RenderWare Studio

Game Production Manager

Version 2.0.1

Published 07 September 2004

Contents

Introduction

The Game Production Manager (GPM) is a set of COM interfaces for creating production-quality game data. The core of the GPM is the `RWSMake` COM object, which is responsible for defining a process whose output is the final game data image on disk. RenderWare Studio uses the Game Production Manager (GPM) to create production-quality data for your game engine.

When you connect to a target console in RenderWare Studio Workspace, the application calls the GPM to build the game data stream, and then sends the stream to the target console.

The GPM builds the game data stream from the game database according to a set of default build rule (p.5) scripts supplied with RenderWare Studio. It processes files in the game database to create a set of *target files* that together form the game data stream. You can customize the build rules, or add your own, to match the specific requirements of your game.

Before you customize the GPM

Before attempting to customize the default rules, you need to understand the basic concepts (p.5) of the GPM. We also recommend that you follow thetutorial (p.22).

Basic concepts

The GPM is a tool for creating the files you want, known as *target files*, from the files you already have, known as *dependent files*. You need to provide the GPM with several pieces of information before you use it:

- How to build set of target files, given a set of dependent files
- The relationships between dependent files and target files, so that a change to a dependent file results in all related target files being rebuilt
- In a multi-step process, the sequence in which target files should be built (because one target file could have another target file as a dependent file)

In RenderWare Studio, all of this information is supplied to the GPM in the form of XML *build rule* files that each contain one or more *rules*.

Rules and tasks

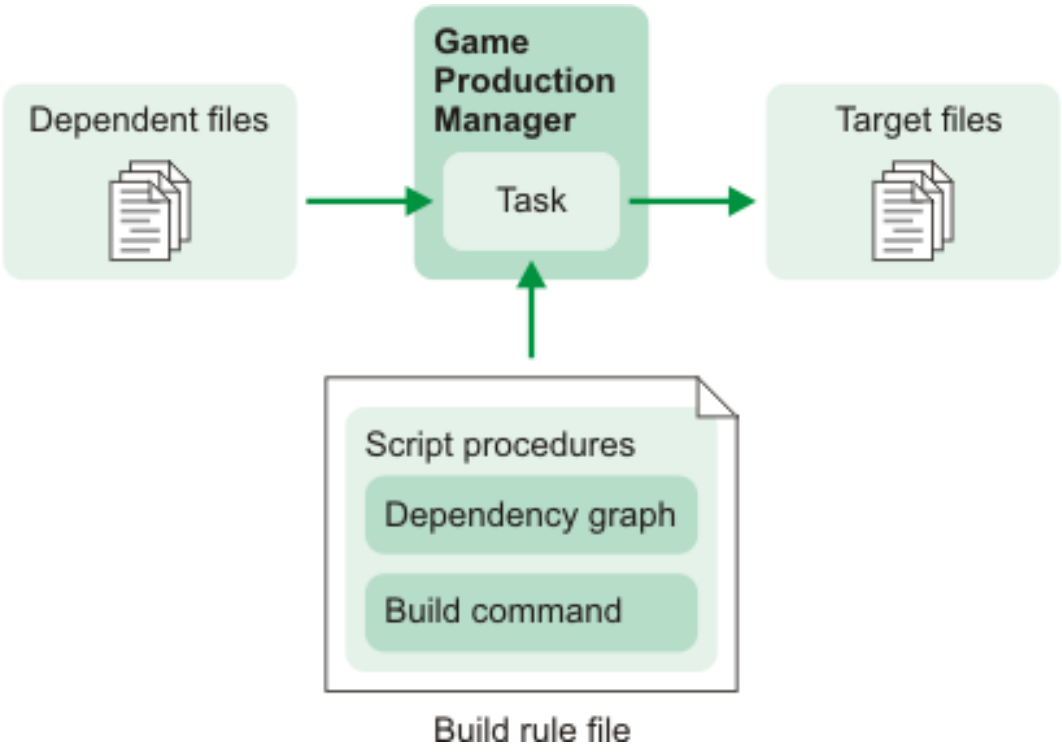
Each GPM rule can contain two script procedures that between them provide the information specified above:

- The *dependency graph procedure* identifies the dependent and target files.
- The *build command procedure* contains the commands necessary to generate the target files.

Typically, the build command procedure uses the contents of the dependent files to create (or update) the target files. However, you can also use the GPM for rules that have:

- Target files, but no dependent files. The GPM runs the build command only if any of the target files do not exist.
- Dependent files, but no target files. The GPM always runs the build command.

Before the build process is invoked, rules are loaded into an `RWSMake` object. These “instances” of GPM rules are referred to as *tasks*.



When the build process is invoked, the dependencies between tasks are evaluated (any tasks whose targets are the dependencies of another task are built first), and any tasks whose dependencies are newer than their targets will be built.

Here is the format of a build rule file:

```
<?xml version="1.0" ?>
<build version = "1.0">
   <rule name="TestRule"
         dependencygraph="DependencyGraphProcedure"
         buildcommand="BuildCommandProcedure">

      <script language="VBScript"><![CDATA[

         Sub DependencyGraphProcedure (Task, Parameters...)
            Task.AddDependency path of dependent file
            ...
            Task.AddTarget path of target file
            ...
         End Sub

         Sub BuildCommandProcedure (Task, Parameters...)
            ...
         End Sub

      ]]></script>
   </rule>

   ...

</build>
```

In the listing above, the formatting represents that rules must have unique names, the procedures may have user-defined names, the scripting language is user-defined, and task parameters are user-defined (apart from the first parameter, which is always an `IRWSTask` object representing the current task). If

you wish, you can also reference other COM objects, type libraries, or external scripts from a rule:

```
<rule name="TestRule"
      dependencygraph="DependencyGraphProcedure"
      buildcommand="BuildCommandProcedure">

  <object id="FileSystem" progid="Scripting.FileSystemObject"/>
  <reference object="CSL.RWSTarget.RWSComms"/>
  <script language="VBScript" src="BufferTools.vbs"/>

   ...

</rule>
```

The `Task` variable in the script represents a task object in which the GPM stores dependency and target file information. `AddDependency()` and `AddTarget()` are methods of the task object.

For more information on the build rule file format, see the detailed reference (p.11).

Running the GPM

To run the GPM, you use script like this:

```
' Create an RWSMake COM object
Set objMake = CreateObject("CSL.RWSBuild.Make")

' Load the build rule
objMake.LoadRule path of build rule file' Add the task
objMake.AddTask "rulename", other arguments...' Do the task
objMake.Build
```

Tip: If you want to create all target files from scratch, rather than updating them only if necessary, then call `objMake.Clean()` before `objMake.Build()`. The `Clean()` method deletes all the target files of the task (and those of any subtasks, as described below).

In the listing above, the `objMake` variable represents the GPM. The `LoadRule()` method loads the build rule file into the GPM, while `AddTask()` defines a task that you want to perform. You specify the rule you want to use, and any arguments you want to pass to the rule's procedures. In the listing above, *`rulename`* refers to the `name` attribute of a `<rule>` element in the build rule file.

The `Build()` method calls the dependency graph procedure of the rule named in the call to `AddTask()`. In effect, this:

```
objMake.AddTask "myrule", argument2, argument3, ...
```

becomes this:

```
Depend objTask, argument2, argument3, ...
```

That is, the GPM replaces the rule name with an object variable representing the task, and passes any additional arguments onto the procedure. When the dependency graph procedure ends, the GPM compares the age of the dependency and target files. If the target files are not up-to-date, then the GPM calls the build command procedure.

Note: The dependency graph and build command procedures for a rule must expect the same set of arguments. The first argument must always be an object variable that represents the current task.

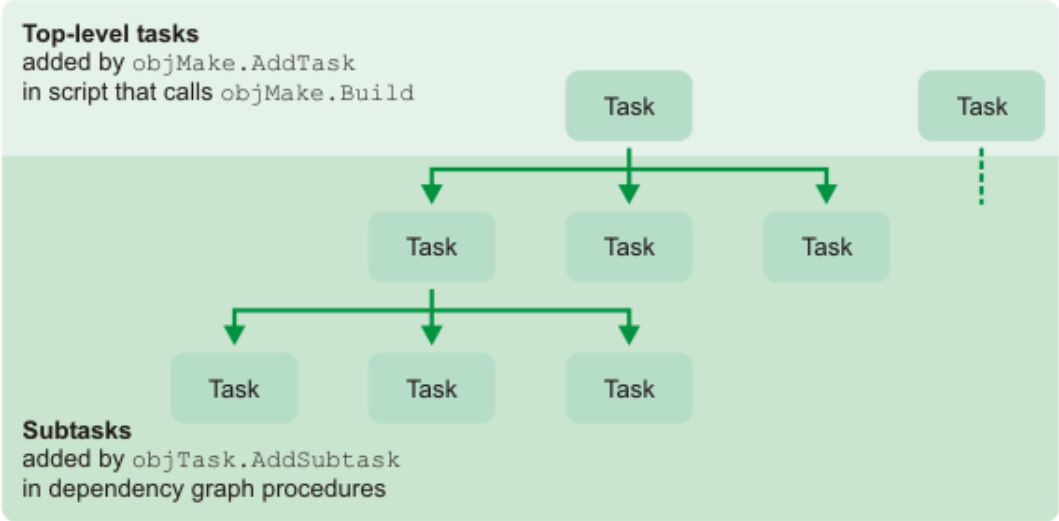
When the build command procedure ends, the GPM returns control to the calling script (the statement following the `Build` method).

Subtasks

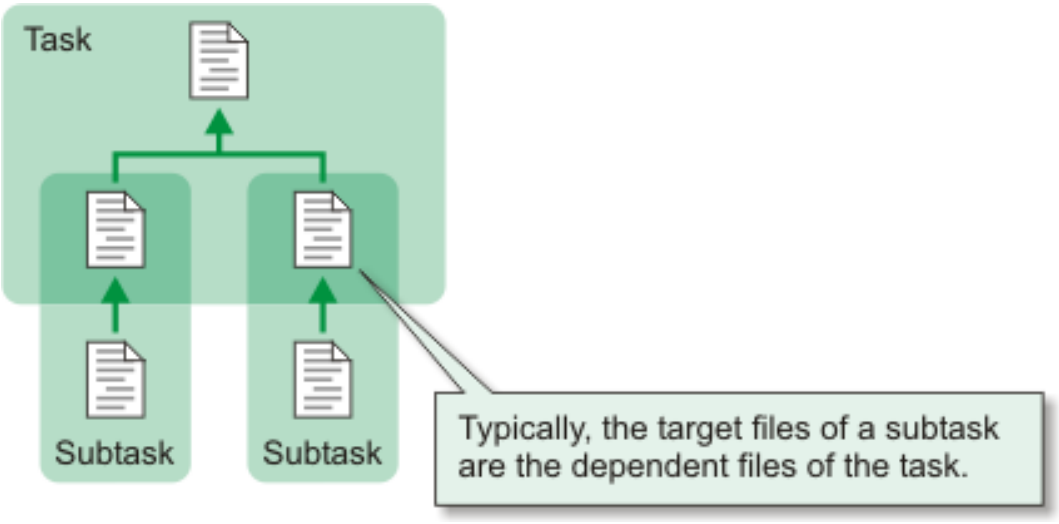
So far, we have considered the simple situation where target files are created from dependent files in a single step. More complex situations can require additional steps, with intermediate files. To make it easier to handle these situations, you can:

- Add *subtasks* to a top-level task.
- Add subtasks to subtasks.
- Add multiple top-level tasks.

The GPM ensures that the target files of a subtask are up-to-date before calling the task's build command procedure.



For example, you might have a set of artwork files in different formats that must be converted into a common format, and then combined into a single file. You could define subtasks to perform the data conversion, and have the top-level task combine the converted files.



To add a subtask, you insert a call to the `AddSubtask()` method (similar to `AddTask()` for top-level tasks) into the dependency graph procedure of the rule:

```
Sub DependencyGraphProcedure (Task, Parameters...)
   Task.AddDependency path of dependent file
   ...
   Task.AddTarget path of target file
   ...

   Task.AddSubtask rule name, other arguments...

   ...
End Sub
```

Sharing data throughout the process

A number of functions are accessible to all tasks' script code via the IRWSGlobal interface. You can call any methods in this interface in the script using global scope. For example, any script function can call IRWSGlobal's `SetParam` method simply by calling `SetParam`:

```
 Sub DependencyGraphFunction (Task, strPath)
     Dim strDest
     strDest = GetParam ("DESTDIR") ' Calling IRWSGlobal.GetParam
     SetParam "CURRENTPATH", strDest & strPath ' Calling
IRWSGlobal.SetParam
 End Sub
```

The IRWSGlobal, IRWSMake, and IRWSTask interfaces each have a built-in mechanism for associating named objects at different scope. This allows data to be shared either globally (throughout the whole process), or locally (within a particular task). To store a named reference to any object (from built-in script types to COM objects), call the `SetParam` method. The object may be retrieved at any point by calling `GetParam`, providing the name as a parameter.

