First Steps in FlashDot

Tobias Elze

This is one of the documents commonly labelled "quick and dirty". It is for impatient people who don't like long handbooks. Read it to get started, but do not take it as a substitute for the reference.

Last modified: September 22, 2009

Contents

1 What is FlashDot? 2

2 Making Things Visible: FlashDot by Example 3

2.1 Stimuli . . . 3
2.2 Images . . . 4
2.3 Scenes . . . 5
2.4 Experiments . . . 7

3 Graphical user interface via XML 10

1 What is FlashDot?

People concerned with psychophysics and vision science require an extraodinary precision and controllability of visual stimuli on computer monitors. A single lost frame or a slight deviation of what you expect to be displayed and what is really displayed can ruin your data.

FlashDot is an "experiment generator" that is optimized for the application in vision research. The program has been in use in several Max-Planck-Institutes in Germany, particularly in Leipzig and Tübingen. We have used it for areas like visual masking, face recognition, and monitor metrology.

This document assumes you have successfully installed FlashDot on your computer. You can find the latest versions of the FlashDot binaries as well as the sources here: `http://www.flashdot.info`.

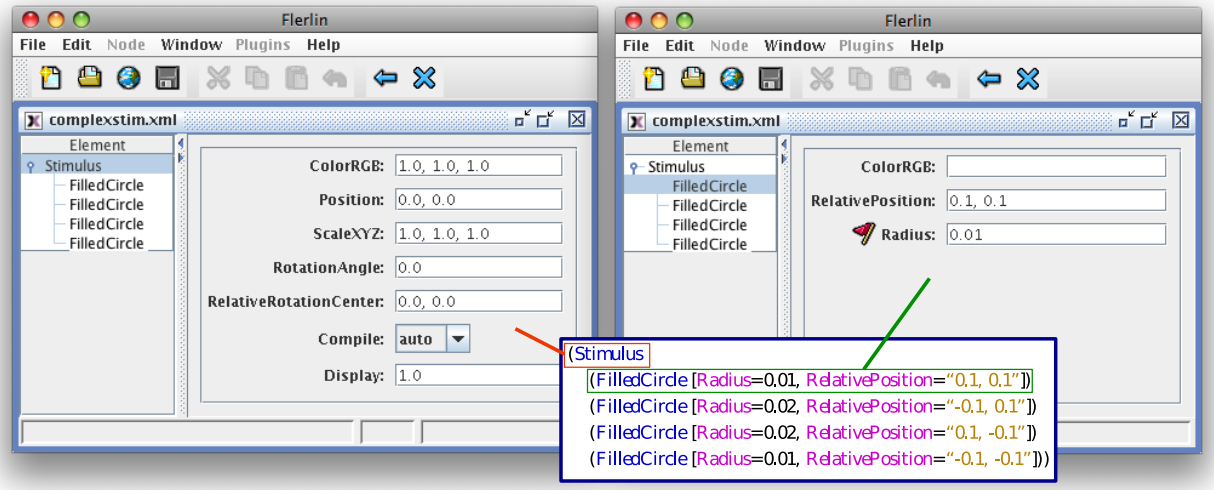


Figure 1: A FlashDot stimulus in S–Expression syntax (source code inside the blue frame) and opened in the XML Editor Flerlin. The left Flerlin window highlights the element Stimulus, the right window the element FilledCircle.

The core of FlashDot is a command line program that executes experiments written in an *experiment generation language*. Currently, there are two syntaxes for this language: First, XML (file name extension: fld), second: S expressions (file name extension: fls).

XML is intended to be used by graphical editors (see Section 3), S expressions, on the other hand, for simple text editors or at the command line. Fig. 1 shows a Flashdot stimulus in both syntaxes and depicts the two ways (text editing and graphical user interface) how to operate FlashDot.

Both syntaxes are equivalent and can be easily transformed into each other, and FlashDot may support more command syntax formats than these two in the future as its "kernel" is kept very flexible.

The following sections start with the S expression syntax. S expressions are parenthesized expressions – they look a little like LISP source code (note that you can use a graphical user interface as well, but S–Expressions are easier to present in a textual introduction like this one).

