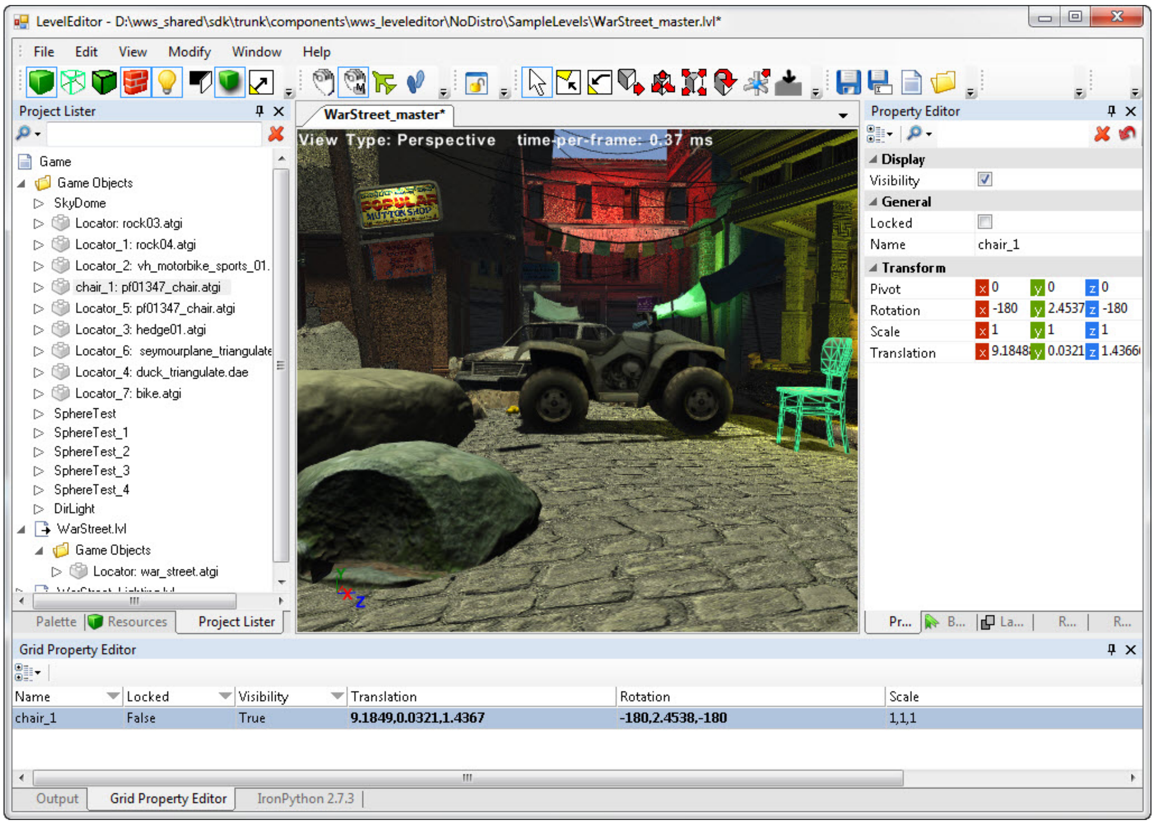
LevelEditor Programming Guide

Table of Contents

1 Overview

The LevelEditor is a powerful tool for constructing and assembling game levels. It provides a WYSIWYG interface (as shown in Figure 1) and tools for creating robust game levels. You can also add plug-ins that customize and extend the LevelEditor. The LevelEditor is based on the Sony Computer Entertainment (SCE) Worldwide Studios (WWS) LevelEditor tool.

Figure 1 LevelEditor WYSIWYG interface



Property Editor

Display			
Visibility	✓		
General			
Locked	☐		
Name	chair_1		
Transform			
Pivot	x 0	y 0	z 0
Rotation	x -180	y 2.4537	z -180
Scale	x 1	y 1	z 1
Translation	x 9.1848	y 0.0321	z 1.4366

Grid Property Editor

Name	Locked	Visibility	Translation	Rotation	Scale
chair_1	False	True	9.1849,0.0321,1.4367	-180,2.4538,-180	1,1,1

This guide describes the LevelEditor design, how to add a new game object type to the Palette, and the C language bridge application programming interface (API) between the managed and native sides of the LevelEditor.

