EventIDE Quick Guide

Introduction

EventIDE is a visual programming platform for designing and running psychological experiments. The platform is the most feature-rich in its class and combines a friendly visual designer with modern instrumentation for writing user code.

The first distinct feature of EventIDE is complete visual representation of a designed experiment and its flow in the GUI. Creating an experiment in EventIDE is somewhat similar to creating an interactive slide presentation, except that one has much more control on flow logic and a possibility to adjust of the designed content at runtime.

The second distinct feature of EventIDE is a very simple coding model. The model does not require writing a control flow, which decides when and what subroutines to call. Instead, the user-code is injected directly into the visually designed experiment. There are no functions, procedural calls and libraries related to the experiment design and correspondingly no necessity to learn them. In result, the user code never has to go beyond standard programming statements. You will also be pleasantly surprised to see your experiment programmed in a lot less lines of code than expected. Yet, you still have the power and tools of a professional programming language fully available.

Finally, the modular architecture of EventIDE allows easy integration of user extensions directly to the program core.

Using Quick Guide

This quick guide is subdivided into chapters describing different stages of an experiment design in EventIDE. The chapters are listed in the same sequential order that you would follow in a real design process. While reading the guide, you can run a second instance of EventIDE and try some described procedures in practice.

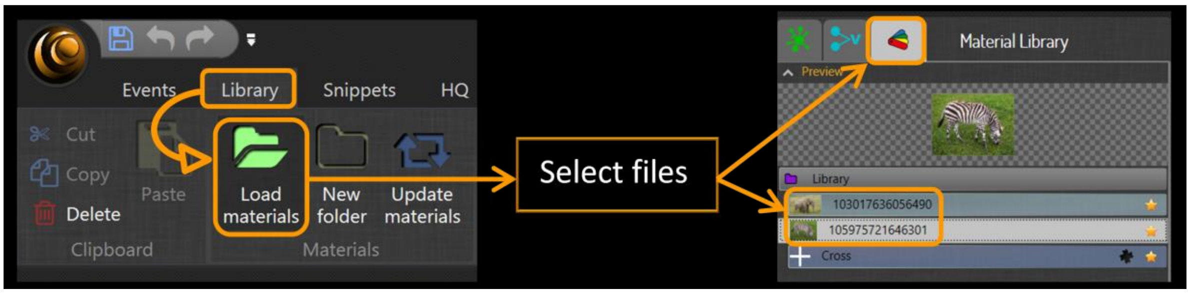
Other help options

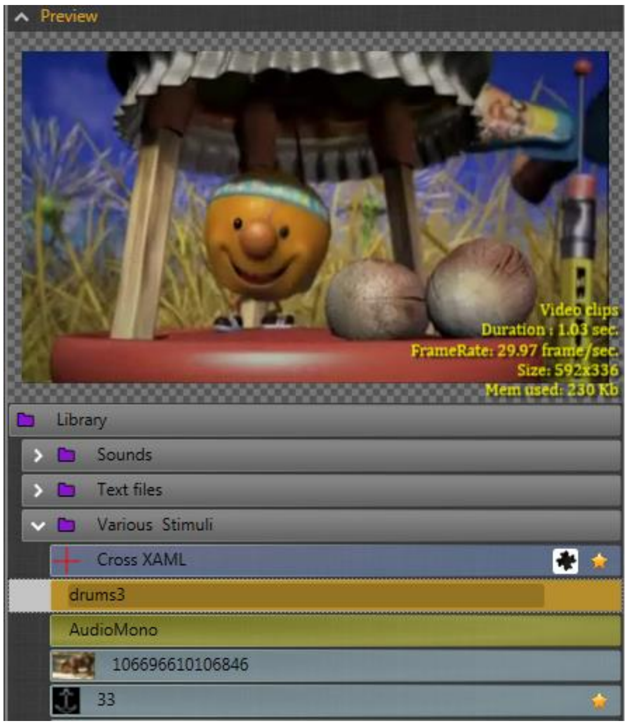
The quick guide describes more practical methods in EventIDE. For more general and detailed documentation please see the online EventIDE Wiki. If you are interested in implementation of the particular experiment paradigm you can try to find a close example either in the demo or template gallery (accessible via the application menu). For frequently asked question please check out FAQ blog. In addition, we will be always glad to answer questions on the EventIDE Forum, by email and Skype.

1. Adding Stimulus Materials

In EventIDE, you normally begin the design by either creating a blank experiment or by loading an experiment template. To create a new experiment, click on application button and select New -> Blank Experiment. Once a new experiment is created, you need to load all necessary stimulus materials to it. This section gives an overview on managing stimulus material in EventIDE.

Each EventIDE experiment has an embedded storage, Material Library, where you create or load material files. The Material Library has a hierarchical folder structure, similar to a file system on your computer. To access and manage the Material Library, use the Library ribbon tab and Material Library panel shown on the screenshots below:





External files and folders can be loaded or simply dragged into the Material Library panel. EventIDE automatically recognizes various file formats that contain images, vector graphics, sounds, videos and 3D models. Recognized files become material items those can be previewed and edited. Unrecognized files are treated as data items and can be used in custom scenarios. You can move, rename and resort items in the Library. You may also export them by dragging the files to File Explorer.

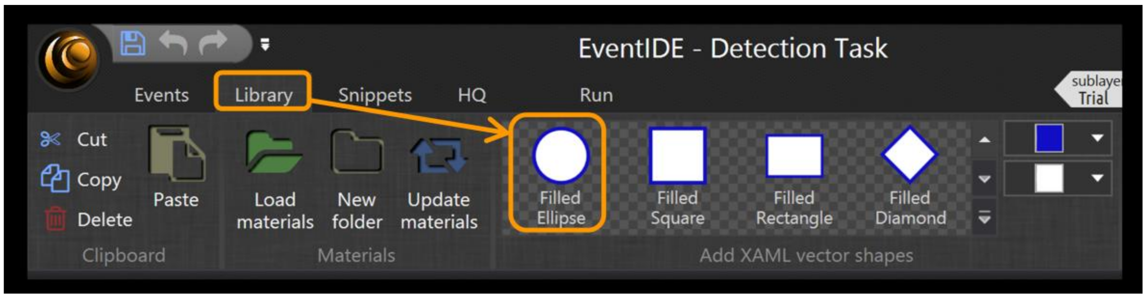
Note that all Library materials are saved inside of the main experiment file, so that an experiment does not rely on external files, after they are loaded into the Library.

XAML Materials

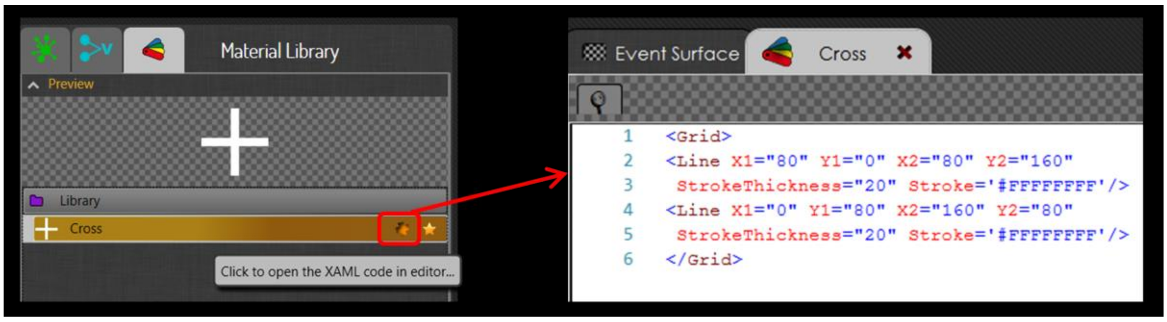
EventIDE also offers a powerful way to create scalable visual stimulus 'in-place', with a vector-graphics engine based on XAML scripting. XAML is a descriptive markup language for visual design, which has a very comprehensible and intuitive syntax. For example, you can create a rectangular vector shape with a single line of code:

```
<Rectangle Width="320" Height="240" Fill=”DarkOrange” StrokeThickness="15" Stroke=”#FFDDFFDD”/>
```

XAML also allows creating compositions of multiple vector shapes and employing different brushes, gradients and visual effects. To learn more about the XAML features, examine the template gallery in the Library ribbon tab:



Once a XAML item is created in Library, the underlying script can be modified in the XAML code editor:



```
<Grid>
<Line X1="80" Y1="0" X2="80" Y2="160"
 StrokeThickness="20" Stroke='#FFFFFFFF'/>
<Line X1="0" Y1="80" X2="160" Y2="80"
 StrokeThickness="20" Stroke='#FFFFFFFF'/>
</Grid>
```

When a vector graphics file in the svg format is loaded into Library, EventIDE automatically converts it into XAML.

Material Reusability

Any material item in Library can be used multiple times, in different parts of an experiment. The items can be updated with a new content without breaking the design. For example, to replace an image with a new one, select the image in the Material Library panel, then click the right mouse button, and select "Update material items" in the context menu.

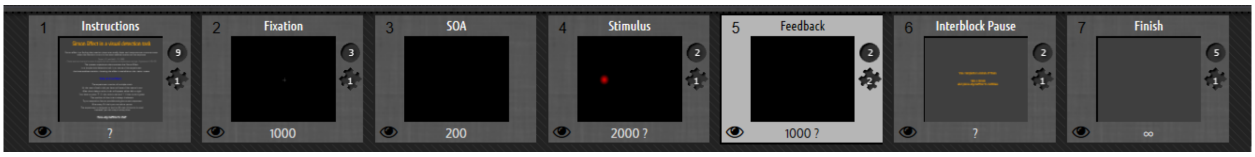
Applying Materials

To use the Library material items in an experiment design, you need an appropriate EventIDE element to handle a particular material type. For example, you can choose the Renderer element to render visual materials on the screen and the Wave player element for playing the wav audio clips. To learn more about the EventIDE elements, please continue with the next sections of the quick guide.

2. Creating Scenario and Trial Structure

Events and Layers

Creating an experiment in EventIDE is somewhat similar to creating a slide presentation, except that one has much more control over the slide flow. An experiment is constructed with 'events' which are similar to presentation slides. Each event is a time period corresponding to a certain logical stage in an experiment (the period is bounded by event's onset and offset times). In visual experiments, events often stand for a sequence of visual displays, but you can also create a non-visual event for any occurrence, for example, for delivering a sound tone.



Events are grouped in layers and can be organized in a hierarchical manner, where one parent event incorporates a number of subevents. Within each layer only one event can be active at any one time, but multiple events can be presented in parallel on different layers.

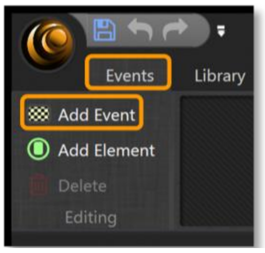
Flow Routes

The order of the events on a layer is defined by directional flow routes that can be created to link any pair of events (or, the same event in a recurrent manner). A type of a flow route, indicated by the route color, defines when linked events are switched:

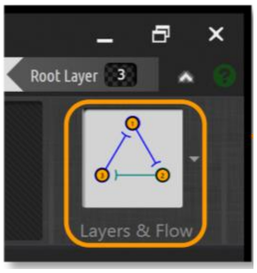
- The time-gated route takes in account a user-defined duration of the ancestor event.
- The conditional route is gated by a user-defined logical expression.
- The mix route is gated by AND/OR combinations of the duration and condition of the event.

The routes allow flexible organization of the event flow in scenarios that require sequential presentations, loops and conditional branches. For example, repeated trials in EventIDE experiments can be implemented with a looped event flow.

