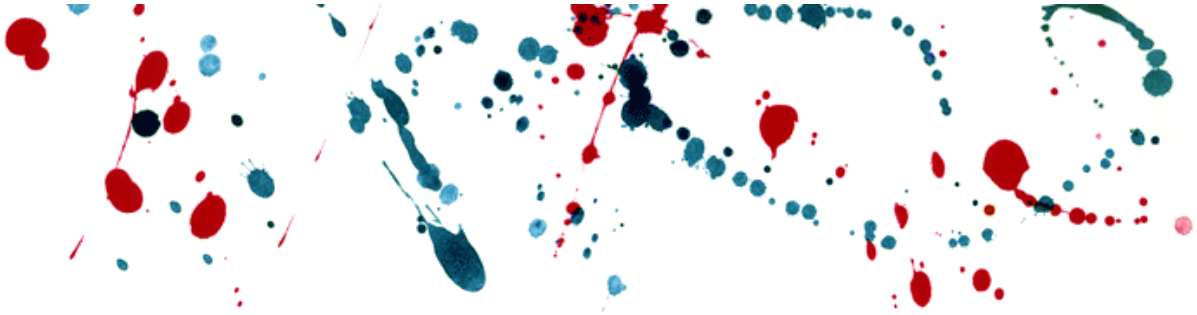
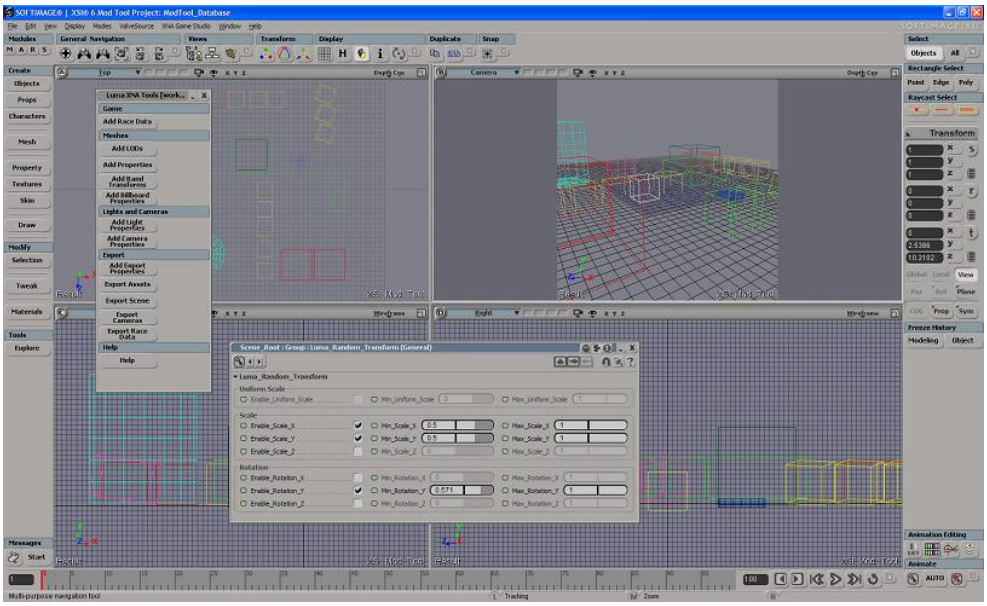
How to Turn XSI Mod Tool into a Level Editor for your XNA Games



For XNA 3.01 v3.0, (Last edited: May 2009)

Herman Tulleken (herman.tulleken@gmail.com)

Versions 1.0 and 2.0 of this tutorial were originally published on Luma Labs (http://www.luma.co.za/labs). The current version (Version 3.0) is published on code-spot (http://www.code-spot.co.za).



1. Get XSI ready for Python scripting

One of the most important tools for making a game is the level editor. A good level editor makes it possible to

- build and modify levels quickly,

- get an impression of how the level will look in the game, and
- edit the setup for game specific features.

