pv Documentation

Release 1.9.0dev

Marco Biasini

Sep 15, 2017

Contents

PV is a WebGL-based viewer for proteins and other biological macromolecules. It aims to be fast and easy to integrate into websites.

CHAPTER 1

Contents:

Getting started with PV

Getting the PV source-code

The simplest way to get PV into your website is by downloading one of the release tarballs from github.com. The release tarballs contain the self-contained development and minified code for PV which can directly be integrated into your website.

Installing with bower

To install PV with bower, change to your project directory and type the following:

```
bower install bio-pv
```

Note: Bower support has only been added to versions 1.8 and newer, so it won't be possible to use this method of installation for older versions.

Installing from git

Alternatively, you can clone the github repository of PV. This approach is only recommended if you are planning to make changes to PV itself. Otherwise it's simpler to either download one of the release tarballs or install through bower.

```
git clone https://github.com/biasmv/pv
cd pv
```

The minified version of PV is checked into the git repository (`bio-pv.min.js`). You may either use this file directly, or create it from the sources. In the case of the latter, you will need Grunt and NPM installed on your system. Use the following commands to build:

```
# setup dev environment for PV
npm install --setup-dev
# runs grunt and applies some additional name mangling to the source
scripts/make.sh
```

Upon success, bio-pv.min.js is placed in the project's top-level folder.

Setting up a small website

The following minimal example shows how to include PV in a website for protein structure visualisation. For that purpose, we will create a small index.html file containing the bare-minimum required to run PV. The example does not depend on any external library. But of course it is also possible to combine PV with jQuery or other popular JS libraries.

In case you want to recreate the example, create a directory for the index.html file and change into that directory.

The index.html file

The page is essentially a bare-bone HTML page which includes the pv.min.js file. In the preamble, we define a meta element to prevent page scrolling and load the PV library.

```
<html>
<head>
  <title>Dengue Virus Methyl Transferase</title>
  <meta name="viewport" content="width=device-width, user-scalable=no, minimum-
↪scale=1.0, maximum-scale=1.0">
</head>
<body>
<div id=viewer></div>
</body>
<script type='text/javascript' src='pv.min.js'></script>
```

Now on to the interesting part. First, we *initialise the viewer* with custom settings. The width and height of the viewer are initialized to 600 pixels with antialising enabled and a medium detail level. These settings have been tested on a variety of devices and are known to work well for typical proteins.

```
<script type='text/javascript'>
// override the default options with something less restrictive.
var options = {
  width: 600,
  height: 600,
  antialias: true,
  quality : 'medium'
};
// insert the viewer under the Dom element with id 'gl'.
var viewer = pv.Viewer(document.getElementById('viewer'), options);
</script>
```

Most of the work happens in loadMethylTransferase. This function will be called when the DOMContentLoaded event fires and we will use it to populate the WebGL viewer with a nice protein structure.

```
<script type='text/javascript'>

function loadMethylTransferase() {
  // asynchronously load the PDB file for the dengue methyl transferase
  // from the server and display it in the viewer.
  pv.io.fetchPdb('1r6a.pdb', function(structure) {
      // display the protein as cartoon, coloring the secondary structure
      // elements in a rainbow gradient.
      viewer.cartoon('protein', structure, { color : color.ssSuccession() });
      // there are two ligands in the structure, the co-factor S-adenosyl
      // homocysteine and the inhibitor ribavirin-5' triphosphate. They have
      // the three-letter codes SAH and RVP, respectively. Let's display them
      // with balls and sticks.
      var ligands = structure.select({ rnames : ['SAH', 'RVP'] });
      viewer.ballsAndSticks('ligands', ligands);
      viewer.centerOn(structure);
  });
}

// load the methyl transferase once the DOM has finished loading. That's
// the earliest point the WebGL context is available.
document.addEventListener('DOMContentLoaded', loadMethylTransferase);
</script>
```

Running the Example

Before running the example, we have to make sure that the pv.min.js file and the PDB file for the methyl transferase are in the right location. The easiest is to copy the pv.min.js file from the release tarball and fetch the PDB file for 1r6a from the PDB website. Then serve the files using Python's SimpleHTTPServer:

```
python -m SimpleHTTPServer
```

And visit the localhost:8000 with a WebGL-enabled browser.

The Viewer

The 3D molecules are managed and rendered by an instance of the viewer class. It serves as the main entry point for the protein viewer and is where most of the action happens. In the following, the full API of *Viewer()* is described. The methods are roughly categorized into the following sections:

- *Initialization and Configuration*
- *Rendering*
- *Camera Positioning/Orientation*
- *Viewer Events*
- *Object Management*

Initialization and Configuration

class pv.**Viewer**(*parentElement*[, *options*])

Construct a new viewer, inserting it as the last child of parentElement. *options* is as dictionary that allows to

control the initial settings for the viewer. Many of these settings can be changed later. The default options are chosen very restrictive on purpose. Valid options are:

- *width* The width (in pixels) of the viewer. The special value 'auto' can be used to set the width to the width of the parent element. Defaults to 500.
- *height* The height (in pixels) of the viewer. The special value 'auto' can be used to set the height to the height of the parent element. Defaults to 500.
- *antialias*: whether full-scene antialiasing should be enabled. When available, antialiasing will use the built-in WebGL antialiasing. When not, it will fall back to a manual supersampling of the scene. Manual antialiasing can be disabled by setting the *forceManualAntialiasing* option to false. Enabling antialiasing improve the visual results considerably, but also slows down rendering. When rendering speed is a concern, the *antialias* option should be set to false. Defaults to false.
- *forceManualAntialiasing*: whether manual antialiasing should be enabled. Manual antialiasing is used when the WebGL context does not support antialiasing. Set this option to false to disable the fallback behavior and only enable antialising when the WebGL context supports it. Defaults to true.
- *quality* the level of detail for the geometry. Accepted values are *low*, *medium*, and *high*. See `quality()` for a description of these values. Defaults to *low*.
- *slabMode* sets the default slab mode for the viewer. See `slabMode()` for possible values. Defaults to 'auto'.
- *background* set the default background color of the viewer. Defaults to 'white'. See *Color Notations*
- *doubleClick* set the event handler for an atom double click/touch event. When the parameter is a function it is added as a new 'doubleClick' event handler. See `addListener()` for details. If it is set to the special value 'center', an event listener is installed that centers the viewer on the double clicked atom, residue. The default is 'center'.
- *click* set the event handler for an atom click/touch event (see *doubleClick*). The default is null (no listener).
- *animateTime* controls the default animation duration in milliseconds. By default, the animation is set to 0 (no animation). By setting it to higher values, rotation, zoom and shift are animated. Note that enabling this can have negative impact on performance, especially with large molecules and on low-end devices.
- *fog* whether depth-cue ('fog') should be enabled. By default, fog is enabled. Pass false to disable fog.
- *fov* the field of view in degrees. Default is 45 degrees.
- *outline* whether outline rendering should be enabled. When enabled, outline rendering draws a uniformly colored outline around the mesh geometries to improve contrast. By default outline rendering is enabled.
- *outlineColor* the color of the outline. Default is black. When outline rendering is disabled, setting this value has no effect.
- *outlineWidth* the width of the outline in pixels. Default is 1.5. When outline rendering is disabled, setting this value has no effect.

The following code defines a new viewer. This can be done during page load time, before the DOMContentLoaded event has been emitted. Render objects can only be added once the DOMContentLoaded event has fired. Typically it's best to put any object loading and display code into a DOMContentLoaded event handler.

```
// override the default options with something less restrictive.
var options = {
  width: 600,
  height: 600,
  antialias: true,
  quality : 'medium'
};
// insert the viewer under the Dom element with id 'gl'.
```

```
var viewer = pv.Viewer(document.getElementById('gl'), options);

viewer.on('viewerReady', function() {
  // add structure here
});
```

pv.Viewer.**quality**([*value*])
Gets (or sets) the default level of detail for the render geometry. This property sets the default parameters for constructing render geometry, for example the number of arcs that are used for tubes, or the number of triangles for one sphere. Accepted values are

- *low* The geometry uses as few triangles as possible. This is the fastest, but also visually least pleasing option. Use this option, when it can be assumed that very large molecules are to be rendered.
- *medium* provides a good tradeoff between visual fidelity and render speed. This options should work best for typical proteins.
- *high* render the scene with maximum detail.

Changes to the quality only affect newly created objects/geometries. Already existing objects/geometries are not affected.

Rendering

This section describes the high-level API for displaying molecular structures on screen. The interface consists of render methods part of *Viewer()* which accept a name and a structure and create a graphical representation out of it. For example, to create a cartoon representation, the following code will do:

```
// creates a cartoon representation with standard parameters
var myCartoon = viewer.cartoon('molecule', myMolecule);
```

These methods will automatically add the object to the viewer, there is not need to call *pv.Viewer.add()* on the object.

pv.Viewer.**lines**(*name*, *structure*[, *options*])
Renders the structure (*Mol()*, or *MolView()*) at full connectivity level, using lines for the bonds. Atoms with no bonds are represented as small crosses. Valid *options* are:

- *color*: the color operation to be used. Defaults to *pv.color.byElement()*.
- *lineWidth*: The line width for bonds and atoms. Defaults to 4.0

Returns The geometry of the object.

pv.Viewer.**points**(*name*, *structure*[, *options*])
Renders the atoms of a structure (*Mol()*, or *MolView()*) as a point cloud. Valid *options* are:

- *color*: the color operation to be used. Defaults to *pv.color.byElement()*.
- *pointSize* relative point size of the points to be rendered. Defaults to 1.0

Returns The geometry of the object.

pv.Viewer.**spheres**(*name*, *structure*[, *options*])
Renders the structure (*Mol()*, or *MolView()*) at full-atom level using a sphere for each atom. Valid *options* are:

- *color*: the color operation to be used. Defaults to *pv.color.byElement()*.

