Cinema Database Specification
Chaplin Release v1.0

Document Version 1.4
(31 January 2017)
LA-UR-17-20645

by

David Rogers dhr@lanl.gov
Jon Woodring woodring@lanl.gov
John Patchett patchett@lanl.gov
David DeMarle dave.demarle@kitware.com
Berk Geveci berk.geveci@lanl.gov

Los Alamos National Laboratory
Bikini Atoll Rd., SM 30
Los Alamos, NM 87545
cinema@lanl.gov

Contents

1 Cinema Overview

Extreme scale scientific simulations are leading a charge to exascale computation, and data analytics runs the risk of being a bottleneck to scientific discovery. Due to power and I/O constraints, we expect in situ visualization and analysis will be a critical component of these workflows.

Options for extreme scale data analysis are often presented as a stark contrast: write large files to disk for interactive, exploratory analysis, or perform in situ analysis to save detailed data about phenomena that a scientist knows about in advance. Cinema represents a novel framework for a third option a highly interactive, image-based approach that promotes exploration of simulation results, and is easily accessed through extensions to widely used open source tools. This in situ approach supports interactive exploration of a wide range of results, while still significantly reducing data movement and storage.

A Cinema database supports an image-based approach to interactive data exploration. It is a set of images and associated metadata that promotes innovative interactions with large datasets. More information about the overall design of Cinema is available in the paper *An Image-based Approach to Extreme Scale In Situ Visualization and Analysis* [1].

A Cinema Database supports the following three use cases. Taken together, these support a novel method for interactively exploring artifacts from extremely large datasets.

1. Searching/querying of meta-data and samples. Samples can be searched purely on metadata, on image content, on position, on time, or on a combination of all of these.
2. Interactive visualization of sets of samples.
3. Playing interactive visualizations, allowing the user on/off control of elements in the visualization.

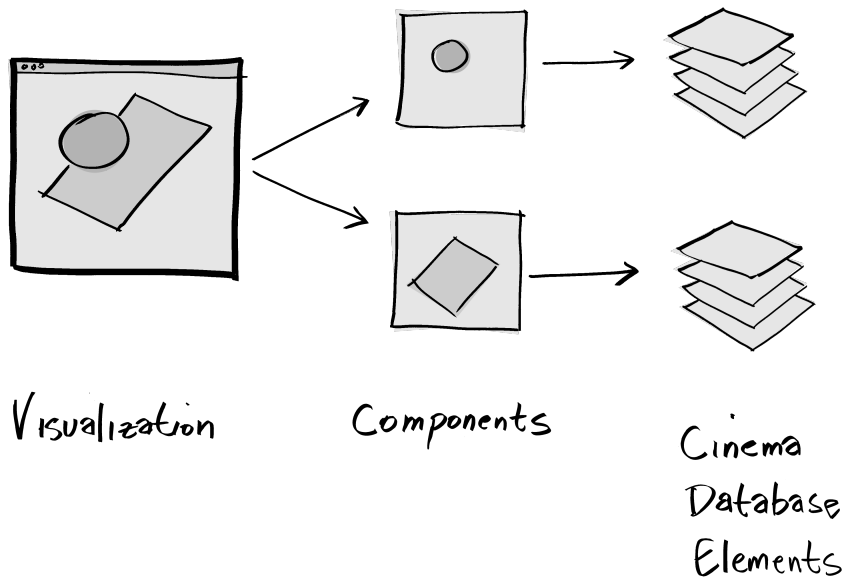

Figure 1: Conceptual diagram of how to map a visualization to elements in a Cinema database. A user starts with a **visualization** that is to be saved as a Cinema database. The user first decides which **components** are desired, and what interactions are needed from the resulting database. This determines the **Cinema database elements** that must be created and saved according to this specification. Elements which are to be independently controlled, queried or analyzed are separately rendered as *images*, and are written into the database as set of *image channels*. The channels written into the database determine what operations are possible on the data when it is retrieved from the database for visualization, analysis or querying. See section 2.2.4.12 for more information on channels.

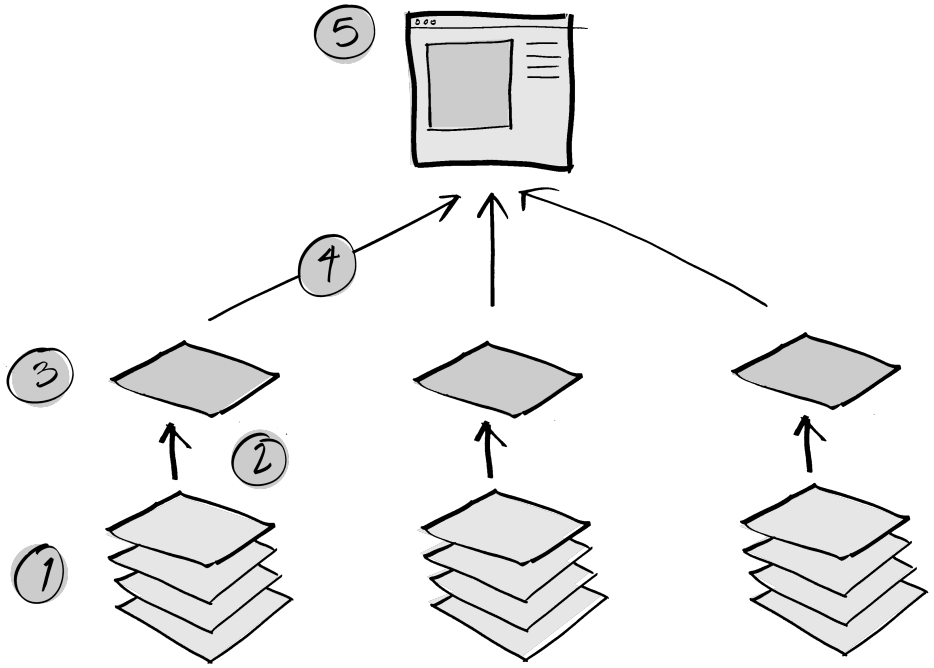

Figure 2: Diagrammatic workflow of components in a Cinema database. *Image channels* (1) are gathered together in a *fusion* operation (2) to create a single *image* (3). The *image channels* and the *fusion* operation are described in section 1.1.2. Any number of *images* can then be combined together in a *compositing operation* (4). The final product of that composite can be shown in a viewer application as a *rendering* (5). The same data workflow can provide *images* or *renderings* to a query application, an analysis operator or other application.

1.1 What is a Cinema Database?

A Cinema database is a set of precomputed visualization components that can be queried and interactively viewed. The user can decide what types of components comprise the database, based on the type of interaction that is desired with the final database. Figure 1 shows a diagram of the way in which final renderings are broken into components, and then finally to image channels that this specification expects. For more detail on the types of interactions that you might create, see [1]. Once a Cinema database is created, a user can create a viewer or analysis operator that somehow makes use of the data. A diagram of this is shown Figure 2. One application that can be created is an interactive viewer that allows a scientist to interactively view the data, as described in [1]. This specification describes data which could be used by such a viewer, but could be ignored by other applications. This is a general design philosophy of Cinema - applications that read a Cinema database can ignore data, read subsets of channels, and otherwise determine which operations to perform. This promotes a wide range of possible interactions with the data.