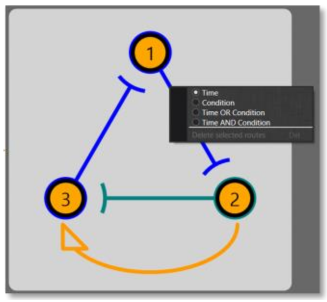
To create an event flow, follow these steps:



1. Create a sequence of the events. The first event in a row will always be a startup event.

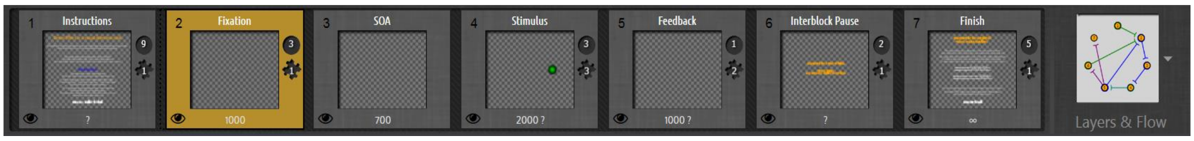


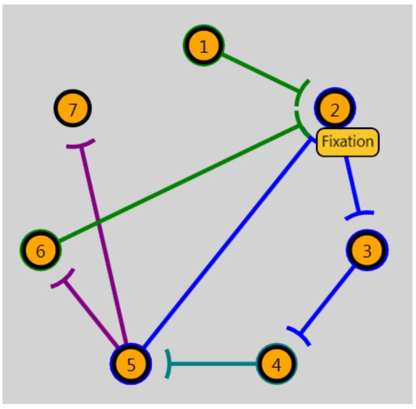
2. Open the Layer & Flow editor by pressing the rightmost button in the Event ribbon tab.



3. Add routes by dragging connectors between pairs of the ancestor and the successor events in the flow panel on the right.
4. Once a route is created, make a right click on it and select the type from the context menu.
5. Logical expressions for the conditional routes can be defined via the Snippets panel in the main window (see section 6 - Coding).

The following pictures shows 7 events and flow routes that make a trial structure for the Simon task, available in the demo gallery:



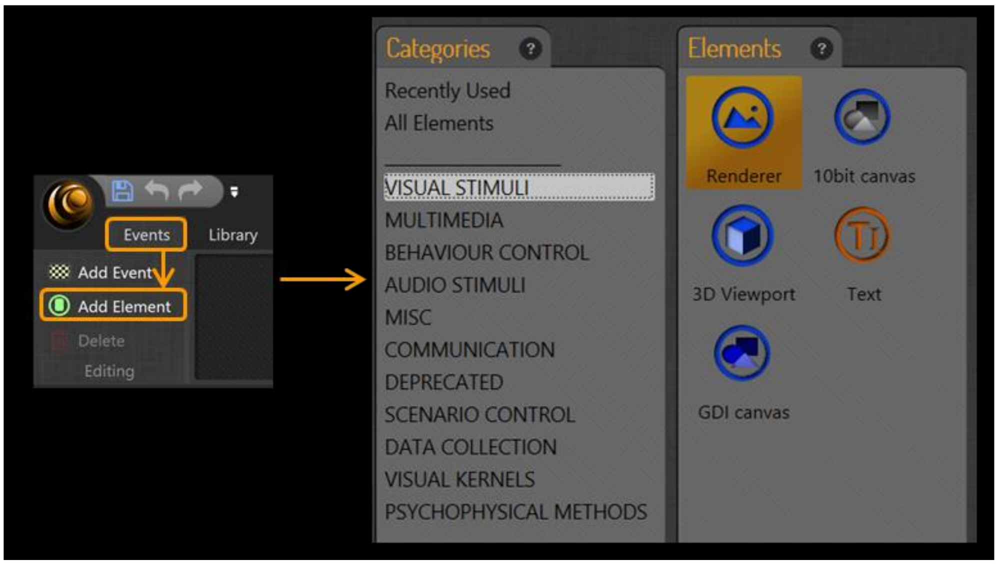


Notice that the trial loop starts here from the second event, since the first event contains an instruction screen. One of conditional routes leads to the last event '7', which shows a final "Thank you" message.

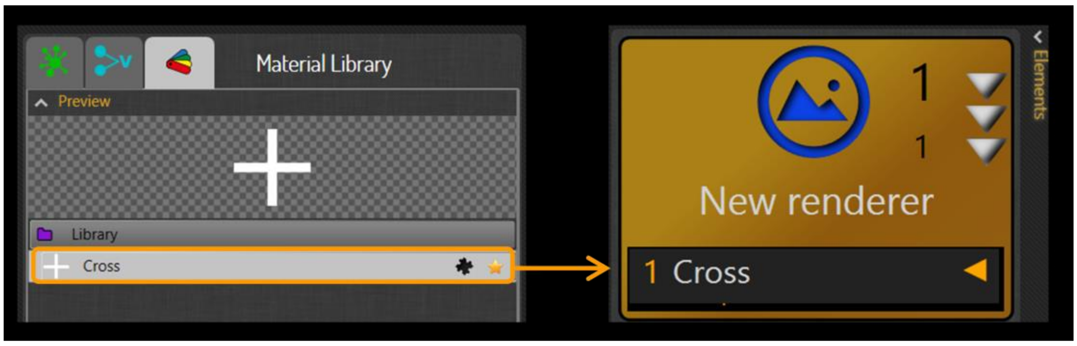
3. Presenting Stimulus and Managing Stimulus list

Presenting Stimuli

EventIDE allows presentation of all types of stimuli in experiments, such as pictures, patterns, text, object animations, videos, sounds, spoken sentences, etc. Some stimuli such as the Gabor patches, can be generated by the program in realtime, others are derived from materials that are loaded into experiment's Material Library. Consider, for example, creating a visual stimulus from a bitmap. After a bitmap is loaded into Library (see section 1 of this guide), you need to create a dedicated EventIDE element: a functional object that will render the bitmap on the surface of the target event. Select the event, where you want to place the stimulus, then navigate to Events ribbon tab and press the 'Add Element' button, as shown below.

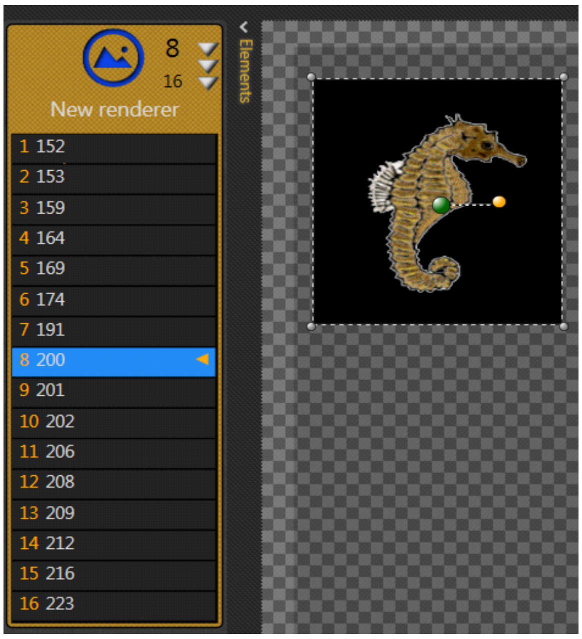


Select the 'Visual Stimuli' category in a dialog window and double-click on the Renderer element icon in the right panel. The Renderer element will be added to the current event and will be listed in the Element panel on the left of the application window. Next, open the Material Library, locate the source bitmap and drag it over the screen on to the Renderer element in the Element panel. After releasing the bitmap, it will be added into an expandable item list of the Renderer element as it shown on the screenshot below. At the same time, the Renderer will draw the bitmap on the event surface within Renderer's viewport. The size and position of the viewport can be adjusted at will.



Stimulus Lists

The Renderer element also allows the creation of switchable stimulus lists. In order to make a list, select several graphics items (images or XAML items) in the Material Library and drag them on to the Renderer elements all together. The items will be added into the element list, with the selected item shown in the Renderer viewport. The list can be expanded and collapsed by pressing the triple down arrow button in the right part of the element icon. At runtime, the item can be selected by its index or name in the list.



Items in elements' stimulus lists can be deleted, moved and resorted manually. However, for a list with many items, it is recommended to choose a sorting mode in the Library before dragging items to an element. Once created in one element, a stimulus list can be copied to other elements.

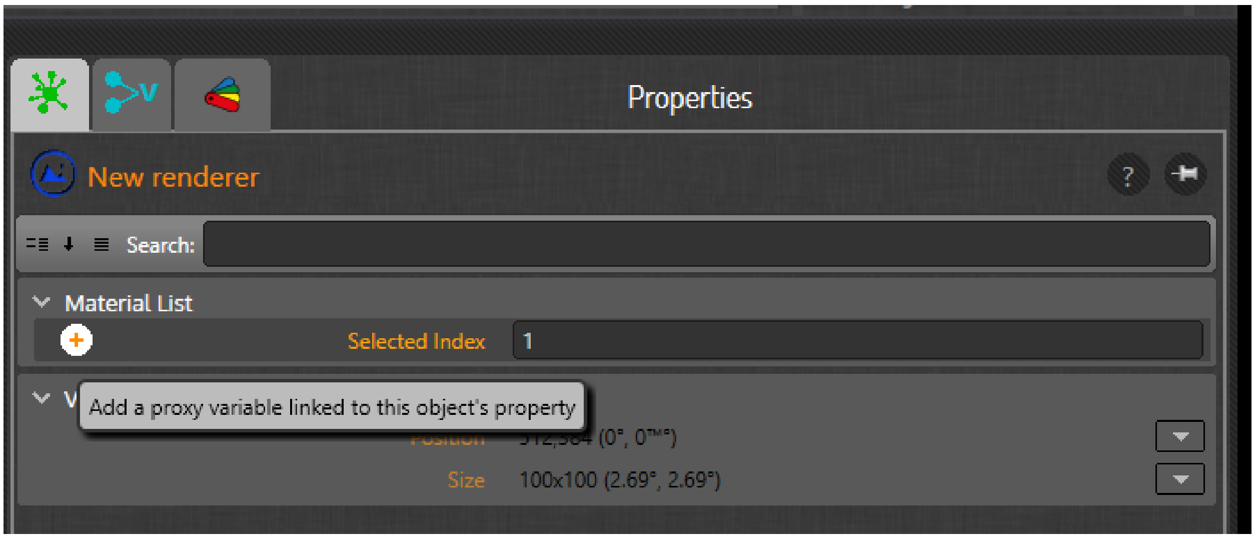
Stimulus lists that contain audio clips or 3D models can be created in a similar way, except that the appropriate element has to be used instead of the Renderer. The following table shows what elements in EventIDE handle the most common types of stimuli.

Stimulus type	Element	Compatible materials	Stimulus list
2D images and vectors graphics, including gratings, gradients, checkboards, etc.)	Renderer element	Bitmaps, XAML vector graphics	Yes
Text, letters	Text element	-	-
Audio clip	Wave Player element	Wave audio	Yes

Audio clip	Direct Sound element	Wave audio	Yes
Auditory clips, music	Audio Player element	Wave, Mp3, MIDI	Yes
Audio beeps	Beeper element	-	No
Video clips	Video Player element	Common video formats	Yes
3D models	3D Viewport	XAML 3D model	Yes
Gabor grating patches	Gabor Grating element	-	No
Circular checkboard	Circular checkboard element	-	No
Circular grating	Circular grating element	-	No
Random-dot motion	Random dot motion element	-	No
Programmable vector graphics	GDI canvas element	-	No
Spoken text	Brazen Head element	-	No

Selecting the current Item in Stimulus List

Usually, the current item in a stimulus list is selected per trial basis. To assess the index of the selected item at runtime, set a proxy variable on either the 'Selected Index' or the 'Selected Item' property of the element with a stimulus list.



Selected Index 1

Add a proxy variable linked to this object's property

Size 100x100 (2.69°, 2.69°)

Then, add a proxy variable assignment into your code, for example:

```
RendererSelectedIndex=N;
RendererSelectedItem="Animal"+N.ToString();
```

For more information about coding, see section 6 of this guide.

