Blorb: An IF Resource Collection Format Standard

Version 2.0.4

Andrew Plotkin <erkyrath@eblong.com>

This is a formal specification for a common format for storing resources associated with an interactive fiction game file. Resources are data which the game can invoke, such as sounds and pictures. In addition, the executable game file may itself be a resource in a resource file. This is a convenient way to package a game and all its resources together in one file.

Blorb was originally designed solely for the Z-machine, which is capable of playing sounds (Z-machine versions 3 and up) and showing images (the V6 Z-machine). However, it has been extended for use with other IF systems. The Glk portable I/O library uses Blorb as a resource format, and therefore so does the Glulx virtual machine. (See <http://eblong.com/zarf/glk/> and <http://eblong.com/zarf/glulx/>.) ADRIFT 5 (see <http://www.adrift.org.uk/>) also uses Blorb, albeit with an extended format list.

This format is named "Blorb" because it wraps your possessions up in a box, and because the common save file format was at one point named "Gnusto". That has been changed to "Quetzal", but I'm not going to let that stop me.

This proposal is longer than I would have liked. However, a large percentage of it is optional stuff — optional for either the interpreter writer, the game author, or both. That may make you feel better. I've also put in lots of examples, explication, and self-justification.

0. Overall Structure

The overall format will be a new IFF type. The FORM type is 'IFRS'.

The first chunk in the FORM must be a resource index (chunk type 'RIdx'.) This lists all the resources stored in the IFRS FORM. There must be exactly one resource index chunk.

The resources are stored in the FORM as chunks; each resource is one chunk. They do not need to be in any particular order, since the resource index contains all the information necessary to find a particular resource.

There are several optional chunks which may appear in the file: the release number (chunk type 'RelN'), the game identifier (chunk type 'IFhd'), and others defined hereafter. They may occur anywhere in the file after the resource index.

Several optional chunks may also appear by convention in any IFF FORM: '(c) ', 'AUTH', and 'ANNO'. These may also appear anywhere in the file after the resource index.

1. Contents of the Resource Index Chunk

```
4 bytes         'RIdx'          chunk ID
4 bytes         n               chunk length (4 + num*12)
4 bytes         num             number of resources
num*12 bytes    ...             index entries
```

There is one index entry for each resource. (But not for the optional chunks.) Each index entry is

12 bytes long:

```
4 bytes         usage           resource usage
4 bytes         number          number of resource
4 bytes         start           starting position of resource
```

The index entries should be in the same order as the resource chunks in the file.

The usage field tells what kind of resource is being described. There are currently four values defined:

- ’Pict’: Picture resource
- ’Snd ’: Sound resource
- ’Data’: Data file resource
- ’Exec’: Code resource

The number field tells which resource is being described, from the game’s point of view. For example, when a Z-code game calls @draw_picture with an argument of 3, the interpreter would find the index entry whose usage is ’Pict’ and whose number is 3. For code chunks (usage ’Exec’), the number should contain 0.

The start field tells where the resource chunk begins. This value is an offset, in bytes, from the start of the IFRS FORM (that is, from the start of the resource file.)

Note that the start field must refer to the beginning of a chunk. It is not strictly required for each resource to refer to a different chunk.

2. Picture Resource Chunks

Each picture is stored as one chunk, whose content is a PNG file, a JPEG (JFIF) file, or a placeholder rectangle. (Note that these are various possible formats for a single resource. It is not possible to have a PNG image and a JPEG image with the same image resource number.)

2.1. PNG Pictures

A PNG resource has a chunk type of ’PNG ’.

PNG is a lossless image compression format. The PNG file format is available at

```
<http://www.libpng.org/pub/png/>
```

2.2. JPEG Pictures

A JPEG resource has a chunk type of ’JPEG’.

JPEG is a lossy image compression format, developed for photograph-like images. For information on JPEG, see

```
<http://www.jpeg.org/jpeg/>
```

2.3. Placeholder Pictures

A third form of picture resource is a placeholder rectangle. A rectangle has only size, but no contents. This format exists to describe the legacy behavior of some V6 Infocom games (Zork Zero, Shogun, and Arthur). Its support in interpreters is optional, and its use is strongly discouraged for any purpose other than conversions of Infocom graphics.

```
4 bytes         'Rect'          chunk ID
4 bytes         8               chunk length
4 bytes         width           rectangle width
4 bytes         height          rectangle height
```

Either or both of the width and height may be zero.

In a Z-code game, a rectangle exists for the purposes of @picture_data and @erase_picture, but its use in @draw_picture or @picture_table is an error. The behavior of rectangles in Glulx and other game files is not defined.

[Thanks to Kevin Bracey for this extension.]

3. Sound Resource Chunks

Each sound is stored as one chunk, whose content is either an AIFF file, an Ogg file, a MOD file, or a song file. (Note that these are various possible formats for a single resource. It is not possible to have an AIFF sound and a MOD sound with the same sound resource number.)

On the Z-machine, we must consider the problems of how the game knows the interpreter can play music, and how sampled sounds are played over music. See the section "Z-Machine Compatibility Issues" later in this document. (These issues are not relevant to Glk and Glulx.)

3.1. AIFF Sounds

An AIFF (Audio IFF) file has chunk type 'FORM', and formtype 'AIFF'. AIFF is an uncompressed digital-sample format developed in the late 1980s. The AIFF format is available at these locations:

```
<http://www.digitalpreservation.gov/formats/fdd/fdd000005.shtml>
<http://eblong.com/zarf/ftp/aiff-c.9.26.91.ps>
```

3.2. Ogg Sounds

An Ogg Vorbis file has chunk type 'OGGV'. This is a high-quality (but lossy) audio compression format, comparable to MP3 (but without the patent concerns that encumber MP3). The Ogg format is available at:

```
<http://www.vorbis.com/>
```

3.3. MOD Sounds

MOD is an Amiga-originated format for music synthesized from note samples. Over the years, other formats of this type — generally called "tracker" or "module music" formats — have arisen. Blorb supports four: original ".MOD" files, ImpulseTracker (".IT"), FastTracker 2 Extended (".XM"), and ScreamTracker 3 (".S3M").

