PhysX/APEX/architecture/design/File Formats

Accuracy:
Depth:
Readability:

From engwiki

< PhysX | APEX | architecture

Contents

1 APEX Asset Formats

There are two primary file formats supported for all APEX assets:

- XML format

  Used during authoring stage for debugging, manual/automatic editing, conversion, etc... Usually 3x times bigger and slower than binary formats.

- Binary format

  Much more efficient (disk space and parsing time) than XML. Endianess and alignments depend on the target platform that was specified during file creation allowing extremely fast inplace deserialization on native platform (objects deserialized into this format, can be fread() into memory, and used with in-place construction, to minimize copy time and memory allocation overhead).

1.1 XML format requirements

XML may contain xml declaration (<?xml ... ?>) which currently does not have any influence on parsing.

NxParameterized-compatible XML must contain a simple DTD <!DOCTYPE NxParams> (at the beginning of file) which allows robust filetype detection.

The root element of XML is NxParams. It may have optional attributes numObjects (number of top-level NxParameterized-objects in file) and version (version of XML format in major.minor-format).

TODO: write about other tags

1.2 Binary format requirements

1.2.1 Dictionary

The binary format has a dictionary that maps unique strings, to unique (within the scope of the stream) unsigned 32-bit integers. All values of string or enum type are stored as integers within the stream, but are mapped back to the actual string value via the dictionary.

Note that the same string should never map to multiple integers in the dictionary. This restriction allows quick string equality checks by simply checking whether the integers they map to are equal.

1.2.2 Multiple objects in one stream (share dictionary)

Multiple objects may be serialized into one stream. This reduces file size by allowing reuse of the dictionary across multiple objects.

1.2.3 File layout

The file is composed of several parts:

1. Header
2. Dictionary
3. Data

   Contains the serialized objects

4. Layout metadata (optional)

   If defined, this section may contain metadata defining the schemas and layouts for the classes that serialized objects within the data section may belong to

5. Pointer fixup table

   Contains information for converting deserialized offset data to direct pointer references (AKA pointer swizzling).

The header must always be the first section, immediately followed by the dictionary. All other sections have no order restrictions, as long as they follow the header and dictionary.

1.2.4 File header

The file header for a serialized set of objects includes:

- Magic

  A unique magic string, useful for identifying the file as an APEX binary

- Major/minor version

  Major and minor version numbers for the file format

- Detailed target platform information (used during deserialization to determine alignments and endians)

  Processor architecture, compiler version, OS version

- Offset to dictionary

  The offset in bytes from the beginning of the file, to the dictionary

- Offset to data section

  The offset in bytes from the beginning of the file, to the data section

- Offset to pointer fixup table (only in platform-optimized binary)

  The offset in bytes from the beginning of the file, to the pointer fixup data

- Offset to layout metadata

  The offset in bytes from the beginning of the file, to the layout metadata section. This is an optional section, so its assumed that this section doesn't exist if this value is 0.

Current implementation (all fields are in big endian for compatibility):

```
#pragma PX_PUSH_PACK(1) //We need this for cross-platform compatibility
struct BinaryHeader
{
        physx::PxU32 magic;
        physx::PxU32 type;
        physx::PxU32 ver;
        physx::PxI32 numObjects;

        //Offsets
        physx::PxU32 fileLength;
        physx::PxU32 dictOffset;
        physx::PxU32 dataOffset;
        physx::PxU32 relocOffset;
        physx::PxU32 metadataOffset;

        //Platform information
        physx::PxU32 archType;
        physx::PxU32 compilerType;
        physx::PxU32 compilerVer;
        physx::PxU32 osType;
        physx::PxU32 osVer;
};
#pragma PX_POP_PACK
```

1.2.5 Data section

Data section starts with table of top-level objects:

```
struct ObjectTableEntry
{
	NxParameterized::Interface *objHeaderOffset; //Absolute offset of object header
	const char *mClassName; //Absolute offset of object class name
	const char *mName; //Absolute offset of object name
	const char *mFilename; //Unused field
} objectTable[numObjects];
```

Table is followed with actual data of top-level objects and their subobjects (included and named references).

1.2.6 Optional metadata section

Warning - this is still a draft.

Data section starts with number of classes with stored metainfo (represented with physx::PxU32) followed by table:

```
struct MetadataTableEntry
{
	const char *className;
	physx::PxU32 classVersion;
	MetadataInfo *root;
} metadataTable[numObjects];
```

This is followed with actual MetadataInfos:

```
struct MetadataInfo
{
	physx::PxU32 type; //TYPE_ARRAY, TYPE_U32, etc.

	physx::PxI32 arraySize; //-1 if dynamic

	const char *shortName;

    const char *structName;

	physx::PxU32 numChildren;
	MetadataInfo *children[numChildren];

	physx::PxU32 numHints;
	HintInfo hints[numHints];

	physx::PxU32 numEnumVals;
	const char *enumVals[numEnumVals];

	physx::PxU32 numRefVariants;
	const char *refVariants[numRefVariants];
};
```

HintInfo is

```
struct HintInfo
{
	physx::PxU32 type; //TYPE_U64, TYPE_F64 or TYPE_STRING
	const char *name;
	union
	{
		physx::PxU64 uval;
		physx::PxF64 fval;
		const char *uval;
	} val;
};
```

All strings are stored in dictionary.

1.3 Per-object data

Each object in the data section, is preceded with a small header, containing per-object data:

1.3.1 Data offset

This is an offset in bytes, from the beginning of the per-object data header, to the beginning of the actual data for the object. It's essentially the size of the per-object header.

This field allows future-proofing, because a reader can use it to skip past new fields that increase the size of the header.

1.3.2 Class name

This is a string (integer that can be looked up in the dictionary) containing the name of the class the object is an instance of.

1.3.3 Class version

A version number, specific to the class.

1.3.4 Class schema checksum

A checksum for the schema of the class. This is useful when the version number is (wrongfully) not updated, even though the schema changed.

1.3.5 Object data checksum

A checksum of the data in the data section of the object.

1.4 Object construction during deserialization

There are two possible methods for constructing application objects, from the serialized data during runtime. Both methods work via callbacks that are provided by the application (APEX) to the file reader. Callbacks are called every time a new object is encountered in the stream:

1.4.1 Method 1

As the file reader encounters each object, it:

1. Immediately invokes a callback with the name of the class of the object, as soon as it reads it from the stream.
2. The callback constructs a new object with default settings for all the parameters), and returns it to the deserializer.
3. The reader calls a deserializeStart() method on the NxParameterized object (doesn't exist yet).
4. For each parameter the reader encounters, it sets the value in the object via the appropriate set method.
5. When all parameters have been set, the reader calls the deserializeEnd() method on the object.

1.4.2 Method 2

This is currently not implemented!

As the file reader encounters each object, it:

1. Aggregates all the object data into a buffer. The data is laid out, such that it matches the expected layout for the object on the platform that's doing the reading. If the layout in the file matches the reading platform (alignment and endian match), then the data can be read verbatim into the buffer.
2. Once all of the data is available in the buffer, a callback is invoked, with the class name and buffer being passed to it. This potentially allows APEX to do in-lace construction on this buffer.