The ArtiSynth File Framework

John E. Lloyd

August 13, 2007

1 Introduction

This document describes the framework for file I/O and persistent storage within ArtiSynth, along with recommended means for implementation. The framework is intended to address the following requirements:

- *Human readability:* Files should be easy to read and understand, at least in smaller instances.
- *Ease of implementation:* As new models and components are added, it should be easy for developers to add the required file support.
- *Hierarchical:* The framework should reflect the hierarchical organization of models and components.
- *Completeness:* The information provided should allow components to be faithfully reconstructued to the state they were in at write time.
- *Ease of parsing:* It should be possible to parse files "on-the-fly", with only one token of lookahead (which is consistent with the capabilities of `ReaderTokenizer`).

2 Design Overview

To meet the above requirements, we have adopted a recursive object-oriented approach:

> All ArtiSynth components requiring persistent storage should be able to read and write themselves to a text stream. In doing so, they may recursively call upon the read/write capabilities of their subcomponents.

Components requiring persistent storage should implement the interface `Scannable`, which contains the following two methods:

```
void write (PrintWriter pw, NumberFormat fmt, Object ref)
   throws IOException;

boolean scan (ReaderTokenizer rtok, Object ref)
   throws IOException;
```

The `write` method writes the component's internal values to a `PrintWriter`, making optional use of numeric format information supplied by a `NumberFormat`.

The `scan` method does the reverse operation: it reads tokens from a `ReaderTokenizer` and uses these to set the component's internal values (for more information on `ReaderTokenizer`, see Section 11). All tokens read should either be consumed or returned to the token stream using `ReaderTokenizer.pushBack()` (note that only one token of push-back is allowed). Both `write` and `scan` may utilize a reference object (`ref`) to identify and resolve external references (Section 7).

Whatever is written by the `write` method should be readable by the `scan` method and should set the component's configuration to be the same as when `write` was called. Any aspects of a component's configuration that are not specified by the persistent storage stream should be set to the same default values that arise when the component is constructed using a no-args constructor.

A component's `write` and `scan` methods may utilize the `write` and `scan` methods of its sub-components, as illustrated by the following examples:

```
void write (PrintWriter pw, NumberFormat fmt, Object ref)
   throws IOException
 {
   ... write information not related to sub-components ...
   for (each sub-component c)
    { ... write field and type information for c ...
```

```
      c.write (pw, fmt, ref);
    }
 }

scan (ReaderTokenizer rtok, Object ref)
   throws IOException
 {
   ... read information not related to sub-components ...
   while (true)
    { ... read any necessary field and type information
          for sub-component c ...
      allocate sub-component c;
      c.scan (rtok, ref);
     }
 }
```

Before scanning a sub-component, the parent must instantiate it, most likely using a no-args constructor. In cases where a sub-component may have several possible class types, the required type may be specified by a preceding *type identifier*, as described in Section 6.

3 Definitions

In this document, a *component* means any ArtiSynth object which implements `Scannable`, including, but not limited to, instances of `artisynth.core.modelbase.ModelComponent`.

The text representation of a component is called its *signature*.

4 Basic Syntax

If a component's signature contains more than one text field, it is recommended that it begin and end with square brackets (`[ ]`). These provide convenient delimiters, particularly when the number of fields within the signature is unknown.

Fields within a signature can be identified either by their explicit position within the signature, or by an identifying field tag consisting of a *field name* followed by an equal sign (`=`):

fieldName =

Field names should consist of the same characters used for Java identifiers.

Explicit positioning has the advantage of brevity (since a field tag is not needed) and so may be preferable for information that must always be present. It is also particularly appropriate for components which are simply homogeneous lists of other components (for example, a list of particles). (For more discussion of component lists, see Section 10).

On the other hand, field tags can be used to present information in any order, and default values do not need to be specified. A field tag also provides implicit documentation about the field's contents. However, excessive use of field tags can result in larger files and greater input parsing time.

Explicit positioning and field tagging can be combined. For example, the signature of a `Particle` component (in `artisynth.core.mechmodels`) consists of its mass, followed by the x, y, and z coordinates of its position, optionally followed by the x, y, and z coordinates of its velocity (if the velocity is non-zero), followed by tagged fields indicating other properties which are different from their default values:

[*mass pos.x pos.y pos.z* $\{$ *vel.x vel.y vel.z* $\}$ $\{$ *non-default-fields* ... $\}$]

The mass, position, and optional velocity information is explicitly placed, while the rest is tagged. A typical particle description looks like this:

```
[    20.0000    10.0000     0.0000    20.0000
  name="foo"
  renderProps=
  [ pointStyle=SPHERE
    sphereRadius=2.0
    pointColor=[ 0.000 1.000 0.000 ]
  ]
]
```

Here the mass is 20, the position is (10, 0, 20), and the velocity is zero since it is not explicitly given. The field names `name` and `renderProps` identify name and render property information that is different from the default values. Render properties are specified by instances of `artisynth.-core.render.RenderProps`, which is itself a `Scannable` component with a signature delimited by square brackets.

