tile-renderer

Release v0.8

7d

Apr 07, 2021

CONTENTS:

Tile renderer for leaflet.js usage, made by 7d. Git repo here

CHAPTER

ONE

CHANGELOG

- **v0.8 (7/4/21)**
  - Text of points are now rendered together with texts of lines and areas
  - reordered rendering of PLAs (excluding road tag & text) into functions from if statements
  - got rid of most `**kwargs`
  - redid integrity checking, mostly with Schema
  - new function: `renderer.utils.skinJsonIntegrity()`
  - background of tile can now be customised by skin file
  - added offset to area centertext
  - added centerimage to areas
- **v0.7 (6/4/21)**
  - new `nodeJsonBuilder.py`, intended for use as an assistance for marking nodes on Minecraft
  - fixed `renderer.tools.lineToTiles()`
  - processing and rendering now show ETA
  - fixed oneway roads showing too many arrows
  - added support for lines with unrounded ends through `unroundedEnds` tag
  - updated `renderer.mathtools.dash()` to support offset
  - added `renderer.mathtools.dashOffset()`
  - fixed dashed roads
  - bounding boxes on texts so they don't overlap
  - new logging function (`renderer.internal.log()`)
    * `renderer.render()` has new `verbosityLevel` optional argument, defaults to 1
  - estimated that last beta release before v1.0 is v0.8 or v0.9
- **v0.6 (11/3/21)**
  - added loads of PLAs to the default skin; there are now about 90 different PLA types :))
  - tweaked `renderer.mathtools.midpoint()` a bit
  - new functions: `renderer.mathtools.polyCenter()`, `renderer.mathtools.dash()`
  - Moved `renderer.tools.lineInBox()` to `renderer.mathtools.lineInBox()`

  - fixed layers
  - image size is now customisable
    - default skin tile size is now 2048 from 1024
  - added one-way roads
  - added dashed roads, but they're a bit broken right now
  - multiple texts can now be shown on a single line/border
  - improved area centertext; it should now render in the correct center
  - *screams in agony again*
- **v0.5 (28/2/21)**
  - "shape" key in PLA structure removed
  - A Roads, B Roads, local main roads, and simplePoint added to default skin
  - New font for renders (Clear Sans), will be customisable later on
  - Added functions `renderer.mathtools.midpoint()`, `renderer.mathtools.linesIntersect()`, `renderer.mathtools.pointInPoly()`, `renderer.tools.lineInBox()`, `renderer.tools.lineInBox()`, `findPlasAttachedToNode()`
  - Not every info printout is green now; some are white or gray
  - `renderer.render()` now able to render:
    - points
    - text on lines
    - text on borders of areas
    - text in center of areas
    - joined roads
  - ahhh
- v0.4.1 (24/2/21)
  - renderer creates new "tiles" directory to store tiles if directory not present
- **v0.4 (24/2/21)**
  - PLA processing: grouping now only works for lines with "road" tag
  - `renderer.render()` now able to render lines and areas
  - New default skin; simpleLine and simpleArea PLA types added
- **v0.3 (23/2/21)**
  - PLA processing for `renderer.render()`
- **v0.2 (15/2/21)**
  - Added functions:
    - `renderer.utils.coordListIntegrity()`
    - `renderer.utils.tileCoordListIntegrity()`
    - `renderer.utils.nodeJsonIntegrity()`
    - `renderer.utils.plaJsonIntegrity()`

    * `renderer.utils.nodeListIntegrity()`
    * `renderer.internal.tupleToStr()`
    * `renderer.internal.strToTuple()`
    * `renderer.internal.readJson()`
    * `renderer.internal.writeJson()`
    * `renderer.tools.nodesToCoords()`
    * `renderer.tools.plaJson_findEnds()`
    * `renderer.tools,plaJson_calcRenderedIn()`
  - added more to `renderer.render()`: sorts PLA into tiles now

- **v0.1 (13/2/21)**
  - two new functions: `renderer.tools.coordToTiles()` and `renderer.tools.lineToTiles()`
  - moved renderer input format documentation to docs page
- v0.0.1 (11/2/21)
  - just a quickie
  - updated input format and added json reading code for test.py
  - added minzoom, maxzoom, maxzoomrange for `renderer.render()`
- **v0.0 (8/2/21)**
  - started project
  - documented JSON dictionary structure

CHAPTER

TWO

ALL FUNCTIONS

Useful information

- To convert tuple to list efficiently use *(tuple)
- PLA = Points, Lines and Areas

2.1 Main

render(*plaList: dict*, *nodeList: dict*, *skinJson: dict*, *minZoom: int*, *maxZoom: int*, *maxZoomRange: int*[, *verbosityLevel=1*, *saveImages=True*, *saveDir="tiles/"*, *assetsDir="skins/assets/"*, *tiles: list*])

Renders tiles from given coordinates and zoom values.