Reporting information

The IRWSGlobal interface provides a way of reporting any events through a centralized reporting mechanism. You can report errors, warnings, and messages by calling the `BuildError`, `BuildWarning`, and `BuildLog` methods respectively. To handle these events, and any system events, simply respond to the events defined in the IRWSMake interface. The following example illustrates handling these events in VBScript:

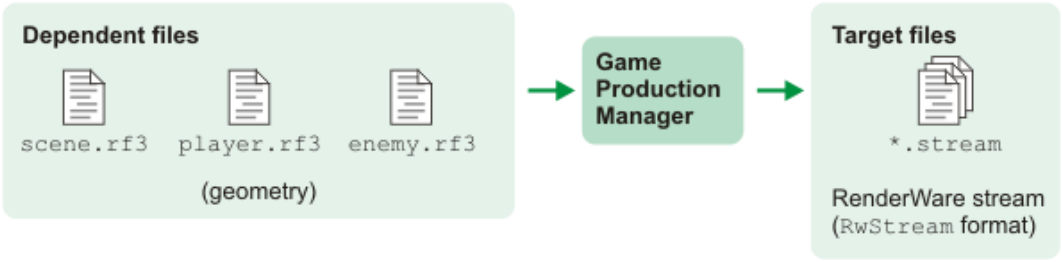
```
' Create the make object and connect its events
Set Make = CreateObject ("CSL.RWSBuild.Make")
WScript.ConnectObject Make, "Make_"

' Load in our rules
Make.LoadRule "Test.Rule"
…' Add the tasks
Make.AddTask "Test"
…' Do the tasks
Make.Build

' Handle a system error
Sub Make_SystemError (BuildError)
   WScript.Echo "[SYSERROR] " & BuildError.Rule & "[" &
BuildError.LineNumber & "] "_
                            & BuildError.Message & " " &
BuildError.Description
End Sub
```

```
' Handle a build error
Sub Make_BuildError (strRule, strError)
   WScript.Echo "[ERROR] " & strError
End Sub

' Handle a build warning
Sub Make_BuildWarning (strRule, strWarning)
   WScript.Echo "[WARNING] " & strWarning
End Sub

' Handle a build message
Sub Make_BuildLog (strRule, strLog)
   WScript.Echo "[LOG] " & strLog
End Sub
```

What has this got to do with RenderWare Studio?

Typically, game engines that use RenderWare (such as the RenderWare Studio Game Framework) require data in a RenderWare stream (`RwStream`) format. This stream combines data (such as geometry) from various files into a file that your game engine loads at run time.

The GPM automates the process of building these RenderWare stream files (the target files) from these various dependent files.



Build rule file format

A build rule file is an XML file with the following format:

```
<?xml version="1.0"?>
<build version="1.0">
   <rule name="rule_name"
         dependencygraph="procedure_name1"
         buildcommand="procedure_name2"
         alwaysbuild="true|false"
         strictvalidation="true|false">
      <script language="VBScript"><![CDATA[

         Sub procedure_name1 (objTask, other arguments...)

            objTask.AddDependency ...
            objTask.AddTarget ...
            objTask.AddSubTask ...
            ...

         End Sub

         Sub procedure_name2 (objTask, other arguments...)

            Script that the GPM runs if the target files are old or do not exist

         End Sub

         Other procedures or functions

      ]]></script>
   </rule>

   Other rules

</build>
```

<?xml version="1.0"?>
This XML declaration on the first line is optional.

<build>
A build rule file must contain a <build> root element. The version attribute on this element refers to the lowest version of the GPM required by this build rule file.

<rule>
A build rule file can contain one or more rules, each defined by a <rule> element (the listing above contains only one rule).

A <rule> element can contain three attributes:

name
Required. Uniquely identifies this rule across all of the rules that you load into the GPM.

dependencygraph
Optional (see note below). Name of the procedure (inside a <script> element; see below) that identifies the target files, the dependent files and the subtasks for this rule.

buildcommand

Optional (see note below). Name of the procedure that the GPM calls:

- If any of the dependent files are newer than any of the target files

  or

- If any of the target files do not exist

  or

- If no target files are defined for this rule

Typically, this procedure uses the contents of dependency files to create target files.

`alwaysbuild`
Optional. Sets validation conditions that the build rule must meet. Values are:

`true`
The rule must have a build command, and its task cannot have dependencies.

`false`
If the rule has no build command, then its task cannot have dependencies or targets.

If `strictvalidation="false"`, then the value of `alwaysbuild` is ignored.

If `strictvalidation="true"`, and the build rule fails to meet the conditions specified by the value of `alwaysbuild`, then an error occurs.

`strictvalidation`
Optional. See `alwaysbuild`.

Note: A rule can have dependent files, but no target files; or target files, but no dependent files. However, a rule cannot have no dependent files *and* no target files.

A `<rule>` element can contain a subset of the elements allowed in a Windows Script (`.wsf`) file:

`<script>`
Contains the procedures named by the `dependencygraph` and `buildcommand` attributes of the `<rule>` element. Can also contain additional procedures or functions that can be called by the `dependencygraph` and `buildcommand` procedures.

The `<script>` element can contain script written in any language supported by the Windows Script engine, such as VBScript (used in the listing above) or JScript.

`<object>`
(msdn.microsoft.com/library/en-us/script56/html/wsEleObject.asp)
(Not shown in the listing above.) Defines a global object for use in the script without using functions such as `CreateObject()`.

`<reference>`
(msdn.microsoft.com/library/en-us/script56/html/wsEleReference.asp)
(Not shown in the listing above.) Allows you to use constants

defined in an external type library.

Introduction

GPM processing is similar to the processing of a makefile (www.gnu.org/software/make/) during a source code build, with a few advantages:

- You write the build rules in a script language (such as VBScript), so the process is much more dynamic.
- You can set up more complex dependency/target relationships through the use of cross-dependencies and independent sub-build processes.
- Dependency/target relationship can optionally be entirely self-contained through the `AddSubtask` method.
- Rule files that define separate tasks are logically separated due to the nature of the architecture.
- Data of any form (from built-in script types to COM objects) can be shared across the whole process, or within each task.
- The process is often simplified because rule arguments can be any type of object.
- Less tools are often required, as code can optionally be written in script.
- Development of the final build process can be very rapid due to the nature of script development.
- Error reporting during development of a process is more accurate (rule name and line/character number).
- The process can be deliberately halted at any point (with meaningful messages) if certain requirements are not met (for example, number of triangles too high).
- Any tools can be better integrated, as there is no longer a restriction on tools being command-line driven.
- Complex processes are simpler to debug because the entire dependency hierarchy is automatically available via the `IRWSSubtask` interface.
- A centralized location for reporting information, warnings, and errors is immediately available at any point in the process.

GPM Walkthrough

In this tutorial:

You are taken on a walkthrough of the Game Production Manager (GPM) is intended to give you a fundamental grasp of how the GPM and build rules work, out-of-the-box, so that you are able to customize and extend them more effectively.

This will not be an exhaustive tour riddled with detail, but a look at key areas to provide an accurate conceptual framework. When you complete the walkthrough, you should have a clear picture of what is happening when you “build” your game.

Requirements:

Some familiarity with the GPM (p.5) as described below and in further more detailed topics.

Familiarity with the script debugging capabilities of Visual Studio .NET interface that we are using later to walk through examples of the build rule files.

Tip: .NET users can install the utility “VSTweak” and associate the .rule extension with the `.vb` file extension to get syntax highlighting in .NET. Associate `.rule` files with .NET to get `.rule` links working from within this document.

GPM Basics

What is the GPM? A set of interfaces to create game-ready data on disk.

What are build rules? Code which determines when and how to build game-ready data. They use the GPM to create files on disk (targets), rebuilding automatically when the files are out of date through dependency-checking.

A rule does not necessarily both have to check dependencies and build data; however, it needs to do one or the other.

- If a rule is to check for dependencies, it needs a `DependencyGraph` routine. This is simply a routine where you specify dependent files (you can think of them as inputs) and target files (or outputs).
- If a rule is to build data, it needs a `BuildCommand` routine. This is where you would specify how the rule builds the data. If there are no dependencies, the data will always be built. If there are dependencies, then this is where you would implement how the dependent files get transformed into target files.

Note: The GPM (a set of COM interfaces) does not define how data is built. The GPM is simply a tool that structures your build process. The build rules contain the code which builds the data through scripts and calls to executables, transforming dependent files into target files. Implementing the `DependencyGraph` routine tells the GPM what files to check so it knows whether to run the `BuildCommand`. Put another way, your custom data will not be processed unless you write a new rule implementing `BuildCommand` or add code to an existing rule's `BuildCommand`.

RenderWare Studio's convention is typically to use the rule's filename for the two routines. For example, in the file, `Folder.rule`:

```
dependencygraph = FolderDependencies,
```

and

```
buildcommand = FolderCommand.
```

DependencyGraph Walkthrough

Before taking a look at how dependencies are created in the default build rules, let's look at the game data itself. In RenderWare Studio, you typically access pieces of the game database through the Game Explorer. You can see that the game database is just a tree of objects that you structure as you see fit. Folders are the main “object type” that give your database structure and make it manageable for you and your team.

Note: If some of your game data is completely outside of RenderWare Studio but still needs processing into a game-ready format, you can still use the GPM for this. Keeping a single build process is a good idea. However, since we are interested in the default build rule behavior (which deals with objects in the game database), that is not covered here.

So what do the default build rules do with folders? Let's look at some code for `Folder.rule`:

```
Sub FolderDependencies
    AddDependency on folder's XML file (Folder.PersistentFilename)
           AddSubTasks for all the folder's Assets
           AddSubTasks for all the folder's AssetFolders
           AddSubTasks for all the folder's Folders
           AddSubTasks for all the folder's Entities
```

As you can see, the code simply cycles through all the possible object types that the game database could contain.

The GPM then ensures that the target files of subtasks are up-to-date before calling the task's build command procedure. It does this by running the subtask's respective dependency and build procedures. In this example, all of the folder's children will be built and up-to-date before the folder itself is built.

```
    Add folder to the sent objects list if not already in it
    AddTarget on the folder's output file (.rws on disk)
End Sub
```

The sent objects list is a list that keeps objects from getting built more than once into a stream; it is literally the order in which objects will be written into the stream.

To get a concrete idea of how this structure works, let's look into a dependency routine. We will start with `AssetDependencies` in `Asset.rule`. Place the keyword `stop` at the beginning of the routine and save.

1. Open the project `Example Animation.rwstudio` that is located in the RenderWare Studio examples directory - typically `c:\rw\studio\examples`. Clean the project and then perform a build. Your debugger should pick up the breakpoint.
2. Before stepping, examine the call stack. Work your way up the stack, noticing that the first asset is the child of an entity. The entity belongs in a

folder that belongs in another folder, in turn. Since this asset will be processed by a subtask, it will be built before the entity or folders.

3. Look into the Asset object to get an idea of the data it contains: ID, persistent filename, type, name, etc.
4. Step through `AssetDependencies` if you like. Much of the code handles `rf3` files (an intermediate file format that requires additional dependencies since it must be exported to a binary format); don't get too lost in it.
5. Continue (F5) through two assets. You should see that this asset is of type `rwID_HANIMANIMATION` and is a child of an asset folder (via the call stack).
6. Stop debugging and remove the stop keyword (and save the rule).

In essence, the `AssetDependencies` routine is doing:

```
Sub AssetDependencies
    AddDependency on asset's XML file (Asset.PersistentFilename)
    AddDependency on file specifying project resource roots
    AddDependency on asset's file (e.g. dff, bsp, anm, txt, etc)
    Add asset to the sent objects list if not already in it
    AddTarget on the asset's output file (.rws on disk)
End Sub
```

If any of the three dependencies are out of date, then the asset needs to be built again. The asset's build command would be run when the build is invoked (after dependency checking is complete).

The `EntityDependencies` routine is very similar, as are most of the dependency routines. Any kind of processing operates under the same mechanisms; you just specify their dependencies and targets, which could be products of other build rules. The resulting dependency graph is then used internally by the GPM to fire off the build routines in the correct order.

Put simply, the game database is a tree of objects. The build rules reflect this, iterating through the database and checking what files need to be built (the target files) according to the dependencies given. When the build is invoked, the order has already been determined by a graph of the dependencies (composed in all the `DependencyGraph` routines). For a visual representation of the `DependencyGraph`, see Anatomy of a default build rule.

