# USDZ File Format Specification



## Purpose

USD provides multiple features that could make it a compelling choice for 3D content delivery, including:

- Robust schemas for interchange of geometry, shading, and skeletal deformation
- High performance data retrieval and rendering, including powerful instancing features
- The ability to package user-selectable content variations, natively
- A sound architecture that is flexible enough to adapt to future needs

However, part of USD's appeal is its ability to create a 3D scene by "composing" many modular data sources (files) together into successively larger and larger aggregations.  While very useful within content creation pipelines, this aspect can be a very large hindrance to using USD to deliver assets in the form that they have been built up.  In particular, even though USD does provide several means of "flattening" multiple USD files into a single file, there is, by design, no mechanism for representing images/textures as the "scene description" encodable in USD files. Content delivery is simplified and useful on a broader range of platforms when the content is:

1. A single object, from marshaling and transmission perspectives
2. Potentially streamable
3. Usable without unpacking to a filesystem

We believe we can address these concerns by leveraging USD's FileFormat plugin mechanism to design an *archive format* that contains and proxies for files of other formats embedded within the archive.  This document provides the specification for such a format, for which we will assign the .**usdz** extension.

## Usdz Specification

| Aspect | Structure/Behavior | Rationale |
|---|---|---|
| Foundation | A usdz file is a zip archive. | Rationale |
| Zip Constraints | A usdz file is a **zero compression, unencrypted** zip archive. We reserve the ability to relax this constraint in the future, but consider it unlikely. | Rationale |
| Layout | The only absolute layout requirement a usdz file makes of files within the archive is that the data for each file begin at a multiple of 64 bytes from the beginning of the archive.<br><br>However, if you wish the archive to be presentable on a UsdStage "as is", or be able to target the archive with a simple reference to the archive itself, then **the first file in the archive must be a native usd file**, but otherwise, there are no constraints on the number or layout of other files in the archive. We refer to this "first USD file" as **the Default Layer**, in analogy to the *defaultPrim* metadata used to allow layer referencers to elide a prim target.<br><br>Clients wishing to deliver "streamable content" may wish to consider other layout constraints, as well. | Rationale |
| File Types | A usdz file can contain only file types whose data can be consumed by the USD runtime via mmap or (eventually) pointer to memory. This excludes, for example, Alembic files, currently. Initially this will include only the native USD file formats and a small number of image formats, but may grow over time. | Rationale |
| USD Constraints | To make usdz archives maximally useful within production pipelines, we impose no further constraints on the allowable content within an archive. Of particular note:<br><br>- it will be possible to reference individual fies within an archive *from outside the archive*<br>- archives can themselves contain other archives.<br><br>When the key intention of an archive is to provide perfectly reproducible results in arbitrary consuming environments, it may be useful to enforce restrictions on the kinds of *asset paths* encoded in USD files within the archive. We explore this further in For Reproducible Results, Encapsulate Using Anchored Asset Paths, and imagine that adherence to such protocols may be something useful to encode in archive metadata (see next item). | Rationale |

| | | |
|---|---|---|
| Editability | A usdz file is read-only - editing its contents requires first unpacking the archive and editing its constituent parts using appropriate tools. Since usdz is a "core" USD file format, one can use *usdcat* and *usdedit* on archives:<br>• If the archive contains a Default Layer, usdcat will print its contents, otherwise, it prints nothing.<br>• If the archive contains a Default Layer file, usdedit will populate an editor with its contents, but with the *--noeffect* option in force, preventing the saving of any changes; otherwise, it will show an empty layer. | |

# Rationale

## Foundation - Zip Files

Zip files are ubiquitous and supported by most modern computing environments. Although we are only using a subset of zip's featureset, by choosing zip over a format of our own devising, we make it easy for usdz files to be useful as a "simple" transport protocol whereby the receiver just unzips the contents and has "normal" USD scene description to work with and inspect.

## Zip Constraints - No Compression or Encryption

A key consideration for efficient *direct* consumption of usdz files is that, given an archive already held in heap storage (possibly arriving over a network) or as a single file on disk, we be able to use the most direct API's available in USD for accessing the data contained within the archive, **without extracting files to disk, or allocating more heap storage**. If we allowed either compression or encryption in usdz archives, we would need to violate one or both of the preconditions that allow, for example, the usda and usdc formats to access their data via direct memory access (typically via mmap).

