Proposal for storage of ID manifests within OpenEXR files

Peter Hillman
Weta Digital Ltd

August 10, 2017

Abstract

This document proposes a scheme for storage of human readable names for IDs within an OpenEXR header as an *ID Manifest*, so that discrete values stored in ID channels may be interpreted as objects. Current methods for storing this information rely on external files or re-purpose existing attributes in an ad-hoc manner. This proposal formalises ID manifests with their own native OpenEXR type. This is a proposal only and not part of the OpenEXR standard.

Contents

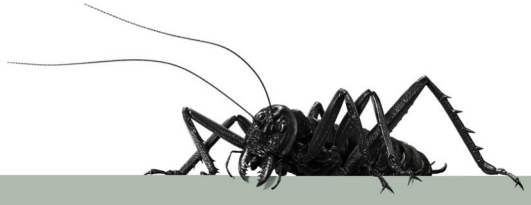
1 IDs within OpenEXR files **1**
1.1 Multi-variate IDs . . . 2
1.2 Encoding ID values in EXR images . . . 2
1.3 Converting text to ID values . . . 2
1.4 ID lifetime . . . 3
1.5 IDs without text . . . 3
1.6 Manifest completeness . . . 3

2 Manifest storage scheme **3**
2.1 ID encoding schemes . . . 4
2.1.1 `id` scheme . . . 4
2.1.2 `id2` scheme . . . 4
2.2 Reserved characters . . . 5
2.3 Hierarchies . . . 5
2.4 Encoding on disk . . . 5
2.5 Multipart files . . . 5

1 IDs within OpenEXR files

Many renderers support assigning an ID to an individual piece of geometry, and will output that ID into the image they produce. There are various applications for such an ID scheme:

- Creation of a mask that covers a particular object, so that it may be selectively modified, for example by changing its colour.
- Removing a particular object from a scene, because it is not required or so it can be individually modified or re-rendered and then recombined with the remaining objects.
- Identifying an object in a rendered image for debugging problems in the render process.
- Checking whether objects appear in particular frames of a render to help optimise rendering by removing unnecessary geometry, or else to minimise effort in re-rendering should the scene change.



Such identification can be carried out in a **frame consistent manner**: the information used to identify an object in one frame can be used to identify the same object every other frame of a render.

1.1 Multi-variate IDs

IDs are not limited to represent the *name* of an object. Other useful information can be encoded into ID channels. Examples of what may be storable in IDs are

- The model name, potentially within a hierarchy, e.g. `Gollum/LeftArm/Hand/Thumb/Nail`
- Material or shader properties, e.g. `Skin`
- A Particle ID
- Geometry Instance ID
- Version numbers

Where multiple pieces of information are to be encoded in the same image, multiple ID channels can be used, one for each piece of information. Alternatively, the information can be concatenated together and stored in a single channel, as a **Multivariate ID channel**. An example multivariate channel might encode "object `Elf/Armour` with material `Leather`" as ID number 100, and "object `Elf/Armour` with material `Steel`" as ID number 101. This means that both IDs need to be selected to extract the entire `Elf/Armour` object. Similarly, multiple IDs will need to be selected to identify everything in an image with material `Steel`. Using multivariate IDs increases the possibility of **hash collisions**, where completely different objects are accidentally assigned the same ID, but the approach can result in significantly smaller storage, and may result in more efficient rendering.

This scheme proposes a method for encoding what information an ID channel stores, as well as a mapping between individual ID values within the image and the information they represent.

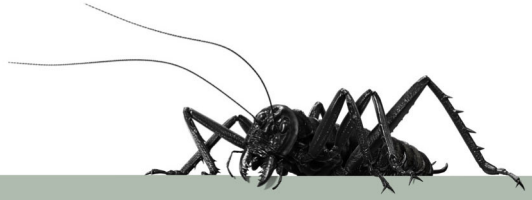
1.2 Encoding ID values in EXR images

It is possible that a pixel be covered by multiple objects. In this case, a single ID per pixel does not carry enough information to identify the pixel perfectly. In particular, it will cause sharp or noisy edges around masks. For applications where cleaner masks are required, OpenEXR deep images can be used to store multiple samples per pixel, with a different ID on each sample. Alternatively, an encoding scheme can be used to store that information in a regular EXR image. Such schemes are outside the scope of this document. The scheme here assumes there is an **unsigned integer** ID, either directly stored in the EXR as HALF or UINT channel(s), or else there is a scheme to derive an unsigned integer ID from the data stored within the EXR. Where HALF values are used to encode IDs, only values in the range 0-2048 inclusive are permitted: this is the range of continuous integers that can be represented by a 16 bit half float value; 2049 is the smallest positive integer which cannot be stored exactly in a half float[1]. The manifest itself supports unsigned integers of up to 64 bits.

1.3 Converting text to ID values

There is no guaranteed relationship between text names and ID values. Typically, the string is hashed to generate IDs. In this case, the manifest will encode which scheme was used. An application may use this to modify the IDs of objects consistently. However, hashing may happen internally within a renderer using an uncontrollable process.