BuildCommand walkthrough

Let's put some breakpoints in the build routines to see exactly what happens when files are created on disk.

Entity files

Open the `Example Animation.rwstudio` project in the RenderWare Studio examples directory. If you have built this project before, delete the Build Output directory below Example Animation\ so when you step through the code, you can see each object's data being created.

1. Put a `stop` just after `EntityCommand` in `Entity.rule`.
2. Now build the project. When the breakpoint is hit, the Asset, Entity, and Folder directories should be empty.
3. Step until `EntityBuffer` is created with `RWSComms.CreateFileBuffer`. This call initializes the file on disk. Step once more and see that an 0 KB

.rws file has been written into the Entity folder.

4. Step until `AppendCreateEntity`. This adds the actual data to the file stream. If you are interested in what the helper routine does, step in. You'll see that first the header is written, then the attribute packets for the entity.
5. Continue stepping until you are suddenly back to `stop` again. What is going on? First, examine the callstack. `oMake.Build` calls `EntityCommand`. `oMake.Build` begins the actual build process and goes through the list of build routines generated from the dependencies phase, one after another. In this case, we happen to be building one entity immediately after another. The previous entity's `.rws` file is complete (you should see a 1 KB `.rws` file).
6. Continue (F5) and you will next stop in `FolderCommand`. (Two `.rws` files should now reside in the `Entity` folder). Since RenderWare Studio sees folders as organizational containers, no data is sent to the target platform (with the exception of root folders, as discussed later). You will notice that a 0 KB file is created in the Folder directory - this is for dependency checking i.e. so that the subtasks will run if the folder's contents have changed.
7. As you continue, you will see .rws files created for every entity being built for the target platform.

Asset files

Prepare to break into the code again by finishing the build process. When it is complete, clean the project. Remove `stop` from `Entity.rule` and place it at the start of `AssetCommand` in `Asset.rule` (and save the rule files).

1. Build the project. When the breakpoint is hit, the Asset, Entity, and Folder directories should be empty.
2. Just as with `EntityBuffer`, `AssetBuffer` is created with `RWSComms.CreateFileBuffer`. As you step you'll see the data created with AppendCreateAsset. For custom asset types, this is the phase when you might call an executable, your own script, or another COM object to operate on the data.
3. Before exiting `AssetCommand`, examine the Asset object in a watch window. It refers to the box.dff file on disk.
4. Continue stepping. The next command routine should be `EntityCommand`. By inspecting the Entity object, you can see its name is “Directors Camera”. The box asset is referenced by the Director's Camera, so is built before the entity as specified in the `EntityDependencies` routine.
5. Continue (F5). The next asset is the Map_All.bsp world. In the game database, its parent entity is “World” which does not reside in a folder (other than the root level folder). The folder rule dictates that child folders should be built before child assets or entities. In many games, a world is one of the first assets that should be sent since other assets may depend on its existence. In this case, the assets that have been built (and will be sent) first are not rendered by the entities so it is not an issue.

You should now have a picture of how individual objects are constructed through their `BuildCommand` routines.

The Glue

How do these bits and pieces get put together to form larger chunks of data? By default, the build rules package data up by levels, i.e. folders at the top-level in the tree. `RootFolder.rule` is then the place to look.

Before gluing it all together, examine RootFolderDependencies and see how it differs from FolderDependencies. In pseudo-code:

```
Sub RootFolderDependencies
    AddDependency on folder's XML file
    AddSubTask to build folder's data - pass Folder to Folder.rule
    AddSubtask for creating the texture dictionary
    AddSubtask for creating placement new file
    AddTarget for final .stream file on disk
End Sub
```

The folder simply hands itself off to the generic folder rule to handle processing of its children. The primary difference is that root folders need placement new information for all data within, as well as texture dictionary data, so these subtasks are added.

At this point, RootFolderDependencies has specified two additional subtasks that need to be completed other than just the folder information. But nothing has been glued together yet. To see this, remove all other breakpoints and put a breakpoint stop at the beginning of RootFolderCommand, then rebuilding.

1. Continue (F5) once. The first root folder is the empty global folder - not interesting so skip it.
2. Begin stepping. You'll notice that a file is created on disk for the root folder that will become our final `.stream` file.
3. Next, at `Set ObjectBuffer`, the placement new header file is read from disk. It is written into the stream file. As you step, you'll see the stream file grow in size as each data chunk is appended.
4. The texture dictionary is read off disk and appended to the stream
5. The sent objects list is loaded from disk.
6. The rule loops over the list of sent objects, loading each object's .rws file from disk and appending it to the stream. When the list has been exhausted, the stream is complete!

As you can see, the significance of `RootFolder.rule` is that it cycles through all game-ready pieces of data and simply glues them together. The ordering has been created with the knowledge of runtime requirements. E.g. our framework needs placement new data before it can create entities and needs texture dictionary data before it can create assets.

Within this high-level ordering, the order of game database objects themselves is dictated by the sent objects list. The build rules write to the sent objects list in many places. But what specifies the order of writing? The `DependencyGraph` structure created by the build rules.

As you've seen, `Folder.rule` specifies much of this structure, as it is recursive and builds assets, asset folders, child folders, and finally, child entities. Other rules enforce order on a smaller scale through their own dependency routines.

For example, `Entity.rule`, which builds its assets before itself (hence they are entered into the sent objects list first). The sent objects list prevents duplicate objects from being built and is the exact order in which your objects are glued together.

Appendix

When you look at the build rules directory, it can be difficult to identify which rules are pertinent to your interests. The following is a rough break down of the default build rules.

Note: most rules have two versions - generic offline building, and building when connected to a target platform.

Core rules:

`Game.rule`

`RootFolder.rule`

`Asset.rule`

`AssetFolder.rule`

`Entity.rule`

`Folder.rule`

Object-specific rules:

`RF3Asset.rule` - builds binary assets from RF3 (xml) files

`SequenceAsset.rule` - compiles sequence assets and handles changes during live update

`StreamEditAsset.rule` - bakes stream edit data into temporary stream files

`Font.rule` - creates font asset and texture dictionary from .met and texture file.

`VolumeEntity.rule` - creates hull data for a volume entity

Top-level processing rules:

`PlacementNew.rule` - creates a stream file containing the placement new header information. This is a list of each behavior type, and an associated count of all occurrences of each behavior within the level.

`TextureDictionary.rule` - creates texture dictionaries

`SentObjects.rule` - builds the sent objects file

`RWSIdentifiers.rule` - Replaces RWSIdentifier userdata tags in a RenderWare stream with RpIds

Connection-related rules:

`TargetLaunch.rule` - run a command on a console

`AttributeShare.rule` - handles changes to attributes shares while connected

`Attribute.rule` - handles changes to attributes while connected

Other rules:

`Workspace.rule` - contains a collection of rules, primarily used when connected to a target platform: `Connect`, `Disconnect`, `DirectorsCamera`, `CameraChange`, `SyncCamera`, `PauseGame`, `TestFireSendEvent`,

`TestFireReceiveEvent`, `ConvertProjectToVersion2`, `ParseSource`.

Creating a simple rule

In this tutorial

We show how the Game Production Manager (GPM) uses the timestamps of dependent and target files to determine whether or not to run the build command of a task.

To do this, we create three files:

- A build rule file (`.rule`) containing a single rule. This rule specifies one dependency file and one target file. When you use the GPM to run a task that uses this rule, and the dependent file is newer than the target, or the target does not exist, then the build command for the rule simply copies the dependent file to the target file.
- A VBScript file that loads the above rule into the GPM, adds a task based on this rule, and then runs the GPM.
- The dependent file (a plain text file).

Creating the build rule file

1. Create a file containing the following code and save it as `CopyFile.rule`:

```
<?xml version="1.0" ?>
<build version="1.0">
   <rule name="CopyFile" dependencygraph="Depend" buildcommand="Build">
      <object id="fso" progid="Scripting.FileSystemObject"/>
      <script language="VBScript">
      <![CDATA[
         Sub Depend (Task, strDependencyFilePath, strTargetFilePath)
            Task.AddDependency strDependencyFilePath
            Task.AddTarget strTargetFilePath
            MsgBox "Defined dependency and target files."
         End Sub

         Sub Build (Task, strDependencyFilePath, strTargetFilePath)
            fso.GetFile (strDependencyFilePath).Copy (strTargetFilePath)

            ' Set timestamp to current system time: fso copy does not do it
            TouchFile strTargetFilePath
            MsgBox "Copied " & strDependencyFilePath & _
                      " to " & strTargetFilePath
         End Sub
      ]]>
      </script>
   </rule>
</build>
```

Creating a script to invoke the Game Production Manager

2. Create a file containing the following code and save it as `CopyFile.vbs` (in the same folder as the `.rule` file):

```
Dim Make

' Create the COM object that represents the GPM
```

```
Set Make = CreateObject ("CSL.RWSBuild.Make")

' Load the build rule file into the GPM
Make.LoadRule "CopyFile.rule"

' Add the primary task to the GPM, using the CopyFile rule'
target.txt is the target file to be created by this task
Make.AddTask "CopyFile", "dependent.txt", "target.txt"

' Do the task
Make.Build
```

Creating a dependent file

Typically, a dependent file contains data that needs further processing, or must be combined with data from other dependent files before being written to a target file that can be used directly by your game. However, to keep this example simple, we're going to use a text file.

3. Create a text file (it doesn't matter what text it contains), and save it as `dependent.txt`.

Testing the rule

4. Start Windows Explorer, and browse to the folder where you have created the files for this tutorial.

   Ensure that this folder contains `CopyFile.rule`, `CopyFile.vbs`, and `dependent.txt`.

5. Run the script (by double-clicking `CopyFile.vbs` in Windows Explorer).

   The script displays a “Defined...” message box, then a “Copied...” message box, and then `target.txt` appears in the folder.

6. Run the script again.

   This time, the “Defined...” message box appears, but no “Copied...” message box. The GPM detects that the target file is up-to-date, and so does not run the build command.

7. Edit the text in `dependent.txt`, and save it.

8. Run the script one more time.

   This time, both the “Defined...” and “Copied...” message boxes appear.

9. Open `target.txt`, and see that it contains the newly edited text.

This is the end of the tutorial.

Reporting build information

In this tutorial

We show how to use the GPM events `BuildLog`, `BuildWarning`, and `BuildError` to report information during a build. This provides an alternative to using standard Windows API calls, such as the `MsgBox` function, to display a message dialog.

Introduction

`BuildLog`, `BuildWarning`, and `BuildError` are events of the GPM `RWSMake` object; they are also methods of the GPM `IRWSGlobal` interface. Calling one of these `IRWSGlobal` methods triggers the `RWSMake` event of the same name. To report information during a build, you first need to write procedures that handle these `RWSMake` events. Then, in the build scripts, you call the `IRWSGlobal` methods. This triggers the events, calling the event handlers.

Creating the build rule file

1. Create the following file, and save it as `Info.rule`:

```
<?xml version="1.0" ?>
<build version="1.0">
   <rule name="MyInfoRule"
         dependencygraph="InfoDepend" buildcommand="InfoBuild">
      <script language="VBScript">
         <![CDATA[
            Sub InfoDepend (Task, strDependencyFilePath, strTargetFilePath)
               BuildLog ("Log information message")
               Task.AddDependency strDependencyFilePath
               Task.AddTarget strTargetFilePath
            End Sub

            Sub InfoBuild (Task, strDependencyFilePath, strTargetFilePath)
               BuildWarning ("Warning message")
               BuildError ("Error message. Halts the GPM.")
            End Sub
         ]]>
      </script>
   </rule>
</build>
```

The `BuildLog` event handler is used to generate basic information about the build process. The `BuildWarning` event handler is used to provide warning information regarding some aspect of the build process. The `BuildError` event handler is used to produce an error message and also to halt the Game Production Manager (GPM), effectively stopping the build process. `BuildLog`, `BuildWarning`, and `BuildError` event handlers are implemented using VBScript or JScript in the script that invokes the GPM. As such their behaviour is always user defined.