This document describes release v1.0 of the Cinema Chaplin Database, in which image channels are stored instead of standalone fully pre-rendered images. The constituent images are inputs to a deferred rendering algorithm which allows the user to control which objects are in view and how each is colored.

1.1.1 Parameters in Cinema

A Cinema database holds a collection of results sampled by a set of visualization parameters. Visualization parameters can be created for any control that the user might have in a traditional visualization session. Examples include isosurfaces, slice planes and viewpoint. Cinema viewing applications give the user a control for each parameter setting and

display the corresponding precomputed results, as if the user had made the same choices in a traditional visualization tool from the full resolution input data. As cinema holds only images, it is much more lightweight and display is always a constant time operation regardless of the parameter that is being changed.

Examples of typical parameters in a cinema database are:

- **Time**. Time varying data can be sampled at arbitrary points along the temporal domain.
- **Camera positions**. In a static camera, the position and orientation is fixed. In a spherical camera, the position varies over a set of positions centered around a chosen focal point. More complicated camera tracks are possible.
- **Visualization Operators**. Many visualization approaches use a data pipeline model that can apply operators such as clipping planes and isocontours. These can be rendered as individual elements that can be shown or hidden when viewing a cinema database.

1.1.2 Cinema Image Channels

In a Cinema workflow, the application producing a Cinema Database creates a set of image channels for each sample in the parameter space. Technically it does so by iterating over all assigned values for the color parameter of the currently displayed object and capturing each image. Likewise viewing applications let the user choose an object, take in the set of color results for that object and use them to draw that object on the screen. Each channel is written to disk as an image file, and these are fused together to create a 'normal' image. The fusion operation depends on which channels are present, and this is detailed in later sections.

Figure 2 shows a diagrammatic workflow for pulling data from a cinema database. An application or library pulls specific images from the database, based on some set of parameters. These images result from fusing several *image channels* from the database. These are described in more detail in section 2.2.4.12.

1.1.3 Cinema Metadata

In addition to the sampled data, a cinema database contains metadata that describes what the samples are, how they are stored and how they can be used. It must:

- identify the specific format (version number and content type) of the Cinema database,
- identify specific storage formats for any artifacts in the database (for example, images),
- enumerate the entire set of parameters that are explorable in the database and describe relationships between the parameters where they exist,
- enable applications to map parameter value combinations to storage locations,
- annotate parameters as necessary to indicate how they need to be handled

2 The Cinema Chaplin Specification

This document describes the Cinema Chaplin database so that collaborators can implement readers, writers, algorithms and viewers that perform different operations on Cinema databases. The document is broken into two main sections, one describing a Chaplin grammar, and one that describes the Chaplin implementation of that grammar. Applications and libraries that read and write databases that comply with both the grammar and the implementation in this document can expect to exchange, read and visualize data in a common format.

Subsections in the Grammar section (2.1) have corresponding sections in the Implementation section (2.2) of the document to fully explain the specification. There are several additional subsections in the Implementation section that provide file examples, and algorithms for mapping metadata to files on disc.

We invite public comment on this specification. Please send comments to `cinema-dev@lanl.gov`.

2.1 Chaplin Grammar

The base Chaplin file, "info.json", must adhere to the JSON standard, such that it can be parsed by JSON readers. Descriptive database elements are stored as JSON lists, objects (dictionaries), strings, and numbers. To describe the grammar (syntax) for this file, we mostly follow from a typical Bachus-Naur form (BNF)-like/ Regular Expression (RE)-like conventions. Certain grammar additions or deviations from BNF and RE are explained below, which are intended to shorten the description of the Chaplin grammar.

The "+"/"++" and "*" repetition tokens are meant to represent a rule of a repetition, as usual, but with the convention that there is a comma separating between each element (i.e., it is shorthand for a comma separated list, which is common in JSON). That is: RULE+ or RULE++ means RULELIST ::= RULE , RULELIST | RULE and RULE* means EMPTYORRULELIST ::= "" | RULELIST. The "++" rule has an additional meaning that in its expansion, each sub-rule must exist at least once and only once (i.e., it is a list of unique keys or key-value pairs that must be present). The "??" token, i.e., RULE??, it means that rule is optional in a "++" production rule. This means that that all right-hand elements that are produced from a "RULE++" are required to appear at least once unless noted with a "??". Furthermore, a "??" element is represented with <RULE> on the left-hand side to further represent that it is optional.

To describe unique JSON values (keys), we introduce two additions to the grammar specification. KEY(NAME, SET) represents that a token, from a production rule NAME, is added to a list SET during parsing, where the list is named by the token generated by the rule SET. The token generated by NAME must be unique within that list SET (a specific token must appear only once in its production rule during parsing). REF(NAME, SET) means that it is referring to a token within the list SET (a token may appear as many times as necessary from a production rule during parsing, but the specific token must exist in the list SET as generated by KEY at the end of parsing). If SET is the token "global" rather than a production rule, it means that the tokens generated by KEY NAME are globally unique. Otherwise, it means that the tokens generated by KEY NAME are "scoped" to a list, specifically named by a token that is generated by the rule SET (the list of unique tokens is the list named by SET). Additionally, unique token production rules are represented as $NAME$ on the left-hand side to indicate that they are meant to be unique.

JSONSTRING represents a valid JSON string, JSONNUMBER represents a valid JSON number, and JSONVALUE represents any valid JSON value, i.e., string, number, list, or dictionary. "[" and "]" represent literal list enclosing elements for JSON, rather than character range elements, such as typically used in RE notation. Likewise, "{" and "}" represent literal dictionary enclosing brackets for JSON objects. "(", ")", and "|" represent their typical RE expression meanings (groups and option).

2.1.1 Top level

As noted before, the Chaplin file must conform to JSON standards, which contains nested dictionaries (objects). The top level anonymous JSON dictionary must contain four elements that describe the contents of a Chaplin database.

```
CHAPLIN ::= { TOP++ }
TOP ::= METADATADICT | NAMEPATTERNSTR | PARAMETERDICT | CONSTRAINTDICT
```

The four sub-sections are the metadata dictionary, a name pattern string, parameter dictionary, and the constraint dictionary, which are described further below.