Because tracker-playing libraries typically handle many formats, it is most practical for Blorb to lump them all together. *Regardless of which tracker format is used*, the chunk type will be 'MOD '.

The formats are described here:

```
<http://www.digitalpreservation.gov/formats/fdd/fdd000126.shtml>
```

This spec does not attempt to distinguish variations within the four supported formats. (".MOD" is particularly ill-defined, although I have saved comments on the original MOD format at <http://eblong.com/zarf/blorb/mod-spec.txt>.) Instead, we recommend that C implementations embed libmodplug, a public-domain tracker-playing library. Its home page is:

```
<http://modplug-xmms.sourceforge.net/>
```

However, some bug fixes are included in the version packaged with Windows Glk:

```
<http://ifarchive.org/if-archive/programming/glk/implementations/>
```

(Note that it may be safer to compile libmodplug with the MODPLUG_BASIC_SUPPORT option, which eliminates many obscure tracker formats that Blorb does not support.)

Where libmodplug is not practical, implementations should use whatever tracker-playing library claims to support the four formats in question. We trust, perhaps beyond reason, that implementation differences will not lead game creators to their doom.

3.4. Song Sounds

The song file format is deprecated, as of Blorb 2.0. It is complicated, non-standard, and hard to use. Its support in interpreters should be considered optional. However, it will continue to be documented here.

A song file has chunk type 'SONG'. This is similar to a MOD file, but with no built-in sample data. The samples are instead taken from AIFF sounds in the resource file. For each sample, the 22-byte sample-name field in the song should contain the string "SND1" to refer to sound resource 1, "SND953" to refer to sound resource 953, and so on. Any sound so referred to must be an AIFF, not a MOD or song. (You can experiment with fractal recursive music on your own time.)

Each sample record in a MOD or song contains six fields: sample name, sample length, finetune value, volume, repeat start, repeat length. In a MOD file, the sample name is ignored by Blorb (it is traditionally used to store a banner or comments from the author.) In a song file, the sample name contains a resource reference as described above; but the sample length, repeat start, and repeat length fields are ignored. These values are inferred from the AIFF resource. (The repeat start and repeat length are taken from the sustainLoop of the AIFF's instrument chunk. If there is no instrument chunk, or if sustainLoop.playMode is NoLooping, there is no repeat; the repeat start and length values are then considered zero.)

Note that an AIFF need not contain 8-bit sound samples, as a sound built into a MOD would. A clever sound engine may take advantage of this to generate higher-quality music. An unclever one can trim (or pad) the AIFF's data to 8 bits before playing the song. In the worst case, it is always possible to trim the AIFF data to 8 bits, append it to the song data, fill in the song's sample records (with the appropriate lengths, etc, from the AIFF data); the result is a complete MOD file, which can then be played by a standard MOD engine.

The intent of allowing song files is both to allow higher quality, and to save space. Note samples are the largest part of a MOD file, and if the samples are stored in resources, they can be shared between songs. (Typically note samples will be given high resource numbers, so that they do not conflict with sounds used directly by the game. However, it is legal for the game to use a note sample as a sampled-sound effect, if it wants.)

4. Data Resource Chunks

Each data file is stored as one chunk, with chunk type 'TEXT' or 'BINA' (denoting text or binary data). The format and contents are up to the game to interpret.

This feature was designed to support Glulx, but data resources can be accessed by any game format if the interpreter supports them.

For Glulx games (and any other game format which uses the Glk API), the data format must follow the conventions described in the Glk spec. (<http://eblong.com/zarf/glk/>, "Resource Streams".)

[To summarize: if the data file is opened via glk_stream_open_resource(), then it will be read as a stream of bytes; text will be assumed to be encoded as Latin-1. If it is opened via glk_stream_open_resource_uni(), then a 'TEXT' chunk will be assumed to be a stream of characters encoded as UTF-8; 'BINA' will be assumed to be a stream of big-endian four-byte integers. If read by lines (glk_get_line_stream(), etc), resource text should use Unix line breaks in all cases.]

5. Executable Resource Chunks

There should at most one chunk with usage 'Exec'. *[But see below.]* If present, its number must be zero. Its content is a VM or game executable. Its chunk type describes its format:

- 'ZCOD': Z-code
- 'GLUL': Glulx
- 'TAD2': TADS 2
- 'TAD3': TADS 3
- 'HUGO': Hugo
- 'ALAN': Alan
- 'ADRI': ADRIFT
- 'LEVE': Level 9
- 'AGT ': AGT
- 'MAGS': Magnetic Scrolls
- 'ADVS': AdvSys
- 'EXEC': Native executable

[This list of formats is taken from the Babel format agreement. See <http://babel.ifarchive.org/> for more information. Most of these development systems do not support Blorb at the present time; the list is available for future use. Other executable formats may also be added in the future. As a convention, the chunk types should be taken from the Babel format name, converted to upper case and padded (if necessary) with spaces.]

[The EXEC (native) chunk type is not likely to be useful, because it is underspecified. Nothing (beyond the chunk data itself) indicates what CPU or operating system the executable is intended for. Again, it is defined here following the Babel format list.]

A resource file which contains an executable chunk contains everything needed to run the executable. An interpreter can begin interpreting when handed such a resource file; it sees that there is an executable chunk, loads it, and runs it.

A resource file which does not contain an executable chunk can only be used in tandem with an executable file. The interpreter must be handed both the resource file and the executable file in order to begin interpreting.

If an interpreter is handed inconsistent arguments — that is, a resource file with no executable chunk, or a resource file with an executable chunk plus an executable file — it should complain righteously to the user.

5.1. Multiple Executable Chunks

As of this spec, no IF system puts more than one 'Exec' chunk in a Blorb file, or has any need to. However, this could change in the future.

One possible use (noted as a comment in earlier versions of this spec) is to support several loadable libraries or game segments. In such a case, chunk zero should contain the code to execute first, or at the top level.