Their formal structure is:
(Function [attribute1=..., attribute2=..., ...] argument1 argument2 ...).

Many but not all functions have attributes and arguments. An example of a function without attributes is (Sin PI), an example of a function without arguments but with attributes is (Rectangle [Width=0.5, Height=0.7]).

Attributes are in most cases optional and have default values. In contrast, arguments are mandatory and cannot be skipped. I'll use consistent coloring for functions and attributes throughout the text.

By the way, in addition to calling FlashDot with an experiment language file, there is also an interactive top level (regard it as a "FlashDot shell") in which you can enter S expressions that will be immediately executed. But this shell is not covered by this document.

Let's stop the theory now and jump in at the deep end...

2 Making Things Visible: FlashDot by Example

For displaying things, FlashDot is hierarchically organized. This section introduces this hierarchy from bottom to top.

2.1 Stimuli

The most basic level consists of plotting commands for drawing points, lines, circles, polygons etc., for instance: (Circle [Radius=...]).

These elementary plotting commands are grouped into *stimuli*. A Stimulus consists of one ore more elementary plotting commands. Listing 1 shows a very stimple stimulus.

```
(Stimulus (Circle [Radius=0.1]))
```

Listing 1: Simple Stimulus

How to run this example? Open a text editor and copy and paste the line above, then save the file to a name like "circle.fls". Then execute the file with FlashDot (Windows: double click, other systems: flashdot circle.fls).

If everything is installed properly, a window of default size (800 × 600 pixels) will pop up, showing a centered circle. The default internal coordinate system ranges from −1 to 1 horizontally and from −0.75 to 0.75 vertically, that is, the circle covers one tenth of the window width. You can exit the window with ESCAPE.

Let's make a more complex stimulus. Listing 2 sets relative positions of its components.

The position of the components are called "relative" because their arrangement is not affected by moving the whole stimulus, which is done by setting the Position attribute, as shown in Listing 3.

Fig. 2 shows this effect: The complex stimulus, consisting of four filled circles, is shifted from the central position (left) into the upper right direction (right).

```
(Stimulus
   (FilledCircle [Radius=0.01, RelativePosition="0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="-0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="0.1, -0.1"])
   (FilledCircle [Radius=0.01, RelativePosition="-0.1, -0.1"]))
```

Listing 2: Relative positions of the elements within the stimulus.

```
(Stimulus [Position="0.2, 0.2"]
   (FilledCircle [Radius=0.01, RelativePosition="0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="-0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="0.1, -0.1"])
   (FilledCircle [Radius=0.01, RelativePosition="-0.1, -0.1"]))
```

Listing 3: The Position attribute moves the whole stimulus but keeps the relative positions.

