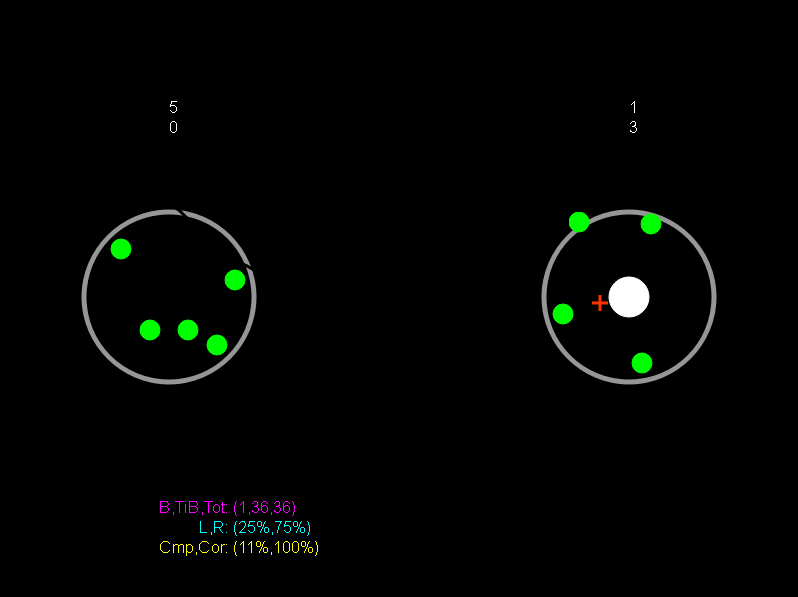
Picto

Lee Lab's Behavioral Stimulus and Data Acquisition Software



*Developed by: Trevor Stavropoulos, Joey Schnurr, Mark Hammond and Matt Gay*

*2009-2015*

Contents

[Overview 4](#_Toc378541868)

[Director 4](#_Toc378541869)

[Proxy 5](#_Toc378541870)

[Server 5](#_Toc378541871)

[Workstation 6](#_Toc378541872)

[Embedded Front Panel 7](#_Toc378541873)

[Installation 7](#_Toc378541874)

[General Instructions 7](#_Toc378541875)

[Configuration 8](#_Toc378541876)

[Server 8](#_Toc378541877)

[Director 8](#_Toc378541878)

[Proxy 8](#_Toc378541879)

[Experimental Design 9](#_Toc378541880)

[Development 9](#_Toc378541881)

[State Machines 9](#_Toc378541882)

[The State Machine Editor 10](#_Toc378541883)

[A Sample Design 11](#_Toc378541884)

[Design Elements 17](#_Toc378541885)

[Editor Tools 20](#_Toc378541886)

[Testing 22](#_Toc378541887)

[Analysis Design 25](#_Toc378541888)

[Development 25](#_Toc378541889)

[A Sample Analysis 25](#_Toc378541890)

[Analysis Elements 29](#_Toc378541891)

[Analysis Tools 30](#_Toc378541892)

[Testing 31](#_Toc378541893)

[Running an Experiment 33](#_Toc378541894)

[Behavioral Sessions 33](#_Toc378541895)

[Pictobox 37](#_Toc378541896)

[Behavioral / Neural Sessions 38](#_Toc378541897)

[Session Playback and Analysis 39](#_Toc378541898)

[Playback 40](#_Toc378541899)

[Playback Speed Slider 40](#_Toc378541900)

[Time Slider 40](#_Toc378541901)

[Play, Pause and Stop 40](#_Toc378541902)

[Operator/Test Subject Dropdown 41](#_Toc378541903)

[Session Recording System 41](#_Toc378541904)

[Analysis 41](#_Toc378541905)

[Upgrading Software 43](#_Toc378541906)

[Troubleshooting 43](#_Toc378541907)

[Server Connectivity Issues 43](#_Toc378541908)

[Large Session Analysis Problems 44](#_Toc378541909)

[Jumpy Eye Position Signal 44](#_Toc378541910)

[Other Issues 44](#_Toc378541911)

# Overview

The Picto suite of experimental control applications is utilized by researchers to perform neurobiological experiments in the study of decision making.  The software provides a simple and intuitive development environment for designing experiments and defining data extraction and analysis algorithms. It abstracts the complexity of coordinating behavioral and neural data collection: obtaining precise timings, dealing with the non-real time nature of the computer operating system, synchronizing the presentation of stimuli with the timing of the visual display, providing rewards, and the like. The result is that researchers are free to focus on research without worrying about the technical details of managing a complex experimental framework.

## Director

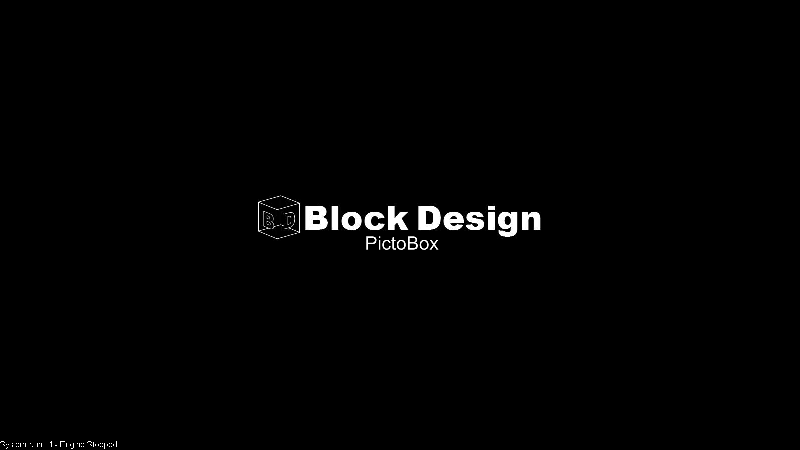


Figure 1 - Picto Director splash screen

The Director application runs the experimental "game" that the test subject "plays." This application typically takes in user input from an eye tracker, but a mouse input, joystick, or any other device that can provide analog x/y data can be used as well. Inputs are monitored to control the flow of the "game" and all user input, reward, and experimental state data are sent over the network to the Server component. A hardware cable connects the director to a neural data acquisition system. This cable is used to send periodic alignment codes so that the neural and behavioral time streams can be aligned on the Server.