Another possibility is to distribute several versions of a game in one Blorb package. IF platforms are famed for the fragility of their save files; a player who downloads an updated game file is likely to find that it no longer loads her old saved games. This could be avoided if the updated Blorb actually contained multiple game files, one per 'Exec' chunk. Chunk zero would be the preferred (most recent) game version, but when loading a save file, the interpreter would select whichever game version was compatible with it.

6. The Game Identifier Chunk

This identifies which game the resources are associated with. The chunk type is 'IFhd'.

This chunk is optional; at most one should appear. If it is present, and the interpreter is given a game file along with a resource file, the interpreter can check that the game matches the IFhd chunk. If they don't, the interpreter should display an error. The interpreter may want to provide a way for the user to ignore or skip this error (for example, if the user is a game author testing changes to the game file.)

If the resource file contains an executable chunk, there is little reason to have an IFhd chunk. It is legal, however, as long as the identifier matches the executable.

For Z-code, the contents of the game identifier chunk are defined in the common save file format specification, section 5. This spec can be found at

```
<http://ifarchive.org/if-archive/infocom/interpreters/specification/savefile_14.txt>
```

The "Initial PC" field of the IFhd chunk (bytes 10 through 12) has no meaning for resource files. It should be set to zero.

For Glulx, the contents of the game identifier chunk are defined in the Glulx specification. This can be found at <http://eblong.com/zarf/glulx/>.

7. The Color Palette Chunk

This contains information about which colors are used by picture resources in the file. The chunk type is ’Plte’. It is optional, and should not appear if there are no ’Pict’ resources in the file. At most one color palette chunk should appear.

The format is:

```
4 bytes         'Plte'          chunk ID
4 bytes         n               chunk length
n bytes         ...             color data
```

There are two possibilities for the color data format. The first is an explicit list of colors. In this case, the data consists of 1 to 256 color entries. Each entry is three bytes, of the form:

```
1 byte         red value (0 = black, 255 = red)
1 byte         green value (0 = black, 255 = green)
1 byte         blue value (0 = black, 255 = blue)
```

The second case is a single byte, which may have either the value 16 or 32 (decimal). 16 indicates that the picture resources are best displayed on a direct-color display which has 16 or more bits per pixel (5 or more bits per color component.) 32 indicates that the resources are best displayed with 32 or more bits per pixel (8 or more bits per color component.)

The two cases are differentiated by checking the chunk length (n). If n is 1, it’s a direct color value; if it’s a positive multiple of 3, it’s a color list, and the number of entries is the length divided by 3. Any other length is illegal.

This chunk is only a hint; there is no guarantee about what the interpreter will do with it. A color list will most likely be useful if the interpreter’s display can only display a limited number of colors (for example, an 8-bit indexed color device). The interpreter may set the display to the colors listed in the palette. Or it may set the display to just some of the colors listed (for example, if it wishes to reserve some colors for text display, or if it just doesn’t have enough colors available.) Or the interpreter may ignore the palette chunk, or do something else.

Similarly, if the interpreter finds a "16" or "32" value, it may set the display to the appropriate bit depth. Or it may set the display to an 8-bit color cube, and dither the images for display. Or, again, it may ignore the palette chunk entirely, or do something else.

It is not required that the palette chunk list every color used in the ’Pict’ resources. It is not required that the colors in the palette all be different, or that they all are actually used by ’Pict’ resources. It is not required that the palette have anything to do with the game art at all. Of course, if you give the interpreter misleading hints, you deserve whatever you get.

8. The Frontispiece Chunk

The Blorb format generally does not specify how images are loaded and displayed; that is the province of the game file format. However, it may be desirable to associate a single image with the game. The image would serve as a frontispiece, or "cover art".

The exact use of a frontispiece image is left open to invention. An interpreter may display it before starting a game. Or it might display frontispieces while the player is *choosing* a game to play (as an aid to locating a particular game). An index of games might extract the frontispieces and use them as catalog illustrations.

If present, the frontispiece is simply an ordinary picture resource. It is singled out as a frontispiece by a chunk with type ’Fspc’; this contains its image resource number. There may not be more than one ’Fspc’ chunk.

The frontispiece image may be of any legal Blorb type (except a placeholder rectangle). The image may be of any size, but is preferred to be square or approximately so. This allows interpreters to display frontispieces in a systematic way, scaling them to fit a layout, without wasting screen space.

(Since the frontispiece image is not loaded by the game file, it may be used even with game files that do not support graphics, such as the V5 Z-machine. In a graphics-capable game file, it is legal for the frontispiece image to also be loaded by the game file in the usual way.)

```
   4 bytes         ’Fspc’          chunk ID
   4 bytes         4               chunk length
   4 bytes         number          number of a Pict resource
```

9. The Resource Description Chunk

For a game to be fully accessible to visually impaired users, it should provide textual descriptions to be displayed as alternatives to images. Similarly, audio resources should have textual descriptions as alternatives for hearing-impaired users.

The resource description chunk allows the Blorb file to offer this information.

```
   4 bytes         ’RDes’          chunk ID
   4 bytes         len             chunk length
   4 bytes         num             number of entries
                   ...             entries
```

The entries are variable-length, and look like:

```
   4 bytes         usage           resource usage
   4 bytes         number          number of resource
   4 bytes         length          length of text (bytes)
   length bytes    text            textual description (UTF-8,
                                     not null-terminated)
```

There should be at most one entry for each resource -- that is, each (usage, number) pair.

Resource descriptions are not required, but they are recommended for significant sounds and images. (Images used for decoration, such as window borders or text dividers, may not need textual descriptions.) Data and executable chunks do not need descriptions; if they appear in this chunk, the interpreter can ignore them.

[An interpreter with a web interface would apply the textual description of an image as an "alt" attribute on the <img> tag.]

10. Metadata

Metadata is a contentious topic, with which the Blorb spec is not entirely unentangled. (The game identifier and frontispiece chunks are answers to small parts of the IF metadata problem.)

Rather than entangle ourselves further, we will merely say that metadata will be stored as XML, in a chunk of type ’IFmd’. The XML structure is documented in the Babel format agreement; see <http://babel.ifarchive.org/>.

```
4 bytes         ’IFmd’          chunk ID
4 bytes         n               chunk length
n bytes         ...             XML document (UTF-8 encoding)
```

The handling of metadata chunks will not be defined here. In particular, the behavior of an interpreter which finds more than one metadata chunk is undefined. It is likely to be a good idea to have at most one.