Parameters

- dict **plaList**: a dictionary of PLAs (see "Renderer input format")
- dict **nodeList**: a dictionary of nodes (see "Renderer input format")
- dict **skinJson**: a JSON of the skin used to render tiles
- int **minZoom**: minimum zoom value
- int **maxZoom**: maximum zoom value
- int **maxZoomRange**: range of coordinates covered by a tile in the maximum zoom (how do I phrase this?) For example, a `maxZoom` of 5 and a `maxZoomValue` of 8 will make a 5-zoom tile cover 8 units
- int **verbosityLevel** *(default: 1)*: the verbosity level of the output by the function. Use any number from 0 to 2
- int **saveImages** *(default: True)*: whether to save the tile imaegs in a folder or not
- str **saveDir** *(default: "tiles/")*: the directory to save tiles in
- str **assetsDir** *(default: "skins/assets/")*: the asset directory for the skin
- list[tuple] **tiles** *(optional)*: a list of tiles to render, given in tuples of `(z,x,y)` where z = zoom and x,y = tile coordinates

Returns

- **list[Image]** A list of tiles as PIL Image objects.

2.2 Tools

`renderer.tools.`**`lineToTiles`**(*coords: list*, *minZoom: int*, *maxZoom: int*, *maxZoomRange: int*)

Generates tile coordinates from list of regular coordinates using `renderer.tools.coordToTiles()`. Mainly for rendering whole PLAs.

Parameters

- list[tuple] **coords** of coordinates in tuples of `(x,y)`
- int **minZoom**: minimum zoom value
- int **maxZoom**: maximum zoom value
- int **maxZoomValue**: range of coordinates covered by a tile in the maximum zoom (how do I phrase this?) For example, a `maxZoom` of 5 and a `maxZoomValue` of 8 will make a 5-zoom tile cover 8 units

Returns

- **list[tuple]** A list of tile coordinates

`renderer.tools.`**`coordToTiles`**(*coord: list*, *minZoom: int*, *maxZoom: int*, *maxZoomRange: int*)

Returns all tiles in the form of tile coordinates that contain the provided regular coordinate.

Parameters

- list[int/float] **coord**: Coordinates provided in the form `[x,y]`
- int **minZoom**: minimum zoom value
- int **maxZoom**: maximum zoom value
- int **maxZoomValue**: range of coordinates covered by a tile in the maximum zoom (how do I phrase this?) For example, a `maxZoom` of 5 and a `maxZoomValue` of 8 will make a 5-zoom tile cover 8 units

Returns

- **list[tuple]** A list of tile coordinates

`renderer.tools.`**`plaJson_calcRenderedIn`**(*plaList: dict*, *nodeList: dict*, *minZoom: int*, *maxZoom: int*, *maxZoomRange: int*)

Like `renderer.tools.lineToTiles()`, but for a JSON or dictionary of PLAs.

Parameters

- dict **plaList**: a dictionary of PLAs (see "Renderer input format")
- dict **nodeList**: a dictionary of nodes (see "Renderer input format")
- int **minZoom**: minimum zoom value
- int **maxZoom**: maximum zoom value
- int **maxZoomValue**: range of coordinates covered by a tile in the maximum zoom (how do I phrase this?) For example, a `maxZoom` of 5 and a `maxZoomValue` of 8 will make a 5-zoom tile cover 8 units

Returns

- **list[tuple]** A list of tile coordinates

`renderer.tools.`**`plaJson_findEnds`**(*plaList: dict*, *nodeList: dict*)

Finds the minimum and maximum X and Y values of a JSON or dictionary of PLAs.

Parameters

- dict **plaList**: a dictionary of PLAs (see "Renderer input format")

- dict **nodeList**: a dictionary of nodes (see "Renderer input format")

Returns

- **tuple** Returns in the form *(xMax, xMin, yMax, yMin)*

renderer.tools.**nodesToCoords**(*nodes: list*, *nodeList: dict*)
Converts a list of nodes IDs into a list of coordinates with a node dictionary/JSON as its reference.

Parameters

- list **nodes**: a list of node IDs
- dict **nodeList**: a dictionary of nodes (see "Renderer input format")

Returns

- **list[tuple]** A list of coordinates

renderer.tools.**findPlasAttachedToNode**(*nodeId: str*, *plaList: dict*)
Finds which PLAs attach to a node.

