Visualization

Mesa Visualization Module

TextVisualization: Base class for writing ASCII visualizations of model state.

TextServer: Class which takes a TextVisualization child class as an input, and renders it in-browser, along with an interface.

ModularServer

A visualization server which renders a model via one or more elements.

The concept for the modular visualization server as follows: A visualization is composed of VisualizationElements, each of which defines how to generate some visualization from a model instance and render it on the client. VisualizationElements may be anything from a simple text display to a multilayered HTML5 canvas.

The actual server is launched with one or more VisualizationElements; it runs the model object through each of them, generating data to be sent to the client. The client page is also generated based on the JavaScript code provided by each element.

This file consists of the following classes:

VisualizationElement: Parent class for all other visualization elements, with

the minimal necessary options.

PageHandler: The handler for the visualization page, generated from a template

and built from the various visualization elements.

SocketHandler: Handles the websocket connection between the client page and

the server.

ModularServer: The overall visualization application class which stores and

controls the model and visualization instance.

ModularServer should *not* need to be subclassed on a model-by-model basis; it should be primarily a pass-through for VisualizationElement subclasses, which define the actual visualization specifics.

For example, suppose we have created two visualization elements for our model, called canvasvis and graphvis; we would launch a server with:

> server = ModularServer(MyModel, [canvasvis, graphvis], name="My Model") server.launch()

The client keeps track of what step it is showing. Clicking the Step button in the browser sends a message requesting the viz_state corresponding to the next step position, which is then sent back to the client via the websocket.

The websocket protocol is as follows: Each message is a JSON object, with a "type" property which defines the rest of the structure.

Server -> Client:

Send over the model state to visualize. Model state is a list, with each element corresponding to a div; each div is expected to have a render function associated with it, which knows how to render that particular data. The example below includes two elements: the first is data for a CanvasGrid, the second for a raw text display.

{ "type": "viz_state", "data": [{0:[{"Shape": "circle", "x": 0, "y": 0, "r": 0.5,

"Color": "#AAAAAA", "Filled": "true", "Layer": 0, "text": 'A', "text_color": "white" }]},

"Shape Count: 1"]

}

Informs the client that the model is over. {"type": "end"}

Informs the client of the current model's parameters { "type": "model_params", "params": 'dict' of model params, (i.e. {arg_1: val_1, …}) }

Client -> Server:

Reset the model. TODO: Allow this to come with parameters { "type": "reset" }

Get a given state. { "type": "get_step", "step:" index of the step to get. }

Submit model parameter updates { "type": "submit_params", "param": name of model parameter "value": new value for 'param' }

Get the model's parameters { "type": "get_params" } *class* **VisualizationElement** [source]

Defines an element of the visualization.

Attributes:

package_includes: A list of external JavaScript and CSS files to

include that are part of the Mesa packages.

local_includes: A list of JavaScript and CSS files that are local to

the directory that the server is being run in.

js_code: A JavaScript code string to instantiate the element. local_dir: A full path to the directory containing the local includes.

If a relative path is given, it is relative to the working directory where the server is being run. If an absolute path is given, it is used as-is. Default is the current working directory.

Methods:

render: Takes a model object, and produces JSON data which can be sent

to the client.

render(*model*) [source]

Build visualization data from a model object.

Args:

model: A model object

Returns:

A JSON-ready object.

class **TextElement** [source]

Module for drawing live-updating text.

class **PageHandler(*application: Application, request: HTTPServerRequest, **kwargs: Any*)** [source]

Handler for the HTML template which holds the visualization.

class **SocketHandler(*application: Application, request: HTTPServerRequest, **kwargs: Any*)** [source]

Handler for websocket.

open() [source]

Invoked when a new WebSocket is opened.

The arguments to *open* are extracted from the *tornado.web.URLSpec* regular expression, just like the arguments to *tornado.web.RequestHandler.get*.

open may be a coroutine. *on_message* will not be called until *open* has returned.

Changed in version 5.1: `open` may be a coroutine.

check_origin(*origin*) [source]

Override to enable support for allowing alternate origins.

