The Kinemage File Format, v1.0

Ian W. Davis, Jane S. Richardson, David C. Richardson

25th June 2004

Abstract

This document describes the syntax and semantics of the core feature-set of the kinemage file format. Since even the core features are extensive, the description has been broken into two logical halves. The first section describes the semantics and high-level syntax of kinemage files—all the information typically needed to author a new kinemage file from scratch. The second section formally describes the low-level syntax that underlies the entire format, which is important for programmers writing kinemage parsers.

This document does not cover the philosophy, rationale, or history of the kinemage format, nor is it a tutorial in using or creating kinemages. It assumes the user has some experience viewing and interacting with existing kinemages and now wants either (1) to create a new kinemage file by hand or programmatically, starting from scratch, or (2) modify an existing kinemage file by hand at the plain-text level.

Contents

Part I
Semantics

This first part of the document describes the semantics and high-level syntax of kinemage files, giving descriptions of the most commonly used keywords and the various options and parameters that accompany them. This level of detail is important both to kinemage authors and to programmers who want to use the kinemage format. While the core features described here are fairly stable and will remain so, additional features may be added with some frequency. Except as noted, all the features described here work in both Mage and KiNG, which are the two primary kinemage viewers as of this writing.

A note about syntax: the full description of kinemage syntax appears in the second part of this document, but it's more information than most authors will need. The kinemage format is a fairly intuitive, mostly free-form language that should be easy to pick up. That said, here are three "gotchas" that new authors should be aware of:

- Keywords (words starting with the @ symbol) *must* appear at the very beginning of a line in order to be recognized. Nothing can come before them, not even spaces.
- Properties (words that end with an = sign) cannot have a space (or anything else) between the word and the = sign. Space after the = is optional, but it is

not permitted before. Thus, we can write `color= red` or `color=red`, but not `color = red`. This looks a little awkward at first, but you'll soon get used to it.

- List definitions must appear all on one line. This is a good rule of thumb for the whole file (though not a requirement): each keyword starts a new line, and all its relevant options and parameters appear on that same line. The next keyword starts another line. When defining a list, one point is listed per line.

1 Overview of a kinemage file

The kinemage file format is a plain-text, human-readable, human-editable format for three-dimensional vector graphics. The overall structure is an optional `@text` block describing the contents of the file, followed by one or more kinemages. The kinemages themselves encode a hierarchical organization of 3-D graphics primitives like lines, balls, and triangles that have been optimized to convey the most possible information about the ideas underlying the visualization. Each of these kinemages begins with a `@kinemage` statement, followed by display options, view and master definitions, *etc.* (the header), and then followed by a series of group, subgroup, list, and point definitions (the body).

2 Lists

Depending on how you look at it, either the list or the point could be considered the fundamental unit of a kinemage. Points specify particular locations in three-dimensional space by listing an X, Y, and Z coordinate. Lists bring together collections of points to describe 3-D *primitives*. Some primitives are defined by only one point (for instance, a ball is specified by it's center), but others need more – a line segment needs two endpoints, and a triangle needs three corners. Each list has enough points in it to describe one or more primitives, and all the primitives in a list are of the same type – all balls, say, or all triangles. For example, a list might contain all the line segments that form the outline of a cube, or all the dots representing one particular data set on a graph. Where it makes logical sense to group a bunch primitives into a single list, it's a good idea, because it will more efficient than creating a separate list for each one.

There are seven basic types of list, corresponding to seven different types of primitive:

Ball lists specify spheres of some finite size. They're typically drawn as flat, filled circles plus a little white highlight, which looks quite convincing as a "real" rendered sphere. However, they may not look right if they intersect other objects, except for line segements with an end point at the ball's center.

Sphere lists are very similar to ball lists, but hint to the display program that the ball is large and may intersect other objects in a complex way. Depending on the kinemage viewer you're using, the spheres may look more realistic, at the expense of taking more time to render.

Dot lists specify small points in space, like a ball or sphere that's just big enough to see. They're a good alternative to very small balls or spheres, because they're faster to draw.

Label lists specify short text labels anchored to a particular point in space. The label swings around as the kinemage is rotated so it's always facing forward and is right side up.

Vector lists specify a set of line segments, which are often connected head-to-tail in a *polyline*. However, a vector list can contain any number of separate line segments, too.

Triangle lists specify filled triangles. Like vectors are often chained together into polylines, triangles are chained into *triangle strips*. Points 1, 2, and 3 make up the first triangle; then points 2 and 3 are reused together with point 4 for a second triangle. On it goes, with a 3, 4, and 5 making the third triangle; 4, 5 and 6 making the fourth, and so on. Traditionally, all the triangles in one list have to be part of a single connected strip.

Ribbon lists are very similar to triangle lists, except that pairs of consecutive triangles are assumed to lie in the same plane. In fact, the lighting effects are manipulated so that the four points look like they form a flat, four-sided polygon even if they really don't. As the name implies, this is used for ribbons that curl through space but need to look smooth.

