GTO: The Kitchen Sink of Data

File Format, Protocols, and Utilities.

Jim Hourihan, Tweak Software

Table of Contents

New BSD License

1 Installation

The GTO source code is built using CMake.

2 Overview

Historically, GTO format’s primary usage is storage of static geometric data (cached geometry). As such, the types of data you might find in a GTO file are things like polygonal meshes, various types of subdivision surfaces, NURBS or UBS surfaces, coordinate systems, hierarchies of objects, material bindings, and even images.

From a historic point of view, the GTO file format is most closely related to the original inventor file format, the Stanford PLY format and the PDB particle format. Like the Wavefont PDB file format, there are a limited number of simple GTO data types (float, int, string, boolean). Like the inventor file format, a GTO can hold an entire transformation hierarchy including geometric leaf nodes. Like the PLY format, the GTO format can contain an arbitrary amount of data per primitive. Most importantly however, the GTO file format

is intended to be very OBJ-like; its relatively easy to read and write and easy to ignore data you don't want or know about.

GTO files can be either binary or text files. Binary files are the preferred format for large data sets. The GTO text format is intended to be human readable/editable; the syntax is simple and concise. The text format is useful when storing "bag of parameters" files and similar data.

The binary file is either big or little endian on disk, but should be readable on any platform.

The GTO reader can be use libz to read and write compressed files natively. We find that compressed GTO files created by most 3D programs are approximately 60% leaner than uncompressed files.

GTO files conceptually contain *objects* which are optionally composed of nested namespaces called *components*. Components are further composed of *properties*. A property contains an array of one of the predefined data types with up to a four dimensional "shape". For example, you might have an object which looks something like this:

```
Object "cube"
    Component "points"
        Property float[3][8] "position"
        Property float[1][8] "mass"
    Component "elements"
        Property byte[1][8] "type"
        Property short[1][8] "size"
    Component "indices"
        Property int[1][32] "vertex"
```

Using the terminology above, the object "cube" contains five properties: **position**, **mass**, **type**, **size**, and **vertex**. The **points** component describes the points that make up the cube vertices. Each point has a position and mass stored in properties of the same name. The position property data is composed of eight float[3] data items (or 8 3D points). The **mass** property is composed of a 8 scalar floating point values (one for each point).

The **elements** component contains two properties. **type** indicates the type of the element (for example, triangle, quad, or triangle strip). In this case the elements might all be quads. **size** indicates the number of vertices in each of the eight faces (elements) of the cube – (4 for a cube). The **vertex** property of the **indices** component contains the actual indices: 4 per face for a total of 32.

Of course you could store much more data with the cube object if you wanted to. For example, if you wanted velocity or color per point, this would be another property in the **points** component.

The meaning of this data is another story altogether. Its all handled by protocol. One application may store things in the GTO file that another application has no method of interpreting even though it can read that data and modify it. In the example above, you need to know to expect that polygonal data is stored in the given properties. The same data could be stored with different property names and a more complex layout. (The "polygon" protocol described later in this document is different and more involved than the above example.)

GTO was (and still is) used by a nunber of film post production facilities for geometry caching. 3D scenes are evaluated and the final geometry is written into GTO files which are later consumed by a renderer (e.g. RenderMan).

A newer geometry caching format called Alembic was introduced by ILM in 2010 which has a similar purpose and has taken over that role.

2.1 New in Version 4

Version 4 adds two new features: nested components and property types with up to four dimensions.

Version 3 files always had the same structure of objects.component.property. Nested components allows any number of component names between the object and property:

```
object.component1.component2.component3.property
```

In text GTO files this looks like this:

```
object
{
    component1
    {
        component2
        {
            component3
            {
                property ...
            }
        }
    }
}
```

Where version 3 allowed only a single "width" for properties, version 4 allows up to four dimensions:

```
float[4,10,20,30][3] = [ ... ]
```

The above declares 3 4x10x20x30 float data object.

3 Binary Format

The GTO file has six major sections which appear in the following order.

1. **Header** (Gto::Header). The header structure contains the GTO magic number (used to determine endianness), the version of the GTO specification that the file was written as, and the number of top level objects in the file. There is one instance of a header in the file. Finally, the header indicates how many strings are in the string table.

```
Magic = 0x0000029f;
Cigam = 0x9f020000; // means the file is opposite endianess

struct Header
```

```
{
    uint32      magic;
    uint32      numStrings;
    uint32      numObjects;
    uint32      version;
    uint32      flags;              // reserved;
};
```

2. **String Table**. After the header, null terminated strings are written in the file. The order of these strings is important. All names and string properties store indices into the string table instead of actual strings. In order to read the file properly, the string table must be available until the file is completely read. (Unless you don't care about any strings!)

   The index number refers the string number in the table not its byte offset. So the string index 9 (for example) refers to the 10th string in the table (string index 0 is the first string in the table).

3. **ObjectHeader** (Gto::ObjectHeader). The object header indicates what kind of protocol to use to interpret it, the object name and the number of components. (More on the **object** protocol later). The name – like all strings in the GTO file – is stored as a string table entry. If the file header indicated N objects in the file, there will be N ObjectHeaders.

```
struct ObjectHeader
{
    uint32      name;               // a string table index
    uint32      protocolName;       // a string table index
    uint32      protocolVersion;
    uint32      numComponents;
    uint32      pad;                // unused
};
```

4. **ComponentHeader** (Gto::ComponentHeader). Like the ObjectHeaders the ComponentHeaders will appear together for all objects in order. The component header indicates the number of properties in the component and the name of the component.

```
enum ComponentFlags
{
    Transposed  = 1 << 0,
    Matrix      = 1 << 1,
};

struct ComponentHeader
{
    uint32      name;               // a string table index
    uint32      numProperties;
    uint32      flags;
    uint32      interpretation;     // a string table index
```

```
    uint32      childLevel;         // nesting level
};
```

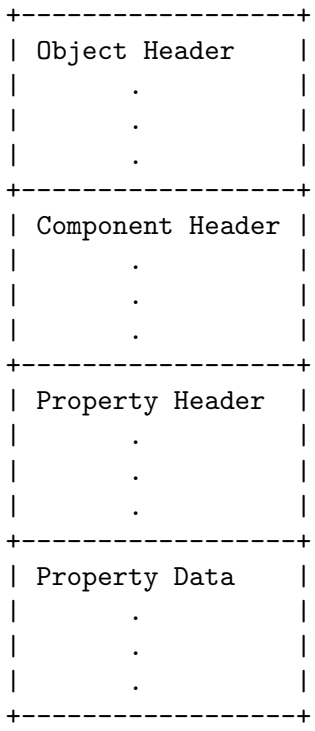
5. **PropertyHeader** (Gto::PropertyHeader). The PropertyHeaders, like the object and component headers, appear en masse in the file. The PropertyHeader contains the name, size, type, and dimension of the property.

```
enum DataType
{
    Int,                // int32
    Float,              // float32
    Double,             // float64
    Half,               // float16
    String,             // string table indices
    Boolean,            // bit
    Short,              // uint16
    Byte                // uint8
};


struct Dimensions
{
    uint32 x;
    uint32 y;
    uint32 z;
    uint32 w;
}

struct PropertyHeader
{
    uint32      name;           // string table index
    uint64      size;
    uint32      type;           // DataType enum value
    Dimensions  dims;
    uint32      interpretation; // string table index
};
```

6. **Data**. The last section of the file contains all of the property data. The beginning and end of a properties data are not marked. The size must be consistant with the description of the property used in the PropertyHeader.

In (Text) diagram form the file looks something like this:

```
+------------------+
| File Header      |
+------------------+
| String Table     |
```



4 Text Format

As of version 3.2, GTO has a text representation in addition to the binary representation. The text representation is designed for human use; it is intended to be easy to modify or create from scratch in a text editor. It is not intended to compete with XML formats (which are typically only human readable in theory) nor is it intended to be used in place of the binary format which is much faster and more economical for storage of large data sets.

4.1 Example of a Cube Stored as a Text GTO

Here's the example from the overview section: a cube stored using the "polygon" protocol:

```
GTOa (4)

# this is a comment

cube : polygon (2)
{
    points
    {
        float[3] position = [ [ -2.5 2.5 2.5 ]
                              [ -2.5 -2.5 2.5 ]
                              [ 2.5 -2.5 2.5 ]
```

```
                                [ 2.5 2.5 2.5 ]
                                [ -2.5 2.5 -2.5 ]
                                [ -2.5 -2.5 -2.5 ]
                                [ 2.5 -2.5 -2.5 ]
                                [ 2.5 2.5 -2.5 ] ]

            float mass = [ 1 1 1 1 1 1 1 1 ]
        }

        elements
        {
            byte type = [ 2 2 2 2 2 2 ]
            short size = [ 4 4 4 4 4 4 ]
        }

        indices
        {
            int vertex = [ 0 1 2 3
                           7 6 5 4
                           3 2 6 7
                           4 0 3 7
                           4 5 1 0
                           1 5 6 2 ]
        }
    }
```

The first line of the file is an identifier to tell the parser what variety of GTO file it is: in this case `GTOa` which indicates a plain ASCII text file. Currently the parser can only handle ASCII encoding; a forthcoming version will allow UTF-8.

Objects are declared using the syntax:

```
OBJECTNAME [ : PROTOCOL [ (PROTOCOL_VERSION) ] ]
{
    ... object contents ...
}
```

The brackets enclose optional syntax. So the `PROTOCOL_VERSION` (including the parens) is optional. The `PROTOCOL` is also optional; if omitted (along with the colon) the protocol defaults to **object**. In the example, "cube" is the name of the object and "polygon" is the name of the protocol–the protocol version is 2.

Components must be declared inside the object brackets or other components. The brackets denote a *namespace* which is either an object namespace or a component namespace. Component namespaces must always be declared inside of an object or component namespace. Object namespaces can only appear at the top level of the file; in other words, objects cannot be inside another namespace.

Components are declared like this:

```
COMPONENTNAME [as INTERPRETATION]
{
```

```
    ... component contents ...
}
```

The `INTERPRETATION` can be any string. Properties can be declared inside of the component namespace optionally followed by nested component declarations. The property declaration is the most flexible; since some aspects of the property (like its size) can be determined by the parser from the property data, you can omit them.

The property syntax in its most general form is:

```
TYPE[XS,YS,ZS,WS][SIZE] PROPERTYNAME as INTERPRETATION = values ...
```

The brackets around `XS,YS,ZS,WS` and `SIZE` are literal in this case; they actually appear in the file. As you can see from the example, some of the property declaration syntax is optional. The `SIZE` can usually be determined from the values so it may be omitted. The dimensions are assumed to be `1` (or scalar) if it is omitted. The `as INTERPRETATION` section of the declaration may also be omitted.

What cannot be omitted is the `TYPE`, `PROPERTYNAME`, and the assignment of values.

4.2 How Strings are Handled in the Text Format

With the exception of keywords and type names, any string in the text GTO file can be either be quoted or non-quoted. Non-quoted strings are restricted to strings which do not represent numbers. In addition, if a string contains punctuation or whitespace, it must be quoted. For example, if the name of the object in the cube example was "four dimensional time-cube" it would have to be declared like this:

```
"four dimensional time-cube" : polygon
{
    ...
}
```

There is one additional exception: if a string is also a keyword or type name, it must be quoted. For example, here's an exceptional property declaration:

```
int "int" as "as" = 1
```

In this case the quoted string "int" is being used as the property name, but because it is also the name of a GTO type, it must be quoted. The string "as" is being used as an interpretation string and must be quoted because "as" is also a keyword in the the GTO file.

When in doubt quote.