The following LevelEditor features help you construct game levels efficiently and collaboratively:

- Work with a variety of file formats
- Associate assets with game objects
- Position, rotate, scale, and snap game objects precisely
- Edit game object properties
- Show or hide groups of game objects to unclutter the view as you work
- Construct Linears (lines and curves)
- Create custom terrains
- Add defined behaviors to game objects with a StateMachine plugin
- Create custom terrains using the Terrain Editor

Additional Resources for LevelEditor Users

For general information about the Sony LevelEditor, see the Wiki pages on GitHub: https://github.com/SonyWWS/LevelEditor/wiki

For developers who have access to the Sony SHared Information Portal (SHIP) for Sony Computer Entertainment (SCE) Worldwide Studios (WWS) game development, you can visit the LevelEditor SHIP page.

LevelEditor System Requirements

You can use the LevelEditor on a computer that provides the following required functionality:

- Microsoft Windows® 7 operating system (64-bit)
- Microsoft DirectX® 10 (or later) compatible graphics card

To modify the LevelEditor, you also need Microsoft Visual Studio® 2010 or Visual Studio 2013, including current Service Packs. Note that for Visual Studio 2010, you also need to install the Windows Software Development Kit (SDK) for Windows 8.1 (to the default location); for Visual Studio 2013, the Windows SDK for Windows 8.1 is already installed.

Installing the LevelEditor

The LevelEditor can be installed from the GitHub site or using the Sony Package Manager; see the *LevelEditor User's Guide*.

What's New for LevelEditor 3.5

LevelEditor 3.5 is a standalone tool, independent of the ATF. This version of the LevelEditor provides the following advantages and new features, compared with the ATF Level Editor 3.2:

- Support for 64-bit Windows operating systems.
- Native rendering (the rendering is done in C++ with DirectX 11).
- Integration with GameEngine through a bridge API.
- Level resources are loaded using a background thread.
- Support for different types of light sources that can be placed anywhere in the level.
- Dynamic shadows to help object placement.
- Can fully or partially use game engine code for rendering.
- Can easily add new shaders.
- Faster level loading.
- Can easily add new file format using native API (Collada DOM, Atgi lib, in-house binary format).

What's New for LevelEditor 3.6

LevelEditor 3.6 adds the following features:

- Package Manager. You now use the Package Manager to install LevelEditor.
- Integration of StateMachine with the LevelEditor. You can use StateMachine assets within the LevelEditor to define specific behaviors for game objects.
- Prototype assets. You can save one or more game objects as a prototype asset that you can then reuse in any game level. This feature replaces the previous version's Prototypes feature.
- Rendering normals. You can render game object normal vectors to help debug an object's geometry.

- Pick-cycling for overlapped objects. You can easily select objects that overlap or are visually obscured by other objects.
- Extension manipulator. You can extend or stretch an object along a single axis.
- Move pivot point command. You can quickly move an object's pivot point to one of several predefined locations.
- Pick filter. You can apply a simple filter to control which objects can be selected in the Design View.
- Sublevel object placement. You can easily drag objects to the Design View and have them appear within a particular sublevel, rather than at the top main level hierarchy.
- Bezier spline linear. You can define a linear as a Polyline, a CatmullRom spline, or a Bezier spline.

What's New for LevelEditor 3.7

LevelEditor 3.7 adds the following features:

- Open Source. The LevelEditor is available as Open Source from GitHub.
- Prefab assets. You can save one or more game objects as a prefab asset that you can then reuse in any game level. You can define both prefab assets and prototype assets.
- Terrain Editor. You can define and edit terrains based on a height map, layers, and decoration maps.
- Visual Studio solution. The solution for the rendering engine is now included in the base LevelEditor solution, and is no longer a standalone solution.

LevelEditor Design Overview

Previous versions of the LevelEditor were delivered with the Sony Authoring Tools Framework (ATF). The ATF is a set of C# and .NET components that enable game tool developers to build rich Windows client applications and game-related tools. For more information about the ATF, see the *Getting Started with ATF* guide (available from https://github.com/SonyWWS/ATF/tree/master/Docs).