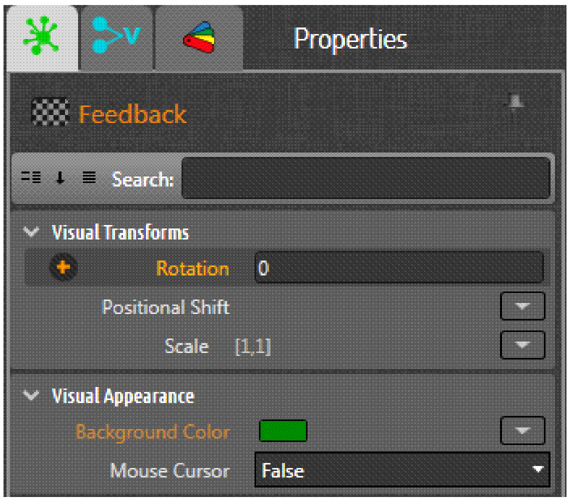
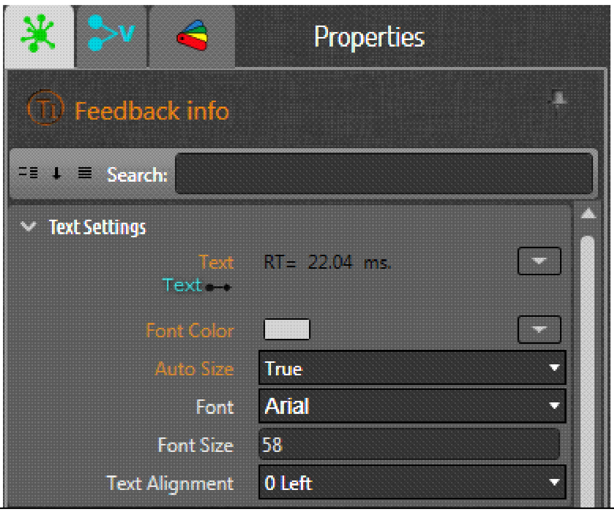
4. Understanding Properties and Proxy Variables

Property Interface

While designing experiments in EventIDE, you operate with three major classes of design objects: experiments, events and elements (so-called E-objects). The experiment is a root container for all other objects. The events define the experiment scenario and host child elements. The elements control presentation of stimulus materials and carry other functions such as response registration and hardware requests.

Each E-object has a fixed list of properties that define or indicate its status. The properties are variables or fields of simple data types: string, boolean, integer and so on. For example, an event has a property called ‘Background Color’ that can be used to set the color of the event background. Another event's property: 'Elapsed time', returns the amount of milliseconds that have passed since the event's onset and therefore can be used as a local timer.

When any of the E-objects is selected in the GUI at design-time, its properties are shown in the Property Panel (on the right of the main window) and can be modified there. The most relevant properties are marked in orange:

Properties of the event	Properties of the text element
	

Properties

Feedback

Search:

Visual Transforms

Rotation	0
Positional Shift	
Scale	[1,1]

Visual Appearance

Background Color	🟩
Mouse Cursor	False

Properties

Feedback info

Search:

Text Settings

Text	RT= 22.04 ms.
Font Color	⬜
Auto Size	True
Font	Arial
Font Size	58
Text Alignment	0 Left

When you first open the property panel, only the relevant properties are shown. Click “Show all properties” at the bottom of the property panel to see all properties.

How and when a particular property can be accessed is defined by the property class. For example, there are 'runtime status' properties (like the name of the pressed button), which can only be read at runtime. The 'settings' properties can only be changed at design-time, before an experiment starts. To learn about data type and class of a particular property, move the mouse over property’s name in the property grid and wait for a floating tooltip.

Properties provide the only interface to control the design objects in EventIDE, both in design and at runtime. The property interface is simple and intuitive, it merges together variables (data) and procedures (actions) found in conventional programming. Imagine that you need to redraw a stimulus at a new position at runtime. The conventional pseudo-code requires a sequence of function calls for this operation:

```
Hide(myStimulus);
Move(myStimulus,Xnew,Ynew);
Show(myStimulus);
```

In EventIDE the same result can be achieved simply by assigning new coordinates to the Position property of a stimulus object:

```
mystimulusPosition.X=Xnew;
myStimulusPosition.Y=Ynew;
```

EventIDE automatically handles all necessary updates for this property change. The stimulus will be immediately removed and redrawn at the new position.

To summarize, the following are important facts about properties in EventIDE:

- In EventIDE, properties provide only an interface to functions and data of the E-objects. The properties can be accessed via the GUI at design-time and in the user code at runtime.
- Properties combine concepts of data variables and functions. A change of the object's property causes an automatic update process. As per example above, if you change a visual property, it will trigger a redrawing process.
- Be careful when you change properties at runtime. Internal update processing may degrade timing accuracy. It is recommended to complete all trial preparations before the stimulus presentation is started.
- At request, properties can also return the current runtime state of E-objects. For example, you can check whether a keyboard button is in the down or up state. The status properties can naturally be used to control the flow of your experiment.

Proxy Variables

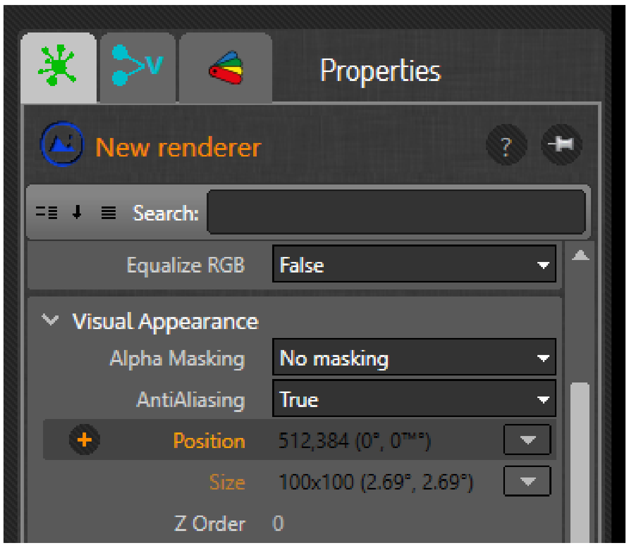
To provide access to the E-object's properties at runtime, the so-called Proxy Variables are introduced in EventIDE. A proxy variable is a reference label that can be created for any property of the selected E-object. A proxy variable then can be accessed by the user code.

Imagine that you need to set the color of one of the text blocks that is shown in your experiment. In EventIDE you can create a proxy variable for the 'Font Color' property of the selected Text Element. Then, the color of the text can be changed by code:

```
FontColor="Red";
```

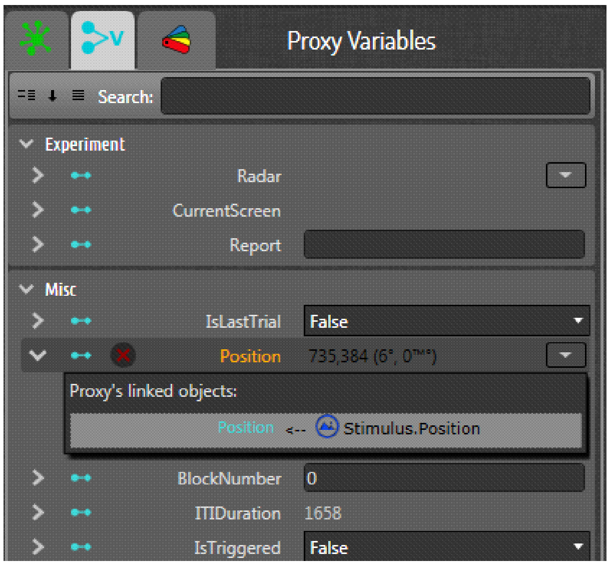
Thus, the proxy variables provide a simplified way to manipulate the E-objects in the runtime code. All proxy variables have generic data types (the same as the underlying property) and behave as conventional global variables in the code.

It is important to note that you can create proxy variables that are linked to multiple E-objects. The 'hub' proxy variable allows changing the same property on multiple objects at once, whereas the 'array' proxy variable allows accessing the same property on multiple objects by the object index.



Equalize RGB	False
Visual Appearance	
Alpha Masking	No masking
AntiAliasing	True
Position	512,384 (0°, 0™°)
Size	100x100 (2.69°, 2.69°)
Z Order	0

To create a proxy variable, select the E-object(s) and click on a chosen property in the Property Panel. If a property can be linked to a proxy variable, the "plus" button will appear to the left of its name (see the picture on the right).



Radar
CurrentScreen
Report

IsLastTrial False
Position 735,384 (6°, 0™°)
Proxy's linked objects:
Position <-- Stimulus.Position
BlockNumber 0
ITIDuration 1658
IsTriggered False

Later, you can use the special proxy panel to administer all created proxy variables. For example to rename them or to browse, add, reorder and remove the linked E-objects (see the picture on the left).

Structural Properties

There are several properties among the E-objects that have structural data types. The struct is a special data type that aggregates several data fields, which can be accessed by a field name. For example, the position, size and color properties on the E-objects have the following struct types: clPoint, clSize and stColor.

When a struct property is accessed in the user code (via a proxy variable), you have to use the 'dot syntax' to access subfields:

```
// stColor
TextColor.A=255; // transparency
TextColor.R=127; // red channel
TextColor.G=127; // green channel
TextColor.B=127; // blue channel

// clPoint
Position.X=512;       // in screen pixels
Position.Y=368;      // in screen pixels
TextPosition.R=5;     // center offset in polar coordinates (vis. degrees)
TextPosition.Theta=90; // angle in polar coordinates (degrees)

// clSize
StimSize.Width=200;       // in screen pixels
StimSize.Height=100;       // in screen pixels
TextAreaSize.aWidth=5.5f; // in visual angles
TextAreaSize.aHeight=2.1f; // in visual angles
```

You can learn more about EventIDE's built-in data types in the wiki article.

5. Creating Randomization Design

Randomization

EventIDE does not force users to use a fixed approach to randomization in experiments. You can always program your own randomized scenario as flexible as needed. However, there are a number of instruments (including a visual designer in the Roulette element) that allow easy implementation of common randomized designs with no or little code.

Firstly, when you need a single random number, you can utilize system random functions of C#. For example the following code produces two random integer numbers in the uniform range of 0...10 and one double number in the range 0..1:

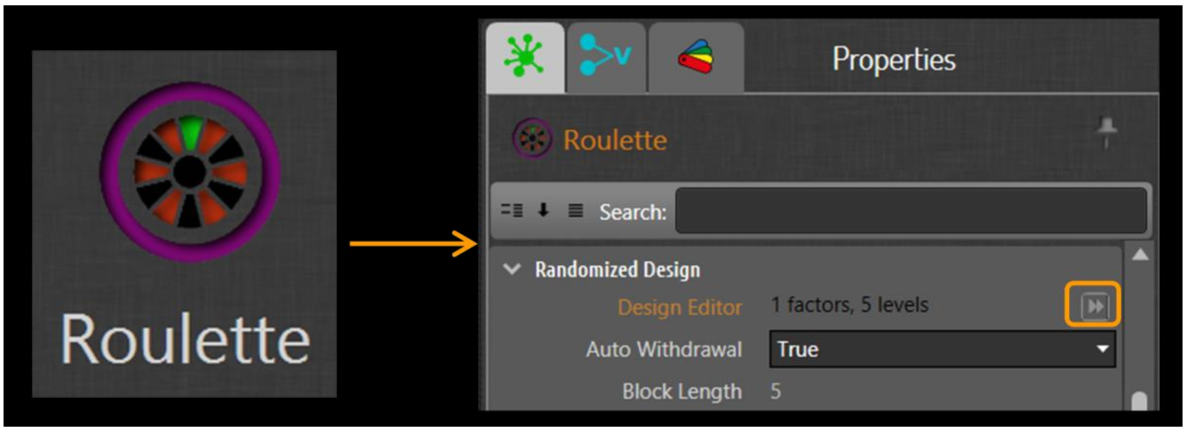
```
Random R=new Random();     // initiate a system random number generator
int NumberA=R.Next(0,10); // 10 is not included
int NumberB=R.Next(0,10);
double NumberD=R.NextDouble();
```

Secondly, you can easily load a randomized condition list generated in other programs, such as Excel or Matlab. Please see this FAQ article for a working example.