5 Indentation

If a component's signature is long (e.g., more than 60 characters), it is recommended that it be split across multiple lines, in order to keep the overall output readable. In particular, a separate line should be used for each field tag. Furthermore, if a component *is* written across multiple lines, then it should

1. place it's leading '`[`' on a new line;
2. indent all subsequent intermediate lines by an extra two spaces;
3. place its final closing '`]`' on an unindented separate line of its own.

In the example above, this is done for both the particle component and for its render properties.

The only effective way to increase indentation is to use a subclass of `PrintWriter` which supports indentation, such as `IndentingPrintWriter` located in `maspack.util`. This class keeps track of the current indentation level, and allows it to be adjusted using the methods getIndentation(), setIndentation(), and addIndentation(). Anything written *after* a new line character will automatically be indented to the current indentation level.

A component's `write` method has no control over whether the `PrintWriter` it receives is an instance of `IndentingPrintWriter`, but it can check this dynamically, and then adjust the indentation if possible. `IndentingPrintWriter` supplies a static method `addIndentation()` which will do this in one call:

```
void write (PrintWriter pw, NumberFormat fmt, Object ref)
   throws IOException
 {
   pw.print ("[ ");  // don't indent leading bracket
   // boost indentation:
   IndentingPrintWriter.addIndentation (pw, 2);
   ... write out components ...
   // decrease indentation:
   IndentingPrintWriter.addIndentation (pw, -2);
   pw.println ("]"); // don't indent trailing bracket
 }
```

6 Type Identification

In general, when a parent component scans in a sub-component, it allocates an instance of that component (usually with a no-args constructor) and then calls it's scan() method. However, it may be that the component's type is not known in advance. For example, the `LinealSpring` components used by `artisynth.core.mechmodels.MechModel` may be subclassed to implement various types of non-linear springs. When reading these in, `MechModel` needs to know the exact type in order to create an initial component instance.

When a component may be one of several class types, that component's signature may be preceded by a `type identifier` which specifies the class type for the component. The type identifier may be either a fully qualified Java class name, or it may be an abbreviated `alias` for that name. Class aliases are stored in an internal table managed by `artisynth.core.util.-ClassAliases`, which supplies the following static methods:

```
Class ClassAliases.resolveClass (String nameOrAlias);

String ClassAliases.getAliasOrName (Class cls);
```

`resolveClass` tries to find a class associated with a name, interpreting it first as an alias, and second as a full class name. `getAliasOrName` performs the inverse operation: it tries to locate an alias for a specified class, and otherwise returns the full class name.

The `resolveClass` method may be used to determine the type of a sub-component. Assuming that the sub-component's signature is delimited by square brackets (`[ ]`), the required code might look something like this:

```
void parentComponent.scan (
   ReaderTokenizer rtok, Object ref)
   throws IOException
 {
   ...
   Class componentType;
   if (rtok.nextToken() == '[')  // no type identifier
    { componentType = defaultComponentType;
      rtok.pushBack();
    }
   else if (rtok.isWordToken())  // has type identifier
```

```
      { componentType = ClassAliases.resolveName (rtok.sval);
        if (componentType == null)
         { ... error ...
         }
      }
     ModelComponent c = componentType.newInstance();
     c.scan (rtok, ref);
     ...
  }
```

Similarly, a parent's `write` method must output a type identifier for any sub-component which is not of the default type. This can be done using `ClassAliases.getAliasOrName`:

```
void parentComponent.write (
   PrintWriter pw, NumberFormat fmt, Object ref)
   throws IOException
 {
   ...
   ModelComponent c;
   if (!(c instanceof defaultComponentType))
    { // need a type identifier
      String typeId = ClassAliases.getAliasOrName(c);
      pw.print (typeId + " ");
    }
   c.write (pw, fmt, ref);
   ...
 }
```

For an example of what type identifiers look like, here is the signature for a list of `LinealSpring` components that might form part of a `MechModel`:

```
[ p0 p1 1 2 0 ]
[ p2 p3 3 2 0 ]
artisynth.models.jaw.Muscle [ p5 p6 1 2 4 5 ]
artisynth.models.jaw.Muscle [ p4 p7 1 3 4 6 ]
[ p2 p4 2 2 0 ]
```

The entries on lines 1, 2, and 5 describe the default `LinealSpring` type, whereas lines 3 and 4 describe instances of a sub-class of `LinealSpring` associated with the full class name `artisynth.models.jaw.Muscle`. This sub-class also has a different signature; in particular, it has more text fields.