Parameters

- str **nodeId**: the node to search for
- dict **plaList**: a dictionary of PLAs (see "Renderer input format")

Returns

- **list[tuple]** A tuple in the form of (plaId, posInNodeList)

2.3 Math Tools

renderer.mathtools.**midpoint**(*x1*, *y1*, *x2*, *y2*, *o*[, *returnBoth=False*])
Calculates the midpoint of two lines, offsets the distance away from the line, and calculates the rotation of the line.

Parameters

- int/float **x1, y1, x2, y2**: the coordinates of two points
- int/float **o**: the offset from the line. If positive, the point above the line is returned; if negative, the point below the line is returned
- bool **returnBoth** *(default=False)*: if True, it will return both possible points.

Returns

- *returnBoth=False* **tuple** A tuple in the form of (x, y, rot)
- *returnBoth=True* **list[tuple]** A list of two tuples in the form of (x, y, rot)

renderer.mathtools.**linesIntersect**(*x1: Union[int, float]*, *y1: Union[int, float]*, *x2: Union[int, float]*, *y2: Union[int, float]*, *x3: Union[int, float]*, *y3: Union[int, float]*, *x4: Union[int, float]*, *y4: Union[int, float]*)
Finds if two segments intersect.

Parameters

- int/float **x1, y1, x2, y2**: the coordinates of two points of the first segment.
- int/float **x3, y3, x4, y4**: the coordinates of two points of the second segment.

Returns

- **bool** Whether the two segments intersect.

renderer.mathtools.**pointInPoly**(*xp: Union[int, float]*, *yp: Union[int, float]*, *coords: list*)
Finds if a point is in a polygon. **WARNING: If your polygon has a lot of corners, this will take very long.**

Parameters

- int/float **xp, yp**: the coordinates of the point.
- list **coords**: the coordinates of the polygon; give in (x,y)

Returns

- **bool** Whether the point is inside the polygon.

renderer.mathtools.**polyCenter**(*coords: list*)
Finds the center point of a polygon.

Parameters

- list **coords**: the coordinates of the polygon; give in (x,y)

Returns

- **tuple** The center of the polygon, given in (x,y)

renderer.mathtools.**lineInBox**(*line: list*, *top: Union[int, float]*, *bottom: Union[int, float]*, *left: Union[int, float]*, *right: Union[int, float]*)
Finds if any nodes of a line go within the box.

Parameters

- list **line**: the line to check for
- int/float **top, bottom, left, right**: the bounds of the box

Returns

- **bool** Whether any nodes of a line go within the box.

renderer.mathtools.**dash**(*x1: Union[int, float], y1: Union[int, float], x2: Union[int, float], y2: Union[int, float], d: Union[int, float] [, o=0, emptyStart=False]*)
Finds points along a segment that are a specified distance apart.

Parameters

- int/float **x1, y1, x2, y2**: the coordinates of two points of the segment
- int/float **d**: the distance between points
- int/float **o** *(default=0)*: the offset from (x1,y1) towards (x2,y2) before dashes are calculated
- bool **emptyStart** *(default=False)*: Whether to start the line from (x1,y1) empty before the start of the next dash

Returns

- **list[list[tuple]]** A list of points along the segment, given in [[(x1, y1), (x2, y2)], etc]

renderer.mathtools.**dashOffset**(*coords: list*, *d: Union[int, float]*)
Calculates the offsets on each coord of a line for a smoother dashing sequence.

Parameters

- list **coords**: the coords of the line
- int/float **d**: the distance between points

Returns

- **list[float]** The offsets of each coordinate

renderer.mathtools.**rotateAroundPivot**(*x: Union[int, float]*, *y: Union[int, float]*, *px: Union[int, float]*, *py: Union[int, float]*, *theta: Union[int, float]*)

Rotates a set of coordinates around a pivot point.

Parameters

- int/float **x, y**: the coordinates to be rotate
- int/float **px, py**: the coordinates of the pivot
- int/float **theta**: how many **degrees** to rotate

Returns

- **tuple** The rotated coordinates, given in (x,y)

2.4 Utilities

renderer.utils.**coordListIntegrity**(*coords: list*)

Checks integrity of a list of coordinates.

Parameters

- list **coords**: a list of coordinates.

Returns

- **bool** Returns True if no errors

renderer.utils.**tileCoordListIntegrity**(*tiles: list*, *minZoom: int*, *maxZoom: int*)

Checks integrity of a list of tile coordinates.