The current version of the LevelEditor is built on the ATF, but is delivered as a standalone tool. As a standalone tool, the LevelEditor comprises three major components:

- LevelEditor.exe (the main C# application)
- LevelEditorCore (a C# assembly)
- LevelEditor plugins (MEF plugins)

The **LevelEditor.exe** file provides the following services:

- Create, load, and save level files (**.lvl** extension) into or from the Document Object Model (DOM) hierarchy. For more information about the DOM, see the ATF DOM documentation: *ATF Programmer's Guide: Document Object Model (DOM)* (available from https://github.com/SonyWWS/ATF/tree/master/Docs).
- A graphical user interface (GUI) for editing level and object properties.
- Editors for parts of the game world, such as the Resources folder and the Project Lister.

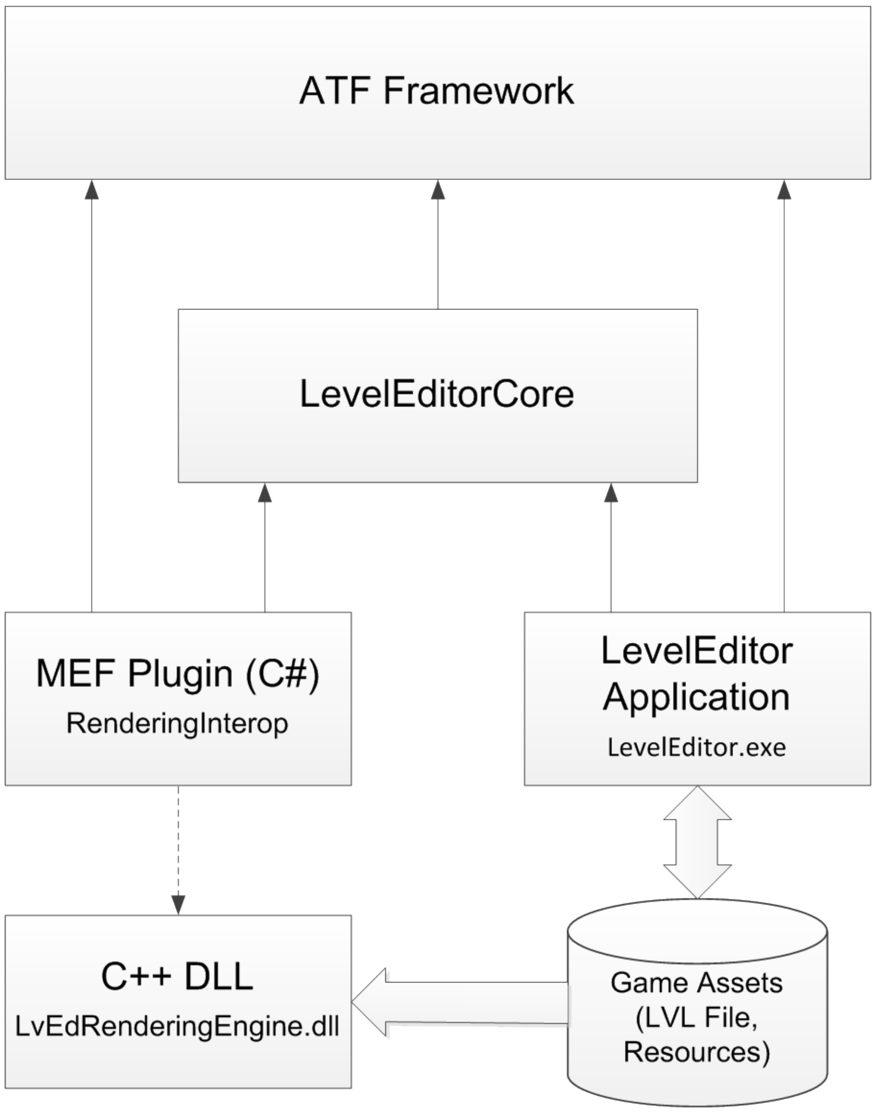
However, because the LevelEditor does not include a built-in rendering engine, it does not show 3D game objects in its Design View. The LevelEditor requires a plugin to provide 3D rendering services. The standalone LevelEditor includes a sample rendering engine plugin, **LvEdRenderingEngine.dll**, which you can use as is or modify as needed.