If the alias `Mus` were defined for `artisynth.models.jaw.Muscle` (by adding it to the table maintained by `ClassAliases`), then this could be used as a type identifiers and the above signature would be shortened to

```
[ p0 p1 1 2 0 ]
[ p2 p3 3 2 0 ]
Mus [ p5 p6 1 2 4 5 ]
Mus [ p4 p7 1 3 4 6 ]
[ p2 p4 2 2 0 ]
```

7 External References

Some components may contain references to other external components. In such cases, address information for these external components needs to be embedded in the signature and handled by the `scan` and `write` methods. The actual reference is obtained from the `ref` argument.

If the external component is an instance of `artisynth.core.modelbase.-ModelComponent`, then the ArtiSynth component path name structure can be used to provide reference addresses. If the `ref` argument is assumed to be a `CompositeComponent` type which contains the referenced components within its hierarchy, then the `write` method can obtain addresses for each external component `c` using either

```
CompositeComponentBase.getPathName (
   (CompositeComponent)ref, c);
```

or

```
CompositeComponentBase.getCompactPathName (
   (CompositeComponent)ref, c);
```

The former returns a standard component path name while the latter returns a more compact version which is harder to read but faster to parse.

The `scan` method must in turn take these component path names and turn them back into component references. This can be done directly using

```
CompositeComponentBase.findComponent (
   (CompositeComponent)ref, path);
```

but it is probably easier to use the convenience routine

```
CompositeComponentBase.scanReference (
   rtok, (CompositeComponent)ref);
```

which gets the path directly from the `ReaderTokenizer`.

A simple of example of this mechanism is afforded by the `LinealSpring` component, which contains references to two `Point` components. The signature for a default `LinealSpring` contains path names for both these references, plus three numbers giving the spring's stiffness, damping and rest length. Using compact path names, it will look something like this:

```
[ m0/p1 m0/p2  5 1 0 ]
```

Ignoring error handling, the corresponding `write` and `scan` methods could be implemented as follows:

```
void write (PrintWriter pw, NumberFormat fmt, Object ref)
   throws IOException
 {
   CompositeComponent ancestor = (CompositeComponent)ref;
   pw.print ("[ ");
   pw.print (CompositeComponentBase.getCompactPathName (
                ancestor, myPnt0)+" ");
   pw.print (CompositeComponentBase.getCompactPathName (
                ancestor, myPnt1)+" ");
   pw.print (" " + fmt.format (myStiffness) +
         " " + fmt.format (myDamping) +
         " " + fmt.format (myRestLength));
   pw.println ("]");
 }

void scan (ReaderTokenizer rtok, Object ref)
   throws IOException
 {
   CompositeComponent ancestor =
      CompositeComponentBase.castRefToAncestor(ref);
   rtok.scanToken ('[');
   myPnt0 = (Point)CompositeComponentBase.scanReference (
                      rtok, ancestor);
   myPnt1 = (Point)CompositeComponentBase.scanReference (
                      rtok, ancestor);
   myStiffness = rtok.scanNumber();
   myDamping = rtok.scanNumber();
   myRestLength = rtok.scanNumber();
   rtok.scanToken (']');
 }
```

A referenced component which has a legitimate null value should be assigned the reserved path name `"null"`. This is done automatically by `getPathName` and `getCompactPathName`, and, correspondingly, `findComponent` and `scanReference` will return null for the path name `"null"`.

If a `scan` method cannot locate a referenced component corresponding to a particular path name, or if the located component has an improper type, then it should throw a `ReferenceNotFoundException` (which is a sub-type of `IOException`). By catching such exceptions, parent components have the option of discarding incomplete objects while reading in those which are well-formed.

It should be noted that for external referencing to work properly, components need to be written and read in the proper order. In particular, any referenced object needs to appear in the input *before* any object which references it. This is why `MechModel` writes out its `Point` components before its `LinealSpring` components.

The discussion in this section assumed that externally referenced objects are instances of `ModelComponent`, but that is not strictly necessary. All that is required is that `write` and `scan` have some way to obtain or resolve the signature address of an external component. The *ref* argument is an optional handle which can be used to assist in this mapping.