Creating a script to invoke the Game Production Manager

2. Create a file containing the following code and save it as `Info.vbs` (in the

same folder as the `.rule` file):

```
Dim Make
Set Make = CreateObject ("CSL.RWSBuild.Make")
WScript.ConnectObject Make, "Make_"

Make.LoadRule "Info.rule"
Make.AddTask "MyInfoRule", "dependent.txt", "target.txt"
Make.Build

' Handle a system error event
Sub Make_SystemError (BuildError)
   WScript.Echo "[SYSERROR] " & BuildError.Rule & _
                "[" & BuildError.LineNumber & "] " & _
                BuildError.Message & " " & BuildError.Description
End Sub

' Handle a build error event
Sub Make_BuildError (strRule, strError)
   WScript.Echo "[ERROR] " & strError
End Sub

' Handle a build warning event
Sub Make_BuildWarning (strRule, strWarning)
   WScript.Echo "[WARNING] " & strWarning
End Sub

' Handle a build message event
Sub Make_BuildLog (strRule, strLog)
   WScript.Echo "[LOG] " & strLog
End Sub
```

Note the use of the WScript class `ConnectObject` (msdn.microsoft.com/library/en-us/script56/html/wsMthConnectObject.asp) method to connect to the `make` object's event sources. As mentioned the `BuildLog`, `BuildWarning` and `BuildError` event handlers in the above code, define how to respond to the corresponding calls from the rule file. The `SystemError` event handler will be invoked to report any unexpected errors that occur during the build process.

Testing the rule

3. Start Windows Explorer, and browse to the folder where you have created the files for this tutorial.

   Ensure that this folder contains `Info.rule` and `Info.vbs`.

4. Run the VBScript file (by double-clicking `Info.vbs`).

5. Messages will be displayed in turn for `BuildLog`, `BuildWarning` and `BuildError`).

6. To test the `Make_SystemError` handler, you must place a deliberate mistake in the rule file. To do this re-open `Info.rule` and remove the closing bracket (after the `strTargetFilePath` parameter) of the `InfoDepend` sub-routine, as shown:

   ```
   Sub InfoDepend(Task, strDependencyFilePath, strTargetFilePath
   ```

7. Re-run the VBScript file. An error will be reported.

This is the end of the tutorial.

Sharing Data

In this tutorial

We show how the Game Production Manager (GPM) can initialize and retrieve data at both a global (accessible by any script or task) and local (only accessible within the task) level.

To demonstrate this, we create two files:

- A build rule file (`.rule`) containing a single rule. This rule sets and retrieves data locally (within the task) and also retrieves data set globally.
- A VBScript file that loads the above rule into the GPM, adds a task based on this rule, and then runs the GPM. This script is also used to set data at a global scope, to be retrieved in the above rule file.

Creating the build rule file

1. Create a file containing the following code and save it as `Data.rule`:

```
<?xml version="1.0" ?>
<build version="1.0">
   <rule name="MyDataRule"
         dependencygraph="DataDepend" buildcommand="DataBuild">
      <script language="VBScript">
         <![CDATA[
            Sub DataDepend (Task, strDependencyFilePath, strTargetFilePath)
               MsgBox ("Depend stage")
               Task.SetParam "LocalData", "localMoreData"
               Task.AddDependency strDependencyFilePath
               Task.AddTarget strTargetFilePath
               MsgBox GetParam ("GlobalData")
            End Sub

            Sub DataBuild (Task, strDependencyFilePath, strTargetFilePath)
               MsgBox ("Build stage")
               MsgBox Task.GetParam ("LocalData")
            End Sub
         ]]>
      </script>
   </rule>
</build>
```

`Task.SetParam` will initialize data to be shared at local scope within the task. In the above code `LocalData` is set to have a value of `localMoreData`. `GetParam` is used in the depend rule, `DataDepend`, to retrieve data which is shared at a global scope. This will be set in the .vbs file, as shown below. `Task.GetParam` is used in the build rule, `DataBuild`, to retrieve data that has been set at local scope within the task.

Creating a script to invoke the Game Production Manager (GPM)

2. Create a file containing the following code and save it as `Data.vbs` (in the same folder as the `.rule` file):

```
Dim Make
Set Make = CreateObject ("CSL.RWSBuild.Make")

Make.SetParam "GlobalData", "globalSomeData"
Make.LoadRule "Data.rule"
Make.AddTask "MyDataRule", "dependent.txt", "target.txt"
Make.Build
```

In the above code `Make.SetParam` initializes `GlobalData` to a value of `globalSomeData`. This data will have global scope.

Testing the rule

3. Start Windows Explorer, and browse to the folder where you have created the files for this tutorial.

   Ensure that this folder contains `Data.rule` and `Data.vbs`.

4. Run the VBScript by double-clicking on the `Data.vbs` file.

This is the end of the tutorial.

Cross Dependencies

In this tutorial

We show how a source file can be made to have multiple dependent files.

To do this, we create a number of files and a new folder:

- A build rule file (`.rule`) containing a single rule. The main purpose of the rule file is to create a source file and establish a number of dependencies to that file. The rule then writes information about each dependency into the source file.
- A VBScript file that loads the above rule into the GPM, adds a task based on this rule, and then runs the GPM.
- A new folder containing a number of files.

Creating the build rule file

1. Create a file containing the following code and save it as `Cross_Depend.rule`:

```
<?xml version="1.0" ?>
<build version="1.0">
   <rule name="MyCrossDepend"
         dependencygraph="CrossDepend" buildcommand="CrossBuild">
      <script language="VBScript">
      <![CDATA[
         Sub CrossDepend (Task, FileName, SourceFolder, target)
            MsgBox "Depend stage start"

            Set FileSystem = CreateObject("Scripting.FileSystemObject")
            Set Folder = FileSystem.GetFolder (SourceFolder)
            Set f = FileSystem.CreateTextFile (FileName, True)

            For Each File In Folder.Files
               AddCrossDependency FileName, File.Path
               f.WriteLine File.Path
            Next

            Task.AddDependency FileName
            Task.AddTarget target
         End Sub

         Sub CrossBuild (Task, FileName, SourceFolder, target)
            MsgBox "Builds stage start"
            Set CrossDepends = GetCrossDependencies (FileName)
            MsgBox CrossDepends.count

            For Each Item In CrossDepends
               MsgBox Item
            Next
         End Sub
      ]]>
      </script>
   </rule>
</build>
```

Referring to the `CrossDepend` subroutine above, we can see it first creates a `FileSystem` object, using the `CreateObject` method call of the `File` class. It then creates a folder object using the name specified by the `Source Folder` parameter.

A text file is then created using the `CreateTextFile` method call. In the build stage of the process we will write a list of dependencies into this file. A “for loop” is then set up. This loop iterates for each file found in the folder specified by the `SourceFolder` parameter. Each time around the loop `AddCrossDependency` is used to create a dependency between the file specified by the `FileName` parameter and a file found in the source folder.

In the `CrossBuild` subroutine of the above code, `GetCrossDependencies` is used to return a collection of dependencies of the file specified by `FileName`. The collection obtained has the built-in properties `Count` and `Item`, which return the number of items currently in the collection and a specific item (at a given index) respectively. Another “for loop” is then used to display the name of each dependency in the collection.

Creating a script to invoke the Game Production Manager

2. Create a file containing the following code and save it as `Cross_Depend.vbs` (in the same folder as the `.rule` file):

```
Dim Make
Set Make = CreateObject ("CSL.RWSBuild.Make")

Make.LoadRule "Cross_Depend.rule"
Make.AddTask "MyCrossDepend", "CrossDepends_List.txt", _
             "Dependent_Files\", "target.txt"
Make.Build
```

`AddTask` is called with 3 extra parameters, a file (to be created), a directory to be used and a dummy target file.

Create Data directory

3. In Windows Explorer browse to the location of the files created above.
4. Create a folder named `Dependent_Files`
5. Copy or create 5 or more files in the new directory. The names and contents of the files created are not significant for the purposes of this tutorial.

Testing the rule

6. Start Windows Explorer, and browse to the folder where you have created the files for this tutorial.

   Ensure that this folder contains `Cross_Depend.rule` and `Cross_Depend.vbs`.
7. Run the VBScript file (by double-clicking `Cross_Depend.vbs`).
8. The script will run and report the number of dependencies of the source file and also print the file name of each dependency.
9. Open the file `CrossDepends_List.txt`. Verify that the file correctly lists the names (including the path) of each dependency.

This is the end of the tutorial.

The GPM and Workspace

Before reading this topic

You should know a little about the nature of Workspace, and its relationship with Enterprise Author. In particular, you should be familiar with thinking of Workspace as a collection of objects that interact through script code.

In Workspace, the interface between the games you create and the Game Production Manager is provided by the *BuildScriptManager* script module. This object contains variables and methods for interacting with the GPM and managing the process of building game-ready data.

Build operations are invoked from the Targets window's context menu, or the **Target** menu, or the *CurrentTarget* toolbar. To see how *BuildScriptManager* is used, we'll take the first of those, so open the *TargetContextMenu* object in Enterprise Author. Around halfway down the script module, you'll find a list of handlers for menu items, which starts like this:

```
Sub OnClean()
   BuildScriptManager.BuildGame True, False, _
                                    False, False, strSelectedTarget
End Sub

Sub OnBuild()
   BuildScriptManager.BuildGame False, True, _
                                    False, False, strSelectedTarget
End Sub

...
```

All of these handlers call `BuildScriptManager.BuildGame()`, but with different argument lists that reflect different purposes. In *BuildScriptManager*, the `BuildGame()` method takes five parameters (four Booleans and a string), as follows:

```
Sub BuildGame(bClean, bBuild, bConnect, bBuildAllFolders, strTarget)
```

`bClean` specifies whether target files should be cleaned.

`bBuild` specifies whether out-of-date target files should be built.

`bConnect` specifies whether Workspace should connect to the console.

`bBuildAllFolders` specifies whether all folders should be built, or just the active folder.

`strTarget` specifies the target console for which the chosen operation should occur.

From this list, it follows that when (for example) `OnBuild()` calls `BuildGame()`, only the active folder is built, the target files are not cleaned, and Workspace does not attempt to connect to the console.

The `BuildGame()` method

In the code for `BuildGame()`, after some information about the selected target

console has been extracted into local variables, this happens:

```
Set Make = CreateObject("CSL.RWSBuild.Make")
GlobalScript.RWSMakeProxy.ConnectMake Make
SetupMake Make, strName, strResourceRoot, _
                          strRF3Template, bEmbedAssets, True
```

This first creates an `RWSMake` object, which represents the GPM. It then "connects" the object so that it fires its events to *GlobalScript*'s `RWSMakeProxy` object, with the result that `RWSMake`'s events can be handled in *GlobalScript*. Lastly, the code calls `SetupMake()`, which does two things:

- It provides several parameters to the `RWSMake` object that give it details of the operation at hand.
- It loads the build rules for the project by calling `LoadRules()`.

In fact, `SetupMake()` calls `LoadRules()` *twice*. This is because the script first attempts to load any custom build rules, from the project's `Build Rules` directory. After that, the default build rules are loaded as necessary. (In other words, if there are no custom rules, then all the default rules will be loaded.)

Back in `BuildGame()`, the next important piece of code is:

```
If LaunchBuildRule(Make, strCurrentTarget, strName, strPlatform, _
      strResourceRoot, bEmbedAssets, bClean, bBuild, bBuildAllFolders) Then
```

This calls `LaunchBuildRule()`, passing it information about the target console, and four Boolean values.

The `LaunchBuildRule()` method

`LaunchBuildRule()` cleans and/or builds the target files based on the Boolean parameters passed, using the build rules that have already been loaded. It starts like this:

```
Set Connection = GlobalScript.RWSComms.CreateNullConnection( _
                                strPlatform, strResourceRoot, bEmbedAssets)
oMake.SetParam "BIGENDIAN",     Connection.BigEndian
oMake.SetParam "UNICODE",       Connection.Unicode
oMake.SetParam "PLATFORM",      Connection.Platform
oMake.SetParam "PLATFORMFLAGS", _
                GlobalScript.RWSComms.GetPlatformFlags(Connection.Platform)
```

A null connection is created so that we can output files for the specified platform to disk, without having actually to connect to the target. After this, some further parameters for the `RWSMake` object are set, specifying the connection's platform, and whether the platform is Unicode, big-endian, etc.

Next, after some code to save the project (and handle any problems with that process), the real work begins:

```
If Len(BSMScript.Game.BuildCommand) > 0 Then
   ' There's a custom game build rule, so use it.
   strBuildRule    = BSMScript.Game.BuildCommand
   strStartMessage = "using the custom game rule '" & strBuildRule & "'"
Else
   ' There isn't a custom game build rule, use one of the defaults.
   If bBuildAllFolders Then
      strBuildRule    = "BuildAllFolders"
      strStartMessage = "all folders"
   Else
      strBuildRule    = "BuildGame"
      strStartMessage = "the active and global folders"
```

```
   End If
End If
strStartMessage = strStartMessage & " for the " & _
                  strConnectionName & " target..."

If bClean Then
   CSLRenderwareStudioWorkspace.ShowObject BuildLog
   BuildLog.Log "[Target] Cleaning " & strStartMessage
   oMake.SetParam "Cleaning", True
   oMake.AddTask strBuildRule, BSMScript.Game, PlatformFilter
   Broadcast.PreCleanProject strConnectionID, strConnectionName
   If Not oMake.Clean() Then
      LaunchBuildRule = False
      oMake.SetParam "Cleaning", False
   End If
   oMake.SetParam "Cleaning", False
   Broadcast.PostCleanProject strConnectionID, _
                              strConnectionName, LaunchBuildRule
End If
```

This code begins by setting the build rule. If no custom build rule has been set (so `BSMScript.Game.BuildCommand` is a zero-length string), then it defaults to either `BuildAllFolders` or `BuildGame`, depending on the value of `bBuildAllFolders`.

