Cryptomatte Specification version 1.2.0

Table of Contents:

EXR File:

Overview

The format consists of RGBA images in a multichannel EXR, which contains a number of RGBA layers and metadata. The channels are organized using a naming convention. EXR header metadata is used to store information about the hash, conversion, and can optionally include a manifest.

Layer naming

An example:

```
CryptoObject
CryptoObject00
CryptoObject01
...
```

Or more formally, there is no requirement to use the word “Crypto” in the name.

```
{typename}
{typename}00
{typename}01
{typename}02
...
```

Channel contents

An arbitrary number of channels, numbered in this format are allowed. The number of channels used should be the same for all frames of the image sequence rendered. Otherwise it may be set automatically, manually, or to a given value based on any criteria the implementer desires. The suggested default value is 6 levels, meaning 3 numbered EXR layers (00, 01, and 02).

The contents of these layers are:

```
{typename}.r   - Unused, or optionally, the ID value filtered normally.
{typename}.g   - Normally filtered color generated from IDs
{typename}.b   - NOrmally filtered color generated from IDs

{typename}00.r - ID rank 0
{typename}00.g - Coverage for ID rank 0
{typename}00.b - ID rank 1
{typename}00.a - Coverage for ID rank 1

{typename}01.r - ID rank 2
{typename}01.g - Coverage for ID rank 2
{typename}01.b - ID rank 3
{typename}01.a - Coverage for ID rank 3

{typename}02.r - ID rank 4
{typename}02.g - Coverage for ID rank 4
{typename}02.b - ID rank 5
{typename}02.a - Coverage for ID rank 5
```

Metadata

The file is required to provide metadata about the cryptomatte layer(s) it contains. It is, however, not required to contain a manifest.

```
exr/cryptomatte/adb0a84/name = CryptoAsset
```

Each "type" of Cryptomatte contained is assigned a (maximum) 7-character key, which will be used to index the metadata for each of them. In this example, the key is "`adb0a84`". This key should be chosen to stay the same from frame to frame and shot to shot. A suggested method for generating this key is to use a hash of the "type name" and use the first seven characters.

The required metadata are:

- **name**: the name of the Cryptomatte type, for example, CryptoAsset.
- **hash**: the type of hash used. "MurmurHash3_32" is the only one currently supported.
- **conversion**: The conversion method used to convert the hash value into pixel color data. Only one conversion method, "uint32_to_float32", is supported.

Optional metadata are:

- **manifest:** The manifest encoded as a JSON manifest. See: manifest
- **manif_file**: A relative path to a "sidecar" JSON file which contains the manifest. This path is always relative to the directory the EXR file is found in and should not start with "../" or "./". An example is below. The file should have the file extension, ".json" and contain the manifest in the same format as if it were contained the header.

For example, on a file containing three sets of Cryptomattes, the last of which has its manifest in a separate file:

```
exr/cryptomatte/adb0a84/name = CryptoAsset
exr/cryptomatte/adb0a84/hash = MurmurHash3_32
exr/cryptomatte/adb0a84/conversion = uint32_to_float32
exr/cryptomatte/adb0a84/manifest = {"bunny":"13851a76",
       "default":"42c9679f"}

exr/cryptomatte/be93ba3/name = CryptoMaterial
exr/cryptomatte/be93ba3/hash = MurmurHash3_32
exr/cryptomatte/be93ba3/conversion = uint32_to_float32
exr/cryptomatte/be93ba3/manifest = {"bunny_porcelain_mat":"b24b26d4"}

exr/cryptomatte/d593dd7/name = CryptoObject
exr/cryptomatte/d593dd7/hash = MurmurHash3_32
exr/cryptomatte/d593dd7/conversion = uint32_to_float32
exr/cryptomatte/d593dd7/manif_file = manifest.1001.json
```

Sidecar manifest path processing examples:

```
EXR file:                 "c:/renders/myFile.1001.exr"
Sidecar path in header:   "mySidecarFile.json"
Resolved path to file:    "c:/renders/mySidecarFile.json"
```

```
EXR file:                 "c:/renders/myFile.1001.exr"
Sidecar path in header:   "manifests/cryptoObject.1001.json"
Resolved path to file:    "c:/renders/manifests/cryptoObject.1001.json"
```