In the kinemage file, lists are specified by an @ symbol followed by the list type and the word "list", all lower case and without any spaces in between. The entire list definition must be on a single line; it cannot be split across multiple lines. The first item after the @list keyword is the name of the list, enclosed in curly braces. The options listed below then follow in any order, as desired:

- The word `off` requests that the list not be initially visible when the kinemage is loaded. If it has an on/off button in the button panel, the user may turn it on manually, or it may be turned on by a master button.
- The word `nobutton` requests that the list not have an on/off button even if it would have otherwise. Note that many lists will not have an on/off button anyway because their subgroup or group is `dominant`.
- `radius= #.#` and `width= #` are used to specify the size of balls/spheres and vectors/dots, respectively. Radius may be a decimal number, but width refers to a number of pixels on the screen and must be an integer between 1 and 7. Lines and dots default to a width of 2. You should always specify a radius for balls and spheres explicitly.
- Ball or sphere lists marked with `nohighlight` will be drawn as flat colored disks and will not have a white highlight drawn to make them look 3-D.

- `color= colorname` specifies the base color for objects in the list; however, the list color interacts with the colors of individual points (if any). By default, lists are white. See 6 for more information about colors.
- `alpha= #.#` specifies the opacity of objects from 1.0 (fully opaque) to 0.0 (invisible). Alpha is currently supported only by KiNG and only for triangle or ribbon lists.
- `master= {mastername}` specifies that this list is controlled by the named master. A given list may have multiple `master=` statements, but you should read 5 to see how multiple masters interact.
- `clone= {listname}` and `instance= {listname}` specify that this list has the same points in it as the named list, which must be of the same type and must have been declared before the current list. This list, however, may have different properties (color, radius, *etc.*) than the other one. Clone is purely a convenience for kinemage authors, the current list is a totally independent copy of the other list that just happens to have the same contents. When the kinemage is saved, both lists will be written out in full without using `clone=`. Instance, on the other hand, actually re-uses the same point data, so editing the points of one list will affect all its clones too. The `instance=` property *will* appear in the saved kinemage. Instance is a good way to make more efficient use of memory in certain cases, such as when an identical object needs to appear in multiple frames of an animation.

A few sample list declarations are shown below.

```
@balllist {Circles} radius= 3.5 nohighlight off
@spherelist {Mars look-alikes} radius= 10000 color= red master= {planets} nobutton
@vectorlist {tiger's tail} color= orange width= 5
@trianglelist {veil} color= yellowtint alpha= 0.25
@trianglelist {a newer veil} color= white alpha= 0.3 clone= {veil}
```

3 Points

Points determine the actual geometry of the objects in a kinemage file. At a minimum, each point must specify an X, Y, and Z coordinate. Coordinates are given in a right handed Cartesian system. The "Cartesian" part just means that the X, Y, and Z axes are all at right angles to each other. The "right handed" part means that if you're looking down the positive Z axis toward the origin, the positive Y axis goes up and the positive X axis goes to the right. Coordinates can be any possible decimal number, but it's a good idea to not make them all really large (say, all in the millions) or really small (thousandths and less), because you may lose accuracy in some kinemage viewers.

In addition to coordinates, it's a very good idea to give every point an ID, even though this is not strictly required. The ID appears when the user picks the point with the mouse, and for points in a label list the ID is actually the text that will be displayed for the label. IDs are allowed to be empty (just a pair of curly braces), and this is

preferrable to no ID at all. The special ID `{"}` means that the point will have the same ID as the point that preceded it; if all the points in a list have the same ID, all but the first can have `{"}` for their ID.

Points are typically writen one to a line, though they can span multiple lines or more than one point can appear on the same line. The ID must come first, enclosed in curly braces, and the coordinates must come last. In between, there are a lot of per-point options that can be employed in any order:

- The `P` flag marks a point in a vectorlist as starting a new polyline. The first point in the list is automatically `P`, but after that line segements are drawn from one point to another until a `P` point is encountered. To draw a series of disconnected single line segments, every other point should be marked `P` (starting with the first one). The `P` flag does not affect triangle lists.
- Points marked with the `U` flag are "unpickable" under normal circumstances, meaning that clicking on them with the mouse will not do anything.
- In KiNG, the `X` flag can be used to break one triangle list into multiple triangle strips, analogous to the `P` flag for vector lists.
- Points can be given their own color just by writing the color name. They can also be assigned alternative colors for different coloring schemes through the use of aspects (see 6). Aspects are lists of single uppercase letters A - Z enclosed in parentheses. If aspects are used in a file, every point should have the same number of aspects specified. The color of a line segment is the color of its second point, not its first; likewise, the color of a triangle is the color of its third point.
- Width and radius can also be specified on a point-by-point basis. Widths are specified as `width1`, `width2`, ... `width7`; radii are specified as `r= #.#`.
- The visibility of points can also be controlled by pointmasters, which are analogous to the masters that control lists, subgroups, and groups. Pointmasters are identified by single character codes (the lowercase letters a - z and the numbers 1 - 6) enclosed in single quote marks. Multiple point masters interact differently than multiple masters do; see 5 for more information.
- Each point can have an additional text "comment" associated with it, which should be enclosed in angle brackets. Some kinemage viewers use these for special purposes, while others may ignore them altogether.