If `bClean` is `True`, the code then instructs Workspace to make the Build Log window visible. The **[Target] Cleaning** message is then added to the window, and a task is added to the `RWSMake` object. Finally, `Clean()` is called to clean all the target files of that task.

Next, provided that things have gone smoothly so far, and the `bBuild` parameter was `True`, a file system object is created (so that we can write the target files to disk), and `LaunchBuildRule()` executes code similar to that for the `bClean` parameter:

```
   Set fso = CreateObject("Scripting.FileSystemObject")
   CSLRenderwareStudioWorkspace.ShowObject BuildLog
   BuildLog.Log "[Target] Building " & strStartMessage
   strPath = oMake.GetParam("DESTDIR")
   strPath = fso.BuildPath(strPath, "Misc")

   ' Create misc folder if it doesn't exist
   If Not fso.FolderExists(strPath) Then
      GlobalScript.CreateDirectory strPath, fso
   End If

   ' Write out the global files (if necessary) that all assets/entities
   ' depend on. This determines whether the workspace resource root,
   ' connection resource root or "embed assets" flags etc. have been
   ' changed since the last build.
   UpdateEntitySettingsFile fso.BuildPath(strPath, "entityglobals.txt"), _
                                                  GlobalScript.g_strSourceRoot
   UpdateAssetSettingsFile fso.BuildPath(strPath, "assetglobals.txt"), _
                           oMake.GetParam("TGTRESOURCEROOT"), _
                           oMake.GetParam("WKSRESOURCEROOT"), bEmbedAssets

   oMake.SetParam "Cleaning", False
   oMake.AddTask strBuildRule, BSMScript.Game, PlatformFilter

   ...

   If Not oMake.Build() Then
      LaunchBuildRule = False
   End If
```

As in the code for cleaning target files, the Build Log window is set to be visible, and the appropriate message is added to it. A destination path is then set and, if necessary, created. Following this, a pair of subroutines are called: `UpdateEntitySettingsFile()` and `UpdateAssetSettingsFile()`. These rebuild `entityglobals.txt` and `assetglobals.txt` respectively (if they don't exist, or are out of date), which contain the source root and resource root paths for entity behaviors and asset files. Finally, the task is added to the `RWSMake` object, and the `Build()` function is called to output the target files.

Workspace's default build rules

RenderWare Studio is supplied with a set of default build rules for building your games. These are stored in the `C:\RW\Studio\Programs\Workspace\Build Rules` directory, and loaded when *BuildScriptManager*'s `BuildGame()` subroutine is called (provided that no custom rules are supplied to override the default rules).

You saw in *BuildScriptManager* that if only the active folder is being built, then the default entry point build rule is `BuildGame`. If you open the `Game.rule` file, you can see that `BuildGame` is the first build rule defined. It contains two subroutines: `BuildGameDependencies()`, and `BuildGameCommand()`.

`BuildGameDependencies()` builds the dependency graph of target and dependency files for the build task. (A task is an instance of a build rule.)

`BuildGameCommand()` is the build command for this rule, which builds out of date or non-existent target files.

When the `BuildGame` rule is invoked, `BuildGameDependencies()` is called to build the dependency graph of the build task. Once the dependency graph has been built, the GPM compares the timestamps of target and dependency files. If any target files are older than their dependency files (or if they don't exist), then the build command, `BuildGameCommand()`, is called. However, since only *subtasks* of `BuildGame` will have any target files, `BuildGameCommand()` is actually an empty subroutine.

`BuildGameDependencies()` looks like this:

```
Sub BuildGameDependencies(Task, Game, PlatformFilter)
   Dim Folder, GlobalFolder

   ' Recursion guard prevents objects being recursed into multiple times
   RecursionGuard.Clear

   If Game.ActiveFolder Is Nothing Then
      BuildError "Unable to determine active folder."
   Else
      ' Build the global folder first
      Set GlobalFolder = Game.GlobalFolder
      If Not GlobalFolder Is Nothing Then
         If PlatformFilter.IsValidPlatform(GlobalFolder.Platform) Then
            Task.AddSubtask "RootFolder", GlobalFolder, _
                              PlatformFilter, RecursionGuard
         End If
      End If

      ' Build the active folder
      If PlatformFilter.IsValidPlatform(Game.ActiveFolder.Platform) Then
         If Game.GlobalFolder.UID <> Game.ActiveFolder.UID Then
            Task.AddSubtask "RootFolder", Game.ActiveFolder, _
                              PlatformFilter, RecursionGuard
         End If
      End If
   End If
End Sub
```

You can see that although no targets or dependencies are added, two subtasks are. Both subtasks are instances of the `RootFolder` build rule: the first builds

the global folder, while the second builds the active folder (unless the global and active folder are the same, in which case only one subtask is added).

If you now open `RootFolder.rule`, you can look at the `RootFolder` build rule. The dependency graph subroutine for this rule is `RootFolderDependencies()`. You'll notice that, unlike `BuildGameDependencies()`, this subroutine *does* add both target and dependency files to the dependency graph. If any of these target files are older than their dependencies, then the build command (`RootFolderCommand()`) will be called to rebuild them after the dependecy graph has been completed. In addition, `RootFolderDependencies()` adds further subtasks to our build task (the `Folder` build rule, for example). These subtasks will in turn add further subtasks, targets and dependencies until the dependency graph for the entire project has been built. At this point, all out-of-date or non-existent target files will be built by the build commands, leaving us with up-to-date target files.

Anatomy of a default build rule

Every build rule contains two subroutines:

- A dependency graph subroutine, to generate a graph of target and dependency files, and a list of subtasks
- A build command subroutine, to build out-of-date or non-existent target files

When a build rule is invoked, the appropriate subroutine generates a dependency graph. Using this graph, the timestamps of target files are compared with those of their dependency files. If any target files are out-of-date, the build command subroutine is called to rebuild them.

We're now going to expand on this subject by looking in more detail at how dependency graphs are generated, and at what actually happens when target files are built.

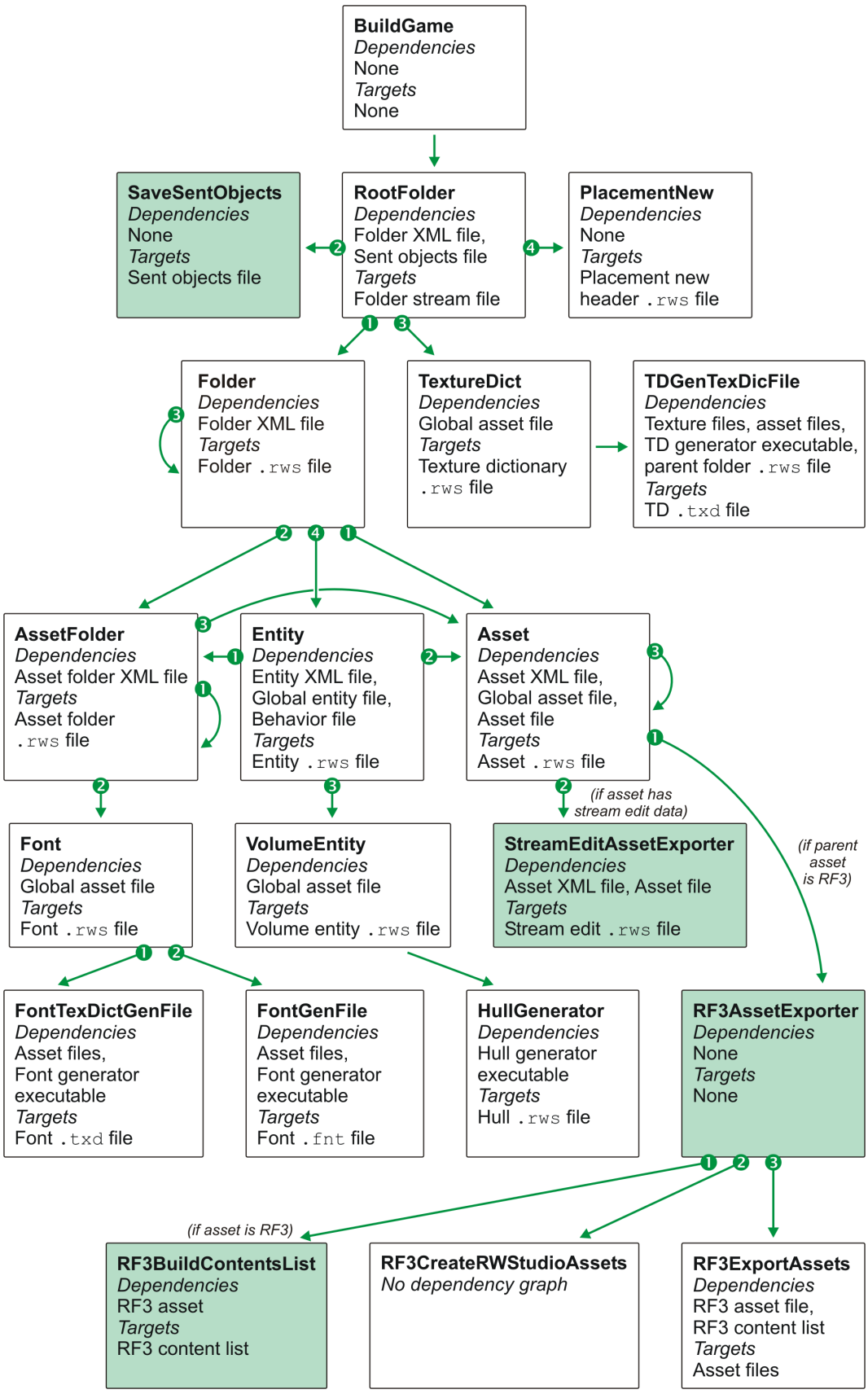
Dependency graphs

In general, the dependency graph subroutine of a build rule performs three operations:

- It adds dependency files to the dependency graph.
- It adds target files to the dependency graph.
- It adds further build rules as subtasks.

In practice, a dependency graph subroutine will be more complex than this, but these three operations are the keys to building the dependency graph of an entire project. We can illustrate how this is the case by looking again at Workspace's default build rule, `BuildGame`.

The following diagram depicts how further rules may be added as subtasks to `BuildGame` in order to produce a complete graph of target and dependency files. Note that the targets of shaded subtasks are built immediately, before any further rules may be invoked.



`BuildGame` is the entry point for creation of the dependency graph. `BuildGame` adds no targets or dependencies, but it does invoke the `RootFolder` rule (usually twice—once for the global folder, and once for the active folder). `RootFolder` is responsible for building the `.stream` file that gets sent to the

console upon connection, so it adds this `.stream` file to the dependency graph as a target.

The `.stream` file is dependent upon the XML file of the folder passed to the rule (which contains basic information about the folder, such as its name and object references), and the sent objects file (a list of assets, entities, etc. that will be contained in the `.stream` file), so these files are added to the graph as dependencies. `RootFolder` then goes on to invoke the following build rules, in order:

1. `Folder` builds `.rws` files, which are the stream files for individual entities, folders, assets, etc. This build rule is recursive, and may call itself for child folders.
2. `SaveSentObjects` builds the sent objects file. This build rule is invoked as a sub-make object, whose target files will be built immediately (that is, construction of the dependency graph will halt while the target files are built).
3. `TextureDict` builds the texture dictionary `.rws` file for the root folder. This is a stream file containing the `.txd` texture dictionary for the folder.
4. `PlacementNew` builds the placement new header `.rws` file, which contains a list of each behavior type and an associated count of all occurrences of each behavior within the level.

Upon completion of the dependency graph, target files can be compared against dependency files in order to determine which target files need to be (re)built. But what *actually* happens when target files are built?