4.3 Value Brackets

Generally, a property value and elements of the value are enclosed in brackets:

```
TYPE[DIMENSIONS] PROPERTYNAME = [ [a b ...] [d e ...] ... ]
```

In this documentation, the *value* of a property is everything to the right of the "=" and an *element* is a fixed size collection of numbers or strings. The *size* of a property is the number of elements in its value. So in the example above, the `[a b ...]` portion of the syntax is an *element*.

Bracketing the property value is optional in one circumstance: when the number of elements in the property value is one. For example, these declarations are equivalent:

```
int foo = 1
int foo = [1]
```

If the width of the type is not one (elements are not scalar), then brackets must be put around each element of the property. If the size is one but the width is not one, then the enclosing brackets are still optional:

```
int[2] foo = [1 2]
int[2] foo = [ [1 2] ]
```

If however the size of the property is greater than one, the enclosing value brackets are required:

```
# property of size 3
int[2] foo = [ [1 2] [3 4] [5 6] ]
```

To declare a property with no value use empty brackets:

```
int foo = []
```

4.4 The Size of a Property

The size of a property can be declared as part of its type declaration:

```
int[1][4] foo = [1 2 3 4]
```

In this case, "foo" contains four scalar elements. Because the size was specified, the following would be a syntax error:

```
int[1][4] foo = [1 2 3 4 5]
```

The parser would complain because five elements were supplied eventhough the property was declared as having only four. If no size is specified than the parser will determine the size from the number of elements in the value:

```
int[1] foo = [1 2 3 4 5]
```

So in this case "foo" has five elements. Note that in order to declare the size specifically, you must also declare the element width – even if the width is one. In the last example, because we did not specify the size, the declaration could also have been:

```
int foo = [1 2 3 4 5]
```

In this case it is understood that the type is actually `int[1][5]`.

Additional dimensions can be added to make e.g. a matrix:

```
float[4,4] M = [1 0 0 0
                0 1 0 0
                0 0 1 0
                0 0 0 1]
```

this can be extended up to four dimensions:

```
byte[4,1920,1080] eightBit1080pImage = [ ... ]
float[3,32,32,32] floatingPoint3DLUT = [ ... ]
```

4.5 Run Length Encoding of Values

In some cases, a value will contain many copies of an element. There is a special syntax for these cases; you can use an ellipsis to indicate that all remaining elements are identical. The ellipsis can only appear directly before the final bracket character.

There is one restriction when using this syntax: the type of the property value must be completely specified (including the size of the property) and the value must be enclosed in brackets. For example:

```
int[1][100] mass = [1 ...]
```

The ellipsis is literal (its actually in the file as three dot characters) The property "mass" will be one for all 100 elements. If the element has a width greater than one:

```
float[3][100] velocity = [ [0 0 0] ... ]
```

The ellipsis is used in place of an element. The following will *not* work:

```
float[3][100] velocity = [ [0 ...] ... ]
```

The intention here is to make all of the velocity elements `[0 0 0]`. However, this syntax is not correct and will produce a parsing error.

4.6 Syntax Reference

The grammar for the text GTO file. *INT* is an integer constant. *FLOAT* is a floating point constant, with a possible exponent part. *STRING* is either a quoted or non-quoted string. All other values are literal. Double quoted strings are keywords.

```
file::
    "GTOa" object_list
    "GTOa" ( INT ) object_list

object_list::
    object
    object_list object

object::
    STRING { component_list_opt }
    STRING : STRING { component_list_opt }
    STRING : STRING ( INT ) { component_list_opt }

component_list_opt::
    nothing
    component_list

component_list::
    component
    component_list component

component_block::
    nothing
    property_list
```

```
    component_list
    property_list component_list

interp_string_opt::
    nothing
    "as" STRING

component::
    STRING interp_string_opt { component_block }

property_list::
    property
    property_list property

property::
    type STRING interp_string_opt = atomic_value
    type STRING interp_string_opt = [ complex_element_list ]

dimensions::
    INT
    INT "," INT
    INT "," INT "," INT
    INT "," INT "," INT "," INT

type::
    basic_type
    basic_type [ dimensions ]
    basic_type [ dimensions ] [ INT ]

basic_type::
    "float" "int" "string" "short" "byte" "half"
    "bool" "double"

complex_element_list::
    nothing
    element_list
    element_list "..."

element_list::
    element
    element_list element

element::
    atomic_value
    [ atomic_value_list ]

atomic_value_list::
```

```
        string_value_list
        numeric_value_list

    atomic_value::
        string_value
        numeric_value

    string_value_list::
        string_value
        string_value_list string_value

    string_value::
        STRING

    numeric_value_list::
        numeric_value
        numeric_value_list numeric_value

    numeric_value::
        float_num
        int_num

    float_num::
        FLOAT
        - FLOAT

    int_num::
        INT
        - INT
```

5 Types of Property Data.

The GTO format pre-defines a small number of data types that can be stored as properties. The currently defined types are:

`double` [Property Type]
: 64 bit IEEE floating point.

`float` [Property Type]
: 32 bit IEEE floating point.

`half` [Property Type]
: 16 bit IEEE floating point

`int` [Property Type]
: 32 bit signed integer.

`int64` [Property Type]
: 64 bit signed integer.

`short` [Property Type]
: 16 bit unsigned integer.

`byte` [Property Type]
: 8 bit unsigned integer (char).

`bool` [Property Type]
: Bit or bit vector. Not currently implemented.

`string` [Property Type]
: The string type is stored as a 32 bit integer index into the GTO file's string table. So storing a lot of strings (especially if there is a lot of redundancy) is reasonably cheap. All strings in the GTO file are stored in this manner.

Each of these data types can be made into a vector of that type. For example the float data type can be made into a point `float[3]` or a matrix `float[16]`. To store a scalar element the size of the vector is 1. (e.g. `float[1]`).

In this document, the types are all specified as 2 dimensional arrays ala the C programming language. Here is a complete list of example type forms:

- `float[3]` - the float triple type.
- `float[1][1]` - a single floating point number.
- `float[3][]` - any number of float triples.
- `float[3][3]` - three float triples.
- `float[16][]` - any number of a 16 float element.
- `float[4,4][1]` - a 4x4 float matrix.
- `float[3,3][]` - any number of 3x3 float matrices.
- `float[4,512,128][7]` - seven four component 512x128 images.
- `float[3,32,32,32][1]` - a 3 component 32x32x32 volume.

6 Interpretation Strings

Each property can have an additional string stored with it call the "interpretation". The intent is to allow applications to provide specific information about the property. For example, a property of type `float[4]` can be interpreted as a homogeneous 3D coordinate, a quaternion, or an RGBA value. The interpretation field can be used to distinguish between them.

Why not just make new primitive GTO types for these? The format's only purpose is storage of data. By decoupling the interpretation of the data from its storage, each application is allowed to make its own policy while maintaining flexibility for simpler applications.

Here's a simple example of `gtoinfo` output of a file with an image object in it created with `gtoimage`:

```
object "image" protocol "image" v1
  component "image"
    property string[1][1] "originalFile" interpret as "filename"
    property string[1][1] "originalEncoding" interpret as "filetype"
    property string[1][1] "type"
    property int[1][2] "size"
    property float[3][199168] "pixels" interpret as "RGB"
```

The example illustrates two points: the interpretation string can be used to better determine the data type of the property (as is the case with the "RGB" string) and it can also be used to interpret the usage (the "filename").

Some of the stings will be application specific. Programs that generically edit GTO files should attempt to preserve the interpretation strings.

It is not an error to define the interpretation for a property as the empty string – in other words, unspecified.

The following strings are not currently part of the format specification but are used by the sample implementation. In a future release we may make these "official". Its ok to have multiple space seperated strings in the interpretation strings (e.g. "4x4 row-major").

`coordinate` [Interpretation String]
: The data can be of any width or type. For width `N` the data represents a point in `N` dimensional space.

`normal` [Interpretation String]
: The data can be of any width or type. For width `N` the data represents a unit vector prependicular to an `N` dimensional surface or in the case of `N` == 2, a curve.

`4x4` [Interpretation String]
: The width of the property data should be 16. The data is intended to be interpreted as a 4x4 matrix. For example, the **object.globalMatrix** property of the **Coordinate System** protocol would be a "4x4" property.

`3x3` [Interpretation String]
: The width of the property data should be 9. The data is intended to be interpreted as a 3x3 matrix.

`row-major` [Interpretation String]
: Indicates that matrix data is in row major ordering.

`column-major` [Interpretation String]
: Indicates that matrix data is in column major ordering.

`quaternion` [Interpretation String]
: The width of the property should be four. The data should be interpreted as a quaternion. Presumably the type of a quaternion property would be `float[4]` or `double[4]` since these are the only types that make sense. The first element of the data is the real part followed by the "i", "j", and "k" imaginary components.

`complex` [Interpretation String]
: The width of the property should be two. The data is interpreted as a complex number with the first element being the real part and the second element the imaginary part.

`indices` [Interpretation String]
The data type should be an integral type. The property contains indices.

`bbox` [Interpretation String]
The data type should have an even width. The property contains bounding boxes.

`homogeneous` [Interpretation String]
The width of the property should be two or more. If the width is three, then the data is a two dimensional homogeneous coordinate. If the width is four, then the data is a three dimensional homogeneous coordinate. So for data of width `N` the data represents a homogeneous coordinate in `N-1` dimensions.

`RGB` [Interpretation String]
The width of the property should be three. The data represents a color with red, green, and blue components.

`BGR` [Interpretation String]
The width of the property should be three. The data represents a color with blue, green, and red components. (Reversed `RGB`)

`RGBA` [Interpretation String]
The width of the property should be four. The data represents a color (or pixel) with red, green, blue, and alpha components.

`ABGR` [Interpretation String]
The width of the property should be four. The data represents a color (or pixel) with alpha, blue, green, and red components. (Reversed `RGBA`)

`bezier` [Interpretation String]
The property represents a 2D bezier curve for animation. The type should be a floting point type and the width six. Each element is a key frame value.

`weighted` [Interpretation String]
In the case of bezier animation curves, the curve should be evaluated with weighted tangents.

7 Object Protocols

The Object data interpretation is not defined by the GTO format. However, there are currently some protocols in use that are well defined and these are documented here. Caveat emptor: gto files in the wild may contain more data than these protocols define, but they presumably will obey the protocol if they indicate it by name. It's also possible that some objects may obey more than one protocol yet only indicate that they follow one. Unfortunately, some protocols also specify optional components and properties in case all of this was not confusing enough.

Protocols also have a version number. The version number is an integer; there are no sub-versions. If there are significant changes to a protocol, the version number should be bumped. The version number is not meant as a method of making alternate protocols with the same name. We have had to make three modifications to the protocols since the file

format was invented; one to the polygon protocol and one to the transform protocol, and the introduction of a new protcol (connections). The changes are documented in those sections.

In this document, properties are all named "comp.prop", where "comp" is the name of the component the property belongs to and "prop" is the name of the property. This is done to prevent ambiguity when two different properties in different components but with the same name exist. In the GTO file and when using the reader library only the property name will appear.

There are two kinds of protocols: major and minor. Every object must have a major protocol that's stored in the ObjectHeader – this is the main indicator of how to interpret the object data. In addition, the object may also have several minor protocols. These indicate optional data and how to interprete it. The next section describes how these are stored in the file.

7.1 Object Protocol

The name of the protocol as it appears in the ObjectHeader is "object" version 1. The protocol does not require any other protocols. Here it is:

`object` [Required Component]
: A container for properties which don't fit into other component catagories well. A catch-all data "per-object" component.

`float[16][1] object.globalMatrix` [Optional Property]
: The global world-space transform for the object.

`float[6][1] object.boundingBox` [Optional Property]
: The global world-space bounding box for the object.

