Overview

We have a generic interface for parsing and translating XML files into Lua files (in the future we'll also be able to go the other way).

In order to do this simply, we use an XML file as a definition format that declares what kinds of XML is included in a file, and what kind of Lua to create from that XML. This format includes some simple validation methods to ensure that the data expected is the data that is received.

We distinguish between three kinds of data: **properties**, **arrays**, and **tables**.

A table represents a Lua based table:

```
letters = { a="a", b="b", c="c" };
```

In an XML data file this table would look like so:

```
<letters a="a" b="b" c="c"/>
```

In an XML definition file this table would look like so:

```
<Table name="letters">
  <Property name="a" type="string"/>
  <Property name="b" type="string"/>
  <Property name="c" type="string"/>
</Table>
```

Each element can be marked as

```
optional="1"
```

in the definition file to declare that it is not required.

There are two types of arrays; one is a simple group of element, like the Lua array:

```
numbers = {0,1,2,3,4,5,6,7,8,9}
```

In the XML data file this will might like so:

```
<numbers>
  <number value="0"/>
  <number value="1"/>
  <number value="2"/>
  <number value="3"/>
  <number value="4"/>
  <number value="5"/>
  <number value="6"/>
  <number value="7"/>
  <number value="8"/>
  <number value="9"/>
</numbers>
```

In the data description file this would look like so:

```
<Array name="numbers" type="int" elementName="number"/>
```

Another type of array is an array of tables; in Lua:

```
wheels = {
  {size=3, weight=10},
  {size=2, weight=1},
  {size=4, weight=20},
}
```

And in the XML data file:

```
<wheels>
  <wheel size="3" weight="10"/>
  <wheel size="2" weight="1"/>
  <wheel size="4" weight="20"/>
</wheels>
```

And in the definition file:

```
<Array name="wheels" elementName="wheel"> <!-- note no type is attached -->
  <Property name="size" type="float"/>
  <Property name="weight" type="int"/>
</Array>
```

Loading and Saving a Table from Lua:

If you have a Lua table that needs to be initialized, one can simply do the following:

```
someTable = CryAction.LoadXML( definitionFileName, dataFileName );
```

In practice, I would suggest that we keep definition files with the scripts that use them, and the data files in a directory that is not the Scripts directory; which directory this is, is yet to be determined.

And to save a table from Lua:

```
CryAction.SaveXML( definitionFileName, dataFileName, table );
```

Data types

The following data types are available, and can be set wherever a "type" attribute is present in the definition file:

- Vec3 - in XML it's

  ```
  "1,2,3"
  ```

  , and in Lua it's

  ```
  {x=1,y=2,z=3}
  ```

- float - a floating point number
- int - an integer
- string - a string
- bool - a boolean value

Example

A definition file:

```
<Definition root="Data">

        <Property name="version" type="string"/>

        <Table name="test">
                <Property name="a" type="string"/>
                <Property name="b" type="int" optional="1"/>
                <Array name="c" type="string" elementName="Val"/>
                <Array name="d" elementName="Value">
                   <Property name="da" type="float"/>
                   <Property name="db" type="Vec3"/>
                </Array>
                <Property name="e" type="int"/>
        </Table>

</Definition>
```

A data file for the above definition:

```
<Data version="Blag 1.0">
        <test
                a="blag"
                e="3">
                <c>
                   <Val value="blag"/>
                   <Val value="foo"/>
                </c>
                <d>
                   <Value da="3.0" db="2.1,2.2,2.3"/>
                   <Value da="3.1" db="2.1,2.2,2.3"/>
                   <Value da="3.2" db="3.1,3.2,2.3"/>
                </d>
        </test>
</Data>
```

Enums

For properties of type string, an optional Enum definition has been added. Property values will be validated against the Enum.

Example:

```
<Property name="view" type="string">
        <Enum>
                <Value>GhostView</Value>
                <Value>ThirdPerson</Value>
                <Value>BlackScreen</Value>
        </Enum>
</Property>
```

Enum support for other datatypes can be added, if necessary.