Build commands

To see what happens during a build command, we're going to look at the `RootFolder` build command subroutine, `RootFolderCommand()`, which must concatenate the object `.rws` files, the placement new header file, and the texture dictionary file into the `.stream` file. The sent objects file is *not* concatenated into the stream. Instead, it is used to determine which of the object `.rws` files should be written to the `.stream` file.

```
Sub RootFolderCommand (Task, Folder, PlatformFilter, RecursionGuard)
   BuildLog "Building " & Folder.Name
   Dim DestDirectory, RootFolderBuffer, APIObject, _
                        ObjectBuffer, TextureDictionaryBuffer

   ' Root folder is a concatenation of all asset/entities and child folders
   DestDirectory = GetParam("DESTDIR")
   Set RootFolderBuffer = RWSComms.CreateFileBuffer( _
                          RootFolderFile(DestDirectory, Folder), faCreate)

   If Not RootFolderBuffer Is Nothing Then
      RootFolderBuffer.BigEndian = GetParam("BIGENDIAN")
      RootFolderBuffer.Unicode   = GetParam("UNICODE")

      ' Attach the placement new header to the root folder stream
      Set ObjectBuffer = RWSComms.CreateFileBuffer( _
                         PNHeaderFile(GetParam("DESTDIR"), Folder), faRead)
      If Not ObjectBuffer Is Nothing Then
         RootFolderBuffer.WriteBuffer ObjectBuffer, AddAsNewHeader
      End If

      ' Attach the texture dictionary file if it exists
      Set TextureDictionaryBuffer = RWSComms.CreateFileBuffer( _
                            Task.GetParam("TEXTUREDICTIONARYFILE"), faRead)
      If Not TextureDictionaryBuffer Is Nothing Then
```

```
         RootFolderBuffer.WriteBuffer _
                              TextureDictionaryBuffer, AddAsNewHeader
      Else
         BuildLog "No texture dictionary available"
      End If

      ' Ensure we've loaded the list of sent objects this folder requires
      Dim SubMake
      Set SubMake = CloneMake()
      If Not SubMake Is Nothing Then
         GetParam("SENTOBJECTS").Clear
         SubMake.AddTask "LoadSentObjects", Folder

         If GetParam ("Cleaning") Then
            SubMake.Clean
         Else
            SubMake.Build
         End If
      Else
         BuildError "Could not create make object to load sent objects"
      End If

      ' Concatenate all objects in the sent objects container together
      For Each APIObject In GetParam("SENTOBJECTS")
         Set ObjectBuffer = RWSComms.CreateFileBuffer( _
                               StreamFile(DestDirectory, APIObject), faRead)
         If Not ObjectBuffer Is Nothing Then
            RootFolderBuffer.WriteBuffer ObjectBuffer, AddAsNewHeader
         End If
      Next
   Else
      BuildError "Could not create root folder stream " & _
                                       RootFolderFile(DestDirectory, Folder)
   End If
End Sub
```

The build command first attempts to create a file buffer for the target file:

```
Set RootFolderBuffer = RWSComms.CreateFileBuffer( _
                       RootFolderFile(DestDirectory, Folder), faCreate)
```

This creates an empty buffer to which the `.stream` file data will be written. If this file buffer is created successfully, then the first thing to write to it is the header data contained in the placement new header file. This requires that the placement new header file be read into its own buffer, and then be written from this buffer to the `.stream` file buffer:

```
Set ObjectBuffer = RWSComms.CreateFileBuffer( _
                   PNHeaderFile(GetParam("DESTDIR"), Folder), faRead)
If Not ObjectBuffer Is Nothing Then
   RootFolderBuffer.WriteBuffer ObjectBuffer, AddAsNewHeader
End If
```

The texture dictionary `.rws` file is then written to the `.stream` file buffer in the same way as the header file. Following this, the sent objects file must be loaded, so that when we write the object `.rws` files (the stream files for individual entities, folders, etc.) to the stream buffer, we only include those objects that are actually required. (In a PC build, for example, the PS2 controller entity could be omitted from the `.stream` file.) Here's the code that loads the sent objects file:

```
Dim SubMake
Set SubMake = CloneMake()
If Not SubMake Is Nothing Then
   GetParam("SENTOBJECTS").Clear
   SubMake.AddTask "LoadSentObjects", Folder
```

```
   If GetParam("Cleaning") Then
      SubMake.Clean
   Else
      SubMake.Build
   End If
Else
   BuildError "Could not create make object to load sent objects"
End If
```

You can see that the sent objects file is loaded by a separate build rule, `LoadSentObjects`. This rule is added as a task to a sub-make object (which is a clone of the main make object). `LoadSentObjects` has no dependency graph subroutine, so when `Build()` is called for the sub-make object, the build command for `LoadSentObjects` will be called. This build command simply adds all the objects in the sent objects file to a global container (retrieved by `GetParam("SENTOBJECTS")`).

With the sent objects file loaded, the necessary `.rws` files are written to the stream buffer:

```
For Each APIObject In GetParam("SENTOBJECTS")
   Set ObjectBuffer = RWSComms.CreateFileBuffer( _
                        StreamFile(DestDirectory, APIObject), faRead)
   If Not ObjectBuffer Is Nothing Then
      RootFolderBuffer.WriteBuffer ObjectBuffer, AddAsNewHeader
   End If
Next
```

This code loops through all the objects listed in the sent objects container that was populated by `LoadSentObjects`. For each object listed, it reads in that object's `.rws` file, and then writes it to the stream buffer. When all objects in the container have been written to the stream buffer, our target `.stream` file is complete!

Tutorial: Overriding the default build rules

In this tutorial

You'll see how to set up a project so that it overrides the default build rule (`BuildGame`). The new rule will cause additional data to be output to the Build Log window during the build process.

For the purposes of this exercise, we're going to use the project created in the “First Steps” tutorial. (You're free to use one of your own projects, should you wish.)

Creating the build rule file

1. Using Windows Explorer, navigate to the `C:\RW\Studio\Examples\Tutorial\Build Rules` folder. This should currently be empty, because none of the default rules is being overridden at the moment.

   To override the `BuildGame` rule, we must create a new rule of the same name within the `Build Rules` directory of our project. (This is not the same directory as the one containing the default rules.)

2. Create the following file using a text editor such as Notepad (or, preferably, a XML editor), and save it as `NewGame.rule`:

```
<?xml version="1.0" ?>
<build version = "1.0">
   <rule name="BuildGame" dependencygraph="BuildGameDependencies"
         buildcommand="BuildGameCommand" alwaysbuild="true">
      <object id="RWSComms" progid="CSL.RWSTarget.RWSComms"/>
      <object id="RecursionGuard" progid="CSL.RWSScript.RWSContainer"/>
      <reference object="CSL.RWSTarget.RWSComms"/>
      <reference object="CSL.RWSScript.RWSScript"/>
      <script language="VBScript"
        src="C:\RW\Studio\Programs\Workspace\Build Rules\BufferTools.vbs"/>
      <script language="VBScript"
        src="C:\RW\Studio\Programs\Workspace\Build Rules\BuildTools.vbs"/>
      <script language="VBScript"><![CDATA[
         '---------------------------------------------------------------
         ' Game.rule - BuildGame rule
         '---------------------------------------------------------------

         Option Explicit ' All variables must be explicitly declared

         '---------------------------------------------------------------
         ' To build game, we only depend on the active folder being built,
         ' so add the active folder's build command as the dependency.
         Sub BuildGameDependencies(Task, Game, PlatformFilter)

            BuildLog "Calling BuildGameDependencies for " & Game.Name

            Dim Folder, GlobalFolder

            ' Recursion guard stops objects being recursed multiple times
            RecursionGuard.Clear

            If Game.ActiveFolder Is Nothing Then
```

```
            BuildError "Unable to determine active folder."
         Else
            ' Build the global folder first
            Set GlobalFolder = Game.GlobalFolder
            If Not GlobalFolder Is Nothing Then
               If PlatformFilter.IsValidPlatform( _
                                 GlobalFolder.Platform) Then
                  BuildLog "Adding subtask RootFolder for folder " & _
                                                      GlobalFolder.Name
                  Task.AddSubtask "RootFolder", GlobalFolder, _
                                       PlatformFilter, RecursionGuard

               End If
            End If

            ' Build the active folder
            If PlatformFilter.IsValidPlatform( _
                              Game.ActiveFolder.Platform) Then
               If Game.GlobalFolder.UID <> Game.ActiveFolder.UID Then
                  BuildLog "Adding subtask RootFolder for folder " & _
                                                 Game.ActiveFolder.Name
                  Task.AddSubtask "RootFolder", Game.ActiveFolder, _
                                       PlatformFilter, RecursionGuard
               End If
            End If
         End If
         BuildLog "BuildGameDependencies has completed for " & Game.Name
      End Sub

      '-------------------------------------------------------------------
      Sub BuildGameCommand (Task, Game, PlatformFilter)
         BuildLog "Calling BuildGameCommand for " & Game.Name
         Beep ()
         BuildLog "BuildGameCommand has completed for " & Game.Name
      End Sub
   ]]></script>
 </rule>
</build>
```

This build rule is *almost* identical to the default `BuildGame` rule that it overrides, with just a few changes. For example, it references two VBScript files, `BufferTools.vbs` and `BuildTools.vbs`. Since these are no longer in the same directory as our rule, their source paths have been changed accordingly:

```
<script language="VBScript"
        src="C:\RW\Studio\Programs\Workspace\Build Rules\BufferTools.vbs"/>
<script language="VBScript"
        src="C:\RW\Studio\Programs\Workspace\Build Rules\BuildTools.vbs"/>
```

(An alternative to changing the source paths would be simply to copy the referenced scripts from the default rules directory to the directory where we created our custom build rule.)

In addition to the source path changes, there are a number of additional logging messages within our new `BuildGame` rule. These will output messages to the Build Log window when:

- `BuildGameDependencies()` begins and completes
- `BuildGameCommand()` begins and completes
- A subtask is added that uses the `RootFolder` build rule

Testing the rule

3. Start Workspace, and open the *Tutorial* project. In the Targets window, right-click on any one of the targets (it doesn't matter which, since we won't be connecting to it) and select **Clean**.

   In the Build Log window, you'll see something similar to the following:

   ```
   [Target] Cleaning...
   [BuildGame] Calling BuildGameDependencies for Game
   [BuildGame] Adding subtask RootFolder for folder Global Folder
   [BuildGame] Adding subtask RootFolder for folder First Steps
   [BuildGame] BuildGameDependencies has completed for Game
   [Target] Build process took 0 minutes and 1 seconds.
   ```

   The additional messages we've added tell us that `BuildGameDependencies()` was called for our game. The `RootFolder` rule was then added as a subtask for both the global and active folders. Finally, `BuildGameDependencies()` terminated. You'll notice that `BuildGameCommand()` is *not* called during the cleaning process, because we're not building any target files. `BuildGameCommand()` is only called when we are building target files (by selecting **Build** or **Rebuild** from the Targets window's context menu).

Tutorial: Adding a custom rule to an entity

In this tutorial

You'll see how to assign a custom build rule to a specific entity. To do this, you're going to create a new build rule called `CustomEntity` that performs two tasks:

- It will output some entity data to a text file.
- It will add the default `Entity` build rule as a subtask, so that the entity is subsequently built as normal.

For the purposes of this exercise, we're going to use the project created in the “First Steps” tutorial. (You're free to use one of your own projects, should you wish.)

Creating the build rule file

1. Create the following file, and save it as `CustomEntity.rule` in the `C:\RW\Studio\Examples\Tutorial\Build Rules` directory:

```
<?xml version="1.0" ?>
<build version = "1.0">
   <rule name="CustomEntity" dependencygraph="CustomEntityDependencies"
                             buildcommand="CustomEntityCommand">
      <object id="FileSystem" progid="Scripting.FileSystemObject"/>
      <script language="VBScript"><![CDATA[
         '---------------------------------------------------------------------
         ' CustomEntity.rule - CustomEntity rule
         '
         ' Outputs entity data to a text file
         ' and then invokes the default Entity rule
         '---------------------------------------------------------------------

         Option Explicit ' All variables must be explicitly declared

         '---------------------------------------------------------------------
         Sub CustomEntityDependencies(Task, Entity, _
                                      PlatformFilter, RecursionGuard)
            BuildLog "Building CustomEntity dependency graph for " & _
                                                            Entity.Name
            Task.AddDependency Entity.PersistentFilename
            Task.AddSubtask "Entity", Entity, _
                                       PlatformFilter, RecursionGuard
            Task.AddTarget GetParam("DESTDIR") & Entity.Name & ".txt"
         End Sub

         '---------------------------------------------------------------------
         Sub CustomEntityCommand(Task, Entity, _
                                      PlatformFilter, RecursionGuard)
            BuildLog "Outputing CustomEntity text file for " & Entity.Name
            Dim EntityFile
            Set EntityFile = FileSystem.CreateTextFile( _
                        GetParam("DESTDIR") & Entity.Name & ".txt", True)
            EntityFile.WriteLine("** Entity file for " & Entity.Name & _
                                 " entity, created on " & Date() & _
                                 " at " & Time() & " **")
            EntityFile.WriteLine("")
```

```
            EntityFile.WriteLine("Build Rule Used: " & Entity.BuildCommand)
            EntityFile.WriteLine("Entity's UID: " & Entity.UID)
            EntityFile.WriteLine("Entity's Behavior: " & Entity.Behavior)
            EntityFile.Close
         End Sub
      ]]></script>
   </rule>
</build>
```

This is the build rule file for our custom rule, `CustomEntity`. The dependency graph subroutine, `CustomEntityDependencies()`, performs three main operations:

- It adds a new dependency, pointing to the entity's `.xml` file. When this file has changed, any targets of the new rule must be rebuilt.
- It adds the default build rule, `Entity`, to the subtasks, so that the entity's standard target files get built as normal.
- It adds a new target file: a text file to which we will send some entity data during the build process.