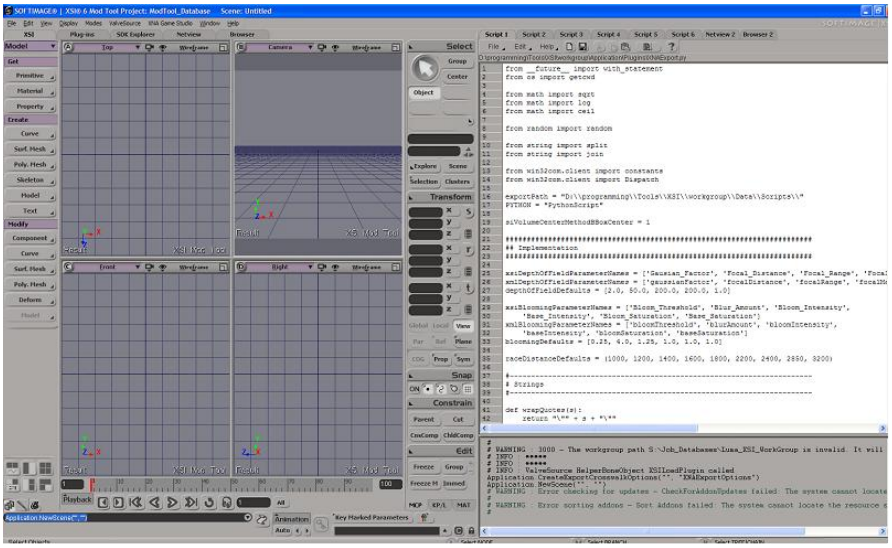
Rolling out your own level editor is a daunting task, and as you will see, also an unnecessary one. The XSI Mod Tool has many features you will need for a level editor, and provides simple mechanisms for adding features specific to the game you are building.

This tutorial describes how to write a self-installing plug-in that can export level data from XSI, and how to set up your XNA projects to handle that data. Python is a great language for scripting in XSI, and is used for this tutorial.[1]

However, if you use one of the other languages, you should still be able to use most of the information presented here.

If Python or PyWin is not installed on your system, follow these steps:

1) If XSI is open, close it.
2) Install Python (http://www.python.org/download/).
3) Install PyWin (http://sourceforge.net/projects/pywin32/).
4) Open XSI, and set the XSI scripting language to Python:
   a) From the **File** menu, select **Preferences...**
   b) Select the **Scripting** node from the tree on the left.
   c) Select **Python ActionX** from **Script Language** combo box from the pane on the right.



[1] For a tutorial that explains Python scripting in XSI, see http://www.luma.co.za/labs/2007/11/28/xsi-scripting-using-python/.

The best mode to use for scripting in XSI is the **XSI Tool** mode, which you can select from the **Modes** menu.

To test that Python works correctly in XSI, type the following in the script window:

```
Application.LogMessage('Hello World!')
```

and press the **Run** button in the toolbar at the top of the scripting window.

2. Exporting a level file

The basic idea is to build your level in the Mod Tool, and then export the relevant data into a level file using a custom script. This does not replace the publishing step – you will still need to publish models separately.

The level file contains the positions, rotations, scale, and other relevant information. In your game, you will need to read this file, and build the level accordingly. How you do this depends on your game and the file format you choose.

I highly recommend that you store your level data in XML:

- it is easy to parse even richly structured data, and C# has build in XML parsing support;
- it is human readable, and will make it a lot easier to spot errors with your export script;
- it is easy to keep XML in separate files without losing the structure.

You will need a way to know what model a piece of your level file refers to. A good way to do this is to store the model location (relative to the Content folder) in a custom parameter of the XSI object, and export this with the other level data. When you parse the level file, you can use this info to load the appropriate level. I usually strip the root folder and extension from the AssetPath parameter in the XNA_Asset property set to get this path.

XSI scene files can become heavy. To speed up level building, you can use cubes or other simple shapes instead of the actual models. All will work fine, as long as you store the right paths with the objects.

You can also build your levels in separate scenes, for example one scene is used for buildings, one scene for trees, and so on. You should then modify your loader to load in all the different level files.

