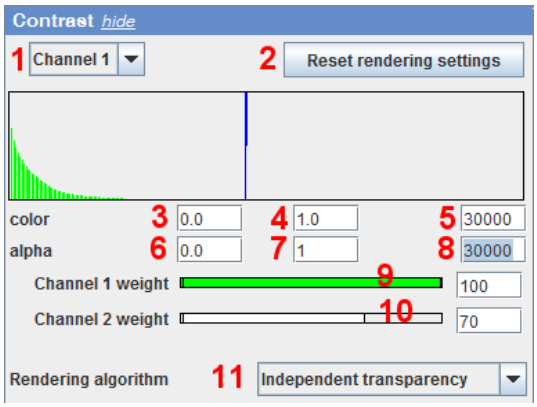
Manual

1 3D RENDERER

1.1 CONTRAST



It is recommended to adjust brightness and contrast (i.e. a channels minimum and maximum displayed value) appropriately for all image channels before starting the 3D Animation plugin. The 3D plugin will pick these values up and use them as default values, which can then be fine-tuned to adjust the rendering outcome to one's needs.

Like contrast adjustments in ImageJ/Fiji, the intensity contrast for the 3D rendering is specified by a minimum (3) and maximum (5) displayed intensity value for each channel. Additionally, the input interval between minimum and maximum can be mapped to the output interval non-linearly using a gamma parameter (4). Opposed to 2D contrast adjustment, for 3D rendering each pixel is transparent to some degree, depending typically on its intensity. This dependency is typically called the transfer function. Analogously to intensity contrast, the transfer function is specified by a minimum value (6) (for full transparency), a maximum value (8) (fully opaque) and a gamma parameter (7) for non-linear mapping of the interval in-between.

The channel weight sliders (9, 10) are used to adjust the contributions of each channel to the final output image. Double-clicking the weight sliders opens a color chooser dialog for adjusting a channel's color and the background color of the 3D canvas.

The combo box (1) at the top is used for switching between channels. Additionally, there is a button (2) at the top right to reset the rendering settings to the default values.

Three different rendering algorithms are implemented (11):

- Independent transparency: Each channel is processed individually, without influencing the other channels.
- Combined transparency: The transparencies of multiple channels influence each other
- Maximum intensity projection: There is no transparency involved at all, the maximum intensity of each channel along the ray of sight is taken for the output pixel

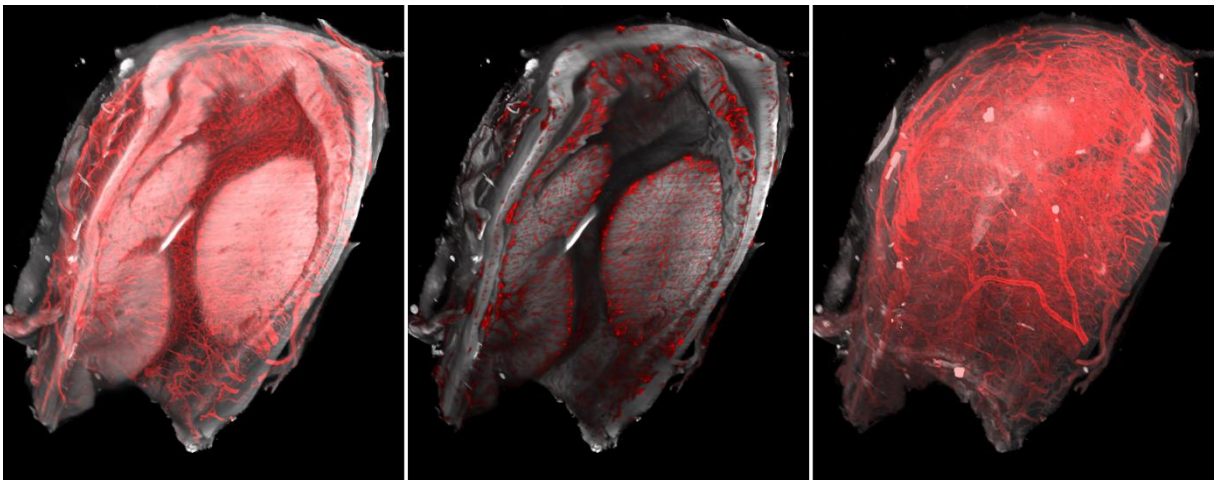
In more detail: The independent and combined transparency modes both use standard front-to-back ray-casting with the following formulas:

$$c = (1 - \alpha) * \alpha_i * c_i$$
$$\alpha = (1 - \alpha) * \alpha_i$$

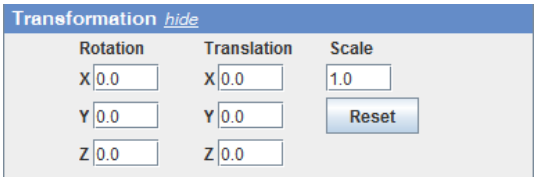
Where c is the output intensity of a ray (i.e. pixel), α is its output opacity, and c_i and α_i are the corresponding input values at discrete sampling positions along each ray.

Independent transparency mode applies these formulas separately for each channel, combined transparency mode uses sums up the values of α_i across all channels before advancing to the next ray sample.

Here is a comparison of the different rendering algorithms (left: independent transparency, middle: combined transparency, right: maximum intensity projection; contrast settings have been adjusted to the corresponding algorithm):



1.2 TRANSFORMATION

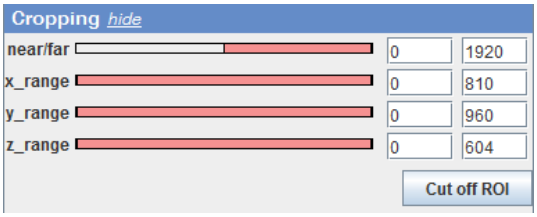


The transformation panel displays the transformation parameters of the rendered 3D object and provides also means for entering exact values. An alternative way for transforming an object is interactively via the mouse (the HAND tool must be selected in the ImageJ/Fiji menu bar):

- Dragging with the left mouse button rotates an object. If the alt key is pressed, rotation is restricted to either the horizontal or the vertical axis.
- Dragging with the left mouse button and holding the SHIFT key pressed translates an object
- The mouse wheel is used for zooming. When zooming with the mouse, the object is automatically translated such that the mouse pointer keeps pointing at the same structure, i.e. one zooms into the mouse pointer.

There is a reset button which resets all transformation parameters to their default values.

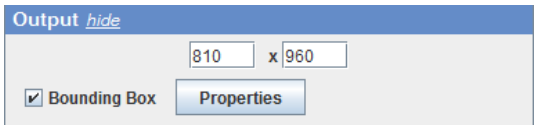
1.3 CROPPING



The sliders for x, y and z crop the data along their original axes. The near/far slider restricts rendering along the view direction, i.e. rendering starts a certain distance away (from the user's eyes) and only continues for a certain distance.

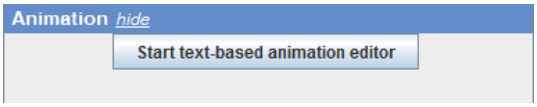
Furthermore, any of the selection tools offered by ImageJ/Fiji can be used to draw a selection into the 3D rendering window. Clicking the 'Cut off ROI' button will then remove the parts of the volume which are currently projected into the selection area.

1.4 OUTPUT



In the output panel, the size of the rendered image can be modified. Additionally, display of the bounding box of the rendered object can be turned on and off, and its parameters (line width and color) can be adjusted (clicking the 'Properties' button). If the bounding box check box is selected, the front clipping plane (see the Cropping section above) will also be indicated (its intersection with the bounding box, to be precise).

1.5 ANIMATION



Currently, there is only a single button in the animation panel, which will show the Animation Editor (see below).

