Cinema Database Specification

Chaplin Release

LA-UR-15-20572

by

David DeMarle (Kitware, Inc)
David Rogers (LANL)
John Patchett (LANL)
Berk Gevevi (Kitware)

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

1 Introduction

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 scientists knows about in advance. We present 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.

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 is a collection of data that supports this image-based approach to interactive data exploration. It is a set of images and associated metadata, and is defined and an example given in the following sections.

1.1 Public Comment

We invite public comment on this specification. Please send comments to cinemascience.org.

1.2 Use Cases

A Cinema Database supports the following three use cases:

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.

1.3 Cinema is Implementation Agnostic

The Cinema Database is implementation agnostic. This database specification separates the metadata description of a set of images from the implementation of how these images are generated and stored. The specification defines a database of URIs that maps metadata to specific data products which can then be accessed and used indepependently of the specific details of the low level data storage structures. In particular, if the images for a specific instance of a database are stored on disk, the design of the directory structure, metadata files, and image filenames on disk is entirely up to the person writing the data as long as a mapping from the database API to the files in question can be described.

1.4 Overview

A cinema database is a set of precomputed visualization samples that can be automatically generated and later queried and interactively viewed. This document describes release v1.0 of the Cinema *Chaplin* Database, in which image constituents are stored instead of 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. A separate document [2] specifies a *Simple* Cinema Database, which contain a fixed set of pre-rendered images for a fixed set of combinations of visualization operations.

In the rest of this document we describe the metadata structure, which describes the overall contents of the cinema data base and the relationships between entities within, and give as an example one low level structures based on files and numbered directories.

The metadata structure for the database is held in a json format text file. In the text file a number of different variables, or *parameters* are of fundamental interest. Each parameter will be associated with one or more values. The parameters represent items that might be changed in a traditional post processing visualization session. Concrete examples include simulation time step, camera position, object visibility, filter settings and choice of colormapped data array. Besides values, parameters can be annotated with additional information.

In *Chaplin*, the database will contain a subset of the full combinatorial set of all variable/value pairs. The contents are a subset because invalid and uninteresting combinations are excluded. An example of an excluded combination is the color choice parameter for an object when the visibility parameter for the object is set to off. The exclusion list takes the form of a *constraints* list.

A low level data storage layer beneath the metadata layer holds individual results in the form of raster images. For example, a depth raster for one particular object drawn from one particular viewpoint at one particular timestep.

1.5 Sample low level file layout

The example storage layer we describe here organizes these results in 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 timesteps, 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.npz
        color=1.png
        color=2.npz
      vis=1/
        color=0.npz
        color=1.png
    time=0/
      vis=0/
        color=0.npz
        color=1.png
        color=2.npz
      vis=1/
        color=0.npz
        color=1.png
  pose=1/
    time=0/
      vis=0/
        color=0.npz
        color=1.png
        color=2.npz
      vis=1/
        color=0.npz
        color=1.png
    time=1/
      vis=0/
        color=0.npz
        color=1.png
        color=2.npz
      vis=1/
        color=0.npz
        color=1.png
```

The directory hierarchy follows an order that satisfies the constraints between variables. Viewing applications that use the cinema python reference library should not depend on any particular file order but should instead use the

reference library's database like API search for, insert and retrieve contents based on parameter names and values. For applications that can not use the refence API, the exact file structure mapping is described in section 3.

1.5.1 Multiple Views

Note that it is 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" }
  ]
}
```

We now return to a more thorough discussion of the metadata content that provides structure for an isolated cinema data base.

2 The Cinema *Chaplin* Specification

A *Chaplin* database is a collection of results sampled by a set of visualization parameters. Each parameter is described with an entry in "parameter_list" section of the json file. Parameters will have a a set of *values*, a *default* value from among the values, a *name* and a suggested *label*. Parameters may also have a suggested widget *type*, a *role* for the parameter that describes its type more fully for generation purposes. Parameters may contain additional annotations especially in the case of color/image component information.

Additional information about the database as a whole is kept in the metadata section. The information in the metadata section is separate from the parameters as it is not of the same combinatorial visualization-space nature.

Examples of typical parameters in a cinema database are:

- **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": "time" },
  ...
}
```

- **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 as described in section 2.1.