Finally, for common randomization scenarios EventIDE provides a dedicated element, called the 'Roulette'. The Roulette allows the creation of randomized factorial designs and applying them in experiments. The element takes a number of user-defined factors and probabilities and generates randomized trial blocks at runtime. When the Roulette element is integrated into a looped trial structure, it can withdraw randomized factor values for each trial and automatically count trials and blocks in experiments.

Creating a Randomized Design with the Roulette Element

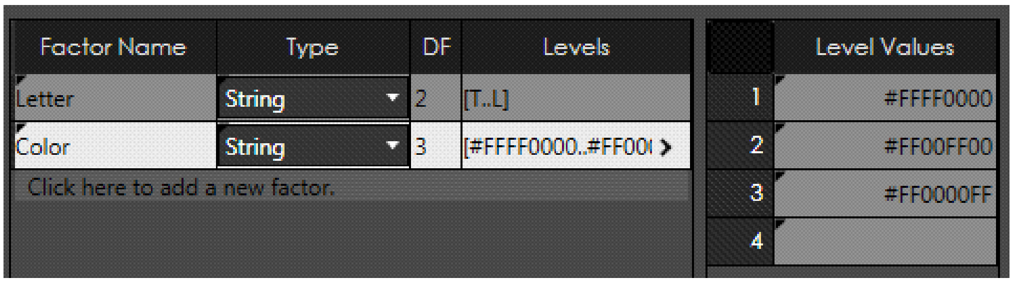
Let's consider creating a two-factor randomized design on an example. Add the Roulette element, select it to access the element's properties and open the design editor via the property panel:



Design Editor: 1 factors, 5 levels
Auto Withdrawal: True
Block Length: 5

In the editor you can start by creating a list of the factor (variables), whose values will be randomized in trials. After adding a new factor, chose its data type in the second column. Each factor has to be supplied with levels. Levels are possible values that the factor can have. For example, an integer factor can have levels of [1,2,3,4,5]. A string factor coding color can have ['#FFFF0000','#FF00FF00','#FF0000FF'] or ['Red','Green','Blue'] values.

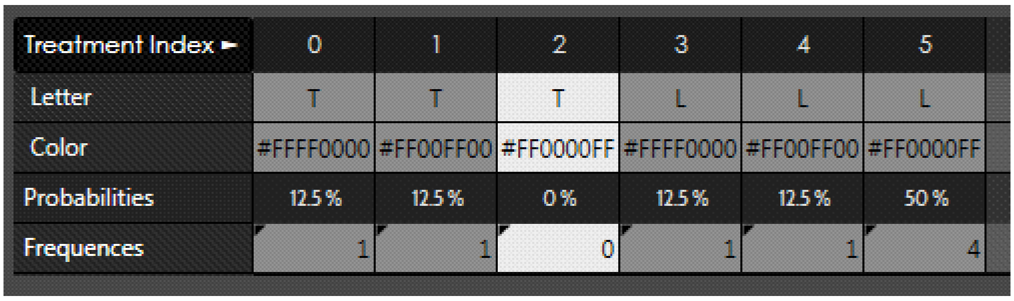
To define levels, select a factor and edit its lookup list in the central grid. You can either type level values manually or paste them from external tables (for example from Excel). Alternatively, you may write and run an inline script to generate level values.



Factor Name	Type	DF	Levels
Letter	String	2	[T..L]
Color	String	3	[#FFFF0000..#FF00(
Click here to add a new factor.			

	Level Values
1	#FFFF0000
2	#FF00FF00
3	#FF0000FF
4	

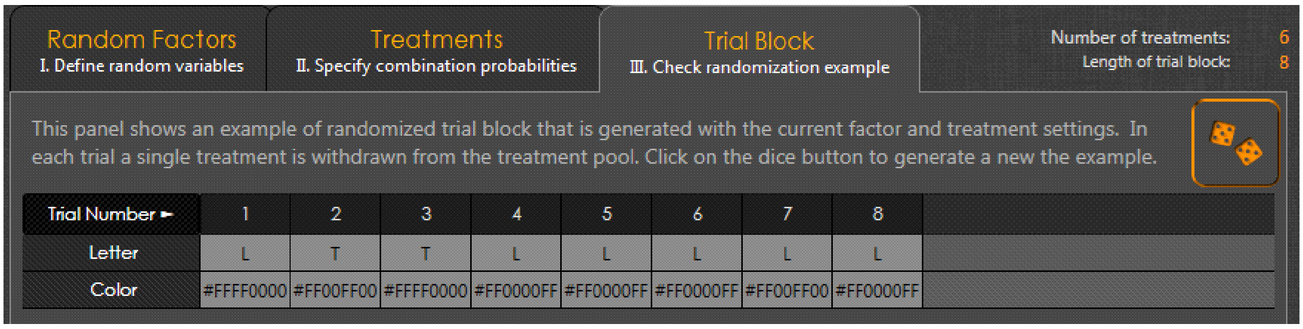
Once you added all factors, switch to the Treatment tab inside of the editor and define probabilities by setting frequencies for trial treatments. The trial treatment is a unique combination of levels from different factors. All possible treatments are shown in columns of the Treatment grid:



Treatment Index ►	0	1	2	3	4	5
Letter	T	T	T	L	L	L
Color	#FFFF0000	#FF00FF00	#FF0000FF	#FFFF0000	#FF00FF00	#FF0000FF
Probabilities	12.5 %	12.5 %	0 %	12.5 %	12.5 %	50 %
Frequences	1	1	0	1	1	4

In the last row you can modify integer frequencies for each treatment in the resulting trial block. Notice that treatment 2 in this example has the frequency 0; it means that this treatment is excluded. Treatment 5 has the frequency 4, meaning that it will appear in a trial block four times, with the auto-estimated probability of 50% across all trials. The sum of all frequencies gives a total number of trials in one block (8 in the above example). If you need to increase a block length, but keep the probabilities then scale up all non-zero frequencies. In the above example, the doubled frequencies 2,2,0,2,2,8 will result in a block of 16 trials.

Finally, you may check the resulting trial block in the Trial Block tab (press the dice icon to generate a new block example). The trial block is made up of the treatments shuffled in a random order, thus it can be used directly for providing random values in a sequence of experiment trials.


Random Factors
I. Define random variables
Treatments
II. Specify combination probabilities
Trial Block
III. Check randomization example

Number of treatments: 6
Length of trial block: 8

This panel shows an example of randomized trial block that is generated with the current factor and treatment settings. In each trial a single treatment is withdrawn from the treatment pool. Click on the dice button to generate a new the example.

Trial Number ►	1	2	3	4	5	6	7	8
Letter	L	T	T	L	L	L	L	L
Color	#FFFF0000	#FF00FF00	#FFFF0000	#FF0000FF	#FF0000FF	#FF0000FF	#FF00FF00	#FF0000FF

Once you close the design editor, the Roulette element will automatically create dynamical properties (and proxy variables, if chosen) that will return the individual factor values at runtime. You need to write code to read these properties and adjust your trial variables (for example a stimulus color) in each trial.

Using the Roulette Element in Experiments

The base function of the Roulette element is to generate a block of random treatments in accordance with the design settings that were chosen. Such blocks can be accessed in code when the Roulette operates in a manual mode (defined by "Withdrawal Mode" property of the Roulette). The individual treatments in the trial block can be obtained by the index (trial number). The block can be regenerated at request.

More conveniently, in the 'repeated trials' mode the Roulette can automatically withdraw a random treatment for every new trial in your experiment. For that mode you have to place the Roulette element into the first event in a trial loop, because Roulette detects the start of a trial, as the onset of its parent event. The element also counts trials and when all treatments in the internal block are withdrawn, the Roulette automatically generates a new random trial block to continue withdrawals.

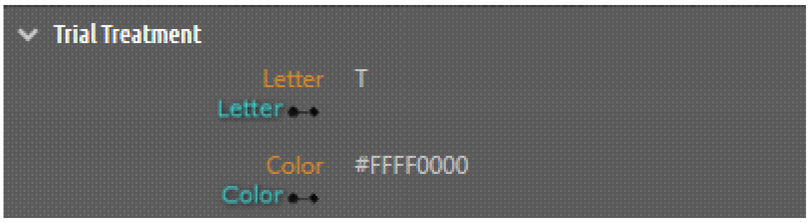
Follow the steps below to set up the Roulette in each of two modes:

Automatic withdraw mode for repeated trials (default)

1) Add the Roulette element to the first event of your looped trial structure. This gives an alignment point for element's internal processing. The Roulette will automatically withdraw a new treatment on the onset of the parent event and also translate a number of such onsets to a trial count.

2) Create a randomized design with the help of the Roulette's design editor.

3) Locate the dynamic properties of the Roulette element in the Trial Treatment group and check which proxy variables are linked to it (these proxies are created automatically):



Trial Treatment

Letter T
Letter

Color #FFFF0000
Color

4) Write code that carries trial preparations (for example stimulus and color selection) using values returned by these proxy variables:

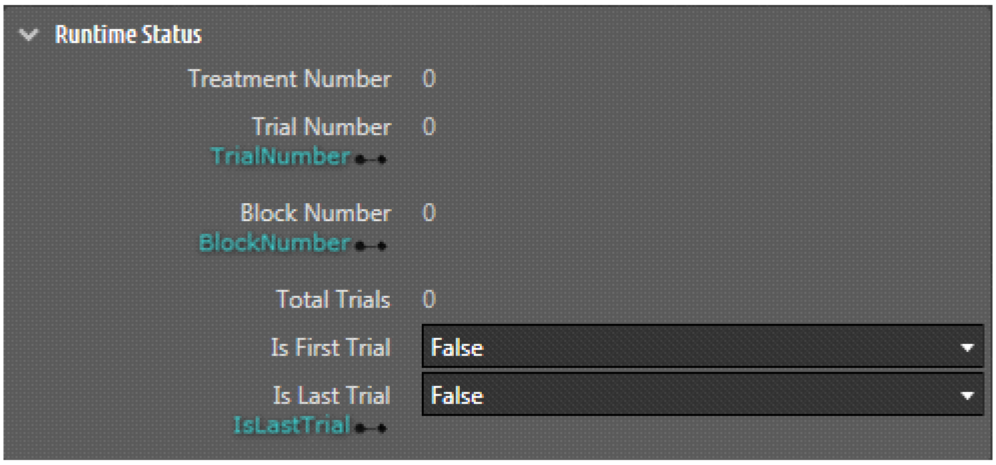
```
StimulusText=Letter;//StimulusText is a proxy on the Text property of the Text element.
TextColor=Color;// TextColor is a proxy on the Text Color property of the Text element.
```

Make sure that the preparation code is called after the onset of the Roulette's event, but before a randomized stimulus is presented. Usually, the best location for trial preparations is the 'After Onset' snippet of the first event in a trial.

5) When you need to cancel a particular withdrawal and repeat the same treatment in later trials (it is often used after the incorrect participant's response), use the Recall Treatment property of the Roulette. Assign 'true' to the property in the code, before a new trial starts.

```
RecallTreatment=true;
//
```

6) Navigate to the status properties of the Roulette located in the 'Runtime Status' group:



Treatment Number	0
Trial Number (TrialNumber)	0
Block Number (BlockNumber)	0
Total Trials	0
Is First Trial	False
Is Last Trial (IsLastTrial)	False

These read-only properties return runtime trial statistics, which can be used to control the flow of an experiment. For example, the property 'Is Last Trial' returns true when the Roulette withdraws the last treatment from its current trial block. You can use this property as a condition to pause a trial loop between blocks and make a break.