- *sphereDetail*: the number of horizontal and vertical arcs for the sphere. The default *sphereDetail* is determined by *pv.Viewer.quality()*.

pv.Viewer.**lineTrace**(*name*, *structure*[, *options*])
Renders the protein part of the structure (*Mol()*, or *MolView()*) as a Carbon-alpha trace using lines. Consecutive carton alpha atoms are connected by a straight line. For a mesh-based version of the Carbon-alpha trace, see *pv.Viewer.trace()*.

- *color*: the color operation to be used. Defaults to *uniform()*.
- *lineWidth*: The line width for bonds and atoms. Defaults to 4.0

pv.Viewer.**sline**(*name*, *structure*[, *options*])
Renders the protein part of the structure (*Mol()*, or *MolView()*) as a smooth line trace. The Carbon-alpha atoms are used as the control points for a Catmull-Rom spline. For a mesh-based version of the smooth line trace, see *pv.Viewer.tube()*.

- *color*: the color operation to be used. Defaults to *uniform()*.
- *lineWidth*: The line width for bonds and atoms. Defaults to 4.0
- *strength*: influences the magnitude of the tangents for the Catmull-Rom spline. Defaults to 0.5. Meaningful values are between 0 and 1.
- *splineDetail*: Number of subdivision per Carbon alpha atom. The default value is is determined by *pv.Viewer.quality()*.

pv.Viewer.**trace**(*name*, *structure*[, *options*])
Renders the structure (*Mol()*, or *MolView()*) as a carbon-alpha trace. Consecutive Carbon alpha atoms (CA) are connected by a cylinder. For a line-based version of the trace render style, see *pv.Viewer.lineTrace()*. Accepted *options* are:

- *color*: the color operation to be used. Defaults to *uniform()*.
- *radius*: Radius of the tube. Defaults to 0.3.
- *arcDetail*: number of vertices on the tube. The default is determined by *pv.Viewer.quality()*.
- *sphereDetail* number of vertical and horizontal arcs for the spheres.

pv.Viewer.**tube**(*name*, *structure*[, *options*])
Renders the structure (*Mol()*, or *MolView()*) as a smoothly interpolated tube.

- *color*: the color operation to be used. Defaults to *pv.color.bySS()*.
- *radius*: Radius of the tube. Defaults to 0.3.
- *arcDetail*: number of vertices on the tube. The default is determined by *pv.Viewer.quality()*.
- *strength*: influences the magnitude of the tangents for the Catmull-Rom spline. Defaults to 1.0. Meaningful values are between 0 and 1.
- *splineDetail* number of subdivisions per Carbon-alpha atom. The default is termined by *pv.Viewer.quality()*.

pv.Viewer.**cartoon**(*name*, *structure*[, *options*])
Renders the structure (*Mol()*, or *MolView()*) as a helix, strand coil cartoon. Accepted *options* are:

- *color*: the color operation to be used. Defaults to *pv.color.bySS()*.
- *radius*: Radius of the tube profile. Also influences the profile thickness for helix and strand profiles. Defaults to 0.3.
- *arcDetail*: number of vertices on the tube. The default is determined by *pv.Viewer.quality()*.

•*strength*: influences the magnitude of the tangents for the Catmull-Rom spline. Defaults to 1.0. Meaningful values are between 0 and 1.

•*splineDetail* number of subdivisions per Carbon-alpha atom. The default is termined by *pv.Viewer.quality()*.

pv.Viewer.**ballsAndSticks**(*name*, *structure*[, *options*])

Renders the structure (*Mol()*, or *MolView()*) as a ball and stick model. Accepted *options* are:

•*color*: the color operation to be used. Defaults to *pv.color.byElement()*.

•*cylRadius*: Radius of the tube profile. Defaults to 0.1.

•*sphereRadius*: Radius of the sphere profile. Defaults to 0.3.

•*arcDetail*: number of vertices on the tube. The default is determined by *pv.Viewer.quality()*.

•*sphereDetail* number of vertical and horizontal arcs for the spheres.

•*scaleByAtomRadius* Whether to scale spheres by atom's van der Waals radius. Defaults to true.

pv.Viewer.**renderAs**(*name*, *structure*, *mode*[, *options*])

Function to render the structure in any of the supported render styles. This essentially makes it possible to write code that is independent of the particular chosen render style.

Arguments

- **mode** – One of 'sline', 'lines', 'trace', 'lineTrace', 'cartoon', 'tube', 'spheres', ballsAndSticks'
- **options** – options dictionary passed to the chosen render mode. Refer to the documentation for the specific mode for a list of supported options.

Returns The created geometry object.

pv.Viewer.**label**(*name*, *text*, *pos*[, *options*])

Places a label with *text* at the given position in the scene

Arguments

- **name** – Uniquely identifies the label
- **text** – The text to be shown
- **pos** – An array of length 3 holding the x, y, and z coordinate of the label's center.
- **options** – Optional dictionary to control the font, text style and size of the label (see below)

Accepted *options* are:

•*font*: name of the font. Accepted values are all HTML/CSS font families. Default is 'Verdana'.

•*fontSize*: the size of the font in pixels. Default is 24.

•*fontColor*: the CSS color to be used for rendering the text. Default is black.

•*fontStyle* the font style. Can by any combination of 'italic', 'bold'. Default is 'normal'.

Returns the created label.

pv.Viewer.**customMesh**(*name*)

Creates a new object to hold user-defined collection of geometric shapes. For details on how to add shapes, see *Drawing Geometric Shapes*

Arguments

- **name** – uniquely identifies the custom mesh.

Returns A new *pv.CustomMesh()* instance.

Camera Positioning/Orientation

pv.Viewer.**setCamera**(*rotation*, *center*, *zoom*[, *ms*])
Function to directly set the rotation, center and zoom of the camera.

The combined transformation matrix for the camera is calculated as follows: First the origin is shifted to the center, then the rotation is applied, and lastly the camera is translated away from the center by the negative zoom along the rotated Z-axis.

Arguments

- **rotation** – Either a 4x4 or 3x3 matrix in the form of a one-dimensional array of length 16 or 9. It is up to the caller to ensure the matrix is a valid rotation matrix.
- **center** – the new camera center.
- **zoom** – distance of the eye position from the viewing center
- **ms** – if provided and non-zero defines the animation time for moving/rotating/zooming the camera from the current position to the new rotation,center and zoom. If zero, the rotation/center and zoom factors are directly set to the desired values. The default is zero.

pv.Viewer.**setRotation**(*rotation*[, *ms*])
Function to directly set the rotation of the camera. This is identical to calling *setCamera()* with the current center and zoom values.

Arguments

- **rotation** – Either a 4x4 or 3x3 matrix in the form of a one-dimensional array of length 16 or 9. It is up to the caller to make sure the matrix is a rotation matrix.
- **ms** – if provided and non-zero defines the animation time rotating the camera from the current rotation to the target rotation. If zero, the rotation is immediately set to the target rotation. The default is zero.

pv.Viewer.**setCenter**(*center*[, *ms*])
Function to directly set the center of view of the camera. This is identical to calling *setCamera()* with the current rotation and zoom values.

Arguments

- **center** – The new center of view of the "center".
- **ms** – if provided and non-zero defines the time in which the camera center moves from the current center the target center. If zero, the center is immediately set to the target center. The default is zero.

pv.Viewer.**setZoom**(*zoom*[, *ms*])
Function to directly set the zoom factor of the camera. This is identical to calling *setCamera()* with the current rotation and center values.

Arguments

- **zoom** – The distance of the camera from the "center". Only positive values are allowed.
- **ms** – if provided and non-zero defines the time in which the camera zoom level moves from thecurrent zoom level to the target zoom. If zero, the zoom is immediately set to the target zoom. The default is zero.

pv.Viewer.**centerOn**(*obj*)
Center the camera on a given object, leaving the zoom level and orientation untouched.

Arguments

- **obj** – Must be an object implementing a *center* method returning the center of the object, e.g. an instance of *pv.mol.MolView()*, *pv.mol.Mol()*

pv.Viewer.**autoZoom**([*ms*])
Adjusts the zoom level such that all objects are visible on screen and occupy as much space as possible. The center and orientation of the camera are not modified.

Arguments

- **ms** – if provided and non-zero defines the time in which the camera zoom level moves from the current zoom level to the target zoom. If zero, the zoom is immediately set to the target zoom. If no value is provided it use the default animation time of the viewer.

pv.Viewer.**fitTo**(*obj*[, *ms*])
Adjust the zoom level and center of the camera to fit the viewport to a given object. The method supports fitting to selections, or arbitrary SceneNodes. To fit to a subset of atoms, pass the selection as the *obj* argument:

Arguments

- **ms** – if provided and non-zero defines the time in which the camera zoom level moves from the current zoom level to the target zoom. If zero, the zoom is immediately set to the target zoom. If no value is provided it will use the default animation time of the viewer.

```
viewer.fitTo(structure.select({rname : 'RVP'});
```

To fit to an entire render objects, pass the object as the *obj* argument:

```
var obj = viewer.cartoon('obj', structure);
viewer.fitTo(obj);
```

Arguments

- **what** – must be an object which implements updateProjectionInterval, e.g. a SceneNode, a *pv.mol.MolView()*, or *pv.mol.Mol()*.

pv.Viewer.**translate**(*vector*, *ms*)
Translate the viewer center.

Arguments

- **vector** – The 3-dimensional vector to translate by. The vector is in screen coordinates, e.g. the vector [1,0,0] is aligned to the X-axis as currently seen on screen.
- **ms** – When provided, the translation is animated from the current to the target position. When omitted (or 0) the camera is immediately set to the target position.

pv.Viewer.**rotate**(*axis*, *angle*, *ms*)
Rotate the viewer around an axis by a certain amount.

Arguments

- **axis** – 3-dimensional axis to rotate around. The axes are in the screen coordinate system, meaning the X- and Y-axes are aligned to the screen's X and Y axes and the Z axis points towards the camera's eye position. The default rotation axis is [0,1,0]. The axis must be normalized.