2.1.2 Metadata

```
METADATADICT ::= "metadata" : { METADATA++ }
METADATA ::= DBTYPESTR | DBVERSIONNUM | STORETYPESTR |
  CAMERAMODESTR | ENDIANSTR | IMAGESIZEPAIR |
  VALUEMODENUM?? | CAMERADATA?? | PIPELINELIST??
```

The metadata section is a collection of miscellaneous non-parametric (input) information, such as database type and version information, camera information, output data format and size, and a representation of the source generating data pipeline. It is further described in section 2.2.5.

```
DBTYPESTR :: = "type" :
  ("composite-image-stack" | "parameteric-image-stack")
DBVERSIONNUM :: = "version" : JSONNUMBER.JSONNUMBER
STORETYPESTR ::= "store_type" : ("FS" | "SFS")
CAMERATYPESTR ::= "camera_model" : ("static" |
  "phi-theta" | "azimuth-elevation-roll" | "roll-pitch-yaw")
ENDIANSTR ::= "endian" : ("little" | "big")
IMAGESIZEPAIR ::= "image_size" : [ JSONNUMBER, JSONNUMBER ]
<VALUEMODENUM> ::= "value_mode": (1 | 2)
```

At at minimum, the metadata requires the database type, database version, storage type, camera model type, byte endianness, and image size. Value mode, the pipeline information, and camera information are optional. The pipeline information and camera information elements are described further below.

```
<CAMERADDATA> ::= LOOK++
LOOK ::= EYELIST | ATLIST | UPLIST | NEARFARLIST | ANGLELIST
EYELIST ::= "camera_eye" : [ CARTESIAN3+ ]
ATLIST ::= "camera_at" : [ CARTESIAN3+ ]
UPLIST ::= "camera_up" : [ CARTESIAN3+ ]
NEARFARLIST ::= "camera_nearfar": [ NEARFARPAIR+ ]
ANGLELIST ::= "camera_angle": [ JSONNUMBER+ ]
CARTESIAN3 ::= [ JSONNUMBER, JSONNUMBER, JSONNUMBER ]
NEARFARPAIR ::= [ JSONNUMBER, JSONNUMBER ]
```

The optional camera information is described by several elements: the eye position list, looking at position list, camera up list, the near-far plane list, and the vertical view angle (FOV) list.

```
PIPELINELIST ::= "pipeline" : [ FILTERDICT+ ]
FILTERDICT ::= { FILTERDATA++ }
FILTERDATA ::= NAMESTR | IDNUM | VISNUM | PARENTLIST | CHILDRENLIST
NAMESTR ::= "name" : JSONSTRING
IDNUM ::= "id" : KEY(UUID, global)
$UUID$ ::= JSONNUMBER
VISNUM ::= "visibility" : (0 | 1)
PARENTLIST ::= "parents" : [ REF(UUID, global)* ]
CHILDRENLIST ::= "children" : [ REF(UUID, global)* ]
```

The optional pipeline information is a description of a set of filters (program objects) that are connected together from inputs to outputs in a directed acyclic graph (DAG). These connections are represented by parent and children relationships. The ID value, UUID, is required to be a globally unique number to identify each filter, in the pipeline DAG. These IDs are referred to in the PARENTLIST and CHILDRENLIST, to create the DAG.

2.1.3 Name Pattern

```
NAMEPATTERNSTR ::= "name_pattern" : JSONSTRING
```

The name pattern section describes the default image channel file type and the file path for image production. This primarily is used to locate the image data channels on storage, as described by the parameters.

2.1.4 Parameters

```
PARAMETERDICT ::= "parameter_list" : { PARAMETERDATADICT+ }
PARAMETERDATADICT ::= KEY(PARAMETERNAME, global) : { PARAMETERDATA++ }
$PARAMETERNAME$ ::= JSONSTRING
PARAMETERDATA ::= LABELSTR | VALUELIST | DEFAULTVAL?? |
  ROLESTR?? | VALUERANGEDICT?? | HINTSTR?? | CHANNELLIST??
```

The parameter section is the dictionary of changeable variables (independent, input data). The database will contain, on storage, image data channels. These channels are the result of the Cartesian product of parameters, which generate results (dependent, output data) for every parameter value, except when a parameter is constrained by another parameter value. PARAMETERNAME is the name of the parameter, which is globally unique in the JSON file, and the right-hand value is a dictionary containing the description of that named parameter. Parameters are described fully in section 2.2.4.

```
LABELSTR ::= "label" : JSONSTRING
VALUELIST ::= "values" : [ (JSONVALUE | KEY(VALUENAME, PARAMETERNAME))+ ]
$VALUENAME$ ::= JSONSTRING
```

At a minimum, the description of a parameter requires a LABELSTR and VALUELIST. If a VALUENAME key appears in VALUELIST (a unique name per PARAMETERNAME of the parameter that VALUELIST is contained in), it will be referenced in VALUERANGEDICT to describe the value range of VALUENAME for that PARAMETERNAME parameter.

```
<DEFAULTVAL> ::= "default" : JSONVALUE
<ROLESTR> ::= "role" : ("layer" | "control" | "field")
<VALUERANGEDICT> ::= "valuesRanges" : { RANGELIST+ }
RANGELIST ::= REF(VALUENAME, PARAMETERNAME) : [ JSONNUMBER, JSONNUMBER ]
<HINTSTR> ::= "type" : ("hidden" | "range" | "option")
<CHANNELLIST> ::= "types" : [ CHANNELSTR+ ]
CHANNELSTR ::= "depth" | "luminance" | "value" | "color" | "lut"
```

Parameter data have optional descriptors, DEFAULTVAL, ROLESTR, VALUERANGEDICT, HINTSTR, and CHANNELLIST. ROLESTR describes the primary use for this parameter. VALUERANGEDICT entries are specific to parameters that have a ROLESTR of “field” to describe the value range of the parameter (primarily to be used for color maps). HINTSTR describes a hint for the how the parameter ought to be presented to the user. CHANNELLIST is used to indicate the type of image channels that are stored in the database for this parameter.

2.1.5 Constraints

```
CONSTRAINTDICT ::= "constraints" : { CONSTRAINEDDICT+ }
CONSTRAINEDDICT ::= REF(PARAMETERNAME, global) : { VALIDLIST+ }
VALIDLIST ::= REF(PARAMETERNAME, global) : [ JSONVALUE+ ]
```

The constraints section describes the relationships between variables and how parameters are restricted by other parameter’s values. It is described further in section 2.2.5. Briefly, a constrained parameter will be ignored (not used to locate image data channels on storage) whenever any of the listed valid parameters, in VALIDLIST, have values outside of the listed set (i.e., it is not used in the production of inputs to outputs).

2.2 Chaplin Implementation

The content of a database is specified in a JSON format text file. The JSON file serves to enumerate the set of visualization samples, the relationships between them, and provides additional information used to interpret the content.

In any Cinema database, a number of different variables, or *parameters* are of fundamental importance. Each parameter represents one setting that the user might vary to inspect the data with. Concrete examples include simulation time step, camera position, visibility, filter settings and choice of data array to color by.

In contrast to simpler Cinema databases, a Chaplin database contains a **subset** of the full combinatorial set of all variable settings. The contents are a subset because invalid and uninteresting combinations are excluded. Examples of excluded combinations are the color choice for a non-visible object and isolevel setting for data that is not being isocontoured.