Here is an example of how a level file might look. This one contains two models, and is set up for reading in by content pipeline classes (see Section 7).

```
<?xml version="1.0" encoding="utf-8" ?>
<XnaContent>
<Asset
Type="System.Collections.Generic.List[EngineLibrary.Objects.B
asicModel]">
      <Item>
            <ModelAsset>Models\spaceship</ModelAsset>
            <Position>0 10 0</Position>
      </Item>
      <Item>
            <ModelAsset>Models\spaceship</ModelAsset>
            <Position>0 1 0</Position>
      </Item>
</Asset>
</XnaContent>
```

The next sections describe the various XSI commands you will need to get object data from your scene. How you write this into a file is up to you.

2.1 Getting basic properties

This is how you get the XSI name you give an object:

```
def getName(obj)
      return obj.Name
```

The type of object (stored as a string) is similarly obtained:

```
def getType(obj):
      return obj.Type
```

Tip

I always keep the following piece of code in one of my script windows to quickly determine the type of an object:

```
obj = Application.Selection[0]
Application.LogMessage(obj.Type)
```

Selecting an object and running the script is much faster than trying to find the information in the documentation.

Object	Type string
Camera Root	'CameraRoot'
Camera	'camera'
Camera Interest	'CameraInterest'
Curve	'crvlist'
Group	'#Group'
Light	'light'
Model, Scene root	'#model'
Null	'null'
Nurbs surface	'surfmsh'
Particle	'particle'
Polygon mesh	'polymsh'
Shader	'Shader'
Visibility property	'visibility'

2.2 The selection

Most of the time, your script functions should only process selected items. The selected items can be obtained as a collection using Application.Selection. The following function will print the names of all selected items.

```
def printSelectedObjectNames():
      for obj in Application.Selection:
            Applciation.LogMessage(obj.Name)
```

2.3 Traversing the object hierarchy

You can get the child objects of an object by using the property Children. The following code will print the name of every object in the object hierarchy:

```
def printNameRecursively(obj):
      Application.LogMessage(obj.Name)
      for child in obj.Children:
            printNameRecursively(child)

def printAllObjectNames():
      printNameRecursively (Application.SceneRoot)
```

2.4 Getting the transforms from XSI objects

Getting the local transforms of an object is easy:

```
def getPos(obj):
      return [obj.posx.Value, obj.posy.Value, obj.posz.Value]

def getRot(obj):
      return [obj.rotx.Value, obj.roty.Value, obj.rotz.Value]

def getScl(obj):
      return [obj.sclx.Value, obj.sclx.Value, obj.sclx.Value]
```

2.5 Properties and Parameters

You will surely need to add data to objects specific to your game. XSI allows you to add property sets to any object. A property set is a set of parameters, which can be of type string, float, bool, and so on. A property set can be opened from the **Explorer**, which brings up a property sheet which is a nice user interface for changing the parameter values.

The following code will add a custom property set to an object:

```
def addPorpertySet(obj, pName):
      return obj.AddCustomProperty(pName)
```

Usually, you want to check whether the property does not already exist, like this:

```
def addPorpertySet(obj, propertyName):
      pSet = obj.Properties(propertyName)

      if pSet == None:
            pSet = obj.AddCustomProperty(propertyName)

      return pSet
```

You can now add custom parameters to your property set:

```
from win32com.client import constants

def addMyPropertySet(obj):
      pSet = addPropertySet(obj, 'Test')
      parmName = 'Float_Test'
      parm = pSet.Parameters(parmName)

      if parm == None:
            pSet.AddParameter3(parmName, constants.siFloat,
                  0.5, 0.0, 1.0)
            parmName = 'Bool_Test'

      parm = pSet.Parameters(parmName)

      if parm == None:
            pSet.AddParameter3(parmName, constants.siBool,
                  True)
      parmName = 'String_Test'
      parm = pSet.Parameters(parmName)

      if parm == None:
            pSet.AddParameter3(parmName, constants.siString,
                  'Hello')
```

The following code will add the property set to all selected objects:

```
for obj in Application.Selection:
      addMyPropertySet(obj)
```

Select a few objects in a scene, and run the code. Open the property sheet from the **Explorer**, and adjust the values.