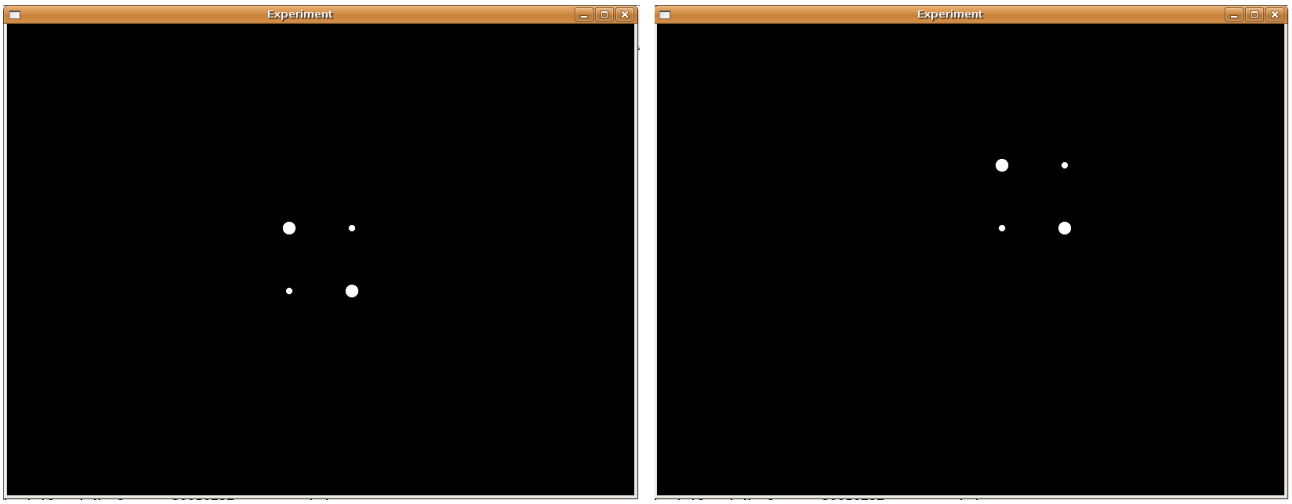


Figure 2: Listing 2 (left) and Listing 3 (right).

2.2 Images

```
(Image [CenterHorizontally="true", CenterVertically="true"]
   (FromImageFile [Filename="flashdot.png"]))
```

Listing 4: An Image loaded from the file flashdot.png.

Images are on the same hierarchical level as stimuli and have many attributes in common with the latter. In contrast to a stimulus, an Image is generated from a *matrix of pixels*. It can either be loaded from an image file, as in Listing 4, or generated within flashdot applying the various built in matrix functions.

Listing 5 shows how an image can be generated from a matrix. The function Gray translates a random matrix of numbers between 0 and 1 to a structure that FlashDot can internally use

```
(Image [LowerLeftCorner="-0.75,-0.3"]
   (Gray (RandomMatrix
         (Continuous (Uniform))
         (NumberOfRows 200)
         (NumberOfColumns 200))))
```

Listing 5: An Image generated from a random matrix.

as an image. Note that the size of the matrix is in *pixels* (in the current case 200×200) and therefore independent of the internal coordinate system. Each pixel of the image will cover exactly one pixel on the monitor. Fig. 3 shows both listings.

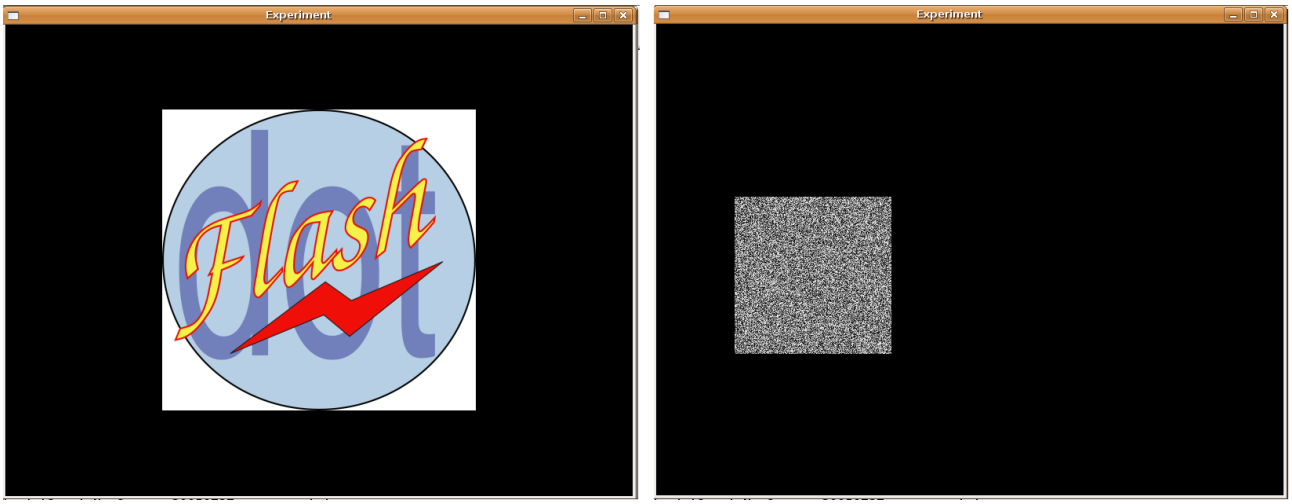


Figure 3: Listing 4 (left) and Listing 5 (right).