The `origin` argument is the value of the `Origin` HTTP header, the url responsible for initiating this request. This method is not called for clients that do not send this header; such requests are always allowed (because all browsers that implement WebSockets support this header, and non-browser clients do not have the same cross-site security concerns).

Should return `True` to accept the request or `False` to reject it. By default, rejects all requests with an origin on a host other than this one.

This is a security protection against cross site scripting attacks on browsers, since WebSockets are allowed to bypass the usual same-origin policies and don't use CORS headers.

Warning

This is an important security measure; don't disable it without understanding the security implications. In particular, if your authentication is cookie-based, you must either restrict the origins allowed by `check_origin()` or implement your own XSRF-like protection for websocket connections. See these articles for more.

To accept all cross-origin traffic (which was the default prior to Tornado 4.0), simply override this method to always return `True`:

```
def check_origin(self, origin):
    return True
```

To allow connections from any subdomain of your site, you might do something like:

```
def check_origin(self, origin):
    parsed_origin = urllib.parse.urlparse(origin)
    return parsed_origin.netloc.endswith(".mydomain.com")
```

New in version 4.0.

on_message(*message*) [source]

Receiving a message from the websocket, parse, and act accordingly.

class **ModularServer(*model_cls*, *visualization_elements*, *name='Mesa Model'*, *model_params=None*, *port=None*)** [source]

Main visualization application.

Args:

model_cls: Mesa model class visualization_elements: visualisation elements name: A String for the model name port: Port the webserver listens to (int)

> Order of configuration: 1. Parameter to ModularServer.launch 2. Parameter to ModularServer() 3. Environment var PORT 4. Default value (8521)

model_params: A dict of model parameters

settings

Create a new visualization server with the given elements.

reset_model() [source]

Reinstantiate the model object, using the current parameters.

render_model() [source]

Turn the current state of the model into a dictionary of visualizations

launch(*port=None*, *open_browser=True*) [source]

Run the app.

Text Visualization

Base classes for ASCII-only visualizations of a model. These are useful for quick debugging, and can readily be rendered in an IPython Notebook or via text alone in a browser window.

Classes:

TextVisualization: Class meant to wrap around a Model object and render it in some way using Elements, which are stored in a list and rendered in that order. Each element, in turn, renders a particular piece of information as text.

ASCIIElement: Parent class for all other ASCII elements. render() returns its representative string, which can be printed via the overloaded __str__ method.

TextData: Uses getattr to get the value of a particular property of a model and prints it, along with its name.

TextGrid: Prints a grid, assuming that the value of each cell maps to exactly one ASCII character via a converter method. This (as opposed to a dictionary) is used so as to allow the method to access Agent internals, as well as to potentially render a cell based on several values (e.g. an Agent grid and a Patch value grid).

class **TextVisualization(*model*)** [source]

ASCII-Only visualization of a model.

Properties:

model: The underlying model object to be visualized. elements: List of visualization elements, which will be rendered

in the order they are added.

Create a new Text Visualization object.

render() [source]

Render all the text elements, in order.

step() [source]

Advance the model by a step and print the results.

class **ASCIIElement** [source]

Base class for all TextElements to render.

Methods:

render: ‘Renders’ some data into ASCII and returns. __str__: Displays render() by default.

render() [source]

Render the element as text.

class **TextData**(*model*, *var_name*) [source]

Prints the value of one particular variable from the base model.

Create a new data renderer.

render() [source]

Render the element as text.

class **TextGrid**(*grid*, *converter*) [source]

Class for creating an ASCII visualization of a basic grid object.

By default, assume that each cell is represented by one character, and that empty cells are rendered as ' ' characters. When printed, the TextGrid results in a width x height grid of ascii characters.

Properties:

grid: The underlying grid object.

Create a new ASCII grid visualization.

Args:

grid: The underlying Grid object. converter: function for converting the content of each cell

to ascii. Takes the contents of a cell, and returns a single character.

render() [source]

What to show when printed.

Modules

Container for all built-in visualization modules.