We do not believe the lack of zip compression will be a serious concern for data size. Most image formats themselves allow internal compression schemes, and the usdc format is quite compact, particularly as you collect more data into a single file; although we do not yet have an end-user tool for doing so, USD now contains all of the core features we require to aggregate an arbitrary composition's worth of usd files into a single file, without removing any composition features from the scene.

## Layout - Alignment and the Default Layer

### Alignment

To achieve access to data within an archive without unpacking it, our primary implementation will rely on mmapping - either the entire archive once, or (more likely), as an offset mapping into the archive, for each file. Some file formats and processing algorithms of the data within those archives benefit from aligning array data on boundaries greater than one byte. Of particular importance, the "zero copy" feature of usdc (crate) files that will appear in the 0.8.5 release of USD returns to clients pointers to array data directly into the mmapping of the crate file; to avoid unspecified behavior, the starting address of each such array must be aligned to the size of the underlying POD datatype. The crate format itself internally aligns (and falls back to copy data into heap memory when the alignment is off. The performance advantages of zero-copy in both time and heap memory are substantial, and we wish to ensure that consumers of usdz files will observe the same benefits; therefore, we require a **minimum alignment of 8 bytes**. The *largest* such alignment constraint for performance boosting of which we are currently aware is the 64 byte alignment for Intel's AVX512 instruction set. While there might in future be an AVX1024 or greater, given that 64 bytes is also the intel processor page size, we hopefully balance archive-bloat with potential optimization and settle on 64 bytes.

Therefore we require that each file begin on a 64 byte alignment within the archive. Fortunately, the zip format gives us a variable sized field in each file's header section that we can appropriate to this purpose. Unfortunately, off-the-shelf zip tools do not allow specification of padding or alignment for zipped content, so USD will need to provide its own wrapping of the core zip API's, and its own packaging tool that uses them.

### Default Layer and defaultLayer.usd

The Default Layer, if it exists, is **always the first file in the archive**. The Default Layer is the layer that will be returned by a call to SdfLayer::FindOrOpen("archive.usdz"), and therefore also the root layer when the archive is placed on a UsdStage, or when referenced or sublayered as a whole. Given that, due to alignment considerations, we will need to provide packaging utilities more sophisiticated than "zip", we can always allow you to specify what the Default Layer should be, as a command option. However, to make the packaging process as simple as possible – particularly focusing on the workflow for "editing" an archive, in which we must unpack an archive, edit files, and then **repackage the archive**, losing the positional information originally present in the archive – we provide some extra affordances to reduce the scenarios in which one must manually specify the Default Layer:

- If the complete set of files handed to the packager contains only a single USD file "at archive root scope", then that will be selected as the Default Layer, and added first to the archive. This does not prevent one from including arbitrary other USD files within subdirectories within the archive. This allows for reliable and easy unpacking/repackaging, and multiple archives can be unpacked into the same

directory as long as their contents are differently named.
- If there is more than one USD file at the archive's root scope, and one of them is named `defaultLayer.usd`, then we select that layer as the Default Layer. This also allows for easy unpacking/repackaging, with a bit of up-front work during original archive creation, and also makes it less likely that one could safely unpack multiple archives directly into the same directory, but provides a clear and familiar (i.e. to defaultPrim metadata) mechanism for encoding arbitrarily complex organizations within an archive.
- Otherwise, the packaging API will fail to run unless the Default Layer is manually specified.

### File Types - DMA

We wish to avoid the ***need*** to ever unpack a usdz file to consume its contents in the USD runtime, both because some target platforms may be unable to do so, and because managing caches of unpacked archive contents between processes is fraught.

### USD Constraints - Asset Resolution

There is no mandated restriction on types of asset paths encodable in a usdz archive, so for example if a studio uses URI's whose resolution depends on a specific ArResolver implementation, it is allowed to embed such paths in an archive, with the understanding that they will not be resolvable at other sites, and therefore likely not as useful for content-delivery to clients. Archives can additionally contain both ***anchored* relative paths**, like `@./model.geom.usd@` , and ***search* relative paths**, like `@materials/metals.usd@`.