2.3 Scenes

The hierarchical level above stimuli and images is the Scene. Scenes allow the declaration of local variables, functions etc. In the context of psychophysical experiments, a scene contains a special arrangement which is to be displayed together for a certain time.

Similarly to the "RelativePosition" concept for spatial positions within stimuli, the scene components can also have temporal characteristics "relative" to the global temporal setting of the scene.

Every scene needs a unique ID, as we will see later on. Listing 6 contains an example of a very simple scene and shows how to delcare a local constant.

Up to now, everything is still static. How to make things change/move? One can either display the whole scene for a certain number of frames, as we will see later, or declare a so–called *VariationInTime* locally within a scene. Run Listing 7 to understand this concept.

As you can see, the square is repeatedly moving from center to right. How does this work? A VariationInTime modifies frame by frame the value of a variable named by the attribute Identifier.

```
(Scene [SceneID="circleSquare"]
  (LocalDeclarations
    (:= squarelength (Constant 0.1)))
  (Stimulus (Circle [Radius=0.05]))
  (Stimulus [Position="-0.2, -0.2"]
    (Rectangle [Width="squarelength", Height="squarelength"])))
```

Listing 6: Simple Scene with a local constant.

```
(Scene [SceneID="circleSquare"]
  (LocalDeclarations
    (:= squarelength (Constant 0.1))
    (VariationInTime [Identifier="xpos"]))
  (Stimulus (Circle [Radius=0.05]))
  (Stimulus [Position="#xpos, -0.2"]
    (Rectangle [Width="squarelength", Height="squarelength"])))
```

Listing 7: The VariationInTime xpos varies from frame to frame.

Its default starting value is 0 (that's why the square starts from the center) and its default final value is 1. Frame by frame, our variable xpos is increased until it reaches 1.

The number of frames it takes from 0 to 1 is by default inherited from the number of frames the scene is shown. We didn't declare any scene duration, so our scene is displayed infinitely long until ESCAPE is pressed. In case of infinite scene durations, the number of frames for each VariationInTime defaults to 100.

The "#" in front of the variable name in the Position attribute of the square stimulus indicates that the value may change within the scene. In this case, this won't slow down calculation of the respective next frame as stimuli are internally compiled as OpenGL display lists, and characteristics like position or color can be set for display lists without speed decreases.

Let's make the motion slower by increasing the number of frames, shown in Listing 8.

```
(Scene [SceneID="circleSquare"]
  (LocalDeclarations
    (:= squarelength (Constant 0.1))
    (VariationInTime [Identifier="xpos", NumberOfFrames=500]))
  (Stimulus (Circle [Radius=0.05]))
  (Stimulus [Position="#xpos, -0.2"]
    (Rectangle [Width="squarelength", Height="squarelength"])))
```

Listing 8: By setting NumberOfFrames to 500, the motion slows down.

Variations in time are not restricted to spatial positions but can be applied to any numerical

value related to the display. In Listing 9, we change the color of the circle in addition to the position of the square.

```
(Scene [SceneID="circleSquare"]
  (LocalDeclarations
    (:= squarelength (Constant 0.1))
    (VariationInTime [Identifier="xpos", NumberOfFrames=500]))
  (Stimulus (Circle [Radius=0.05, ColorRGB="#xpos, 0, 0"]))
  (Stimulus [Position="#xpos, -0.2"]
    (Rectangle [Width="squarelength", Height="squarelength"])))
```

Listing 9: VariationInTime applied to color.

Variations in time are both compact and powerful representations of changing entities. One can do much more with them, for example add a function to them that controls the modification. Imagine you want a sinusoidal motion of the square, that is, while it moves from left to right, its vertical motion can be described by the full period of the sine function.