Some example point definitions are shown below; these points are not intended to all belong to the same list!

```
{clown nose} red r= 2.4 1.0 2.0 3.0
{x-axis}P U 0 0 0
{"} <other end of X axis> U width1 10 0 0
{really complicated} 'aeg' (HZTU) 8.31 19.78 42.13
```

4 Kinemages, groups, and subgroups

Complicated kinemages may have hundreds of lists in them, which would quickly become unmanagable for the user. Groups and subgroups allow us to organize lists hierarchically, so that sets of related objects can be shown or hidden as a unit, and unneccessary detail can be supressed. There are also cases where several kinemages deal with different aspects of the same visualization problem, and the kinemage format provides for collecting these multiple kinemages into a single file.

Only the start of each kinemage, group, subgroup, or list is marked, and not the end. A kinemage declaration must appear at the start of the file, and everything else in that file is considered part of the kinemage until another kinemage declaration is found. In the same way, a group includes all the subgroups and lists that follow it, until another group is declared or the end of the file is reached. Likewise, subgroups contain all the lists that follow them, until another group or subgroup declaration is encountered. Lists contain all the points that follow them, until another list, subgroup, or group declaration is encountered. In this way, a hierarchical organization is built up with points gathered into lists, lists gathered into subgroups, subgroups gathered into groups, and groups gathered into kinemages.

Kinemage declarations are very simple: the `@kinemage` keyword, followed by an identifying number. The first kinemage in a file should be number 1, the second should be number 2, and so on. Thus, every kinemage file starts like this, with nothing preceding it except possibly a `@text` block (see 8):

```
@kinemage 1
```

Group and subgroup declarations are only slightly more complicated. They start with `@group` or `@subgroup`, respectively, followed by the (sub)group name in curly braces, possibly followed by some of the following flags. As with lists, group and subgroup declarations may not span multiple lines.

- The word `off` requests that the (sub)group not be initially visible when the kinemage is loaded. If it has an on/off button in the button panel, the user may turn it on manually, or it may be turned on by a master button.
- The word `nobutton` requests that the (sub)group not have an on/off button even if it would have otherwise. Note that some subgroups will not have an on/off button anyway because their group is `dominant`.
- The word `dominant` requests that the buttons of objects below this (sub)group in the hierarchy not be shown. Dominant subgroups hide the buttons of their lists; dominant groups hide the buttons of their subgroups and their lists.
- The word `collapsable` is similar to dominant. When a collapsable group is on, the buttons of its subgroups and groups are visible as usual. When the collapsable group is off, however, those buttons are supressed, as though it were `dominant`. The situation is analogous for collapsable subgroups and the lists under them.

- `master= {mastername}` specifies that this (sub)group is controlled by the named master. A given (sub)group may have multiple `master=` statements, but you should read 5 to see how multiple masters interact.

- The words `animate` and `2animate` can only be used with groups. Groups so marked become part of the first or second animation, respectively. When the kinemage is loaded, all animate groups except the first one are turned off, regardless of any `off` flags. The user can the cycle the animation forward/backward so that the next/previous group is on and all the others are off. Animate groups can be turned on or off by the user without any restrictions, but stepping forward or backward in the animation will again ensure that only one of them is on at a particular time. The `2animate` flag lets authors establish a second, unrelated animation that behaves in exactly the same way. In general, no group should be marked with both `animate` and `2animate`.

The following are typical group and subgroup declarations:

```
@group {first frame} dominant animate
@group {not visible} dominant nobutton master= {use this instead}
@group {lots of stuff} collapsable
@subgroup {not very important} off master= {optional stuff}
```

5 Masters

The so-called “master” buttons provide an important facility for complex kinemages: the ability to group and organize the elements by a secondary scheme that may be very different from the primary, hierarchical organization. For example, if you were making an interactive map of the world, you might decide to make one group for each continent, and one subgroup for each country. However, it might also be nice to turn on and off all the rivers together, or all the cities. There might be a `{rivers}` lists in each country, but without masters all of them would have to be toggled individually. With masters, you can have all of the rivers toggled by a single button that lives outside the ordinary hierarchy of groups, subgroups, and lists.

