MegaMol Simple Particle Data File Format (*.mmspd)

Date: 07.11.2011
Revision: 5
Status: Released
Author: Sebastian Grottel
Contributors: Guido Reina, Bertram Thomaß

Visualization Research Center, University of Stuttgart

1 About this Document

This document describes the MegaMol Simple Particle Data File Format (*.mmspd). This format is meant for interactive visualization of medium-sized particle data sets, allowing for flexible particle type definitions. Especially, this format is not meant to store derived data (features, like surfaces, etc.) from the particle data, but only the particle data itself.

This document is meant as reference for Implementations for loading or writing data in this format. For further information about MegaMol, visit the project website:
http://www.visus.uni-stuttgart.de/megamol/

2 General Information

The mmspd-file format represents particle data in binary form or in text form. Big endian and little endian are supported, however, little endian is strongly recommended.

For the file format in text form, the following encodings are supported:

Encoding	Description
ASCII	It is strongly recommended only to use 7-Bit ASCII to avoid any problems with code pages (i.e. the 8th bit of each byte is zero).
UTF-8	(With or Without BOM) The format marker will specify Unicode to resolve ambiguity with ASCII.

The use of the binary format is recommended. Alternatively, the use of UTF-8 with BOM is recommended.

When storing the data in text form any leading and trailing spaces are to be ignored, since they can be introduced for performance reasons, except for leading spaces in the first text line, as this line will be used as format marker to detect the file format. Line breaks can be '`0x0A`' (Unix) and '`0x0D 0x0A`' (Windows). Ill-placed line break characters might result in wrong empty lines, which, however, must not result in misinterpretation of the content of the file.

Encoding integer numbers as text is done as follows: an optional minus sign '-' is directly followed by any number (at least one) of digits. Leading zeros are permitted, but not recommended. Any other characters (Group separators, spaces, etc.) are not allowed. If the format descriptions speaks of sized integers (e.g. Int16) this refers to the range of representable numbers and does not directly influence the formatting.

Encoding float numbers as text is done as follows: an optional minus sign '-' is directly followed by any number of digits, optionally followed by a dot '.' as decimal separator, followed by any number of digits, optionally followed by one of the characters ('e' or 'E'), followed by '+' or '-' followed by any number of digits (at least one). Leading zeros are permitted, but not recommended. Any other characters (Group separators, spaces, etc.) are not allowed. Especially symbols like "inf" or "nan" are not allowed.

Separating words in a text representation is done by white spaces. It is recommended only to use the "normal" space (`0x20`). Other spaces (e.g. non-breaking space) or tabs might be unsupported by some reading applications, and might result in undefined behavior.

2.1 Data File Structure

- Mmspd-Files start with a *format marker*,
- followed by a *header line*.
- After this header line, the *particle types* are defined.
- After this, the *particle values* for the individual time steps follow.

2.2 Type Definitions

The following types will be used by the file format (especially by the binary representation):

- [uint8/bool] is a 8-bit unsigned integer value (one byte), which only stores two relevant values: `0` – meaning 'false' and Non-Zero (`1` is recommended) – meaning 'true'. In Text format the strings 'true' and 'false' (case insensitive) can also be used to model these constants.
- [double] is a common 64-bit double-precision floating point value (IEEE specification)
- [uint32] is a common 32-bit unsigned integer value
- [uint64] is a common 64-bit unsigned integer value
- [string] is a String. In text format, this is any sequence of non-whitespace characters. In binary format, this is a zero-terminated 7-bit ASCII string.
- [X] is a value depending on other specifications. Here: it depends on the value of 'typeID'.

3 Format Marker

The first 24 bytes (or less) of the file form the format marker, identifying the mmspd-file format. A reading application should read these 9 bytes, and then re-seek depending on the encountered marker:

Bytes	Length	Text	Description
`4D 4D 53 50 44 62`	6	MMSPDb	Marker for binary stored data (endieness is checked with later field)
`4D 4D 53 50 44 61`	6	MMSPDa	Marker for 7-Bit Ascii
`4D 4D 53 50 44 75`	6	MMSPDu	Marker for UTF-8, without BOM
`EF BB BF 4D 4D 53 50 44 75`	9	MMSPDu	Marker for UTF-8, with BOM

When the binary data format marker is encountered, the next field of the format marker begins exactly at byte 6. When any other format marker is encountered (text), a reading application should set up the correct text decoding and re-start reading from the beginning of the file, as the format marker will then be the first text line.