11. Chunks Specific to the Z-machine

The Z-machine’s graphics and sound capabilities were added late in Infocom’s history, but early in the history of data format standardization. As a result, the Z-machine’s audio and image models are both too rigid and too flexible to work well with modern file formats.

To compensate for this, we add additional information to the Blorb file. Interpreters can use these hints to display the resource information correctly.

Some of these hints are needed only to handle legacy Infocom games and their resources. Others will be useful for the creation of new Z-code games.

Each of these chunks is optional; no more than one of each should appear.

11.1. The Release Number Chunk

This chunk is used to tell the interpreter the release number of the resource file. It is meaningful only in Z-code resource files.

The interpreter passes this information to the game when the @picture_data opcode is executed with an argument of 0. The release number is a 16-bit value. The chunk format is:

```
4 bytes         ’RelN’          chunk ID
4 bytes         2               chunk length
```

```
2 bytes         num             release number
```

This chunk is optional. If it is not present, the interpreter should assume a release number of 0.

11.2. The Resolution Chunk

This chunk contains information used to scale images. The chunk type is 'Reso'. It is optional. This chunk is meaningful only in Z-code resource files.

A scalable image is one which the author says should be larger when more space is available, and smaller when less space is available. (Note that the Z-code game does *not* directly control the scaling of images. The interpreter controls the scaling of images, in response to the information in the resolution chunk. The interpreter then provides the scaled size in response to @picture_data queries, and the game draws its display based on those queries.)

It is also possible to create images that have a fixed scaling ratio; they are always scaled up or down by a particular amount, regardless of window size.

Not all images have to be scalable. Unless the resolution chunk gives scaling data for an image, that image is assumed to be non-scalable. Non-scalable images are always displayed at their actual size. (One image pixel per screen pixel.)

This chunk is optional; if it is not present, then all of the images in this file are non-scalable.

```
4 bytes         'Reso'          chunk ID
4 bytes         num*28+24       chunk length
4 bytes         px              standard window width
4 bytes         py              standard window height
4 bytes         minx            minimum window width
4 bytes         miny            minimum window height
4 bytes         maxx            maximum window width
4 bytes         maxy            maximum window height
num*28 bytes    ...             image resolution entries
```

The "standard window size" is the normal size, the author's original chosen size, for the Z-machine window. It is not the only possible size; a good V6 game should be prepared for any window the interpreter chooses to create. The idea is that when the Z-machine window is exactly the standard size, scalable images are presented at their original size. When the Z-machine window is larger than the standard size, scalable images are scaled up; when it is smaller, scalable images are scaled down.

The minimum and maximum window sizes are provided as a hint to the interpreter, when it is choosing a window size. It may also use the standard window size as a hint for this purpose. (If the interpreter lacks the ability to choose its own window size, of course, it will ignore these hints.) The idea is that the minimum and maximum sizes define the range in which the game can draw itself successfully.

Any or all of minx, miny, maxx, maxy can indicate "no limit in this direction" by containing a value of zero. However, px and py must contain non-zero values. Unless the min or max values are zero, it must be true that minx ≤ px ≤ maxx, miny ≤ py ≤ maxy.

Important note: The standard, minimum, and maximum window size values are measured in *screen pixels*. Furthermore, unscaled pictures should be drawn in screen pixels — one image pixel per screen pixel. (This may seem dumb as rocks, and maybe it is, but my rationale is presented at the end of this document.)

Also note that I have not mentioned Z-pixels. This standard does not concern itself with Z-pixels.

On with the show.

The standard, minimum, and maximum window sizes are followed by a set of image entries, one for each scalable image. (Non-scalable images do not have an entry in this table; that's how they are declared to be non-scalable.) Each image entry is 28 bytes, of the form:

```
4 bytes         number          image resource number
4 bytes         ratnum          numerator of standard ratio
4 bytes         ratden          denominator of standard ratio
4 bytes         minnum          numerator of minimum ratio
4 bytes         minden          denominator of minimum ratio
4 bytes         maxnum          numerator of maximum ratio
4 bytes         maxden          denominator of maximum ratio
```

The number is the picture number; in other words, this entry applies to the resource whose usage is 'Pict' and whose number matches this value.

The entry then contains a standard, minimum, and maximum image scaling ratio. Each ratio is a real number, represented by two integers:

- stdratio = ratnum / ratden
- minratio = minnum / minden,
- maxratio = maxnum / maxden.

Minratio can indicate zero ("no minimum limit") by having both minnum and minden equal to zero. Similarly, maxratio can indicate infinity ("no maximum limit") by having maxnum and maxden equal to zero. It is illegal to have only half of a fraction be zero.

To compute the actual scaling ratio for this image, the interpreter must first compute the overall game scaling ratio, or Elbow Room Factor (ERF). If the actual game window size is (wx,wy), and the standard window size is (px,py), then

- ERF = (wx/px) or (wy/py), whichever is smaller.

(Note that if the game's window is exactly its standard size, ERF = 1.0. If the window is twice the standard size, ERF = 2.0. If the window is three times the standard width and four times the standard height, then ERF = 3.0, because there's really only enough room for the game's standard layout to be tripled before it overflows horizontally.)

The scaling ratio R for this image is then determined:

- If ERF*stdratio < minratio, then R = minratio.
- If ERF*sdtratio > maxratio, then R = maxratio.
- If minratio ≤ ERF*stdratio ≤ maxratio, then R = ERF*stdratio.

If minratio and maxratio are the same value, then R will always be this value; ERF and stdratio are ignored in this case. (This indicates a scalable image with a fixed scaling ratio.)

The interpreter then knows that this image should be drawn at a scale of R screen pixels per image pixel, both vertically and horizontally. The interpreter should report this scaled size to the game if queried with @picture_data (as opposed to the original image size).

Yes, this is an ornate system. The author is free to ignore it by not including a resolution chunk. If the author wants scaled images, or variably-scalable images, this system should suffice.