The following code shows how to access parameter values from script:

```
def printMyPropertySetValues(obj):
      pSet = obj.Properties('Test')
      if pSet == None:
            raise Exception(
                  'The object contains no property '\
                  'named \'Test'\.')
      parameter = pSet.Parameters('Bool_Test')

      if parameter == None:
            raise Exception(
```

```
                    'The property set contains no parameter'\
                    'named \'Bool_Test'\.')

      Application.LogMessage(parameter.Value)
      parameter = pSet.Parameters('Float_Test')

      if parameter == None:
            raise Exception(
                  'The property set contains no parameter '\
                  'named \'Float_Test'\.')
      Application.LogMessage(parameter.Value)
      parameter = pSet.Parameters('String_Test')

      if parameter == None:
            raise Exception(
                  'The property set contains no parameter '\
                  'named \'String_Test'\.')
            Application.LogMessage(parameter.Value)
```

2.6 Property sheets and layout

You can change the layout of property sheets, and control the type of GUI controls used to manipulate parameter values in the property sheet.

```
def doMyPropertySetLayout(obj):
      property = obj.Properties('Test')
      layout = property.PPGLayout
      layout.Clear()
      layout.AddGroup('Test Properties')
      layout.AddItem('Float_Test','Float Test',
            constants.siControlNumber)
      layout.AddRow()
      layout.AddItem('Bool_Test', 'Bool Test',
      constants.siControlCheck)
      layout.AddItem('String_Test', 'String Test',
            constants.siControlText)
      layout.EndRow()
```

All items added between AddGroup and EndGroup are added in a bordered group. Items added between AddRow and EndRow are placed in the same row. You can also use tabs with AddTab (there is no EndTab – the next tab begins when you call AddTab again).

When you change the layout of a property sheet, you should register it as a property with your plug-in, otherwise the layout will be

destroyed when you close the scene. This is discussed in the section *Making a self-installing plug-in*.

2.7 Lights

Many of a light's properties are stored in a shader. Lights can have more than one shader, but in most cases will have only one, which can then be obtained with light.Shaders(0).

```
def getColor(light):
      shader = light.Shaders(0)
      color = shader.Parameters('color')
      red = color.Parameters('red').Value
      green = color.Parameters('green').Value
      blue = color.Parameters('blue').Value

      return (red, blue, green)

def getIntensity(light):
      shader = light.Shaders(0)
      intensity = shader.Parameters('intensity').Value

      return intensity
```

2.8 Cameras

If you have more than one camera in your game, you might benefit from setting up your camera data in XSI. The perspective camera provides the basic properties you will need; you can add custom parameters for any additional properties. The following code stores some useful camera properties in variables:

```
def getCameraProperties(cameraRoot):
      if cameraRoot.Type == 'CameraRoot':
            camera = cameraRoot.Camera
            fov = camera.fov.Value
            aspect = camera.aspect.Value
            near = camera.near.Value
            far = camera.far.Value
            interestVector = [
            camera.Interest.posx.Value,
            camera.Interest.posy.Value,
            camera.Interest.posz.Value]
      else:
            raise Exception('You tried to obtain camera '\
                  'properties from an object that is not '\
                  'a camera.')
```

2.9 Curves

All curves are represented as lists of points. How those points are interpreted as a curve depends on which type of curve you draw. The following function retrieves all the points of a curve.

```
def getPoints(curve):
      pointsList = []
      if curve.Type == 'crvlist':
            points = curve.ActivePrimitive.Geometry.Points
            for point in points:
                  pos = point.Position
                  p = [pos.X, pos.Y, pos.Z]
                  pointsList.append(p)

            return pointsList
      else:
            raise Exception('You tried to obtain curve '\
                  'properties from an object that is not '\
                  'a curve.')
```

If the curve passed to this function was drawn using Bézier knot points, the points in the list are in the format [**p0**, p1, p2, **p3**, p4, p5, **p6**, p7, ..., **pn**], where points in bold re points actually on the curve. As far as I can tell, there is no way to determine the type of curve with the scripting commands. You should either always use the same curve, or add a custom parameter for the curve type.