3.1 Binary File Format Marker

Directly following the format marker (at byte 6) two padding bytes follow with the values `0x00` and `0xFF`.

The next four bytes represent an unsigned int (32) of the value `2018915346u` resulting in either `0x12 34 56 78` (little endian) or `0x78 56 34 12` (big endian). These bytes are used to identify the endianess of the file in binary mode.

The next four bytes represent two unsigned int (16) values, which specify the major and the minor version of the format.

These numbers must be `1` and `0` for now.

The last four bytes of the header should have the values of 128 or more (e.g. `0x8C 9D AE BF`). These bytes are used for padding, so that the header is 20 bytes in size, as well as to surely mark the file as binary file, e.g. for file transportation applications like FTP.

Example:

```
4D 4D 53 50 44 62 00 FF 12 34 56 78 00 01 00 00 81 82 83 84
```

3.2 Text File Format Marker

The first text line of the file represents the format marker. The first word is the format marker described above, followed by white spaces (at least one).

The second word is an unsigned integer, directly followed by a dot, directly followed by an unsigned integer, altogether specifying the format's major and minor version number.

These numbers must be "`1.0`" for now.

The rest of the line must not hold any characters except for white spaces. The next line must directly start the first block of the file (i.e. the header block).

Example:

```
MMSPDu 1.0
```

4 Header Line

The header line is a sequence of 10 values. In text format, these values are separated by one or more white-space characters. All values must be present. The type of the values (in square brackets) is relevant for the binary file format and the way of parsing text data (floating point or integer):

1. [uint8/bool] hasIDs – Flag whether or not the particles explicitly store ID values
2. [double] minx – The minimum value of the data extents in X direction. This value specifies the bounding box of the data set.
3. [double] miny – The minimum value of the data extents in Y direction. This value specifies the bounding box of the data set.
4. [double] minz – The minimum value of the data extents in Z direction. This value specifies the bounding box of the data set.
5. [double] maxx – The maximum value of the data extents in X direction. This value specifies the bounding box of the data set.
6. [double] maxy – The maximum value of the data extents in Y direction. This value specifies the bounding box of the data set.
7. [double] maxz – The maximum value of the data extents in Z direction. This value specifies the bounding box of the data set.
8. [uint32] timeCount – The number of time steps present in the data. The value must be at least 1.
9. [uint32] typeCount – The number of particle types defined in the file. This value must be at least 1. If this value is larger than 1, particles have to store their type explicitly.

10. [uint64] particleCount – The number of particles per time step. If this value is 0, the number of particles can change for each time step. If it is any other number, the particle count must not change for each time step.

The fields minx, miny, minz, maxx, maxy, maxz define the data set bounding box. The reference points (e.g. mid-points) of all particles will always be inside this area. If the particle representations have a size themselves (e.g. spheres with a non-zero radius), then these objects may partially be outside the bounding box area.

4.1 Example

The following header line specifies particles without explicit ID (meaning that are not traceable over time). There will only be a single particle type, meaning that no type values will be stored for each particle, and that particles will all be of type zero. All particles will be inside the bounding box (-100, -100, -100)-(100, 100, 100). The will only be a single data frame and it will store one million particles.

```
0 -100 -100 -100 100 100 100 1 1 1000000
```

5 Particle Definitions

The particle definitions directly follow the header line. In text format, each particle definition is stored in a single line. The particle types are implicitly numbered (zero-based) by their order, meaning particle type 0 is the first particle data type specified. A particle definition consists of a base type and a number of data fields, which have global values or which will have values for each particle.

There are two special data fields ‘id’ and ‘type’. These represent the particle ID and the particle type. Both fields will be present only if the corresponding header fields have the right values (i. e.: hasIDs is non-zero; typeCount has a value larger than 1). Regardless, these fields will never be present in the particle type definitions!