Here are some examples. They're not necessarily examples of good art design, but they do demonstrate how a given set of desires translate into images and resolution values. All are for a game with a standard size of (600,400).

The game wishes a title image that covers the entire window, and all the resolution should be visible at the standard size. (So if the window is twice the standard size, the image will be stretched and coarse-looking; if the window is half the standard size, the image will be squashed and lose detail.)

- Image size (600,400); stdratio 1.0; minratio zero; maxratio infinity.

The game has a background image of a cave, made from a scanned photograph. At standard window size, this should cover the entire window, but not all the detail needs to be visible. If the window is larger, the image should still cover the entire window; more detail will be visible, up to twice the standard size (at which point all the resolution should be visible.) If the window is larger than twice the standard size, the image should not be stretched farther; instead the game will center it and have blank space around the edges.

- Image size (1200,800); stdratio 0.5; minratio zero; maxratio 1.0.

The game has small monochrome icons indicating different magical perceptions, which it will draw interspersed with the text. The icons should always be drawn at double size, two screen pixels per image pixel, regardless of the window size.

- Image size (20,20); stdratio 1.0; minratio 2.0; maxratio 2.0. (In this case, remember, the stdratio value is ignored.)

The game has a graphical compass rose which it will draw in the top left corner. This should be 1/4 of the window size in the standard case, and shrink proportionally if the window is smaller. However, if the window is larger than standard, the rose should not grow; all the extra space can be allotted for text. All detail (image pixels) should be visible in the standard case.

- Image size (150,100); stdratio 1.0; minratio zero; maxratio 1.0.

The same compass rose, still to be 1/4 of the window size — but this time it is critical that all the image detail be visible when the window is as small as half-standard (that is, when the rose is 75 by 50 pixels). At standard scale (150 by 100), it will therefore appear stretched and coarse. If the window is smaller than half the standard size, the rose should not shrink beyond 75x50, so that pixels are never lost.

- Image size (75,50); stdratio 2.0; minratio 1.0; maxratio 2.0.

End of verbose examples.

11.3. The Adaptive Palette Chunk

This chunk contains a list of pictures that change their colors according to the pictures plotted before. The chunk type is 'APal'. It is optional. This chunk is meaningful only in Z-code resource files.

This format exists to describe the legacy behavior of some V6 Infocom games (Zork Zero and Arthur). Its support in interpreters is optional, and its use is strongly discouraged for any purpose other than conversions of Infocom graphics.

```
4 bytes         'APal'          chunk ID
4 bytes         num*4           chunk length
num*4 bytes     ...             adaptive palette entries
```

Each entry is 4 bytes, of the form:

```
4 bytes         number          picture resource number
```

If this chunk is present:

- All pictures in the Blorb file will be PNGs or Rects.
- All PNGs will be indexed-color (color type 3).
- All PNGs will use only color indices 2 through 15.
- All PNGs will have no more than 16 entries in their PLTE chunk.
- PNGs may have a tRNS chunk marking color 0 only as fully transparent, in which case color index 0 may also be used. No other forms of the tRNS chunk are valid.

However, the following rules still apply from the PNG standard:

- Any bit depth of PNG is valid (1, 2, 4, or 8 bits per pixel).
- The PLTE chunk is required by the PNG standard, and it must have sufficient entries to cover every color used in the PNG, even in adaptive-palette pictures.
- The PLTE chunk may not have more entries than can be represented by the PNG's bit depth.
- The PNGs may have gAMA, cHRM and sRGB or iCCP chunks describing the color space. Interpreters should make every effort to support at least gAMA. For the Infocom graphics at least, cHRM, sRGB and iCCP are probably beyond the call of duty.

These restrictions, though intricate, serve to make the interpreter's life easier at the expense of constraining the creator. The constraints are natural given the form of the original Infocom graphics.

The interpreter should keep track of the "Current Palette". This will be a 14-entry table, covering color indices 2-16. For ease of implementation, this will probably be a 16-entry table, whose first two entries are not significant.

Whenever a picture *not* listed in the APal chunk is plotted, its palette (as derived from its PLTE, gAMA, cHRM and sRGB/iCCP chunks) should be copied into the Current Palette. If its palette has fewer than 16 entries, then only those entries of the Current Palette are changed. (Possible interpreter implementation: transform the PNG's PLTE chunk according to the gAMA, cHRM, sRGB chunks, then copy it into your Current Palette which is always in the screen color-space. With libpng, use png_get_PLTE, after calling png_update_info).

Whenever a picture listed in the APal chunk is plotted, its palette should be ignored, and it should be plotted with the Current Palette. (Possible interpreter implementation: strip out the PLTE, gAMA, cHRM and sRGB/iCCP chunks from the PNG, and insert the Current Palette as its PLTE. Or with libpng, use png_set_PLTE before reading the data).

The behavior is undefined if any adaptive-palette pictures are plotted before a non-adaptive picture has been plotted.

If picture caching (through @picture_data or otherwise) is implemented, special attention may need to be paid to ensure that adaptive images that are cached are still appropriate for the Current Palette when plotted. It would appear that the Zork Zero does reset the cache after a palette change, but this has not been exhaustively investigated.

Alternatively, for the full retro-gaming experience, the pictures can be handled in the same way as the Amiga and IBM MCGA interpreters, as follows: Use a 16-color screen mode. Copy non-adaptive pictures' palette (apart from the first two entries) into the screen palette when plotted. Use color indices 0 and 1 for the window background and text respectively. This mimics the IBM MGA and Amiga display, where drawing a picture can change the colors of graphics already on the screen, but it is not the preferred rendering.

Shogun and Journey do not use any adaptive-palette images, but on some platforms the effect of pictures already on the screen changing color is visible. To give an interpreter the ability to do this if desired, and to signal that optimizations may be possible because of the limited nature of the graphics, the Blorb files for Shogun and Journey contain an empty APal chunk.

[Thanks to Kevin Bracey for this extension.]

11.4. The Looping Chunk

This chunk contains information about which sounds are looped, in a V3 Z-machine game. The chunk type is 'Loop'. It is optional.