As always, additional information about the database that is not of the same combinatorial visualization-space nature is useful. Examples of these include version numbers and time-dependent camera specific information.

The semantics and meaning of the four required top-level elements of the JSON file follow.

2.2.1 Top level

A valid Chaplin database has four sections in a JSON file: `metadata`, `name_pattern`, `parameter_list` and `constraints`. Details of the implementation are described in the following sections.

2.2.2 Metadata

In addition to the sampled data as represented by the parameter space and constraint set, a cinema database contains additional data that describes the version of the database, supplemental information about cameras, and additional information about relationships between objects in the scene.

2.2.2.1 Database Type (Required) The *type* entry states whether this cinema store holds a *Simple* non-compositable database, or a compositable database. Non-compositible databases have a value of "parametric-image-stack". These are described in the Astaire Specification document. Chaplin's compositible databases have a value of "composite-image-stack".

```
"metadata": {
  "type": "composite-image-stack",
  ...
},
```

2.2.2.2 Database Version (Required) A *version* entry state the specific revision level within the type. The current revision level described in this document is 0.2.

```
"metadata": {
  ...
  "version": "0.2",
  ...
},
```

2.2.2.3 Store Type (Required) The *store_type* entry describes the lower level file format that the image channel data is kept in. It is possible to store the actual data under various containers. The one described in this document is implemented as named files within a filesystem directory as described in section 2.2.6. The corresponding code is "FS" that is based on named files and directories. "SFS" is reserved for a storage format that places files within a VTK's XML image data volume files.

```
"metadata": {
  ...
  "store_type": "FS",
  ...
},
```

2.2.2.4 Camera Model (Required) The camera model describes the camera behaviors encapsulated in the database. Cameras have different capabilities and constraints, and different supporting data is required for cameras in the `metadata` and `parameter` sections. Supported models are:

- `static` This is a camera that looks at a fixed position.
- `phi-theta` This corresponding to regularly sampled camera positions sitting on a ruled surface looking inward at a location in space. Requires `phi` and `theta` parameters to define the angular samplings of the grid. This is described in Section 2.2.4.9.
- `azimuth-elevation-roll` This is an inward facing camera. Requires a `pose` parameter to define the arbitrarily oriented normalized matrices that define the sampled camera directions. May optionally be supplemented with time varying offset location information about which the camera moves over time. This is described in the Section 2.2.4.9.
- `yaw-pitch-roll` This is an outward facing camera. Requires a `pose` parameter to define the arbitrarily oriented normalized matrices that define the sampled camera directions. may optionally be supplemented with time varying offset location information about which the camera moves over time. This is described in the section 2.2.4.9.

```
"metadata": {
  ...
  "camera_model": "azimuth-elevation-roll",
  ...
},
```

2.2.2.5 Endian (Required) We store depth and value image channels as raw zlib compressed 32-bit floating point files. The endian entry records the endianness that the floats are stored in so that the underlying stream of bytes can be correctly interpretted as numbers when read back in on a different computer.

```
"metadata": {
  ...
  "endian": "little",
  ...
},
```

2.2.2.6 Image Size (Required) We store depth and value image channels as raw zlib compressed 32-bit floating point files. The image size entry records the width and height of the data so that the underlying stream can be resized back to the original rectangular shape when it is read back in.

```
"metadata": {
  ...
  "image_size": [ 446, 955 ],
  ...
},
```

2.2.2.7 Value Mode (Optional) This is a record of the type for value image channels. When the entry is absent, or contains the value "1", it means that the value image channels were made using the approximative method. Value images are floating point images in which each pixel is a number between 0.0 and 1.0 (the values are normalized) and then stored in a zlib compressed ".Z" file.

Note that if the `value_mode` entry is present and contains the value "2" it means that the value images contain exact numerical quantities (the original data values) instead of normalized ones.

```
"metadata": {
  ...
  "value_mode": 2,
  ...
},
```

2.2.2.8 Camera Data (Optional) Supplemental camera information may be present to support the `azimuth-elevation-roll` and `yaw-pitch-roll` camera models. In either case cinema's parameter system provides an arbitrary list of camera directions. Those directions are placed at specific locations in space at each time step with the addition of offset information.

```
"metadata": {
  "camera_model": "azimuth-elevation-roll",
  "camera_eye": [[0.0, 0.0, 66.92],  /*corresponds to time 0.0e+00*/
                 [1.0, 0.0, 66.92],  /*corresponds to time 1.0e-04*/
                 [2.0, 0.0, 66.92]], /*corresponds to time 8.0e-04*/
  "camera_at": [[0.0, 0.0, 0.0], /*as above */
                [1.0, 0.0, 0.0],
                [2.0, 0.0, 0.0]]
  "camera_up": [[0.0, 1.0, 0.0], /*as above */
                [0.0, 1.0, 0.0],
                [0.0, 1.0, 0.0]],
  ...
}
```

Furthermore, two additional optional pieces of information server to precisely define the viewing frustum at each sample. These are the camera_nearfar and camera_angle's.

2.2.2.9 Pipeline (Optional) A pipeline entry directly encodes relationships between objects in the generating scene. It, in combination with the constraints and roles information is helpful for building up GUIs.

Here a graph of objects in the scene is explicitly recorded. This is not entirely redundant with the constraints system described in section 2.2.5 because abstractly the later variable centric view is more inclusive and concretely because there is not generally a one to one relationship between objects and parameters. When objects are present the pipeline information facilitates organizing the controllable options to present them to the user.

```
"metadata": {
  "pipeline": [
    { "children": [],
      "parents": [
        "2488"
      ],
      "id": "2687",
      "visibility": 1,
      "name": "Contour1" },
    { "children": [
        "2687"
      ],
      "parents": [
        "0"
      ],
      "id": "2488",
      "visibility": 1,
      "name": "Wavelet1" },
    ...
  ],
  ...,
}
```

2.2.3 Name Pattern

The `name_pattern` entry specifies the file format for RGB type color component image channels (inluding 'color', 'lut' and 'luminance') within the "FS" type database. The trailing file extension defines this type. In comparison

'depth' and 'value' images are alwayed stored as zlib compressed raw data in ".Z" files and thus do not need to be specified.

```
"name_pattern": "{dontcare}.png",
```

2.2.4 Parameters

A Chaplin database is a collection of results sampled by a set of visualization parameters. The collection of parameters that vary are stored in the `parameter_list` section of the json file. Each parameter in the set is fully described by one of the entries in the overall key-values pair. The values for each entry contain information that together fully define the parameter, including what is being sampled, the values it takes, and how the results are to be interpretted. In Chaplin everything from simulation time, to scene graph components, to operator controls, to colors are represented by annotated parameters.

The pieces of information are stored within a dictionary within each paremeter. The types of information are described next. Examples of the most important parameters typically found in a Cinema database follow them.