## Proxy

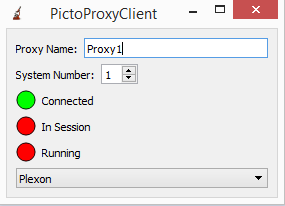


Figure 2 - Picto Neural Proxy

The Proxy application runs on a neural data acquisition computer (currently supported models are Plexon and TDT). It collects data from the acquisition system, including neural spikes, local field potential data and alignment codes coming in from the Director, repackages the data and sends it over the network to the Picto Server component.

## Server

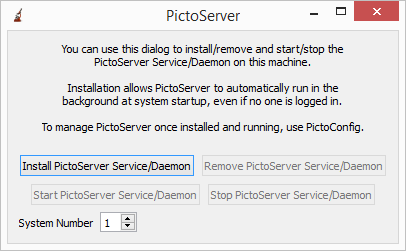


Figure 3 - Picto Server

The Server service is the main hub of the Picto application suite. The Server takes in data from the Proxy and Director, aligns the time streams from the two applications based on their alignment code data and saves the data to disk in a Session file. It also makes Director/Proxy data available in real time for monitoring from the Workstation application. The Server acts as the glue that binds the Workstation to the Proxy and Director. Whenever the Workstation sends a command to be handled on either the Proxy or the Director, the command is sent to the Server. The Server takes care of forwarding the command to the appropriate component and informs the Workstation when and if the component complies with the command directive. While the Server is doing a great deal of work behind the scenes, however, on a day to day basis the user’s main interaction with the Server is as the network drive on which Picto session files are saved.

## Workstation

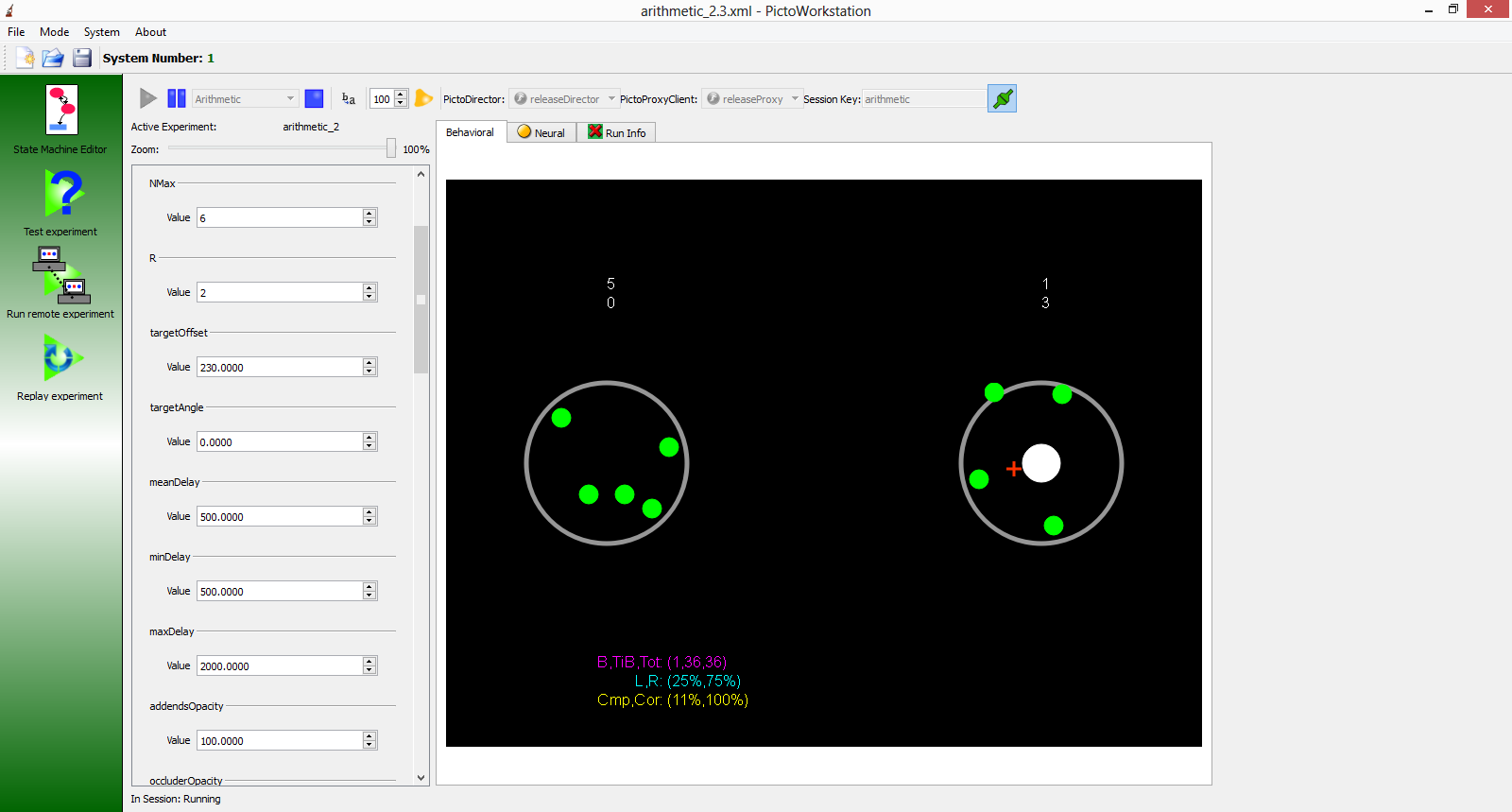


Figure 4 - Picto Workstation - Remote Viewer

The Workstation component is the Researcher's window into the Picto system. It includes a Remote Viewer which Researchers use to start/stop Experimental sessions, control them and monitor their activity. It includes a State Machine Editor, used to design the Experiment that runs on the Director and the Analysis that is used to extract data from session files. It includes a Testing Viewer for locally testing and debugging Experimental designs and analyses. Lastly, the Workstation includes a Playback Viewer which is used to playback session files, record session activity to a video file, and run session analyses.