Parameters

- list **tiles**: a list of tile coordinates.
- int **minZoom**: minimum zoom value
- int **maxZoom**: maximum zoom value

Returns

- **bool** Returns True if no errors

renderer.utils.**nodeListIntegrity**(*nodes: list*, *nodeList: dict*)

Checks integrity of a list of node IDs.

Parameters

- list **nodes**: a list of node IDs.
- dict **nodeList**: a dictionary of nodes (see "Renderer input format")

Returns

- **bool** Returns True if no errors

renderer.utils.**nodeJsonIntegrity**(*nodeList: dict*)

Checks integrity of a dictionary/JSON of nodes.

Parameters

- dict **nodeList**: a dictionary of nodes (see "Renderer input format")

Returns

- **bool** Returns True if no errors

renderer.utils.**plaJsonIntegrity**(*plaList: dict*, *nodeList: dict*)
Checks integrity of a dictionary/JSON of PLAs.

Parameters

- dict **plaList**: a dictionary of PLAs (see "Renderer input format")
- dict **nodeList**: a dictionary of nodes (see "Renderer input format")

Returns

- **bool** Returns True if no errors

renderer.utils.**skinJsonIntegrity**(*skinJson: dict*)
Checks integrity of a skin JSON file.

Parameters

- dict **skinJson**: the skin JSON file

Returns

- **bool** Returns True if no errors

CHAPTER

THREE

FORMATS

3.1 PLAs

```
{
  "(nameid)": {
    "type": "(type)",
    "displayname": "(displayname)",
    "description": "(description)".
    "layer": layer_no,
    "nodes": [nodeid, nodeid, nodeid],
    "attrs": {
      "(attr name)": "(attr val)",
    // etc
    }
  },
  //etc
}
```

3.2 Nodes

(Note: Nodes != Points)

```
{
  "(nodeid)": {
    "x": x,
    "y": y,
    "connections": [
      {
        "nodeid": nodeid,
        "mode": nameid, //lines only
        "cost": cost, //lines only, time will be calculated from distance and speed
      },
      // etc
    ]
  }
}
```

Note: Connections is not implemented yet

3.3 Skins

```
{
    "info": {
        "size": size,
        "font": {
            "": "(tff file location in assets)",
            "b": "(tff file location in assets)",
            "i": "(tff file location in assets)",
            "bi": "(tff file location in assets)"
        },
        "background": [r, g, b]
    },
    "order": [
        "(type)",
        "(type)",
        // etc
    ],
    "types": {
        "(type-point)": {
            "tags": [],
            "type": "point",
            "style": {
                "(maxZ), (minZ)": [
                    {
                        "layer": "circle",
                        "colour": "(hex)" / null,
                        "outline": "(hex)" / null,
                        "size": size,
                        "width": width
                    },
                    {
                        "layer": "text",
                        "colour": "(hex)" / null,
                        "offset": [x, y],
                        "size": size,
                        "anchor": null / (anchor)
                    },
                    {
                        "layer": "square",
                        "colour": "(hex)" / null,
                        "outline": "(hex)" / null,
                        "size": size,
                        "width": width
                    },
                    {
                        "layer": "image",
                        "file": "(image file location in assets)",
                        "offset": [x, y]
                    }
                ],
                //etc
            }
        },
        "(type-line)": {
            "tags": [],
            "type": "line",
```

(continues on next page)

(continued from previous page)

```
            "style": {
                "(maxZ), (minZ)": [
                    {
                        "layer": "back",
                        "colour": "(hex)",
                        "width": width,
                        *"dash": 24 (Optional)*
                    },
                    {
                        "layer": "fore",
                        "colour": "(hex)",
                        "width": width,
                        *"dash": 24 (Optional)*
                    },
                    {
                        "layer": "text",
                        "colour": "(hex)",
                        "size": size,
                        "offset": offset
                    }
                ],
                //etc
            }
        },
        "(type-area)": {
            "tags": [],
            "type": "area",
            "style": {
                "0, 5": [
                    {
                        "layer": "fill",
                        "colour": "(hex)",
                        "outline": "(hex)"
                    },
                    {
                        "layer": "bordertext",
                        "colour": "(hex)",
                        "offset": offset,
                        "size": size
                    },
                    {
                        "layer": "centertext",
                        "colour": "(hex)",
                        "size": size,
                        "offset": [x,y]
                    },
                    {
                        "layer": "centerimage",
                        "file": "(image file location in assets)",
                        "offset": [x, y]
                    }
                ],
                //etc
            }
        }
    }
}
```

INDEX

B

R