The build *command* subroutine, `CustomEntityCommand()`, uses the file system object to create and write to the target text file.

Testing the rule

2. Start RenderWare Studio Workspace and open the *Tutorial* project.
3. In Game Explorer, locate the *World* entity, right-click, and select **Properties**.

   This displays the Entity Properties dialog, which contains two tabs: General and Build.
4. Select the Build tab, and you'll see a single edit box labeled **Build Rule**. This dialog allows you to set custom build rules for individual entities.
5. Enter the name of our custom build rule, `CustomEntity`, into the edit box, and click **OK**.
6. In the Targets window, right-click **Local PC - DirectX** and select **Build**.

   This will build the project, using our custom build rule for the *World* entity.
7. Navigate to the `C:\RW\Studio\Examples\Tutorial\Build Output\Local PC - DirectX` directory, where you'll see a file called `World.txt`. This is the target file created by our build rule, `CustomEntity`.

If you make any modifications to the *World* entity and rebuild the project, you should see that the timestamp on `World.txt` changes. This is because altering the *World* entity updates its XML file, which is the dependent file for our custom rule. Since the dependent file is now newer than the target file, the target file gets rebuilt when you invoke the build process.

Tagging Reference Objects

The purpose of this tutorial:

To illustrate how easily the GPM can be customized. Some preliminary setup is required to tag objects in the game database, so the basics of using Enterprise Author are also covered. Once tags have been applied, a simple rule can be written that uses them in a sensible way.

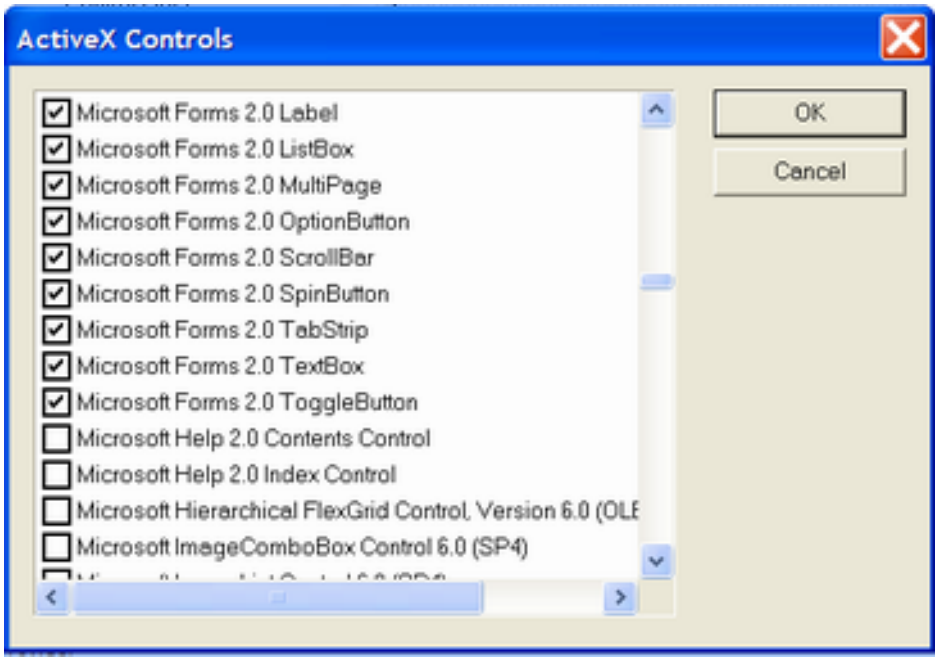
In this tutorial, objects will be tagged as having a reference to another object. The assumption is that the referenced object data is required for the referrer object to run correctly on the target platform. This reference will be checked on build by a new rule, and if necessary, enforced by building in the referenced object before the referrer object.

Creating an Interface with Enterprise Author

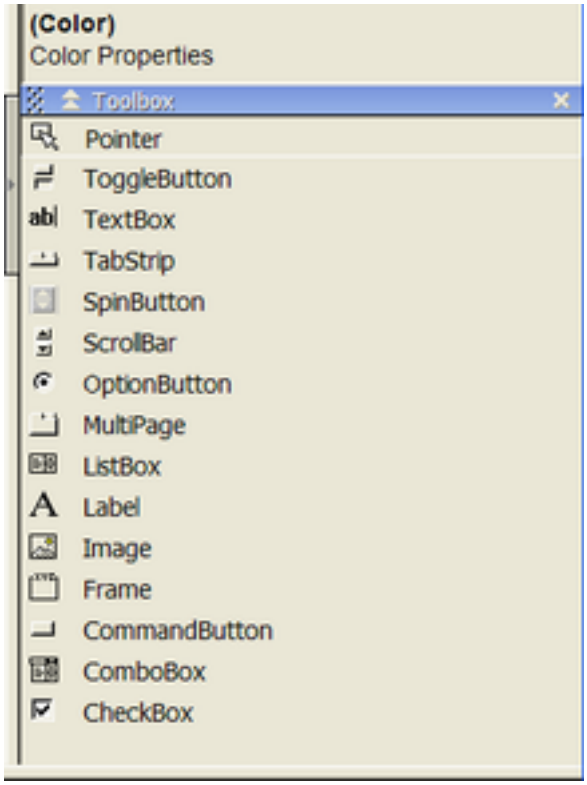
We need a simple interface for attaching tags to objects in the database. Create labels for displaying information, a text box to enter object names, and buttons to set and clear tags.

1. Open Enterprise Author
2. Open RenderWareStudio.ren
3. Save As... to a new .ren file e.g. RefTagging.ren
4. If the Toolbox is empty, right click and select Add/Remove Components, checking all Microsoft 2.0 Forms.

   a.



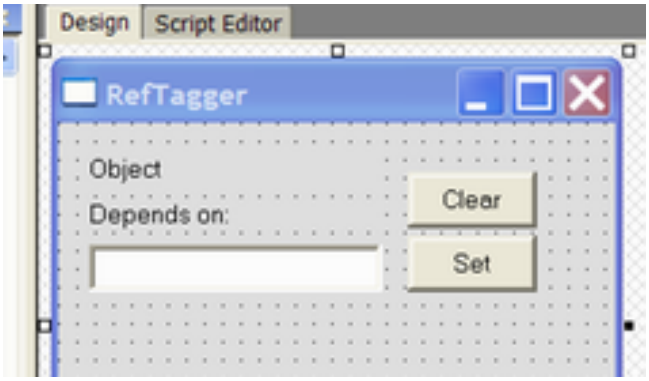
   b.



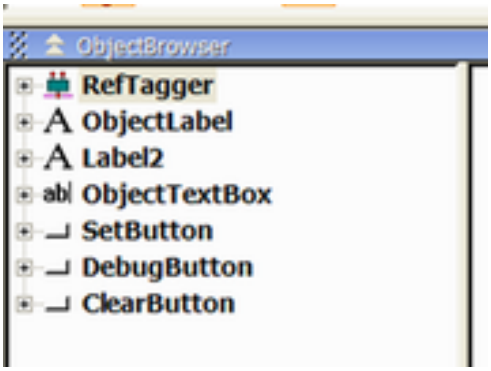
5. Select Create > Forms Page on the menu bar
6. Browse the Objects pane for the new REN page (e.g. RENPage13) and rename it to RefTagger. Set any desired properties in the PropertyList, such as the caption.

Now an empty “page” is ready to be populated with forms.

1. Click the **Label** form in the Toolbox. Click and drag inside the page to create a label area for the Object name.
2. Repeat Step 7, creating another Label, two CommandButtons, and a TextBox.



3. Give the UI elements meaningful names by clicking on each one, then right clicking near the border and selecting **Rename**. In the ObjectBrowser,



Now we can add this page to a RenderWare Studio layout.

4. Double-click CSLRenderwareStudioWorkspace and Drag RefTagger from the Object pane into the layout e.g. stack it above the Game Explorer.
5. Save the project
6. Run your custom setup by creating a shortcut on your desktop with the target set to C:\RW\Studio\Programs\EnterpriseHost.exe /host C:\projects\RwStudio\RENS\RefTagging.ren (directed to your .ren file).
7. You should now see your custom interface in Studio. Adjust the layout as you like and exit.

   Next, add some script placeholders for your form events.
8. Double-click RefTagger and select the Script tab to begin editing script.
9. Add the following, assuming you followed my naming convention:

```
Sub SetButton_Click ()
  End Sub

  Sub ClearButton_Click ()
      ObjectTextBox.Text = ""
  End Sub
```

Finally, add script placeholders to pick up selection events in Studio. This allows us to operate on objects that have been selected in Studio.

10. Create and connect to a selection object. The string “RefSelection” is arbitrary but will be the name prefix for the routines to catch selection events. For more information, refer to *Selecting game entities* in the User Guide.

```
' attach to selection object so can receive events
  Dim RefSelection
  Set RefSelection =
CreateObject("CSL.RWSSelection.Selection")
  RENHost.ConnectObject RefSelection, "RefSelection"
```

11. Choose the appropriate selection identifier (many listers have their own selections e.g. assetlister, behaviorLister... but we are only interested in data directly under the game tree). In order to get selections from the Game Explorer or Design View, use the global selection string in GlobalScript.

```
Sub OnLoad ()
      ' respond to global selection, i.e. that used by
      ' game explorer and design view.
      RefSelection.SelectionIdentifier =
GlobalScript.g_strGlobalSelection
  End Sub
```

12. Add placeholders for selection events.

```
Sub RefSelection_OnAddSelection ()
  End Sub

  Sub RefSelection_OnRemoveSelection ()
  End Sub

  Sub RefSelection_OnClearSelection ()
  End Sub
```

Setting and Retrieving Tags

Note: The full code-listing for these operations can be found in the Appendix to this topic

To be able to set or retrieve tags, you first need to choose a string that represents the tag and then register the tag.

```
' tag: object references (depends on) another
  Const conReferenceTag = "CUSTTAG_REFERENCE"

  Dim ObjectToTag, RefTag
```

ObjectToTag will store the object we want to tag. RefTag will contain the actual IRWSTag. To get this, just register the tag above with the IRWSScript object.

```
Sub Broadcast_PostLoadProject (strFileName)
      ' register the new tag
      Set RefTag = _
                GlobalScript.RWSScript.RegisterTag
(conReferenceTag)
  End Sub
```

This is done in the PostLoadProject event because it uses RWSScript from another module, so it must be initialized before used. For more information on the Broadcast mechanism, see the User Guide.

Now we can tag objects in the database.

Use the helper routine FindObjectByName (see included source code) to lookup and return a database object by its name (entered in the text box).

```
Sub SetButton_Click ()
      Dim Object
      Set Object = FindObjectByName (ObjectTextBox.Text)

      If Not Object Is Nothing And Not ObjectToTag Is Nothing Then
          RefTag.Set ObjectToTag.ID, Object.ID
          ObjectToTag.BuildCommand = "Tagged"
      Else
          MsgBox "No object of that name found. _
                  Reference data untouched."
      End If
  End Sub
```

Setting the tag data on ObjectToTag is straightforward - the data we are attaching is the ID of the referenced object. Also, you can set the build rule (BuildCommand) that you will write in Section 4 automatically for the object.