Masters are automatically created whenever they’re mentioned the `master=` part of a list, subgroup, or group declaration. Their buttons appear in the same button panel as group/subgroup/list buttons, but after all of those and somewhat separated from them. You can control the order and presentation of masters a little bit better by using the `@master` keyword, which usually appears in the kinemage “header” – after `@kinemage` but before the group, subgroup, and list declarations. `@master` is followed by the master name in curly braces, which must exactly match the name used in `master=` statements. The name may be followed by the `indent` flag, which hints that its button should be indented relative to the other master buttons so as to imply the same sort of hierarchy that occurs in the regular buttons. (However, for the masters this is purely cosmetic.)

The effect of a master on a list, subgroup, or group is transient – turning something on or off with a master does not prevent the user from turning that item on or off man-

ually. However, for items marked with more than one master, the masters do interact with each other. Consider the following list:

```
@dotlist {demo list} master= {A} master= {B} master={C}
```

If *any* one of the master buttons is toggled from on to off, our list will be turned off. However, if one of the masters is toggled from off to on, our list will be toggled on if and only if *all* of the other masters that control it are also currently on. (Of course, if the list was already on, it will remain so.) If, for example, masters B and C are off and master A is on, then turning B on will *not* turn the list on (because C is off). Subsequently turning C on *will*, however, turn the list on (because both A and B are also on).

Individual points can also be controlled by a master-like mechanism, called *pointmasters*. Due to memory limits, there are only 32 possible pointmasters that can be used in a particular kinemage. They are identified by the lowercase letters *a* - *z* and the digits 1 - 6. One or more of these single-character codes are listed inside of single quote marks for some or all of the points in a kinemage. Each pointmaster code is associated with a named master button by a line that starts with `@pointmaster`, then one (or rarely, more than one) single-character code between single quotes, then the master name in curly braces. If the name matches with the name of an ordinary master, then that button will control both the list/subgroup/group master and the pointmaster.

Unlike masters, pointmasters do not interact with each other. Turning a pointmaster on turns on all of the points under its control, regardless of which other pointmasters they are controlled by. The same goes for turning a pointmaster off.

Shown below are some typical master and pointmaster declarations:

```
@master {rivers}
@pointmaster 'a' {large cities}
@pointmaster 'b' {small cities}
@pointmaster 'ab' {all cities}
@master {dual purpose} indent
@pointmaster 'c' {dual purpose}
```

6 Colors and aspects

Color-coding is one of the most-used feature in any visualization system, so the kinemage format provides lots of options related to coloring. We've already seen how to assign a color to a whole list or a single point in their respective sections, and we've seen that point colors, when present, generally override the color specified for the list. Below are all 28 of the color names that can be used with lists and points:

Saturated colors	*Semi-sat. colors*	*Pastel colors*	*Neutrals*
red (A)	pink (N)	pinktint (V)	
orange (B)	peach (P)	peachtint (Q)	
gold (C)			
yellow (D)	yellow (D)	yellowtint (R)	
lime (E)			
green (F)	sea (G)	greentint (S)	
sea (G)			white (W)
cyan (H)			gray (X)
sky (I)			brown (Y)
blue (J)	sky (I)	bluetint (T)	
purple (K)	lilac (O)	lilactint (U)	invisible (Z)
magenta (L)			deadwhite
hotpink (M)			deadblack

The letters listed in parentheses are the aspect codes for each color, which will be discussed below. Colors are organized so that the columns form progressions of hue, and the rows form progressions of saturation, although the relationships are somewhat different on a white background. Some colors appear in two different places in the chart because they serve two different "purposes". See the palette kinemage built into KiNG for more details on how to use color effectively.

Sometimes you might develop a kinemage in which tens or hundreds of different lists should all be the same color – but you aren't sure *which* color. Instead of using the find-and-replace feature of a text editor to test out different options, you can define a symbol to stand in for the color, and then change only the definition of the symbol. This symbolic or "variable" color name is called a *colorset*, and is declared with the `@colorset` keyword, followed by the symbolic name in curly braces and then the name of a normal kinemage color. Later on, you can assign the symbolic color name to lists (but not individual points). For example,

```
@colorset {water color} sky
@vectorlist {river} color= {water color}
@balllist {ponds} color= {water color}
```

Since color schemes are so important to visualization, it is sometimes useful to have multiple color schemes within one kinemage. For example, a map might be colored by elevation, by rainfall, or by population density, depending on its intended use. *Aspects* provide a mechanism for specifying more than one possible point color for each point, only one of which is active at a particular time.

Aspects must be declared with the `@aspect` keywords in the kinemage header; the declaration is not optional as it is *e.g.* for masters. Each point that has aspect coloring (not all points in a kinemage have to) should have the same number of aspect codes as there are `@aspect` definitions in the kinemage. As explained in the section on points, the single-letter aspects codes for a point appear inside parentheses as part of the point definition (see 3). For example:

```
@1aspect {Population density}
```

```
@2aspect {Quality of universities}
@3aspect {Number of bars}
@balllist {Cities in the Triangle} color= white radius= 2
{Durham} (ABC) 0 1 0
{Raleigh} (DEF) 1 0 0
{Chapel Hill} (GHI) -1 0 0
{Cary} (JKL) 0.5 -0.5 0
```

7 Views and display options

There are a number of keywords that control the default presentation of a kinemage to the user. Choosing the right options is an important guide to the user, although s/he can override them later.