2.10 Tricks with Groups

If you add a property set to a group, that property set can be accessed from any object in the group as if the property set is attached to that object.

```
from win32com.client import constants

group = Application.CreateGroup('Test')
pSet = group.AddCustomProperty('GroupTest')
pSet.AddParameter3('IsAlive', constants.siBool, True)

for obj in group.Members:
      pSet = obj.Properties('GroupTest')

      if pSet != None:
            parm = pSet.Parameters('IsAlive')
            isAlive = parm.Value
            Application.LogMessage(isAlive)
```

3. Making a self-installing plug-in

3.1 Custom Commands

To be able to call your functions from toolbars and other scripts, you must make a custom command of them.

To register a command, you need to define two functions. If your definition name is myFun, you will need to implement myFun_Init and myFun_Execute.

```
def myFun_Init( io_Context ):
      oCmd = io_Context.Source
      Application.LogMessage("myFun_Init called" )
      oCmd.Description = ""
      oCmd.ToolTip = ""
      oCmd.ReturnValue = True

      return True

def myFun_Execute( ):
      Application.LogMessage("myFun_Execute called" )
      addRaceDataImpl()
      return True
```

3.2 Properties

Registered properties can receive callbacks, which makes them very powerful. Here are some of the callbacks you might want to implement:

- OnInit
- Define
- DefineLayout
- button_OnClicked
- parameter_OnChanged

Callbacks are defined as normal Python functions, with the property name prefixed. For example:

```
def MyProperty_Init(ioContext):
      Application.LogMessage('MyProperty has been '\
            'initialised')
```

The OnClick and onChanged callbacks also take the relevant parameter or button as part of the name:

```
def MyProperty_MyParameter_OnChanged():
      Application.LogMessage('MyParameter changed')
```

The Define callback

In this callback you write the code to add custom parameters to the property set.

```
def PropertyTest_Define(ioContext):
      pSet = ioContext.Source
      pSet.AddParameter3('FloatTest', constants.siFloat,
            True)
      pSet.AddParameter3('BoolTest', constants.siBool, True)
      pSet.AddParameter3('StringTest', constants.siString,
            'Hello')
```

The DefineLayout callback

This callback determines the layout:

```
def PropertyTest_DefineLayout(ioContext):
      layout = ioContext.Source
      layout.Clear()
      layout.AddGroup('Test Properties')
      layout.AddItem('Float_Test', 'Float Test',
            constants.siControlNumber)
      layout.AddRow()
      layout.AddItem('Bool_Test', 'Bool Test',
            constants.siControlCheck)
      layout.AddItem('String_Test', 'String Test',
            constants.siControlText)
      layout.EndRow()
      layout.AddButton('Done')
```

Note that we also added a button. The logic that will be executed when this button is clicked will be defined in the OnClick callback, described below.

The parameter OnChange callback

This callback is called whenever the parameter for which it is defined is changed.

```
def PropertyTest_Bool_Test_OnChanged():
      PPG.String_Test.Enable(PPG.Bool_Test.Value)
```

Notice how the PPG object is used to get hold of parameters in the property set.

The button OnClick callback

This callback is called whenever a button (added to the layout) has been clicked. In the following example, the property self-destructs when the button is clicked.

```
def PropertyTest_Done_OnClicked():
      Application.DeleteObj(PPG.Inspected[0])
      PPG.Close()
```

3.3 Registering your plug-in

To make a self-installing plug-ins, you must define the method XSILoadPlugin, in which you register all your custom commands and properties, and set up some other plug-in data.

The following example registers the custom command and property set up in the previous two sections.

```
def XSILoadPlugin( in_reg ):
      sCommandFile = in_reg.FileName
      in_reg.Author = 'My Name'
      in_reg.Name = 'Test Plug-in'
      in_reg.Major = 1
      in_reg.Minor = 0
      in_reg.RegisterCommand('myFun')
      in_reg.RegisterProperty('PropertyTest')

      return True
```

Your plug-in is reloaded when you open XSI, or whenever you save the script from the XSI script editor. To verify that your plug-in was loaded:

1) Select **File** | **Plug-in Manager...**
2) Open the User Root or workgroup where your plug-in script is located.
3) Open the **Plug-ins** node.
4) Verify that your plug-in is under that node
5) Verify that there is no red triangle indicating an error.

Errors that occur on loading are reported in the output pane of the script editor.

4. Toolbars

Once you have defined a custom command, you can link it to a toolbar. To create a new toolbar, select **View | New Custom Toolbar...** from the menus.