```
"metadata": {
  "camera_model": "phi-theta",
  ...
},
"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" },
  ...
}
```

- **Objects**. Different items may be displayed in the scene. A visibility parameter controls which items are visible at any given time. For a composite database the generator will ensure that only one item is visible in any given raster while the images are being produced. The viewer uses the same parameter to allow multiple items to be visible and depth composites multiple rasters together to produce the effect. Visibility parameters have a *role* annotation of "layer".

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

- Zero or more **operators**, such as clipping plane and isocontour samples along with their respective ranges. Each result is sampled and saved in isolation from all others with nothing else visible in the scene. In a viewing application, the user can choose any number of these samples and see them rendered together with correct occlusion culling. Operator parameters have a *role* annotation of "control".

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

- **Color** components for deferred rendering as described in 2.2. Color parameters have a *role* annotation of "field".

2.1 Cameras

We discretize the infinite set of camera possibilities by dividing first into different camera types. The camera type is given in the "camera_model" entry of the metadata.

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

In a *static* camera the position and orientation is fixed an unchanging. Static cameras do not require a corresponding parameter entry.

In a *phi-theta* spherical type camera the camera "from" position varies over a set of regularly sampled angular positions centered around a chosen focal point. Phi-Theta 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.

In the more general *pose* based cameras, new to Cinema v1.0, we store complete camera reference frames from an arbitrary collection of viewpoints. Pose based camera are allowed to move over time and track objects in the scene.

Pose cameras consist of a parameter named pose, 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. Two simple variations on this theme move the coordinate frame about specific from points *yaw-pitch-roll* or outward-facing, and around specific lookat points *azimuth-elevation-roll* or inward-facing. Inward facing camera are an improvement upon the earlier phi-theta camera type.

For example a pose based, inward facing, object tracking camera for a dataset with three timesteps might contain,

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

```
},
```

Besides the entries above additional pose camera metadata includes camera near and far planes and camera field of view settings. The complete set is optional, but when present it allows for the calculation of the exact coordiantes of every pixel in the database.

2.2 Colors

In a Cinema workflow, the application producing a Cinema Database creates a set of image constituents for each sample in the parameter space. Viewing applications use these consituents to draw objects and to color them dynamically dependent on the user's choices for solid color or colormapped value arrays. We describe the set of image consituents for a given object with a color parameter. These image constituents can be:

- **Depth** image. This encodes a depth value for every pixel, relative to the camera. Required for compositing. In *Chaplin*, depth images are stored as floating point rasters, ranging from 0 for the near plane to 255 for the far plane, and kept in numpy ".npz" files.
- **Luminance** image. 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.
- **Color** image. 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.
- **Value** image. This encodes an arbitrary array value associated with the data that is visualized at each pixel. Global ranges for each array, ie the min and max value for a value over all timesteps and parameter settings, are recorded in the meta data file. In *Chaplin*, value images are stored as floating point rasters, where the value is either normalized between 0.0 and 1.0 or kept unmodified, and stored in a numpy ".npz" file. This is described more fully in section 2.3.2.

An example of a color specification is as follows:

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

Here we have all possibilities for the color constituents of an object named "Contour1". The color constituents consist of a depth raster, a luminance raster, and a value raster for a scalar quantity called "RTData". The RTData array varies over the entire simulation and for all objects between 37 and 277.

2.3 Metadata

In addition to the sampled data as represented by the parameter space, a cinema database contains additional data that describes the version of the database, the relationships between parameters and information about how they are stored.

2.3.1 Version Information

Version information allows cinema based applications to allow for backward compatibility. The *type* entry states whether this cinema store holds a *Simple* non-compositable database, or a compositable database as described in this document. Furthermore a *version* entry state the specific revision level of both types. The *store_type* entry descibes the lower level file format that the raster and other data is kept in. This should be "FS" for the reference implementation described in section 3 that is based on named files and directories.