`string[1][1] object.parent` [Optional Property]
: Name of this object's parent in a scene heirarchy.

`string[1][1] object.name` [Optional Property]
: The name of the object. This name should be identical to the name in the Object-Header.

`string[1][] object.protocol` [Optional Property]
: Additional protocols. This property may contain the main protocol name and any other minor protocols that the object adheres to. If a protocol name appears in this property, the object must adhere to that protocol. Its not an error for a program to output this property with only the major protocol as its value; this is of course redundant since the protocol name is required by the ObjectHeader. It is also not an error for this property to exist but contain nothing.

`int[1][] object.protocolVersion` [Optional Property]
: Additional protocol version numbers. This property may exist if the **object.protocol** property exists. Each entry in this property corresponds to the same entry indexed in the **object.protocol** property. This property must contain the same number of elements that the **object.protocol** property does.

You may be asking why the **object** protocol exists at all. The name of an object is stored in the ObjectHeader in the file and in the C++ library is passed to the reader code. The "name" property is redundant right? Well yes. But some programs will output the name both in the ObjectHeader and in an **object** component as the property "name".

The main point of this protocol is to define the **object** component. This component is meant to hold data that is "per object" and which doesn't really fit neatly into other components. The name is one such case. The coordinate system protocol also defines properties in the **object** component and the minor protocols are optionally stored here.

7.2 Coordinate System Protocol

The name of the protocol as it appears in the ObjectHeader is "transform" version 3[1]. The protocol requires the **object** protocol. Objects which obey the **transform** protocol will have global matrices and possibly a parent.

`object` [Required Component]
: From the "object" protocol.

`float[16][1] object.globalMatrix` [Required Property]
: A 4x4 matrix of floating point numbers. This matrix describes the world matrix of the coordinate system.

`string[1][1] object.parent` [Optional Property]
: The name of an object to which this coordinate system is parented. Presumably this object (if it appears in the gto file) will also obey the **transform** protocol. If this property does not exist or the name is "" (the empty string) then the coordinate system presumably is a root coordinate system.[2]

7.3 Particle Protocol

The name of the protocol as it appears in the ObjectHeader is "particle" version 1. The protocol may include the **object** and **transform** protocols.

`points` [Required Component]
: The points component is transposable. That means that all of its properties are required to have the same number of elements.

`float[3][] points.position` [Optional Property]
`float[4][] points.position` [Optional Property]
: The position property is intended to hold the position of the particle in its own coordinate system or world space if it has no coordinate system. The element is either a 3D or 4D (homogeneous) point.

`float[3][] points.velocity` [Optional Property]
: The velocity property – if it exists – should hold the velocity vector per-point in the same coordinate system that the "position" property is in.

---

[1] In version 1, the transform protocol's object.globalMatrix property used to be of type `float[1][16]`. This was a mistake that has been corrected in version 2.

[2] In version 3 of the **transform** protocol, the **object.parent** property is redundant and therefor deprecated. The **connection** protocol handles the transformation hierarchy information and in a much more elegant manner See Section 7.11 [Inter-Object], page 25.

`int[1][] points.id` [Optional Property]

The "id" property should it exist will *always* be defined as an integer per particle (or other integral type if it ever changes). This number should be unique for each particle. Ideally, multiple GTO files with a point that has the same "id" property for a given particle animation *should* be the same particle.

The **particle** protocol defines the **points** component that many other protocols are derived from. For example, the **NURBS** protocol uses the points defined by the particle protocol as control vertices. There can be any number of properties associated with particles including string per-particle.

The **points** component is marked transposable in the its ComponentHeader. This means that the properties in the component are guaranteed to have the same number of elements. Because of this, the data for the properties in a transposable component may be stored differently than other components. For example, the normal state of affairs is to write data like this:

```
position0 position1 position2 .... positionN
velocity0 velocity1 velocity2 .... velocityN
mass0 mass1 mass2 .... massN
```

So that you must read through all of the particle positions before you can read the first particle's velocity. But this is not usually the best way to read particle data for rendering. You may want to cull the particles as you read them without storing the data. In order to do this the data needs to be laid out like this:

```
position0 velocity0 mass0
position1 velocity1 mass1
position2 velocity2 mass2
...
```

In this case, each particle is scanned in one chunk allowing for optimizations. Obviously this complicates reading, but in the case of giga-particle renderers, this can be a huge memory savings.

7.4 Strand Protocol

A **strand** object contains a collection of curves. This is somewhat analogous to an object of protocol **particle** as described above.

`points` [Required Component]

`float[3][] points.position` [Required Property]

The CVs which make up each curve. The number of CVs per curve can vary by curve type and size.

`strand` [Required Component]

Information that is relevent to the *all* strands in the object.

`string[1][] strand.type` [Required Property]

String describing curve type. Currently, supported values are `linear` for degree 1 curves, or `cubic` for degree 3 curves.

`float[1][1] strand.width` [Optional Property]
If each end of all curves is the same width, you can just specify that one number instead of the list as with **elements.width** below.

`elements` [Required Component]
Information that applies to each separate strand in the object.

`int[1][] elements.size` [Required Property]
This is a list of the sizes of each curve in this object. For example, if there are two curves in this object, with 4 CVs and 3 CVs respectively, then:

```
elements.size = [ 4 3 ]
```

`float[2][] elements.width` [Optional Property]
This is a list of the widths of each end of each curve. The width for each curve will be linearly interpolated over the length.

7.5 NURBS Protocol

The name of the protocol as it appears in the ObjectHeader is "NURBS" version 1. The protocol requires the **particle** protocol and optionally includes the **object** and **transform** protocols.

`points` [Required Component]
see **particle** protocol. The points describe data per NURBS control vertex.

`float[3][] points.position` [Required Property]
`float[4][] points.position` [Required Property]
The position property holds the control point positions in its own coordinate system or world space if it has no coordinate system. The element is either a 3D or 4D (homogeneous) point. If the type is float[4] the fourth component of the element will be the rational component of the control point position. The control points are laid out in *v*-**major** order (*u* iterates more quickly than *v*).

`float[1][] points.weight` [Optional Property]
If the position property is of type `float[3][]` there may optionally be a "weight" property. This property holds the homogeneous (rational) component of the position. Older GTO writers may export data in this manner. The preferred method is to use a `float[4]` element position.

`surface` [Required Component]
Properties related to the definition of a NURBS surface are stored in this component.

`float[1][2] surface.degree` [Required Property]
The degree of the surface in *u* and *v*.

`float[1][] surface.uKnots` [Required Property]
`float[1][] surface.vKnots` [Required Property]
The NURBS surface knot vectors in *u* and *v* are stored in these properties. The knots are not piled. The usual NURBS restrictions on how numbers may be stored in the knot vectors apply.

`float[1][2] surface.uRange` [Required Property]
`float[1][2] surface.vRange` [Required Property]
The range of the knot parameters in *u* and *v*.

The **NURBS** protocol currently does not handle trim curves, points on surface, etc. Ultimately, the intent is to handle the trim curves and other nasties as NURBS curves-on-surface which will be stored in additional components. UBS surfaces can be stored as NURBS with non-rational uniform knots.

7.6 Polygon Protocol

The name of the protocol as it appears in the ObjectHeader is "polygon" version 2[3]. The protocol requires the **particle** protocol and optionally includes the **object** and **transform** protocols.

There are a number of alternative configurations of this protocol depending on the value of the **smoothing.method** property. All of these involve the placement of normals in the file.

`points` [Required Component]
See **particle** protocol. The points describe data per vertex.

`float[3][] points.position` [Required Property]
The positions for regular polygonal meshes are stored as `float[3]`.

`float[3][] points.normal` [Optional Property]
Normals per vertex. The **smoothing.method** property will have the value of *Smooth* if this property exists. Note that use of the *Smooth* smoothing method does not require that this property exists. If it does not the method is merely and indication of how the normals should be constructed.

`normals` [Optional Component]
This component will exist if the value of **smoothing.method** is *Partitioned* or *Discontinuous*.

`float[3][] normals.normal` [Required Property]
This property is required only if the **normals** component exists and the value of **smoothing.method** is *Partitioned* or *Discontinuous*.

`elements` [Required Component]
The elements component is transposable. All properties in the elements component must have the same number of elements. Each element corresponds to a polygonal primitive.

`byte[1][] elements.type` [Required Property]
Elements are modeled after the OpenGL primitives of the same name. The vertex order is identical to that defined by GL. The type numbers outside those given here are not defined but reserved for future use. So far, these are the define type numbers:

[3] In version 1 of the polygon protocol, the **element.size** and **element.type** properties were combined into an **element.primitive** property. We felt that this was adding unnecessary complexity and because the primitive property was an int, it was taking up extra space.

0 – Polygon
General N-sided polygon. This can be used for any polygon that has 3 or more vertices.

1 – Triangle
A three vertex polygon.

2 – Quad A four vertex polygon.

3 – TStrip Triangle strip.

4 – QStrip
Quad strip.

5 – Fan Triangle fan.

`short[1][] elements.size` [Required Property]
The size of each primitive. Because the type is short, there is a limit of 65k veritices per primitive.

`short[1][] elements.smoothingGroup` [Optional Property]
This property may exist if the value of **smoothing.method** is *Partitioned*. In that case, this property indicates the smoothing group number associated with each element. These can be used to recompute the normals. These numbers are the same as those found in the Wavefront .obj file format's "s" statements. A value of 0 indicates that an element is not in a smoothing group.

`float[3][] elements.normal` [Optional Property]
Normals per element. The **smoothing.method** property will have the value of *Faceted* if this property exists. Note: the use of *Faceted* smoothing method does not require that this property exists. If it does not, the smoothing method is merely and indication of how the normals should be created.

`indices` [Required Component]
The indices component is transposable. All of its properties are required to have the same number of elements. Each entry in the indices component corresponds to a polygonal vertex.[4]

`int[1][] indices.vertex` [Required Property]
A list of all the polygonal vertex indices in the same order as the **elements.primtives**. The indices refer to the **points.position** property. So if the first polygonal element is a triangle and second is a general four vertex polygon then vertex indices will be something like:

```
0 1 2 1 0 3 4 ...
```

which would be grouped as:

```
(0 1 2) (1 0 3 4) ...
```

The first group is the triangle and the second the polygon.

[4] The **indices** component in a polygonal object contains values which are analogous to the RenderMan `facevarying` type modifier.

`int[1][] indices.st` [Optional Property]
: Similar to the vertex indices but indicates indices into *st* coordinates. These are usually stored in the "mappings" component but may also appear in the **points** component.

`int[1][] indices.normal` [Optional Property]
: Indices into stored normals if there are any. The **smoothing.method** property will have the value of *Partitioned* or *Discontinuous* if this property exists.

`mappings` [Optional Component]
: Contains parametric coordinates. The property names in mappings usually correspond to names found in the **indices** component but not always. For example **mappings.st** would be a `float[2][]` property holding texture coordinates indexed by **indices.st**.

`smoothing` [Optional Component]
: The smoothing component exists to hold the smoothing method and any ancillary data for the method. If there is no smoothing component (and hence no **smoothing**) you can assume anything you want.

`int[1][1] smoothing` [Required Property]
: There five defined smoothing methods (0 through 4). They are:

  0 – None No smoothing method specified. No additional properties associated with normals will appear in the object.

  1 – Smooth
  One normal at every vertex. There will be a `float[3][]` **normal** property as part of the **points** component. Each vertex has a unique normal.

  2 – Faceted
  One normal for each face. There will be a `float[3][]` **normal** property in the **elements** component. Each element has a unique normal.