- **angle** – the rotation angle in radians. When positive, the rotation is in counter-clockwise direction, when negative, the rotation is in clockwise-direction. The rotation angle is always used modulo 2π.
- **ms** – When provided, the rotation is animated from the current to the target rotation. When omitted (or 0) the camera is immediately rotation to the target rotation.

pv.Viewer.**spin**(*enable*)
pv.Viewer.**spin**(*speed*[, *axis*])
Enable/disable spinning of the viewer around a screen axis.

The first signature enables/disables spinning with default parameters, the second allows to control the speed as well as the axis to rotate around.

Arguments

- **enable** – whether spinning should be enabled. When false, spinning is disabled. When true, spinning is enabled around the y axis with a default speed of Math.PI/8, meaning a full rotation takes 16 seconds.
- **axis** – 3 dimensional axis to rotate around. The axes are in the screen coordinate system, meaning the X- and Y-axes are aligned to the screen's X and Y axes and the Z axis points towards the camera's eye position. The default rotation axis is [0,1,0]. The axis must be normalized.
- **speed** – The number of radians per second to rotate. When positive, rotation is in counter-clockwise direction, when negative rotation is in clockwise direction.

Returns true when spinning is enabled, false if not.

pv.Viewer.**requestRedraw**()
Request a redraw of the viewer, e.g. to refresh the content visible on the screen. Most of the time, you will not have to call this function directly. However, if you notice that a certain change is not taking effect, try adding requestRedraw().

Fog and Slab Modes

Proteins come in all sizes and shapes. For optimal viewing, some camera parameters must thus be adjusted for each molecule. Two of these parameters are the near and far clipping planes of the camera. Only geometry between the near and far clipping plane are visible on the screen. Geometry in front of the near and at the back of the far clipping planes are clipped away. Typically, the near and far clipping planes must be set such that contain all visible geometry in front of the camera. However, sometimes it is desired to only show a certain 'slab' of the molecule. To support both of these scenarios, PV has multiple modes, called slab modes.

pv.Viewer.**slabMode**(*mode[,options*)
Sets the current active slab mode of the viewer. *mode* must be one of 'fixed' or 'auto'.

•When slab mode is set to 'auto', the near and far clipping planes as well as fog are adjusted based on the visible geometry. This causes the clipping planes to be updated on every rotation of the camera, change of camera's viewing center and when objects are added/removed.

•When the slab mode is set to 'fixed', automatic adjustment of the near and far clipping planes as well as fog is turned off. The values are kept constant and can be set by the user. To set specific near and far clipping planes provide them in a dictionary as the option argument when calling slabMode:

```
viewer.slabMode('fixed', { near: 1, far : 100 });
```

Viewer Events

Custom viewer event handlers can be registered by calling *pv.Viewer.addListener()*. These callbacks have the following form.

pv.Viewer.**addListener**(*type*, *callback*)
pv.Viewer.**on**(*type*, *callback*)

Arguments

- **type** – The type of event to listen to. Must be either 'atomClicked', 'atomDoubleClicked', 'viewerReady', 'keypress', 'keydown', 'keyup', 'mousemove', 'mousedown', 'mouseup', or 'viewpointChanged'.

When an event fires, callbacks registered for that event type are invoked with type-specific arguments. See documentation for the individual events for more details

Initialization Event (viewerReady)

Invoked when the viewer is completely initialized and is ready for displaying of structures. It's recommended to put calls to any of the *geometry-creating funtions* into a viewerReady callback as they expect a completely constructed viewer. It's however possible to start loading the structure data before 'viewerReady', as long as they are not added to the viewer.

Callbacks receive the initialized viewer as the first argument.

When the 'viewerReady' callback is registered *after* the page has finished loading, the event callback is directly invoked from *addListener/on*.

The following code example shows how to add a yellow sphere to the center of the scene:

```
// insert the viewer under the Dom element with id 'gl'.
var viewer = pv.Viewer(document.getElementById('gl'), options);

viewer.on('viewerReady', function(viewer) {
  var customMesh = viewer.customMesh('yellowSphere');
  customMesh.addSphere([0,0,0], 5, { color : 'yellow' });
});
```

Mouse Interaction Events (click, doubleClick)

Mouse selection events are fired when the user clicks or double clicks on the viewer.

The arguments of the callback function are *picked*, and *originalEvent* which is the original mouse event. Picked contains information about the scene nodes that was clicked/doubleClicked as well as target of the event. For representations of molecules, the target is always an atom, for custom meshes target is set to the user-specified data stored in the mesh when calling *addTube()*, or *addSphere()*. When no object was under the cursor, picked is null.

It also contains a transformation matrix, that if set needs to be applied to the atom's position to get the correct position in global coordinates. This is illustrated in the second example below.

The following code simply logs the clicked atom to the console when an atom is clicked and does nothing otherwise.

```
viewer.addListener('click', function(picked) {
  if (picked === null) return;
  var target = picked.target();
  if (target.qualifiedName !== undefined) {
    console.log('clicked atom', target.qualifiedName(), 'on object',
```

```
                picked.node().name());
  }
});
```

The following code shows how to listen for double click events to either make the selection the focal point and center of zoom, or zoom out to the whole structure if the background is double clicked.

```
var structure = .... // point to what you want the default background selection to␣
↪view
viewer.on('doubleClick', function(picked) {
  if (picked === null) {
    viewer.fitTo(structure);
    return;
  }
  viewer.setCenter(picked.pos(), 500);
});
```

Camera Position/Rotation/Zoom Changed Event (experimental)

The *viewpointChanged* event is fired whenever the camera orientation/center or zoom changes. The callback is invoked with the camera object as the first argument. As an example, the following code shows how to synchronize the orientation of two viewers. Whenever the orientation of one of them changes, the other is updated as well:

```
viewer1.on('viewpointChanged', function(cam) {
  viewer2.setCenter(cam.center());
  viewer2.setCamera(cam.rotation(), cam.center(), cam.zoom());
});
viewer2.on('viewpointChanged', function(cam) {
  viewer1.setCenter(cam.center());
  viewer1.setCamera(cam.rotation(), cam.center(), cam.zoom());
});
```

This is an experimental feature and might change in future releases.

Object Management

Multiple render objects can be displayed at once. To be able to refer to these objects, all objects need to be assigned a name that uniquely identifies them. *Viewer()* offers methods to conveniently add, retrieve objects, or remove them from the viewer.

pv.Viewer.**add**(*name*, *obj*)
: Add a new object to the viewer. The object's name property will be set to name, under which it can be referenced in the future. Typically, there is no need to call add, since the objecs will be automatically added to the viewer when they are created.

  Returns A reference to *obj*.

pv.Viewer.**get**(*name*)
: Retrieve the reference to an object that has previously been added to the viewer. When an object matching the name could be found, it is returned. Otherwise, null is returned.

pv.Viewer.**hide**(*globPattern*)
pv.Viewer.**show**(*globPattern*)
: Hide/show objects matching glob pattern. The render geometry of hidden objects is retrained, but is not longer visible on the screen, nor are they available for object picking.

pv.Viewer.**rm**(*globPattern*)
Remove objects matching glob pattern from the viewer.

pv.Viewer.**clear**()
Remove all objects from the viewer. In case you are calling this function, but are not adding new content after that, you will need to call *requestRedraw()* to update the content of the screen.

Rendered Molecules

The displaying of molecules is handled by *pv.BaseGeom()*, and subclasses. The two subclasses LineGeom and MeshGeom are use for line and mesh-based render styles, respectively.The former for render styles which are based on simple lines (e.g. lines, smooth line trace and line trace), the latter for all other render styles, e.g. cartoon, balls and sticks, spheres, tube and trace.

class pv.**BaseGeom**()
Represents a geometric object. Note that this class is not part of the public API. New instances are create by calling one of the *render functions*.

pv.BaseGeom.**showRelated**()
pv.BaseGeom.**setShowRelated**(*what*)
Controls the display of symmetry-related copies of a molecular structure. When set to 'asym', no symmetry-related copies are rendered, even when they are available. When set to a non-empty string, the Assembly of the given name is used. In case no such assembly exists, the asymmetric unit is shown. See symmetry for a more detailed description.

Arguments

- **what** – the new name of the symmetry related copies to be displayed

Returns the name of the symmetry related copy shown.

pv.BaseGeom.**setOpacity**(*alpha*)
Set the opacity of the whole geometry to a constant value. See *Opacity* for details.

Arguments

- **alpha** – The new opacity in the range between 0 and 1.

pv.BaseGeom.**colorBy**(*colorOp*)
pv.BaseGeom.**colorBy**(*colorOp*, *view*)
Color the geometry by the given color operation. For a description of available color operations, see *Coloring Molecular Structures*.

Arguments

- **colorOp** – The color operation to be applied to the structure.
- **view** – when specified, the color operation will only be applied to parts contained in the view. Other parts will be left untouched. When omitted, the color operation will be applied to the whole structure.

pv.BaseGeom.**getColorForAtom**(*atom*, *color*)
Convenience function to obtain the current color of a given atom.

Arguments

- **atom** – the atom for which to retrieve the color. Can be an *AtomView()*, or *Atom()* instance, independent of whether the geometry was created with a *Mol()*, or *MolView()*
- **color** – array of length 4 into which the color is placed

Returns the array holding the color, or null if the atom is not part of the rendered geometry

pv.BaseGeom.**setSelection**(*selection*)
pv.BaseGeom.**selection**()
Get/set selection of the render geometry, e.g. the part of the structure that is drawn as selected. The viewer draws a halo around the selected parts of the structure using the current highlight color.

Arguments

- **selection** – the subset of the structure to be selected/highlighted.

pv.BaseGeom.**eachCentralAtom**(*callback*)
Helper function for looping over all visible central atoms, including symmetry related ones

This function invokes the callback function for all symmetry copies of every visible central atom contained in this object. The callback takes two arguments, the first being the central atom, the second the atom position with the symmetry-operator's transformation matrix applied. Note that the transformed atom position is only to be used inside the callback. If you want to store the transformed position, or modify it, a copy must be obtained first.