## Embedded Front Panel



Figure 5 - Embedded Front Panel - Running in Pictobox LCD Display

When the Director application is running on a Pictobox, the Embedded Front Panel application controls its LCD display. Embedded Front Panel communicates with the Director and presents general information like the Director’s name and connection status. The application also allows the user to change various settings like default reward duration by using the rotary dial.

# Installation

While we have not yet created a standalone installer for Picto, the process is fairly straightforward. Whenever we install any Picto application, we just go ahead and install all of them, so the installation procedure is mostly the same across all of the different components of the system. We will go through this “general” installation process in [General Instructions](#_General_Instructions), then go through specific configuration issues for different components in the [Configuration](#_Configuration) section.

## General Instructions

Installing Picto is usually as easy as copying the contents of a Picto bin folder to the target computer’s C:\projects\picto\bin folder. In Lee Lab, Picto bin folders are located in \\cog\it\Software\Picto\Versions\Version?\_?\_?\Bin where Version?\_?\_? is a directory that indicates the version of the software.  The applications require various libraries that come with Visual Studio, so if that is not installed on the target computer, the vcredist\_x86.exe installer will have to be run as well. In Lee Lab, that installer can be found in \\cog\it\Software\Picto\Installation\Dependences.

If you are installing the Workstation application, that’s all there is to it.  Start the application by running C:\projects\picto\bin\PictoWorkstation.exe. When you first run this or any of the Picto applications your firewall will request permission to allow it to send network data.  This permission must be granted in order for the Picto application to communicate with the rest of the Picto system on the network.

For installation of other applications there are a few additional configuration steps that will be detailed below.

## Configuration

### Server

If you are installing a Picto Server, create a shortcut to the PictoServer.exe application in the bin directory.  In the shortcut properties, go to the "Compatibility" tab.  Make sure that "Run this program as an administrator" is checked.  Go back to the "Shortcut" tab.  In the Target field, add " -interactive" after the executable path.  Run the shortcut, and wait for windows to ask you to allow the application through the firewall.  Allow it.  Go back to the shortcut properties and change "-interactive" to "-gui".  Run the application again and you will get a dialog.  Select a System Number that will be unique for your Director / Proxy / Workstation setup on the local network. If this number is not unique, you will have problems with cross talk from other Picto systems on the network. Next, press “Install PictoServer Service/Daemon”. When that is done, press "Start PictoServer Service/Daemon". That completes installation of the PictoServer application as a service on your computer. Whenever your computer starts up, the PictoServer service will run automatically in the background.

### Director

After installing the Picto files, DirectX and the current Picto version of the NIDAQmx drivers will need to be installed.  In Lee Lab, these installers are located in:

\\cog\it\Software\Picto\Picto\Installation\Dependences\Director\dxwebsetup.exe  
and  
\\cog\it\Software\Picto\Picto\Installation\Dependences\Director\NIDAQ Drives and Tools\NIDAQ???.exe  
The installers should also be available online from the National Instruments and Microsoft DirectX websites.

Now that all of the software and dependencies are installed, a .bat script should be set up to start the Picto Director application and placed in the Start menu’s Startup folder so that it will start the Director application automatically when the Director computer turns on. Command line options for the script will vary depending on your experiment requirements; however, a typical script for a Director running on a Pictobox will look something like the following:

*start /d C:\picto\bin\ PictoDirector.exe -name Pictobox1 -systemNumber 1 -pictobox -xChan 0 -yChan 1 -xDiamChan 2 -yDiamChan 3 -positionPeriod 2 -diamPeriod 4*

Also make sure that the system resolution is set to 800x600 and don’t forget to allow the application through the firewall when you first run it. This can sometimes be difficult with the Director since it takes up the full screen. Pressing Alt-Tab should give you a temporary view of the desktop though. You can also always press Escape to end the Director application.

### Proxy

There is not much configuration required when running the Proxy application. Just make sure to enter an appropriate name in the Proxy Name field, enter the correct System Number and select the correct neural data acquisition device type from the drop down menu. Also, don’t forget to give the application access through the firewall.