  3 – Partitioned
  Same as the Wavefront .obj smoothing groups. There will be a **normals** component containing a `float[3][]` **normal** property and an `int[1][]` **normal** propery in the **indices** component. Each element vertex will have an index into the **normals.normal** property.

  4 – Discontinuous
  Like *Partitioned* but with additional lines and points of discontinuity. The same properties that hold the *Partitioned* information will hold the *Discontinuous* information. There will also be a component called **discontinuities** which will have a `int[1][]` property called **indices** indicating the points and lines of discontinuity.

7.7 Subdivision Surface Protocols

The name of the protocol as it appears in the ObjectHeader is "catmull-clark" or "loop" depending on the intended subdivision scheme. The protocol requires the **polygon** protocol.

The smoothing and any normals properties on the **polygon** protocol should be ignored if they exist.

The protocol indicates how the surface should be treated. Note that the canonical element type for each of the two schemes is not guaranteed to be the only element type stored in the file. For catmull-clark this means that triangles and general polygons will need to be made into quads. Similarily loop surfaces may have quads and other non-triangle primitives that need to be triangulated.

These protocols do not currently define methods for storing edge creasing parameters.

Disclaimer: there are restrictions on what kind of topology surfaces are allowed to have for a given renderer (for example). In most cases surfaces need to be manifold. Some applications can deal with special cases better than others.

7.8 Image Protocol

The Image protocol describes image data in the form of an object. This data makes it possible to store texture maps, backgrounds, etc, directly in the GTO file.

When images are stored in a GTO file, use of Gzip compression is highly recommended if the data is unencoded. As of version 2.1, the supplied Reader and Writer classes default to using zlib compression.

If the image data is encoded, its better *not* to use compression on the GTO file (especially if the file contains only image data).

`image` [Required Component]
: The image data and other information will be stored in the **image** component.

`int[1][] image.size` [Required Property]
: The size (and dimension) of the image. There will be N sizes in this property corresponding to the N dimensions of the image.

`string[1][1] image.type` [Required Property]
: The image type. For interactive purposes, the image channels may correspond to a particular fast hardware layout.

  - `RGB` Three channels corresponding to red, green, and blue in that order.
  - `BGR` Three channels corresponding to blue, green, and red in that order.
  - `RGBA` Four channels corresponding to red, green, blue, and alpha in that order.
  - `ABGR` Four channels corresponding to alpha, blue, green, and red in that order.
  - `L` One channel corresponding to luminance.
  - `HSV` Three channels corresponding to hue, saturation, and value. (The HSV color space).
  - `HSL` Three channels corresponding to hue, saturation, and lightness. (The HSL color space).
  - `YUV` Three channels corresponding to the YUV color space.

`int[1] image.cs` [Optional Property]
: The coordinate system of the image. The value of **image.cs** can be any one of the following:

0 – Lower left origin.
The first pixel in the image data is the lower left corner of the image data and corresponds to NDC coordinate (0,0).

1 – Upper left origin.
The first pixel in the image data is the upper left corner of the image data and corresponds to NDC coordinate (0,0).

Any one of the following properties are required to hold the actual image data:

`byte[N][] image.pixels` [Property]
`short[N][] image.pixels` [Property]
`half[N][] image.pixels` [Property]
`float[N][] image.pixels` [Property]
The element width determines the number of channels in the image. For example, the type `byte[3][]` indicate a 3 channel 8-bit per channel image. The number of elements in this property should be equal to `image.size[0] * image.size[1] * ... image.size[N]` where **image.size** is the property defined above.

7.8.1 Additional Image Properties Used by GTV Files.

The base GTO library does not deal with encoded image data or tiling of images. GTV is a specialization of the GTO format for storing movie frames. Some of the GTV properties are documentated here. (See documentation for the GTV library for more info).

`string[1] image.encoding` [Optional Property]
If the pixel data is encoded this property will indicate a method to decode it. Typical valies are "jpeg", "jp2000", "piz", "rle", or "zip". The pixels will be stored in the **image.pixels** as `byte[1][]`.

7.9 Material Protocol

The name of the protocol as it appears in the ObjectHeader is "material". The material protocol groups a parameters and a method (shader) for rendering. The material protocol can optionally include the **object** protocol.

The material definition is renderer and pipeline dependant. Material assignment is implemented using the **connection** protocol. See Section 7.11 [Inter-Object], page 25.

The **material** protocol is intended for use with software renderers. Interactive material definitions may be more easily defined on the assigned object.

`material` [Required Component]
Properties unrelated to parameters appear in the **material** component.

`string[1][1] type` [Required Property]
The value of the **material.type** property is renderer dependant. For a RIB renderer, the value of type might be "Surface", "Displacement", "Atmosphere" or a similar shader type name.

`string[1][1] shader` [Optional Property]
The name of the shader. For RenderMan-like renderers this might be the name of an ".sl" file.

`string[1][1] genre` [Optional Property]
: A property to further identify the material. This is most useful for identifying the target renderer for a material.

`parameters` [Optional Component]
: The set of parameters corresponding to the **material.type**.

7.10 Group Protocol

7.11 Inter-Object Connection Protocol

The name of the protocol as it appears in the ObjectHeader is "connection" version 1.

Files which employ the **connection** protocol will typically contain a connection object with the special cookie name ":connections" indicating the purpose of the object as well as preventing namespace pollution. See Section 8.3 [Special Cookies], page 29.

Each component in a connection object is a connection type. For example, the "parent_of" connection type is used to represent transformation hierarchies. In a connection object, there will be a single component called "parent_of" which will contain the required properties **parent_of.lhs** and **parent_of.rhs** at a minimum. Some connection types may have additional data in the form of additional properties.

Connection components are transposable. The number of elements in properties comprising a connection component will be consistant. So a single "parent_of" component can encode an entire scene transformation hierarchy.

Connection components have the following properties. Note that where **connection_type** occurs in the property name, you would substitute in the actual name of the connection type. ("parent_of" for example).

`string[1][] connection_type.lhs` [Required Property]
`string[1][] connection_type.rhs` [Required Property]
: The left-hand-side and right-hand-side of the connection.
  - If the connection is directional, then an arrow indicating the direction would have its tail on the left-hand-side and its head pointing at the right-hand-side.
  - If the connection type does not require a direction then these properties are still used to describe the two ends of the connection.
  - Each entry will be the name of an object. There is no requirement that the ends of the connection exist in the file. For example, one end of the connection could be an image on disk.
  - The empty string is a valid value. You could think of the empty string as indicating a grounded connection.
  - Its ok for both ends of the connection to have the same value.

The GTO specification includes a couple of basic connection types.

7.11.1 Transformation hierarchies.

The "parent_of" connection type is used to store transformation hierarchies. The connection type requires only the **lhs** and **rhs** properties. Transformation hierarchies are usually tree structures, but can also be DAGs (as is the case with Maya or Inventor).

Using "parent_of" as a cyclic generalized network connection is probably an error for most applications. To be safe the topology of a "parent_of" network should be a tree.

7.11.2 Material Assignment

The "material" connection type indicates a material assignment to an object. The left-hand-side name is a renderable object in a GTO file The right-hand-side is the name of a material object in a GTO file.

7.11.3 Container Assignment

The "contains" connection type indicates membership in a group or similar type of container object. The LHS is the group or container, the RHS is the object which is a member.

7.12 Difference File Protocol

If the **object.protocol** property contains the string "difference" then the object contains difference data; the data is relative to some other reference file.

For example, for animated deforming geometry its advantageous to write a reference file for geometry in its natural undeformed state then write only the **points.position** property in a gto file per frame to store animation. The **difference** minor protocol can apply to any major protocol.

If a reference file and a difference for file it exists, you can reconstruct the file represented by the difference file using the `gtomerge` command. See Section 13.3 [gtomerge], page 44.

7.13 Sorted Shell File Protocol

If the **object.protocol** property contains the string "sorted" and the object's major protocol is **polygon** then the object contains sorted shell data.

This protocol guarantees that the vertices and elements of shells — isolated sections of polygonal geometry — will be continguous in the **points** and **elements** components of the object.

`shells` [Required Component]
: The **shells** component is transposable. Each property in the component should have the same number of elements.

`int[1][] shells.vertices` [Required Property]
: The number of contiguous vertices that make up the Nth element's shell.

`int[1][] shells.elements` [Required Property]
: The number of contiguous elements that make up the Nth element's shell.

7.14 Channels Protocol

This minor protocol declares data mapped onto geometric surfaces. Usually the data is mapped using one of the parameterizations found in the **mappings** component of polygonal or sub-d geometry or possibly using the natural parameterization of a surface as is often the case with NURBS.

Each declared channel appears as a `string[1][]` property of a **channels** component on the geometry. The name of the property is the name of the channel. The property should contain at least one element.

The first element of the property should indicate the name of the mapping to use. This is either the name of one of the properties in the **mappings** component or "natural" indicating that the natural parameterization of the surface should be used.

The second and subsequent elements should contain the name of data to map. This could be a texture map file on disk, an image object in the GTO file, or a special cookie string. The lack of second element can be used as a special cookie.

`channels` [Required Component]

The component holds the names of all the channels on the geometry.

7.14.1 Example

Here is a cube with "color", "specular", and, "bump" channels assigned.

```
Object "cube" protocol "polygon"
    Component "points"
        Property float[3][8] "position"
    Component "elements"
        Property byte[1][8] "type"
        Property short[1][8] "size"
    Component "indices"
        Property int[1][32] "vertex"
        Property int[1][32] "st"
    Component "mappings"
        Property float[2][24] "st"
    Component "channels"
        Property string[1][2] "color"
        Property string[1][2] "specular"
        Property string[1][2] "bump"
```

The contents of the "channels" properties might be:

```
string[1] cube.channels.color    = [ "st" "cube_color.tif" ]
string[1] cube.channels.specular = [ "st" "cube_specular.tif" ]
string[1] cube.channels.bump     = [ "st" "cube_bump.tif" ]
```

7.15 Animation Curve Protocol

The animation curve protocol defines a single component called **animation** in which each property holds an animation curve or data stream. The property's interpretation string indicates how the data should be evaluated.

7.15.1 Example

Here is a cube with animation curves.

```
Object "cube" protocol "polygon"
    Component "points"
        Property float[3][8] "position"
```

```
Component "elements"
    Property byte[1][8] "type"
    Property short[1][8] "size"
Component "indices"
    Property int[1][32] "vertex"
Component "animation"
    Property float[6][2] "xtran" interpret as "bezier"
    Property float[6][2] "ytran" interpret as "bezier"
    Property float[6][2] "ztran" interpret as "bezier"
    Property float[6][5] "xrot" interpret as "bezier"
    Property float[6][8] "yrot" interpret as "bezier"
    Property float[6][10] "zrot" interpret as "bezier"
    Property float[1][100] "xscale" interpret as "stream"
```

8 Naming Conventions

GTO files can contain cross references to parts of themselves, objects outside the file, or virtual/logical objects in applications. Because of the potential morass that can result from complete free-form naming, there are conventions which are part of the file specification.

Failure to follow the guidelines does not mean a GTO file is ill-formed; there's always a good reason to ignore guidelines. But having a basis for consistancy is usually a good idea.

Some of these topics are a bit "advanced" in that they build off ideas that present themselves after using the file format for a while. If you are just learning about the format, consider this a reference section and skip it. If you're trying to decrypt a complicated GTO file with strange garbled naming, then this section is for you.

8.1 Valid Names

Names should be valid C identifiers, but should not contain the dollar-sign character ($). This means that no whitespace or punctuation is allowed.

Note that this does *not* apply to protocol names.

There is nothing in the sample `Reader` or `Writer` classes which enforces the valid name guideline. However, some applications (Maya) cannot handle names with whitespace and/or punctuation. So plug-ins which implement GTO reading/writing will have to enforce the application's specific naming requirements.