2 CREATING ANIMATIONS

2.1 CREATING A SIMPLE ROTATION

To start composing animations, click on “Start text-based animation editor” button (see above). A very basic example for an animation is a plain rotation around a single axis. The appropriate animation text is

```
From frame 0 to frame 180 rotate by 360 degrees horizontally
```

To actually render the animation, click on the “Run” button. A new movie window opens, and new frames are added as they were rendered.

For a quick preview of the movie, adjust the frame rate (▸ Image ▸ Stacks ▸ Animation ▸ Animation Options…, or simply right-click the play button next to the time slider) and start playback by clicking on the play button.

2.2 EXPORTING MOVIES

There are a number of ways to export movies. My favorite way is to save them as uncompressed AVI (using ImageJ’s ▸ File ▸ Save As ▸ AVI… command) and then encode this temporary video file using a proper video encoding software such as ffmpeg. The command line that I am using is one of the following two versions

```
ffmpeg.exe -i input.avi -vcodec wmv2 -qscale:v 1 -f asf output.avi

ffmpeg.exe -i input.avi -vcodec mpeg4 -qscale:v 1 output.mp4
```

The first line seems to work also for older versions of PowerPoint.

For Microsoft Windows, there exists also a nice GUI frontend for FFMPEG, ffe.

2.3 STRUCTURE OF THE ANIMATION LANGUAGE

Each sentence (i.e. instruction) consists of a time interval, an action, corresponding action parameters and optionally of one of the easing keywords for non-linear transitions.

2.3.1 The time interval

The time interval is given in number of frames, there exist two variants, for either specifying a single time point or a time range:

```
From frame <begin> to frame <end> …

At frame <time> …
```

2.3.2 Action

Actions include rotations, translations, zoom and rendering parameter changes.

Rotations:

```
… rotate by <angle> degrees horizontally …

… rotate by <angle> degrees vertically …

… rotate by <angle> degrees around (axis_x, axis_y, axis_z)
```

Translations:

```
… translate horizontally by <x> …

… translate vertically by <y> …

… translate by (x, y, z) …
```

Zoom:

```
… zoom by a factor of <r> …
```

Change:

- general properties:

```
… change bounding box min x to <n> …
… change bounding box max x to <n> …
… change bounding box x to (min, max) …

… change bounding box min y to <n> …
… change bounding box max y to <n> …
… change bounding box y to (min, max) …

… change bounding box min z to <n> …
… change bounding box max z to <n> …
… change bounding box z to (min, max) …

… change front clipping to <n> …
… change back clipping to <n> …
… change front/back clipping to <n> …
```

- channel-specific properties:

```
… change channel <c> min intensity to <n> …
… change channel <c> max intensity to <n> …
… change channel <c> intensity gamma to <n> …
… change channel <c> intensity to (min, max, gamma) …

… change channel <c> min alpha to <n> …
… change channel <c> max alpha to <n> …
… change channel <c> alpha gamma to <n> …
… change channel <c> alpha to (min, max, gamma) …

… change channel <c> color to (r, g, b) …

… change channel <c> weight to <n> …
```

2.3.3 Easing

```
… linearly          (no acceleration)

… ease-in           (slow start, then accelerating)

… ease-out          (linear start, then decelerating)

… ease-in-out       (slow start and slow end)

… ease              (same as ease-in-out, but not as dramatic)
```

2.3.4 Multiple instructions

Oftentimes, multiple consecutive instructions share the same start, i.e. when several instructions are active during the same time interval. In that case, the common prefix (the time interval) can be shared between the two instructions, by placing it on an extra line that ends with a colon. Instructions which share this prefix should start with a dash:

```
From frame 0 to frame 180:
- rotate by 360 degrees horizontally
- change channel 1 weight to 0
```

In this example, between frames 0 and 180 the object performs a full 360-degree rotation, while during the same time period, the weight of channel 1 continuously decreases to 0.

2.4 EXAMPLE

This example demonstrates step-by-step how the animation of a human cornea is composed.

