Assumptions

Assumptions: 1

Consider the assumption that indexed color mode does not allow partial transparency. How did this assumption spread into the file format?

Answer

Because they assume indexed-color mode does not allow partial transparency, they are able to redundantly use alpha-channel values 0-127 to represent transparent, and 128-255 to represent opaque. This file format hence cannot distinguish between a perfectly-transparent color in this version of GIMP from a partially-transparent color in a future version of GIMP that does support partial-transparency in indexed-color mode.

Assumptions: 2

Suppose you're a GIMP developer in 2014, and you'd like to add support for semitransparent pixels in indexed color mode. There are many old XCF files floating around which use arbitrary alpha values in indexed color mode, since they're all treated the same. You'd like new versions to display these images the same as old versions. What changes do you need to make to support this?

Answer

All layers in indexed-color, or at least those with semitransparent colors, must be saved with something (i.e.: version number) distinguishing it from files saved with an older GIMP. When loading an older file, a pass must be done over all semitransparent pixels in all colormaps to convert their alpha-channel to value 0 or 255.

Assumptions: 3

In truth, the GIMP developers did not alter the file format at all when they started supporting semitransparent index colors. Suppose you created an indexed-mode image with semitransparent pixels in a new version of GIMP, and opened it in 2.8.22 or older. What would you expect to happen?

Answer

Those pixels would display as either fully opaque or fully transparent.

Assumptions: 4

What should the GIMP developers have done to prevent this assumption from spreading into the file format? How could they have prevented this problem in the first place?

Answer

They should have normalized all transparent colors to alpha 0 and all opaque colors to alpha 255.

Openness

Openness: 1

Explain how to add a new property to the XCF file format, akin to the existing properties (`PROP_COLORMAP` , `PROP_COMPRESSION` , etc). Is XCF open in the set of properties?

Answer

Add a new property code to the PropType enum in xcf-private.h . Any place where a property list can appear, emit the new property prior to PROP_END . When GIMP reads it in, it will use xcf_skip_unknown_prop to skip past this property, and will read the rest in. XCF is indeed open in the set of properties.

Openness: 2

Pick two different properties. Explain how to add a new field to each. Are these properties open in the list of fields?

Answer

Easy examples:

`PROP_OPACITY` , `PROP_FLOAT_OPACITY` , `PROP_VISIBLE` , `PROP_LINKED` , `PROP_COLOR_TAG` , and more:

These fields all consist of a single 32-bit integer or float. You can easily modify them to add another 32-bit integer or float. However, old versions of GIMP will expect the property to have a smaller size, and would erroneously attempt to read this extra field as the start of the next property. To prevent this, the version number or something would need to be bumped to prevent old versions from reading this file. Alternatively, the property can be given a new code.

Because GIMP ignores the length field when reading in properties, properties are not open in the list of fields. n

Complexity Ratchets

Complexity Ratchets: 1

For each of Format 1, Format 2, and Format 3 of the path header, write down the format as a flat sequence type.

Answer

Format 1:

```
string ++ uint32 ++ byte ++ uint32 ++ uint32 ++ S(1)
 ↪  ++ List[int32 ++ int32 ++ int32]
```

Format 2:

```
string ++ uint32 ++ byte ++ uint32 ++ uint32 ++ S(2) ++ uint32
 ↪  ++ List[int32 ++ float ++ float]
```

Format 3:

```
string ++ uint32 ++ byte ++ uint32 ++ uint32 ++ S(3) ++ uint32 ++
 ↪  uint32 ++ List[int32 ++ float ++ float]
```

Complexity Ratchets: 2

Are any of these types a subtype of another? Think about the Liskov substitution principle.

Answer

None of these is a sutype of the other. However, if a program ignores version codes and has the ability to ignore lengths, then Format 3 would be a subtype of Format 2.

Common feedback:

Format 3 would be a subtype of format 2 if fields were labeled by name rather than by binary index, or if the format contained a length and used it to skip past extra fields. As it stands currently, format 3 is not quite a subtype of format 2.