This guideline is broken by Section 8.3 [Special Cookies], page 29. Its also broken by Section 8.4 [Cross References], page 29.

8.2 Exactly Specifying a Property or Component

By convention, the full name or path name of a property is refered to like this:

```
OBJECTNAME.[COMPONENTNAME.]+PROPERTYNAME
```

where there can be any number of COMPONENTNAME parts.

When indicating a property name relative to an object then:

```
[COMPONENTNAME.]PROPERTYNAME
```

should suffice. In this manual, names of components and properties are disambiguated using the dot notation. In addition, this is the format of output from the `gtoinfo` command. There is nothing about the GTO file itself which relates to this notation other than the cross-referencing naming convention discussed below.

8.3 Indicating Special Handling

Some objects, components, or properties in the GTO file may contain data for which names are not particularily useful or that may simply pollute the object or component namespace.

In other cases (component names most notably) the name may be used as information necessary to interpret data associated with it.

In order to distinguish these names from run-of-the-mill names, you should include a colon in the name. Names with colons are considered "special cookie" names and objects which have them may be handled differently than other objects.

The **connection** object protocol, for example, requires that a special file object exist to hold data. This object is not necessarily related to a logical object in the application, its just a container for the connection data. These objects are named using the special cookie syntax. Usually the name is ":connections". See Section 7.11 [Inter-Object], page 25.

There is no rule regarding the placement of the colon in the name; it can appear anywhere in the name that is useful for the application. However, if the entire name is a special cookie — there is not additional information encoded in the name beyond itself — the recommend form is to have the colon be the first character.

8.4 Cross References Encoded in Names

Sometimes there is a need to have a property or component *refer* to another property, component, or object in the file (or somewhere else).

To cross reference the data in one property with another, simply name the property the full (or partial) path to the referenced property. For example, here's the output of `gtoinfo` on a GTO file which has cross referencing properties:

```
object "gravity" protocol "gravity" v1
  component "field"
    property float[3][1] "direction"
    property float[1][1] "magnitude"
  component ":datastream"
    property float[3][300] "field.direction"
    property float[1][300] "field.magnitude"
```

As you can guess, the intention here is that the properties called "field.direction" and "field.magnitude" in the ":datastream" component are data that is associated with the properties "direction" and "magnitude" in the "field" component.

9 Issues and Questionable Aspects of the Format

- There are currently no (publicly available) tools which verify that a file claiming to follow some protocol is correct.
- There is no 3D curve(s) protocol defined.
- The **NURBS** protocol does not handle trim curves. See Section 7.5 [NURBS Surfaces], page 19.
- The format does not contain dedicated space for auxillary information like the name and version of the program that wrote the file, the original owner, copright information, etc. However, our tools use the string table for these type of data – since its not an error have an unused interned string, we store the data as such. In our opinion, this is a fairly innocuous method. You can read unreferenced strings by using the `gtoinfo` command with the `-s` option. Note that these strings are often lost when programs read and write the file. See Section 13.1 [gtoinfo], page 43.
- Although the format specification includes transposable components (those marked with the Gto::Matrix flag may be transposed), the current reader/writer library does not handle files with transposed components. It does handle components that are marked as Gto::Matrix but not transposed. See Section 7.3 [Particles], page 17.
- The use of special cookie names and special cross-reference names seems to seriously complicate the format if the protocol is not carefully conceived. For example, using `gtomerge` to merge files containing connections does not work — the connections are merged like all the other data in the file. The correct behavior would be to combine the connections, but merge the other object data. Perhaps this is just a case for integrating `gtocombine` into `gtomerge`?
- Future versions should incorporate some form of check sum or some similar mechanism to do better sanity checking.
- There are many examples of properties whose data indexes into other property data. The most obvious of these are the polygon protocol **indices** properties. In order to combine gto files (concatenate polygonal data together for example) its necessary to know which properties are indexes and which are not. Index properties must be offset to be combined.
- The Boolean (bit fields) and Half data types are not implemented in the supplied writer library. Both of these types are useful in compressing geometric (and image) data.
- Material, Texture, and similar assignments and storage are usually very specialized at any particular production facility. The idea that a single method of encoding this information can be determined or enforced — or even usefully be stored in a GTO file — is not realistic. However, we hope that some method can be determined that at least preserves a good portion of common data for transfer.

  All of the protocols related to these concepts are marked **PROPOSED** in this document. See Section 7.9 [Material], page 24. See Section 7.11.2 [material], page 26.

10 Extending Protocols or the File Format

If you have an extension to a protocol or would like to change an existing protocol, we would like to hear about it. You can send mail to with the changes you're using or would

like made. We will collect ideas and proposals and try to make releases in a timely manner. We'd also like to hear from you if you're using it unmodified.

There is currently a small number of facilities that use the GTO format, but there is a large collection of tools that use it. Most proposals should maintain some backwards compatibility. However, we recognize that the there may be flaws that require revamping significant pieces and we're open to making changes to accommodate other facilities.

If you are using the format for academic purposes and are looking for a specific tool to munge GTO files, we may already have that tool even though it is not released. Contact us; we might be able to help you out.

We have been using the format since summer 2002 on a regular basis in production at Tweak Films and have found it stable and useful. The version of the code we use is identical to the released version.

11 C++ Library

The GTO Reader/Writer library is written in a subset of C++. The intention was to make the library as portable as possible. Unfortunately we have only tried it on platforms that support gcc 2.95 and greater. It is known to work on various Linux versions and Mac OS X. In either case it has been compiled with gcc.

11.1 Gto::Reader class

The Reader class (in namespace Gto) is designed as a fill-in-the-blank API. The user of the class derives from it; the base class defines a number of virtual functions which pass data to the derived class and ask the derived class questions about what data it wants.

The Reader class handles most of the difficult work in reading the file like keeping track of headers, sizes of properties, and the order of data. In addition, it handles the string table and looking up property string values. If the file was written by a machine with different sex (endianess) it will translate the data for you.

In addition, you can compile the GTO library with zlib support. This enables the Reader class to read gzipped GTO files natively and the Writer class to write them. This can be a significant space savings on disk and on saturated networks can make file loading faster. You can also pass a C++ istream object to the Reader if you want to read "in-core".

As the file is read, the Reader class will call its virtual functions to declare objects in the file to the derived class. The derived class is expected to return a non-null pointer if it wishes to later receive data for that object.

`Reader::Reader` (*unsigned int* `mode`) [Constructor]

The constructor argument *mode* indicates how the reader will be used. This value is a bit vector of the following or'ed flags:

Reader::None

The reader will be used in its standard *streaming* mode. The reader will attempt to read all the data in the file. This is the default value (or 0).

Reader::HeaderOnly
: The reader will stop once it has read the header sections of the GTO file. This is an optimization that applies to binary files only. This option is ignored when reading a text file.

Reader::RandomAccess
: The reader will read the header sections but not the data, however, it will initialize for use of the `Reader::accessObject()` function. Only binary GTO files can be read using the radom access mode.

Reader::BinaryOnly
: Only binary GTO files will be accepte by reader.

Reader::TextOnly
: Only text GTO files will be accepte by reader.

`Reader::~Reader` () [Destructor]
: Closes file if still open.

`bool Reader::open` (*const char** ***`filename`***) [Method]
: Open the file. The Reader will attempt to open file *filename*. If the file does not exist and zlib support is compiled in, the Reader will attempt to look for *filename*.gz and open it instead.

`bool Reader::open` (*std::istream&, const char** ***`name`***) [Method]
: Reads the GTO file data from a stream. The *name* is supplied to make error messages make sense.

`void Reader::close` () [Method]
: Close the file and clean up temporary data. If the stream constructor was used, the stream is *not* closed.

`std::string& Reader::fail` (*std::string* ***`why`***) [Method]
: Sets the error condition on the Reader and sets the human readable reason to *why*.

`std::string& Reader::why` () [Method]
: Returns a human readable description of why the last error occured. (Set by the `fail()` function).

`const std::string& Reader::stringFromId` (*unsigned int*) [Method]
: Given a string identifier, this method will return the actual string from the string table.

`const StringTable& Reader::stringTable` () [Method]
: Returns a reference to the entire string table.

`bool Reader::isSwapped` () *const* [Method]
: Returns true if the file being read needed to be swapped. This occurs if the machine the file was written on is a different sex than the machine reading the file (for example a Mac PPC written file read on an x86 GNU/Linux box).

`unsigned int Reader::readMode` () *const* [Method]
: Returns the mode value passed into the Reader constructor.

`const std::string& Reader::infileName` () *const* [Method]
Returns the name of the file or stream being read. This is the value passed in to the `Reader::open()` function.

`std::istream* Reader::in` () *const* [Method]
Return the input stream created by or passed into `Reader::open()`. If the GTO file is compressed binary, this function will return NULL.

`int Reader::linenum` () *const* [Method]
For text GTO files, the return value will be the current line being parsed. For binary GTO files, the return value is always 0.

`int Reader::charnum` () *const* [Method]
For text GTO files, the return value will be the current char column (in the current line) being parsed. For binary GTO files, the return value is always 0.

`Header& Reader::fileHeader` () *const* [Method]
Returns a reference to a Gto::Header structure corresponding to the file currently being read. This function is required by the text file parser. The function may disappear from future versions. See the `Reader::header()` function below for a better way to get header information.

The following functions are called by the base class.

`void Reader::header` (*const Header&* `header`) [Virtual]
This function is called by the Reader base class right after the file header has been read (or created).

`void Reader::descriptionComplete` () [Virtual]
This function is called after all file, object, component, and property structures have been read. For binary files, this is just before the data is read. For text files, this is after the entire file has been read.

The following functions return a `Reader::Request` object. This object takes two parameters: a boolean indicating whether the data in question should be read by the reader and a second optional data `void*` argument of user data to associate with the file data.

`Reader::Request::Request` (*bool* `want`, *void** `data`) [Constructor]
want value of true indicates a request for the data in question. *data* can be any void*. *data* is meaningless if the *want* is false.

`Reader::Request Reader::object` (*const std::string&* `name`, *const std::string&* `protocol`, *unsigned int* `protocolVersion`, *const ObjectInfo&* `header`) [Virtual]
This function is called whenever the Reader base class encounters an ObjectHeader. The derived class should override this function and return a Request object to indicate whether data should be read for the object in question. If it requests not to have data read, the Reader will not call the corresponding component() and property() functions.

`Reader::Request Reader::component` (*const std::string&* **`name`**, *const ComponentInfo&* **`header`**) [Virtual]
This function is called when the Reader base class encounters a ComponentHeader in the GTO file. If the derived class did not express interest in a particular object in the file by returning `Request(false)` from the object() function, the components of that object will not be presented to the derived class. The derived class should return `Request(true)` to indicate that it is interested in the properties of this *component*.

`Reader::Request Reader::property` (*const std::string&* **`name`**, *const char** **`interpString`**, *const PropertyInfo&* **`header`**) [Virtual]
This function is called when the Reader base class encounters a PropertyHeader in the GTO file. If the derived class did not express interest in a particular object or the component that the property belongs to, the properties of that component will not be presented to the derived class. The derived class should return `Request(true)` to indicate it is interested in the property data.

`void* Reader::data` (*const PropertyInfo&, size_t* **`byts`**) [Virtual]
This function is called before property data is read from the GTO file. The function should return a pointer to memory of at least size *bytes* into which the data will be read. The type, size, width, etc, of the data can be obtained from the `PropertyInfo` structure.

`void Reader::dataRead` (*const PropertyInfo&*) [Virtual]
This function is called after the `data()` function if the data was successfully read.