Example sidecar manifest path processing Python code:

```
def resolve_manifest_paths(self, exr_path, sidecar_path):
    import os
    return os.path.join(os.path.dirname(exr_path), sidecar_path)
```

Manifest

The manifest contains both the names and the IDs of all of the objects in the Cryptomatte type. It may be included in full in the header, or it may be a separate file. The format of the manifest is a JSON dictionary, where the names are the keys and the IDs are the values.

```
exr/cryptomatte/d593dd7/manifest = {"bunny":"13851a76",
       "default":"42c9679f"}
```

In the case of `uint32_to_float32` style conversion (the only style currently supported), the hashes are hexadecimal versions of bytes making up the 32-bit float hashes (little-endian). These are always 8 digits of hexadecimal.

In this C++ sample code for converting a floating point hash to hexadecimal, `float_value` is the value which will be used in the pixels of the image. Its bits are copied to unsigned integer, and the hex digits of this integer are written to a string:

```
uint32_t float_bits;
std::memcpy(&float_bits, &hash_value, 4);
char hex_chars[9];
sprintf(hex_chars, "%08x", float_bits);
```

These strings of hex characters are decoded back into floats in Python using the struct module. In this example, hex_string is a string containing 8 characters of hexadecimal:

```
import struct
float_value = struct.unpack('=f', hex_string.decode('hex'))[0]
```

Including the manifest is optional but very good for user experience when decoding. Decoders should not rely on it to accurately extract mattes.

ID Generation

Overview

IDs are generated from string values inside the renderer. In our implementation, we use names, but any string values may be used.

Names

Any string values may be used to compute IDs. Where these string values overlap, the IDs will be combined (an expected and desired behavior).

It’s suggested that in most implementations, the most useful groupings will come from namespace, object name (sans namespace), and material name (also sans namespace), but arbitrary data can be used.

Each set of names should be written to the manifest. If this is not possible (due to deferred loading), matte extraction will still work but users will not have access to the names.

Hash

Only one hash is currently supported, the 32 bit variant of MurmurHash3 (Also known as MurmurHash3_x86_32).

If there is no object in the sample, the recorded hash value should be zero.

Hashes may be computed while rendering or may be pre-cached. Our implementation does hashing while ray tracing, allowing it to handle edge cases such as delayed load render archives. MurmurHash3's speed enables us to do this.

String Encoding (UTF-8)

Unicode is supported via UTF-8. The UTF-8 encoding is identical to ASCII for strings containing ASCII characters, and as such this allows interoperation with encoders and decoders that only support ASCII (for strings containing only these characters).

Names should be encoded in UTF-8 prior to hashing. The bytes of the UTF encoding (and not the character code points) are hashed.

Decoders should try to support unicode characters, although if this is not possible only manifest-related features will fail. Encoders only need to support UTF-8 if names containing Unicode characters are possible in their environment.

Floating point conversion

We store our IDs as 32 bit floating point values. While not ideal, this is necessary for a variety of software specific reasons.

We use as many 32-bit floating point values as possible to help minimize hash collisions. Unfortunately, all 32 bits cannot be used, due to NaN and inf signal values potentially being indistinguishable from each other in certain contexts, and being problematic for computation in general.

Our conversion of the 32 bit integer hash to a 32 bit (single precision) float is effectively a bit for bit copy, with logic to clamp the exponent to acceptable values. The exponent is modified by clamping it to be within the [1:254] range.

Only little-endian values are supported.

Example code and conversions

This example code shows conversion of hashes to float in Python and C++. The hashing function used is a standard MurmurHash3 and is not shown here. The C++ example also shows a method for creating two additional channels of data for the normally filtered “keyable” image.

Python example code:

```
def mm3hash_float(name):
    import pymmh3
    hash_32 = pymmh3.hash(name)
    if hash_32 < 0:
        hash_32 = (-hash_32 - 1) ^ 0xFFFFFFFF

    mantissa = hash_32 & ((1 << 23) - 1)
    exp = (hash_32 >> 23) & ((1 << 8) - 1)
    exp = max(exp, 1)
    exp = min(exp, 254)
    exp =  exp << 23
    sign = (hash_32 >> 31);
    float_bits = exp | mantissa
    packed = struct.pack('@l', float_bits)
    if sign == 1:
        return -struct.unpack('@f', packed)[0]
    elif sign == 0:
        return struct.unpack('@f', packed)[0]
```