# Experimental Design

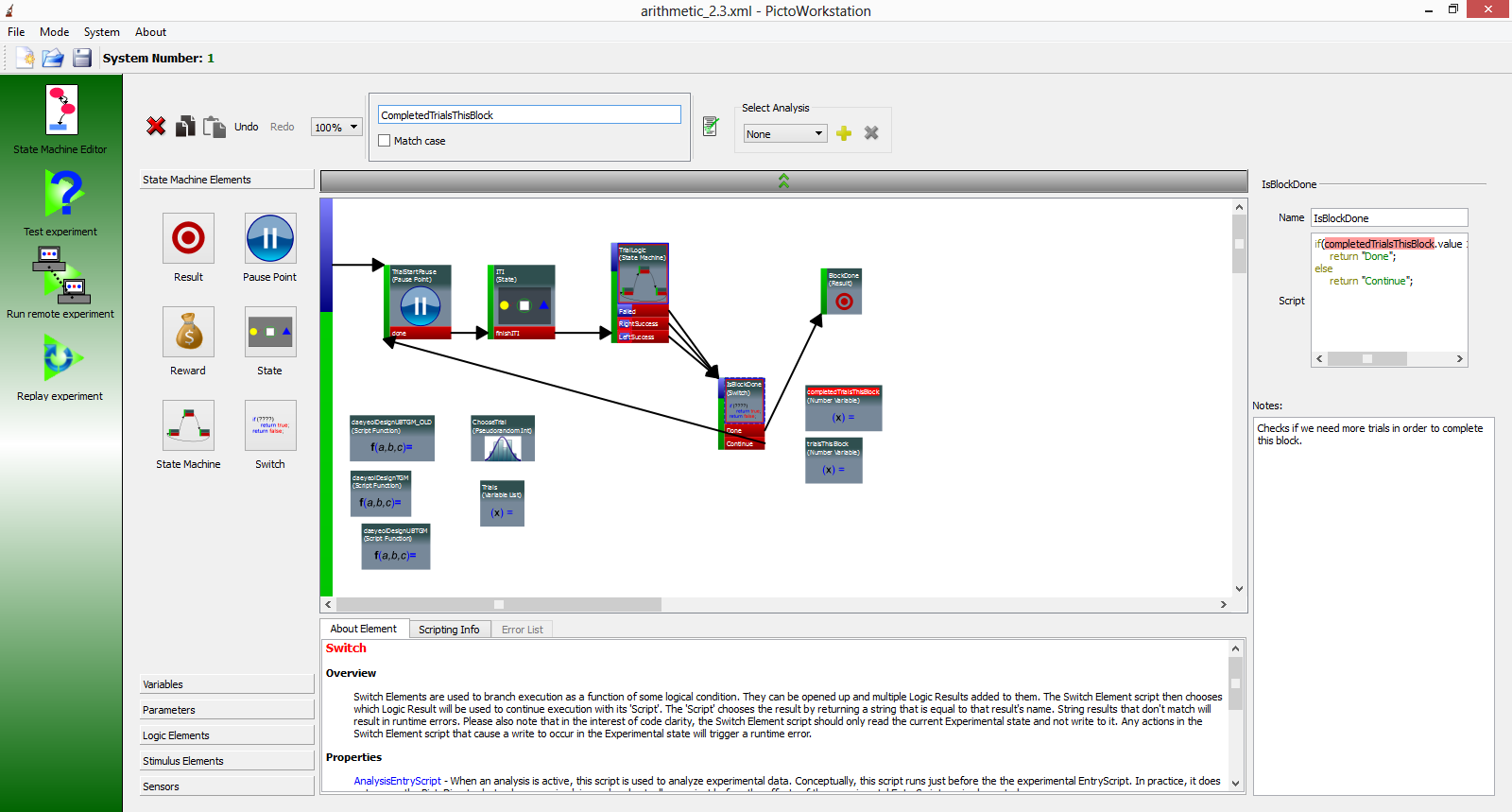


Figure 6 - State Machine Editor

The Picto State Machine Editor is a "drag and drop" development environment. Execution flow is defined by simply drawing the execution flow chart on screen. Visual elements defining the subject’s display, state variables, and standard control logic components are "dragged and dropped" into the workspace as well. More detailed custom execution logic is added to the flow chart where needed in simple javascript. The result is a clear, robust, self-documenting experimental design.

## Development

### State Machines

In order to understand the Picto design model, it is important to be aware of the concept of the state machine. A state machine is a standard way of designing simple applications by breaking up their operating conditions into multiple states that the system can transition into depending on particular events. The typical model is a vending machine. The vending machine contains several operating states. For our purposes, we will say that it has a WAIT state, a SELECT state, a VEND state, and a CHANGE state. The machine starts in the WAIT state. It simply sits and waits for a person to show up and put money in. Once an amount of money meeting or exceeding the cost of the products has been entered, a transition is triggered which moves the program into a SELECT state wherein the machine waits for the person to make a drink selection. When the person pushes a button, another transitions is triggered moving the program into the VEND state wherein the selected drink is released from the machine. When that process completes, the program goes to a CHANGE state wherein it releases the change that the user is owed. At this point the sale is complete and a final transition is triggered that moves the program back to the WAIT state so that the cycle can start again when the next person arrives. This simple programming model is the basis for all experimental designs in Picto.

### The State Machine Editor

The Picto State Machine Editor makes building up simple state machine structures quick and easy. States are laid out on a canvas. Each one is defined with custom Results and each of these Results can be wired to a new State with a Transition arrow. As an example, if we were to build the vending machine state machine from the last section, it would look something like this:

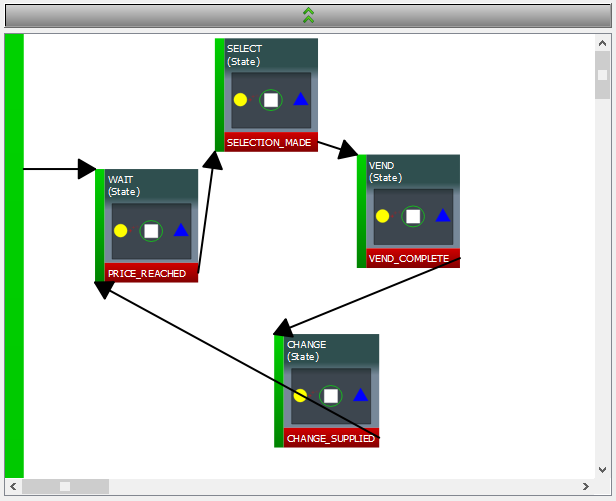


Figure 7 - Vending machine program as Picto state machine

Of course, things don’t always work perfectly and there might be some other conditions that we would like to handle. We could fill out the vending machine state machine with more options, like this:

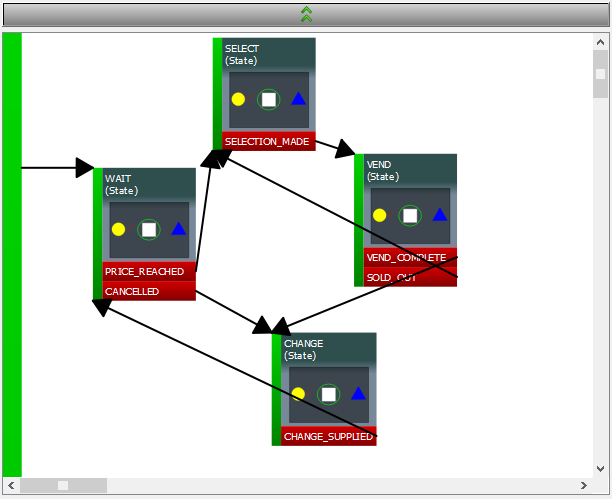


Figure 8 - More detailed vending machine design

Now we have additional options. If the person changes their mind and decides that they don’t want a drink after all, we can transition directly to the CHANGE state to get them their money back. If the person selects a drink, we can transition back to the SELECT state to give them another chance to select a drink. Clearly there are more transitions that we would add here to handle a real world case, and in fact state machines can become quite complicated. In code, a program like this starts looking complicated fairly quickly, particularly to people that don’t spend their time looking at code all day. One of the reasons that we decided to use a graphical editor for Picto was to make complex code as clear and easy to read as possible for people who are not programmers: the people that are actually designing the experiments that Picto will run.