A new empty toolbar will appear. To save the toolbar, right-click on it, and save it. Toolbars saved in a workgroup will be available for all users connected to the workgroup – see below.

To add you custom command to the toolbar, right-click on the toolbar, and select **Customize Toolbar...** from the context menu. Select the **Custom Script Commands** from the group list. Hunt for your custom command in the list on the right and drag it onto the toolbar.

To customize the button, right click on the button, and select **Customize Button**. A dialog will appear from which you can change the button size, label, and icon.

Toolbars are not automatically saved: save it when you have made all your changes!

5. Workgroups

Workgroups provide a convenient way for a group of people (in a studio, for example) to share plug-ins and add-ons.

To create a workgroup, create a folder accessible to everyone that will use the workgroup. To connect XSI to the workgroup, follow these steps:

- From the **File** menu, select **Preferences...**
- Select the **Data Management** node in the tree.
- In the pane on the right, scroll down to near the bottom, and click on **Manage Workgroups...**
- Click on the **Connect...** button.
- Use the file browser to locate the folder you have created for the workgroup, and select it.
- Click **OK**, and dismiss the preferences window.

To verify that plug-ins in the workgroup are also working in your XSI, do the following:

- From the File menu, select **Plugin Manager...**

- In the tree, you will see the **Factory Root**, the **User Root**, and all the workgroups you are connected to. Expand the relevant workgroup node to see what plug-ins and other goodies are installed under the workgroup.

Workgroup menu items are marked with a [w]. User menu items are similarly marked with a [u].

5.1 Workgroups for Developers

I found the following setup works well for developing for workgroups.

Create two workgroups: one is a private workgroup from where you will develop; the other is a public workgroup users will link to. Both must be linked to the same SVN (or CVS) repository. After you made changes to the plugins in the private workgroup, simply commit the changes, and update the public workgroup.

It is not a good idea to use your user root for development. Using a private workgroup instead allows you to:

- keep your development separate from any plug-ins you might install from other vendors and developers;
- disconnect from it when its faulty;
- let others connect to it to debug machine specific issues.

6. Loading XNA Content Dynamically

(This section is adapted from information on Shawn Hargreaves' excellent blog: http://blogs.msdn.com/shawnhar/archive/2007/06/06/wildcard-content-using-msbuild.aspx.)

For your game to load levels from level files, you need to automate asset building.

Follow the steps described below.

6.1 Create a dynamic content project

Create a separate project for all your dynamic content. You might still want some static content as well (that is, content you add manually to the project).

Add the following DLLs as references to the project, as well to the nested content project:

- Softimage.XWImporter.dll
- XSIXNARuntime.dll

These DLLs come with the XSI XNA add-on. (At the time of writing, no DLLs that work for XNA 3.0 has been released. You should still be able to use XSI with other formats, such as FBX, though).

6.2 Create an include project

Create an empty file in the same folder as your content project, and call it include.proj. Put the following in the file, and adapt it as you find necessary.

```
<?xml version="1.0" encoding="utf-8"?>
<Project
DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
<ItemGroup>

<WildcardContent Include="Content/Models/**/*.xsi">
      <XNAUseContentPipeline>true</XNAUseContentPipeline>
      <Importer>Importer</Importer>
      <Processor>Processor</Processor>
</WildcardContent>

<WildcardContent Include="Content/Textures/**/*.tga">
      <XNAUseContentPipeline>true</XNAUseContentPipeline>
      <Importer>TextureImporter</Importer>
      <Processor>SpriteTextureProcessor</Processor>
</WildcardContent>
</ItemGroup>
</Project>
```

Every WildCardContent node specifies where the content is located, whether to use the content pipeline, and the importers and processors to use. The example above specifies all .xsi files in the folder Content/Models/ and its subfolders, as well as all the .tga files in the folder Content/Textures/ and its sub folders.

