Geometry Processing

June 1, 2017

Contents

1 AVector

1.1 Summary

Efficiently store and retrieve numerically indexed static data vectors of variable length. An `AVector{T}` is a data type consisting of an anchor `Vector{Int}` and a data `Vector{T}`. The elements of the anchor vector are indices to the starting positions of segments of the data vector.

- **Constructor** `AVector`

  Receives segmented data as a `Vector{Vector{T}}` and constructs the corresponding `AVector{T}` for it.

  ```
  av = AVector([[1, 2, 3, 4], [5, 6], [7, 8, 9]])
  ```

```
GeometryProcessing.AVector{Int64}
Anchors
[1,5,7,10]
Vector
[1,2,3,4,5,6,7,8,9]
```

- **Reader** `av[<index>]`

  Returns the `i` th segment of `av` 's data vector.

  ```
  av[1], av[2], av[3]

  ([1,2,3,4],[5,6],[7,8,9])
  ```

2 LArray

2.1 Summary

`LArray` is a wrapper type to multidimensional Julia `Array`, created to efficiently address and manipulate them specifying dimensions with symbolic labels instead of plain numbers, thus achieving superior expressivity and clarity. An `LArray{T, N}` is a data type consisting of an (multidimensional) data `Array{T, N}` together with a `Vector{Symbol}` of length `N`. The symbols serve as labels for the dimensions. A configuration is a `Vector` of `Symbol` and/or `Vector{Symbol}` elements, which specify an arrangement of dimensions. Every function described below is non-destructive.

- **Field** `array::Array{T, N}`

  Contains the underlying native multidimensional array.

- **Field** `labels::Vector{Symbol}`

  Contains the labels for the dimensions of the `array`.

- **Constructor** `LArray`

  Accepts as input an `Array{T, N}` and a `Vector{Symbol}` of length `N` and returns an `LArray{T, N}`. Starting with a 3D array `a`:

  ```
  a = zeros(2, 3, 4); for i in 1:length(a) a[i] = i end; a

  2×3×4 Array{Float64,3}:
  [:, :, 1] =
   1.0  3.0  5.0
   2.0  4.0  6.0

  [:, :, 2] =
   7.0   9.0  11.0
   8.0  10.0  12.0
  ```

```
[:, :, 3] =
 13.0  15.0  17.0
 14.0  16.0  18.0

[:, :, 4] =
 19.0  21.0  23.0
 20.0  22.0  24.0
```

construct an `LArray` `la` with dimensions labeled by the (arbitrary) symbols `:a`, `:b`, `:c`:

```
la = LArray(a, [:a, :b, :c])
```

```
GeometryProcessing.LArray{Float64,3}
2×3×4 Array{Float64,3}
[1.0 3.0 5.0; 2.0 4.0 6.0]

[7.0 9.0 11.0; 8.0 10.0 12.0]

[13.0 15.0 17.0; 14.0 16.0 18.0]

[19.0 21.0 23.0; 20.0 22.0 24.0]
Symbol[:a,:b,:c]
```

- **Accessor** `la[<index_specifiers>...]`

  Access (read and write) to `la` is provided in two ways:

  1. Transparently to the underlying array via index specifiers (matching by position).

     ```
     la[2, [1, 3], 1:2:4]
     ```

     ```
     2×2 Array{Float64,2}:
      2.0  14.0
      6.0  18.0
     ```

  2. Through tuple pairs of labels and index specifiers.

     ```
     la[(:b, [1, 3]), (:c, 1:2:4), (:a, 2)]
     ```

     ```
     GeometryProcessing.LArray{Float64,2}
     2×2 Array{Float64,2}
     [2.0 14.0; 6.0 18.0]
     Symbol[:b,:c]
     ```

     In this case, the result is an LArray, where singleton dimensions (accessed by single numerical index) are eliminated. Especially in the case of a scalar value, the value itself is accessed:

```
la[(:b, 1), (:c, 1), (:a, 2)]
```

```
2.0
```