### A Sample Design

Now let’s look at an example of a state machine design that we could actually use in a Picto experiment. Here we will create a simple Trial based Task where each Trial contains a fixation stage and a stimulus target after the fixation. We start by opening the editor and using File>New to create a new experiment. The experiment will start with one “automatic” task, the eye calibration task used to set the X,Y Offset and Gain values to calibrate the eye tracking signal for the current session. We can create a new Task by clicking on the Task button in the Element Library on the left hand side of the screen, then clicking somewhere in the central canvas. Once we click, the Task appears on the canvas and is automatically selected so that we see its properties in the frame on the top right side of the canvas. Let’s change its name to FixationTask, noticing that the name is updated on the Task diagram automatically.

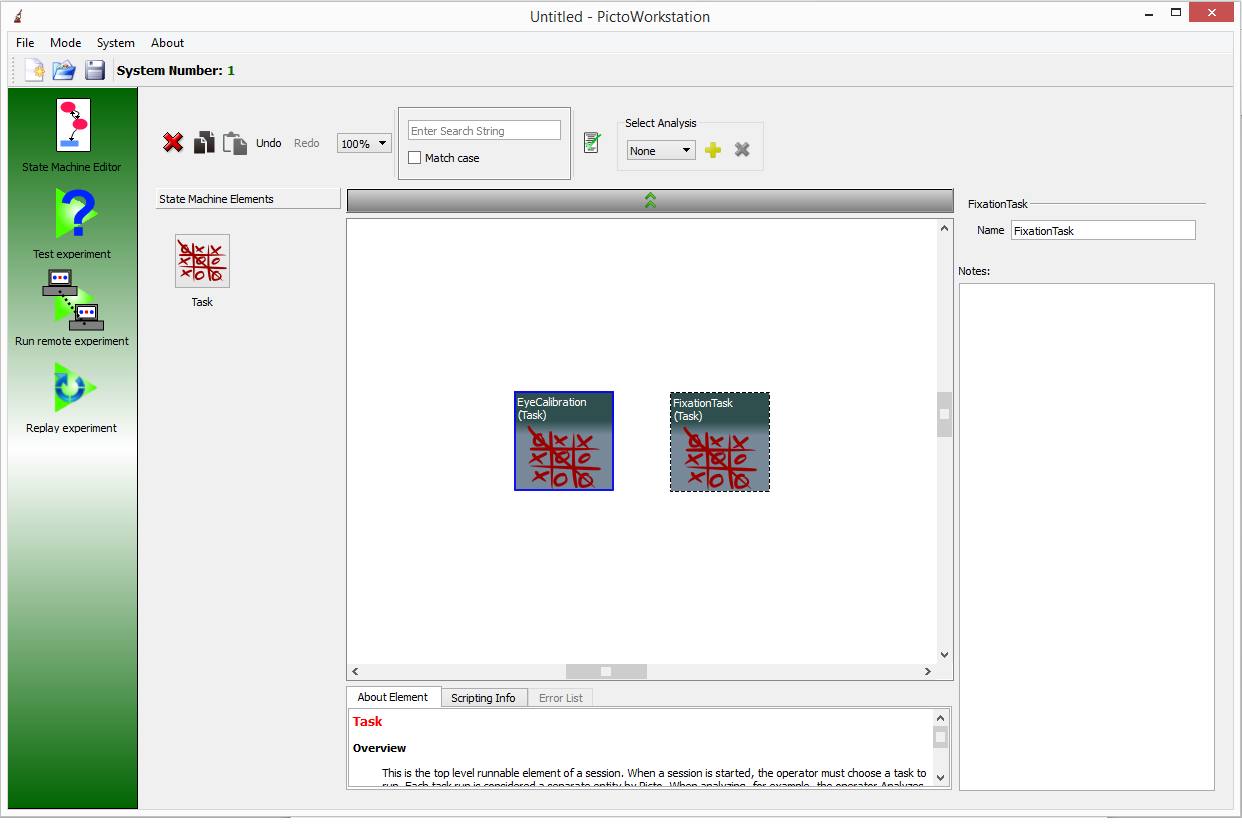


Figure 9 - Creating a new Task

We can now double click on our new Task to open it up and start designing its contents. Inside you will find a single State Machine element. This will be the top level of our State Machine and we can name it accordingly, “Top”. If we double click on “Top” we open an empty canvas containing only a green vertical bar on the left hand side of the screen. This bar is called the “Start Bar” and as the name describes, this is where control flow begins in the state machine. Click on the State Machine Elements tab in the Element Library and select State Machine. Drop that element in the canvas. We can now move the mouse over the green “Start Bar”, click and create a transition from the beginning of this state machine to the beginning of its child state machine. Lets name this state machine Trial.

This is a good time to discuss the multi-layer character of the Picto state machine. While our vending machine example only had a single level, normal Picto state machines operate on multiple levels. Each state machine element can contain multiple other state machine elements. As soon as control flow reaches a state machine element, control enters inside that element and starts moving through its internal state machine. This allows us to focus on control flow for each individual level without worrying about the details of each element of that level. In this case, we are creating a Trial state machine. Next we will add a “Done” result to the trial, and we will set up logic to run the trial 3 times before ending the experiment. We will be able to focus entirely on the logic at the current level before working on any of the details of the trial itself.