8 Scanning and writing property information

Components which export properties (i.e., implement the `HasProperties` interface) and which maintain those properties using a `PropertyList` structure, can use several convenience methods supplied by that structure for scanning and writing property values:

```
   boolean scanProp (
      HasProperties host, ReaderTokenizer rtok)
      throws IOException

   void writeProps (
      HasProperties host, PrintWriter pw, NumberFormat fmt)
      throws IOException

   boolean writeNonDefaultProps (
      HasProperties host, PrintWriter pw, NumberFormat fmt)
      throws IOException

   boolean writeNonDefaultProps (
      HasProperties host, PrintWriter pw, NumberFormat fmt,
      String leadPrefix)
      throws IOException
```

The first, `scanProp`, checks the input stream for a field-tagged property declaration of the form

propertyName = *value*

where *propertyName* is the name of one of the properties exposed by *host* and *value* is that property's signature. If `scanProp` finds a declaration for a known property, it sets that property to the supplied value and returns true. Otherwise, it returns false. `scanProp` can be called in a loop to read in a whole set of property values. For instance, the `scan` routine for some ArtiSynth components concludes with

```
   ...
   while (getAllPropertyInfo().scanProp (this, rtok))
      ;
   rtok.scanToken (']');
```

which parses an unstructured list of property values. The property list itself is obtained by calling `getAllPropertyInfo()`, which ensures that the method will work in sub-classes which might maintain a different property list.

The three `writeProps` methods can be used to output field-tagged property declarations. The first writes a declaration for all properties contained in the list for which automatic writing is enabled. Automatic writing is enabled for properties by default, but can be enabled or disabled (using the property descriptor's `setAutoWrite(enable)` method) to accommodate situations where it is written by some other means, or perhaps not written at all.

The second method, `writeNonDefaultProps`, writes out declarations for all properties in the list for which automatic writing is enabled *and* which are not equal to their default value. This is useful for reducing file output size (and input parsing time) in situations where only a few properties are likely to have values different from their defaults. The method returns false if no properties meeting this criterion were actually written.

The third method provides the same functionality as the second, except that if any properties are actually written, the output is preceded by a specific *leadPrefix*, and the indenting level is increased by 2 (if possible), using `IndentingPrintWriter.addIndentation()`. This helps facilitate output formatting. For example, the `write` method for the `Particle` component terminates with

```
...
if ((getAllPropertyInfo().writeNonDefaultProps (
   this, pw, fmt, "\n  "))
 { IndentingPrintWriter.addIndentation (pw, -2);
 }
pw.println ("]");
```

where again the property list is obtained using `getAllPropertyInfo()`.

Using the `HasProperties` support routines to write and scan property information has the advantage that properties can be added to a component without changing the file format, and, indeed, without having to even change the `write` or `scan` code.

9 Scanning components subclassed from ModelComponentBase

`ModelComponentBase` provides a default implementation of `scan()`:

```
public void scan (ReaderTokenizer rtok, Object ref)
   throws IOException
 {
   CompositeComponent ancestor =
      CompositeComponentBase.castRefToAncestor (ref);
   rtok.scanToken ('[');
   while (rtok.nextToken() != ']')
    { rtok.pushBack();
      if (!scanItem (rtok, ancestor))
       { throw new IOException (
            "Unexpected token: " + rtok);
       }
    }
 }
```

Individual sub-component values are scanned in using `scanItem()`, which reads the next token from the input stream uses this to identify the input. If the input is recognized, it is scanned and the method returns `true`. If the input is not recognized, the token is pushed back to the input stream and the method returns `false`.

The default implementation of `scanItem()` checks for and scans property values, which are assumed to be in the form described in Section 8:

propertyName = *value*

where `propertyName` is the name of a property exported by the component. Each call to `scanItem()` scans one property value.

In order to extend `scan` so that it reads in more than property values, the user may override `scanItem` and/or `scan` itself.

An overridden version of `scanItem()` should look like this:

```
   protected boolean scanItem (
     ReaderTokenizer rtok, CompositeComponent ancestor)
     throws IOException
   {
     if (super.scanItem (rtok, ancestor))
      { // input recognized by super class
        return true;
      }
     else
      { int token = rtok.nextToken();
if (token is recognized)
        { ... finish scanning input ...;
   return true;
```

```
         }
else
         { rtok.pushBack();
           return false;
         }
      }
   }
```

At the time of this writing, `CompositeComponentBase` and several components within the `mechmodels` package override `scanItem()`.

