A BNF Specification for the Open Inventor ASCII File Format

Paul S. Strauss

1.0 Introduction

This document is intended to be a reasonably formal description of the ASCII file format defined by Open Inventor version 2.0. It describes the global key words and symbols of the format and goes into some detail on the specific formats of the class types built into Open Inventor.

The following conventions are used in the BNF descriptions:

- Literal keywords and symbols are in **boldface**.
- Names in *italics* are types defined in other rules.
- The ∅ symbol represents an empty production.
- Any word ending in "name" or "Name" is defined to be any valid C identifier, except in the case of "defName." These names (used for defining object names for instancing or for application look-up) need not be valid C identifiers, but must not gein with a digit or contain white space, control characters, or any of these characters: `' " \ { }`.

☞ **Important note:** Any developer using Inventor is free to extend the set of node, engine, and field classes. Therefore, most of the BNF "rules" described in this document do not necessarily apply to all Open Inventor files. For example, a developer could create a node whose data is represented without using Inventor fields; the file format would be defined by that developer. However, since most developers will use standard Inventor fields for storing data in nodes and engines, the rules here usually apply.

2.0 Files

An Open Inventor file may contain any number of node or path descriptions; the contents of a file are defined by the application. The Inventor file-reading code is usually used to read one or more nodes, but there is an additional call to read in a path, if that is what is expected in the file.

2.1 Headers

The first line of any Inventor file is a header string that identifies the format of the file. Recognized ASCII format header strings are (in version 2.0):

```
#Inventor V2.0 ascii

#Inventor V1.0 ascii
```

Note that the header must begin at the first character of the first line, and the header must appear exactly as above, with no extra white space. Currently, nothing is allowed between the end of the header string and the newline that ends the first line. However, this restric-

tion will likely be removed in a later version to allow additional file-typing information to be stored and retrieved.

2.2 Comments and White Space

Any other line or part of a line that begins with a pound sign is treated as a comment and is ignored up to the next newline. Comments are not retained once a file has been read in.

White space (spaces, tabs, newlines, and form feeds) is used to separate tokens in ASCII files. Multiple white space characters are treated as a single space. Comments may appear wherever white space is legal.

White space and comments within quoted strings are treated like any other string characters.

2.3 Imbedded Data

Inventor data may be imbedded in larger data files. The only requirements are that the Inventor data begin with a header comment and that the Inventor file-reading methods are called to read the correct amount and type of data from the file, starting at the first character of the header.

3.0 Nodes

The syntax for a node is as follows:

node → *nameSpec* nodeTypeName { *fieldSpecs fieldData children* }
| **USE** defName

nameSpec → ∅
| **DEF** defName

fieldSpecs → ∅
| **fields [** *fieldDescriptions* **]**

fieldDescriptions → ∅
| *fieldDescription*
| *fieldDescription* **,** *fieldDescriptions*

fieldDescriptions → fieldTypeName fieldName

fieldData → ∅
| *field fieldData*

children → ∅
| *node children*

Note that the name in the "node" rule is the name of the node class. For built-in Inventor nodes, this name is the portion of the class name after the "So" prefix. Also note that the name after the **USE** keyword must correspond to the name of a node previously assigned with **DEF** in the same file.

4.0 Engines

The syntax for an engine is as follows:

engine → *nameSpec* engineTypeName { *inputSpecs outputSpecs inputData* }
| **USE** defName

inputSpecs → ∅
| **inputs [** *fieldDescriptions* **]**

outputSpecs → ∅
| **outputs [** *fieldDescriptions* **]**

inputData → ∅
| *field inputData*

Engine class names follow the same rule as node class names.

5.0 Global Fields

The syntax for a global field is as follows:

globalField → *nameSpec* **GlobalField { type** name *field* **}**
| **USE** defName

Note that the specified field must be of the type named after the **type** keyword. The field type is the class name of the field without the "So" prefix.