Let’s go ahead and do this. Double click on the Trial state machine, select a “Result” from the library, drop it into the Trial canvas and call it “Done”. Next click the large arrow button above the canvas to go back up one level in the State Machine and you will notice that a red “Done” result was added to the bottom of our Trial element like this:

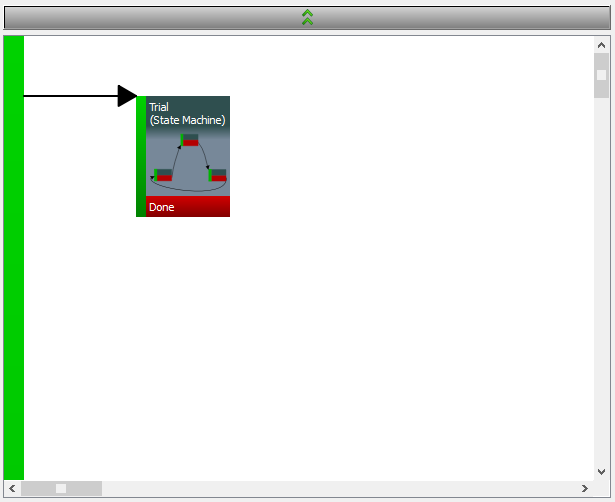


Figure 10 - Trial element with Done result

Now let’s add a result to this level and call it “TaskEnded”. Next, find the Switch element and place it in the canvas. The switch element allows us to define a javascript script that returns a string value. If you are unfamiliar with javascript, there are hundreds of tutorials online and we will leave it to you to go through one. You only need a minimal level of knowledge to be able to work with Picto. We can now create multiple results inside the switch. Whichever result’s name is returned from the script is the result that is triggered to continue control flow. Name the switch “isTaskOver”, then add two results inside, “Yes” and “No”. You’ll notice that there is no Start Bar on the canvas inside the switch element. This is because control flow in the switch element is defined by its script result so transitions wouldn’t make sense inside it. When you have the switch results ready, wire up “Yes” to TaskEnded, and “No” back to “Trial”. “Done” on “Trial” should go to IsTaskDone. In order to check if the task is actually done, we are going to need to keep track of the trial number. Find the “Number Variable” under Variables in the Element Library, add it to the canvas and call it “trialNum”. Your canvas will now look something like this:

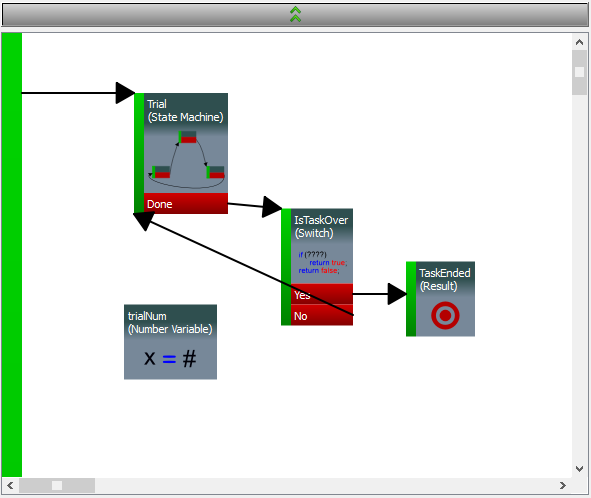


Figure 11 - Trial counting control flow

Now, click on the trialNum variable and look at the “About Element” tab at the bottom of the canvas. You can slide it up into the canvas by grabbing its border to see more. This tab contains context sensitive documentation. It tells you what the Number Variable does. What its properties are for, and how you can read and write its data with scripts. The Number Variable has a “value” script property. Let’s use that now in our IsTaskOver Switch. Enter the following code into the IsTaskOver Switch script:

if(trialNum.value >= 5)

return "Yes";

return "No";

Now our switch will choose the “No” path until the Trial number reaches 5 at which point the Task will end. But first we need to actually increment the Trial number somewhere. Click on the green bar of the Trial element and you will see two script properties in the property browser: “EntryScript” and “ExitScript”. EntryScripts are called as soon as control flow enters a script and exit scripts are called as soon as control flow exits it (after result scripts are called). Enter:

trialNum.value++;

into the Trial ExitScript. Notice that the bottom of Trial’s green bar was highlighted blue when you entered text into its script. This lets you know that there’s something there when you are reading through the code. If you hover your mouse over the blue highlight you will notice that your script appears in a tooltip as well.

We have now completed the logic for the top level of the state machine, so let’s work on the Trial level. Go into Trial, find the State element and add three of them to the canvas. Name them: “ITI”, “Fixation”, and “Stimulus”. Add “Done” results to each of these states plus a second “Failed” result for the Fixation state. Note that Picto supports copy/paste actions, so you can use these to speed things up. Now wire the Start Bar to ITI. Wire ITI>Done to Fixation. Wire Fixation>Done to Stimulus. Wire Stimulus>Done to Done, and wire Fixation>Failed back to ITI.

This is a good point to discuss timing in Picto. In Picto, unless the Task is paused (which only is allowed at special elements called PausePoints), there is only one place where experimental time actually passes and that is the State element. As far as the designer is concerned, all of the other transitions, switches, results, etc take no time at all. Inside a state, though, time passes until a “Control Element” decides that the state is done. Multiple control elements can be added to the same state. Each one’s logic is checked every frame to see if the frame is done. If none of them trigger any of their results, control remains in the State until the next frame.

For the ITI, we will use a Stopwatch controller to wait for a simple timeout. Open the ITI state and add a Stopwatch Controller. Call it Timeout, and set it to timeout in 2 seconds. Now wire Timeout’s Success result to Done and go back up to the Trial canvas.

At this point we want to add a fixation graphic that the user can fixate on. We will use a simple white circle. Find the “Circle” under Stimulus Elements and add it to the canvas. Call the circle fixateGraphic, set its position to 400,300. Set its color to white. Deselect visible and enter 30 for radius.

At this point we will need to discuss Picto scope. In Picto, every element that you add to the canvas is accessible to its parent, and all of its parents descendants. In general, this means that any of those elements can access and change the elements values with scripts. For visual elements, this also means that if the graphic is set visible, it will appear on screen if control flow is in any states for which the element is in scope. If we were to leave the circle as visible, it would have appeared during the ITI, so we initialize it to not be visible. Let’s set it visible at the beginning of the Fixation state though. Add:

fixateGraphic.visible = true;

to the EntryScript of Fixation.