Example:

```
var obj = viewer.get('my.object');
var sum = vec3.create();
var count = 0;
obj.eachCentralAtom(function(atom, transformedPos) {
  count += 1;
  vec3.add(sum, sum, transformedPos);
});
var center = vec3.scale(sum, sum, 1.0/count);
viewer.setCenter(center);
```

Drawing Geometric Shapes

Geometric shapes can be added to the 3D scene through *pv.CustomMesh()*. At the moment, only two shapes are supported: tubes and spheres. More can be added on request. A new *pv.CustomMesh()* instance can be obtained by calling *pv.Viewer.customMesh()*.

Example

```
var cm = viewer.customMesh('cross');
cm.addTube([-50,0,0], [50,0,0], 1, { cap : true, color : 'red' });
cm.addTube([0,-50,0], [0,50,0], 1, { cap : true, color : 'green' });
cm.addTube([0,0, -50], [0,0,50], 1, { cap : true, color : 'blue' });
cm.addSphere([0, 0, 0], 3, { color : 'yellow' });
```

class pv.**CustomMesh**()
Holds a collection of user-defined geometric shapes

pv.CustomMesh.**addTube**(*start*, *end*, *radius*[, *options*])
Adds a tube (open or capped) to the custom mesh container

Arguments

- **start** – 3-dimensional start coordinate of the tube
- **end** – 3-dimensional end coordinate of the tube
- **radius** – radius in Angstrom

- **options** – a dictionary with the following keys. *color*: when provided, used as the color for the tube, *cap* when set to false, the tube is left open, meaning the ends are not capped. *userData*: when provided the user data is added to the object. This data is available when a pick event (click/double click occurs on the object as the target of the pick event. When not provided, userData is set to null.

`pv.CustomMesh.`**addSphere**(*center*, *radius*[, *options*])

Adds a sphere to the custom mesh container

Arguments

- **center** – 3-dimensional center coordinate for the sphere
- **radius** – radius in Angstrom
- **options** – a dictionary with the following keys. *color*: when provided, used as the color for the tube. *userData*: when provided the user data is added to the object. This data is available when a pick event (click/double click occurs on the object as the target of the pick event. When not provided, userData is set to null.

Coloring Molecular Structures

This document describes how a molecular structure's color can be controlled.

The coloring scheme can be specified when generating the render geometry, e.g. when using one of the *render*. functions. Coloring can also be changed later on using the *pv.BaseGeom.colorBy()* function. The latter also allows to apply coloring to subparts of the structure only. These two different ways to control the coloring is described in the following code example:

```
// color the whole structure in red, while generating the geometry.
var geom = viewer.lines('myStructure', myStructure,  { color: pv.color.uniform('red')␣
↪});
// oh, no, I changed my mind: We want everything in blue!
geom.colorBy(pv.color.uniform('blue'));
```

Coloring is implemented with coloring operations. These operations are small function objects which map a certain atom or residue to a color. They can be as simple as coloring a complete structure in *one color*, or as complex as mapping a *numeric property to a color gradient*. PV includes a variety of coloring operations for the most common tasks. For more complex applications it is also possible to extend the coloring with new operations.

Available color operations

The following color operations are available:

`pv.color.`**uniform**([*color*])

Colors the structure with a uniform color.

Arguments

- **color** – a valid color identifier, or rgb instance to be used for the structure. Defaults to red.

`pv.color.`**byElement**([*palette*])

Applies the CPK coloring scheme to the atoms. For example, carbon atoms are colored in light-grey, oxygen in red, nitrogen in blue, sulfur in yellow.

Arguments

- **palette** – an optional object of colors to draw from. Include H, C, N, O, S, P. Defaults to CPK.

pv.color.**byChain**([*gradient*])
Applies a unique uniform color for each chain in the structure. The chain colors are drawn from a gradient, which guarantees that chain colors are unique.

Arguments

- **gradient** – An optional gradient to draw colors from. Defaults to a rainbow gradient.

pv.color.**ssSuccession**([*gradient*[, *coilColor*]])
Colors the structure's secondary structure elements with a gradient, keeping the color constant for each secondary structure element. Coil residues, and residue without secondary structure (e.g. ligands) are a colored with *coilColor*.

Arguments

- **gradient** – The graident to draw colors from. Defaults to rainbow.
- **coilColor** – The color for residues without regular secondary structure. Defaults to lightgrey.

pv.color.**bySS**([*palette*])
Colors the structure based on secondary structure type of the residue. Distinct colors are used for helices, strands and coil residues.

Arguments

- **palette** – An optional *gradient* or *object* to draw colors from. Object must include *C*, *H*, *E* fields. Defaults to lightgrey, blue, green.

pv.color.**rainbow**([*gradient*])
Maps the residue's chain position (its index) to a color gradient.

Arguments

- **gradient** – An optional gradient to draw colors from. Defaults to a rainbow gradient.

pv.color.**byAtomProp**(*prop*[, *gradient*[, *range*]])
pv.color.**byResidueProp**(*prop*[, *gradient*[, *range*]])
Colors the structure by mapping a numeric property to a color gradient. *pv.color.byAtomProp()* uses properties from atoms, whereas *byResidueProp()* uses properties from residues. By default, the range of values is automatically determined from the property values and set to the minimum and maximum of observed values. Alternatively, the range can also be specified as the last argument.

Arguments

- **prop** – name of the property to use for coloring. It is assumed that the property is numeric (floating point or integral). The name can either refer to a custom property, or a built-in property of atoms or residues.
- **gradient** – The graident to use for coloring. Defaults to rainbow.
- **range** – an array of length two specifying the minimum and maximum value of the float properties. When not specified, the value range is determined from observed values.

Opacity

In addition to RGB color, the opacity of structures can be controlled as well. Opacity (alpha) is handled like the other RGB components. To render a structure semi-transparently, simply pass a color with an alpha smaller than one to the color operations.

Additionally, the opacity of a rendered structure can directly be changed by calling *pv.BaseGeom.setOpacity()*, for example, to change the opacity of all structures to 0.5,

```
// assuming viewer is an instance of pv.Viewer
viewer.forEach(function(object) {
  object.setOpacity(0.5);
});
```

Adding a new color operation

A coloring operation is essentially an object with 3 methods:

- *colorFor* is called on every atom of the structure (or carbon alpha atoms for trace-based rendering styles.
- *begin* is called once before coloring a structure, allowing for preprocessing such as determining the number of chains in the structure. *begin* may be undefined, in which case it is ignored.
- *end* is called after coloring a structure, allowing or cleanup and freeing of resources. *end* may be undefined in which case it is ignored.

The following will add a new color operation which colors atoms based on their index. Atoms with an even index will be colored in red, atoms with an odd index will be colored in blue.

```
function evenOdd() {
  return new pv.color.ColorOp(function(atom, out, index) {
    // index + 0, index + 1 etc. are the positions in the output array
    // at which the red (+0), green (+1), blue (+2) and  alpha (+3)
    // components are to be written.
    if (atom.index() % 2 === 0) {
      out[index+0] = 1.0; out[index+1] = 0.0;
      out[index+2] = 0.0; out[index+3] = 1.0;
    } else {
      out[index+0] = 0.0; out[index+1] = 0.0;
      out[index+2] = 1.0; out[index+3] = 1.0;
    }
  });
}
```

Color Notations

Whenever a function takes a color as its argument, these colors can be specified in different ways:

- using RGB hex-code notation with an optional alpha value. Either as 6 (8 with alpha) hexadecimal numbers, or as 3 (4 with alpha) hexadecimal numbers. These color strings must be prefixed with a hash (`'#'`) sign. Examples: `'#badcode'`, or `'#abc'`.
- as an array of floating-point values. Each RGBA component is in the range between 0 and 1. The array must either be of length 3 (implicit alpha of 1.0) or length 4.
- as one of the hardcoded color strings

white	black	
grey	lightgrey	darkgrey
red	darkred	lightred
green	darkgreen	lightgreen
blue	darkblue	lightblue
yellow	darkyellow	lightyellow
cyan	darkcyan	lightcyan
magenta	darkmagenta	lightmagenta
orange	darkorange	lightorange

Examples

```
// These colors are all the same (guess what)
var color1 = [ 1, 0, 0 ]; // implicit alpha of 1
var color2 = [ 1, 0, 0, 1 ];
var color3 = 'red';
var color4 = '#f00';      // implicit alpha of f
var color5 = '#ff0000';   // implicit alpha of ff
var color6 = '#ff0000ff';
var color7 = '#f00f';
```

Custom Color Palettes

The default color palette can be replaced with custom color definitions. This is useful to match the colors to the stylesheet on your website, or to provide more color-blind frienly color palettes.

pv.color.**setColorPalette**(*palette*)

Replaces the current color palette with the specified palette. This will replace the color definitions itself as well as use the newly provided color definitions for the default gradients. All functions that accept color names will from now on us the new color definitions.

In case you want to change the color palette, it's best to do so before initializing the viewer component as it will make sure that all the code sees the new palette. Some of the methods translate the color names to RGB triplets and as such will not adjust to the new palette.

Arguments

- **palette** – a dictionary of color names (see example below).