[1] If deemed useful, this proposal could be extended with a more complex scheme to store a wider range of IDs in a HALF



It is not necessary that the relationship between text and IDs be a hashing scheme. If HALF values are used, only 2049 values are available. This is insufficient for a hashing scheme, since the probability of collision is high, so instead some other scheme must be used to generate the ID numbers for each object. It may also be, for example, a unique ID assigned by an asset management database.

1.4 ID lifetime

The **lifetime** of an ID mapping indicates whether the ID value can change. For example `Gollum` may be encoded with different ID values in each frame. If the ID lifetime is single frame, then the name of the required object must be used to look up the ID value used to identify it in each frame.

1.5 IDs without text

It is possible to use an ID channel which has no mapping back to a real name. For example, particle IDs are likely to be simple single numbers. In this case, the object might be identified by clicking on it, rather than browsing through a list of names. This scheme supports 'nameless' IDs: if the Manifest object is not present, the IDs are considered to be nameless. Otherwise, a manifest can exist with an empty mapping table entry for that particular ID channel.

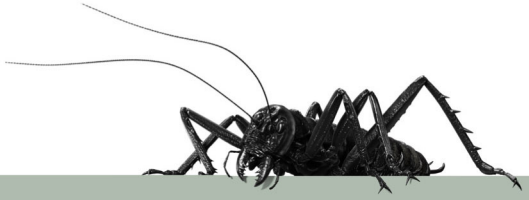
1.6 Manifest completeness

Where a sequence of files is rendered, each will contain its own manifest. There is no guarantee that every frame will encode the entire manifest of all objects appearing within the sequence. For example, if an object enters the shot half way through, the corresponding entry in the manifest for that object may not appear in the first frames of the sequence. Conversely, it should not be assumed that every ID listed in the manifest of a given frame will be used within that image, or in any other image of the sequence.

2 Manifest storage scheme

We propose a single attribute stored in the OpenEXR header which represents the mapping table, which identifies the table used by each **group of channels**, that is, it is a **map** of ID **tables**. This will be stored in the header with the name `idmanifest`. Each table may relate to more than one ID channel. In this case, the channel group will contain more than one channel name. For example, the list may contain `id` and `reflection.id`. The table for each channel group contains a small header indicating what information is encoded in the corresponding channels, and a description of the hashing scheme. Figure 2 shows a possible C++ implementation of this scheme. The ID table is a map of ID values to names. This means that multiple IDs can have exactly the same text. Using the ID number as the map key and the text as the value allows much faster searching and insertion into the table than the opposite scheme; a `std::multimap` object can be generated if reverse lookups are required.

Figure 1 shows a possible value of the ID manifest. In this case the image contains two ID channel groups, each with one channel. The first group encodes both model and shader name, with each ID expanding to a pair of strings. This mapping is stable: since the ID is hashed (using a custom scheme), every time the same text appears it will have the same ID. The second group, containing the channel particleId, has IDs with no mapping to a string. The IDs are stable throughout a render, but not constantly stable. (If the particles are re-simulated with a different random seed then all the particle IDs will change)



[id]	*lifetime:*	STABLE	
	hashScheme:	custom	
	encodingScheme:	id	
	components:	**model**	**shader**
	1234	Snowflake	snow
	1539	Gollum/LeftArm/Hand/Thumb	skin
	2457	Gollum/Head/LeftEye/Pupil	pupil
	3242	Gollum/Head/RightEye/Pupil	pupil
	3245	Gollum/LeftArm/Hand/Thumb/Nail	fingernail
[particleId]	*lifetime:*	SHOT	
	encodingScheme:	id	
	hashScheme:	none	
	components:	**particleId**	

Figure 1: Example manifest, storing information for two ID channels

2.1 ID encoding schemes

The **encodingScheme** string encodes how the unsigned integers in the manifest are encoded in the image itself. The value is a string, rather than an enumeration. This allows proprietary schemes to be encoded using a reverse-URL format such as `nz.co.wetafx.customid`. Two simple schemes are predefined, but other schemes may be used. Note that some schemes might involve usage of extra channels not listed within the manifest, or additional attributes.

2.1.1 `id` scheme

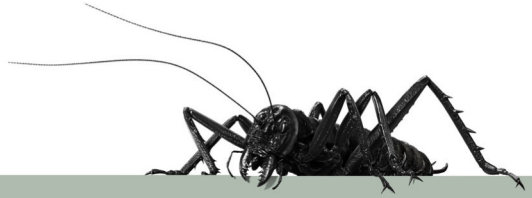
In this scheme, each ID is directly encoded into a pixel, or deep sample, using a single UINT or HALF channel. Note that the channel name `id` is intended to be interpreted as an ID using this scheme[2], particularly in deep images, even in the absence of the `idmanifest` attribute. As such, where only one ID is being stored using the `id` scheme, the channel name should also be `id`.