The properties showing trial and block numbers are handy automatic counters that can be used anywhere in your code.

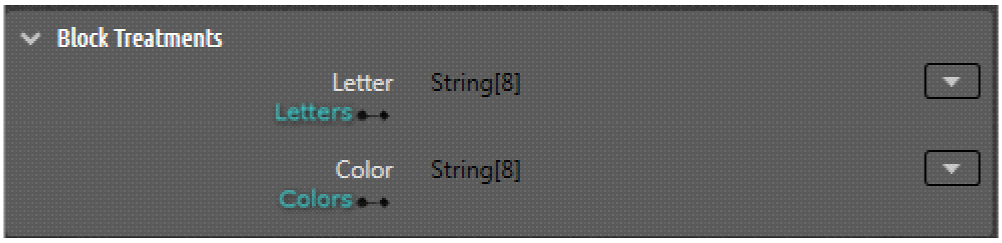
7) When the Roulette element returns factor values for the last trial in a block, it automatically generates a new trial block with the same design settings. The only difference is that trial treatments are reshuffled in a new order.

Manual withdraw mode

1) Add the Roulette element to any event in your experiment. In manual mode the element does not need to be aligned with the parent event's onset. This is because the trial block will be accessed manually.

2) Create a randomized design with the design editor of the Roulette element.

3) Locate dynamic properties of the Roulette element under the 'Block Treatment' group and add proxy variables for the factors:



Letter (Letters)	String[8]
Color (Colors)	String[8]

4) The proxies that are created allow you to read any treatments within a trial block by an index. For example:

```
int n=7;                      // seventh trial in the block.
StimulusText=Letters[n-1];
// StimulusText is a proxy on the Text property of the Text element
TextColor=Colors[n-1];
// TextColor is a proxy on the Text Color property of the Text element. Note, n-1 is
used because indices begin at zero.
```

Since treatments are already randomly shuffled in the trial block, you can use an incremental index from trial to trial.

5) You should not use runtime status properties of the Roulette, because in manual mode they are not updated at runtime.

6) When you need a new trial block, assign ‘true’ to the Reset Block property of the Roulette. A new trial block will be generated with the same design settings.

Using the Roulette to obtain a Random Subset of a larger Value Pool

Imagine that in each trial of your experiment you need to select a subset of 5 non-repeated colors that come from a larger pool of predefined colors (for example to present random and unique color targets). For this task you have to: 1) Switch Roulette into manual mode. 2) Define a complete list of color values in a one-factor randomized design. Then link two proxy variables on the Roulette properties:

1. For the color factor name that will appear under the ‘Block Treatment’ group
2. For ‘Reset Block’ under the ‘Runtime Commands’ group

The following code shows how these proxy variables can be used:

```
for (int i=0;i<5;i++){
StimulusColor[i]=Colors[i]; //assign random colors, taken by the index from
                             Roulette's block, to stimuli.
}
ResetBlock=true; // force Roulette to regenerate a new block, so that a new subset of c
olors will be available on the next trial.
```

6. Writing Code

EventIDE is a programming platform that supports a family of .NET languages for user coding. The current version only supports C# 4.0. C# (pronounced "C sharp") is among the most popular languages currently used worldwide. It represents a next generation of the classic C with the brand new features which include simplified syntax and language power comparable to the professional C++.

Coding with C# in EventIDE is a simple task even for a novice user:

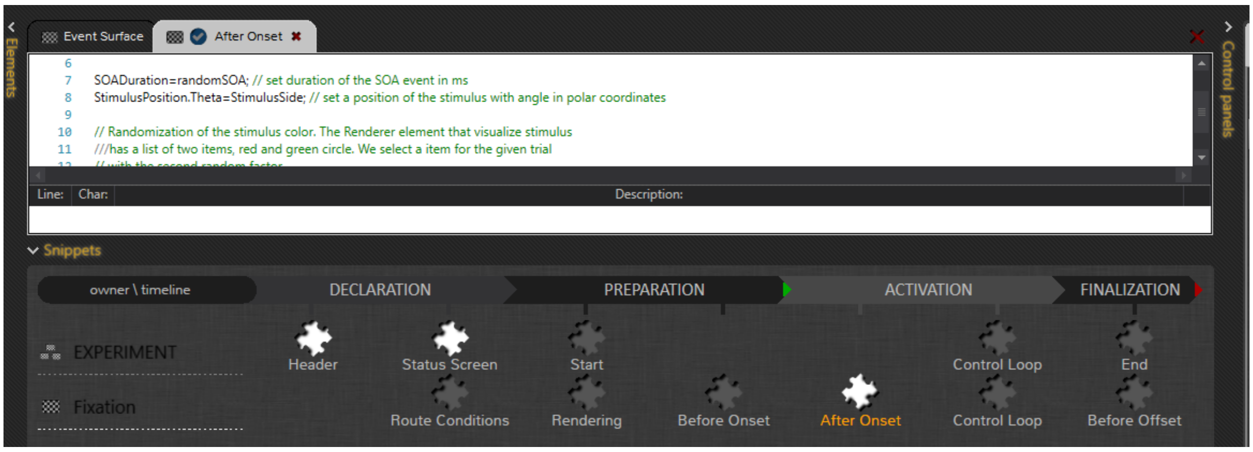
- The C-alike syntax of C# is very generally known and familiar.
- C# is well documented, so you can easily find a lot of examples and tutorials online.
- No knowledge of the object-oriented paradigm is necessary in EventIDE.
- There are no functions, procedural calls and libraries related to the experiment design and, correspondingly, no necessity to learn them. Instead, EventIDE provides the property interface to its functional objects (see the section 4 of this guide).

As result, your code never has to go beyond standard programming statements such as variable assignments, loops and conditions.

Event-driven Model and Code Snippets

Coding in EventIDE requires a different approach to traditional procedural programming. In the procedural programming one usually writes a long program containing a main control routine, where sequential calls to other functions and subroutines are made. Instead, the event-driven paradigm in EventIDE requires no main control routine at all, because a sequence of designed events already represents an experiment control flow visually and logically. You write your code in small individual pieces, called snippets, which are attached to selected stages of your experiment. In other words, you create code handlers (or callbacks) for various experiment events. For example, you can write code that will be executed on pressing the response button and other code is executed on the start of a new trial.

Although the snippets are independent pieces of code, or mini-routines, you can always exchange and share data across them. First, you can define conventional global variables that are accessible in every snippet. Second, the EventIDE proxy variables also act as global variables and are accessible anywhere in the code. Finally, you can define global functions that can be called from different snippets.



```

SOADuration=randomSOA; // set duration of the SOA event in ms
StimulusPosition.Theta=StimulusSide; // set a position of the stimulus with angle in polar coordinates

// Randomization of the stimulus color. The Renderer element that visualize stimulus
///has a list of two items, red and green circle. We select a item for the given trial
```

Types of the Code Snippets

The type of a code snippet is determined based upon where and when the snippet is called in the course of an experiment. The three classes of EventIDE E-objects: experiment, events and element, provide a fixed number of snippet points (see the table below), that are aligned with objects' runtime activity. For example, you can add a code snippet to the onset of any event. Thus, the order of snippet calls in EventIDE is solely defined by the flow of your experiment.

The notable exceptions are the Header and the Status Screen snippets of the experiment object. These 'declarative' snippets are not called at runtime, you use them to declare the global variables and functions in the Header snippet and the XAML content and layout in the Status Screen snippet.

Another special type of snippet is the 'Control Loop' snippet that is called repeatedly (usually in intervals of ~1ms). The Experiment object handles the global 'Control Loop' snippet, which is called over the entire experiment. Every event may have its own 'Control Loop' snippet, which is called while the event is active. The 'Control Loop' snippet is handy for continuous operations, such as status checks, animations and data readouts. The periodicity of the 'Control Loop' snippet is not deterministic and depends on the current load of the EventIDE core.

All snippet types are listed in the following table:

Snippet owner	Snippet Name	Description and role
Experiment object		

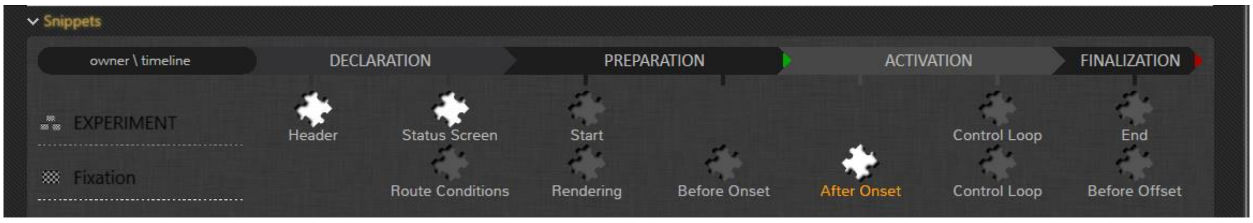
	Header	The ‘Header’ snippet is intended for the declaration of global variables and functions, which will be accessible in other snippets.
	Status Screen	The ‘Status Screen’ snippet is intended for creating a runtime status screen (if required) with a descriptive XAML language.
	Start	The ‘Start’ snippet is called only once, at the start of the experiment and, therefore, intended for various initialization actions and variable assignments at runtime.
	Control Loop	The ‘Control Loop’ snippet of the experiment object is called repeatedly and intended for continuous actions during an entire experiment run. To ensure accurate timing in an experiment, this snippet should contain only relatively short code that executes quickly.
	End	The ‘End’ snippet is called once, at the end of experiment. Therefore, it is intended for various finalization actions.
Event object		
	Route Conditions	The ‘Route Conditions’ snippet is intended for gating the conditional flow routes, which originate in the owner event. Each conditional snippet may contain a single logical expression of a boolean type. The expression will be continuously evaluated at runtime until its result returns true. Then the corresponding route is selected for a step in an event flow.
	Rendering	The ‘Rendering’ snippet is called each time when the visual content of the owner event is changed (both on-screen and off-screen). The snippet is intended for signalling or collecting statistics of visual changes and, therefore you should not use it to program the experiment logic.
	Before Onset	The ‘Before Onset’ snippet is called just before the onset of the owner event, when the event is not activated and yet stays off-screen. This snippet is normally used for ‘last moment’ changes. Since a snippet call takes time, it is recommended to avoid using this snippet when accurate onset timing of the owner event is required.
	After Onset	The ‘After Onset’ snippet is called once immediately after the event onset. It is intended for all actions that have to be synchronized with the onset of the owner event. The snippet is safer for timing accuracy when compared to ‘Before Onset’ and ‘Before Offset’ snippets. Therefore, it is recommended to use in all cases, when you are uncertain in where to place your code.
	Control loop	The ‘Control Loop’ snippet of the event object is called repeatedly, while the event is active. The snippet is intended for continuous actions within the event. To ensure accurate timing in an experiment, this snippet should contain only short and fast code.
	Before Offset	The ‘Before Onset’ snippet is called when an event flow is switching another event. The snippet is intended for offset actions right before onset of another event. This snippet is called

		in a pair with the 'Before Onset' snippet of next event. It is recommended to avoid using this snippet, when an accurate timing of the next event onset is required.
Element object		
	Triggered	The 'Triggered' snippet is called, when a status of the owner element is changed and it has to be reported. The status change can be the detection of a button press or arrival of data transmission through a port, in accordance with a role of the owner element. The 'Triggered' snippet can be called asynchronously, at any time, while element's parent event is active. Notice that not all elements provide the triggering snippet.