Many components override `scan`, typically to read input which is not specified using tagged fields. For example, the `scan` method for `Particle` looks like this:

```
   public void scan (ReaderTokenizer rtok, Object ref)
      throws IOException
    {
      CompositeComponent ancestor =
         CompositeComponentBase.castRefToAncestor(ref);
      rtok.scanToken ('[');
      double mass = rtok.scanNumber();
      setMass (mass);
      myState.pos.x = rtok.scanNumber();
      myState.pos.y = rtok.scanNumber();
      myState.pos.z = rtok.scanNumber();
      if (rtok.nextToken() == ReaderTokenizer.TT_NUMBER)
       { myState.vel.x = rtok.nval;
         myState.vel.y = rtok.scanNumber();
         myState.vel.z = rtok.scanNumber();
       }
      else
       { myState.vel.setZero();
         rtok.pushBack();
       }
      while (rtok.nextToken() != ']')
       { rtok.pushBack();
         if (!scanItem (rtok, ancestor))
          { throw new IOException (
               "Unexpected token: " + rtok);
```

```
            }
        }
    }
```

This scans the particle signature described in Section `sec:fields`, which consists of mass, position, and optional velocity information, followed by other quantities expressed as tagged fields. `scanItem()` is not overridden.

10 Scanning ComponentLists

A ComponentList is a ModelComponent which implements a list of components, each of which is an instance of a specific default type associated with the list. ComponentList provides default implementations of `scan()` and `write()`.

The signature produced by `write()` is delimited by square brackets (`[ ]`) and starts with all non-default property values (in field-tagged format, as described above), followed by all list components in index order. List components are themselves assumed to have signatures which are delimited by square brackets. Any list component which is a sub-class of the default component type (as returned by `getDefaultType()` will be preceded with a type identifier, as described in Section 6.

The `scan()` method will automatically read any component list with the same signature as that produced by `write()`.

11 The ReaderTokenizer class

This section gives a quick overview of `ReaderTokenizer` and how it may be used to parse input. `ReaderTokenizer` provides the same functionality as `java.io.StreamTokenizer`, with enhancements which allow it to read numbers with exponents, read integers formatted in hex, and save and restore settings.

The tokenizer reads characters from a `Reader`, removes comments and whitespace, and parses the remainder into tokens. One token is parsed for each call to `nextToken()`. The following tokens may be generated:

- *Numeric values:* a floating point number, or decimal or hexadecimal integer. Number recognition is enabled by default but can be controlled using the method `parseNumbers()`.

- *Words:* a sequence of word characters not beginning with a digit. By default, word characters consist of alphanumerics, underscores (_), and any character whose Unicode value exceeds 0xA0.
- *Quoted strings:* a sequence of characters enclosed by a pair of quote characters. Be default, `'` and `"` are enabled as quote characters.
- *End of line:* an end-of-line character. Recognition of this token is optional and is enabled using the method `eolIsSignificant()`.
- *End of file:* the end of input from the underlying stream.
- *Single character token:* Any single character which is not used to form one of the above tokens.

The default comment character is (`#`), which discards all characters between itself and the next end-of-line. C and C++ comment styles are also supported.

The basic usage paradigm for `ReaderTokenizer` is:

- Read a token;
- Interpret this token and act accordingly;
- Read another token ...

We first give a simple example that reads in pairs of words and numbers, such as following:

```
foo 1.67
dayton 678
thadius 1e-4
```

This can be parsed with the following code:

```
HashMap<String,Double> wordNumberPairs =
   new HashMap<String,Double>();
ReaderTokenizer rtok =
   new ReaderTokenizer (
      new FileReader ("foo.txt"));
while (rtok.nextToken() != ReaderTokenizer.TT_EOF)
```

```
{ if (rtok.ttype != ReaderTokenizer.TT_WORD)
   { throw new IOException (
        "word expected, line " + rtok.lineno());
   }
  String word = rtok.sval;
  if (rtok.nextToken() != ReaderTokenizer.TT_NUMBER)
   { throw new IOException (
        "number expected, line " + rtok.lineno());
   }
  Double number = new Double(rtok.nval);
  wordNumberPairs.put (word, number);
}
```

The application uses `nextToken()` to continuously read tokens until the end of the file. Once a token is read, its type is inspected using the `ttype` field. If the type is inappropriate, then an exception is thrown, along with the tokenizer's current line number which is obtained using `lineno()`. The numeric value for a number token is stored in the field `nval`, while the string value for a word token is stored in the field `sval`.

Once a token has been read, it can be returned to the tokenizer using `pushBack()`, which will then cause it to be returned by the next call to `nextToken()`. Only one token of push-back is supported.

More detailed information about `ReaderTokenizer` can be found in the class documentation.

Note: ArtiSynth application using `ReaderTokenizer` should set '.' and '/' to word characters, to enable the correct handling of path names. This can be done as follows:

```
ReaderTokenizer rtok;
...
rtok.wordChars ("./");
```