2.2.4.1 Label (Required) Each parameter will have a label. This is a human readable string that is intended to be presented to the user in a viewing application next to the controls over the parameter.

```
"parameter_list": {
  "parameter1": {
    "label": "slice plane offset",
    ...
```

2.2.4.2 Values (Required) Within the dictionary the most important element is a list of values that the parameter takes on within the database.

```
"parameter_list": {
  "parameter1": {
    ...
    "values": [ -0.5, -0.1, 0, 0.5, 42.0, 500.0 ],
    ...
```

2.2.4.3 Default Value (Optional) For viewing applications it is helpful to designate one entry within the list of values as the default one to picked when the user has not otherwise made a selection. If the default value is not written into the file, the first entry in the list should be used instead.

```
"parameter_list": {
  "parameter1": {
    ...
    "default": 42.0,
    ...
```

2.2.4.4 Role (Optional) Parameters may also have a *role*. The roll more fully describes the purpose of the parameter for generation and display purposes.

A role of "layer" indicates that this parameter defines what specific object or objects should be shown. There is typically only one parameter with the role in the database.

```
"parameter_list": {
  "vis": {
    "label": "objects in scene",
    "values": [
      "Slice1",
      "Superquadric1"
```

```
    ],
    "role": "layer"
    ...
  }
```

A role of "control" indicates that this parameter is one of a set that controls aspects of a specific object (i.e. filter settings) and of the objects (i.e. downstream filters) that depend on it as well.

```
"parameter_list": {
  "Slice1": {
    "label": "Slice1",
    "values": [ -0.5, 0.0, 0.5 ],
    "role": "control"
  },
```

A role of "field" indicates that this parameter represents the set of captured Image Channels for an object.

```
"parameter_list": {
  "colorObject1": {
    "label": "Color Inputs for Object1",
    "values": [
      "depth",
      "Pressure_0",
      "Temperature_0"
    ],
    "types": [
      "depth",
      "lut",
      "lut"
    ],
    "role": "field"
  },
```

For parameters with a role of "field" additional annotations (*types* and *valueRanges*) serve to better specify the charactersitics of the Image Channels.

2.2.4.5 Types – Image Channels (Optional) *types* is the mechanism by which the different channel types are distinquished. It lists the specific image component type corresponding to each entry in the field type parameter's value list. In the following example we have three image channels, the first called "depth" is a depth image. The second "L" is a luminance image. The last two channels "pressure_0" and "temperature_0" are value images.

```
"parameter_list": {
  "colorObject": {
    "label": "color for a meteo result",
    "values": [
      "depth",
      "L",
      "pressure_0",
      "temperature_0"
    ],
    "role": "field",
    "types": [
      "depth",
      "luminance",
      "value",
```

```
      "value"
    ],
    ...
  },
```

2.2.4.6 Value Range (Optional) For value type image channels it is useful to record the minimum and maximum recorded values for the corresponding data array or arrays. The value range annotation does this within a dictionary of names to minimum and maximum value pairs. For every "value" type image channel, there will be a corresponding entry in the value range dictionary. Continuing the above example we add storage for the range of both the pressure and temperature data.

```
"parameter_list": {
  "colorObject": {
    "label": "color for a meteo result",
    "values": [
      "depth",
      "L",
      "pressure_0",
      "temperature_0"
    ],
    "role": "field",
    "types": [
      "depth",
      "luminance",
      "value",
      "value"
    ],
    "valueRanges": {
      "pressure_0": [ 29.4, 29.8 ],
      "temperature_0": [33.0, 36.5 ]
    },
    ...
  },
```

2.2.4.7 Display Hint (Optional) Parameters may have a suggested widget *type*. The type is used as a hint for viewing applications to inform what type of GUI widget is most appropriate for this control. A type of "option" is best displayed with a combobox type widget that lets the user select many values at once. A type of "range" is best displayed with a menu, scalar bar, or slider, that lets the user select just one value at any give time time. A type of "hidden" should generally not be displayed to the user.

```
"parameter_list": {
  "vis": {
    "label": "vis",
    "role": "layer",
    "type": "option",
    ...
  "slice": {
    "label": "slice plane offset",
    "role": "control",
    "type": "range",
    ...
  "color": {
    "label": "color channels",
```

```
 "role": "field",
 "type": "hidden",
 ...
}
```

2.2.4.8 Parameter Example: Time Time varying data can be sampled at arbitrary points along the temporal domain.

```
"parameter_list": {
  "time": { "default": "0.000000e+00",
            "values": ["0.000000e+00", "1.000737e-04", "1.999051e-04"],
            "type": "range",
            "label": "simulation time" },
  ...
}
```

2.2.4.9 Parameter Example: Camera Positions In a traditional visualization setting the user can manipulate the camera arbitrarily. For cinema we discretize and organize the range of captured motions into of several camera motion classes. The camera model in use within a database is listed in the `camera_model` entry of the json files metadata section. See section 2.2.2.4 for specifics.

- `static` No addition information needed in this section. Static cameras do not require a corresponding parameter entry.
- `phi-theta` A model in which the discrete camera positions are described by the product of two parameters with angular positions. Additional information needed defines the `phi` and `theta` values. The camera "from" position varies over a set of regularly sampled angular positions centered around a chosen focal point. These cameras will have one or two corresponding parameters entries. In Cinema, `phi` is defined to be rotations around the vertical axis, going from -180 to 180 degrees, inclusive on the negative only. `theta` is defined to be rotations from south to north pole, ranging from -90 to 90 degrees inclusive.

  ```
  "parameter_list": {
    "phi": { "default": -180,
             "values": [-180, -150, -120, -90, -60, -30,
                        0,
                        30, 60, 90, 120, 150],
             "type": "range",
             "label": "phi" },
    "theta": { "default": -90,
               "values": [-90, -64, -38, -12, 13, 38, 63, 88],
               "type": "range",
               "label": "theta" },
    ...
  }
  ```

- `yaw-pitch-roll` and `azimuth-elevation-roll` both require a `pose` section. In these more general `pose` based cameras, we store complete camera reference frames from an arbitrary collection of viewpoints.