If you are using the Reader class in `Reader::RandomAccess` mode, you may call these functions after the read function has returned:

`Reader::Objects& Reader::objects` () [Method]
Returns a reference to an std::vector of Reader::ObjectInfo structures. These are only valid after `Reader::open()` has returned. You can use these structures when calling `Reader::accessObject()`.

`const Reader::Components& Reader::components` () [Method]
Returns a reference to an std::vector of Reader::ComponentInfo structures. These are only valid after `Reader::open()` has returned. This method is most useful when deciding how to call the `accessObject` function.

`const Reader::Properties& Reader::properties` () [Method]
Returns a reference to an std::vector of Reader::PropertyInfo structures. These are only valid after `Reader::open()` has returned. This method is most useful when deciding how to call the `accessObject` function.

`void Reader::accessObject` (*const ObjectInfo&*) [Method]
Calling this function on a GTO file openned for `RandomAccess` reading will cause the reader to seek into the file just for the data related to the object passed in. This is most useful when the objects' data cannot be held in memory and the order of retrieval is unknown. The reader attempts to be efficient as possible without using too much memory.

11.2 Gto::Writer class

The Writer class (in namespace Gto) is designed as an API to a state machine. You indicate a conceptual hierarchy to the file and then all the data. The writer handles generating the string table, the header information, etc.

The following is an example that outputs a polygon cube using the **polygon** protocol.

```
float points[3][] =
{   { -2.5, 2.5, 2.5 },  { -2.5, -2.5, 2.5 },
    { 2.5, -2.5, 2.5 },  { 2.5, 2.5, 2.5 },
    { -2.5, 2.5, -2.5 }, { -2.5, -2.5, -2.5 },
    { 2.5, -2.5, -2.5 }, { 2.5, 2.5, -2.5 } };

unsigned char type[] = { 2, 2, 2, 2, 2, 2 };
unsigned char size[] = { 4, 4, 4, 4, 4, 4 };

int indices[] = {0, 1, 2, 3,    7, 6, 5, 4,
                 3, 2, 6, 7,    4, 0, 3, 7,
                 4, 5, 1, 0,    1, 5, 6, 2 };

Gto::Writer writer;
writer.open("cube.gto");

writer.beginObject("cube", "polygon", 2);   // polygon version 2

    writer.beginComponent("points");
        // will write 8 float[3] positions
        writer.property("positions", Gto::Float, 8, 3);
    writer.endComponent();

    writer.beginComponent("elements");
        // one per face
        writer.property("size", Gto::Short, 8, 1, 1);
        writer.property("type", Gto::Byte, 8, 1, 1);
    writer.endComponent();

    writer.beginComponent("indices");
        // one per vertex per face
        writer.property("vertex", Gto::Int, 24, 1, 1);
    writer.endComponent();

writer.endObject();

// repeat writer object blocks if more objects

// output all the data in order declared
```

```
writer.beginData();
writer.propertyData(points);
writer.propertyData(type);
writer.propertyData(size);
writer.propertyData(indices);
writer.endData();
```

`Writer::Writer` () [Constructor]
Creates a new Writer class object. Typically you'll make one of these on the stack. This constructor requires you call the open function to actually start writing the file.

`Writer::Writer` (*std::ostream&*) [Constructor]
Creates a new Writer class object which will output to the passed C++ output stream.

`Writer::~Writer` () [Destructor]
Closes file opened with the `open()` function if still open. The destructor will not close any passed in output stream.

`bool Writer::open` (*const char** **`filename`**, *FileType* **`mode`** = *CompressedGTO*) [Method]
Open the file. The Writer will attempt to open file *filename*. If the file is not writable for whatever reason, the function will return false. If *mode* is `CompressedGTO` (the default value), the Writer class will output a binary compressed file. If the value is `BinaryGTO` the file will be binary uncompressed. If *mode* is `TextGTO` a text GTO file will be written. Compressed GTO files can be uncompressed manually using `gzip`. Compression is available only if the library is compiled with zlib support.

`bool Writer::open` (*const char** **`filename`**, *bool* **`compress`** = *true*) [Method]
This function exists for backwards compatibility. Use the other `open()` function instead. This function can open a file for binary output only (it cannot write a text GTO file).

`void Writer::close` () [Method]
Close the file and clean up temporary data. If the stream constructor was used, the stream is *not* closed.

`void Writer::beginObject` (*const char** **`name`**, *const char** **`protocol`**, *unsigned int* **`version`**) *const* [Method]
Declares an object. Its components and properties must be declared before `endObject()` is called. The *name* is the name of the object as it will appear in the gto file. The *protocol* is the protocol string indicating how the object data will be interpreted and the *version* number indicates the protocol version. The Writer class does not verify that the data output conforms to the protocol.

`void Writer::beginComponent` (*const char** **`name`**, *bool* **`transposed`**=*false*) [Method]
Declares a component. The component properties must be declared before a call to endComponent(). The *name* is the name of the component as it will appear in the gto file. The *transposed* flag is optional and indicates whether or not the component property data should be output transposed or one property at a time (the default).

`void Writer::property` (*const char** **`name`**, *Gto::DataType* **`type`**, *size_t* **`numElements`**, *size_t* **`partsPerElement`**=*1*, *const char** **`interpString`**=*0*) [Method]

Declare a property. The *name* is the name of the property as it appears in the gto file. The *type* is one of `Gto::Double`, `Gto::Float`, `Gto::Int`, `Gto::String`, `Gto::Byte`, `Gto::Half`, or `Gto::Short`. *numElements* indicates the number of elements of size *partsPerElement* that will be in the property data. So for example, if the property is declared as a Gto::Float of with *partsPerElement* of 3 and there 10 of them, then the writer will expect an array of 30 floats when the propertyData is finally passed to it. The last argument *interpString* is an optional interpretation string that can be stored with the property.

`void Writer::endComponent` () [Method]

Closes the declaration of a component started by `beginComponent()`.

`void Writer::endObject` () [Method]

Closes the declaration of an object started by `beginObject()`.

`void Writer::intern` (*const char** **`string`**) [Method]

Declares a string to the Writer for inclusion in the file string table. When writing properties of type `Gto::String`, its necessary to call this function before the `beginData()` is called. Each string in the property data must be interned. When outputing the property, the property will be an array of `Gto::Int` in which each int is the result of the `lookup()` function which retrieves a unique int corresponding to interned strings.

`void Writer::intern` (*const std::string&* **`string`**) [Method]

Same as above, but takes an `std::string`.

`int Writer::lookup` (*const char** **`string`**) [Method]

Retrieve the identifier of the previously interned string *string*.

`int Writer::lookup` (*const std::string&* **`string`**) [Method]

Same as above, but takes an `std::string&`.

`void Writer::beginData` () [Method]

Begins data declaration to the Writer class. Only calls to `lookup()`, `propertyData()`, `propertyDataInContainer()`, and `endData()` are legal after `beginData()` is called.

`void Writer::propertyData` (*const TYPE** **`type`**) [Method]

`propertyData()` is a template function which takes a pointer to continuously stored data. The data must be the same as declared earlier by the `property()` function. Calls to `propertyData()` and `propertyDataInContainer()` must appear in the same order as the `property()` declarations calls.

`void Writer::propertyDataInContainer` (*const TYPE&* **`container`**) [Method]

`propertyDataInContainer()` is a template function which takes an stl-like container as an argument. The data must be the same as declared earlier by the `property()` function. Calls to `propertyData()` and `propertyDataInContainer()` must appear in the same order as the `property()` declarations calls. This function is a convenience function; it calls propertyData() to actually output the data. This function may make a copy of the data in the container.

`void Writer::endData ()` [Method]
Closes the definition of data started by `beginData()` and finishes writing the gto file.

`const Writer::Properties& Writer::properties ()` *const* [Method]
Returns a vector of previously declared properties; these are the result of calls to the `property()` function.

11.3 Gto::RawDataReader/Gto::RawDataWriter classes

These classes provide a quick method of reading the contents of a GTO file into memory for basic editing. The RawDataReader and RawDataWriter both use the same very primitive data structure that can be found in the RawData.h file. For examples of use, see `gtomerge` and `gtofilter` source code.

The RawData class shows how to both read and write using the supplied classes. In addition the reader subclass shows how to convert string data.

12 Python Module

The `gto` module implements a reader/writer library for the Python language. The module is implemented on top of the C++ reader and writer classes. The API is similar to the C++ API, but takes advantage of Python's dynamic typing to "simplify" the design. The Python module also implements a significant number of safety checks not present in the C++ library, making it an ideal way of exploring the Gto file format.

12.1 gto.Reader

The Reader class is designed as a fill-in-the-blank API much like the C++ library. The user of the class derives from it; the base class defines a number of functions which you override to pass data to the derived class and receive data from it.

As the file is read, the Reader class will call specific functions in itself to declare objects in the file. The derived class is handed data or asked to return whether or not it is interested in specific properties in the file.

The biggest difference from the C++ Reader class is that the `data()` method of the C++ class, which returns allocated memory for the library to read data into, cannot be overloaded in Python. Instead, the `dataRead()` method of the Python gto.Reader class is handed pre-allocated Python objects containing the data.

status `gto.Reader (`*mode*`)` [Constructor]
Create a new gto.Reader instance. Possible values for *mode*:

gto.Reader.NONE
The reader will be used in its standard *streaming* mode. The reader will attempt to read all the data in the file. This is the default value (or 0).

gto.Reader.HEADERONLY
The reader will stop once it has read the header sections of the GTO file.

gto.Reader.RANDOMACCESS
The reader will read the header sections but not the data, however, it will initialize for use of the `gto.Reader.accessObject()` method.

gto.Reader.BINARYONLY
The reader will only accept binary GTO files.

gto.Reader.TEXTONLY
The reader will only accept text GTO files.

`gto.Reader.open (`*`filename`*`)` [Method]
Opens and reads the GTO file *filename*. The function will raise a Python exception if the file cannot be opened.

`wants` `gto.Reader.object (`*`name`*`, `*`protocol`*`, `*`protocolVersion`*`,` [Method]
`objectInfo``)`
This function is called by the base class to declare an object in the GTO file. The return value *wants* should evaluate to True or False, indicating whether or not the base class should read the object data. *name* and *protocol* are strings declaring name and protocol of the object, *protocolVersion* is an integer. *objectInfo* is an instance of a generic class which contains the same information as the Gto::ObjectInfo C++ struct.

`wants` `gto.Reader.component (`*`name`*`, `*`interpretation`*`,` [Method]
`componentInfo``)`
This function is called by the base class to declare a component in the GTO file. The return value *wants* should evaluate to True or False, indicating whether or not the base class should read the component data. *name* is a string declaring the component name. *componentInfo* is an instance of a generic class which contains the same information as the Gto::ComponentInfo C++ struct.

`wants` `gto.Reader.property (`*`name`*`, `*`interpretation`*`,` [Method]
`propertyInfo``)`
This function is called by the base class to declare a property in the GTO file. The return value *wants* should evaluate to True or False, indicating whether or not the base class should read the property data. *name* is a string declaring the full property name. *propertyInfo* is an instance of a generic class which contains the same information as the Gto::PropertyInfo C++ struct.

`gto.Reader.dataRead (`*`name`*`, `*`data`*`, `*`propertyInfo`*`)` [Method]
If a property has been requested, the dataRead() function will eventually be called by the base class with the actual data in the file. The *name* is the name of a property, *data* is a tuple containing the property data, *propertyInfo* is an instance of a generic class which contains the same information as the Gto::PropertyInfo C++ struct.

`gto.Reader.stringFromID (`*`id`*`)` [Method]
Returns the stringTable entry for the given string table id. Since the Python gto module returns strings directly, it is unlikely that you'll need to use this.