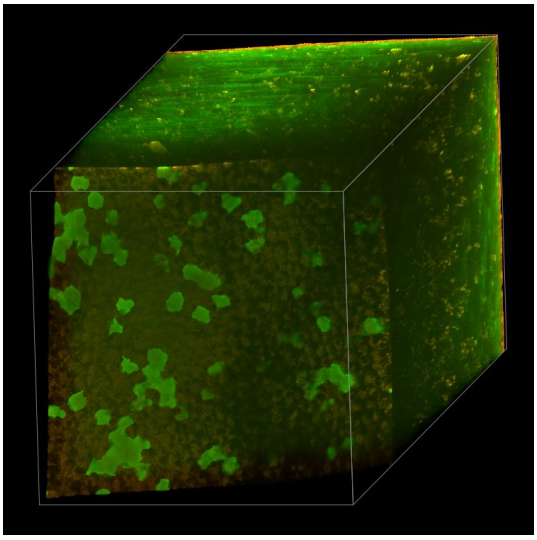
2.4.1 Step 1: Find a nice starting transformation

The first step is to interactively rotate, translate and zoom to get a nice starting position. For the animation here, I chose a 10-degree rotation around both the x- and the y-axis, and zoomed out by a factor of 0.6 so that the entire object fits into the 3D panel.

One could now type in the corresponding lines into the Animation Editor:

```
At frame 0:
- rotate by 10 degrees vertically
- rotate by 10 degrees horizontally
- zoom by a factor of 0.6
```

Clicking on the “Run” button renders a single image:



Instead of typing the lines above manually, we can also select ▸ Record ▸ Record transformation from the Animation Editor’s menu bar, which will insert these lines automatically.

2.4.2 Adding a full rotation

Next, we add a 360-degree rotation around the x-axis:

```
At frame 0:
- rotate by 10 degrees vertically
- rotate by 10 degrees horizontally
- zoom by a factor of 0.6

From frame 0 to frame 90 rotate by 180 degrees vertically
```

Clicking the “Run” button again now renders a movie with 90 frames. Starting from the initial orientation which we adjusted in the last paragraph, the cornea is now rotated around (the view’s) x-axis. Note that transformations (here rotations) are applied in the order they appear in the text. First, the 10-degree rotations are applied, and then the cornea is rotated around the (already rotated) x-axis. If we wanted the cornea to rotate around the original x-axis, we needed to reverse the order of the transformations:

```
From frame 0 to frame 90 rotate by 180 degrees vertically

At frame 0:
- rotate by 10 degrees vertically
- rotate by 10 degrees horizontally
- zoom by a factor of 0.6
```

2.4.3 Dynamic zooming using a macro

Unfortunately, the stack of the cornea is too big, so that only parts of it are visible once it turned by 90 degrees. We could zoom out more, but then lots of space is wasted at the animation beginning. What we want to add here is a dynamic zoom: We start with a zoom of 0.6, as we are approaching 90 degrees, we zoom further out to 0.3, as we rotate further towards 180 degrees, we zoom in again to 0.6, etc.

We can express the zoom factor as a function of time (measured in frames). A formula that works well is

$$zoom = 0.6 - 0.3 * \left|\sin\frac{2\pi t}{180}\right|$$

where the unit of *t* is frames.

You can easily verify the formula by setting at an angle of 0° (*t* = 0, *zoom* = 0.6), 90° (*t* = 45, *zoom* = 0.3, 180° (*t* = 90, *zoom* = 0.6), 270° (*t* = 135, *zoom* = 0.3) and 360° (*t* = 180, *zoom* = 0.6).