All of the following keywords are used to define a pre-set view of the kinemage. The keywords are given a leading number that defines *which* view they belong with (show below for view 1, but 2, 3, *etc.* can be substitued to define additional views). View 1 is the default that will be shown when the kinemage is first opened. Although a view definition may legally omit any of these components, it's best to define all of them explicitly to ensure the desired behavior. In the definitions below, the symbol # stands for any number, decimal or integer.

@1viewid {VIEW NAME} gives a label that will identify this view to the user. It should be unique, but is not required to be.

@1center # # # gives the coordinates of the center of the view. The model will rotate around this point, which will be centered.

@1matrix # # # # # # # # # gives an orthonormal rotation matrix that defines the orientation of the model. (*Orthonormal* meaning all the row vectors are orthogonal to each other, all the column vectors are orthogonal to each other, and all these vectors have length 1.) If you intend to multiple this matrix by your coordinates (as a column vector), you should read the numbers as going down the first column, then down the second, and so on. If you intend the multiply your coordinates (as a row vector) by the matrix, then you should read the series of numbers as going across the first row, then across the second, *etc.* That is, one version of the matrix is the transpose of the other.

@1span # defines how much of the kinemage is visible—whether you're zoomed in close or zoomed way out. Specifically, the given distance in model coordinates will just fill the graphics area either horizontally or vertically (whichever is smaller). Thus, larger spans show more of the model, and smaller spans show less (but in greater detail).

@1zoom # is an alternative specification for span; it controls how much of the kinemage is visible. A zoom of 1.0 ensures the whole kinemage just fits within the graphics area, and larger zooms cause the view to zoom in closer. It's better to give a span than a zoom, because zoom depends on the space the kinemage occupies. If later

add (or remove) something to the kinemage that changes its "envelope", then your predefined views will shift to show something other than you had originally intended. Span, on the other hand, is independent of the content of the kinemage.

@1zslab # is the complement of span or zoom—it defines how thick a slice of the model you can see. Obviously, if the full model was displayed in-focus when you were zoomed in very close, all the extra detail in the far background could be extremely distracting. Ditto for things right in front of your nose that could blot out the area of interest. Thus, everything that's more than a certain distance in front of or behind the center of rotation is not shown. (Computer graphics folks call this a "slab" or a pair of "clipping planes.") The units here are arbitrary: a value of 200 means that the front-to-back distance between clipping planes is equal to (the lesser of) the width or height of the graphics area. Other values mean the slab will be #/200 times this wide, so smaller values give a thinner slab and larger values, a thicker one.

There are a number of other useful keywords that correspond to display settings common in most kinemage viewers. As note above, these are only hints: the user can always choose to override them, and the kinemage viewer is not even guaranteed to pay attention to them.

@perspective suggests that the kinemage been shown with simulated perspective projection. This is often desirable for geometric objects, so that parallel lines actually converge in the distance, cubes really look like cubes, and so on. By default, kinemages are shown with orthographic projection.

@flat hints that there is no useful depth (Z-coordinate) information in the kinemage, and that the default mode of interaction should be translation (sliding the kinemage around in the X-Y plane) rather than rotation. This is helpful for things like 2-D charts and graphs.

@onewidth asks that lines be drawn in a consistent width regardless of their location. By default, lines in the front of the view are thicker and lines in the back are thinner, to aid in giving a feeling of depth and three-dimensionality.

@thinline suggests all lines be drawn as thin as possible, regardless of their specified `width=`.

@whitebackground hints that the kinemage would look best on a white background, with its associated color palette. By default, the black background and palette are used.

@listcolordominant asks that the individually specified colors of points be ignored in favor of the base color of their list.

8 Metadata

In addition to describing a geometrical object or scene, the kinemage language allows authors to describe the *meaning* of the graphical objects. This sort of information

is thus data about the (primary) data, a.k.a. *metadata*. The following keywords are supported:

@text marks the beginning of a block of free-form, plain-text information that should be made available to users of the kinemage. The text continues until the next keyword is encountered; thus, the only restriction on the content of text block is that it not contain any @ symbols at the very beginning of a line. Indenting the @ with a space is a perfectly acceptable as a way of getting around this limitation. Text is specified for the kinemage *file* as a whole. Thus, it will probably pertain to all the kinemages in that file, whether by describing them sequentially or discussing the relationships among them. Some kinemage viewers support special hypertext links in the text, which are delimited by `*{` and `}*`. The specific syntax is described elsewhere. Multiple `@text` blocks in the same file will be concatenated together in the order they appear.