Accessing the Snippets

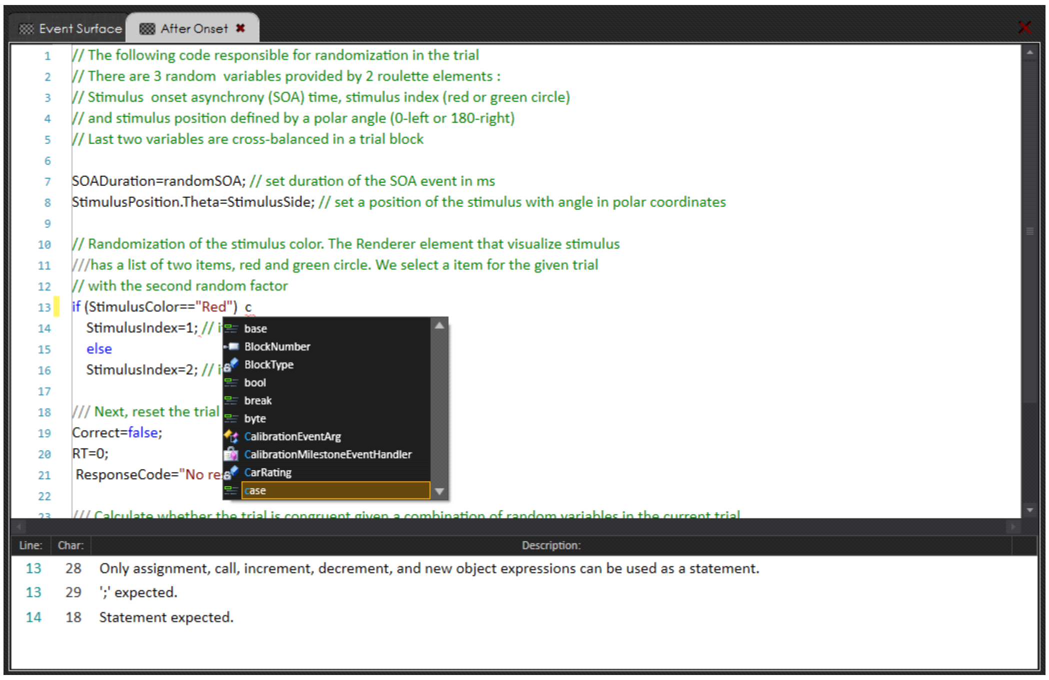
At the bottom of the main window, in the Snippets Panel, you can select a snippet and open it by double clicking on its icon.

The content of the Snippet panel is refreshed upon the selection of the E-objects in the GUI (for example you need to select a particular event to access its snippets). Thus, the Snippet Panel always shows the snippets of the experiment object in the first row, snippets of the selected event in the second row and snippets of the selected event's elements in the third and furthers rows.



Working with the Code Editor

EventIDE provides a code editor with automatic syntax highlighting, error checking and code completion. The code completion (the black menu on the screenshot below) helps to complete typing of common language statements and names of global and proxy variables. Multiple snippets can be opened at the same time in the code editor. All code changes are automatically saved when closing the editor.



```
1   // The following code responsible for randomization in the trial
2   // There are 3 random  variables provided by 2 roulette elements :
3   // Stimulus  onset asynchrony (SOA) time, stimulus index (red or green circle)
4   // and stimulus position defined by a polar angle (0-left or 180-right)
5   // Last two variables are cross-balanced in a trial block
6
7   SOADuration=randomSOA; // set duration of the SOA event in ms
8   StimulusPosition.Theta=StimulusSide; // set a position of the stimulus with angle in polar coordinates
9
10  // Randomization of the stimulus color. The Renderer element that visualize stimulus
11  ///has a list of two items, red and green circle. We select a item for the given trial
12  // with the second random factor
13  if (StimulusColor=="Red")  c
14     StimulusIndex=1; // i
15     else
16     StimulusIndex=2; // i
17
18  /// Next, reset the trial
19  Correct=false;
20  RT=0;
21   ResponseCode="No re
22
23  /// Calculate whether the trial is congruent given a combination of random variables in the current trial
```

Line:	Char:	Description:
13	28	Only assignment, call, increment, decrement, and new object expressions can be used as a statement.
13	29	';' expected.
14	18	Statement expected.

Important Notes about coding in EventIDE

- While running an experiment, EventIDE continuously monitors and maintains accurate timing of events. When a code snippet is called, this work is interrupted and full control is given to the user code. Therefore, try to avoid any blocking statements that may delay a return from your snippet. For example, the following cycle block is undesirable, because the while loop may block a processing in EventIDE core for a significant amount of time:

```
While (ElapsedTime<5000)
{
//Wait
}
```

- When you use the proxy variables in code snippets, notice that properties of objects, linked to the proxies, are not updated until the snippet call is compete. Inside of the snippet, the proxy variables can be changed multiple times, but only the final value will be actually passed to the linked object.

- You can find out how long it took to run your snippet. Once an experiment run is completed, you can find the measured execution time in the statistical properties of the owner event.

- If you have a complex piece of code that is repeatedly called in many experiments, e.g. implementation of a psychophysical method, consider creating a custom reusable element, which will perform the same computations and return results via the property interface in EventIDE This approach is more efficient for re-utilizing complex code, as snippets are intended mainly for a control of the experiment flow.

- There are runtime debugging tools in EventIDE, which allow: 1) watch and apply changes in global and proxy variables 2) monitor snippet calls and set breakpoints. In addition, you can always output your own debugging information into the data report or the Status Screen.

7. Responses Registration

EventIDE elements provide various options for registering participants' responses and other input signals. Some elements communicate with standard human input devices, such as keyboards, joysticks and microphones. Some elements can receive and send data through computer ports or over a network. Other elements read an input of specialized research hardware, such as eye-trackers and response boxes.

According to the EventIDE element's logic, the input is registered by an element within the duration of the parent event. This is helpful if you want to limit a response interval for participants. When you need a listening input during the entire experiment, you can create a super event (a container for other events) and place a chosen input element there. In addition, some EventIDE elements work in global scope by design.

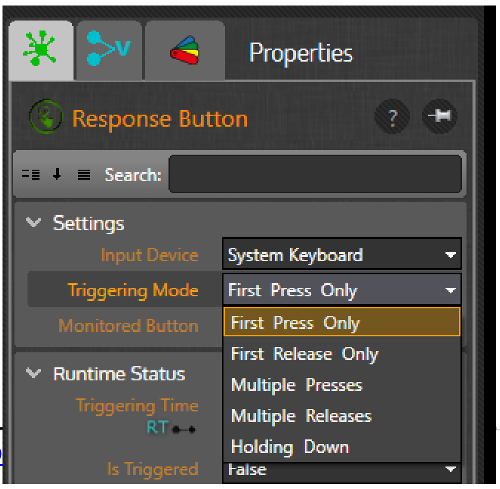
The following table lists all input elements.

Element	Description
Input Registration	
Button	The Button element detects a button presses/releases on standard input devices that are supported by the DirectInput API. Such devices include keyboard, mouse and a variety of joysticks and gamepads.
HID Input	The HID Input element can receive and decode an input from HID (Human Interface Devices) protocol. This protocol is supported by a broad range of USB computer input devices such as keyboards, mice, joysticks and importantly, the majority of response boxes. The element is capable of reading both device's button states and analogue axis.
MIDI input	The element allows reading and logging an input coming from external MIDI devices such as a piano that is connected via USB or MIDI port.
ROI	The ROI element creates a 'Region Of Interest' on the screen and continuously evaluates any positional input in relation to that area. By default, the positional input is provided by the system mouse, but it can be also used with eye-trackers and joysticks.
Voice Recorder	The Voice Recorder element is capable of capturing audio input such as a voice, and then recording it into a WAV file.
Speech listener	The Speech Listener element allows real-time recognition of words that are spoken into a microphone. The recognized speech is converted into text, which can be used to classify a participant voice response.
Key Logger	Key Logger element registers all button presses and releases in the parent event and saves them into a text log
Cedrus XID	Cedrus XID element detects button and line states on the Cedrus input devices that support the XID (eXperimental Interface Device) protocol. The supported XID devices include the RB Series and Lumina response pads, StimTracker marker and SV-1 voice key.
Ink canvas	The Ink canvas element allows to record and visualize painting strokes made by an participant
Communications	

[LPT port](#)	The LPT port element allows bidirectional data exchange through a parallel port (LPT). The element can be used for sending/receiving synchronization triggers common in EEG studies.
UDP port	The UDP port element allows bidirectional data exchange through a standard computer network.
[COM port](#)	The element provides bidirectional data exchange through the serial port (COM) on a PC computer. The element can be used for sending digital codes that are synchronized with events in your experiment. Also, the serial port element can be used for asynchronous data transfers to external devices.
NI port	The NI port element allows bidirectional data exchange through the National instruments DAQ card.
MC DAQ port	The MC DAQ Port element allow to send and receive data over a digital port on the Measurement Computing DAQ card. For example, the element can be used to send synchronization triggers from a presentation computer.
TCP port	The TCP element allows listening input communications via the TCP network ports. The element currently does not support sending data via a TCP port, please don't use it in the output modes, as other port elements!
Mitsar Trigger	The Mitsar Trigger element sends synchronization triggers to an EEG recording carried by Mitsar EEG Studio.
NI Counter Pulse	The element allows sending a single pulse or pulse train through the output counter channel on the National Instruments DAQ cards
NI Analog Port	The element allows sending a single pulse or pulse train through the output counter channel on the National Instruments DAQ cards
Research Hardware	
Eye-trackers	A family of eye-tracking elements allow obtaining online gaze data from research eye-trackers and calibrate the gaze position to the presentation display. The eye-trackers elements are available via EventIDE AddIns (navigate to the Application menu->AddIn Manager)
EEG amplifiers	A number of EEG elements (available via EventIDE AddIns) allow reading of signals provided by EEG amplifiers
Kinect sensor	A number of Kinect elements allow communication with the low-cost [Kinect sensor](#) for 3D body and face tracking.

Using the Input Elements (on the example of the Button element)

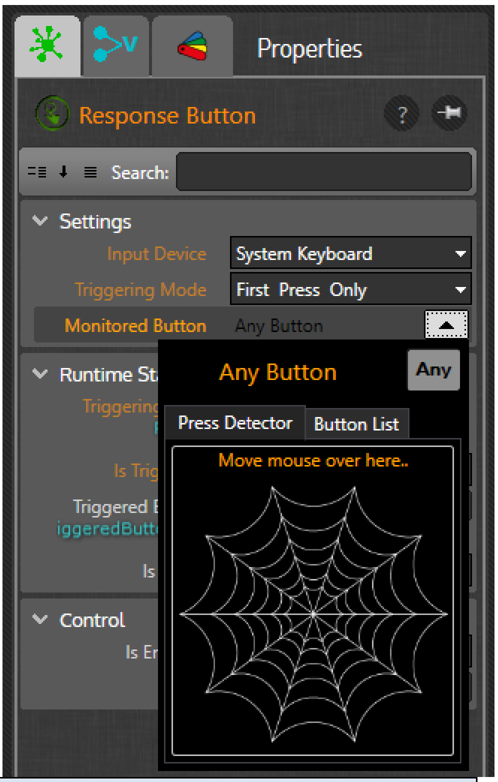
First, you need to add an appropriate input element into an EventIDE event, where you want to register participant's input. Often it can be the same event where a stimulus is presented. Once the input element is added, go to its properties and select the particular input device. Most of the input elements support different working modes. For example, [the Button element](#) allows you to select which button state should be monitored (via the Triggering Mode property).



Input Device: System Keyboard
Triggering Mode: First Press Only
Monitored Button: First Press Only / First Release Only / Multiple Presses / Multiple Releases / Holding Down
Triggering Time RT
Is Triggered: False

Since the element can detect both the press and the release of the button, the detection result is called "triggering" (in all input element as well). For the Button element you have to choose whether you want to register triggering of any device button or particular button only. For the latter case, select a particular button name via the Monitored Button property.

Note that in the same event you can use either multiple button elements detecting different buttons, or have just one element that will detect triggering of any button and report its name.