The LevelEditor.Core assembly provides a common framework that is referenced by the **LevelEditor.exe** and by LevelEditor plugins.

LevelEditor plugins provide the primary means for extending the LevelEditor. All plugins are Managed Extensibility Framework (MEF) components of the .NET Framework 4.0. You can use ATF-provided plugins with the LevelEditor or write your own plugin to work more effectively with your game engine.

Figure 2 shows the relationships between the various components within the LevelEditor design architecture.

Figure 2 LevelEditor design architecture



The LevelEditor source code is delivered in Visual Studio solutions: **.\build\LevelEditor.vs2010.sln** and **.\build\LevelEditor.vs2013.sln**. These solutions include the following projects:

- LevelEditor.vs2010
- LevelEditorCore.vs2010
- LvEdRenderingEngine (or LvEdRenderingEngine.vs2013)
- RenderingInterop.vs2010

The solution also refers to several ATF projects.

The MEF-based rendering-related plugins are included within the **RenderingInterop.vs2010** project. The "-Interop" suffix of project name indicates that it includes native components.

The **RenderingInterop.vs2010** project includes the major MEF components and classes listed in Table 2.

Table 1 Selected RenderingInterop components

Component	Description
NativeGameEditor	A MEF component that performs the following tasks: Parses the schema for native annotations, and tags DomNode objects that have a native counterpart. Registers DomNodeAdapter objects that are used for pushing DomNode events to native code. Listens to Document added and Document removed events, and creates or destroys levels in native code.
NativeDesignView	A MEF component that manages the quad view control. It is responsible for defining and registering the views to native code.
Commands\RenderCommands	A component that provides toolbar commands and property editor for toggling global render states.
DomNodeAdapters\ NativeGameWorldAdapter	A DomNodeAdapter that listens to child insert and child remove events for the entire game world. It also keeps the C++ rendering engine updated with actions on the C# side.
DomNodeAdapters\ NativeObjectAdapter	An adapter for DomNode objects that have a runtime counterpart. It also holds native object IDs and pushes DomNode property changes to native objects.

The default rendering engine, **LvEdRenderingEngine.dll**, is a native C/C++ DLL used by RenderingInterop. This DLL provides the following functionality:

- Loading 3D assets and creating the associated, required graphics resources.
- Creating the game world (the level) and game objects.
- Rendering the entire game world and design-time visuals, such as bounding boxes and manipulators.
- Culling and picking of game objects.
- Allowing the LevelEditor to control the life time of game objects and to edit their properties.
- Providing well-defined C-style entry points that are called by NativeRenderingEngineInterop.

To build your own LevelEditor executable from the source code, use Microsoft Visual Studio 2010 to build the **LevelEditor.exe** using the following solution: **.\build\LevelEditor.vs2010.sln**. This solution builds the LevelEditor GUI (including all required ATF project code), the native C/C++ rendering engine DLL, and the MEF-based rendering-related plugins.

2 Adding a New Game Object Type

The LevelEditor provides a limited number predefined game object types that the user can drag from the Palette to the Design View. You can modify these predefined game objects or define new game object types for your users. The basic tasks for generating a new game object type are:

- Define the new type
- Generate the C# code for the type
- If the new type has a C++ counterpart, generate and register schema objects, create C++ code for the type, and generate a new rendering DLL

The first two tasks work with the LevelEditor itself, the last set of tasks works with the LevelEditor rendering engine.

Defining the Type

All of the predefined game object types for the LevelEditor are defined in the LevelEditor XML schema file, **level_editor.xsd**. This file is located in the **\LevelEditor\schemas** directory. The **gap.xsd** file in the same directory defines the base types used in the **level_editor.xsd** file.

You can edit this XML schema file to modify existing game object types or define new ones. By default, the schema defines the game object types listed in Table 3. You can use any of the types defined in this file as the basis for your new game object types.