No, we would like to add the following function to a new variation in time `ypos`: $f(x) = 0.5 \cdot \sin(2\pi x)$. In S expressions, mathematical functions look a little strange at the first glance, but one can get used to it. Listing 10 shows the resulting code.

```
(Scene [SceneID="circleSquare"]
  (LocalDeclarations
    (:= squarelength (Constant 0.1))
    (VariationInTime [Identifier="xpos", NumberOfFrames=500])
    (VariationInTime [Identifier="ypos", NumberOfFrames=500]
      (Function [x] (* 0.5 (Sin (* 2 PI x))))))
  (Stimulus (Circle [Radius=0.05, ColorRGB="#xpos, 0, 0"]))
  (Stimulus [Position="#xpos, #ypos"]
    (Rectangle [Width="squarelength", Height="squarelength"])))
```

Listing 10: VariationInTime ypos with the attached function $0.5 \cdot \sin(2\pi x)$

Try it out. Nice, isn't it? By the way, an easy way to switch a whole stimulus on and off is to set its attribute Display to 0 (not displayed) or a value different from 0 (displayed). This way, a variation in time can easily generate a blinking stimulus, for instance.

2.4 Experiments

The highest hierarchical level is an Experiment. This does not only group several scenes but also controls their durations, their order of invocation, allows global declarations for variables, functions, display objects like stimuli, etc. In addition, it allows such important settings as full

screen, frame rate (as far as supported by the display), horizontal and vertical number of pixels of the display and so on.

Listing 11 shows how the embedding of our scene into a simple experiment might look like.

```
(Experiment [CheckRefreshRate="false"]
  (SimpleDisplayMode
    [FullScreen="false"])
  (Scene [SceneID="instructions"]
    (Stimulus [Position="-0.5, 0"]
      (TextLine
        [Text="Give some instructions here... Press ENTER to proceed."]))
    (Event
      (Key (Special [SpecialKey="Enter"]))
      (Action (JumpToNextPresentation))))
  (Scene [SceneID="circleSquare"]
    (LocalDeclarations
      (:= squarelength (Constant 0.1))
      (VariationInTime [Identifier="xpos", NumberOfFrames=500])
      (VariationInTime [Identifier="ypos", NumberOfFrames=500]
        (Function [x] (* 0.5 (Sin (* 2 PI x))))))
    (Stimulus (Circle [Radius=0.05, ColorRGB="#xpos, 0, 0"]))
    (Stimulus [Position="#xpos, #ypos"]
      (Rectangle [Width="squarelength", Height="squarelength"])))
  (Sequence
    (Presentation [SceneID="instructions", Duration="-1"])
    (Presentation [SceneID="circleSquare", Duration="1000"])))
```

Listing 11: Simple Experiment.

As you can see, order of scene presentation and scene durations are set by a so–called Sequence. Each scene is referred to by its identifier. The Duration attribute takes by default integer values representing the number of frames the scene is displayed. In the example above, our scene "circleSquare" is displayed for 1000 frames. Since no further scene follows, the program terminates after this.

A negative duration means that the scene is displayed eternally. This makes only sense, of course, if you are waiting for user input.

Our scene "instructions" is displayed forever until ENTER is pressed. The ENTER event terminates this scene and jumps to the next scene in the Sequence.

Finally, some important remarks about the tags and options related to the display settings (SimpleDisplayMode etc.). As you can see, we set the full screen option to "false". For a measurement condition, that's usually not what you want.

Before allowing full screen presentation, always specify a sensible screen resolution. LCD monitors, for instance, have a *native* resolution, and resolution settings different from this result in interpolated and therefore blurred stimuli.

The horizontal and vertical resolutions can be set as attributes CanvasWidth (default: 800) and CanvasHeight (default: 600) to the Experiment tag. Alternatively, you can set the attribute GuessResolution to "true" which will ignore the former settings and guess the resolution from operating system's settings. It is implemented for the operating systems Linux, MacOS X, and Windows, but might fail for other operating systems.

In addition, you may want to set the internal coordinate system (default: $(-1, 1)$ horizontally and $(-0.75, 0.75)$ vertically) to values related to your native resolution. However, by doing so, you need to care for all your position and length settings of your stimuli. The internal coordinate system can be set as further attributes to Experiment, e. g.: HorizontalRange="1, 1024", VerticalRange="1, 768".

What's the strange attribute CheckRefreshRate that is by default set to "true"? If true, FlashDot will guess the refresh rate over the first 100 frames (which are then displayed black), check if it corresponds to your desired refresh rate, if you specified one, and, more important, *log the time between every two synchronizations of your graphics adapter with the vertical blank of your monitor.*

That means, FlashDot indirectly checks the duration of every frame. Any strong deviations of a frame duration from the guessed refresh rate are logged then throughout the whole experiment. This way, you can check after the experiment if you lost any frames during the presentations.