Example

The following code block replaces the default palette with color-blind friendly colors.

```
var MY_COLOR_PALETTE = {
  white :      rgb.fromValues(1.0,1.0 ,1.0,1.0),
  black :      rgb.fromValues(0.0,0.0 ,0.0,1.0),
  grey :       rgb.fromValues(0.5,0.5 ,0.5,1.0),
  lightgrey :  rgb.fromValues(0.8,0.8 ,0.8,1.0),
  darkgrey :   rgb.fromValues(0.3,0.3 ,0.3,1.0),
  red :        rgb.hex2rgb("#AA00A2"),
  darkred :    rgb.hex2rgb("#7F207B"),
  lightred :   rgb.fromValues(1.0,0.5 ,0.5,1.0),
  green :      rgb.hex2rgb("#C9F600"),
  darkgreen :  rgb.hex2rgb("#9FB82E"),
```

```
  lightgreen :   rgb.hex2rgb("#E1FA71"), // or D8FA3F
  blue :         rgb.hex2rgb("#6A93D4"), // or 6A93D4
  darkblue :     rgb.hex2rgb("#284A7E"), // or 104BA9
  lightblue :    rgb.fromValues(0.5,0.5 ,1.0,1.0),
  yellow :       rgb.hex2rgb("#FFCC73"),
  darkyellow :   rgb.fromValues(0.5,0.5 ,0.0,1.0),
  lightyellow :  rgb.fromValues(1.0,1.0 ,0.5,1.0),
  cyan :         rgb.fromValues(0.0,1.0 ,1.0,1.0),
  darkcyan :     rgb.fromValues(0.0,0.5 ,0.5,1.0),
  lightcyan :    rgb.fromValues(0.5,1.0 ,1.0,1.0),
  magenta :      rgb.fromValues(1.0,0.0 ,1.0,1.0),
  darkmagenta :  rgb.fromValues(0.5,0.0 ,0.5,1.0),
  lightmagenta : rgb.fromValues(1.0,0.5 ,1.0,1.0),
  orange :       rgb.hex2rgb("#FFA200"), // or FFBA40
  darkorange :   rgb.fromValues(0.5,0.25,0.0,1.0),
  lightorange :  rgb.fromValues(1.0,0.75,0.5,1.0),
  brown :        rgb.hex2rgb("#A66A00"),
  purple :       rgb.hex2rgb("#D435CD")
};
pv.color.setColorPalette(MY_COLOR_PALETTE);
```

Molecular Structures

Molecular structures are represented by the *pv.mol.Mol()* class. While nothing restricts the type of molecules stored in an instance of *pv.mol.Mol()*, the data structure is optimized for biological macromolecules and follows the same hierarchical organizing principle. The lowest level of the hierarchy is formed by chains. The chains consist of one or more residues. Depending on the type of residues the chain holds, the chain is interpreted as a linear chain of residues, e.g. a polyeptide, or polynucleotide, or a collection of an unordered group of molecules such as water. In the former case, residues are ordered from N to C terminus, whereas in the latter the ordering of the molecules does not carry any meaning. Each residue consists of one or more atoms.

Tightly coupled to *pv.mol.Mol()* is the concept of structural subset, a *pv.mol.MolView()*. MolViews have the exact same interface than *pv.mol.Mol()* and in most cases behave exactly the same. Thus, from a user perspective it mostly does not matter whether one is working with a complete structure or a subset thereof. In the following, the APIs for the *pv.mol.Mol()* and *pv.mol.MolView()* classes are described together. Where differences exist, they are documented.

Obtaining and Creating Molecular Structures

The most common way to construct *molecules* is through one of the io functions. For example, to import the structure from a PDB file, use *pv.io.pdb()*. The whole structure, or a subset thereof can then be displayed on the screen by using one of the *rendering functions*.

The following code example fetches a PDB file from PDB.org imports it and displays the chain with name 'A' on the screen. For more details on how to create subsets, see *Creating Subsets of a Molecular Structure*.

```
$.ajax('http://pdb.org/pdb/files/'+pdbId+'.pdb')
.done(function(data) {
    // data contains the contents of the PDB file in text form
    var structure = pv.io.pdb(data);
    var firstChain = structure.select({chain: 'A'});
    viewer.cartoon('firstChain', firstChain);
});
```

Alternatively, you can create the structure *by hand*. That's typically not required, unless you are implementing your own importer for a custom format. The following code creates a simple molecule consisting of 10 atoms arranged along the x-axis.

```
var structure = new pv.mol.Mol();
var chain = structure.addChain('A');
for (var i = 0; i < 10; ++i) {
  var residue = chain.addResidue('ABC', i);
  residue.addAtom('X', [i, 0, 0], 'C');
}
```

Creating Subsets of a Molecular Structure

It is quite common to only apply operations (coloring, displaying) to subset of a molecular structure. These subsets are modelled as *views* and can be created in different ways.

- The most convenient way to create views is by using *pv.mol.Mol.select()*. Select accepts a set of predicates and returns a view containing only chains, residues and atoms that match the predicates.
- Alternatively for more complex selections, one can use *pv.mol.Mol.residueSelect()*, or *pv.mol.Mol.atomSelect()*, which evaluates a function on each residue/atom and includes residues/atoms for which the function returns true.
- Selection by distance allows to select parts of a molecule that are within a certain radius of another molecule.
- Views can be assembled manually through *pv.mol.MolView.addChain()*, *pv.mol.ChainView.addResidue()*, *pv.mol.ResidueView.addAtom()*. This is the most flexible but also the most verbose way of creating views.

Loading Molecular Structures

The following functions import structures from different data formats.

`pv.io.`**`pdb`**(*pdbData*[, *options*])
: Loads a structure from the *pdbData* string and returns it. In case multiple models are present (as designated by MODEL/ENDMDL), only the first is read. This behavior can be changed by passing `loadAllModels : true` to the options dictionary. In that case all models present in the string are loaded and returned as an array. Secondary structure and assembly information is assigned to all of the models.

The following record types are handled:

- *ATOM/HETATM* for the actual coordinate data. Alternative atom locations other than those labelled as *A* are discarded.
- *HELIX/STRAND* for assignment of secondary structure information.
- *REMARK 350* for handling of biological assemblies

`pv.io.`**`sdf`**(*sdfData*)
: Load small molecules from *sdfData* and returns them. In case multiple molecules are present, these molecules are returned as separate chains of the same *pv.mol.Mol()* instance.

  Currently, only a minimal set of information is extracted from SDF files:

  •atom position, element, atom name (set to the element)

  •connectivity information

  •the chain name is set to the structure title

pv.io.**fetchPdb**(*url*, *callback*[, *options*])
pv.io.**fetchSdf**(*url*, *callback*)
Performs an adjax request the provided URL and loads the data as a structure using either *pv.io.pdb()*, or *pv.io.sdf()*. Upon success, the callback is invoked with the loaded structure as the only argument. *options* is passed as-is to *pv.io.pdb()*.

Mol (and MolView)

class pv.mol.**Mol**()
Represents a complete molecular structure which may consist of multiple polypeptide chains, solvent and other molecules. Instances of mol are typically created through one of the io functions, e.g. *pv.io.pdb()*, or *pv.io.sdf()*.

class pv.mol.**MolView**()
Represents a subset of a molecular structure, e.g. the result of a selection operation. Except for a few differences, it's API is identical to *pv.mol.Mol()*.

pv.mol.Mol.**eachAtom**(*callback*)
pv.mol.MolView.**eachAtom**(*callback*)
Invoke callback for each atom in the structure. For example, the following code calculates the number of carbon alpha atoms.

```
var carbonAlphaCount = 0;
myStructure.eachAtom(function(atom) {
  if (atom.name() !== 'CA')
    return;
  if (!atom.residue().isAminoacid())
    return;
  carbonAlphaCount += 1;
});
console.log('number of carbon alpha atoms', carbonAlphaCount);
```

pv.mol.Mol.**eachResidue**(*callback*)
pv.mol.MolView.**eachResidue**(*callback*)
Invoke callback for each residue in the structure or view.

pv.mol.Mol.**full**()
pv.mol.MolView.**full**()
Convenience function that always links back to *pv.mol.Mol()*. For instances of *pv.mol.Mol()*, returns this directly, for instances of *pv.mol.MolView()* returns a reference to the *pv.mol.Mol()* the subset was derived from.

pv.mol.Mol.**atomCount**()
pv.mol.MolView.**atomCount**()
Returns the number of atoms in the structure, subset of structure.

pv.mol.Mol.**center**()
pv.mol.MolView.**center**()
Returns the geometric center of all atoms in the structure.

pv.mol.Mol.**chains**()
pv.mol.MolView.**chains**()
Returns an array of all chains in the structure. For *pv.mol.Mol()*, this returns a list of *pv.mol.Chain()* instances, for *pv.mol.MolView()* a list of *pv.mol.ChainView()* instances.

pv.mol.Mol.**select**(*what*)
pv.mol.MolView.**select**(*what*)
Returns a *pv.mol.MolView()* containing a filtered subset of chains, residues and atoms. *what* determines

how the filtered subset is created. It can be set to a predefined string for commonly required selections, or be set to a dictionary of predicates that have to match for a chain, residue or atom to be included in the result. Currently, the following predefined selections are accepted:

- *water*: selects residues with names HOH and DOD (deuteriated water).
- *protein*: returns all amino-acids found in the structure. Note that this might return amino acid ligands as well.
- *ligand*: selects all residues which are not water nor protein.
- *polymer*: selects all residues which are part of polymers. At the moment, this only returns nucleotides and peptides. Residues are considered to be part of polymers if they have a bond to at least one other residue of the same type. Note that the behavior of *polymer* is not identical *protein*. The latter also returns single amino acids.

Matching by predicate dictionary provides a flexible way to specify selections without having to write custom callbacks. A predicate is a condition which has to be fullfilled in order to include a chain, residue or atom in the results. Some of the predicates match against chain ,e.g. *cname*, others against residues, e.g. *rname*, and others against atoms, e.g. *ele*. When multiple predicates are specified in the dictionary, all of them have to match for an item to be included in the results.

Available Chain Predicates:

- *cname/chain*: A chain is included iff the chain name it is equal to the *cname/chain*. To match against multiple chain names, use the plural forms cnames/chains.

Available Residue Predicates:

- *rname*: A residue is included iff the residue name it is equal to *rname*. To match against multiple residue names, use the plural form rnames.
- *rindexRange* include residues at position in a chain in the interval *rindexRange[0]* and *rindexRange[1]*. The residue at *rindexRange[1]* is also included. Indices are zero-based.
- *rindices* includes residues at certain positions in the chain. Indices are zero based.
- *rnum* includes residues having the provided residue number value. Only the numeric part is honored, insertion codes are ignored. To match against multiple residue numbers, use the plural form *rnums*.
- *rnumRange* include residues with numbers between *rnumRange[0]* and *rnumRange[1]*. The residue with number *rnumRange[1]* is also included.