Table 2 Predefined game object types

Type	Description
billboardTestType	A billboard object
BoxLight	A directional light source, limited to its boundary
coneTestType	A cone object
controlPointType	A control point for working with linears
cubeTestType	A cube object
curveType	A linear
cylinderTestType	A cylinder object
DirLight	A directional light source
orcType	A creature object
planeTestType	A plane object
PointLight	An omnidirectional light source, limited to its boundary
shapeTestType	A generic shape object
skyDomeType	A sky dome (background for the game level)
sphereTestType	A sphere object

Generating the C# Code

After you edit the **level_editor.xsd** file, you need to use it generate corresponding C# code that the LevelEditor uses to populate the Palette window and manage the predefined game object types.

To generate the C# code:

(1) Open a Windows command prompt window (**Start > All Programs > Accessories > Command Prompt**).

(2) Navigate to the **\LevelEditor\schemas** directory.

(3) Run the **GenSchemaDef.bat** file: Enter `GenSchemaDef` at the Windows command prompt and press **Enter**.

This batch file runs the ATF DomGen utility, which converts an XSD file to a C# file, with the following parameters: the schema file name (**level_editor.xsd**), the output file name (**Schema.cs**), the schema name space (**gap**), and the class name space (**LevelEditor**).

(4) Build the **LevelEditor.exe** using the following solution: **.\build\LevelEditor.vs2010.sln**.

The DomGen utility generates the schema C# code in the **Schema.cs** file in the **\LevelEditor\schemas** directory.

Important: Do not edit this file; if you need to update the schema, edit the **level_editor.xsd** file and re-run the **GenSchemaDef.bat** file.

If your new game object type requires an icon in the Palette, perform the following tasks:

- Create a 16x16 pixel icon for the new game object type.
- Store the icon in the **\LevelEditorCore\Resources** directory.
- Add an image attribute to the game object in the **level_editor.xsd** file. For example, add the following element within the `<xs:appinfo>` element:

  ```
  <scea.dom.editors name="MyNewObj" category="Examples"
  image="LevelEditorCore.Resources.MyNewObj16.png" description="My New Object
  Type"/>
  ```
- Update the **Resources.cs** file in the **\LevelEditorCore** directory to refer to the icon. For example:

  ```
  /// <summary>
  /// MyNewObj icon name</summary>
  [ImageResource("MyNewObj16.png")]
  public static readonly string MyNewObjImage;
  ```
- Build the **LevelEditor.exe** using the following solution: **\build\LevelEditor.vs2010.sln**.

Generating the C++ Code

If your new type has a C++ counterpart, you need to generate corresponding C++ code that your rendering engine can use.

Generating and Registering Schema Objects

To generate the C++ code:

- Open a Windows command prompt window (**Start > All Programs > Accessories > Command Prompt**).
- Navigate to the **\LevelEditorNativeRendering\LvEdRenderingEngine\Bridge** directory.
- Run the **GenSchemaObjects.cmd** file: Enter `GenSchemaObjects` at the Windows command prompt and press **Enter**.

  This command file runs the LevelEditor CodeGenDom utility, which converts an XSD file to a C++ file, with the following parameters: the schema file name (**level_editor.xsd**), the output file name (**RegisterSchemaObjects.cpp**), and the class name space (**LvEdEngine**).

The CodeGenDom utility generates the game-object C++ code in the **RegisterSchemaObjects.cpp** file in the **\LevelEditorNativeRendering\LvEdRenderingEngine\Bridge** directory. The **RegisterSchemaObjects.cpp** file contains code that bridges the C# and C++ sides of the project.

Important: Do not edit this file; if you need to update the game schema object definitions, edit the **level_editor.xsd** file and re-run the **GenSchemaObjects.cmd** file.

Creating the C++ Code

You must create a C++ class for the new type that you defined. This C++ class should implement all of the functions declared in the **RegisterSchemaObjects.cpp** file for the type.

For example, if your new type is an icosahedron (a 20-sided regular solid), you need to provide a default constructor for the type or modify the CodeGenDom utility code to generate code to use your custom game-object factory for creating new objects.