To implement this, replace the last line with

```
From frame 0 to frame 90 zoom by a factor of zoom
```

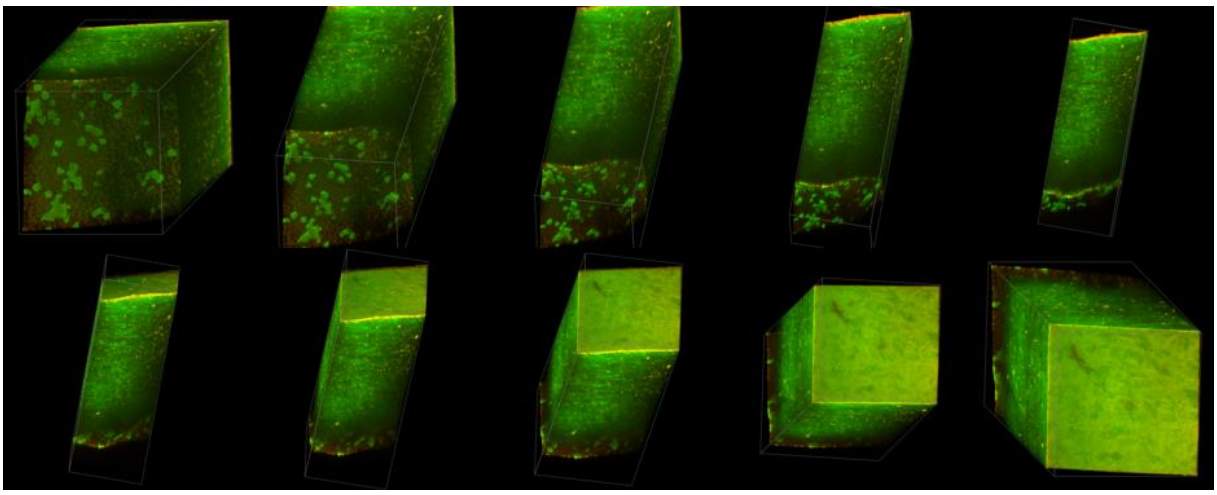
Instead of a value for zoom, we just enter an arbitrary word (here: zoom). The Animation Editor recognizes that there is a name instead of a number and expects an ImageJ macro with the same name, taking the time (i.e. frame) as a single parameter. Once you press the "Enter" key at the end of the line the corresponding macro function body is inserted:

```
script
function zoom(t) {
	return 0;
}
```

The "script" keyword indicates that the following lines contain a macro.

After entering the formula above, the entire script reads now:

```
At frame 0:
- rotate by 10 degrees vertically
- rotate by 10 degrees horizontally

From frame 0 to frame 90:
- zoom by a factor of zoom
- rotate by 180 degrees vertically


script
function zoom(t) {
	return 0.6 - 0.3 * abs (sin (2 * PI * t / 180));
}
```



2.4.4 A rotating scroll-through

After the rotation, we want to scroll through the cornea while it is rotating around it's z-axis. Scrolling through is implemented by adding the following lines:

```
From frame 90 to frame 450:
- change bounding box max z to 0
```

Note that we decrease the bounding box max z (and not increase min z) because we turned the cornea by 180 degrees.

For the rotation, we could just add another line right after:

```
- rotate by 360 degrees around (0, 0, 1)
```

However, this would apply the rotation after the 10-degree orientation adjustments in the beginning, i.e. rotate around the view's and not the cornea's z-axis. This looks a bit strange in combination with the scroll-through. Instead, the rotation is added at the beginning of the script:

```
From frame 90 to frame 450:
- rotate by 360 degrees around (0, 0, 1)

At frame 0:
- rotate by 10 degrees vertically
- rotate by 10 degrees horizontally
- change front clipping to -120
- change bounding box max z to 500

From frame 0 to frame 90:
- rotate by 180 degrees vertically
- zoom by a factor of zoom


From frame 90 to frame 450:
- change bounding box max z to 0

script
function zoom(t) {
      return 0.6 - 0.3 * abs (sin (2 * PI * t / 180));
}
```

This is the entire script for the cornea animation.

3 WRITING EXTENSIONS

The parsing and animation modules are designed such that it is straightforward to use them in combination with other 3D rendering engines. This involves the following steps:

1. Extending the `RenderingState` class.
2. Implementing the `IRenderer3D` interface to render a frame given a `RenderingState` object.
3. Defining one's own keywords for the animation text.
4. Implementing a keyword factory class.
5. Writing a class, e.g. an ImageJ plugin, that serves as the extension's entry point.