Complexity Ratchets: 3

Write down a common supertype of all three formats. If given a value of this supertype, what code would you need to write to destruct such a value?

Answer

```
string ++ uint32 ++ byte ++ uint32 ++ uint32 ++ (version: S(1) U S(2)
 ↪  U S(3))
   ++ (if version >= 2 then uint32 else ())
   ++ (if version == 3 then uint32 else ())
   ++ List[int32 ++ (if version == 1 then int32++int32 else
    ↪  float++float)]
```

Alternative rendition:

```
string ++ uint32 + byte + uint32 + uint32 +
    ( S(1)
    U S(2) ++ uint32
    U S(3) ++ uint32 ++ uint32)
   * List[int32 ++ (if version == 1 then int32++int32 else
    ↪  float++float)]
```

The first 5 fields could be read unguarded; the version would need to be checked to see if it's a valid value.

The next 2 fields would be conditionally read, guarded by a version check.

The list would be read in a loop, with the `x/y` either read as a `float` or as an `int`, using an if-statement on the version.

Complexity Ratchets: 4

Compare your answer in the previous question to the actual code that reads one of these paths. How could the GIMP authors have predicted the need for this if-statement from the initial design of the `PROP_PATHS` format?

Answer

It's read almost-exactly as predicted above. The differences are that they choose to move a couple common-elements inside if-statements: the dummy field is read separately for versions 2 and 3, and point-types are read in both branches of the if-statement for versions 1 and 2/3.

This was completely predictable from version 1: they did not have a way to add fields without breaking compatibility (i.e.: no subtypes), meaning if-statements were necessary to read from multiple versions.

Complexity Ratchets: 5

Give a different way of expressing Formats 1, 2, and 3 such that Format 3 is a subtype of Format 2, and Format 2 is a subtype of Format 1. How would this enable pre-1999 versions of GIMP to read newer files? How would the `xcf_read_old_paths` code differ in modern versions of GIMP?

Answer

Begin the data structure with the length of fields before the point-list, and with the length of each point. Then old versions of GIMP can skip past newer fields. Newer versions can conditionally read these fields using some variant of a read_if_exists or read_with_default function.

The change of point coordinates from int32 to float is more troublesome, although it could have been anticipated by giving a description of the Point structure, so that the read-code can condition on a type descriptor rather than on the version, allowing it to use more general machinery. However, since the current GIMP interprets the integers of PATHS v1 as floats, it's not clear any change is actually necessary.

Common feedback on coercive vs. structural subtyping:

A point of confusion these all share is confusing structural and coercive subtyping (which, in fairness, we didn't really discuss in the course). Coercive subtyping is the "has enough information" definition. Under that definition, the mean of a set of integers is a supertype of the set itself. Coercive subtyping is relevant to your ability to write a format-specific converter to some common format that you can code against. Structural subtyping concerns the actual layout of data and the ability to write identical code dealing with different formats, without needing an extra conversion layer. Having a new format be a structural subtype of the old format means you can write code that deals with both cleanly. Having a new format be a coercive subtype of the old format means it's possible to write code that deals with both if it's possible to determine which format a given file comes from, but the code will need some branching and will not be clean.

Bonus questions

Bonus questions: 1

Compare the `PROP_VECTORS` format to `PROP_PATHS`. What changes can be made to `PROP_VECTORS` while allowing forward-compatibility (i.e.: old versions can read newer files)? How is `PROP_VECTORS` more open than `PROP_PATHS`?

Answer

- An arbitrary number of parasites may be added per path
- More floats may be added per point
- New types of points may be added

`PROP_VECTORS` is open in the fields of paths and points, unlike PROP_PATHS.

Bonus questions: 2

Suppose you were a GIMP developer in 1996. What rules might have led you to design a format more similar to `PROP_VECTORS` than `PROP_PATHS`, so that the switch from

`PROP_PATHS` to `PROP_VECTORS` (and associated loss of compatibility) need never have happened?

Answer

Always assume new fields will be added, and make it easy to do so.