  These cameras require a `pose` parameter, which contains any number of 3x3 normalized camera reference frames. To keep the combination of camera positions over time and view directions manageable, we vary the camera's local coordinate frame consistently at every time step, but offset them from a different location at each time step. An example is shown below.

  ```
  "parameter_list": {
   "time": {
  ```

```
      "values": ["0.0e+00",
                 "1.0e-04",
                 "8.0e-04"],
       ... },
     "pose": {
       "values": [[[-1.0, 1.224e-16, 7.498e-33],
                   [0.0, 6.123e-17, -1.0],
                   [-1.224e-16, -1.0, -6.123e-17]],
                  [[-1.0, 1.109e-16, 5.175e-17],
                   [0.0, 0.422, -0.906],
                   [-1.224e-16, -0.906, -0.422]],
                  ... ],
       ...
       },
     ...
    }
```

2.2.4.10 Parameter Example: Objects Different items may be displayed in the scene. During image generation it is important that only one object be displayed at a time so that we can capture its depth, color, value and other image channels in isolation. From the viewing application the end user selects zero or more entries from the same parameter to show any number of objects together at the same time. Visibility parameters have a *role* annotation of "layer".

```
"parameter_list": {
  "vis": {
    "values": [
      "Contour1",
      "Wavelet1",
      ...
    ],
    "role": "layer",
    "type": "option",
    ...
  }
}
```

2.2.4.11 Parameter Example: Operators Zero or more operators, such as clipping plane and isocontour samples along with their respective ranges are represented by parameters. Operators, like other parameters are cumulative. For each operator, the value is applied and the resulting scene is recorded. Operator parameters have a *role* annotation of "control".

```
"parameter_list": {
   "Contour1": {
     "values": [37.3531, 97.2221, 157.091, 216.96, 276.829],
     "role": "control",
     ...
   },
   ...
}
```

2.2.4.12 Parameter Example: Output Image Channels In a Cinema workflow, the application producing a Cinema Database creates a set of image channels for each sample in the parameter space. Viewing applications use these channels to draw objects and to color them dynamically dependent on the user's choices for solid color or colormapped value arrays. Recall that "types" and "valueRanges" annotations are important for interpreting the output of each possible value for a color control.

The types entries describe the specific image channels type. There are several possible image channels.

- **Required**. One of the following image channels is required. Both **cannot** be present.
  - **RBG**. An image containing colored pixel information. This can be one of two types:
    - **Color** channel. This encodes a standard RGB value for each pixel - the result of rendering the viewpoint from the camera. Color images can not be dynamically color mapped.
    - **LUT (Lookup Table)** channel. This contains pixels that have been colored by applying a colormap to a single variable.
  - **Value** channel. This encodes an arbitrary array value associated with the data that is visualized at each pixel. Global ranges for each array, i.e. the min and max value for a value over all time steps and parameter settings, are recorded in the meta data file. In Chaplin, value images are stored as floating point image channels, where the value is either normalized between 0.0 and 1.0 or kept unmodified, and stored in a ".Z" file. This is described more fully in section 2.2.2.7.
- **Optional**. These channels enable other types of operations.
  - **Depth** channel. This encodes a depth value for every pixel, relative to the camera. Required for compositing. In Chaplin, depth images are stored as floating point image channels, ranging from 0 for the near plane to 255 for the far plane, and kept in zlib compressed ".Z" files. As with value images the word size is 32bit. The endianness and image dimensions are recorded in the metadata section's endian and image_size entries.
  - **Luminance** channel. This encodes a rendered shading brightness for each pixel. Required for lighting to be included in the final rendering. In Chaplin, luminance images are stored in RGB images where the R component contains the ambient gray value, G contains the diffuse, and B contains the specular component. Of these, currently only the diffuse component is used. All components range from 0 to 255.

An example of this parameter list section follows. Here we have all possibilities for the image channels of an object named "Contour1". The image channels consist of depth, luminance, and a value for a scalar quantity called "RTData". The RTData array varies over the entire simulation and for all objects between 37 and 277.

```
"parameter_list": {
  "colorContour1": {
    "values": ["depth", "luminance", "RTData_0"],
    "types": ["depth", "luminance", "value"],
    "valueRanges": {"RTData_0": [37.3531, 276.829]},
    "role": "field"
  },
  ...
 }
```

2.2.5 Constraints

Constraints are relationships between parameters. These are included in the specification because a visualization is sometimes designed so that parameters are constrained by and dependent on others. For example, a visualization might include two unrelated objects with different sets of arrays on each object. In such a scene, the choice of what array to use to apply colors then depends on what object is displayed.

In cinema we store this information in a constraints entry in the json file. It consists of a list of parameters, and for each parameter in the list there is a sub list of one or more parameters and associated parameter values, that the containing parameter's presence depends upon. See section 2.1.5.

A specific example of this with an object's color inputs are only output when the vis layer parameter shows that the object is selected. Likewise for the color of the contour object.

```
"constraints": {
   "colorContour1": {
```

```
      "vis": [
        "Contour1"
      ]
    },
    "colorObject1": {
      "vis": [
        "Object1"
      ]
    },
    ...
```

A more complicated example which adds in two levels of dependencies is as follows. Here we represent a the three element data pipeline that might create a visualization. In the pipeline a Sphere object is fed into a Slice filter and the Slice filter is fed into a Clip filter.

```
 "constraints": {
    "colorSphere1": {
      "vis": [
        "Sphere1"
      ]
    },
    "colorSlice1": {
      "vis": [
        "Slice1"
      ]
    },
    "colorClip1": {
      "vis": [
        "Clip1"
      ]
    },
    "Sphere1": {
      "vis": [
        "Clip1",
        "Sphere1",
        "Slice1"
      ]
    },
    "Slice1": {
      "vis": [
        "Clip1",
        "Slice1"
      ]
    }
    "Clip1": {
      "vis": [
        "Clip1"
      ]
    },
    ...
```

To begin with there is the color relationship information described above. Besides that there is information about the parameters for objects along the pipeline.

Starting at the end of the pipeline the clipping plane value is sampled whenever the clipped data is shown. Going one step up the pipeline the slice plane's value must be varied whenever either the slice or the clip output are visible.

This is necessary because the slice filter is an input to the clip filter so its value affects the appearence of the clipped result. Likewise hypothetical parameters of the sphere object itself (the radius for example) have to vary whenever any of the three objects were visible. Remember also that if the visibility parameter selects something other than one of these three objects, then none of these parameters will be exercised.

2.2.6 Storage Structure

A low level data storage layer beneath the metadata layer holds individual results in the form of image channels saved as images. For example, a depth image channel for one particular object drawn from one particular viewpoint at one particular time step. For each valid combination of parameter settings, their will be one file.

The storage layer organizes results into numbered directories within a tree hierarchy. Each directory corresponds to a variable and numbered contents correspond to different values for the variable. The mapping of numbers to data values comes from the list of values for each parameter in the metadata. In the example below we have a data base with two time steps, each with two camera poses and two objects. The first object has three color components, the second has only two.

```
results/
  info.json
  pose=0/
    time=0/
      vis=0/
        color=0.Z
        color=1.png
        color=2.Z
      vis=1/
        color=0.Z
        color=1.png
    time=0/
      vis=0/
        color=0.Z
        color=1.png
        color=2.Z
      vis=1/
        color=0.Z
        color=1.png
  pose=1/
    time=0/
      vis=0/
        color=0.Z
        color=1.png
        color=2.Z
      vis=1/
        color=0.Z
        color=1.png
    time=1/
      vis=0/
        color=0.Z
        color=1.png
        color=2.Z
      vis=1/
        color=0.Z
        color=1.png
```

2.2.6.1 Constructing a Path to Output Files The scheme we use to consistently arrange files into the file system is to sort alphabetically the parameter names, and then lay out the data in sub-directories in a dependency following order.