The **RegisterSchemaObjects.cpp** file calls the default constructor, as shown below:

```
//--------------------------------------------------------------------------------
//IcosahedronGob
//--------------------------------------------------------------------------------
Object* IcosahedronGob_Create(ObjectTypeGUID tid, void* data, int size)
{
    return new IcosahedronGob();
}
```

If your new type includes other properties, such as colors, diffuse and normal lighting modes, texture transforms, and so on, the **RegisterSchemaObjects.cpp** file will declare functions for each of these properties, and you need to create code for the rendering engine to implement each of these properties.

Generating the Rendering DLL

After you have created your C++ code for the new type, you need to generate the rendering DLL.

Build the **LvEdRenderingEngine.dll** using the **\build\LevelEditor.vs2010.sln** or **.\build\LevelEditor.vs2013.sln** solution. The build places the **LvEdRenderingEngine.dll** file the **\NativePlugin\x64** directory, for example, the **.\bin\Release\NativePlugin\x64** directory.

3 C Bridge API

This chapter describes the C API that provides a bridge between rendering engine C++ and LevelEditor C# code. This bridge is implemented in a dynamic library (**LvEdRenderingEngine.dll**):

- For C++ code, this API is defined in the **LvEdRenderingEngine.h** file in the **\LevelEditorNativeRendering\LvEdRenderingEngine** directory.
- For C# code, this API is used by the **GameEngine.cs** file in the **\LevelEditorNativeRendering\NativeInterop** directory.

Table 4 lists the functions that comprise the C bridge API.

Table 3 C bridge API functions

Function	Description
LvEd_Begin	Begin rendering
LvEd_Clear	Clear the game world
LvEd_CreateFont	Create a font object
LvEd_CreateIndexBuffer	Create an index buffer
LvEd_CreateObject	Create a new instance of the specified game object type
LvEd_CreateVertexBuffer	Create a vertex buffer
LvEd_DeleteBuffer	Delete the specified index or vertex buffer
LvEd_DeleteFont	Delete the specified font object
LvEd_DestroyObject	Delete the specified game object
LvEd_DrawIndexedPrimitive	Draw the specified indexed primitive
LvEd_DrawPrimitive	Draw the specified primitive
LvEd_DrawText2D	Draw the specified text in the specified screen space
LvEd_End	End rendering
LvEd_FrustumPick	Select the specified frustum (3D region)
LvEd_GetObjectChildListId	Get the list ID for the specified game object type and list
LvEd_GetObjectProperty	Get a property for the specified game object
LvEd_GetObjectPropertyId	Get the property ID for the specified game object type and property
LvEd_GetObjectTypeId	Get the type ID for the specified game object (class name)
LvEd_Initialize	Initialize the rendering engine
LvEd_ObjectAddChild	Add the specified game object to its parent object within the specified list
LvEd_ObjectRemoveChild	Remove the specified game object from its parent object within the specified list
LvEd_RayPick	Select the specified ray
LvEd_RenderGame	Render the game world
LvEd_SetObjectProperty	Set a property for the specified game object
LvEd_SetRenderState	Set the global render state
LvEd_SetSelection	Set the selection
LvEd_Shutdown	Shut down the rendering engine
LvEd_Update	Update the game world

See the **LvEdRenderingEngine.h** file for detailed descriptions of each of these functions.

4 Defining Custom Metadata for Resources

You can define custom metadata for game resources. For example, textures, models, or audio files can have metadata, such compression type or memory layout. The metadata is stored in a separate file with the same name as the resource file, but with the "metadata" extension. For example:

- bricks_diffuse.dds
- bricks_diffuse.dds.metadata

Within the LevelEditor, you can use the Resource Metadata pane to view metadata for any object that you select in the Resources lister.

Classes for Working with Custom Metadata

Within the LevelEditor source code, the following classes and interfaces provide support for working with custom metadata:

ResourceMetadataEditor (LevelEditorCore.ResourceMetadataEditor)

> A MEF component that displays a PropertyEditor for the metadata of a currently selected resource.