Input Device: System Keyboard
Triggering Mode: First Press Only
Monitored Button: Any Button

The button element is now ready for runtime work, but it is your task to interpret and record the registered data. When a button is pressed or released, the element refreshes its status properties and calls the Triggered snippet, where you can write the call-back code. The following runtime status properties are available for readouts on the Button element:

Status properties	Description
Triggering Time	Returns a local time of a button triggering. (The local time is a time elapsed since onset of the parent event). The triggering time is the most accurate measure of real button press and, therefore, can be used as a reaction time to stimulus.
Triggered Button	Returns a name of the triggered button.
Is Triggered	The element reports whether it has been triggered at least once.
Is Down	Indicates whether a button is currently in the pressed state.

The 'Is Triggered' property is Boolean and therefore can naturally be used to control an event flow with the conditional flow routes. For example, the parent event can be terminated after a response button is pressed. To use the button element in this way, set a proxy variable on the 'Is Triggered' property and write the following condition for an outgoing flow route:

```
(IsTriggered==true)
// the route will be opened, after a button is triggered (pressed/released)
```

Other input elements can be used in a similar fashion:

1. Select an input device in element's properties.
2. Choose whether you want to register all incoming input or a particular pattern in it.
3. Wait for a triggering input and read the status properties of the input element in your code.

8. Data Collection

Since EventIDE is a fully-featured programing platform, you virtually have no limits in choosing what data to collect and record in experiments. You can always create external files in your code and log participant responses, trial settings and statistics under custom protocols. Moreover, EventIDE performance allows you read, analyse and record raw input of hardware devices in realtime, for example EEG signals and eye-tracker samples.

Writing Data Report

EventIDE also provides a build-in data file called the 'Report'. Each time, when an experiment is started, the program creates an internal storage file with high-speed access and protection against crashes. The file uses a text format and accepts any data that you would like to record. To record your data line, simply make a code assignment to a dedicated proxy variable 'Report' that has the generic string type:

```
Report=TrialNumber+";"+StimulusIndex+";"+RT; // record data in three columns, separated
 by semicolon
//
```

If you want to make several recordings in one snippet, accumulate it in the Report variable:

```
int DataA=5;
Report= DataA; // ToString() the standard explicit operator converting from all data ty
pes to string
Report=Report+"\nTrial Number,StimulusIndex,RT"; adds a new line with headers
Report=Report+"\n"+TrialNumber+","+StimulusIndex+","+RT; // adds a new line of separate
d data
```

After the snippet is executed, an accumulated content of the Report variable is added to the end of the storage file.

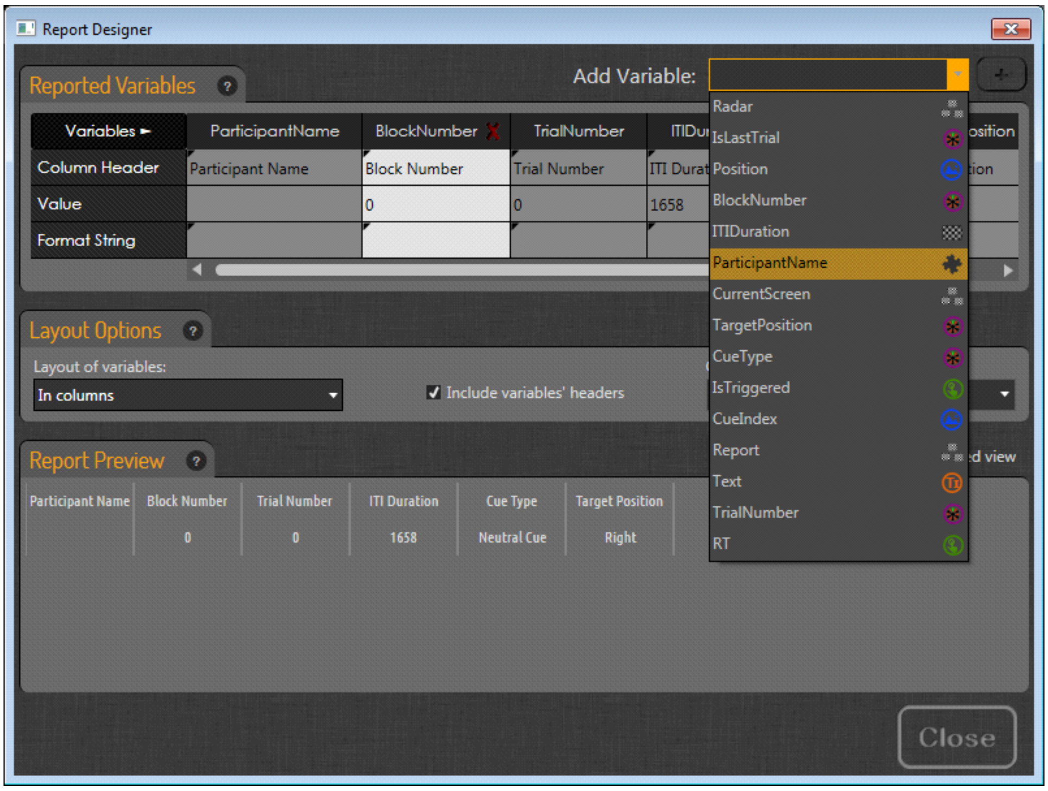
Reporter Element

In the majority of behavioural experiments data collection is standardized. Experimenters collect a list of relevant variables at the end of every trial and record them as a row line in a large table. The subject info and relevant parameters are often placed at the top of the table. If this approach suits you, EventIDE offers a simple and fully automatic way to collect data using the Reporter element.

The Reporter element removes the need to write code for collecting and formatting variables, as it shown in the example above. Instead you choose 'to-record' variables and formatting options in a visual designer. At runtime, the element will periodically collect values of the variables and generate formatted data blocks (such as table rows), which will be automatically added to the EventIDE data report.

Follow the steps below to apply the Reporter element in your experiment:

1) Add the Reporter element into the last event in your trial loop. In a typical scenario it is a moment, when all trial data is collected and variables are updated.
2) Open the visual designer via the Report Designer property of the Reporter element



Variables ►	ParticipantName	BlockNumber	TrialNumber	ITIDu	osition
Column Header	Participant Name	Block Number	Trial Number	ITI Durat	ion
Value		0	0	1658	
Format String					

Layout of variables: In columns

✓ Include variables' headers

Participant Name	Block Number	Trial Number	ITI Duration	Cue Type	Target Position
	0	0	1658	Neutral Cue	Right

3) In the dropdown list select the global and proxy variables that you want to log and add them to the table. Don't forget to recompile the Header snippet to keep the global variables updated in the list.

4) Choose an order of the added variables in the table (by dragging the column headers).

5) Define the layout preferences: layout mode, auto-headers, column separator.

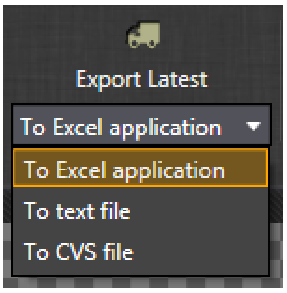
6) Check a preview of the generated data report at the bottom of the designer (only one row is shown).

7) Close the designer, find the Recording Point property of the Reporter element and select a time point when a row recording will be done. The point can be either on onset or offset of the parent event.

8) The element is now ready. Run your experiment and then check the data report for the result.

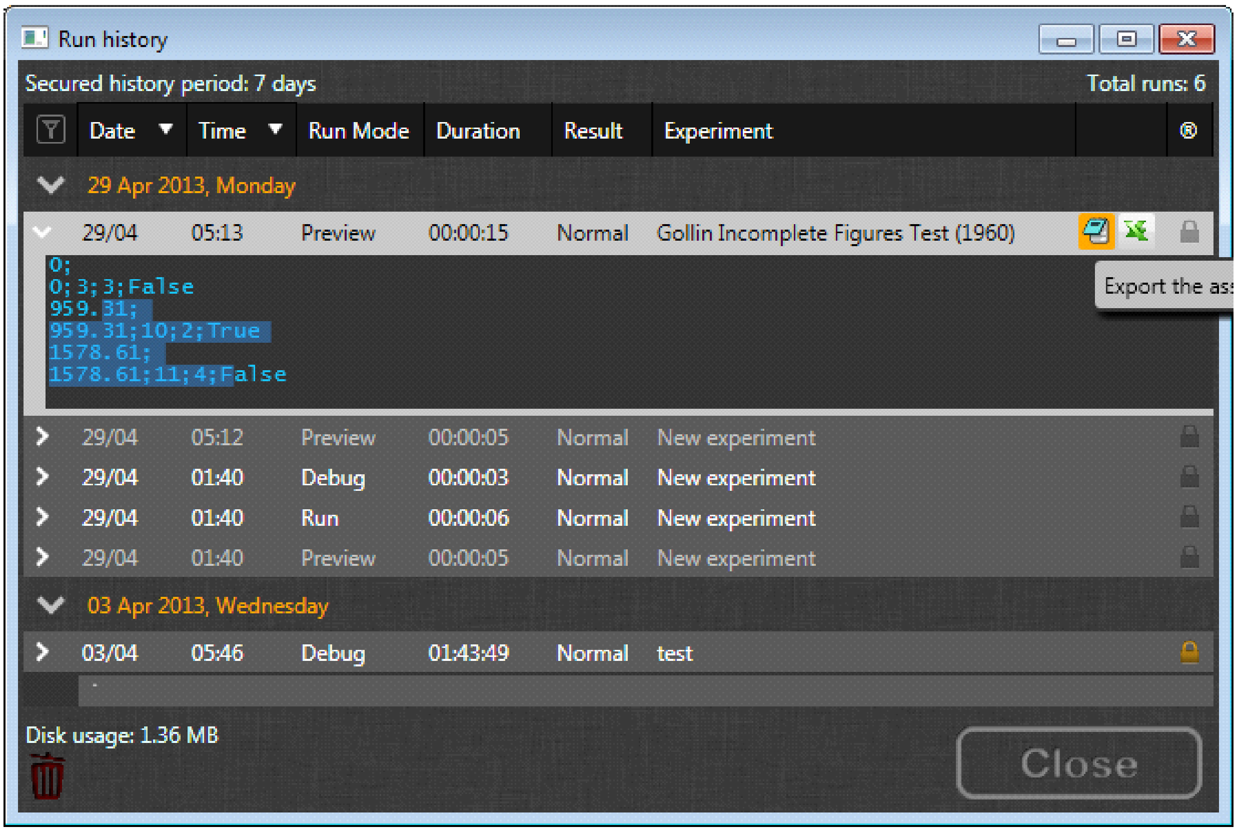
Exporting Data Reports

After an experiment run is complete, you can review the collected data report in EventIDE and export it either to a text/cvs file or to Excel. Exporting to Excel is only possible if it is installed on the computer (see the data report commands in the Run ribbon tab, as on the screenshot below). While sending a data report to Excel, EventIDE detects separator characters in the text and automatically splits data into Excel columns. In the properties of the experiment object you can define which separator character will be used. The default choice are the semicolon, comma and tab characters.



Runs History

EventIDE tracks all experiment runs and maintains the associated data reports, but only for a certain period of time (for 7 days by default). This limit is set to preserve disk space. The stored data reports can be found at any time in the Runs History viewer that can be opened via the Run ribbon tab:


Run history

Secured history period: 7 days — Total runs: 6

Date	Time	Run Mode	Duration	Result	Experiment
29 Apr 2013, Monday					
29/04	05:13	Preview	00:00:15	Normal	Gollin Incomplete Figures Test (1960)
29/04	05:12	Preview	00:00:05	Normal	New experiment
29/04	01:40	Debug	00:00:03	Normal	New experiment
29/04	01:40	Run	00:00:06	Normal	New experiment
29/04	01:40	Preview	00:00:05	Normal	New experiment
03 Apr 2013, Wednesday					
03/04	05:46	Debug	01:43:49	Normal	test

```
0;
0;3;3;False
959.31;
959.31;10;2;True
1578.61;
1578.61;11;4;False
```

Export the as

Disk usage: 1.36 MB

If you want to keep a data report over an extended period of time, you can either lock the related record in the history or increase the secure period duration in the application preferences. You may set the period to 'zero' for all reports to be stored permanently.