A particle type definition is a sequence of multiple values:

1. [string] baseType – The particle base type
2. [uint32] fixFieldCount – The number of particle data fields which are constant for all particles of this type
3. [uint32] varFieldCount – The number of particle data field which vary for each particle

then, for each fixed particle data field:

1. [string] fieldID – The value field identifier
2. [string] typeID – The value type identifier
3. [x] value – The value for this field

then, for each variable particle data field:

1. [string] fieldID – The value field identifier
2. [string] typeID – The value type identifier

5.1 Particle Base Types

The following identifiers can be used for particle base types (case insensitive). You can use the one-character identifier of the short-string identifier. Using one-character identifiers is recommended:

- d - Dot – Particles are interpreted as dots (single pixels). The fields ‘x’, ‘y’, and ‘z’ are required. The color fields ‘cr’, ‘cg’, and ‘cb’ are likely to be evaluated.
- s - Sphere – Particles are rendered as spheres. The fields ‘x’, ‘y’, and ‘z’ are required. The field ‘r’ will be interpreted as sphere radius (a default value of 0.5 is used if ‘r’ is missing). The color fields ‘cr’, ‘cg’, and ‘cb’ are likely to be evaluated.
- e - Ellipsoid – Particles are interpreted as ellipsoids. The fields ‘x’, ‘y’, and ‘z’ are required. The fields ‘rx’, ‘ry’, and ‘rz’ will be used as radii of the ellipsoid along its main axes. If any one of

these is missing, the value of field 'r' is used. If 'r' is also missing, a default value of 0.5 is used. The fields 'qi', 'qj', 'qk' and 'qr' are used as orientation (rotation) quaternion, if all four values are present. The color fields 'cr', 'cg', and 'cb' are likely to be evaluated.

- c - Cylinder – Particles are interpreted as cylinders. The fields 'x', 'y', and 'z' are required. The field 'r' will be interpreted as radius of the cylinder (a default value of 0.5 is used if 'r' is missing). The color fields 'cr', 'cg', and 'cb' are likely to be evaluated. The fields 'dx', 'dy', and 'dz' will be evaluated as direction of the cylinder. The cylinder will then be spanned between the points (x, y, z) and (x + dx, y + dy, z + dz). (A default direction vector of (1, 0, 0) will be assumed if one of these fields is missing).

5.2 Field Identifiers

The field identifiers, used in the particle type definitions, are short, machine-readable strings. Any string without white spaces can be used. Note that no field identifier must be present more than once. The following field identifiers are especially interpreted (case sensitive):

- x – The x position of a particle (mandatory)
- y – The y position of a particle (mandatory)
- z – The z position of a particle (mandatory)
- r – Radius of a particle (used for presenting the particle as spheres; optional); if missing a value of 0.5 is assumed.
- rx – Radius of a particle in x direction (used for presenting the particle as ellipsoid; optional); if missing the value of 'r' is used, or a value of 0.5 is assumed if 'r' is also missing.
- ry – Radius of a particle in y direction (used for presenting the particle as ellipsoid; optional); if missing the value of 'r' is used, or a value of 0.5 is assumed if 'r' is also missing.
- rz – Radius of a particle in z direction (used for presenting the particle as ellipsoid; optional); if missing the value of 'r' is used, or a value of 0.5 is assumed if 'r' is also missing.
- cr – The red color component for the particle (in the range of 0 – 1 (float) or 0 – 255 (byte); might get clamped to that range; optional); if missing a value of 0.75 is assumed.
- cg – The green color component for the particle (in the range of 0 – 1 (float) or 0 – 255 (byte); might get clamped to that range; optional); if missing a value of 0.75 is assumed.
- cb – The blue color component for the particle (in the range of 0 – 1 (float) or 0 – 255 (byte); might get clamped to that range; optional); if missing a value of 0.75 is assumed.
- qi – The imaginary i component of the rotation quaternion. This is only evaluated if 'qi', 'qj', 'qk', and 'qr' are present. The quaternion of these four values will be normalized when loading.
- qj – The imaginary j component of the rotation quaternion. This is only evaluated if 'qi', 'qj', 'qk', and 'qr' are present. The quaternion of these four values will be normalized when loading.
- qk – The imaginary k component of the rotation quaternion. This is only evaluated if 'qi', 'qj', 'qk', and 'qr' are present. The quaternion of these four values will be normalized when loading.
- qr – The real component of the rotation quaternion. This is only evaluated if 'qi', 'qj', 'qk', and 'qr' are present. The quaternion of these four values will be normalized when loading.
- dx – The x component of a direction vector for the particle
- dy – The y component of a direction vector for the particle
- dz – The z component of a direction vector for the particle

The direction vector fields 'dx', 'dy', and 'dz' are meant to define any generic particle related vector. This can be an orientation, a speed vector, a force vector, etc. Any other vectors can be defined with any other field names, e.g. 'vx', 'vy', and 'vz'.