```
 "metadata": {
   "type": "composite-image-stack",
   "store_type": "FS",
   "version": "0.1",
   ...
 },
```

2.3.2 Value Mode

Another piece of information in the metadata section for composite type databases is the *value_mode* entry. This is a record of the type for value raster images. When the entry is absent, or contains the value "1", it means that the value rasters were made using the approximative method that predates version *Chaplin*. Prior to *Chaplin*, value raster images were approximated by scaling numerical quantities to the range of 0 to 2^{24}, outputting the results into standard RGB image files, and then using the range information for the array to recover numbers that were close to the original values. In *Chaplin* these value rasters are automatically converted to floating point rasters in which each pixel is a number between 0.0 and 1.0 and then stored in a numpy ".npz" file.

If the entry is present and contains the value "2" it means that the value rasters were created directly and contain exact numerical quantities instead of normalized ones.

2.3.3 Constraints

Relationships between parameters are included because in a visualization session some parameters are constrained by others. A scene with two unrelated objects in it could have entirely different sets of arrays for each one of them and thus the choice of colors to colormap by depends on the choice of object displayed. Relationships turn otherwise order independent options into a graph of parent child relationships.

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. In general:

```
<parameter_name1> : {
    <parameter_name2> : [list of permissible values that enable parameter 1],
    <parameter_name3> : [list of permissible values that enable parameter 1],
}
```

A specific example being:

```
 "constraints": {
    "Contour1": {
      "vis": [
        "Contour1"
      ]
    },
    "colorContour1": {
      "vis": [
        "Contour1"
      ]
    },
    "colorWavelet1": {
      "vis": [
        "Wavelet1"
      ]
    },
    "Wavelet1": {
      "vis": [
        "Contour1",
        "Wavelet1"
```

```
      ]
    }
```

Two other optional pieces of information may be found in the json file.

2.3.4 Pipeline

A pipeline entry encodes the relationships between objects in the scene. It, in combination with the constraints and roles information is helpful for building up GUIs.

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

2.3.5 Name Pattern

The last entry that may be of use is the *name_pattern* entry. This is a legacy of the simple database format where it is used to fully describe the directory structure of the "FS" type database. In the composite database, the constraints and hierarchical nature of the data make it impractical to fully describe the layout with a regular expression. It is used only to specify the default raster file format for RGB type color component rasters. The trailing file extension determines this.

3 Low level file Structure

This information is provided for those who wish to bypass the cinema_python reference libray but still use or create compatible file stores. In this situtation it is necessary to know exactly how to map parameter value combinations to file names. There are three important aspects, one is to derive the file type and file format extension, another is to derive a specific filename for a particular value, and the last is to derive a consistent directory path given a collection of values.

Note that the exact format of the store has changed over time as Cinema has evolved. The reference library uses a cinema store's included version markers to provide backwards compatibility. We here describe the latest version information as of Cinema Chaplin version 0.1. For further details consult the FileStore class's _get_filename() method in the reference library.

3.1 File Type

Unlike in Astaire, there are a variety of content types in a *Chaplin* store: depth and value images are rasters of floating point numbers, while luminance and color images are rasters 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', 'luminance', or 'normals' 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 numpy compressed array ".npz" 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.

3.2 File Name

In Chaplin, 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.

3.3 Directory Path

The scheme we use to consistently arrange files into the filesystem is to sort alphabetically the parameter names, and then lay out the data in subdirectories 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.

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.npz
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.npz
a_param=0/b_param=1/d_param=2.npz
a_param=1/b_param=0/c_param=0.npz
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.npz
a_param=1/b_param=1/d_param=2.npz
```