The next question should be, “What about when we reach trial two? Won’t the graphic be visible during the ITI?” Actually, part of Picto’s scoping system is that each property’s “run values” are set to the values that appear in the property browser (“init values”) each time control flow reaches their parent, so the circle will be set invisible again as soon as control flow re-enters the Trial element.

Now we want to set up a fixation window for the subject to look at when the fixateGraphic is visible. Grab a “Circle Target” from the Logic Elements section, add it to the canvas, and call it fixateTarget. Place it at 400,300 and give it a 60 pixel radius. Notice that the Circle Target is marked visible by default but also OperatorView is selected and not SubjectView. In general, OperatorView means that you, the researcher will see the graphic and Subject View means the test subject will see it. This is all qualified by the Visible property. So the subject will not see the Circle Target but you will. The other special case with Circle and Rectangle Targets is that they are never visible to anyone unless a Control Element is actively checking to see if the subject is looking at them.

While we’re adding visual elements, let’s just add the stimulus. Add a Box graphic to the canvas. Call it box, put it at 600,300, make it red, give it dimensions of 50x50 and set it invisible by default. Now add:

box.visible = true;

to the Stimulus EntryScript and we will have finished this level of the state machine. The Trial canvas should now look like this:

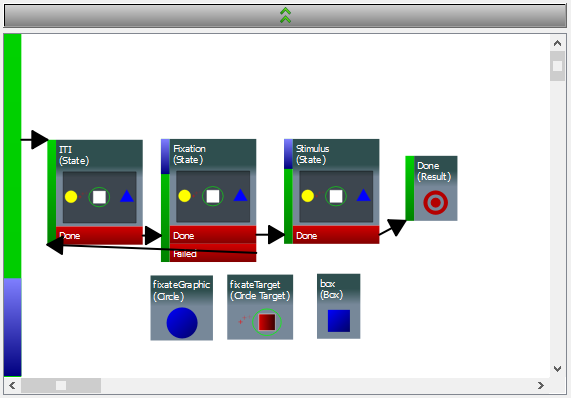


Figure 12 - Trial canvas

At this point, all we have left to do is fill out the Fixation and Stimulus states. Lets keep the Stimulus state simple and not track any fixation. Copy the 2 second timer from ITI, paste it in Stimulus and wire its result to Done. Next open the Fixation state, find the Target Controller, add it to the canvas and call it “Fixation Controller”. Type fixateTarget into the “ControlTarget” field. This sets the fixateTarget region as the area where the Target Controller wants the subject to fixate. All of the other properties can be left with their default values. Wire the Success result to Done and everything else to Failed.

That completes our design. Press the green checkbox in the toolbar to make sure that all of your syntax is correct. This check is pretty good at finding structural problems in the experiment like missing transitions. It finds some script errors as well, but JavaScript in general is so flexible that it is possible to make a lot of mistakes that can only be caught at run time. If we have these types of errors, the debugger will pop up in testing and we will catch them then.

Before we jump into testing our experiment though, let’s go through a brief overview of the State Machine Editor design elements and tools that we have and have not used in our sample design.

### Design Elements

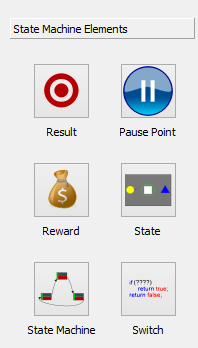


Figure 13 - State Machine Editor Element Library

In this section we will present a quick list of some of the most important elements in a Picto Experimental Design. This list is by no means exhaustive and descriptions are very broad. For complete documentation of the design elements, the About Element tab in the bottom-center of the State Machine Editor should be used. This tab includes a context dependent documentation frame. Whenever you select an Element in the canvas or press its button in the Element Library, the frame will be updated with detailed documentation of that element’s function, properties and scripting interface.

* State Machine – Each of these represents a new level of the State Machine. Each Task includes a single top level State Machine element by default. On lower levels within the design, State Machine elements can be used to encapsulate functionality and to define scoping and property initialization times.
* State – This is the place in the design where time passes. Everything else is simply defining transitions and logic, but rendering to the display actually occurs in the State. In addition to the scripts available on other elements of the state machine, the State has a FrameScript that is run just before each frame is rendered to the display. Control flow always stays inside a State until one of its ControlElements triggers a result.
* Result – Results are the starting points of all Transitions as well as the interface for moving “up” one level in the state machine. Control flow at each level of the state machine ends when it reaches a result. Since results contain no internal logic they only define EntryScripts not ExitScripts.
* Switch– A switch allows us to branch control flow over one of multiple transitions based on logical tests. Multiple results are defined inside the element and the Switch script returns a string with the name of the result that should be triggered. Note that a Switch script should only contain “read” not “write” logic. If the script inside a Switch causes any data in the design to change, a runtime error is triggered. Scripts that change design data should be in EntryScript, ExitScript and FrameScript areas.
* Reward– This provides a reward to the test subject. As far as the Director is concerned, this means setting an output TTL high for a duration defined in the element. In Lee Lab, this is used to provide a set quantity of juice. Multiple rewards can be provided by a single Reward element with a set reward period. No time passes while control flow passes through a reward element. It is provided in parallel with the rest of the experimental design. This means that the reward will be started during the course of the first State to follow the Reward element. By designing things this way, we allow experiments to alter elements on the display during the course of reward supply. Tokens representing rewards can be set to slowly disappear while each reward is being supplied, for example.
* Target Controller– A control element used in State to check whether the test subject is fixating on a particular control target or not.
* Choice Controller– A control element used in State to check whether the test subject is fixating on one of a number of defined control targets.
* Script Controller– A generalized control element used in State where a script can be defined to decide when a state will end. This works a lot like Switch. Results are defined inside, and the Controller doesn’t end until one of their names is returned as a string. Like in the Switch, the Script Controller’s script may not change any data in the design but should be read only. If it does change data, a runtime error will be triggered in the Test Viewer.
* Stopwatch Controller– A control element used to end a State when a timer runs out.
* Pause Point– When the experiment operator presses the pause button, the Task continues to play until control flow reaches a pause point. It resumes from that point when play is next pressed. This allows the design to control the point at which pauses occur so that they will not cause unexpected changes in timing during a trial.
* Number Variable– Stores a number for use over multiple elements’ scripts.
* String Variable– Stores a string for use over multiple elements’ scripts.
* Variable List– Stores a list for multiple numbers or strings to be used over multiple elements’ scripts.
* Variable Map– Stores a keyed lookup table for multiple numbers or strings to be used over multiple elements’ scripts.
* Integer Range– Stores an integer number for use over multiple elements’ scripts. Can be exposed as a parameter in the run time user interface to allow the operator to change design settings.
* Boolean– Stores a true false value for use over multiple elements’ scripts. Can be exposed as a parameter in the run time user interface to allow the operator to change design settings.
* Float Range– Stores an floating point number for use over multiple elements’ scripts. Can be exposed as a parameter in the run time user interface to allow the operator to change design settings.
* Circle Target– Defines a target region for use in checking for fixation with the Target Controller or Choice Controller.
* Script Function– Used to define a script function so that the same code can be re-used by multiple scripts in the design.
* Box– Defines a box graphic to appear on the test subject’s display.
* Circle – Defines a circle graphic to appear on the test subject’s display.
* Text– Defines a text area to appear on the display. Typically this is used to provide Task status information to the operator and is set invisible to the test subject.
* Token Tray– Defines a ring of “token” graphics to appear on the test subject’s display. These are useful for representing things like points, quantities, etc. Various values can be controlled for the tokens as a group or individually, size, color, shape, etc.
* Token Factory– A generalized Token Tray that allows scripts to control the individual token positions as well.
* Signal Value– Allows the value of a defined signal (ie. Position) to be retrieved directly from a sript.
* Timer– Used to measure time between different parts of the state machine.