Available Atom Predicates:

- *aname* An atom is included iff the atom name it is equal to *aname*. To match against multiple atom names, use the plural form anames.
- *hetatm* An atom is included iff the atom hetatm flag matches the provided value.

Examples:

```
// select chain with name 'A' and all its residues and atoms
var chainA = myStructure.select({cname : 'A'});

// select carbon alpha  of chain 'A'. Residues with no carbon alpha will not be
// included in the result.
var chainACarbonAlpha = myStructure.select({cname : 'A', aname : 'CA'});
```

When none of the above selection mechanisms is flexible enough, consider using *pv.mol.Mol.residueSelect()*, or *pv.mol.Mol.atomSelect()*.

Returns *pv.mol.MolView()* containing the subset of chains, residues and atoms.

pv.mol.Mol.**selectWithin**(*structure*[, *options*])
pv.mol.MolView.**selectWithin**(*structure*[, *options*])

Returns an instance of *pv.mol.MolView()* containing chains, residues and atoms which are in spatial proximity to *structure*.

Arguments

- **structure** – *pv.mol.Mol()* or *pv.mol.MolView()* to which proximity is required.
- **options** – An optional dictionary of options to control the behavior of selectWithin (see below)

Options

•**radius** sets the distance cutoff in Angstrom. The default radius is 4.

•**matchResidues** whether to use residue matching mode. When set to true, all atom of a residue are included in result as soon as one atom is in proximity.

pv.mol.Mol.**residueSelect**(*predicate*)
pv.mol.MolView.**residueSelect**(*predicate*)

Returns an instance of *pv.mol.MolView()* only containing residues which match the predicate function. The predicate must be a function which accepts a residue as its only argument and return true for residues to be included. For all other residues, the predicate must return false. All atoms of matching residues will be included in the view.

Example:

```
var oddResidues = structure.residueSelect(function(res) {
  return res.index() % 2;
});
```

pv.mol.Mol.**atomSelect**(*predicate*)
pv.mol.MolView.**atomSelect**(*predicate*)

Returns an instance of *pv.mol.MolView()* only containing atoms which match the predicate function. The predicate must be a function which accepts an atom as its only argument and return true for atoms to be included. For all other atoms, the predicate must return false.

Example:

```
var carbonAlphas = structure.atomSelect(function(atom) {
  return res.name() === 'CA';
});
```

pv.mol.Mol.**addChain**(*name*)

Adds a new chain with the given name to the structure

Arguments

- **name** – the name of the chain

Returns the newly created *pv.mol.Chain()* instance

pv.mol.MolView.**addChain**(*chain*, *includeAllResiduesAndAtoms*)

Adds the given chain to the structure view

Arguments

- **chain** – the chain to add. Must either be a *pv.mol.ChainView()*, or *pv.mol.Chain()* instance.

- **includeAllResiduesAndAtoms** – when true, residues and atoms contained in the chain are directly added as new *pv.mol.ResidueView()*, *pv.mol.AtomView()* instances. When set to false (the default), the new chain view is created with an empty list of residues.

Returns the newly created *pv.mol.ChainView()* instance

pv.mol.Mol.**addResidues**(*residues*, *includeAllAtoms*)
Adds all residues to their respective chain

Arguments

- **residues** – list of new residues
- **includeAllAtoms** – when true, all atoms of the residue are directly added as new AtomViews to the residue. When set to false (the default), a new residue view is created with an empty list of atoms.

Returns a map of chain name to chain for the affected chains with new residues.

pv.mol.MolView.**addAtom**(*atom*)
Adds the given atom to the view. If the atom is already contained in the view, it is not added again. If an atom's residue or chain are not yet part of the view, they are added as well.

Arguments

- **atom** – the atom to add. Must either be a *pv.mol.AtomView()*, or *pv.mol.Atom()* instance.

Returns the newly created *pv.mol.AtomView()* instance, or the existing atom if the atom was already contained in the view.

pv.mol.MolView.**removeAtom**(*atom*, *removeEmptyResiduesAndChains*)
Remove the given atom from the view.

Arguments

- **atom** – The atom to remove must either be a *pv.mol.AtomView()*, or *pv.mol.Atom()* instance.
- **removeEmptyResiduesAndChains** – when true removes now-empty residues an chains from the view. When false, empty residues an chains remain in the view.

Returns true if the atom was part of the view an was removed, false if not.

pv.mol.Mol.**chain**(*name*)
pv.mol.MolView.**chain**(*name*)
Alias for *pv.mol.Mol.chainByName()*

pv.mol.Mol.**chainByName**(*name*)
pv.mol.MolView.**chainByName**(*name*)
Returns the chain with the given name. If no such chain exists, null is returned.

pv.mol.Mol.**chainsByName**(*names*)
pv.mol.MolView.**chainsByName**(*names*)
Returns the list of chains matching the specified names. In case a chain does not exist (or is not part of the view), the chain name is ignored, as if it were not specified.

Chain (and ChainView)

class pv.mol.**Chain**()
Represents either a linear chain of molecules, e.g. as in peptides or an unordered collection of molecules such

as water. New instances are created by calling *pv.mol.Mol.addChain()*.

class pv.mol.**ChainView**()
Represents a subset of a chain, that is a selected subset of residues and atoms. New instances are created and added to an existing *pv.mol.MolView()* instance by calling *pv.mol.MolView.addChain()*.

pv.mol.Chain.**name**()
pv.mol.ChainView.**name**()
The name of the chain. For chains loaded from PDB, the chain names are alpha-numeric and no longer than one character.

pv.mol.Chain.**residues**()
pv.mol.ChainView.**residues**()
Returns the list of residues contained in this chain. For *pv.mol.Chain()* instances, returns an array of *pv.mol.Residue()*, for *pv.mol.ChainView()* instances returns an array of *pv.mol.ResidueView()* instances.

pv.mol.Chain.**eachBackboneTrace**(*callback*)
pv.mol.ChainView.**eachBackboneTrace**(*callback*)
Invokes *callback* for each stretch of consecutive amino acids found in the chain. Each trace contains at least two amino acids. Two amino acids are consecutive when their backbone is complete and the carboxy C-atom and the nitrogen N could potentially form a peptide bond.

Arguments

- **callback** – a function which accepts the array of trace residues as an argument

pv.mol.Chain.**backboneTraces**()
pv.mol.ChainView.**backboneTraces**()
Convenience function which returns all backbone traces of the chain as a list. See *pv.mol.Chain.eachBackboneTrace()*.

pv.mol.Chain.**addResidue**(*name*, *number*[, *insCode*])
Appends a new residue at the end of the chain

Arguments

- **name** – the name of the residue, for example 'GLY' for glycine.
- **number** – the numeric part of the residue number
- **insCode** – the insertion code character. Defaults to '\0'.

Returns the newly created *pv.mol.Residue()* instance

pv.mol.Chain.**residueByRnum**(*rnum*)
pv.mol.ChainView.**residueByRnum**(*rnum*)
Returns the first residue in the chain with the given numeric residue number. Insertion codes are ignored. In case no residue has the given residue number, null is returned. This function internally uses a binary search when the residue numbers of the chain are ordered, and falls back to a linear search in case the residue numbers are unordered.

Returns if found, the residue instance, and null if no such residue exists.

pv.mol.Chain.**residuesInRnumRange**(*start*, *end*)
pv.mol.ChainView.**residuesInRnumRange**(*start*, *end*)
Returns the list of residues that have residue number in the range *start*, *end*. Insertion codes are ignored. This function internally uses a binary search to quickly determine the residues included in the range when the residue numbers in the chain are ordered, and falls back to a linear search in case the residue numbers are unordered.

Example:

```
// will contain residues with numbers from 5 to 10.
var residues = structure.chain('A').residuesInRnumRange(5, 10);
```

pv.mol.ChainView.**addResidue**(*residue*, *includeAllAtoms*)
Adds the given residue to the chain view

Arguments

- **residue** – the residue to add. Must either be a *pv.mol.ResidueView()*, or *pv.mol.Residue()* instance.
- **includeAllAtoms** – when true, all atoms of the residue are directly added as new AtomViews to the residue. When set to false (the default), a new residue view is created with an empty list of atoms.

Returns the newly created *pv.mol.ResidueView()* instance

Residue (and ResidueView)

class pv.mol.**Residue**()
Represents a residue, e.g. a logical unit of atoms, such as an amino acid, a nucleotide, or a sugar. New residues are created and added to an existing *pv.mol.Chain()* instance by calling *pv.mol.Chain.addResidue()*.

class pv.mol.**ResidueView**()
Represents a subset of a residue, e.g. a subset of the atoms the residue contains. New residue views are created and added to an existing *pv.mol.ChainView()* by calling *pv.mol.ChainView.addResidue()*.

pv.mol.Residue.**name**()
pv.mol.ResidueView.**name**()
Returns the three-letter-code of the residue, e.g. GLY for glycine.

pv.mol.Residue.**isWater**()
pv.mol.ResidueView.**isWater**()
Returns true when the residue is a water molecule. Water molecules are recognized by having a one-letter-code of HOH or DOD (deuteriated water).

pv.mol.Residue.**isAminoAcid**()
pv.mol.ResidueView.**isAminoAcid**()
Returns true when the residue is an amino acid. Residues which have the four backbone atoms N, CA, C, and O are considered as amino acids, all others not.

pv.mol.Residue.**num**()
pv.mol.ResidueView.**num**()
Returns the numeric part of the residue number, ignoring insertion code.

pv.mol.Residue.**index**()
pv.mol.ResidueView.**index**()
Returns the index of the residue in the chain.

pv.mol.Residue.**atoms**()
pv.mol.ResidueView.**atoms**()
Returns the list of atoms of this residue. For *pv.mol.Residue()*, returns an array of *pv.mol.Atom()* instances, for *pv.mol.ResidueView()*, resturns an array of *pv.mol.AtomView()* instances.

pv.mol.Residue.**atom**(*nameOrIndex*)
pv.mol.ResidueView.**atom**(*nameOrIndex*)
Get a particular atom from this residue. *nameOrResidue* can either be an integer, in which case the atom at that index is returned, or a string, in which case an atom with that name is searched and returned.