Note that in V5 and later, the @sound_effect opcode determines whether a sound loops. The looping chunk is ignored. Therefore, this chunk should not be used at all in Blorb files intended for games which are not V3 Z-machine games.

The format is:

```
4 bytes        'Loop'         chunk ID
4 bytes        num*8          chunk length
num*8 bytes    ...            sound looping entries
```

Each entry is 8 bytes, of the form:

```
4 bytes        number         sound resource number
4 bytes        value          repeats
```

The repeats flag is one if the sound is to be played once; it is zero if the sound is to repeat indefinitely (until it is stopped or another sound started.) If there is no entry for a particular sound resource, or if the looping chunk is absent, the V3 interpreter should assume the flag is one, and play the sound exactly once.

12. Other Optional Chunks

A resource file can contain extra user-level information in 'AUTH', '(c) ', and 'ANNO' chunks. These are all optional. An interpreter should not do anything with these other than ignore them or (optionally) display them.

These chunks all contain simple ASCII text (all characters in the range 0x20 to 0x7E). The only indication of the length of this text is the chunk length (there is no zero byte termination as in C, for example).

The 'AUTH' chunk, if present, contains the name of the author or creator of the file. This could be a login name on multi-user systems, for example. There should only be one such chunk per file.

The '(c) ' chunk contains the copyright message (date and holder, without the actual copyright symbol). There should only be one such chunk per file.

The 'ANNO' chunk contains any textual annotation that the user or writing program sees fit to include.

13. Deprecated Chunks

Some older Z-code Blorb files contain an 'SNam' (story name) chunk, which contains the game's title. The format of this chunk is Unicode UTF-16, with the 16-bit values stored big-endian. Modern Blorb files should not have an 'SNam' chunk; this information should be stored in the metadata chunk instead.

14. Presentation and Compatibility

14.1. File Suffixes

Previous versions of the Blorb spec did not discuss file naming. However, with the relapse of MacOS into filename suffix semantics, it is impossible for us to pretend that the issue is an implementation detail.

It is always legal for a Blorb file to have a ".blorb" filename suffix. However, interpreters have a natural interest in locating *their* sort of Blorb files -- Z-code, Glulx, or so on -- and it is generally easier for them to do this by filename suffix, rather than by opening each Blorb and looking at its resource index. Therefore, ".zblorb" and ".gblorb" should be used to designate Blorb files containing Z-code and Glulx games, respectively.

On platforms which limit filename suffixes to three characters, the suffixes ".blb", ".zlb", and ".glb" may be used instead. But this practice, at least, I can deprecate without qualm. I hope.

14.2. MIME Types

Historically, Blorb files have been associated with the MIME type `application/x-blorb`.

We can use the profile feature of MIME to differentiate the contents: `application/x-blorb;profile="zcode"` and `application/x-blorb;profile="glulx"` for the common virtual machines.

(Previous versions of this spec suggested `application/x-blorb-zmachine` and `application/x-blorb-glulx`. These are now deprecated.)

14.3. Z-Machine Compatibility Issues

The image system presented in this document is fully backwards-compatible with Infocom's interpreters. Infocom V6 games, such as Arthur, Journey, and Zork Zero, contain only non-scalable image resources. The game files are written to deal with both variations in window size and variations in image size (since the interpreters for different platforms had different window sizes and different art.) Therefore, if you construct a Blorb file containing the images from a particular platform (say, the Mac) and give it the suggested window size of the Infocom Mac interpreter, the game file will deal with it correctly.

The image system is also sort of forwards-compatible, in the following sense. If you take a Blorb file whose standard (intended) window size is the same as the Infocom interpreter's, and break it out into Infocom image files, the Infocom interpreter should display it correctly. The interpreter will not scale images, but since the window size is equal to the standard size, the Blorb rules require the images to be displayed unscaled anyway.

Also, of course, if you take a Blorb file which contains only non-scalable images, an Infocom interpreter will act correctly, since it will not scale the images regardless of the standard size.

The sound system is slightly more problematic. A game file can announce that it uses sound, by setting a header bit; the interpreter can announce that it does not support sound, by clearing that bit. But there is no way to distinguish a game that uses sampled sound only, from one that uses sampled sound and music. (And similarly for the interpreter's support of samples versus music.) This may be addressed in a future revision of the Z-machine. In the meantime, games should set that header bit if any kind of sound is used (samples or music or both.) And interpreters should clear that bit only if *no* sound support is available. If the interpreter supports sampled sound but not music, it should leave the header bit set, announcing that it does "support sound." It should then ignore any request to play a music resource.

There is also the question of overlapping sounds. The Z-Spec (9.4.2) says that starting a new sound effect automatically stops any current one. But it is not desirable that a sound effect such as footsteps should interrupt the playing of background music. Therefore, the interpreter should amend this rule, and consider sampled sounds and music to be in seperate "channels". Samples interrupt samples, and music interrupts music, but one form of sound does not interrupt the other.

This is an actual variance in the behavior of the Z-machine, and worse, a variance which depends on data format. (One sound will either stop another, or not, depending on whether the sound is stored in AIFF (sampled) or Ogg/MOD (music) format.) We apologize for the ugliness.

Again, future versions of the Z-machine may address this issue, and allow a more general system where any sound can be overlaid on any other sound, or interrupt it, as the game desires and regardless of storage format. (After all, there can be background *sounds* as well as background *music*.) Such a system would also allow the interpreter to announce its limitations and capabilities — whether it can play music, whether it can play two pieces of music at once, how many sampled sounds it can play at once, etc.

A final, ah, note: The remark at the end of Z-Spec chapter 9, about sequencing sound effects to simulate the slow Amiga version of "The Lurking Horror", should not be applied to music sounds. New music should interrupt old music immediately, regardless of whether keyboard input has occurred since the old music started.

14.4. Glk Compatibility Issues

The Glk I/O library was designed with portable resources in mind, so there should be no incompatibility.

Remember that the resolution and scaling data is not used by Glk. That chunk is ignored by Blorb-capable Glk libraries.