IResourceMetadataService (LevelEditorCore.IResourceMetadataService)

> An interface implemented as a MEF component. It provides a method to get a list of file extensions reserved for metadata and another method to get metadata objects for given resource URIs.

ResourceMetadataService (LevelEditor.ResourceMetadataService)

> A client implementation of IResourceMetadataService. You can customize this implementation, as needed.

ResourceMetadataDocument (LevelEditor.DomNodeAdapters.ResourceMetadataDocument)

> Derives from **DomDocument** and represents a resource metadata document. This class writes the document to a file when any property changes.

Defining Custom Metadata

The following sections describe the steps for defining custom metadata for each resource type.

Step 1: Define a New Complex Type

Define a new complex type in your main LevelEditor XML schema file, **level_editor.xsd**, to store metadata for each type of resource. This file is located in the **\LevelEditor\schemas** directory.

The following example defines metadata for texture resource. You can define more complex types and type hierarchy than is shown in this simple example.

```
// define the complex type
<xs:complexType name="textureMetadataType">
     <xs:attribute name="keywords" type="xs:string" />
     <xs:attribute name="compressionSetting" type="xs:string" default="BC1" />
     <xs:attribute name="memoryLayout" type="xs:string" default="Linear" />
</xs:complexType>
```

```
// define the root element
<xs:element name="textureMetadata" type="textureMetadataType"/>
```

Step 2: Annotate the Type

Annotate the type so that it can be used by the ResourceMetadata subsystem. You can add annotations as with any other types in the schema.

The following example annotates the texture resource from the previous section:

```
// define the complex type
<xs:complexType name="textureMetadataType">
     <xs:annotation>
          <xs:appinfo>
               <ResourceMetadata metadataFileExt =".metadata" resourceFileExts
=".dds;.tga;.jpg;.jpeg;.bmp;.png;.tif;.tiff;.gif"/>
          </xs:appinfo>
     </xs:annotation>

     <xs:attribute name="keywords" type="xs:string" />
     <xs:attribute name="compressionSetting" type="xs:string" default="BC1" />
     <xs:attribute name="memoryLayout" type="xs:string" default="Linear" />
</xs:complexType>

// annotation
<ResourceMetadata
     // specify file extension for the metadata file.
     metadataFileExt =".metadata"
     // specify one or more resource file extensions
     // that apply to this metadata.
     resourceFileExts =".dds;.tga;.jpg;f"
/>
```

Step 3: Regenerate schema.cs

Run the **GenSchemaDef.bat** utility to regenerate the **Schema.cs** file. This utility is in the **\LevelEditor\schemas** directory.

Step 4: Register ResourceMetadataDocument

Register **ResourceMetadataDocument** on the metadata type (“textureMetadataType” in the examples in these sections).

The following C# code registers the texture resource from the previous sections:

```
          Schema.textureMetadataType.Type.Define(new
          ExtensionInfo<ResourceMetadataDocument>());
```

The following example adds DOM editor attributes to the texture resource from the previous sections:

```
<xs:complexType name="textureMetadataType">
     <xs:annotation>
          <xs:appinfo>
               <ResourceMetadata metadataFileExt =".metadata" resourceFileExts
=".dds;.tga;.jpg;.jpeg;.bmp;.png;.tif;.tiff;.gif"/>
               <scea.dom.editors.attribute
                 name="compressionSetting"
                 displayName="Compression Setting"
                 description="Compression Setting"
                 category="Metadata"
editor="Sce.Atf.Controls.PropertyEditing.EnumUITypeEditor,Atf.Gui.WinForms:BC1,BC3,Normal,HD
R-32bit,HDR-64bit,PVRT-2bpp,PVRT-4bpp,P4,P8,Uncompressed"/>
               <scea.dom.editors.attribute
                 name="memoryLayout"
                 displayName="Memory Layout"
                 description="Memory Layout"
                 category="Metadata"
editor="Sce.Atf.Controls.PropertyEditing.EnumUITypeEditor,Atf.Gui.WinForms:Linear,Swizzled,T
iled"/>
               <scea.dom.editors.attribute
                 name="mipMap"
                 displayName="Generate Mip Maps"
                 description="Generate Mip Maps"
                 category="Metadata"
editor="Sce.Atf.Controls.PropertyEditing.BoolEditor,Atf.Gui.WinForms"/>
          </xs:appinfo>
     </xs:annotation>
     <xs:complexContent>
          <xs:extension base="resourceMetadataType">
               <xs:attribute name="compressionSetting" type="xs:string" default="BC1" />
               <xs:attribute name="memoryLayout" type="xs:string" default="Linear" />
               <xs:attribute name="mipMap" type="xs:boolean" default="true" />
          </xs:extension>
     </xs:complexContent>
</xs:complexType>
```