Modular Canvas Rendering

Module for visualizing model objects in grid cells.

class **CanvasGrid**(*portrayal_method*, *grid_width*, *grid_height*, *canvas_width=500*, *canvas_height=500*) [source]

A CanvasGrid object uses a user-provided portrayal method to generate a portrayal for each object. A portrayal is a JSON-ready dictionary which tells the relevant JavaScript code (GridDraw.js) where to draw what shape.

The render method returns a dictionary, keyed on layers, with values as lists of portrayals to draw. Portrayals themselves are generated by the user-provided portrayal_method, which accepts an object as an input and produces a portrayal of it.

A portrayal as a dictionary with the following structure:

“x”, “y”: Coordinates for the cell in which the object is placed. “Shape”: Can be either “circle”, “rect”, “arrowHead” or a custom image.

For Circles:

“r”: The radius, defined as a fraction of cell size. r=1 will

fill the entire cell.

“xAlign”, “yAlign”: Alignment of the circle within the cell.

Defaults to 0.5 (center).

For Rectangles:

“w”, “h”: The width and height of the rectangle, which are in

fractions of cell width and height.

“xAlign”, “yAlign”: Alignment of the rectangle within the

cell. Defaults to 0.5 (center).

For arrowHead:

“scale”: Proportion scaling as a fraction of cell size. “heading_x”: represents x direction unit vector. “heading_y”: represents y direction unit vector.

For an image:

The image must be placed in the same directory from which the server is launched. An image has the attributes “x”, “y”, “scale”, “text” and “text_color”.

“Color”: The color to draw the shape in; needs to be a valid HTML

color, e.g.”Red” or “#AA08F8”

“Filled”: either “true” or “false”, and determines whether the shape is

filled or not.

"Layer": Layer number of 0 or above; higher-numbered layers are drawn

above lower-numbered layers.

"text": The text to be inscribed inside the Shape. Normally useful for

showing the unique_id of the agent.

"text_color": The color to draw the inscribed text. Should be given in

conjunction of "text" property.

Attributes:

portrayal_method: Function which generates portrayals from objects, as

described above.

grid_height, grid_width: Size of the grid to visualize, in cells. canvas_height, canvas_width: Size, in pixels, of the grid visualization

to draw on the client.

template: "canvas_module.html" stores the module's HTML template.

Instantiate a new CanvasGrid.

Args:

portrayal_method: function to convert each object on the grid to

a portrayal, as described above.

grid_width, grid_height: Size of the grid, in cells. canvas_height, canvas_width: Size of the canvas to draw in the

client, in pixels. (default: 500x500)

render(*model*) [source]

Build visualization data from a model object.

Args:

model: A model object

Returns:

A JSON-ready object.

Chart Module

Module for drawing live-updating line charts using Charts.js

class **ChartModule**(*series*, *canvas_height=200*, *canvas_width=500*, *data_collector_name='datacollector'*) [source]

Each chart can visualize one or more model-level series as lines

with the data value on the Y axis and the step number as the X axis.

At the moment, each call to the render method returns a list of the most recent values of each series.

Attributes:

series: A list of dictionaries containing information on series to

plot. Each dictionary must contain (at least) the “Label” and “Color” keys. The “Label” value must correspond to a model-level series collected by the model’s DataCollector, and “Color” must have a valid HTML color.

canvas_height, canvas_width: The width and height to draw the chart on

the page, in pixels. Default to 200 x 500

data_collector_name: Name of the DataCollector object in the model to

retrieve data from.

template: “chart_module.html” stores the HTML template for the module.

Example:

schelling_chart = ChartModule([{“Label”: “happy”, “Color”: “Black”}],

data_collector_name=”datacollector”)

TODO:

Have it be able to handle agent-level variables as well.

More Pythonic customization; in particular, have both series-level and chart-level options settable in Python, and passed to the front-end the same way that “Color” is currently.

Create a new line chart visualization.

Args:

series: A list of dictionaries containing series names and

HTML colors to chart them in, e.g. [{“Label”: “happy”, “Color”: “Black”},] canvas_height, canvas_width: Size in pixels of the chart to draw. data_collector_name: Name of the DataCollector to use.

render(*model*) [source]

Build visualization data from a model object.

Args:

model: A model object

Returns:

A JSON-ready object.