C++ example code:

```
float hash_to_float(uint32_t hash) {
    uint32_t mantissa = hash & (( 1 << 23) - 1);
    uint32_t exponent = (hash >> 23) & ((1 << 8) - 1);
    exponent = std::max(exponent, (uint32_t) 1);
    exponent = std::min(exponent, (uint32_t) 254);
    exponent = exponent << 23;
    uint32_t sign = (hash >> 31);
    sign = sign << 31;
    uint32_t float_bits = sign | exponent | mantissa;
    float f;
    std::memcpy(&f, &float_bits, 4);
    return f;
}

void hash_name_rgb(const unsigned char *name, float &r_out, float &g_out,
    float &b_out)
{
    // If called with char* strings,  use static_cast<unsigned char *>
    // unsigned char* are required for UTF-8.
    // This puts the float ID into the red channel, and the human-readable
    // versions into the G and B channels.
    uint32_t m3hash = 0;
    MurmurHash3_x86_32(name, strlen(name), 0, &m3hash);
    r_out = hash_to_float(m3hash);
    g_out = ((float) ((m3hash << 8)) /  (float) UINT32_MAX);
    b_out = ((float) ((m3hash << 16)) / (float) UINT32_MAX);
}
```

Some examples of the results of these hashes and conversions. Note that signed and unsigned versions of the hashes are given. Unlike our C++ code, pymmh3 produces signed hash values, though the bits of both values are the same.

```
Name: torus
UTF-8 bytes: 74 6f 72 75 73
Hash value (signed): -241404931
Hash value (unsigned): 4053562365
Float converted: -1.54943624832e+30

Example containing unicode:

Name: равнина
UTF-8 bytes: d1 80 d0 b0 d0 b2 d0 bd d0 b8 d0 bd d0 b0
Hash value (signed): -1776070370
Hash value (unsigned): 2518896926
Float converted: -1.31926312124e-25
```

Ranking Pixel Filter

Overview

To encode Cryptomattes, a ranking pixel filter is required. For a better illustration of this, please see the poster, “Fully Automatic ID Mattes with Support for Motion Blur and Transparency”.

Non-refractive transparency should be taken into account by the filter as well while weighting samples, in whatever way matches the way the renderer computes opacity.

While rendering, computed IDs per sample are stored in AOVs. The ranking filter uses an arbitrary filter kernel that should ideally match the kernel used to filter the associated “beauty” renders. While iterating through the samples contributing to the pixel, those samples are weighted by the kernel. The weights of each unique ID are summed. Non-refractive transparency is taken into account here as well, to match the way the renderer computes opacity. The weights for each ID are normalized by dividing by the total weight of samples contributing to the pixel, and these normalized weights become "coverage" values.

Each unique ID found contributing to the pixel is ranked by its "coverage" values, and the highest ranked ID is ID 0, the next is ID 1, etc. These IDs and associated coverage values are encoded into the EXR channels as described above.

Alternative ranking

It is not a problem for the ranking filter to determine importance by other criteria than weight. For example, it could allow a user to specify objects that are always ranked first if they contribute to the pixel at all, regardless of how insignificant the coverage is. In this way it could allow users to specify objects that always have accurate mattes.

Change Log

Version 0.1.0

This version of the specification is just a formal writeup of what the Psyop tool is doing currently. It is only meant as a starting point for the committee.

Version 0.2.0

Changes the hash to MurmurHash3.

Version 0.3.0

Changed the format of the metadata to allow multiple Cryptomatte types per image and required the metadata to specify both the hash and conversion method for each of the Cryptomatte types in the EXR file.

Version 1.0.0

Fixed some minor errors in code.

Version 1.0.2

Fixed python code to work with signed ints coming from compiled mmh3 module and updated pymmh3 module.

Version 1.1.0

Changed metadata to use an arbitrary (max) 7-character string as the "metadata key", rather than just an integer. Also in metadata, changed "manifest_file" to "manif_file", in order to accommodate 7 character keys and the EXR metadata limit.

Version 1.2.0

Changed sidecar manifest scheme to only support relative paths, always relative to the image. String encoding codified as UTF-8. Added hashing and encoding examples. Removed vendor-specific data type from example code and changed function declaration to unsigned char*. Clarified exact variant of MurmurHash3 used. Little-endian specified. Changed specification to a PDF file.