Returns For *pv.mol.Residue()*, a *pv.mol.Atom()* instance, for *pv.mol.ResidueView()*, a *pv.mol.AtomView()* instance. If no matching atom could be found, null is returned.

pv.mol.Residue.**addAtom**(*name*, *pos*, *element*)
Adds a new atom to the residue.

Arguments

- **name** – the name of the atom, for example CA for carbon-alpha
- **pos** – the atom position
- **element** – the atom element string, e.g. 'C' for carbon, 'N' for nitrogen

Returns the newly created *pv.mol.Atom()* instance

pv.mol.ResidueView.**addAtom**(*atom*)
Adds the given atom to the residue view

Returns the newly created *pv.mol.AtomView()* instance

Atom (and AtomView)

class pv.mol.**Atom**()
Stores properties such as positions, name element etc of an atom. Atoms always have parent residue. New atoms are created by adding them to an existing residue through *pv.mol.Residue.addAtom()*.

class pv.mol.**AtomView**()
Represents a selected atom as part of a view. New atom views are created by adding them to an existing *pv.mol.ResidueView()* through *pv.mol.ResidueView.addAtom()*.

pv.mol.Atom.**name**()
pv.mol.AtomView.**name**()
The name of the atom, e.g. CA for carbon alpha.

pv.mol.Atom.**element**()
pv.mol.AtomView.**element**()
The element of the atom. When loading structures from PDB, the atom element is taken *as is* from the element column if it is not empty. In case of an empty element column, the element is guessed from the atom name.

pv.mol.Atom.**bonds**()
pv.mol.AtomView.**bonds**()
Returns a list of all bonds this atom is involved in.

pv.mol.Atom.**pos**()
pv.mol.AtomView.**pos**()
The actual coordinates of the atom.

pv.mol.Atom.**isHetatm**()
pv.mol.AtomView.**isHetatm**()
Returns true when the atom was imported from a HETATM record, false if not. This flag is only meaningful for structures imported from PDB files and will return false for other file formats.

pv.mol.Atom.**occupancy**()
pv.mol.AtomView.**occupancy**()
Returns the occupancy of this atom. In case this value is not available, null will be returned.

pv.mol.Atom.**tempFactor**()

pv.mol.AtomView.**tempFactor**()
Returns the temperature factor (aka B-factor) of this atom. In case this value is not available, null will be returned.

Bond

class pv.mol.**Bond**()

to be written...

Superposition of structures

PV has support for pair-wise superposition of structures through the *pv.mol.superpose()* function. The function takes two structures (subject and reference) and calculates a transformation that transforms subject's atoms to the reference.

pv.mol.**superpose**(*subject*, *reference*)

Arguments

- **subject** (*Mol()* or *MolView()*) – the structure to be transformed.
- **reference** (*Mol()* or *MolView()*) – the structure to superpose onto

Both structures must have the exact same number of atoms and contain at least 3 atoms. If any of these conditions is violated, no superposition is performed and false is returned. The atoms in the two structures are paired in order they appear in the two structures. For creating matching structures betwen reference an subject consider using *matchResiduesByNum()*, or *matchResiduesByIndex()*.

Upon success, all atoms in *subject* are shifted and rotated according to the calculated transformation matrix. When *subject* is a view, atoms that are part of the full structure but not part of the view are transformed as well. This allows to use a subset of atoms for the superposition, while still transforming all of the *subject* atoms.

pv.mol.**matchResiduesByIndex**(*inA*, *inB*[, *atoms*])
pv.mol.**matchResiduesByNum**(*inA*, *inB*[, *atoms*])
Helper functions to create views with matching number of atoms that can be used as input to the superpose function.

In case the structure contains multiple chains, the chains are matched by their index. When the two structures do not contain the same number of chains, chains that do not have a corresponding chain in the other structure are discarded.

The matching of residues in a chain depends on the exact function used:

- *matchResiduesByIndex()* matches residues in a chain by their index. It is required that the matched chains have the same number of residues. If this condition does not hold, matching is aborted and null returned.
- *matchResiduesByNum()* matches residues by their residues number. The residues in the output view appear in the same order they appear in the first structure.

For each matched residue pair only atoms present in both residues are included. When the *atoms* parameter is provided, the atoms are further filtered by the specified criteria. When *atoms* is set to `'all'` or `null`, all atoms that are present in both residues are included in the result. When *atoms* is `'backbone'`, only backbone atoms are included. Otherwise *atoms* is a comma-separated list of atoms names to be included.

Arguments

- **inA** (*Mol()* or *MolView()*) – First structure

- **inB** (*Mol()* or *MolView()*) – Second structure
- **atoms** – The subset of atoms to be included in the two views. Must either be null, string or a list of strings.

Returns An array of length two containing the two created views as `[outA, outB]`.

PV Usage Code Samples

Display protein together with ligands

This sample shows how to render a protein in cartoon mode and display the contained ligands as balls-and-sticks. For this particular example, we have chosen the dengue methyl transferase structure (1r6a) which contains a s-adenosyl homocysteine and the inhibitor ribavirin 5' triphosphate. Source Code

```
<script>
var parent = document.getElementById('viewer');
var viewer = pv.Viewer(parent,
                       { width : 300, height : 300, antialias : true });
pv.io.fetchPdb('_static/1r6a.pdb', function(structure) {
  // select the two ligands contained in the methyl transferase by name, so
  // we can display them as balls and sticks.
  viewer.on('viewerReady', function() {
    var ligand = structure.select({rnames : ['RVP', 'SAH']});
    viewer.ballsAndSticks('ligand', ligand);
    // display the whole protein as cartoon
    viewer.cartoon('protein', structure);

    // set camera orientation to pre-determined rotation, zoom and
    // center values that are optimal for this very protein
    var rotation = [
      0.1728139370679855, 0.1443438231945038,  0.974320650100708,
      0.0990324765443802, 0.9816440939903259, -0.162993982434272,
      -0.9799638390541077, 0.1246569454669952,  0.155347332358360
    ];
    var center = [6.514, -45.571, 2.929];
    viewer.setCamera(rotation, center, 73);
  });
});
</script>
```

Display an NMR ensemble

In this sample we are going to use the *loadAllModels* option of the *PDB parser* to load all structures present in a multi-model PDB file. The models are then displayed together in the viewer using the *cartoon render mode*. Source Code

```
<script>
var viewer = pv.Viewer(document.getElementById('viewer'),
                       { width : 300, height : 300, antialias : true });

pv.io.fetchPdb('_static/1nmr.pdb', function(structures) {
    // put this in the viewerReady block to make sure we don't try to add the
    // object before the viewer is ready. In case the viewer is completely
    // loaded, the function will be immediately executed.
```

```
    viewer.on('viewerReady', function() {
      var index = 0;
      structures.forEach(function(s) {
        viewer.cartoon('structure_' + (index++), s);
      });
      // adjust center of view and zoom such that all structures can be seen.
      var rotation = pv.viewpoint.principalAxes(viewer.all()[0]);
      viewer.setRotation(rotation)
      viewer.autoZoom();
    });
},  { loadAllModels : true });
</script>
```

Highlight atom under mouse cursor

This sample shows how to highlight the atom under the mouse cursor by changing it's color to red and display it's name.

Note: While it's possible to temporarily change the color for highlighting purposes, it's recommended to use the *selection highlighting functionality* added in in PV 1.8.0 instead.

This sample requires PV 1.7.0 and higher to work as it relies on functionality that was added in 1.7.0. Source Code

```
<div id='picked-atom-name' style='text-align:center;'> </div>
<script>
var parent = document.getElementById('viewer');
var viewer = pv.Viewer(parent,
                       { width : 300, height : 300, antialias : true });


function setColorForAtom(go, atom, color) {
    var view = go.structure().createEmptyView();
    view.addAtom(atom);
    go.colorBy(pv.color.uniform(color), view);
}

// variable to store the previously picked atom. Required for resetting the color
// whenever the mouse moves.
var prevPicked = null;
// add mouse move event listener to the div element containing the viewer. Whenever
// the mouse moves, use viewer.pick() to get the current atom under the cursor.
parent.addEventListener('mousemove', function(event) {
    var rect = viewer.boundingClientRect();
    var picked = viewer.pick({ x : event.clientX - rect.left,
                               y : event.clientY - rect.top });
    if (prevPicked !== null && picked !== null &&
        picked.target() === prevPicked.atom) {
      return;
    }
    if (prevPicked !== null) {
      // reset color of previously picked atom.
      setColorForAtom(prevPicked.node, prevPicked.atom, prevPicked.color);
    }
    if (picked !== null) {
      var atom = picked.target();
```

```
      document.getElementById('picked-atom-name').innerHTML = atom.qualifiedName();
      // get RGBA color and store in the color array, so we know what it was
      // before changing it to the highlight color.
      var color = [0,0,0,0];
      picked.node().getColorForAtom(atom, color);
      prevPicked = { atom : atom, color : color, node : picked.node() };

      setColorForAtom(picked.node(), atom, 'red');
    } else {
      document.getElementById('picked-atom-name').innerHTML = ' ';
      prevPicked = null;
    }
    viewer.requestRedraw();
});
pv.io.fetchPdb('_static/1crn.pdb', function(structure) {
    // put this in the viewerReady block to make sure we don't try to add the
    // object before the viewer is ready. In case the viewer is completely
    // loaded, the function will be immediately executed.
    viewer.on('viewerReady', function() {
      var go = viewer.cartoon('structure', structure);
      // adjust center of view and zoom such that all structures can be seen.
      viewer.autoZoom();
    });
});
</script>
```

Select atoms/residues using mouse and keyboard

This sample demonstrates how to select residues with mouse an keyboard by using the selection highlighting introduced in PV 1.8 to visually highlight a subset of residues and atoms.