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

4 Example

This example is based on the above JSON schema outline.

```
{
  "parameter_list": {
    "vis": {
      "label": "vis",
      "type": "option",
      "role": "layer",
      "values": [
        "Slice1",
        "Superquadric1"
      ],
      "default": "Slice1"
    },
```

```
"colorSuperquadric1": {
  "types": [
    "depth",
    "luminance",
    "value",
    "value"
  ],
  "type": "hidden",
  "role": "field",
  "valueRanges": {
    "TextureCoords_0": [
      0.0,
      1.0
    ],
    "TextureCoords_1": [
      -0.00033474931842647493,
      1.0
    ]
  },
  "values": [
    "depth",
    "luminance",
    "TextureCoords_1",
    "TextureCoords_0"
  ],
  "label": "colorSuperquadric1",
  "default": "TextureCoords_0"
},
"Slice1": {
  "label": "Slice1",
  "type": "hidden",
  "role": "control",
  "values": [
    -0.5,
    0,
    0.5
  ],
  "default": -0.5
},
"colorSlice1": {
  "types": [
    "depth",
    "luminance",
    "value",
    "value"
  ],
  "type": "hidden",
  "role": "field",
  "valueRanges": {
    "TextureCoords_0": [
      0.0,
      1.0
    ],
    "TextureCoords_1": [
```

```
      0.0,
      1.0
    ]
  },
  "values": [
    "depth",
    "luminance",
    "TextureCoords_1",
    "TextureCoords_0"
  ],
  "label": "colorSlice1",
  "default": "TextureCoords_0"
},
"pose": {
  "label": "pose",
  "type": "range",
  "values": [
    [
      [ 1.0, 0.0, 7.498-33 ],
      [ 7.498e-33, -6.123e-17, -1.0 ],
      [ 0.0, 1.0, -6.123e-17 ]
    ],
    [
      [ 1.0, 1.224e-16, 1.224e-16 ],
      [ 1.224 -1.0, 0.0 ],
      [ 1.224e-16, 1.499e-32, -1.0 ]
    ],
    [
      [ -1.0, -1.224e-16, 0.0 ],
      [ 1.224 -1.0, 0.0 ],
      [ 0.0, 0.0, 1.0 ]
    ],
    [
      [ 1.0, 2.449e-16, 7.498e-33 ],
      [ 7.498e-33, -6.123e-17, 1.0 ],
      [ 2.449 -1.0, -6.123e-17 ]
    ],
    [
      [ -1.0, 1.224e-16, 7.498e-33 ],
      [ 0.0, 6.123e-17, -1.0 ],
      [ -1.22 -1.0, -6.123e-17 ]
    ],
    [
      [ -1.0 0.0, 1.224e-16 ],
      [ 0.0, 1.0, 0.0 ],
      [ -1.2 0.0, -1.0 ]
    ],
    [
      [ 1.0, 0.0, 0.0 ],
      [ 0.0, 1.0, 0.0 ],
      [ 0.0, 0.0, 1.0 ]
    ],
    [
      [ -1.0, -1.224e-16, 7.498e-33 ],
```

```
        [ 0.0, 6.123e-17, 1.0 ],
        [ -1.2 1.0, -6.123e-17 ]
      ]
    ],
    "default": [
      [ 1.0, 0.0, 0.0,
      [ 0.0, 1.0, 0.0,
      [ 0.0, 0.0, 1.0
    ]
  },
},
"constraints": {
  "Superquadric1": {
    "vis": [
      "Slice1",
      "Superquadric1"
    ]
  },
  "colorSuperquadric1": {
    "vis": [
      "Superquadric1"
    ]
  },
  "Slice1": {
    "vis": [
      "Slice1"
    ]
  },
  "colorSlice1": {
    "vis": [
      "Slice1"
    ]
  }
}
"metadata": {
  "version": "0.1",
  "store_type": "FS",
  "value_mode": 2,
  "type": "composite-image-stack",
  "camera_model": "azimuth-elevation-roll",
  "camera_eye": [
    [
      -0.021166744801978096,
      1.7656863321901966,
      2.174890404450956
    ]
  ],
  "camera_at": [
    [
      0.0,
      0.0,
      0.0
    ]
  ],
```

```
    "camera_up": [
      [
        -0.021825711219248648,
        0.776071754521858,
        -0.6302668245776063
      ]
    ],
    "camera_nearfar": [
      [
        0.008157218972537116,
        8.157218972537116
      ]
    ],
    "camera_angle": [
      30.0
    ],
    "pipeline": [
      {
        "children": [],
        "parents": [
          "3937"
        ],
        "id": "4206",
        "visibility": 1,
        "name": "Slice1"
      },
      {
        "children": [
          "4206"
        ],
        "parents": [
          "0"
        ],
        "id": "3937",
        "visibility": 1,
        "name": "Superquadric1"
      }
    ],
  },
  "name_pattern": "{pose}.png",
}
```


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.

[2] David Rogers, James Ahrens, and John Patchett. Cinema simple database specification. Technical Report LA-UR-15-20572, Los Alamos National Laboratory, January 2015.