In particular parameters with no dependencies go first, and then as dependencies are satisfied, dependent parameters follow, always within alphabetical order at each dependency level.

A pseudocode summary of the standard implementation's _get_filename() method of the file_store class is as follows.

```
keys = [sort(parameters_names)]
ordered = []
while len(keys):
    k = keys.pop(0)
    parents = k.getdependees()
    ready = all parents already in ordered
    if not ready:
        keys.append(k) # try back later
    else:
        ordered.append(k) # this one is next
for k in ordered:
    index = offset for parameter[k]'s value
    filename = filename + "/" + key + "=" + index
```

With parameters:

```
"b_param": {"values": [1,-2]}
"a_param": {"values": ["a","b"]}
"c_param": {"values": [42.0, 3.14159265358979323846264338327950288419 71],
            "types": ["depth","rgb"],
            "role": "field" }
"d_param": {"values": ["I","II","III"],
            "types": ["luminance","value","depth"],
            "role": "field"}
```

and constraints:

```
constraints": {
   "c_param": {
      "b_param": [1]
   },
   "d_param": {
      "b_param": [-2],
   }
}
```

and filename_pattern "dontcare.tiff": the resulting layout would be:

```
a_param=0/b_param=0/c_param=0.Z
a_param=0/b_param=0/c_param=1.tiff
a_param=0/b_param=1/d_param=0.tiff
a_param=0/b_param=1/d_param=1.Z
a_param=0/b_param=1/d_param=2.Z
a_param=1/b_param=0/c_param=0.Z
a_param=1/b_param=0/c_param=1.tiff
a_param=1/b_param=1/d_param=0.tiff
a_param=1/b_param=1/d_param=1.Z
a_param=1/b_param=1/d_param=2.Z
```

Because "a_param" comes before "b_param" alphabetically. "c_param" and "d_param" follow their shared dependee "b_param" and would do so even if they were renamed to something less than "b_param".

Adding another dependency level would change the layout with the new parameter following its own dependee.

```
"aa_param": {"values": [10, 11]}
```

additional constraint:

```
   "aa_param": {
      "d_param": ["I","III"]
   }
```

```
  a_param=0/b_param=0/c_param=0.Z
  a_param=0/b_param=0/c_param=1.tiff
  a_param=0/b_param=1/d_param=0/aa_param=0.tiff
  a_param=0/b_param=1/d_param=0/aa_param=1.tiff
  a_param=0/b_param=1/d_param=1.Z
  a_param=0/b_param=1/d_param=2/aa_param=0.Z
  a_param=0/b_param=1/d_param=2/aa_param=1.Z
  a_param=1/b_param=0/c_param=0.Z
  a_param=1/b_param=0/c_param=1.tiff
  a_param=1/b_param=1/d_param=0/aa_param=0.tiff
  a_param=1/b_param=1/d_param=0/aa_param=1.tiff
  a_param=1/b_param=1/d_param=1.Z
  a_param=1/b_param=1/d_param=2/aa_param_0.Z
  a_param=1/b_param=1/d_param=2/aa_param_1.Z
```

2.2.6.2 File Type There are a variety of content types in a Chaplin store: depth and value images are files of floating point numbers, while luminance and color images are files of RGB tuples. The content type is determined from a "field" or color parameter's "types" entry. For each value that one of these parameters takes, the corresponding "types" entry indicates if it is 'rgb', 'depth', 'value', or 'luminance' type content.

```
 "parameter_list": {
   "colorContour1": { "values": ["val1", "val2", "val3"],
                      "types": ["depth", "luminance", "value"],
                      "valueRanges": {"RTData_0": [37.3531, 276.829]},
                      "role": "field" },
   ...
  }
```

In the example above, whenever the colorContour1 parameter takes on "val1", the content is a depth image. When it is "val2" the content is a luminance image. When it is "val3" it is a value image. In Chaplin depth and value images are stored in zlib compressed ".Z" files. Everything else is stored in a standard image format. The choice of a specific image format, ".png", ".tiff", ".jpeg" and the like is made based on the trailing file extension from the *name_pattern* entry.

2.2.6.3 File Name Parameter settings are encoded into file and directory names. We use a "key=index" scheme for all parameters where the key is the parameter name and the index is the 0 based index into the values list to a specific value. In the example above when colorContour1 parameter takes on "val1" the result will appear in a file or directory named "colorContour1=0". Using indices instead of values avoids a number of conversion problems involving decimal precision and also the fact that non-scalar values do not map well onto file system naming rules.