**Anchored relative paths** will always resolve to other files within the archive, or be unresolvable. Because usdz archives are themselves containers, we think it a useful builtin behavior that, for any **search relative path** contained in the archive, the usdz archive itself will attempt to pre-resolve any path to other files within the archive, because a generic ArResolver cannot "see inside" an archive, and therefore cannot resolve to files inside an archive:

- First attempt to anchor the path to the layer-within-the-archive in which the path is authored, to attempt to locate another file within the archive
- Should that fail to locate a file, attempt to anchor the path to the archive's Default Layer, which is the same behavior one would get from unpacking the archive and putting its unpacked Default Layer on a stage, when using the ArDefaultResolver.
- Should that fail to locate a file, perform normal asset resolution on the path with whatever ArResolver is installed.

We can construct references to any file inside an archive using SdfFileFormatArguments in identifiers, like `@foo.usdz:SDF_FORMAT_ARGS:path=/path/to/file/within/archive.usd@` . The Arguments are provided to the UsdzFileFormat when opening a layer, which it uses to locate the desired file within the archive. When an asset path authored inside an archived file is resolved, it will return a path of this form. Astute readers familiar with the Sdf and Ar API's will notice that we are relying on SdfFileFormat behavior that currently only triggers for USD layer access, not more general asset file types like images/textures. Our long-term goal is to abstract the notion of "file" to "asset" and FileFormat to AssetFormat. In the nearer term, we will provide some API that, given a resolved, localized asset path can:

- tell you whether there **is** a free file available to fopen()
- regardless, make the file's contents available in memory, returning an object that contains a size and pointer to (possibly mmapped) the file's contents in memory, and whose lifetime preserves the validity of that pointer.
- regardless, create an actual file on the filesystem if there was no free file already, such that the asset path can be fopen()d.

## Packaging Considerations for Streaming and Encapsulation

### File Ordering Within Archive for Streaming

There are many interpretations and possible implementations/encodings of "streaming usd context", and we feel it is, at this time, beyond the scope of USD's concerns to dictate how streaming *must* be achieved. We limit our concern in this matter to the already-stated "Default Layer" semantics, which dictates the first file in the archive must be a usd file for the archive to be imageable directly on a stage. This means that a consumer app can, using the zip (or wrapper, provided by the usdz file format) API, determine the size of the first file and therefore know when it has been fully delivered, and display the contents of that file in its "default state", while waiting for subsequent files (described in the layer's metadata) to be delivered. This enables many kinds of streaming; to suggest just a few:

- LOD, in which the first file contains low-complexity geometry, with a variantSet that can bring in higher quality LOD's in other files, when switched
- LOD, in which the first file binds no or simple materials, with a variantSet that binds texture-driven Materials when the textures are available
- Animation, in which the first file contains a static pose, with a payload arc to another file containing animation.

It is important to note that the USD runtime has no knowledge or consideration of streaming: it will be the responsibility of the client application to manage the scene it is streaming, and ensuring not to ask the USD runtime to consume any part of the archive that has not yet been delivered. In particular, it must use the provided API's to construct an SdfLayer directly using offsets into an incompletely downloaded usdz archive, rather than trying to point a UsdStage directly at the incomplete usdz archive, as the UsdzFileFormat will assume and require that the archive it is consuming is complete and intact.

### For Reproducible Results, Encapsulate Using Anchored Asset Paths

To insure uniform consumption of assets shipped to clients via usdz, we feel it is prudent to curtail the power of USD's asset resolution system, so that the asset references within an archive resolve uniformly on any consuming system, regardless of how that system's USD ArResolver is configured. Rather than make restrictions within the FileFormat itself, we propose that such "encapsulation" of archives be achieved by restricting the *content* to only use **anchored** paths (paths that begin with "./" or "../", both of which are interpreted in the virtual filesystem described by the archive's internal layout.

This does not prevent one from overriding the textures or other non-USD assets contained in an "encapsulated" archive, by *specifically overriding the attributes that name the assets, in a layer stronger than the archive in a composition.*

## Toolset

The UsdzFileFormat will include helper API's for introspecting and extracting data from an archive, as well as tailored wrappings of the zip API functions for creating and adding files to an archive that satisfy the usdz layout constraints.

Additionally, we will provide standalone API and comman-line tool for creating usdz archives with various options.  This archiver will handle the process of "localizing" all the files referenced by any USD file added into the archive, as well as modifying asset paths within the layers to reflect the "localized archive filestructure".  The very first version of this archiver may only support localization via UsdStage::Flatten(), to expedite the process of making available something that works, acknowledging that it does not preserve composition arcs, so will not initially support delivery of variantSets.