The names 'id' and 'type' are reserved for special, internal usage and must not be used.

5.3 Type Identifiers

Type identifiers are small machine-readable strings. The following type identifiers are supported (other values will result in loading error). Use either the one-character strings, or the short strings to specify the type. Using one-character identifiers is recommended.

- b - byte – 8-bit unsigned integer [0...255]

- f - float – 32-bit single-precision float value (IEEE)
- d - double – 64-bit double-precision float value (IEEE)

It is strongly recommended to use 'byte' (or 'b') only for the color fields 'cr', 'cg', and 'cb' and to use a floating-point type for all other fields.

5.4 Example

The following two lines specify two types:

```
S 4 3 cr b 255 cg b 255 cb b 0 r f 0.75 x f y f z f
E 3 10 cr f 1 cg f 0 cb f 0 x d y d z d rx f ry f rz f qi f qj f qk f qr f
```

The first type (Type 0) specifies yellow spheres (byte values) of radius 0.75. The spheres positions are given by float values per particle

The second type (Type 1) specifies red ellipsoids (float values). The ellipsoids positions are given in doubles, while the radii and the orientation quaternion is given as floats.

6 Particle Data

The particle data is a sequence of time frames, each being a sequence of particles. Each time frame is started by a marker line.

6.1 Time Frame Marker

The time frame marker marks the beginning of a frame and specifies the number of particles in this frame. If the header field 'particleCount' had a value different from 0, the number of particles specified by each frame must be the same number as in the header field. Only if the header field 'particleCount' had a value of 0, the number of particles might be different for each frame.

In text format, the time frame marker is a line started by the greater-than character '>'. Separated by white space the number of particles in this frame follows. All further trailing characters will be ignored. If the loader expects a time frame marker but the loaded text line did not start with a greater-than character, all lines are ignored until a loaded line starts with the hash character.

In binary format, the time frame marker is only the number of particles in this frame, stored as [uin64]. The actual particle data is then directly following these eight bytes.

6.2 Example Time-Frame Marker

The following time-frame marker starts a time frame with four particles:

```
> 4
```

6.3 Particle List

In the particle list, all values for all value fields of the particles are defined. The list directly follows the time-frame marker. In text format, each particle is stored in a text line of its own, and the values are white space separated. The type for the values is given by the type identifiers specified in the particle type definitions.

If the file header field 'hasIDs' is true, each particle additionally stores a value of type [uint64] as first field, specifying the unique ID of this particle.

If the header field 'typeCount' has a value larger than 1, each particle also stores a value of type[uint32] before all other fields (except for the id field), which tells which type definition to use for this particle. If 'typeCount' is 1, all particles will use this one and only type, without every specifying it.

6.4 Example

The following particle data assumes 'hasIDs' as false and 'typeCount' to be 2, as well as 'particleCount' to be zero. The specified time frame has four particles. The particle type 0 is a sphere with the fields 'x', 'y', and 'z' being variable as floats. The particle type 1 is an ellipsoid with the fields 'x', 'y', 'z', 'rx', 'ry', 'rz', 'qi', 'qj', 'qk', and 'qr' being variable. Those types are similar to the example showing for Particle Definitions: All but the second particle are spheres (type 0). The second particle is an ellipsoid:

```
> 4
0 55.65 -24.3391 0.0012
1 90 85.75 0.25 10.0 5.5 2.75 0 0 0 1
0 -12.0 0 0
0 -99.5 -99.5 -99.5
```

7 Simple Example

This is a simple example file of two yellow spheres moving around in a simple file with four time frames:

```
MMSPDa 1.0
0 -10 -10 -10 10 10 10 4 1 2
s 4 3 r f 0.5 cr f 1 cg f 1 cb f 0 x f y f z f
> 2  These trailing character will be ignored
5.5 0 0
0 0 9.25
This line will get ignored, because two particles were read and the loader now
   searches for the 'time frame marker'
> 2
0 6.5 0
7.25 0 0
> 2
-5.5 0 0
0 0 -5.25
> 2
0 -6.5 0
-7.25 0 0
```

Note that the two comments about ignored characters only indicate two possible positions where characters can be added without breaking the file format. It is strongly recommended to **not** use these areas.

Other example data files might be available by contacting the MegaMol developers.