Frame duration deviations as well as guessed refresh rate etc. are written by default to the file logfile.xml.

What about SimpleDisplayMode? If you have native OpenGL hardware support, you can either run OpenGL with your usual display settings (that's what SimpleDisplayMode does, by default full screen), or, as recommended, in the so–called OpenGL "Game Mode". This mode, as indicated by the name intended for computer games with high hardware demands, is always full screen and allows, at least theoretically, color depth and refresh rate settings.

Whereas most CRT monitors allow arbitary refresh rate settings over a large interval, most LCD monitors allow only one or only a very few refresh rates, usually only 60 Hz. The OpenGL Game Mode is invoked by EnhancedDisplayMode instead of SimpleDisplayMode. Try out the settings in Listing 12.

If you have native OpenGL support of your graphics hardware, this should work nicely. Check logfile.xml after running the script above to find out if the values stated there make sense. If they don't make sense, or if your display looks quite strange (wrong colors etc.), there is probably something wrong with your OpenGL settings.

The distinction between normal and game mode (SimpleDisplayMode vs. EnhancedDisplayMode) might even make a difference for the mode your *monitor* switches to. It might be possible that, if the monitor receives "game mode input", it will switch to a special mode adapted to computer gamers (but I'm not sure). It's worth to try this out.

One more important remark: Of course, synchronization of your graphics adapter to the

```
(Experiment
  [GuessResolution="true"]
  (EnhancedDisplayMode [RefreshRate=60])
  (Scene [SceneID="instructions"]
    (Stimulus [Position="-0.5, 0"]
      (TextLine
        [Text="Give some instructions here... Press ENTER to proceed."]))
    (Event
      (Key (Special [SpecialKey="Enter"]))
      (Action (JumpToNextPresentation))))
  (Scene [SceneID="circleSquare"]
    (LocalDeclarations
      (:= squarelength (Constant 0.1))
      (VariationInTime [Identifier="xpos", NumberOfFrames=500])
      (VariationInTime [Identifier="ypos", NumberOfFrames=500]
        (Function [x] (* 0.5 (Sin (* 2 PI x))))))
    (Stimulus (Circle [Radius=0.05, ColorRGB="#xpos, 0, 0"]))
    (Stimulus [Position="#xpos, #ypos"]
      (Rectangle [Width="squarelength", Height="squarelength"])))
  (Sequence
    (Presentation [SceneID="instructions", Duration="-1"])
    (Presentation [SceneID="circleSquare", Duration="1000"])))
```

Listing 12: Experiment with enhanced display settings.

vertical blank signal of the monitor needs to be switched ON! Most operating systems allow to switch this off, and unfortunately, on some systems it is switched off by default.

If logfile.xml indicates that this synchronization does not work (usually a very high guessed refresh rate, as the graphics card sends the signal as fast as it can calculate the next frame, no matter what the monitor does at that time), find out where to switch this synchronization on or off for your operating system and check it.

For the operating systems Linux and MacOS X, FlashDot makes these settings automatically. Windows may hide this setting somewhere in your graphics card configurations. Open your graphics card driver settings and search for something like "OpenGL settings" and then "sync to vertical blank" (or similar).

3 Graphical user interface via XML

Alright, now you know how FlashDot works ;) Well, you may argue now: Even if I remember all these commands, how can I know the thousands of attributes and their default values? Especially since there is no complete documentation of FlashDot yet? (I'm working hard on

this, though.)

That's right, but, as mentioned above, we are not restricted to the S expression syntax and simple text editors. Instead, FlashDot allows a second syntax which consists of *XML*.

Don't worry if you have never heard of XML yet. The only thing you need to know about XML is that there are numerous *graphical editors* which check the syntax while you are generating your file, show you all possible following commands for each command that you are currently working with, and, for sure, all possible attributes together with their default values.

That is, such an XML editor implicitely guides you to generate your script and can be operated almost without any prior knowledge.