Usage

- **click:** select clicked residue/atom
- **shift click:** select clicked residue/atom an extend selection
- **return**: center view on selected atoms

Source Code

```
<script>
viewer = pv.Viewer(document.getElementById('viewer'), {
    width : '300', height: '300', antialias : true,
    outline : true, quality : 'medium', style : 'hemilight',
    background : 'white', animateTime: 500,
    selectionColor : '#f00'
});

pv.io.fetchPdb('_static/1crn.pdb', function(s) {
  viewer.on('viewerReady', function() {
    viewer.cartoon('crambin', s);
    viewer.autoZoom();
  });
});
```

```
// register a keypressed listener and check for return key
// presses. Whenever the return key is pressed,  the camera zooms
// in on the currently selected residues.
document.addEventListener('keypress', function(ev) {
  if (ev.keyCode === 13) {
    var allSelections = [];
    viewer.forEach(function(go) {
      if (go.selection !== undefined) {
        allSelections.push(go.selection());
      }
    });
    viewer.fitTo(allSelections);
  }
});

viewer.on('click', function(picked, ev) {
  if (picked === null || picked.target() === null) {
    return;
  }
  // don't to anything if the clicked structure does not have an atom.
  if (picked.node().structure === undefined) {
    return;
  }
  // when the shift key is pressed, extend the selection, otherwise
  // only select the clicke atom.
  var extendSelection = ev.shiftKey;
  var sel;
  if (extendSelection) {
    var sel = picked.node().selection();
  } else {
    var sel = picked.node().structure().createEmptyView();
  }
  // in case atom was not part of the view, we have to add it, because
  // it wasn't selected before. Otherwise removeAtom took care of it
  // and we don't have to do anything.
  if (!sel.removeAtom(picked.target(), true)) {
    sel.addAtom(picked.target());
  }
  picked.node().setSelection(sel);
  viewer.requestRedraw();
});

</script>
```

Displaying a static label on top of the viewer

This sample demonstrates how to add a custom static label on top of the viewer.

The label is added as the first child of the element containing the viewer and is positioned with absolute coordinates. The background is set to have an alpha component of 0, so that the viewer shines through the label.

In the sample code, the label is inserted using JavaScript. That's not a requirement and you can also just define it inside the viewer element directly. The important part is that it is placed using absolute coordinates on top of the viewer. Source Code

```
<style>
.static-label {
```

```
  position:absolute;
  background: #0000;
  text-align:right;
  z-index: 1;
  font-weight:bold;
  width:290px;
}
</style>
<script>
var parent = document.getElementById('viewer');
var staticLabel = document.createElement('div');
staticLabel.innerHTML = 'crambin';
staticLabel.className = 'static-label';
parent.appendChild(staticLabel);
viewer = pv.Viewer(parent, {
    width : '300', height: '300', antialias : true,
    outline : true, quality : 'medium', style : 'hemilight',
    background : 'white', animateTime: 500,
    selectionColor : '#f00'
});

pv.io.fetchPdb('_static/1crn.pdb', function(s) {
  viewer.on('viewerReady', function() {
    viewer.cartoon('crambin', s);
    viewer.autoZoom();
  });
});

</script>
```

Displaying a label on an atom

This sample demonstrates how to add a label in the 3d scene to annotate an atom. Source Code

```
<script>
var parent = document.getElementById('viewer');
viewer = pv.Viewer(parent, {
    width : '300', height: '300', antialias : true,
    outline : true, quality : 'medium', style : 'hemilight',
    background : 'white', animateTime: 500,
    selectionColor : '#f00'
});

pv.io.fetchPdb('_static/1crn.pdb', function(s) {
  viewer.on('viewerReady', function() {
    viewer.cartoon('crambin', s);
    var carbonAlpha = s.atom('A.31.CA');
    // override a few default options to show their effect
    var options = {
     fontSize : 16, fontColor: '#f22', backgroundAlpha : 0.4
    };
    viewer.label('label', carbonAlpha.qualifiedName(),
                 carbonAlpha.pos(), options);
    viewer.autoZoom();
  });
});
```

```
</script>
```

Adding custom geometry to the 3D scene

This sample demonstrates how to add custom geometry to the 3d scene. The code places small spheres on a helical structure and colors them using a gradient. Source Code

```
<script>
var parent = document.getElementById('viewer');
viewer = pv.Viewer(parent, {
    width : '300', height: '300', antialias : true,
    outline : true, quality : 'high', style : 'hemilight',
});
viewer.on('viewerReady', function() {
  var helix = viewer.customMesh('custom');
  for (var i = -50; i < 50; ++i) {
    var x = Math.cos(i * 0.4);
    var y = i * 0.1;
    var z = Math.sin(i * 0.4);
    var color = i * 0.01 + 0.5;
    // add sphere at the given position with a radius of 0.1
    helix.addSphere([x,y,z], 0.1, { color : [color, color, 0]});

    // add a capped tube  in the center of the helix with a
    // radius of 0.1
    helix.addTube([0, -5, 0], [0, 5, 0], 0.1,
                  { cap : true, color : 'blue' });

    // set zoom to a pre-determined value. Alternatively,
    // viewer.autoZoom() can be used.
    viewer.setZoom(14);
  }
});

</script>
```

Measure distance between two atoms

This sample shows how add support for measuring the distance between two atoms.

Usage

Click on two atoms to measure the distance between them. After clicking the second atom, a line will be drawn that connects the two atoms with the distance displayed in a label. Source Code

```
<script>
var parent = document.getElementById('viewer');
var viewer = pv.Viewer(parent,
                       { width : 300, height : 300, antialias : true,
                         selectionColor : 'red' });

pv.io.fetchPdb('_static/1crn.pdb', function(structure) {
```

```
  viewer.on('viewerReady', function() {
    viewer.cartoon('crambin', structure);
    viewer.autoZoom();
  });
});

var lastAtom = null;


// register click handler that does all the distance-measure-foo
viewer.addListener('click', function(picked) {
  if (picked === null) return;
  var target = picked.target();
  if (target.qualifiedName === undefined) {
    return;
  }
  var node = picked.node();
  var view = node.structure().createEmptyView();
  if (lastAtom !== null) {
    // remove distance-related objects from previous distance
    // measurements.
    viewer.rm('dist.*');
    var g = viewer.customMesh('dist.line');
    var midPoint = pv.vec3.clone(lastAtom.pos());
    pv.vec3.add(midPoint, midPoint, target.pos());
    pv.vec3.scale(midPoint, midPoint, 0.5);
    // add a tube to connect the two atoms
    g.addTube(lastAtom.pos(), target.pos(), 0.1,
              { cap : true, color : 'white' });
    var d = pv.vec3.distance(lastAtom.pos(), target.pos());
    var l = viewer.label('dist.label', d.toFixed(2), midPoint);
    lastAtom = null;
  } else {
    lastAtom = target;
    view.addAtom(target);
  }
  node.setSelection(view);
  viewer.requestRedraw();
});
</script>
```

Color structure by custom property

This sample demonstrates how to color a structure by using a user-defined property on residues. In analogy, these properties may be defined on atoms as well. Source Code

```
<script>
viewer = pv.Viewer(document.getElementById('viewer'), {
    width : '300', height: '300', antialias : true,
    outline : true, quality : 'medium', style : 'hemilight',
    background : 'white', animateTime: 500,
    selectionColor : '#f00'
});

viewer.on('viewerReady', function() {
  pv.io.fetchPdb('_static/1crn.pdb', function(structure) {
```

```
    // for demonstration purposes, define a property funky on all residues
    // that cycles through the values 0-9.
    var index = 0;
    structure.eachResidue(function(r) {
      r.setProp('funky', index++ % 10);
    });
    // use pv.color.byResidueProp in combination with the "funky" property
    // defined above. Analogously to color by atom property, use
    // pv.color.byAtomProp()
    viewer.cartoon('structure', structure, {
      color : pv.color.byResidueProp('funky')
    });
    viewer.fitTo(structure);
  });
});
</script>
```

PV for developers documentation

How to Contribute

Contributions of any kind (code, documentation, bug reports) are more than welcome.

Coding conventions

Apart from the basic rules listed below, there is no detailed style guide. In doubt, just look at the existing code.

- always put braces around if/else/while/for statements.
- and indent is two spaces
- camelCase your variables and function names
- use an _ prefix for your private variables that should not be accessed from outside the class
- wrap code at 80 characters
- use === and !== for comparisons

Commits

Use descriptive commit messages using the following format:

- a short single-line summary, e.g. “add transparency for mesh and line geoms”. This line summary should not exceed 60-70 characters.
- optionally a block of text that describes the change in more detail, e.g.

  > color information is now stored as an RGBA quadruplet to accomodate one alpha value for each vertex. Coloring operations have grown the ability to specify alpha values. In case they are omitted, they default to 1.0 (fully opaque) structure.

  the block should be wrapped at 80 characters.

In case you are submitting a larger feature/bugfix, split your work into multiple commits. The main advantage is that your change becomes easier to review.

Before submitting

Before submitting, or sending the pull request

- make sure that there are no unrelated changes checked in.
- run grunt to check for any coding convention violations and in general make sure that grunt is still able to minify your code without problems.
- in case you are adding new functionality, make sure you area adding it to the API documentation
- make sure you don't check-in changes to bio-pv.min.js. These changes are very likely to cause conflicts.

How to release a new version of PV

These are the steps to release a new version of PV:

Release Testing

- Check that all unit tests pass
- Check that the samples in the documentation work
- Check that all samples in the demo work
- Check that the release note contain all major changes
- Update version number
  - Update version number in release notes (README.md)
  - Update version number in package.json
  - Update version number in doc/conf.py
- Create NPM package with `` `npm pack` ``.
- Test that the npm package works by unzipping it in a separate directory and running all the snippets

```
cp bio-pv-$version.tgz /tmp
cd /tmp
open -W bio-pv-$version.tgz
cd package
biojs-sniper
open http://localhost:9090/snippets
```

Release Publishing

- Publish the package to npm: `npm publish`
- tag the release using `git tag v$version -m "tagging v$version"`
- Upload the release package to github releases
- Create a readthedocs documentation version for the new tag
- Set the tag as the default version

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

P