6.3 Include 'include.proj' in the content project

Find the file content\Content.contentproj in your dynamic content folder. Open the file in a text editor, and find the line:

```
<Import Project="$(MSBuildExtensionsPath)\Microsoft\XNA Game
Studio\v2.0\Microsoft.Xna.GameStudio.ContentPipeline.targets"
/>
```

and put the following line directly below it:

```
<Import Project="includes.proj" />
```

In the same file, also add the following. You can add it below the line above, or modify the commented-out XML.

```
<Target Name="BeforeBuild">
 <CreateItem
Include="@(WildcardContent)"
AdditionalMetadata="Name=%(FileName)">
<Output TaskParameter="Include" ItemName="Compile" />
 </CreateItem>
 </Target>
```

6.4 Create a batch file for building and copying binaries

Create a batch file for each configuration (Debug, Release, etc.) that does the following:

- Call MSBuild on your content project. You can use the option /property:Configuration=Debug to set the configuration.
- Redirect output to a log file using the > redirect operator.

The line typically looks like this:

```
%windir%\Microsoft.NET\Framework\v3.5\MSBuild.exe
/property:Configuration=Debug "engine.csproj" > log.log
```

This line builds the project "engine.csproj" with the MSBuild distributed with .Net 3.5, and redirects the output to the file log.log.

(Note: with previous versions of XNA, you also had to copy the compiled assets to the correct locations. This is not necessary for XNA 3.0).

7. Parsing XML level files

An XML asset can represent a game object or list of such objects. For instance, we can use an XML file to represent a level – the file gives information on the placement and state of the various objects in the level.

This example uses BasicModel as the game object for which we want to use XML files.

Create two projects. One is the actual game, the other a library.

In addition to BasicShape, we also need to implement a content reader and content writer. All three these files must be in the library project.

The game project should reference the library project.

The BasicModel

This section is somewhat based on a ZiggyWare article: http://www.ziggyware.com/readarticle.php?article_id=150.

This won't be necessary for XNA 3.1. See http://blogs.msdn.com/shawnhar/archive/2009/03/25/automatic-xnb-serialization-in-xna-game-studio-3-1.aspx.

- Implement all the properties you want to save as C# properties.
- Mark any properties that should not be serialised with the [ContentSerializerIgnore] property.
- Mark the class with the [Serializable] attribute.
- The namespace of this class is important. Suppose, for this example, it is EngineLibrary.Objects. EngineLibrary is the library project.

The Writer

- This class must extend from ContentTypeWriter<BasicModel> in our example. The type BasicModel must be replaced with whatever class must be serialised.
- Mark the class with the [ContentTypeWriter] attribute.
- Implement the Write and GetRunTimeReader methods. The write methods should write th properties from the value (an instance of the class we are serialising, in this case BasicModel) to the output writer. The GetRuntimeReader method should return the appriopriate reader, like this:

```
public override string GetRuntimeReader(
      TargetPlatform targetPlatform)
{
return typeof(BasicModelContentReader).AssemblyQualifiedName;
}
```

The Reader

- This class must extend ContentTypeReader<BasicModel>.
- You only have to implement the Read method. This method should read properties from the input stream, and assign their values to the properties of a freshly created instance of the

sterilized type (in this case, BasicModel), and return this instance.

Notes

- You can get templates for the reader and writer when you create new items.
- The template uses the following lines:

```
// TODO: replace this with the type you want to read.
using TRead = System.String;
```

You only need to replace System.String with the appropriate type (BasicModel, in our example). All references to the serialised type is to TRead. Alternatively, you can replace all occurrences with TRead with the appropriate type, and remove the two lines above.

- The Reader and Writer files should not be in the same project as the content. It is customary to have the Writer in a pipeline extension project, and the reader in a library project (the same one that contains the class associated with it).

The XML File

- The XML file should have XnaContent as the root.
- The Asset should be in an Asset tag, with the Type attribute set to the appropriate type.
- List items should each be in an Item tag.
- Each property of the item should be in its own tag. Below is an example.

```
<?xml version="1.0" encoding="utf-8" ?>
<XnaContent>
<Asset
Type="System.Collections.Generic.List[EngineLibrary.Objects.B
asicModel]">
      <Item>
            <ModelAsset>Models\spaceship</ModelAsset>
            <Position>0 10 0</Position>
      </Item>
      <Item>
            <ModelAsset>Models\spaceship</ModelAsset>
            <Position>0 1 0</Position>
      </Item>
</Asset>
</XnaContent>
```