5 Working with DomNodeAdapters

In general, working with DomNodeAdapters in the GameObject hierarchy is straightforward: you define schema types and the ATF Core converts the types to DomNodeTypes. You can register one or more DomNodeAdapter on any given DomNodeType.

However, in some cases, you might accidentally register the same adapter multiple times for a single DomNodeType. This chapter describes how to avoid this problem.

See the ATF programmer documentation for more information about DomNodeAdapters.

Define a New Type

Suppose that you define a schema type for curves:

```
<xs:complexType name="curveType" …
```

Then, within your C# code, you create the **Curve** class and register the type:

```
class Curve : DomNodeAdapter  { ... }
// register Curve on curveType
Schema.curveType.Type.Define(new ExtensionInfo<Curve>());
```

Now, you define a new type ("edgeType") that derives from curveType:

```
<xs:complexType  name="edgeType"  …  >

     <xs:extension base="curveType">

     …

</xs:complexType>
```

Finally, you typically create an adapter for the new type:

```
class Edge  : Curve { ... }
// In the schema, we have Edge that derives from Curve
// so why not do the same for Adapters?

// register
Schema.edgeType.Type.Define(new ExtensionInfo<Edge>());
```

The Problem

Note that for the following code, two extensions will be created:

```
DomNode node = new DomNode("schema.edgeType.Type");
node.InitializeExtensions();
```

This code creates an instance of type class **Edge** and another instance of type class **Curve**.

A problem with having two extensions arises because **Edge** derives from **Curve**. If your code has any local data in **Curve** or has event listener code, the local data will be duplicated and any event will be processed twice. Such duplication can make it difficult to debug.

Avoiding the Problem

The following two simple rules help you to avoid the problem described in the previous section:

- Do not derive a new adapter from any adapter that is already registered.

  For example, do not derive another adapter from **Curve**, because the **Curve** adapter is already registered.
- Do not derive a new adapter from any other adapter that has been indirectly registered.

Example:

Suppose you have the following type hierarchy defined in your schema:

```
complexType  name="gameObjectType"

complexType  name="curveType"       base="gameObjectType"

complexType  name="edgeType"        base="curveType"
```

Table 5 shows some example code that you might consider for working with this type hierarchy. The table shows code that is OK to use because it avoids the double extension problem, and code that is not Ok to use.

Table 4 Example code for the example type hierarchy

Code OK	Code Not OK

`abstract class GameObject : DomNodeAdapter`
 	
`// GameObject is not registered on any type.`	

`Class Curve : GameObject`
 	
`// now you can register Curve on curveType.`	

	`Class Edge : Curve`
	
	`// not OK because Curve is registered on curvetype which is parent of edgeType.`

	`Class Edge : GameObject`
	
	`// not OK because GameObject is indirectly registered on curveType by registering class Curve`

`Class Edge : DomNodeAdapter   // is OK`	

The following code shows how to work with the type hierarchy:

```
abstract class GameObject : DomNodeAdapter
// GameObject is not registered on any type.
```

```
Class Curve : GameObject      // is OK
// now you can register Curve on curveType.

Class Edge : DomNodeAdapter   // is OK
{
   void SomeMethod()
   {
      // What if I need to access few properties of Curve
      Curve cv = this.As<Curve>();
      // Because adapter Curve is registered on curveType and
      // edgeType is derived from curveType, you can adapt this node to Curve

      // you can adapt this to GameObject
      GameObject gob = this.As<GameObject>();

   }
}
```