`gto.Reader.stringTable ()` [Method]
Returns the entire stringTable as a list of strings.

`gto.Reader.isSwapped ()` [Method]
Returns True if the file on disk is not in the machine's native byte order.

`gto.Reader.objects ()` [Method]
Returns a list of the gto.ObjectInfo instances for all the objects in the file. This method is only available if the file was opened with gto.Reader.RANDOMACCESS. Usable at any time after the constructor is called.

`gto.Reader.components ()` [Method]
Returns a list of the gto.ComponentInfo instances for all the components in the file. This method is only available if the file was opened with gto.Reader.RANDOMACCESS. Usable at any time after the constructor is called.

`gto.Reader.properties ()` [Method]
Returns a list of the gto.PropertyInfo instances for all the properties in the file. This method is only available if the file was opened with gto.Reader.RANDOMACCESS. Usable at any time after the constructor is called.

`gto.Reader.accessObject (`*`objInfo`*`)` [Method]
Given an instance of gto.ObjectInfo (obtained via gto.Reader.objects(), gto.Reader.components(), or gto.Reader.properties()), tells the reader to access that object directly. This will cause the gto.Reader.object(), gto.Reader.component(), and gto.Reader.dataRead() methods to be called with the information from the given object.

12.2 gto.Writer

`gto.Writer ( )` [Constructor]
Creates a new writer instance, no arguments needed.

`gto.Writer.open (`*`filename`*`, `*`mode`*`)` [Method]
Open the file. The Writer will attempt to open file *filename*. If the file is not writable for whatever reason, the function will raise a Python exception. The *mode* argument can be `BINARYGTO`, `COMPRESSEDGTO` (the default) or `TEXTGTO`.

`gto.Writer.close ( )` [Method]
Close the file and clean up temporary data. Because of Python's garbage-collection, you can never be sure when a class's destructor will be called. Therefore, it is *highly* recommended that you call this method to close your file when it's done writing. You have been warned.

`gto.Writer.beginObject (`*`name`*`, `*`protocol`*`, `*`version`*`)` [Method]
Declares an object. Its components and properties must be declared before endObject() is called. The *name* is the name of the object as it will appear in the gto file. The *protocol* is the protocol string indicating how the object data will be interpreted and the *version* number indicates the protocol version. The Writer class does not verify that the data output conforms to the protocol.

`gto.Writer.beginComponent (`*`name`*`, `*`interpretation`*`, `*`transposed`*`)` [Method]
Declares a component. The component properties must be declared before a call to endComponent(). The *name* is the name of the component as it will appear in the gto file. The *transposed* flag is optional and indicates whether or not the component property data should be output transposed or one property at a time (the default).

`gto.Writer.property (`*`name`*`, `*`type`*`, `*`numElements`*`, `*`partsPerElement`*`, `*`interpretation`*`)` [Method]

Declare a property. The *name* is the name of the property as it appears in the gto file. The *type* is one of gto.DOUBLE, gto.FLOAT, gto.INT, gto.STRING, gto.BYTE, gto.HALF (*Not implemented*), or gto.SHORT. *numElements* indicates the number of elements of size *partsPerElement* that will be in the property data. So for example, if the property is declared as a gto.FLOAT of with *partsPerElement* of 3 and there 10 of them, then the writer will expect a sequence of 30 floats when the propertyData is finally passed to it.

`gto.Writer.endComponent ()` [Method]

Closes the declaration of a component started by beginComponent().

`gto.Writer.endObject ()` [Method]

Closes the declaration of an object started by beginObject().

`gto.Writer.intern (`*`string`*`)` [Method]

Declares a string to the Writer for inclusion in the file string table. When writing properties of type gto.String, its necessary to call this function for each string in the property data before the beginData() is called. The Python version of intern() can accept individual strings, as well as lists or tuples of strings.

`int gto.Writer.lookup (`*`string`*`)` [Method]

Retrieve the identifier of the previously interned string. Valid only after beginData() has been called.

`gto.Writer.beginData ()` [Method]

Begins data declaration to the Writer class. Only calls to lookup(), propertyData(), and endData() are legal after beginData() is called.

`gto.Writer.propertyData (`*`data`*`)` [Method]

The propertyData() function must get exactly *one* parameter. That parameter can be any of the following:

- A single int, float, string, etc.
- An instance of mat3, vec3, mat4, vec4, or quat (`http://cgkit.sourceforge.net/`). DO NOT explicitly cast mat3 or mat4 into a tuple or list: `tuple(mat4(1))`. It will be silently transposed (a bug in the cgtypes code?). ADDING it to a tuple or list is fine: `(mat4(1),)`
- A tuple or list of any combination of the above that makes sense.

Tuples and lists are flattened out before they are written. As long as the number of atoms is equal to size x width, it'll work. Calls to propertyData() must appear in the same order as declared with the property() method.

`void gto.Writer.endData ()` [Method]

Closes the definition of data started by beginData() and finishes writing the gto file. Does *not* actually close the file–use the close() method for that.

12.3 Classes used by gto.Reader

Note that as of the 3.0 release, these classes will contain the actual strings rather than string table IDs.

`gto.ObjectInfo` [Class]

This class emulates the Gto::ObjectInfo struct from the C++ Gto library. It is passed by the Python gto.Reader class to your derived `object()` method. The only methods implemented are `__getattr__` and `__repr__`. Available attributes are:

- `name` - String
- `protocolName` - String
- `protocolVersion` - Integer
- `numComponents` - Integer
- `pad` - Integer

`gto.ComponentInfo` [Class]

This class emulates the Gto::ComponentInfo struct from the C++ Gto library. It is passed by the Python gto.Reader class to your derived `component()` method. The only methods implemented are `__getattr__` and `__repr__`. Available attributes are:

- `name` - String
- `numProperties` - Integer
- `flags` - Integer
- `interpretation` - String
- `pad` - Integer
- `object` - Instance of gto.ObjectInfo

`gto.PropertyInfo` [Class]

This class emulates the Gto::PropertyInfo struct from the C++ Gto library. It is passed by the Python gto.Reader class to your derived `property()` and `dataRead()` methods. The only methods implemented are `__getattr__` and `__repr__`. Available attributes are:

- `name` - String
- `size` - Integer
- `type` - Integer
- `width` - Integer
- `interpretation` - String
- `pad` - Integer
- `component` - Instance of gto.ComponentInfo

13 Utilities

13.1 The `gtoinfo` Utility

Usage: `gtoinfo [OPTIONS] infile.gto`

Options:

`-a/-all` Output property data and header.

`-d/--dump`
: Output property data (no header data is emitted).

`-l/--line`
: Output property data one item per line. Can be used with either `-d` or `-s`.

`-h/--header`
: Output header data.

`-s/--strings`
: Output sting table data.

`-n/--numeric-strings`
: Output sting data as the raw string id instead of the string itself.

`-i/--interpretation-strings`
: Output interpretation string data for components and properties if it exists.

`-r/--readall`
: Force reading of the enitre gto file even if only the header is being output.

`-f/--filter expression`
: Only output information for properties who's long name (object.component.propname) matches the shell-like *expression*. Section 13.2 [gtofilter], page 43 for examples of filter expressions. This option is similar to `gtofilter --include` option.

`--help` Output usage message.

`gtoinfo` outputs the part of all of the contents of a gto file in human readable form. Its invaluable for debugging or just getting a quick understanding of what a gto file contains.

13.2 The `gtofilter` Utility

Usage: '`gtofilter [OPTIONS] -o out.gto in.gto`'

Options:

`-v` Set verbose output. Whenever a pattern matches gtofilter will inform you.

`-ee/--exclude`
: Regular expression which will be used to exclude properties.

`-ie/--include`
: Regular expression which will be used to include properties.

`-regex` Use POSIX regular expression syntax.

`-glob` Use shell-like regular expression (fnmatch). This is the default.

`-t` Output text GTO file.

`-nc` Output uncompressed binary GTO file.

`-o` *`out.gto`*
Output .gto file

`gtofilter` can be used to remove objects, components, and properties from a gto file. You supply an include shell-like expression and/or an exclude shell-like expression. (The pattern matching is done using the fnmatch() function – see the man page for details.)

The patterns match each full property name. So for example a cube might have these properties:

```
cube.points.position
cube.elements.type
cube.elements.size
cube.indices.vertex
cube.indices.st
cube.indices.normal
cube.normals.normal
cube.mappings.st
cube.smoothing.method
cube.object.globalMatrix
cube.object.parent
```

Using the `--exclude` option, you can remove the object component by doing this:

```
gtofilter --exclude "*.object.*" -o out.gto cube.gto
```

or if you wanted to pass through only the positions:

```
gtofilter --include "*.*.positions" -o out.gto cube.gto

    -or-

gtofilter --include "*positions" -o out.gto cube.gto
```

13.3 The `gtomerge` Utility

Usage: '`gtomerge -o` *`outfile.gto infile1.gto infile2.gto ...`*'

Options:

`-o outfile.gto`
The resulting merged file to output.

`-t` Ouput text GTO file.

`-nc` Ouput uncompressed binary GTO file.

`gtomerge` takes a number of .gto input files and merges them into a single output .gto file. This is done by first creating output geometry that is identical to the first input file and then adding only those properties that are not already defined from subsequent gto files. The order of input files determines what will be in the final output file.

For **difference** files, you can use gtomerge to reconstruct a final file like this:

```
gtomerge -o out.gto difference.gto reference.gto
```

13.4 The gto2obj Utility

Usage: '`gto2obj [OPTIONS] infile outfile`'

Options:

`-o NAME` When outputing GTO files, the name of an object in the GTO file to output. If not specified, the translator will output the first polygon, or subdivision surface it finds.

`-c` When outputing GTO files, this option will force the protocol to be "catmull-clark".

`-l` When outputing GTO files, this option will force the protocol to be "loop".

`-t` Ouput text GTO file.

`-nc` Ouput uncompressed binary GTO file.

`gto2obj` takes either an input GTO file or Wavefront .obj file and outputs the other file type.

```
gto2obj in.obj out.gto
gto2obj in.gto out.obj
gto2obj -c in.obj out.gto ## output obj as subdivision surface
```

13.5 The gtoimage Utility

Usage: '`gtoimage infile outfile`'

`-t` Ouput text GTO file.

`-nc` Ouput uncompressed binary GTO file.

`gtoimage` reads a TIFF file and converts it into a GTO file containing one image object. 32 bit floating point images, 16 bit and 8 bit integral images are directly converted. `gtoimage` expects the image to be two dimensional with three or four channels where the fourth channel is an optional alpha value. The output object conforms to the **image** protocol. See Section 7.8 [Image], page 23.

You can use `gtomerge` to merge the image object into another GTO file. See Section 13.3 [gtomerge], page 44.

It is highly recommend that the resulting output GTO file be written with compression or gzipped to reduce its size. Gzipped GTO files can be read directly by the supplied readers.

13.6 The RiGtoRibOut Utility

The `RiGtoRibOut` command is useful for:

- It can be used as a debugging tool for the RiGtoPlugin RenderMan plugin.
- It can be used as a drop-in replacement for RiGtoPlugin, for RIB renderers that do not support `Procedural DynamicLoad`, but that *do* support `Procedural RunProgram`. Note that this is substantially slower than using RiGtoPlugin, as all data needs to be

translated to ASCII and back. It does have the one space-saving advantage of not needing to save ASCII RIB on disk.

- It could be used to generate RIB files that are read with `ReadArchive`. This is not recommended, as it negates all the advantages of using GTO in the first place. But if nothing else works, this should.

The command-line parameters are the same as the *CONFIG_STRING* for RiGtoPlugin. See Section 13.8 [RiGtoPlugin], page 46.