Let us first convert the fls file above to XML. If you saved the file as circle.fls, you can do this by

```
flashdot -fls2fld circle.fls
```

which generates a file circle.fld. In the following, I describe how to work with the Java open source XML editor Xerlin (or better, if available, my derived version "Flerlin" which is shipped with some of the precompiled FlashDot binary packages).

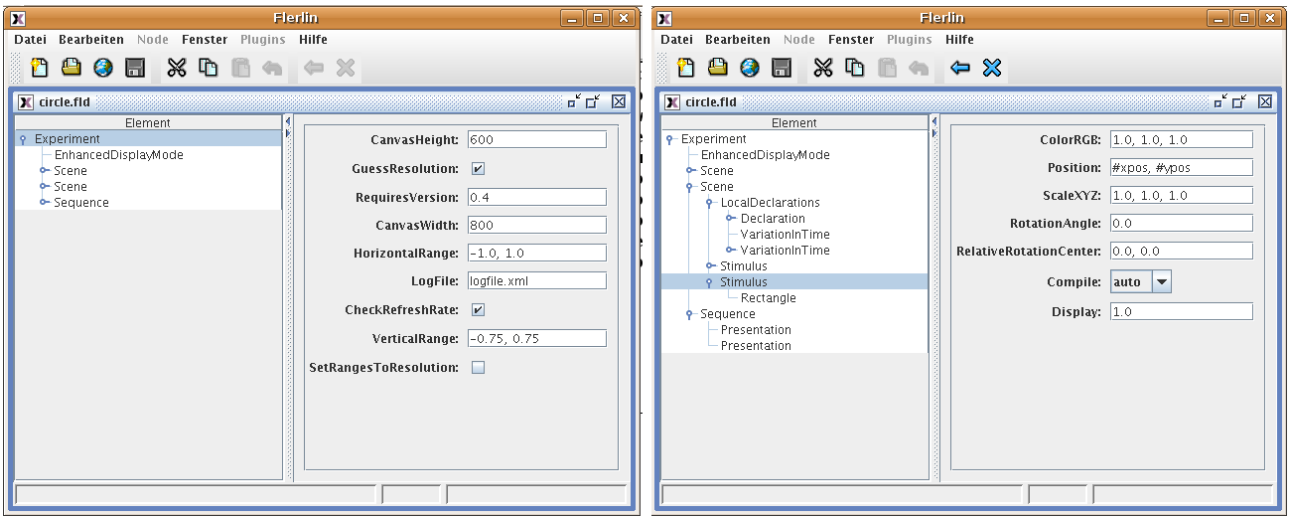


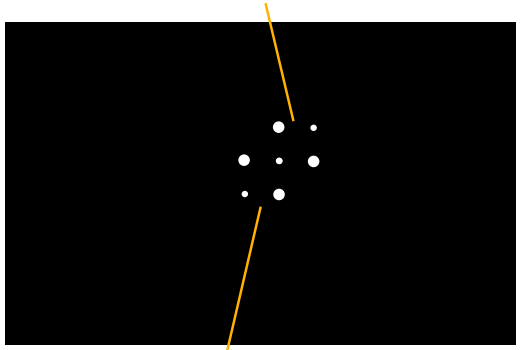
Figure 4: XML editor Flerlin, editing the root node Experiment (left) and a Stimulus (right). The Flerlin window is split into two parts, where the left part shows the nodes and the right part the corresponding attributes with their values. Flerlin is opened on Ubuntu in a German locale here.

Look for the file "experiments.dtd" in your flashdot directory and copy it to the same location as circle.fld. Then open Xerlin, and from there, open circle.fld. You can see the whole experiment as a tree there (see Fig. 4). If you select any node, on the right hand side you can see all its attributes with their default values.

If you right click any of the nodes, you will recognize how considerably Xerlin lightens your work load: It lets you only add exactly these new nodes that are syntactically valid at this position. If you add a new node to the experiment, Xerlin's coloring indicates if the experiment is syntactically complete (black color) or not yet complete and your new node needs further subnodes (light red color of the respective branch of the tree).