2.1.2 `id2` scheme

In this scheme, each ID is encoded as a 64 bit number in a pair of UINT channels. The channel group within the manifest lists both channels in the pair. The first channel of each pair contains the least significant 32 bits of the ID, the second contains the most significant 32 bits (OpenEXR uses a little-endian scheme). Thus, if the channel group lists the channels as `[id.lsb,id.msb]` the 64 bit ID `0x0123456789ABCDEF` would be encoded by storing `0x89ABCDEF` in `id.lsb` and `0x01234567` in `id.msb`.

64 bit IDs are supported due to the danger of hash collisions when storing only 32 bit IDs. With 64 bit IDs it is much less likely that two strings are hashed to the same ID. Even so, it is usually possible to isolate an individual object using only one of the two 32 bit channels. In the above example, suppose the object `0x0123456789ABCDEF` is to be isolated. The `id.lsb` channel alone can isolate the object if no other entry in the manifest has `0x89ABCDEF` as the least significant 32 bits. Even if there is a collision in `id.lsb`, the `id.msb` channel may be unique.

[2] for more information on channel naming conventions see `http://www.openexr.com/InterpretingDeepPixels.pdf`



2.2 Reserved characters

It may be useful to join the components of an ID into a single string. In this case, the semicolon character ';' (ASCII `0x3B`) is recommended. Use of this character component name and value strings is discouraged.

2.3 Hierarchies

Many ID types are hierarchical in nature, for example the object hierarchy `Gollum/LeftArm/Hand/Thumb`, or a material hierarchy `Plants/Trees/Leaves/PineNeedle`. The forward slash '/' character (ASCII `0x2F`) shall be used as a separator to indicate levels of a hierarchy. Note this scheme does not explicitly exploit hierarchical representation for more efficient storage, though helper classes may be provided to turn a manifest into a tree structure that lists all of the unique ID values at each level of a hierarchy.

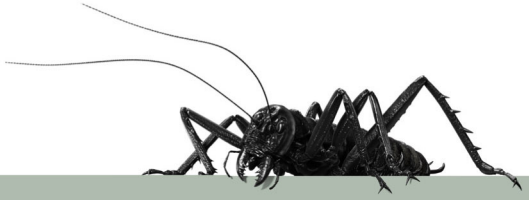
2.4 Encoding on disk

The OpenEXR library will present an API for reading and writing values within the table, searching the hierarchy (both mapping ID number to names and names to numbers), and reference hashing schemes. It will also provide a `ManifestAttribute` which reads/writes the file to disk. ZIP compression will be used to encode the table into the file to reduce storage space. The attribute must be explictly decompressed to read it. Delaying the decompression step until the attribute is interpreted reduces storage and processing overhead in applications which do not need to interpret the manifest.

Functionality to write that file to an arbitrary stream will also be provided, to allow storage of manifests independently to OpenEXR images.

2.5 Multipart files

Where EXR-2.0 multipart files are used, the `idmanifest` attribute should either appear in the part for which it is relevant, or else in part 0. If a part does not contain an `idmanifest` attribute, the table for part 0 should be used instead. Note it is possible for a file to contain different "contradictory" manifests for different parts. For example the left eye render may contain a different set of IDs to the right eye. It is also possible that id channels are spread across multiple parts. In the case of the example in Figure 1, `id` could be stored in part 0, and `particleId` in part 1



```
// structure that identifies the purpose of all ID channels within an OpenEXR part,
// and (optionally) stores a name for every ID value within a file
namespace IdManifest
{

  // indication of how long a mapping between an ID name and an ID value holds for

  enum IdLifetime
  {
     FRAME, // The mapping may change every frame:
     SHOT,  // The mapping is consistent for every frame of a shot
     STABLE // The mapping is consistent for all time.
  };


  //
  // hashing scheme is stored as a string rather than an enum, to allow
  // proprietary schemes to be encoded with less danger of collision
  // proprietary schemes should be encoded in a reverse-URL syntax
  //
  static const std::string NOTHASHED = "none";          // no relationship between text and ID
  static const std::string CUSTOM = "custom";           // text is hashed using defined scheme
  static const std::string MURMURHASH3 = "MurmurHash3"; // MurmurHash3 is used


  struct TableHeader
  {
      std::vector <std::string> _components; //list of components represented by this channel group
      enum IdLifetime _lifeTime;
      std::string _hashScheme; //one of above strings or custom value e.g "nz.co.wetafx.cleverhash2"
      std::string _encodingScheme; //string identifying scheme to encode ID numbers within the image
  };

  typedef std::vector<std::string> TableEntry;
  typedef std::map<uint64_t,TableEntry> IDTable;


  //
  // Description of the information represented by a single channel
  //
  struct ChannelManifest
  {
     TableHeader _header;
     IDTable _table;
  };

}

typedef std::map< std::set<std::string> , IdManifest::ChannelManifest > Manifest;
```

Figure 2: C++ representation of storage scheme