These steps are detailed in the following paragraphs.

3.1 EXTENDING THE **RENDERINGSTATE** CLASS

An extended `RenderingState` object should contain all information to render a frame. The base class already stores information about the spatial 3D transformation of an object. Additionally, it keeps references to 2 (in the base class uninitialized) arrays: `channelproperties` and `nonchannelproperties`. The extending class should initialize and fill these arrays with required parameters.

Assuming that a very simple 3rd party rendering engine is used which renders a 3D object based on a single parameter, say brightness. Typically, the extending class would first define a static final constant with the index into the `nonchannelproperties` array:

```
public static final int BRIGHTNESS = 0;
```

The constructor would initialize the `nonchannelproperties` array:

```
public ExtRenderingState(int frame, CombinedTransform fwdTransform) {
              super(frame, fwdTransform);
              nonChannelProperties = new double[1];
              channelProperties = null;
}

public ExtRenderingState(
              int frame,
              double brightness,
              CombinedTransform fwdTransform) {
       this(frame, fwdTransform);
       nonChannelProperties[BRIGHTNESS] = brightness;
}
```

Finally, the `clone()` method needs to be overridden:

```
@Override
public ExtRenderingState clone() {
       ExtRenderingState kf = new ExtRenderingState(0, null);
       kf.setFrom(this);
       return kf;
}
```

3.2 IMPLEMENTING THE IRENDERER3D INTERFACE

The `IRenderer3D` interface defines the following methods:

```
public RenderingState getRenderingState();

public ImageProcessor render(RenderingState kf);

public ImagePlus getImage();
```

An implementing class typically keeps references to the input image stack, to the 3rd party rendering engine and to a `RenderingState` object:

```
private ExtRenderingState rs;

private ImagePlus input;

// private final ThridPartyRenderer renderer;
```

The constructor initializes the 3rd party renderer and the `RenderingState` object:

```
public ExtRenderer(ImagePlus input) {
	this.input = input;
	// renderer = new ThirdPartyRenderer();

	float[] pdIn = new float[] {
		(float)input.getCalibration().pixelWidth,
		(float)input.getCalibration().pixelHeight,
		(float)input.getCalibration().pixelDepth
	};

	float[] pdOut = new float[] {pdIn[0], pdIn[0], pdIn[0]};

	float[] rotcenter = new float[] {
			input.getWidth()   * pdIn[0] / 2,
			input.getHeight()  * pdIn[1] / 2,
			input.getNSlices() * pdIn[2] / 2};

	CombinedTransform transformation = new CombinedTransform(
						pdIn, pdOut, rotcenter);

	this.rs = new ExtRenderingState(0, transformation);
}
```

The most important method is `render()`: It uses the 3rd party rendering engine to produce an output image:

```
public ImageProcessor render(RenderingState rs) {
	double brightness = rs.getNonchannelProperty(
				ExtRenderingState.BRIGHTNESS);
	// renderer.setBrightness(brightness);

	float[] matrix = rs.getFwdTransform().
				calculateInverseTransform();
	// renderer.setTransformation(matrix);

	ImageProcessor output = null;
	// output = renderer.render();

	return output;
}
```

3.3 DEFINING KEYWORDS FOR THE ANIMATION TEXT

This is typically done by creating a Java enum that implements the `Keyword` interface. The Keyword interface defines the following methods:

```
public interface Keyword {

	/**
	 * The actual keyword text.
	 */
	public String getKeyword();

	/**
	 * A list of strings that describe each expected value
	 */
	public String[] getAutocompletionDescriptions();

	/**
	 * Returns an array with the indices of the rendering
	 * state properties (i.e. the indices in the
	 * nonchannelproperties and channelproperties
	 * arrays defined in RenderingState).
	 */
	public int[] getRenderingStateProperties();

	/**
	 * Returns the length of the keyword.
	 */
	public int length();

	/**
	 * Returns a map that assigns pre-defined strings to values.
	 */
	public Map<String, double[]> getReplacementMap();
}
```

This may seem overly complicated at first glance, but most of these functions may return null, but allow to greatly improve functionality. We'll go through it step by step.