Stimulus attributes are applied to all stimulus contents

```
(Stimulus [Position="0.2, 0.2"]
   (FilledCircle [Radius=0.01, RelativePosition="0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="-0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="0.1, -0.1"])
   (FilledCircle [Radius=0.01, RelativePosition="-0.1, -0.1"]))
```



```
(Stimulus
   (FilledCircle [Radius=0.01, RelativePosition="0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="-0.1, 0.1"])
   (FilledCircle [Radius=0.02, RelativePosition="0.1, -0.1"])
   (FilledCircle [Radius=0.01, RelativePosition="-0.1, -0.1"]))
```

Stimuli group a number of drawing primitives with relative positions

Scene with VariationInTime

```
(Scene [SceneID="circleSquare"]
  (LocalDeclarations
    (:= squarelength (Constant 0.1))
    (VariationInTime [Identifier="xpos", NumberOfFrames=500])
    (VariationInTime [Identifier="ypos", NumberOfFrames=500]
      (Function [x] (* 0.5 (Sin (* 2 PI x))))))
  (Stimulus (Circle [Radius=0.05, ColorRGB="#xpos, 0, 0"]))
  (Stimulus [Position="#xpos, #ypos"]
    (Rectangle [Width="squarelength", Height="squarelength"])))
```

VariationInTime applicable to all kinds of variables

Exemplary experiment

Display settings

Scene with unique ID

Catching events

Keyboard input

User instructions

Action envoked by event (go to next scene)

Local declarations (only valid inside the scene)

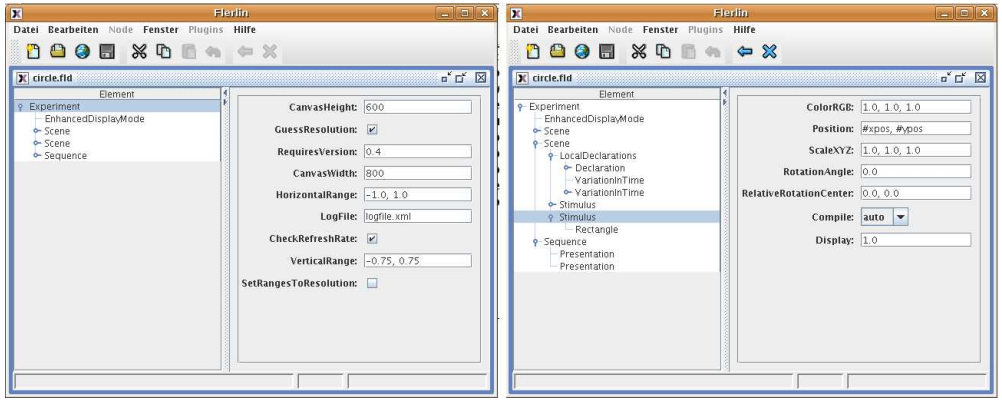
VariationInTime lasting 500 frames

Sequence of scenes together with their durations

"-1": infinite duration

```
(Experiment
  [GuessResolution="true"]
  (EnhancedDisplayMode [RefreshRate=60])
  (Scene [SceneID="instructions"]
    (Stimulus [Position="-0.5, 0"]
      (TextLine
        [Text="Give some instructions here.. Press ENTER to proceed."]))
    (Event
      (Key (Special [SpecialKey="Enter"]))
      (Action (JumpToNextPresentation))))
  (Scene [SceneID="circleSquare"]
    (LocalDeclarations
      (:= squarelength (Constant 0.1))
      (VariationInTime [Identifier="xpos", NumberOfFrames=500])
      (VariationInTime [Identifier="ypos", NumberOfFrames=500]
        (Function [x] (* 0.5 (Sin (* 2 PI x))))))
    (Stimulus (Circle [Radius=0.05, ColorRGB="#xpos, 0, 0"]))
    (Stimulus [Position="#xpos, #ypos"]
      (Rectangle [Width="squarelength", Height="squarelength"])))
  (Sequence
    (Presentation [SceneID="instructions", Duration="-1"])
    (Presentation [SceneID="circleSquare", Duration="1000"])))
```

Source code (S-Expr.)



GUI (Flerlin)