Assignment is achieved by the conventional notation `la[<index_specifiers>...] = x`.

```
la_copy = deepcopy(la); la_copy[(:c, 1), (:a, 2)] = [33, 66, 99]; la_copy
```

```
GeometryProcessing.LArray{Float64,3}
2×3×4 Array{Float64,3}
[1.0 3.0 5.0; 33.0 66.0 99.0]

[7.0 9.0 11.0; 8.0 10.0 12.0]

[13.0 15.0 17.0; 14.0 16.0 18.0]

[19.0 21.0 23.0; 20.0 22.0 24.0]
Symbol[:a,:b,:c]
```

2.2 Interface

- **Function** `Base.permutedims(la::LArray, conf::Vector{Symbol})`

  Permute the dimensions of `la` according to the supplied configuration `conf`.

```
permutedims(la, [:c, :a, :b])
```

```
GeometryProcessing.LArray{Float64,3}
4×2×3 Array{Float64,3}
[1.0 2.0; 7.0 8.0; 13.0 14.0; 19.0 20.0]

[3.0 4.0; 9.0 10.0; 15.0 16.0; 21.0 22.0]

[5.0 6.0; 11.0 12.0; 17.0 18.0; 23.0 24.0]
Symbol[:c,:a,:b]
```

- **Function** `arrange_with_inverse(la::LArray, conf::Vector)`

  Return a lower-dimensional layout of the data array together with a function that reconstructs the given `la::LArray` from it. The `conf` argument is a vector of `Symbol` and/or `Vector{Symbol}` (which dictate the order and fusion of dimensions in the resulting array). For example the configuration `[[:c, :a], [:b :d]]` for an `la::LArray` with `la.labels = [:a, :b, :c]` is equivalent to `reshape(la.array, size(la.array, 3)*size(la.array, 1), size(la.array, 2))`. The superfluous `:d` label is discarded, i.e. it's equivalent to the configurations `[[:c, :a], [:b]]` and `[[:ca, :a], :b]`.

```
arranged, inflate = arrange_with_inverse(la, [:b, [:c, :a]]); arranged
```

```
3×8 Array{Float64,2}:
 1.0   7.0  13.0  19.0  2.0   8.0  14.0  20.0
 3.0   9.0  15.0  21.0  4.0  10.0  16.0  22.0
 5.0  11.0  17.0  23.0  6.0  12.0  18.0  24.0
```

The 2nd result of `arrange_with_inverse` (bound to `inflate` here) is a function implementing the reverse transformation of the `arranged` array back to `la`.

```
inflate(arranged)
```

```
GeometryProcessing.LArray{Float64,3}
2×3×4 Array{Float64,3}
[1.0 3.0 5.0; 2.0 4.0 6.0]

[7.0 9.0 11.0; 8.0 10.0 12.0]

[13.0 15.0 17.0; 14.0 16.0 18.0]

[19.0 21.0 23.0; 20.0 22.0 24.0]
Symbol[:a,:b,:c]
```

- **Function** `arrange(la::LArray, conf::Vector)`

  Same as `arrange_with_inverse`, but return only the arranged `Array`.

- **Function** `Base.size(la::LArray, label::Symbol)`

  Return the size of `la` across the specified dimension.

  ```
  size(la, :b)
  ```

  ```
  3
  ```

- **Function** `Base.size(la::LArray, labels::Vararg{Symbol})`

  Return the product of the sizes of `la` across the specified dimensions.

  ```
  size(la, :a, :c)
  ```

  ```
  8
  ```

- **Function** `divide(la::LArray, label::Symbol, block_sizes::Vector{Int})`

  Divide `la` across the specified dimension to blocks of the supplied sizes.

  ```
  divide(la, :c, [1, 2, 1])
  ```

```
3-element Array{GeometryProcessing.LArray{Float64,3},1}:
 GeometryProcessing.LArray{Float64,3}
2×3×1 Array{Float64,3}
[1.0 3.0 5.0; 2.0 4.0 6.0]
Symbol[:a,:b,:c]

 GeometryProcessing.LArray{Float64,3}
2×3×2 Array{Float64,3}
[7.0 9.0 11.0; 8.0 10.0 12.0]

[13.0 15.0 17.0; 14.0 16.0 18.0]
Symbol[:a,:b,:c]

 GeometryProcessing.LArray{Float64,3}
2×3×1 Array{Float64,3}
[19.0 21.0 23.0; 20.0 22.0 24.0]
Symbol[:a,:b,:c]
```

