEX 1.5;
        ) )
    ) )
```

Note that the “mass-vertex” custom property and the assigned value of 1.5 is tied to each of the vertices. Game Exchange wrote this data directly to the GBF content file, deriving the format from the Property Editor template rather than the standard Game Exchange template. Knowing that this custom data was being written directly to the GBF file, you could design your Game Exchange writer to extract and convert it as necessary for import into your target application.

You can see from this example using the Property Editor affords you a great deal of flexibility in writing out custom data to Game Exchange.

USING VERSIONING INFORMATION

You read in *Header Section* on page 30 that later versions of Mirai and Nendo write out detailed versioning information to the header section of each Game Exchange export file. This section will discuss in more detail how you can make full use of versioning information.

Recent versions of Mirai and Nendo write out version numbers for the following Game Exchange pieces:

- parsing tables
- template files
- the library

This information is reported inside each Game Exchange file's header section in a format similar to the following:

```
GrammarVersion 2.1.0.0;  _____ current parsing tables
TemplateVersion 2.1.0.0;  _____ current template version
LibraryVersion 2.1.0.0;  _____ current library version
```

When you begin coding a writer, it's a good idea to note the current version numbers for *GrammarVersion*, *TemplateVersion*, and *LibraryVersion*. That way later you'll have a reference by which to gauge how many modification have been made to Game Exchange since you created your writer.

The following sections make suggestions for using versioning information from *GrammarVersion*, *TemplateVersion*, and *LibraryVersion*.

USING GRAMMAR VERSION INFORMATION

You read in *Parsing Tables* on page 40 that Game Exchange requires access to two tables in order for parsing to occurred—the DFA and LLR tables. *GrammarVersion* increments any changes that Nichimen makes to these tables. Additionally, it increments any changes that Nichimen makes to the general output format of parsed files (for example, if Nichimen decided to write a new element to each Game Exchange file's header section, this change would be incremented in *GrammarVersion*).

After you create your writer, you'll want to pay careful attention to any changes to this version number. Any change, even one that is quite minor, may require modifications to your writer in order for it to continue working properly.

USING TEMPLATE VERSION INFORMATION

Back in “Template Files” on page 18, we mentioned that as Mirai and Nendo continue to evolve new features, Nichimen will update each product’s template files in order to offer export support for as much data as possible. *TemplateVersion* increments such changes to the template files.

Changes to this number should not impair the performance of your writer, but you may want to keep track of the status of *TemplateVersion* in case Nichimen adds support to the template files for a data element that you can make use of in your pipeline.

USING LIBRARY VERSION INFORMATION

Anytime that Nichimen makes an implementation change to one of the Game Exchange library classes or adds an entirely new class to the library, it will increment the *LibraryVersion* number. The *LibraryVersion* number that you see output in your Game Exchange files represents the version of the Game Exchange library for which the files are optimized.

If you’re writing Game Exchange files from a source application that is relatively new and is thus optimized for a more recent version of the Game Exchange library than the one that you used in coding your writer, you may find that certain functions in your writer will not work properly. Nichimen Support (800-366-4743) can help you sort out such issues.



APPENDIX C

GLOSSARY OF TERMS

This appendix contains a glossary of key terms used in this manual.

content file	A Game Exchange export file that contains actual data from the source application. See “ <i>Game Exchange File Types</i> ,” on page 18.
declaration	A statement included in a template file composed of a type name, a variable name, and a set of values (optionally) that creates a new data type for export. This acts as a template for future exports See “ <i>Declarations</i> ,” on page 25.
Game Exchange Format	A standardized grammar for exported data from Nichimen and other third-party applications. The format permits conversion to other formats via writers created using the Game Exchange library.
Game Exchange Library	A collection of classes designed to aid in the implementation of writers for converting data written in the Game Exchange format into a format that is readable by a target application.
header	A section in a file which identifies it as being Game Exchange output and includes versioning information that helps the Game Exchange library determine how to parse the file. See “ <i>Header Section</i> ,” on page 30).
include	A section in a Game Exchange file which includes a list of references to other Game Exchange files. Include sections permit parsing of an entire collection of files. See “ <i>Include Section</i> ,” on page 31.
instantiations	A statement written to a content file that contains exported data in the format specified by its corresponding declaration. See “ <i>Instantiations</i> ,” on page 28.
list iterator	An instance of the <code>geListIterator</code> class that sequentially accesses data elements in a list or, with modification, the nodes of a composite node. See “ <i>The geListIterator Class</i> ,” on page 57.
composite node	An instance of the <code>geCompositeNode</code> that incorporates the elements of a custom type or array. This becomes a branching structure in a data tree. See “ <i>Composite Nodes</i> ,” on page 37.
parsing	The process of converting Game Exchange output files into a memory structure, also known as a tree, which is composed of type nodes and composite nodes.
template file	A Game Exchange export file that contains the declarations for supported data elements. See “ <i>Game Exchange File Format</i> ,” on page 23.
tree	The hierarchical data structure that is created when a Game Exchange file or collection of files is parsed. Parsing typically produces two trees, one for template data and one for content data. See “ <i>What is Parsing?</i> ,” on page 36.
type checking	The process of comparing the template and content trees against each other to check for data errors. See “ <i>An Example of Parsing</i> ,” on page 37.