So, where do we get the ObjectToTag? Grab this from the selection and update the UI by implementing RefSelection_OnAddSelection. Then you can see which

object will be tagged in your UI. (Much of this code is error handling, the interesting bits are highlighted)

```
' ObjectType constants, as in Manager COM interface documentation
  Const otEntity = 2
  Const otFolder = 4
  Const otAsset = 8

  Sub RefSelection_OnAddSelection (id)
      If RefSelection.Count > 0 Then
          ' selections with more than one object
          ' not currently supported, do nothing
          If RefSelection.Count > 1 Then
              ResetUI
              ObjectLabel.Caption = "<multiples unsupported>"
              Exit Sub
          End If

          Set ObjectToTag = _
                   GlobalScript.ObjectFromId
(RefSelection.Item(1))
          If ObjectToTag.IDType <> otEntity And _
             ObjectToTag.IDType <> otAsset And _
             ObjectToTag.IDType <> otFolder Then
              ' unsupported object type
              ResetUI
              Exit Sub
          End If

          ObjectLabel.Caption = ObjectToTag.Name

          ' check for previous tag, update text box
          Dim LookupID
          LookupID = RefTag.Get (ObjectToTag.ID)
          If LookupID <> 0 Then ' not equal to
              ' restore previous tag, update UI
              Dim RefObject
              Set RefObject = GlobalScript.ObjectFromId (LookupID)
              ObjectTextBox.Text = RefObject.Name
          Else
              ObjectTextBox.Text = ""
          End If
      End If
  End Sub
```

Add support for the other basic selection events:

```
Sub RefSelection_OnRemoveSelection (id)
      If RefSelection.Count = 0 Then
          ResetUI
      Else ' any other case can be handled as-is by AddSelection
          RefSelection_OnAddSelection (id)
      End If
  End Sub

  Sub RefSelection_OnClearSelection ()
      ResetUI
  End Sub
  ' helper function for resetting UI and internal state
  Sub ResetUI ()
      ObjectLabel.Caption = "<none>"
      ObjectTextBox.Text = ""
      Set ObjectToTag = Nothing
  End Sub
```

As you can see, most of the work is in getting the UI to work properly and filter

out bad data. Tagging itself just consists of registering a tag and calling get/set.

Writing a Build Rule to Handle Tags

Finally, write a simple rule to check if referenced objects have been built. If not, build them first by adding subtasks to the rule (Note: subtasks are always executed and completed before the parent task can execute).

Recall that a build rule is typically composed of a dependency graph routine and a build command routine. However, a rule is not required to have both. In this case, checking dependencies is the only purpose of the reference tags. Simply add other tasks to do the building.

Create Tagged.rule in your project's Build Rules directory. Begin with the rule markup, which just includes some useful scripts and objects:

```
<?xml version="1.0" ?>
  <build version = "1.0">
     <rule name="Tagged" dependencygraph="TaggedDependencies"
                         strictvalidation="false">
        <object id="RWSScript" progid="CSL.RWSScript.RWSScript"/>
        <reference object="CSL.RWSScript.RWSScript"/>
        <script language="VBScript"
src="c:\rw\studio\programs\workspace\Build Rules\BuildTools.vbs"/>
        <script language="VBScript"><![CDATA[
```

Just as before, register the tag and get the data associated with it.

```
Sub TaggedDependencies (Task, APIObject, _
                          PlatformFilter, RecursionGuard)

     Dim RefTag, RefObject, RefObjectID
   ' entity depends on another entity
     Const conReferenceTag = "CUSTTAG_REFERENCE"

     Set RefTag = RWSScript.RegisterTag (conReferenceTag)
     RefObjectID = RefTag.Get (APIObject.ID) ' get tag data from
ID
```

Now find the Object by it's ID and type:

```
   ' get actual object from ID
     Select Case RWSScript.IDType (RefObjectID)
        Case otEntity Set RefObject = RWSScript.Entity
(RefObjectID)
        Case otFolder Set RefObject = RWSScript.Folder
(RefObjectID)
```

```
      Case otAsset  Set RefObject = RWSScript.Asset (RefObjectID)
```

If there is no data or an unexpected object type, add the task that would have been called if it wasn't tagged:

```
      Case Else
           BuildWarning "Tagged rule used on object not yet
supported or on object without a tag.  Check: " & APIObject.Name &
".  Building normally."
           ' process normally
           Task.AddSubtask GetTypeString (APIObject), _
                   APIObject, PlatformFilter, RecursionGuard
           Exit Sub
     End Select
```

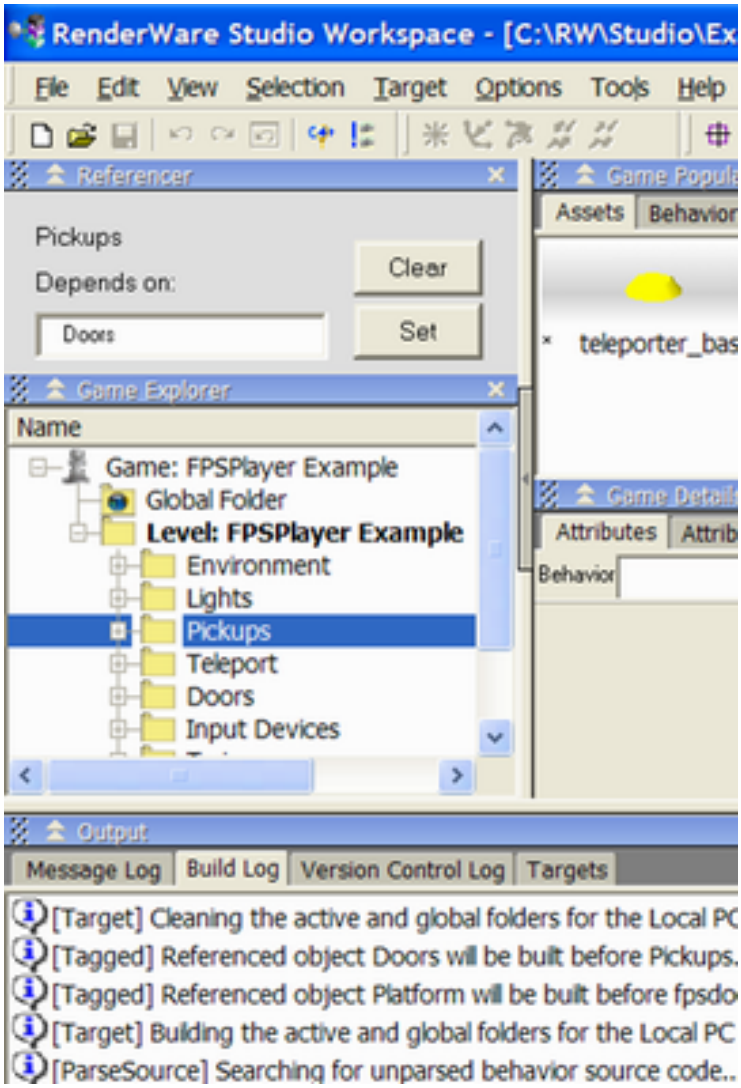
For an entity, the task used would be the default entity.rule. I.e. Task.AddSubtask

“Entity”, EntityObject, ...

At this point the Object and Tag info should be valid. Leverage the SentObjects list by checking if the referenced object is already in the list to send. If not, it would normally come later. But here you can enforce an ordering so the stream data will continue to run on the target platform, even though the ordering of objects is incorrect in the game database.

```
   ' check sent objects list.
     If Not GetParam ("SENTOBJECTS").Exists (RefObject) Then
        ' process the referenced object before this one
        BuildLog "Referenced object " & RefObject.Name & _
                 " will be built before " & APIObject.Name & "."
        Task.AddSubtask GetTypeString (RefObject), _
                        RefObject, PlatformFilter, RecursionGuard
     End If

     ' process this object normally
     Task.AddSubtask GetTypeString (APIObject), _
                     APIObject, PlatformFilter, RecursionGuard
  End Sub
      ]]></script>
   </rule>

</build>
```

The referenced object then will be in the list before the referrer. When the list is traversed by the rule that glues the individual object streams together, they will be concatenated in the correct order. By default, this is the RootFolder rule, in the build command routine.



You should now be able to click objects in the game explorer, type in object names as the reference, click **Set** and guarantee the correct ordering is maintained. Rebuild your project and check the BuildLog to see the order that objects are built in.

Suggestions

By default, tags do not get saved to disk. Reloading a project will lose your data. There are two options:

- Use Properties, which work in essentially the same way as tags, but allow complex data types. Properties automatically persist to disk, but also are sent to the target. If this is undesirable, you can modify the build rules to not include this data or strip it out.
- Use Tags and write code which persists the information to disk.

Searching by Entity name is not advised; it is used for simplicity. In a real application you might have a filtered drop-down list pre-populated with objects which could be referenced, so you would already know (or could cache) the IDs.

Appendix

```
Option Explicit
```

```
' ObjectType constants, as in documentation
Const otEntity = 2
Const otFolder = 4
Const otAsset = 8


' attach to selection object so can receive events
Dim RefSelection
Set RefSelection = CreateObject("CSL.RWSSelection.Selection")
RENHost.ConnectObject RefSelection, "RefSelection"


' store the object that might be tagged, and the IRWSTag object
Dim ObjectToTag, RefTag
Const conReferenceTag = "CUSTTAG_REFERENCE" ' object references
(depends on) another


'
---------------------------------------------------------------------

Sub OnLoad ()
    ' respond to global selection, i.e. that used by game explorer
and design view.
    RefSelection.SelectionIdentifier =
GlobalScript.g_strGlobalSelection
End Sub


'
---------------------------------------------------------------------

Sub Broadcast_PostLoadProject (strFileName)
    ' register the new tag
    Set RefTag = GlobalScript.RWSScript.RegisterTag
(conReferenceTag)
End Sub


'
-------------------------------------------------------------------------------
'   Notify edit control of selection event

Sub RefSelection_OnAddSelection (id)
    If RefSelection.Count > 0 Then
        ' selections with more than one object not currently
supported, do nothing
        If RefSelection.Count > 1 Then
            ResetUI
            ObjectLabel.Caption = "<multiples unsupported>"
            Exit Sub
        End If



        Set ObjectToTag = GlobalScript.ObjectFromId
(RefSelection.Item(1))
        If ObjectToTag.IDType <> otEntity And ObjectToTag.IDType
<> otAsset And ObjectToTag.IDType <> otFolder Then
            ' unsupported object type
            ResetUI
            Exit Sub
        End If

        ObjectLabel.Caption = ObjectToTag.Name
```

```
        ' check for previous tag
        Dim LookupID
        LookupID = RefTag.Get (ObjectToTag.ID)
        If LookupID <> 0 Then ' not equal to
            ' restore previous tag, update UI
            Dim RefObject
            Set RefObject = GlobalScript.ObjectFromId (LookupID)
            ObjectTextBox.Text = RefObject.Name
        Else
            ObjectTextBox.Text = ""
        End If
    End If
End Sub

'
--------------------------------------------------------------------------------
'   Notify edit control of deselection event

Sub RefSelection_OnRemoveSelection (id)
    If RefSelection.Count = 0 Then
        ResetUI
    Else
        RefSelection_OnAddSelection (id)
    End If
End Sub

'
--------------------------------------------------------------------------------
'   Notify edit control of clear event

Sub RefSelection_OnClearSelection ()
    ResetUI
End Sub
'
------------------------------------------------------------------------

Sub ApplyRef()
    Dim Object
    Set Object = FindObjectByName (ObjectTextBox.Text)
    If Not Object Is Nothing And Not ObjectToTag Is Nothing Then
        RefTag.Set ObjectToTag.ID, Object.ID
        ObjectToTag.BuildCommand = "Tagged"
    Else
        MsgBox "No object of that name found. Reference data
untouched."
    End If
End Sub

'
------------------------------------------------------------------------

Sub SetButton_Click ()
    ApplyRef
End Sub


'
------------------------------------------------------------------------

Sub ClearButton_Click ()
    If Not ObjectToTag Is Nothing Then
        RefTag.Set ObjectToTag.ID, 0
        ObjectToTag.BuildCommand = ""
        ObjectTextBox.Text = ""
```

```
    End If
End Sub


'
--------------------------------------------------------------------------

Sub ResetUI ()
    ObjectLabel.Caption = "<none>"
    ObjectTextBox.Text = ""
    Set ObjectToTag = Nothing
End Sub



' ------------------------------------------------------------------
' spin through DB looking for the object name.
' a necessary evil for looking up by text string, but works
' note: finds first instance of ObjectName


Function FindObjectByName (ObjectName)
    Dim Object
    Set FindObjectByName = Nothing
    For Each Object in GlobalScript.RWSScript.Folders
        If Object.Name = ObjectName Then
            ' found a matching folder
            Set FindObjectByName = Object
            Exit Function
        End If
    Next
    For Each Object in GlobalScript.RWSScript.Entities
        If Object.Name = ObjectName Then
            ' found a matching entity
            Set FindObjectByName = Object
            Exit Function
        End If
    Next
    For Each Object in GlobalScript.RWSScript.Assets
        If Object.Name = ObjectName Then
            ' found a matching asset
            Set FindObjectByName = Object
            Exit Function
        End If
    Next
End Function
```