13.7 The `gtoIO.so` Maya Plug-In

The Maya plugin comes in two parts: the C++ plugin which implements a Maya scene translator and an accompanying MEL script which implements the user interface.

The plugin handles export of NURBS surfaces (but not trim curves), polygonal geometry (which can be written as sub-division surfaces), and generic transforms. A Maya particle export tool is in the works. Additional user defined attributes can be emitted into the GTO file.

The plugin can import everything that it exports and also particle GTO files generated by other applications.

13.7.1 BUGS

The internal perfomance of Maya has changed between the 4.x and 5.0 versions. In Maya 5.0, the Maya API is *extremely* slow when importing polygonal normals. Importing of normals is disabled in Maya 5.0.

13.8 The `RiGtoPlugin` RenderMan plugin

Here you will find information on using the GTO RenderMan plugin. The documentation is complete enough to get started with, but should be considered a work in progress.

13.8.1 RIB Instantiation

The plugin is instantiated in a RIB Stream using the standard DynamicLoad procedural mechanism, like so:

```
Procedural "DynamicLoad" [ "RiGtoPlugin.so" "CONFIG_STRING" ] [ Bounding Box ]
```

If a bounding box is not known, the infinite box may be used:

```
Procedural "DynamicLoad" [ "RiGtoPlugin.so" "CONFIG_STRING" ] [ -1e6 1e6 -1e6 1e6 -1e6
```

13.8.2 Config String Syntax

The configuration string passed into RiGtoPlugin consists of a variable number of space-separated tokens. They are, in order:

1. Reference Pose GTO File Name
2. Shutter Open GTO File Name (optional)
3. Shutter Close GTO File Name (optional)
4. Primary On List (optional)
5. Primary Off List (optional)

6. Secondary On List (optional)
7. Secondary Off List (optional)

As shown, the only necessary element is the reference GTO file. For objects which do not have movement and do not require on lists or off lists, this is completely sufficient.

The logic behind the geometry instantiation mechanism is as follows:

- Read Reference GTO file The plugin reads all of the geometry in the reference GTO file. As a starting point, the shutter open and close geometry is set equal to the reference geometry.
- If requested, read Shutter Open GTO file The plugin then reads any geometry from the Shutter Open file that matches the name and geometry type of geometry that has already been read from the reference file - this geometry is stored as both the shutter open AND close geometry.
- If requested, read Shutter Close GTO file The plugin then reads any geometry from the Shutter Close file that matches the name and geometry type of geometry that has already been read from the reference file - this geometry is stored as the shutter close geometry
- Instantiate Geometry: For any piece of geometry that appears in BOTH on-lists and does not appear in EITHER off-lists, the plugin calls the appropriate RIB functions to create the requested geometry.

13.8.3 On-List/Off-List Syntax

The syntax of the on-lists and off-lists is as follows:

`NULL` is a special on-list/off-list which is interpreted as *all on* or *none off*.

Otherwise, the on-lists and off-lists are essentially shell-like regular expressions. The following rules apply:

- The `*` character matches any number of characters
- The `?` character matches any single of character
- Bracket expressions `[]` are supported. (See `man 7 regex`)
- Multiple patterns can be strung together with the `|` character.
- The pattern must match the *whole* object name. Thus, the pattern `"*Sphere1"` will match the object `nurbsSphere1` but *not* `nurbsSphere1Shape`. This is a very common "gotcha".

As an example, suppose you wanted to turn off all of the geometry named `LeftLeg*Shape*` and `RightLeg*Shape*` in a render - you would create an off-list that looked like:

```
"LeftLeg*Shape*|RightLeg*Shape*"
```

13.8.4 Cache Management

By default, RiGtoPlugin maintains an internal cache of all of the file sets it has read. The cache is keyed off of Ref-Open-Close filename triplets. The reason for this is to facilitate easy material assignment, which will be discussed in greater detail below in the "Strategy" section.

In situations where memory is precious and the renderer needs as much memory as it can get, it may be advantageous to force RiGtoPlugin to discard its cached file sets. There is special syntax to facilitate this.

- To erase everything in RiGtoPlugin's cache:

  ```
  Procedural "DynamicLoad" [ "RiGtoPlugin.so" "__FLUSH__" ] [ Bounding Box ]
  ```

- To erase the cache associated with a given file triplet: (Using REF.gto, OPEN.gto and CLOSE.gto as standins for whatever files were actually passed in)

  ```
  Procedural "DynamicLoad" [ "RiGtoPlugin.so" "REF.gto OPEN.gto CLOSE.gto __FLUSH__"
  ing Box ]
  ```

There is also an environment variable, `TWK_RI_GTO_NO_CACHE`, which if defined and set to anything other than "0", "FALSE", "False" or "false", will cause caching to be disabled entirely.

13.8.5 Environment Variables

`TWK_RI_GTO_NO_SUBDS` [Environment Variable]

If this environment variable is defined and set to anything except "0", "FALSE", "False", or "false", RiGtoPlugin will treat all catmull-clark subdivision surfaces read from a GTO file as polygons instead.

`TWK_RI_GTO_NO_CACHE` [Environment Variable]

If this environment variable is defined and set to anything except "0", "FALSE", "False", or "false", RiGtoPlugin will disable all caching of geometry data to save memory.

13.8.6 Usage Strategy

The RiGtoPlugin was designed with a particular data structure in mind. Used ideally, there would be a GTO file consisting of all of the geometry corresponding to a particular high-level creature or set in the scene. All of the surfaces corresponding to a hippo or a giraffe or a cyborg-monkey would be in a single GTO file. The animation data for this geometry would be contained in light-weight GTO files that contain only points that have moved and transformation matrices that have moved. The RiGtoPlugin only reads points and matrices from the Shutter-Open and Shutter-Close file, facilitating very light-weight "difference" files for animation data.

Because all of the geometry in a creature will have different materials assigned to it, on-lists and off-lists can be used to separate out only the geometry that shares a particular material.

Suppose we have a creature consisting of many surfaces but only three different materials - skinMtl, eyeMtl and hairMtl. The parts of the model have been named intelligently (for this example) such that the skin parts all have names like Skin*Shape*, the eye parts all have names like Eye*Shape*, and the hair parts all have names like Hair*Shape*. Then, the RIB for declaring this creature with material assignments might look like this:

```
AttributeBegin
Surface "skinShader" [ shader param settings ]
Procedural "DynamicLoad" [ "RiGtoPlugin.so" "thing.ref.gto thing.0013.open.gto thing.0
```

```
AttributeEnd

AttributeBegin
Surface "hairShader" [ shader param settings ]
Procedural "DynamicLoad" [ "RiGtoPlugin.so" "thing.ref.gto thing.0013.open.gto thing.0
AttributeEnd

AttributeBegin
Surface "eyeShader" [ shader param settings ]
Procedural "DynamicLoad" [ "RiGtoPlugin.so" "thing.ref.gto thing.0013.open.gto thing.0
AttributeEnd
```

Because of RiGtoPlugin's cache mechanism, the geometry associated with the file-set thing.*.gto is only read and interpreted one time - the on-lists control which parts of the geometry are instantiated at which times. To nuke the cache of these files (if memory is important), you would use the syntax:

```
Procedural "DynamicLoad" [ "RiGtoPlugin.so" "thing.ref.gto thing.0013.open.gto thing.0
```

13.8.7 Miscellaneous RenderMan Stuff

RiGtoPlugin stores some useful data in attributes that can be used by shaders if desired.

`Pref` [Shader parameter]
: On ALL geometry RiGtoPlugin creates "varying point Pref" as part of its geometry declaration. This data can be accessed by simply putting "varying point Pref" in your shader parameters. The position of the model in the reference GTO file is always used for this parameter value.

`Name` [RIB Attribute]
: RiGtoPlugin always places the name of the geometry, as retrieved from the GTO file, in an attribute that may be queried. It is exactly as if the following line of RIB were declared before the geometry were instantiated:

```
Attribute "identifier" "name" ["whatever my name is"]
```

`RefToWorld` *matrix* [RIB Attribute]
: RiGtoPlugin places the transformation matrix *objectToWorld* from the reference model into a user attribute called `refToWorld`. To prevent this attribute from being munged by the current transformation matrix, it is cast as a float[16] instead of a matrix. It is equivalent to this line of RIB:

```
Attribute "user" "float refToWorld[16]" [ the matrix values ]
```

Appendix A Description of Changes

- Version 4.0
  - Property data size can now be larger than 4Gb in the file and readable in full on 64 bit architectures (if the library is compiled 64 bit). The file headers are now version 4. The new file is incompatible with version 3 GTO readers.

    - The manual has been updated with real-world usage examples from the film and game industries.
- Version 3.4
    - The GTO license terms have been changed: the code is still covered by the LGPL, but with additional **exceptions** similar to those used by the FLTK library. These exceptions make it easier to use GTO in commercial projects.
    - Houdini I/O plugin (ggto) reads and writes GTO geometry.
    - The library no longer attempts to be source code compatible with older Microsoft compilers. Some functions may throw on error.
    - Maya plugin (loadGtoAnim) loads animation from GTO files (transform matrices only). Useful for getting animation from GTO difference files.
    - Maya plugin (GtoDeformer) makes it possible to use GTO files as geometry cache files. The deformer will read vertex/cv positions from existing GTO files and applies them to scene geometry of the same name.
    - Maya plugin (GtoParticleDisplay) loads particles from GTO files for viewing in Maya
    - Maya plugin (GtoParticleExport) writes particles from Maya as GTO files. Can be used as a replacement for pdb and pdc files.
    - Maya plugin (GtoCacheEmitter) loads particles into Maya from GTO files via an emitter.
    - Bug fixes to the C++ Gto::Writer class for output of text GTO files.
    - Run-time error checking of the Gto::Writer API. The class will complain if the API is used in a undocumented manner. It may throw an exception.
    - The GTO source code distribution now comes with Maya and Houdini plugins for cached deforming geometry and particle export and display.
- Version 3.2
    - Human readable plain text version of GTO. Some readers may not function if they assume that the property size is known when the property() virtual function is called. The property size is only really known when the data() virtual function is called. Only version 3.2 GTO readers can read the text version.
    - Animation curves are now stored per-object using the animation protocol.
    - Bug fixes to Gto::Reader class to allow reuse of existing class with a newly opened file.
- Version 3.1
    - RenderMan plug-in documentation added.
- Version 3.0
    - An interpretation string has been added to the property header.
    - An additional uint32 has been added as padding to the object, component, and property headers for future expansion slop.
    - A section on interpretation strings has been added to the documentation and to the reader/writer classes.
    - Added a type reference to the documentation.

- Version 2.1
  - `gtofilter` was changed to optionaly accept POSIX style regular expressions in addition to shell-like "glob" expressions.
  - The C++ writer class now defaults to writing compressed files when the open() function is called. A second bool argument can be passed to it to prevent the compression.
  - The proposed texture assignment protocol (from version 2.0.4) has been rejected.
  - A new protocol "channel" is introduced for assigning mapped surface varying data on geometry. An arbitrary number of texture maps may be assigned to the geometry. See Section 7.14 [Channels], page 26.
  - The material protocol has been fleshed out. See Section 7.9 [Material], page 24.
  - The polygon protocol was missing the definition of the optional **mappings** component. See Section 7.6 [Polygonal Surfaces], page 20.
- Version 2.0.5 Bug fix version. Repaired problems with the configuration scripts. Missing headers.
- Version 2.0.4 Bug fix version. Some configuration problems solved.
- Version 2.0 File headers changed. The format is not compatible with 1.0.
- Version 1.0

Appendix B Reference

Properties

Functions

G

R

W

Types