6.0 Fields

The discussion of fields in this section applies to the following:

- Fields in nodes
- Input fields in engines
- Fields in global fields

Each field is defined by the container (node, engine, or global field) to be of a specific type. This type determines two things:

- Whether the field contains a single value or multiple values
- The format of the data for that field

Regardless of the container type or the field type, all fields use the following general syntax:

field → fieldName *valueSpec connectionSpec*

valueSpec → **~**
| *value*
| *value* **~**

connectionSpec → ∅
| **=** *connectionField*

connectionField → *nodeField*
| *engineInput*
| *engineOutput*
| *globalFieldField*

nodeField → *node* **.** fieldName

engineInput → *engine* **.** inputName

engineOutput → *engine* **.** outputName

globalFieldField → *globalField* **.** fieldName

Note: the use of ignore flags (the **~** in the above rules) is usually discouraged.

6.1 Single-Value Fields

A single-value field contains the specification of one value whose format depends on the type of the field:

value → *singleValue*

The following are the formats for the supported Inventor single-value fields.

6.2 SFBitMask

singleValue → name
| **(** *bitMasks* **)**

bitMasks → flagName
| flagName | *bitMasks*

6.3 SFBool

singleValue → **FALSE**
| **TRUE**
| **0**
| **1**

6.4 SFColor

Colors are specified as red, green, blue triples:

singleValue → float float float

6.5 SFEngine

singleValue → *engine*

6.6 SFEnum

singleValue → enumName

6.7 SFFloat

singleValue → float

6.8 SFImage

An image is specified by three integers, representing the width in pixels, height in pixels, and the number of pixel components, followed by the pixel data. Each pixel is represented as a single unsigned number in decimal, octal, or hexadecimal notation.

singleValue → int int int unsignedInt ...

6.9 SFLong

singleValue → int

6.10 SFMatrix

Matrices are specified as 16 floating-point numbers, in row-major order:

singleValue → float float float float float float float float
float float float float float float float float

6.11 SFName

singleValue → name

6.12 SFNode

singleValue → *node*

6.13 SFPath

singleValue → *path*

6.14 SFPlane

A plane is specified as the normal to the plane (3 floats), followed by the distance of the plane from the origin in the direction of the normal:

singleValue → float float float float

6.15 SFRotation

A rotation is specified as the axis vector (3 floats), followed by the rotation angle in radians:

singleValue → float float float float

6.16 SFShort

singleValue → int

6.17 SFString

A string is defined to be a sequence of ASCII characters within double quotes. If the string does not contain any white space, the quotes may be omitted. A double quote may be included within a quoted string by preceding it with a backslash.

singleValue → string

6.18 SFTime

A relative time is specified as a double-precision floating-point value, indicating the number of seconds. Absolute times are specified as seconds since 00:00:00 GMT, January 1, 1970.

singleValue → float

6.19 SFTrigger

Trigger fields contain no values:

singleValue → ∅

6.20 SFULong

singleValue → int

6.21 SFUShort

singleValue → int

6.22 SFVec2f

singleValue → float float

6.23 SFVec3f

singleValue → float float float

6.24 SFVec4f

singleValue → float float float

6.25 Multiple-Value Fields

A multiple-value field contains the specification of one ore more values whose formats depends on the type of the field.

value → *singleValue*
| **[** *values* **]**

values → ∅
| *singleValue*
| *singleValue* **,** *values*

The value formats for the specific multiple-value fields are identical to those for the corresponding single-value fields. Supported multiple-value field types are:

MFBitMask, MFBool, MFColor, MFEngine, MFEnum, MFFloat, MFLong, MFMatrix, MFName, MFNode, MFPath, MFPlane, MFRotation, MFShort, MFString, MFTime, MFULong, MFUShort, MFVec2f, MFVec3f, MFVec4f

7.0 Paths

A path through a chain of nodes is represented as:

path → **Path** { *node* int int ... }

The node is the first node in the path chain. The first integer is the number of nodes following this node in the chain. The remaining integers are the child indices of these nodes. That is, each integer is the index (starting at 0) of each node as a child in the previous node in the chain.