type name	A designation for the kind of data that a variable may hold (for example, integer, boolean, float). You can choose from several type names that are built in to the library, or define your own custom type names. See “ <i>Game Exchange File Structure</i> ,” on page 29.
type node	An instance of the <code>geTypeNode</code> that contains the values for a particular data element. If derived from a template file, a type node may contain default values. If derived from a content file, the type node will contain actual values derived from the exported scene. This becomes a leaf structure on a data tree. See “ <i>Type Nodes</i> ,” on page 36.
writer	A program designed to convert export files from a source application into a format which is readable by a target application. In the context of Game Exchange, a writer includes functionality for parsing output files, iterating over the resulting data tree, and writing out values from the appropriate nodes. See “ <i>Using the Game Exchange Library to Convert Data to a Platform Specific Format</i> ,” on page 14.
variable name	A placeholder for a value of a specific type. See “ <i>Variable Names and Values</i> ,” on page 24.
vectors	A container for a small set of data elements (usually a maximum of four). In a Game Exchange writer, a vector is an instance of the <code>geVector</code> class.



INDEX

B

boolean type 24

C

content files 19
 defined 104
 GAF file 19
 GBF file 19
 GCF file 19
 GEF file 19
 GLF file 19
 GMF file 19
 GOF file 19
 GRP file 19
 GSF file 19
header sections within 30
include sections within 31
instantiations included within 28
template instantiation sections within 32

D

data trees
 defined 104
declarations 25
 defined 104
 syntax for 26

F

float type 24

G

GAF file 19
Game Exchange Format
 applications which support 14
 defined 104
 exporting content in 14
Game Exchange Library 8, 14
 defined 104
 geAbstractIterator class 42, 56
 geAbstractNode class 42, 43
 geCompositeNode class 42, 44
 geGapi class 42, 46
 geHash class 42, 58
 geHashIterator class 42
 geList class 42, 48
 geListIterator class 42, 57
 geString class 55
 geString classes 42
 geTypeNode class 42, 45
 geVector class 42, 52
 including in a preprocessor directive 42
 reporting problems with 10
 system requirements for using 8
GBF file 19
GCF file 19
geAbstractIterator class 42, 56
 begin method 57
 end method 45, 57
 increment method 45
geAbstractNode class 42, 43
 geAbstractNode constructor 45
 getChildNodeCount method 45
 getName method 43, 44
 getType method 43

- geCompositeNode class 42, 44
- GEF file 19
- geGapi class 42
 - gapi.parse method 47
 - ignoreWarnings method 47
 - setTablesPath method 47
- geHash class 42, 58
 - geHash constructor 59
 - getAt method 59
 - lookUp method 59, 60
 - setItem method 59
- geHashIterator class 42
- geList class 42, 48
 - addHead method 49, 50, 51, 57
 - addTail method 49, 52
 - geList constructor 50, 51, 57
 - getAt method 51, 52
 - getHead method 51
 - getTail method 51
 - insertAfter method 50
 - insertBefore method 50
 - printList method 49, 50, 51
 - removeAt method 51
 - removeHead method 49
 - removeTail method 49
 - using to iterate over the nodes of a list node 57
- geListIterator class 42, 57
 - geListIterator constructor 57
 - using to iterate over a list node 58
- geNullIterator class 56
- geString class 42, 55
- geTypeNode class 42, 45
 - casting down to 46
 - getValue method 46
 - geTypeNode constructor 54, 55
 - setValue method 54
- geVector class 42, 52
 - getAt method 46, 55
 - geVector constructor 53, 54
 - print method 53
 - setAt method 53, 54
- geVector constructor
 - setLength constructor 53
- GLF file 19
- GMF file 19
- GOF file 19

- GRP file 19
- GSF file 19

H

- hash tables 58, 71, 81
 - creating 59
 - extracting values from 59
 - items within 58, 60
 - keys within 58, 60
- header sections 30
 - defined 104
 - versioning information contained within 15, 30

I

- include sections 31
 - defined 104
- instantiations 28
 - defined 104
 - syntax for 28
- integer type 24

L

- list iterator
 - defined 104
- list nodes 37
 - defined 104
- lists 48
 - adding elements to 48
 - creating 48
 - extracting elements from 51
 - removing elements from 48

P

- parsing
 - data trees created by 39
 - defined 36, 104
 - groups of files together 20
 - required tables 15, 40
 - DFA file 40, 47
 - LLR file 40, 47
 - type checking 40
 - with the geGapi class 46

R

reserved keywords 25

__root@@ 25

boolean 25

extends 25

float 25

include 25

integer 25

properties 25

string 25

template 25

vec2f 25

vec2i 25

vec3f 25

vec3i 25

vec4f 25

vec4i 25

S

string type 24

T

template declaration sections 32

template files 18

changes to 15

declarations included within 25

defined 104

header sections within 30

include sections within 31

template declaration sections within 32

template instantiation sections 32

type checking 40

defined 104

type names 29

boolean 24

custom 24

defined 105

float 24

integer 24

string 24

vec2f 24

vec2i 24

vec3f 24

vec3i 24

vec4f 24

vec4i 24

type nodes 36

casting down to 46, 54

defined 105

V

values 24

actual values 28, 29

default values 24, 28, 40

variable names 24

defined 105

vec2f type 24

vec2i type 24

vec3f keyword 25

vec3f type 24

vec3i type 24

vec4f type 24

vec4i keyword 25

vec4i type 24

vectors 52

creating 53

defined 105

extracting data from 53

W

writers 64

building a sample writer on Irix 65

building a sample writer on NT 65

coding 64

defined 105

modifying a sample writer 74

running a sample writer on Irix 66

running a sample writer on NT 65

sample code 84

simple.cpp file 87

simple.h file 71

simpleMain.cpp file 71, 95

sample output 68