- **Function** `divide(la::LArray, label::Symbol, block_size::Int)`

  Divide `la` across the specified dimension to blocks of the supplied size (last block has size `mod(size(la, :label), block_size)`.

```
divide(la, :b, 2)
```

```
2-element Array{GeometryProcessing.LArray{Float64,3},1}:
 GeometryProcessing.LArray{Float64,3}
2×2×4 Array{Float64,3}
[1.0 3.0; 2.0 4.0]

[7.0 9.0; 8.0 10.0]

[13.0 15.0; 14.0 16.0]

[19.0 21.0; 20.0 22.0]
Symbol[:a,:b,:c]

 GeometryProcessing.LArray{Float64,3}
2×1×4 Array{Float64,3}
[5.0; 6.0]

[11.0; 12.0]

[17.0; 18.0]

[23.0; 24.0]
Symbol[:a,:b,:c]
```

- **Function** `Base.cat{T, N <: Any}(label::Symbol, las::Vector{LArray{T, N}})`

  Concatenate the contents of `las::Vector{LArray{T, N}}` across the specified dimension.

  ```
  cat(:c, divide(la, :c, 3))
  ```

  ```
  GeometryProcessing.LArray{Float64,3}
  2×3×4 Array{Float64,3}
  [1.0 3.0 5.0; 2.0 4.0 6.0]

  [7.0 9.0 11.0; 8.0 10.0 12.0]

  [13.0 15.0 17.0; 14.0 16.0 18.0]

  [19.0 21.0 23.0; 20.0 22.0 24.0]
  Symbol[:a,:b,:c]
  ```

- **Function** `Base.cat{T, N <: Any}(label::Symbol, las::Vararg{LArray{T, N}})`

  Concatenate the supplied `LArray` arguments across the specified dimension.

  ```
  cat(:c, divide(la, :c, 3)...)
  ```

  ```
  GeometryProcessing.LArray{Float64,3}
  2×3×4 Array{Float64,3}
  [1.0 3.0 5.0; 2.0 4.0 6.0]

  [7.0 9.0 11.0; 8.0 10.0 12.0]

  [13.0 15.0 17.0; 14.0 16.0 18.0]

  [19.0 21.0 23.0; 20.0 22.0 24.0]
  Symbol[:a,:b,:c]
  ```

3 Partitioning

3.1 Summary

Represents a partitioning of a set of vertex indices.

- **Field** `vp_inds::Vector{Int}`

  A `Vector` containing at each index `i` the index of the partition to which the vertex v_i belongs.

- **Field** `partitions::AVector{Int}`

  An `AVector` containing each partition's vertex indices for all partitions.

- **Constructor** `Partitioning(filename::String)`

  Construct a partitioning from a text file, where each line i contains the index of the partition to which the vertex v_i belongs.

3.2 Interface

- **Function** `vertex_partition(prt::Partitioning, vi::Int)`

  Return the index of the partition containing vertex `vi` in the partitioning `prt`.

- **Function** `partition_vertices(prt::Partitioning, pi::Int)`

  Return the indices of the vertices contained in partition `pi` in the partitioning `prt`.

- **Function** `is_edge(conn::Connectivity, prt::Partitioning, vi::Int)`

  Predicate to test if the given vertex with index `vi` is an edge vertex (of a partition) in a connectivity `conn` and partitioning `prt`. This function essentially tests whether any of the vertex's neighbours belongs to a different partition.

- **Function** `edge_vertices(conn::Connectivity, prt::Partitioning, pi::Int)`

  Return the edge vertices of the partition with index `pi` in connectivity `conn` and paritioning `prt`.