14.5. ADRIFT 5 Compatibility Issues

ADRIFT supports more media formats than Blorb, but has adopted Blorb as a packaging format. To permit this, the following chunk types may be used in ADRIFT blorbs:

For images: ’GIF ’.

For sounds: ’WAV ’, ’MIDI’, ’MP3 ’.

ADRIFT Blorb files should use MIME type `application/x-blorb;profile="adrift"`. The filename suffix should be ".blorb" or ".adriftblorb". *["A" or "AD" is unfortunately not a unique prefix when it comes to IF systems!]*

15. The IFF Format

A description of the IFF format can be found at

```
<http://eblong.com/zarf/blorb/iff.html>
```

In the interests of simplicity, this proposal does not use IFF LISTs or CATs, even though its purpose is to contain concatenated lists of data. Therefore, the format can be quickly described as follows:

```
4 bytes        'FORM'         Magic number indicating IFF
4 bytes        n              FORM length (file length - 8)
4 bytes        'IFRS'         FORM type
n-4 bytes      ...            The chunks, concatenated
```

Each chunk has the following format:

```
4 bytes        id             Chunk type
4 bytes        m              Chunk data length
m bytes        ...            Chunk data
```

Remember that the chunk length *never* includes the eight-byte header (the type and length). This is true for the top-level FORM and internal FORMs as well.

If a chunk has an odd length, it *must* be followed by a single padding byte whose value is zero. (This padding byte is not included in the chunk length m.) This allows all chunks to be aligned on even byte boundaries.

All numbers are two-byte or four-byte unsigned integers, stored big-endian (most significant byte first.) Character constants such as ’FORM’ are stored as four ASCII bytes, in order from left to right.

When reading an IFF file, a program should always ignore any chunk it doesn’t understand.

16. Other Resource Arrangements

It may be convenient for an interpreter to be able to access resources in formats other than a resource file. In particular, when developing a game, an author will want to load images and sounds from individual files, rather than having to re-package all the resources whenever any one of them changes.

Such resource arrangements are platform-specific, and the details are left to the interpreter. However, one suggestion is to have a single directory which contains all the resources as files, with one file per resource. (PNG files for images, and so on. The contents of each file would be exactly the same as the contents of the equivalent chunk, minus the initial eight bytes of type/length information.) Files would be named something like "PIC1", "PIC2"..., "SND1", "SND2"..., "DATA1", "DATA2"..., and so on. An executable game file (if present) would be named "STORY". Other chunks would be named as follows:

- "IDENT": game identifier chunk
- "PALETTE": color palette
- "FRONTIS": frontispiece identifier
- "RESDESC": resource textual descriptions
- "METADATA": metadata document
- "RELEASE": release number
- "RESOL": resolution chunk
- "ADAPTPAL": adaptive palette list
- "LOOPING": looping chunk

(Naturally, file suffixes would be added in platforms that require them.) The interpreter would be started up and handed the entire directory as an argument; or possibly the directory along with a separate Z-code file.

It is of course possible to break a Blorb file down into a directory in this format. When doing this, one must remember that AIFF (and no other chunk type) uses an IFF form as its single-file representation. Therefore, the SND... file representing an AIFF will begin "FORM<length>AIFF", followed by the chunk data. All other chunk types would be turned into files simply by extracting the chunk data.

17. Rationales and Rationalizations

- Why have a common resource collection format?

Infocom chose not to standardize their resource formats; they had a different picture format for each platform. This was a reasonable choice for them, since they were writing all the games, all the art, and all the interpreters. They therefore had the capacity to translate the art into platform-specific formats for all the platforms they supported.

In the modern age, an IF author does not have access to all the platforms his game will be played on. It is therefore reasonable to distribute art in a single format, and leave interpreter writers the job of supporting that format.

- Why an IFF-based format?

IFF does what we want; it’s a known, very simple way to concatenate chunks of data together.

Also, the common save-file format is IFF-based. This allows interpreters to use the same code for reading both save files and resource files.

- Why not compress data as well as archiving it? Why just concatenate everything together as chunks?

Any reasonable sound or image format already incorporates compression.

- Why is there a "number" field in the entries in the resource index chunk? Why not just assume chunks are numbered consecutively?

On the Z-machine, pictures are not necessarily numbered contiguously (Z-Spec 8.8.6.1.) Sounds are numbered consecutively, but sounds 1 and 2 are bleeps, so the game-specific sound resources start at 3. (Z-Spec 9.2.) In Glk, resources need not be contiguous at all. Rather than jigger the numbering or require place-holder chunks, I decided there should be an index which maps resource numbers to chunks.

- Why only two image formats? Why not allow any image format?

The whole point of this exercise is to assure the author that the player can view his art. If we allow lots of different formats, we can’t possibly insist that every interpreter must display all the formats. This leaves us just about back where we started. Individual game authors would be negotiating with individual interpreter authors to support particular formats, and it would just be icky.

Therefore, we *do* insist that every interpreter be able to display all the formats listed in this standard. That means a small number of formats. See the next two questions.

It is very strongly suggested that an interpreter use standard open-source libraries for interpreting sound and image resources. To rely on OS services, while tempting, is a road paved with incompatibility problems.

- Okay, then, why three sound formats?

Because a sampled-sound format (like AIFF) can reproduce anything, and a compressed digital format (like Ogg) can reproduce large sounds efficiently.

MOD can reproduce music even more efficiently, but it’s really retained in the spec more for backwards compatibility than for any technical reason. Existing games use it, and while it’s not very well standardized, it seems to work.

(The "song" format was never widely used, which is both the qualification and the justification for deprecating it.)

- Why PNG and JPEG for images?

The PNG format is not burdened with patent restrictions; it is free; it’s not lossy; and it can efficiently store many types of images, from 1-bit (monochrome) images up to 48-bit color images. JPEG is lossy and not optimal for images other than photographs, but compresses photographs well. Earlier versions of Blorb specified only PNG, but JPEG was a popular request, and the two formats should complement each other.