3.3.1 Basic example

The most basic version for the example above would look like this:

```
public enum ExtKeyword implements Keyword {

	BRIGHTNESS("brightness", ExtRenderingState.BRIGHTNESS);

	private final String keyword;
	private final int[] rsProperties;

	private ExtKeyword(String text, int... rsProperties) {
		this.keyword = text;
		this.rsProperties = rsProperties;
	}

	@Override
	public int[] getRenderingStateProperties() {
		return rsProperties;
	}
```

```
	@Override
	public String[] getAutocompletionDescriptions() {
		return null;
	}

	@Override
	public String getKeyword() {
		return keyword;
	}

	@Override
	public int length() {
		return keyword.length();
	}

	@Override
	public Map<String, double[]> getReplacementMap() {
		return null;
	}
}
```

So the important and required methods are

- getKeyword(): Returns the keyword as a text
- length(): Returns the length of the keyword
- getRenderingStateProperties(): Returns the indices of the properties in the `RenderingState` class which are modified by this keyword

In the example above, only one property is modified. This is the simplest case, but it is also possible to change multiple properties at once. We'll come back to that later.

When typing in the Animation Editor, the sentence is automatically completed step by step:

```
From frame 0 to frame 180 change brightness to
```

3.3.2 Automatically adding a placeholder for the expected value

To further auto-complete (and add a descriptive placeholder for the value that should be entered), the `getAutocompletionDescriptions()` must return a meaningful value:

```
public enum ExtKeyword implements Keyword {

	BRIGHTNESS("brightness",
			new String[] {"<0..1>"},
			ExtRenderingState.BRIGHTNESS);

	...
	private final String[] autocompletionDesc;

	private ExtKeyword(String text,
				String[] autocompletionDesc,
				int... rsProperties) {

		this.keyword = text;
		this.rsProperties = rsProperties;
		this.autocompletionDesc = autocompletionDesc;
	}
```

```
@Override
	public String[] getAutocompletionDescriptions() {
		return autocompletionDesc;
	}

	...
}
```

Code changes are indicated using **bold** font.

When typing in the Animation Editor now, the sentence is automatically completed and a description is added for the value of brightness (in our case “<0..1>”):

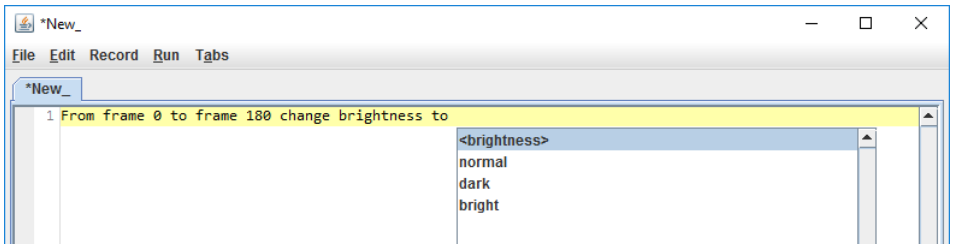
```
From frame 0 to frame 180 change brightness to <0..1>
```