@caption works much like `@text`, but is generally shorter (a few lines at most) and pertains to a single kinemage. Thus, `@caption` must appear somewhere after a `@kinemage` statement, while `@text` can be the first thing in a file.

@title {KIN TITLE} gives a brief title that identifies this kinemage, as a more user-friendly label than its index number.

@copyright {COPYRIGHT INFO 2004} notifies users of who owns the copyright to this kinemage file.

@pdbfile {FILENAME} lists a Protein DataBank file that corresponds to the model shown in this kinemage. Used only for kinemages showing macromolecular structures.

@mapfile {FILENAME} lists an electron density map that corresponds to the model shown in this kinemage. Used only for kinemages showing macromolecular structures.

@command {UNIX CMD} suggests a command that the user or the kinemage viewer could run to generate additional kinemage data, which could then be merged into the current file.

9 Alternative spellings

Some of the keywords in kinemage files may take alternate forms, some of which are historical artifacts and some of which are attempts to accomodate both American and British spelling. The forms given above are the preferred ones; the alternatives listed below may not be supported by all kinemage viewers.

@balllist @ball

collapsable collapsible recessiveon

color= colour=

deadblack black

@dotlist @dot

@flat @flatland @xytranslation

gray grey

L l D d *(the unneccessary point flag: L for Line-to, D for Draw-to)*

@labellist @label

@listcolordominant @listcolordom

nohighlight nohilite nohi

orange rust

P p M m *(the point flag: P for Point, M for Move-to)*

@ribbonlist @ribbon

sea seagreen

sky skyblue

@spherelist @sphere

@subgroup @set

@trianglelist @triangle

U u *(the point flag)*

@vectorlist @vector

@whitebackground @whiteback @whitebkg

X x *(the point flag)*

yellowtint paleyellow

@zslab @zclip

Part II
Syntax

This part of the document describes the low-level syntax that is common to all kinemage formats, regardless of how many additional functionalities (semantics) they incorporate. The descriptions are very precise, at the cost of being somewhat long and tedious. However, this level of detail is necessary for programmers who wish to interpret

kinemage files reliably, and may be helpful to authors as well. This level is expected to be extremely stable and change very slowly.

The descriptions in the two parts of this document assume a similar division of labor in the implementation of computer programs that process kinemages: the low-level syntax (this part) is handled by a tokenizer, which can separate a stream of characters into meaningful atomic units. The semantics and high-level syntax (the preceding part) are handled by a parser, which is responsible for understanding, *e.g.*, the relationships among graphics objects and their implied hierarchical organization.

10 Characters in kinemage files

Kinemage files are plain text files encoded according to the ASCII standard[1], which defines 128 characters. Each character is stored in the lower 7 bits of a single byte. Only ASCII characters between 32 and 126 inclusive, plus 9 (horizontal tab), 10 (newline), 12 (formfeed), and 13 (carriage return) are legal characters in a kinemage file (numbers given are in decimal).

A kinemage tokenizer may check for and report illegal characters, but is not required to. If the tokenizer does find illegal characters, they should not cause a fatal error, but should instead be treated as alphanumerics (see 10.2).

10.1 Whitespace

Whitespace characters are the space (32), horizontal tab (9), newline (10), formfeed (12), carriage return (13), and the comma (44). Commas are defined as whitespace to simplify treatment of a sequence of numbers, which is often writen out with commas as separators.

The kinemage format is whitespace insensitive: these characters carry no meaning and may be discarded at the tokenizer level. Where whitespace is called for, one or more whitespace characters may be used, and any sequence of continguous whitespace characters is treated as a single occurance of whitespace. However, there is one important semantic attribute conveyed by whitespace: the newline and carriage return characters impart the beginning-of-line (BOL) property to any token immediately following them. See 11.1 for details.

When using whitespace, keep in mind that kinemage files should be human-readable and human-editable. Line length should not exceed 80 characters, but superfluous line breaks should be avoided. Single spaces are the preferred form of whitespace within a line. These suggestions are merely matters of style, and a kinemage tokenizer must not rely on them being followed.

10.2 Alphanumerics

Alphanumeric characters are the uppercase letters A-Z, the lowercase letters a-z, and the digits 0-9. Note that kinemage files are case sensitive. Kinemage tokenizers must

[1] See `http://www.asciitable.com/` for details.

not convert or mangle the case of any tokens in a kinemage file, and tokens that differ only by case must still be considered distinct from one another.

10.3 Punctuation

All legal characters that are neither classified as whitespace nor as alphanumerics are regarded as punctuation. These characters have a variety of functions in the kinemage format. The following characters already have well-defined function and syntax associated with them:

```
@ ( ) - = + { } “ ’ < . >
```

At the moment, no special significance has been attached to the following characters:

```
‘ ~ ! # $ % ^ & * _ [ ] \ | : ; / ?
```

However, a future version of the format may define meanings for them.