As to other possibilities: GIF is a popular format, but was previously owned by twits who restricted its use. (Life has improved, but we have PNG now and we will stick with it.) TIFF has been suggested, but it seems to be overly baroque. Blorb is likely to stay with PNG and JPEG for the foreseeable future.

- So why does ADRIFT get a bye on these format decisions?

Game authors and interpreters need to agree on what formats they will use. Z-code had no cross-platform agreement when Blorb was invented, and Glulx was created to use Blorb, so Blorb's role for them is normative. GIF and MP3 are not going to become standard Blorb format types.

ADRIFT, in contrast, already had cross-platform interpreters when it adopted Blorb. Blorb can therefore be valuable to ADRIFT as a packaging and metadata format, while taking a descriptive role on media formats. (The alternative would be to disallow ADRIFT Blorb files, which seems silly.)

[It is worth noting, however, that IF interpreters are often ported by adapting existing IF display code. IF interpreter ports can also be based on Glk libraries and the Glk API. Both routes entail the Blorb standard media format list, to some extent. Therefore, game authors have some reason to consider sticking to those formats.]

- What is the Blorb Policy on Color Depth?

The idea is that each author can decide what kind of color requirements his game will have.

The alternative (which we did *not* choose) would be to mandate a fixed set of color requirements for all graphical games — for example, an 8-bit color display set to a color-cube set of colors. This seems like a dumb idea. Any fixed set of requirements is going to be impossible for some machines and standard equipment on others, and both these sets will change over time. The requirements would quickly become obsolete.

Instead, we choose to allow any kind of art in graphical games. If the author includes only monochrome images, the game will run anywhere. If the author includes full-color 32-bit images, he is creating a game which wants a powerful graphics machine to display itself on. That's the author's choice. If the player's machine only allows 8-bit color, his interpreter will have to dither or otherwise reduce the color information of the game art. The player can accept this, buy a more powerful computer, or throw away the game. There's no way around that. The problem can only be avoided by outlawing 32-bit color images, which we do not wish to do.

- What's the idea of the palette chunk?

The palette chunk itself is provided for the benefit of interpreters which can control their display palettes or color depth. The palette declares the minimum set of colors (or direct-color depth) which the author wants you to have in order for the game to "look okay." It may be a good idea to switch palettes in order to play a particular game; the palette chunk tells the interpreter this advice.

Now, the interpreter is not *required* to follow this advice. This is for the player's benefit; if the player has a monochrome machine, or just doesn't like changing palettes, he is not denied the opportunity to play the game. He'll just get reduced-quality art. That's his choice. As stated above, he can accept it, upgrade, or throw the game away.

- What is the Blorb Policy on Pixel Size?

We make a couple of assumptions.

Assumption one: Image pixels are square. Your images should have the correct aspect ratio when drawn with square pixels — that is, when the number of image-pixels-per-inch is the same vertically and horizontally. If your art program doesn't understand square pixels, get a real art program. There. That's resolved.

(This means that if an image is 400 pixels wide and 200 pixels high, the interpreter should draw it on the screen with a physical width twice its physical height. Anything else will look distorted.)

(It has been noted that this does not exactly apply to Infocom's V6 games; their art was probably designed for an era of computers that did not have exactly square pixels (IBM EGA, Apple II machines displaying on television screens, and other such barbarisms.) However, this does not seem to have concerned them. Infocom interpreters which are running on modern machines, with square-pixel displays, display their art with square pixels. We will do the same.)

Assumption two: It is always okay to draw images at their "actual size" — one image pixel per screen pixel.

Now you think I'm crazy. It is true that many modern screens can be adjusted to different pixel sizes. However, *I declare this to be an illusion.* If a user sets his monitor to smaller pixels, it's because he wants a given image to be smaller. So he can fit more of them on screen. He also wants his text to be smaller, and his windows. That's the way web browsers work, that's the way Adobe Photoshop works, and that's damn well good enough for the Z-machine.

Perhaps in the future there will be monitors that break this rule — much smaller pixels, 300 or 600 pixels per inch, for example. At that time there will be some consensus on how to display images. (Frankly, I expect it will be "draw them at 55, 72, or 88 pixels per inch, depending on the user's previous preference." Or some such set of standard options.) Z-machine interpreters can follow that plan when it emerges.

Until then, the right size for a non-scaled picture is one image pixel per screen pixel. If an image is to be scaled by a ratio of 2.0, then the right size for it is one image pixel per two screen pixels (vertically and horizontally). And so on.

- Where do Z-pixels come into all this?

The definition of Z-pixels is entirely up to the interpreter. This standard says nothing on the subject, and does not care.

It is true that the interpreter must tell the game what the window size and image sizes are, as measured in Z-pixels. That's the interpreter's job. The interpreter knows how big its window is, in screen pixels; it translates that into Z-pixels — using whatever definition it has — and reports it to the game. Then, the scaling rules of this spec define what the display sizes of the images will be, as measured in screen pixels. The interpreter translates these sizes into Z-pixels — using the same definition — and reports them to the game. All consistent and well-defined.

- What is the Blorb Policy on Interpreters that do Funky Stuff?

The interpreter is Allowed. It's okay to be ugly.

For example, someone may (in a fit of insanity) write a Blorb-compliant interpreter for the Apple II. The Apple II had non-square pixels. But (assume) it doesn’t have the processing power to scale all its images by a factor of 1.2 (or whatever) to adjust for this. Well, it’s legal to write an interpreter that draws art at one image pixel per (non-square) screen pixel. The art will look distorted; the user can like it or play on a different machine.

For another example, someone may want their entire game display doubled in size. All the art twice as large (in screen pixels) as this spec says it should be. An interpreter which has this option is legal. It’s the moral equivalent of mounting a magnifying glass on your monitor — that certainly doesn’t violate any software standards.

- What about playing Blorb-packaged games on original Infocom interpreters?

It’s possible. You’ll have to unpack the PNG art and translate it into the format that Infocom used. Since the Infocom interpreters had a hard-wired screen size, you can precompute all the scaling factors, and do any necessary scaling in the translation process. 32-bit color images will have to be color-reduced; that’s the way it goes. But the result should be fully playable on Infocom’s interpreters.

- Have you considered —

Yes.