9. Running Experiment

To run the experiment, navigate to the Run ribbon tab and press the Run button. The program will automatically:

1) Compile your code.

2) Merge it with the designed content.

3) Allocate hardware/memory resources and processor load.

4) Start an experiment flow.

Running Modes

Experiments in EventIDE can be run in several modes:

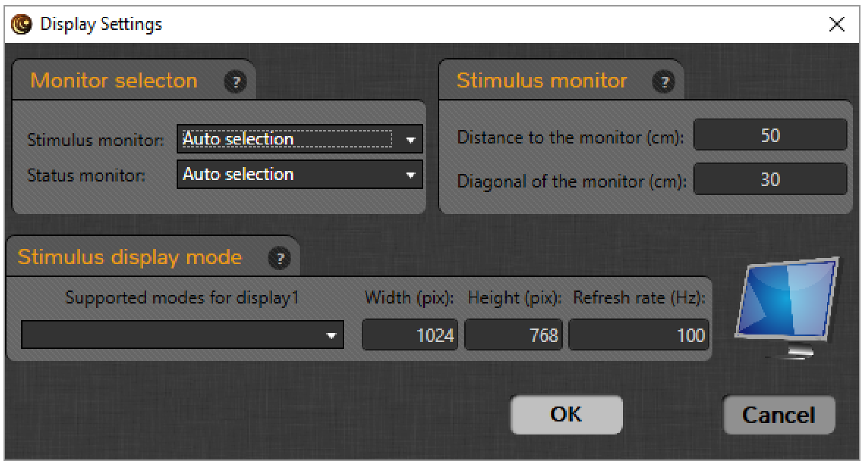
- **Full screen mode:** the program runs the stimulus presentation full-screen and takes an exclusive control over graphics input and processor. This mode provides the most precise timing, accurate stimulus presentation and the highest execution speed.
- **Preview mode:** the stimulus display is shown in a floating window. The startup process is fast and therefore this mode is recommended for quick tests of stimulus presentation and the event flow logic.

- **Debug mode:** the program shows an extra floating window with debugging tools. This mode is recommended for code debugging and monitoring snippets and variables in realtime.
- **Overlay mode:** use this mode to run EventIDE experiment in a transparent overlay layer over Windows Desktop and other applications. In this mode you can enable data recording, e.g. eye-tracking, with any third-party application.
- **GUI mode:** use this mode for tasks with interactive GUI elements, e.g. Web Browser. The mode makes the GUI elements more responsive, but does not guarantee accurate timing.

Graphics Mode and Status Screen

EventIDE allows you to run experiments in any of 32-bits graphics mode that your PC can support.

In order to choose the graphics mode for the stimulus display, navigate to Experiment ribbon tab and click Graphics Mode to open the Display Setting dialog. On multi-monitor setups you may also have to select the monitor for the stimulus presentation.



Stimulus monitor: Auto selection
Status monitor: Auto selection
Distance to the monitor (cm): 50
Diagonal of the monitor (cm): 30

Supported modes for display1 | Width (pix): 1024 | Height (pix): 768 | Refresh rate (Hz): 100

If you program your own status screen and would like to use it in the real run mode, you need a second monitor connected to the PC. Make sure that the Dual monitor setup in Windows is set to the 'extended desktop' option. If this option is not present in the system, try to update drivers for the graphics card.

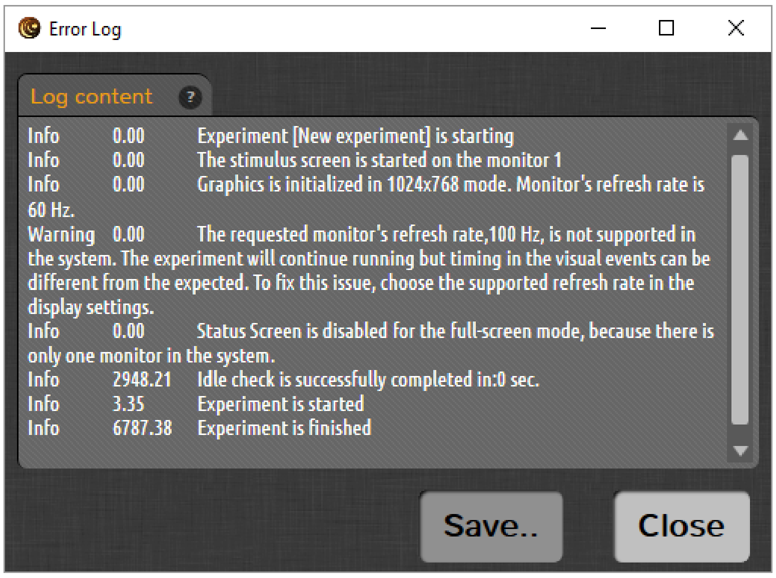
Run Results

Whether an experiment run in EventIDE finishes normally or gets aborted, run results are always obtained, including:

a) The associated data report (described in the previous section).

b) Error Log.

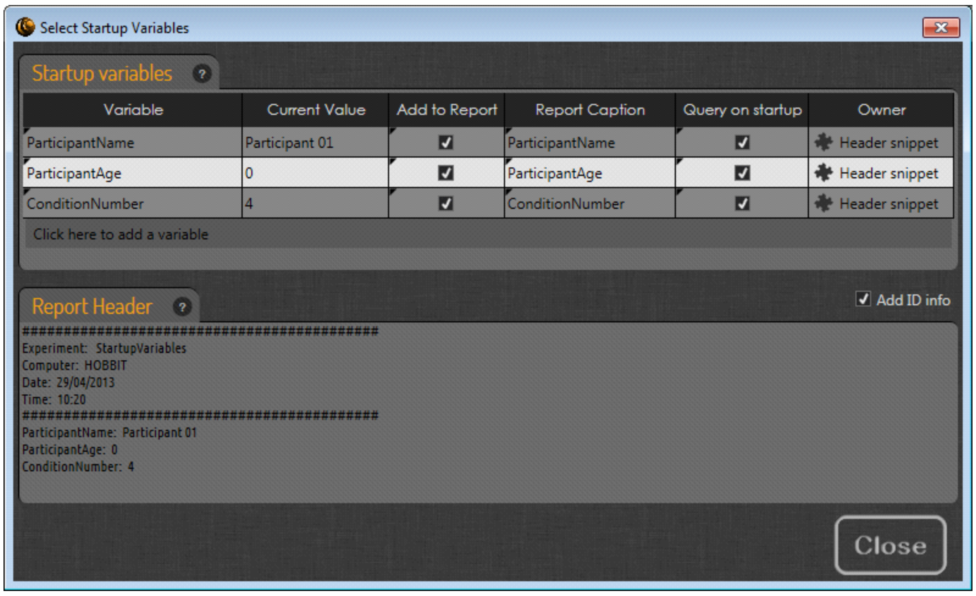
The Error Log contains a list of internal runtime messages: hardware initialization status, etc. When an experiment is aborted by an error, or something does not work as expected, you may inspect the chronicle for the error details. The latest Error Log can be opened via the Run ribbon tab.



```
Info      0.00     Experiment [New experiment] is starting
Info      0.00     The stimulus screen is started on the monitor 1
Info      0.00     Graphics is initialized in 1024x768 mode. Monitor's refresh rate is
60 Hz.
Warning   0.00     The requested monitor's refresh rate,100 Hz, is not supported in
the system. The experiment will continue running but timing in the visual events can be
different from the expected. To fix this issue, choose the supported refresh rate in the
display settings.
Info      0.00     Status Screen is disabled for the full-screen mode, because there is
only one monitor in the system.
Info      2948.21  Idle check is successfully completed in:0 sec.
Info      3.35     Experiment is started
Info      6787.38  Experiment is finished
```

Startup Variables

If you need to change some variables or parameters each time when an experiment is started, then you can create a list of the startup variables. To do so, open the startup variables window via the Run ribbon tab:



Select Startup Variables

Startup variables

Variable	Current Value	Add to Report	Report Caption	Query on startup	Owner
ParticipantName	Participant 01	✓	ParticipantName	✓	Header snippet
ParticipantAge	0	✓	ParticipantAge	✓	Header snippet
ConditionNumber	4	✓	ConditionNumber	✓	Header snippet

Click here to add a variable

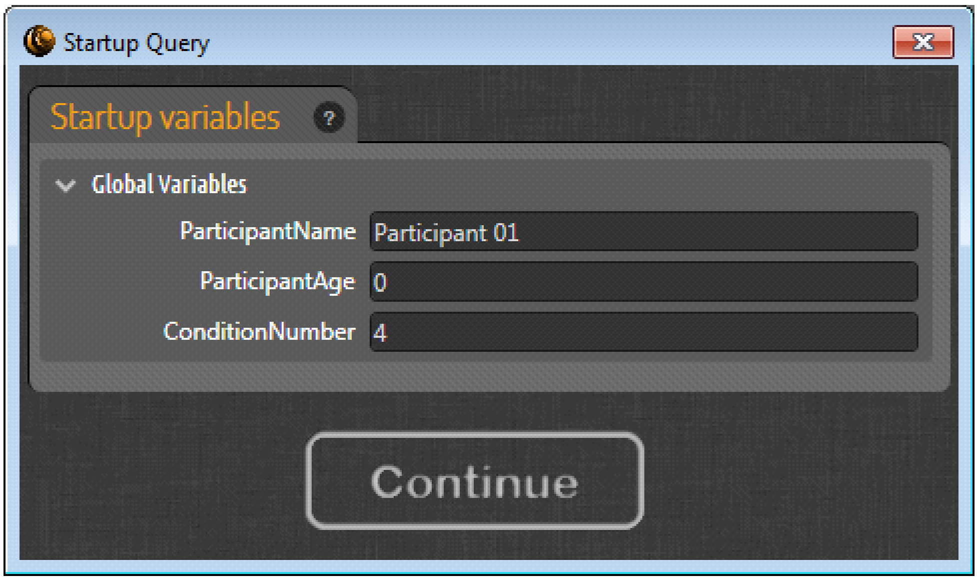
Report Header

[✓] Add ID info

```
############################################
Experiment:  StartupVariables
Computer:  HOBBIT
Date:  29/04/2013
Time:  10:20
############################################
ParticipantName:  Participant 01
ParticipantAge:  0
ConditionNumber:  4
```

Select any of the available proxy of global variables from the list. In the example above three variables declared in the Header snippet are added: Participant Name, Participant Age and Condition Number. For each added variable you can choose whether it will be queried on the startup and whether it will be added at the top of the data report. A preview of the data report is shown at the bottom of the window.

If the 'Query on startup' option is checked in the Run ribbon tab, the query dialog will be shown, when you start an experiment:



Startup Query

Startup variables

Global Variables

ParticipantName	Participant 01
ParticipantAge	0
ConditionNumber	4

The values entered in this dialog will be only applied in the current experiment run.

File Associations, Multiple EventIDE Instances and Player Mode

After EventIDE is run for the first time, it creates a file association with the '.eve' extension. Now, if you double click on any of the .eve files in Windows Explorer, it will be loaded into a new instance of EventIDE.

You can start as many instances of the EventIDE application as necessary, but only one instance can run an experiment at any one time.

When you need to run an experiment without opening the EventIDE designer, you can open the application in the 'standalone player' mode. The player mode is activated when EventIDE is run with two command line arguments: /p and a file name for the experiment, for example:

```
C:\Research\EventIDE\EventIDE.exe /p
c:\Research\Experiments\Billiant_experiment_47.eve
```

You can add this line to a Windows batch file (and run it afterwards) or create a dedicated desktop shortcut with the command line arguments appended in the target line.