11 Tokens in kinemage files

Files in kinemage format can be thought of a sequences of tokens (meaningful), each separated from the others by zero or more whitespace characters (meaningless). Tokens are divided into two classes, quoted and unquoted. Quoted tokens have clear start and end signals, so they can occur with no intervening whitespace and still be separable. Unquoted tokens lack clear start and/or end signals. Thus, at least one whitespace character is *required* between two unquoted tokens in order to separate them from one another.

Theoretically, each token may be of any length, from one character (even zero characters, for quoted tokens) up to the largest string that will fit in memory. In practice, however, tokens should be fairly short; 20 characters or less is a reasonable guideline. No token should exceed 256 characters in length, and more stringent restrictions on length may be imposed on some tokens by the higher-level syntax.

The names given to token types below reflect their usual function in a kinemage file, but they are not restricted to that function. For example, an identifier usually names some object, but it can also enclose a command line, a file name, and so on.

11.1 Beginning-of-line

Beginning-of-line (BOL) is a property of certain tokens that may influence their interpretation by the parser. For instance, for a token to be recognized as a keyword (see 11.3.1), it must occur at the beginning of a line. A token is considered BOL under any of the following conditions:

- The first character of the token is the first character in the file
- The first character of the token is immediately preceded by a newline
- The first character of the token is immediately preceded by a carriage return

11.2 Quoted tokens

Quoted tokens all have explicit markers for the beginning and end of the token. This simplifies the parsing of these tokens, and enables one to classify the type of token present after parsing the first character of it. However, care must be taken to close every token that is opened. To aid authors in discovering such errors in their kinemages, it is recommended that kinemage tokenizers report a non-fatal error when they encounter the end of the file before closing an open quoted token.

11.2.1 Identifiers

Identifiers are strings quoted by curly braces, like this:

```
{an identifier}
```

An identifier token begins when an opening curly brace is encountered outside of any other quoted token (but possibly "inside", *i.e.*, immediately following, an unquoted token). It terminates as soon as the number of closing curly braces encountered in the course of parsing this token equals the number of opening curly braces encountered. That is, curly braces may be nested within an identifier, but only as long as they are balanced. Otherwise, an identifier may contain any legal character for a kinemage file.

11.2.2 Comments

Comments are strings quoted by angle brackets, like this:

```
<a comment>
```

A comment token begins when an opening angle bracket is encountered outside of any other quoted token (but possibly "inside", *i.e.*, immediately following, an unquoted token). It terminates as soon as the number of closing angle brackets encountered in the course of parsing this token equals the number of opening angle brackets encountered. That is, angle brackets may be nested within a comment, but only as long as they are balanced. Otherwise, a comment may contain any legal character for a kinemage file.

11.2.3 Aspects

Aspects are strings quoted by parentheses, like this:

```
(an aspect)
```

An aspect token begins when an opening parenthesis is encountered outside of any other quoted token (but possibly "inside", *i.e.*, immediately following, an unquoted token). It terminates as soon as the number of closing parentheses encountered in the course of parsing this token equals the number of opening parentheses encountered. That is, parentheses may be nested within an aspect, but only as long as they are balanced. Otherwise, an aspect may contain any legal character for a kinemage file.

11.2.4 Single quoted strings (pointmasters)

Pointmasters are represented as strings deliminted by single quote marks, like this:

```
'abc'
```

A single quoted token begins when a single quote mark is encountered outside of any other quoted token (but possibly "inside", *i.e.*, immediately following, an unquoted token). It terminates as soon another single quote mark is encountered. That is, single quoted strings may not contain embedded single quotes, and no mechanism exists to escape this limitation. Otherwise, a single quoted string may contain any legal character for a kinemage file.

11.2.5 Double quoted strings

Double quoted strings are defined analogously to single quoted strings, like this:

```
"abc"
```

A double quoted token begins when a double quote mark is encountered outside of any other quoted token (but possibly "inside", *i.e.*, immediately following, an unquoted token). It terminates as soon another double quote mark is encountered. That is, double quoted strings may not contain embedded double quotes, and no mechanism exists to escape this limitation. Otherwise, a double quoted string may contain any legal character for a kinemage file.

At the moment, no function has been ascribed to double quoted strings in kinemage files. Until a meaning is defined, parsers should ignore them.

11.3 Unquoted tokens

Unquoted tokens are somewhat harder to parse than quoted tokens, because their start and end signals are less obvious. Also, the entire token may need to be parsed before one is able to decide what kind of token it is. However, rules for parsing these tokens are well-defined. An unquoted token may begin in any of the following positions:

- At the beginning of the file
- Immediately following the end of a quoted token
- Following one or more whitespace characters, outside of all quoted tokens

An unquoted token may begin with any non-whitespace character that does not begin a quoted token. An unquoted token is then terminated by the first of these encountered after the initiating character:

- The end of the file
- Any whitespace character
- The equals sign (ASCII 61)

- Any character that begins a quoted token: `{ < ( ' "`

While the initiating character is considered part of the token, the terminating character may or may not be considered part of the token. Whitespace will be discarded, and quoted token initiators will be part of the next (quoted) token. The equals sign will be kept as part of this token. Note that this means tokens ending in the equals sign (called "Properties"; see 11.3.2) are effectively half-quoted: there need not be whitespace after the equals sign to separate this token from the next (unquoted) token.