3.3.3 Pre-defined values

Sometimes, a rendering property allows only discrete values, or they are just more convenient. In the example here, we can introduce 3 pre-defined values for brightness: dark (with a value of 0.3), normal (with a value of 0.5) and bright (with a value of 1). To do so, `getReplacementMap()` must return a meaningful value:

```
public enum ExtKeyword implements Keyword {

	BRIGHTNESS("brightness",
			new String[] {"<brightness>"},
			makeReplacementMap(),
			ExtRenderingState.BRIGHTNESS);

	private final Map<String, double[]> replacementMap;

	private ExtKeyword(String text,
			String[] autocompletionDesc,
			int... rsProperties) {
		this(text, autocompletionDesc, null, rsProperties);
	}

	private ExtKeyword(String text,
			String[] autocompletionDesc,
			Map<String, double[]> replacementMap,
			int... rsProperties) {
		this.keyword = text;
		this.rsProperties = rsProperties;
		this.autocompletionDesc = autocompletionDesc;
		this.replacementMap = replacementMap;
	}

	@Override
	public Map<String, double[]> getReplacementMap() {
		return replacementMap;
	}

	private static Map<String, double[]> makeReplacementMap() {
		HashMap<String, double[]> map =
				new HashMap<String, double[]>();
		map.put("dark",   new double[] {0.3});
		map.put("normal", new double[] {0.5});
		map.put("bright", new double[] {1.0});
		return map;
	}
}
```

This is what auto-completion provides now:



3.3.4 Keywords that change multiple rendering properties

Assume we have another rendering property, an object color. Starting from the beginning, we would add these lines to the class that extends `RenderingState` (I'm omitting other code adjustments in this class which are straightforward):

```
public static final int COLOR_RED   = 1;
public static final int COLOR_GREEN = 2;
public static final int COLOR_BLUE  = 3;
```

Of course we also need to adjust the class implementing the IRenderer3D interface, which I will also skip here.

We want now to introduce a keyword 'color' which changes the red, green and blue component simultaneously. To do so, we need to specify multiple `RenderingState` properties in the keyword constructor

```
...
COLOR("color", null, null,
             ExtRenderingState.COLOR_RED,
             ExtRenderingState.COLOR_GREEN,
             ExtRenderingState.COLOR_BLUE),
...
```

which allows us to write in the animation text

```
From frame 0 to frame 180 change color to (255, 0, 0).
```

Of course it also makes sense here to add value descriptions and pre-defined colors:

```
COLOR("color",
       new String[] {"<r>", "<g>", "<b>"},
       makeColorReplacementMap(),
       ExtRenderingState.COLOR_RED,
       ExtRenderingState.COLOR_GREEN,
       ExtRenderingState.COLOR_BLUE);

private static Map<String, double[]> makeColorReplacementMap() {
       HashMap<String, double[]> map =
                            new HashMap<String, double[]>();
       map.put("red",     new double[] {255.0, 0.0, 0.0});
       map.put("green",   new double[] {0.0, 255.0, 0.0});
       map.put("blue",    new double[] {0.0, 0.0, 255.0});
       map.put("yellow",  new double[] {255.0, 255.0, 0.0});
       map.put("cyan",    new double[] {0.0, 255.0, 255.0});
       map.put("magenta", new double[] {255.0, 0.0, 255.0});
       return map;
}
```

3.4 Implementing a keyword factory class.

Implementing a keyword factory class is simple and straightforward:

```
public class ExtKeywordFactory implements IKeywordFactory {

	private static ExtKeywordFactory instance = null;

	private ExtKeywordFactory() {}

	public static ExtKeywordFactory getInstance() {
		if(instance == null)
			instance = new ExtKeywordFactory();
		return instance;
	}

	@Override
	public Keyword[] getNonChannelKeywords() {
		return ExtKeyword.values();
	}

	@Override
	public Keyword[] getChannelKeywords() {
		return null;
	}
}
```

3.5 Writing a class that serves as the extension's entry point.

This might for example be an ImageJ plugin. This is also simple and straightforward:

```
public class Main implements PlugInFilter {

	private ImagePlus image;

	@Override
	public int setup(String arg, ImagePlus imp) {
		this.image = imp;
		return DOES_ALL;
	}

	@Override
	public void run(ImageProcessor ip) {
		run(image);
	}

	public static void run(ImagePlus image) {
		ExtRenderer renderer = new ExtRenderer(image);
		AnimationEditor editor = new AnimationEditor(
					renderer, null);
		editor.setVisible(true);
	}
}
```

That's all.

The entire source code for the example mentioned above can be downloaded from GitHub (https://github.com/bene51/3Dscript).