2.2.7 Single Database Example

2.2.7.1 JSON file This example is based on the above JSON schema outline.


```
{
  "metadata": {
    "type": "composite-image-stack",
    "version": "0.2",
    "store_type": "FS",
    "image_size": [ 446, 955 ],
    "endian": "little",
    "value_mode": 2,
    "camera_model": "azimuth-elevation-roll",
    "camera_eye": [
      [ -0.023820051408253964, 1.3749762463487758, 2.409604270242429 ]
    ],
    "camera_at": [
      [ 0.0, 0.0, 0.0 ]
    ],
    "camera_up": [
      [ 0.003238554846559421, 0.8685535085368621, -0.49558482076311905 ]
    ],
    "camera_nearfar": [
      [ 0.007973907729484015, 7.973907729484015 ]
    ],
    "camera_angle": [
      30.0
    ],
    "pipeline": [
      {
        "children": [],
        "parents": [ "3970" ],
        "id": "4254",
        "visibility": 1,
        "name": "Slice1"
      },
      {
        "children": [ "4254" ],
        "parents": [ "0" ],
        "id": "3970",
        "visibility": 1,
        "name": "Superquadric1"
      }
    ]
  },
  "name_pattern": "{pose}.png",
  "parameter_list": {
    "colorSlice1": {
      "types": [
        "depth",
        "luminance",
        "value",
        "value"
      ],
      "type": "hidden",
```

```
  "role": "field",
  "valueRanges": {
    "TextureCoords_0": [ 0.0, 1.0 ],
    "TextureCoords_1": [ -0.0017080907709896564, 1.0 ]
  },
  "values": [
    "depth",
    "luminance",
    "TextureCoords_1",
    "TextureCoords_0"
  ],
  "label": "colorSlice1",
  "default": "TextureCoords_0"
},
"Slice1": {
  "label": "Slice1",
  "type": "hidden",
  "role": "control",
  "values": [ -0.5, 0, 0.5 ],
  "default": -0.5
},
"pose": {
  "label": "pose",
  "type": "range",
  "values": [
    [[1.0, 0.0, 7.498798913309288e-33],
     [7.498798913309288e-33, -6.123233995736766e-17, -1.0],
     [0.0, 1.0, -6.123233995736766e-17]],
    [[1.0, 1.2246467991473532e-16, 1.2246467991473532e-16],
     [1.2246467991473532e-16, -1.0, 0.0],
     [1.2246467991473532e-16, 1.4997597826618576e-32, -1.0]],
    [[-1.0, -1.2246467991473532e-16, 0.0],
     [1.2246467991473532e-16, -1.0, 0.0],
     [0.0, 0.0, 1.0]],
    [[1.0, 2.4492935982947064e-16, 7.498798913309288e-33],
     [7.498798913309288e-33, -6.123233995736766e-17, 1.0],
     [2.4492935982947064e-16, -1.0, -6.123233995736766e-17]],
    [[-1.0, 1.2246467991473532e-16, 7.498798913309288e-33],
     [0.0, 6.123233995736766e-17, -1.0],
     [-1.2246467991473532e-16, -1.0, -6.123233995736766e-17]],
    [[-1.0, 0.0, 1.2246467991473532e-16],
     [0.0, 1.0, 0.0],
     [-1.2246467991473532e-16, 0.0, -1.0]],
    [[1.0, 0.0, 0.0],
     [0.0, 1.0, 0.0],
     [0.0, 0.0, 1.0]],
    [[-1.0, -1.2246467991473532e-16, 7.498798913309288e-33],
     [0.0, 6.123233995736766e-17, 1.0],
     [-1.2246467991473532e-16, 1.0, -6.123233995736766e-17]]],
  "default": [
     [1.0, 0.0, 0.0],
     [0.0, 1.0, 0.0],
     [0.0, 0.0, 1.0]
   ]
```

```
  },
  "colorSuperquadric1": {
    "types": [
      "depth",
      "luminance",
      "value",
      "value"
    ],
    "type": "hidden",
    "role": "field",
    "valueRanges": {
      "TextureCoords_0": [ 0.0, 1.0 ],
      "TextureCoords_1": [-0.0017080907709896564, 1.0 ]
    },
    "values": [
      "depth",
      "luminance",
      "TextureCoords_1",
      "TextureCoords_0"
    ],
    "label": "colorSuperquadric1",
    "default": "TextureCoords_0"
  },
  "vis": {
    "label": "vis",
    "type": "option",
    "role": "layer",
    "values": [
      "Slice1",
      "Superquadric1"
    ],
    "default": "Slice1"
  }
},
"constraints": {
  "colorSlice1": {
    "vis": [
      "Slice1"
    ]
  },
  "Slice1": {
    "vis": [
      "Slice1"
    ]
  },
  "colorSuperquadric1": {
    "vis": [
      "Superquadric1"
    ]
  },
  "Superquadric1": {
    "vis": [
      "Slice1",
      "Superquadric1"
```

```
      ]
    }
  }
}
```

2.2.7.2 Associated File Structure Using the above example, the following paths would be expected for the resulting image channels:

```
./info.json
./pose=0/vis=0/Slice1=0/colorSlice1=0.Z
./pose=0/vis=0/Slice1=0/colorSlice1=1.png
./pose=0/vis=0/Slice1=0/colorSlice1=2.Z
./pose=0/vis=0/Slice1=0/colorSlice1=3.Z
./pose=0/vis=0/Slice1=1/colorSlice1=0.Z
./pose=0/vis=0/Slice1=1/colorSlice1=1.png
./pose=0/vis=0/Slice1=1/colorSlice1=2.Z
./pose=0/vis=0/Slice1=1/colorSlice1=3.Z
./pose=0/vis=0/Slice1=2/colorSlice1=0.Z
./pose=0/vis=0/Slice1=2/colorSlice1=1.png
./pose=0/vis=0/Slice1=2/colorSlice1=2.Z
./pose=0/vis=0/Slice1=2/colorSlice1=3.Z
./pose=0/vis=1/colorSuperquadric1=0.Z
./pose=0/vis=1/colorSuperquadric1=1.png
./pose=0/vis=1/colorSuperquadric1=2.Z
./pose=0/vis=1/colorSuperquadric1=3.Z
./pose=1/vis=0/Slice1=0/colorSlice1=0.Z
./pose=1/vis=0/Slice1=0/colorSlice1=1.png
...
./pose=6/vis=1/colorSuperquadric1=2.Z
./pose=6/vis=1/colorSuperquadric1=3.Z
./pose=7/vis=0/Slice1=0/colorSlice1=0.Z
./pose=7/vis=0/Slice1=0/colorSlice1=1.png
./pose=7/vis=0/Slice1=0/colorSlice1=2.Z
./pose=7/vis=0/Slice1=0/colorSlice1=3.Z
./pose=7/vis=0/Slice1=1/colorSlice1=0.Z
./pose=7/vis=0/Slice1=1/colorSlice1=1.png
./pose=7/vis=0/Slice1=1/colorSlice1=2.Z
./pose=7/vis=0/Slice1=1/colorSlice1=3.Z
./pose=7/vis=0/Slice1=2/colorSlice1=0.Z
./pose=7/vis=0/Slice1=2/colorSlice1=1.png
./pose=7/vis=0/Slice1=2/colorSlice1=2.Z
./pose=7/vis=0/Slice1=2/colorSlice1=3.Z
./pose=7/vis=1/colorSuperquadric1=0.Z
./pose=7/vis=1/colorSuperquadric1=1.png
./pose=7/vis=1/colorSuperquadric1=2.Z
./pose=7/vis=1/colorSuperquadric1=3.Z
```

2.2.8 Combining Multiple Databases

It is often convenient to group cinema data bases together. This is useful, for example in the case of multi-view applications. In this case, independent databases can be collected as siblings in a higher level directory, and another json file which lists the directories can provide information about the set.

An example directory hierarchy containing the above and corresponding json file follow.

```
collection.cdb/
  info.json
  results/
    info.json /*etc as above*/
  moreresults/
    info.json /*similar to above*/
```

The top level info.json here would be as follows.

```
{
  "metadata": {
    "type": "workbench"
  },
  "runs": [
    { "title": "a view of some data",
      "description": "interesting results ...",
      "path": "results" },
    { "title": "a different view",
      "description": "even more interesting results ...",
      "path": "moreresults" }
  ]
}
```

3 Notes on Previous Specifications

The Cinema project updates specifications as new capabilities are integrated into the overall design. Specifications are named in alphabetical order.

- *Astaire* is currently supported. The specification is available on the Cinema website `www.cinemascience.org`.
- *Bacall* has been deprecated.
- *Chaplin* is this specification.

4 Contact Information

There are a lot of ways to get in touch with us. First, you can contact the authors of this document. In addition:

- Developer mailing list: `cinema-dev@lanl.gov`.
- General Cinema mailing list: `cinema@lanl.gov`.
- Cinema website: `cinemascience.org`. This includes documentation, specifications, examples and applications (viewers, etc.)
- Cinema website: `cinemaviewer.org`. This site shows several examples of browser-based cinema viewers with various scientific databases.

References

[1] James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H. Rogers, and Mark Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 424–434, Piscataway, NJ, USA, 2014. IEEE Press.