The following sections present rules for categorizing unquoted tokens. The rules are in order of precedence—that is, a token must be classified according to the *first* rule it matches from this list. This resolves the ambiguity that would arise if, *e.g.*, a token began with an "at" sign (like a keyword) and ended with an equals sign (like a property).

11.3.1 Keywords

Keywords define the major sections of a kinemage. Each keyword begins with the "at" sign (64). For example, all of the following are keywords:

```
@kinemage @master @vectorlist
```

Furthermore, in order to be recognized as a keyword, the "at" sign must occur at the beginning-of-line (BOL; see 11.1). In addition to enforcing good style, this streamlines the processing of plain text segments (see 11.4).

11.3.2 Properties

Properties are generally used for labeling the meaning of the next token in the file. Each property ends with an equals sign (61). The following are all properties:

```
color= master= radius=
```

Note that, by definition of an unquoted token, whitespace is forbidden before the equal sign. Although some old kinemages may allow this syntax, it requires the tokenizer to read ahead through an arbitrary amount of whitespace following every unquoted token in order to determine if it is a property or not. This behavior could be undesirable if the kinemage contains sections of plain text (see 11.4) or is embedded within some other data format.

As described above, there may be whitespace after the equals sign, but it is not required, even if the next token is unquoted. This semi-quoted (quoted at the end, but not the beginning) behavior of property tokens is a historical feature of the kinemage format that has been retained for backward compatibility. The preferred format for new kinemages is to have a space following the equals sign.

There are no low-level syntactic restrictions on the positioning of properties; however, at a higher level, syntax generally requires that each property be followed by a non-keyword, non-property token. For example:

```
color= red
width= 7
master={backbone}
radius=2.5
```

11.3.3 Integers

Integers are exactly that: text representations of integer numbers. Legal integers are either the single digit zero, or a non-zero digit followed by zero or more additional digits and optionally preceded by a plus or minus sign. The following are legal integers:

```
0 +1 7 -365 2020
```

The following are *not* legal integers:

```
-0 007 5+2
```

Tokens that are not legal integers but consist only of digits 0-9 and the plus and minus signs (*e.g.*, the above) may be interpretted as integers or as literals on a case-by-case basis, at the discretion of the tokenizer. It is recommended that a warning be issued if such a token is encountered.

11.3.4 Numbers

Numbers are a superset of the integers: text representations of real numbers in decimal or scientific notation. Legal numbers follow the pattern[2] below:

```
number ::= integer fraction? exponent?
fraction ::= '.' digit+
exponent ::= ('e' | 'E') integer
```

Basically, there must be something before the decimal point, even if it's a zero; there must be something after the decimal point, if there is one; and the exponential part (if present) may be indicated with either a capital or a lowercase E. The following are legal numbers:

```
-0.42 1e5 3.14 6.022E+23
```

Tokens that are not legal numbers but consist only of digits 0-9, the letters **e** and **E**, the decimal point, and the plus and minus signs may be interpretted as numbers or as literals on a case-by-case basis, at the discretion of the tokenizer. It is recommended that a warning be issued if such a token is encountered.

[2] See `http://www.garshol.priv.no/download/text/bnf.html` for an introduction to Extended Backus-Naur Form.

11.3.5 Literals

Legal unquoted tokens that cannot be otherwise classified are lumped together as literals. Note that, by the definitions provided for unquoted tokens, a literal may begin with a numeric digit. This is in contrast to many programming languages. Those defining new semantics for kinemages are strongly advised against defining literals that are not numbers but use only characters allowed in numbers; the interpretation of such tokens is poorly defined (11.3.4). In fact, it is recommended that literals contain only alphanumeric characters and that they start with a letter rather than a number. The following are all legal literals:

```
animate 2animate red blue green big_long_literal
```

11.4 Plain text blocks

In addition to the ordinary, tokenizable parts of a kinemage file, sections of text data that do not conform to the rules for tokens may be embedded. This data cannot be processed as usual by the tokenizer for two reasons:

1. The data is in an unknown format, and whitespace may be significant.
2. The data may "open" a quoted string but never close it, thereby hiding the remaining content of the file.

An example of this is the plain text write-ups that follow the `@text` keyword; however, future kinemages could conceivably contain embedded HTML, base-64 encoded binary resources, *etc.* At the moment, there is no purely syntactic means for identifying such regions. However, upon the request of the parser, the tokenizer must be able to deliver the unaltered text content of the file from the current position until reaching a kinemage-format keyword (*i.e.*, a new line or carriage return followed by an "at" symbol).