4 Connectivity

5 File processing

6 Animation

6.1 Summary

`Animation` is a data type describing an animation in space and time.

- **Field** `data::LArray{Float, 3}`

  A 3D `LArray` with dimension configuration `[:coors, :vertices, :frames]`, containing x, y, z coordinates at each frame of each vertex of the animated model.

- **Field** `vertex_data::Dict{Symbol, LArray}`, `face_data::Dict{Symbol, LArray}`

  Two dictionaries containing vertex and face data, respectively. Data is in the form of an `LArray` with labels `[:data, <element>]` in the case of frame-invariant data or `[:data, <element>, :frames]` otherwise (`element = {:vertices | :faces}`).

- **Field** `fv_inds::LArray{Int, 2}`

  An `LArray` with dimension labels `[:v_inds, :faces]`, holding the vertex indices of each triangular face of the animated model (assumed to be static).

- **Field** `conn::Connectivity`

  A connectivity (see 4) object encoding the neighbours of each vertex of the animated model (the set of vertices with which it is connected via edges).

- **Constructor** `Animation(directory::String, name::String, ext::String)`

  This constructor searches the specified `directory` for files for which `name` matches a substring of their name and `ext` match their extension. These files are by default sorted if there is numbering between the matched name and the extension. The files are then read (each describing a frame of the animation) and a new Animation object is created, with the fields `data`, `fv_inds` and `conn` appropriately initialized (and empty `Dict{Symbol, LArray}` for `vertex_data` and `face_data` fields).

6.2 Interface

- **Function** `export_animation_pvd(anm::Animation, dir::String, name::String)`

  Export the `anm::Animation` in the specified directory `dir` as a ParaView Data (PVD) file format named after `name`.

- **Function** `export_animation_obj(anm::Animation, dir::String, name::String)`

  Export the `anm::Animation` in the specified directory `dir` as a collection of Wavefront OBJ files named after `name` (sequentially numbered per frame).

- **Function** `anm_conf(conf_name::Symbol)`

  Return a label configuration

  ```
  anm_conf(:eg)
  ```

  ```
  2-element Array{Array{Symbol,1},1}:
   Symbol[:coors,:frames]
   Symbol[:vertices]
  ```

- **Function** `neighbours(anm::Animation, vi::Int)`

  Return the neighbour vertex indices of the vertex with index `vi` in the animation `anm`.

- **Function** `adjacency_matrix(anm::Animation)`

  Return the adjacency matrix of the animation `anm` connectivity.

- **Function** `edge_vertices(anm::Animation, prt::Partitioning, pi::Int)`

  Return the edge vertices of the partition with index `pi` in the animation `anm` according to partitioning `prt`.

- **Function** `edge_vertices(anm::Animation, prt::Partitioning)`

  Return the edge vertices of all partitions in an animation `anm` according to partitioning `prt`.

- **Function** `add_vertex_data!(anm::Animation, name::Symbol, data::Array)`

  Add or replace the vertex data under the specified `name` in animation `anm`. The `data` is interpreted differently depending on its size and dimensions (in any case, one of them must match the number of vertices):

1D Interpreted as frame-invariant scalar values.

2D If the other dimensions's size matches the number of frames, data are interpreted as frame-variant scalar values, otherwise as frame-invariant vector values.

3D Treated as frame-variant vector values (one dimension's size must match frame count).

Independent of order, the `data` array's dimensions will be appropriately permuted and labeled with labels `:data`, `:vertices` and `:frames`.

- **Function** `add_face_data!(anm::Animation, name::Symbol, data::Array)`

  Same as previous, but for face instead of vertex data (labeled with `:faces` at the end).

- **Function** `remove_vertex_data!(anm::Animation, name::Symbol)`

  Remove vertex data under the specified `name` from animation `anm`.

- **Function** `remove_face_data!(anm::Animation, name::Symbol)`

  Remove face data under the specified `name` from animation `anm`.