### Editor Tools

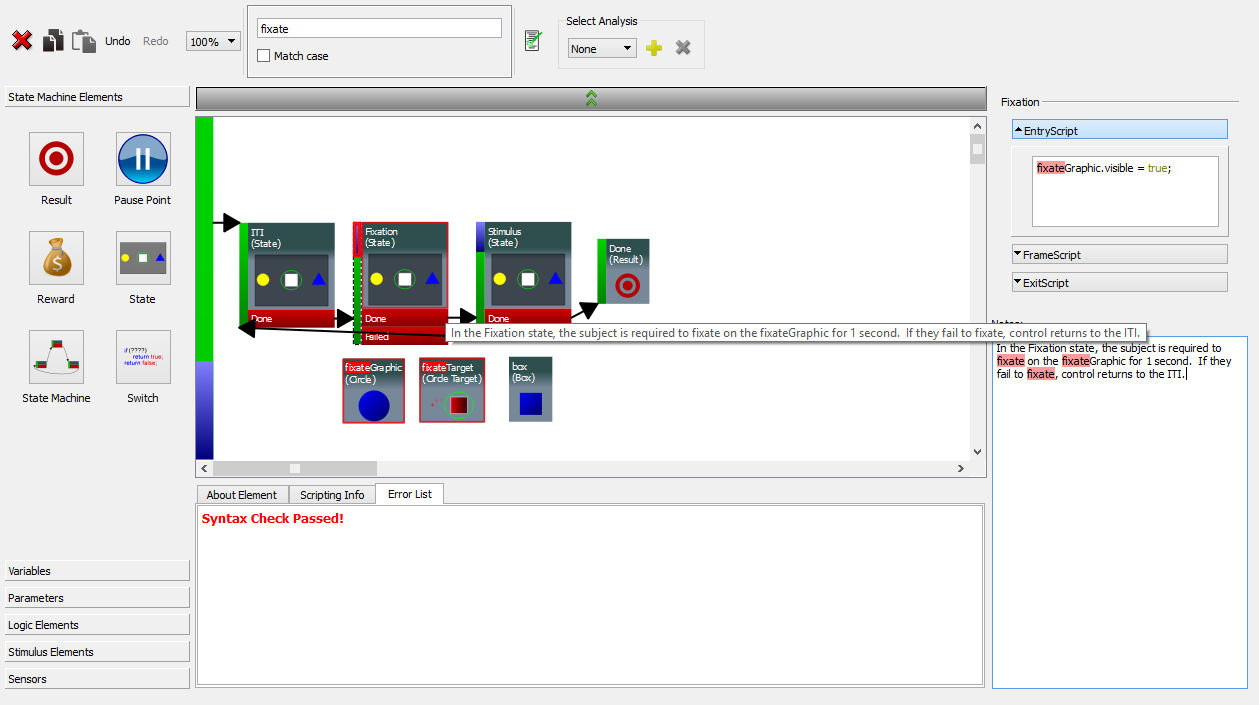


Figure 14 - State Machine Editor Tools

We have made every effort to keep the State Machine Editor user interface as clean as possible so as to not overload the user with unnecessary tools. A number of particularly useful tools for design and testing are included though and we go through them here.

Obviously, the most important elements of the user interface are the Element Library, central Canvas and Property frame. We have used these in the sample design and won’t go through them here. Another very important tool is the Notes box at the bottom right corner of the display. Every element that you select can have notes saved with it. These notes will be visible whenever anyone selects the element. The notes will also appear in a tooltip whenever the user’s mouse hovers over an element with notes.

The toolbar at the top of the editor has many standard tools: Delete, Copy, Paste, Undo, Redo, and Zoom. One of the most useful tools in the editor though is the Search Box. Any text typed into this box is highlighted throughout all parts of the window. If the search string appears in descendants of any element in the window, a red outline highlights that element. If any direct ancestor of the current canvas or its direct children contain the search string, the “up arrow button” is highlighted red. This becomes extremely useful in large projects. The next button is the “Syntax Checker” which is used to check design syntax before testing it. The Analysis controls to the right of that will be discussed in the [Analysis Tools](#_Analysis_Tools) section.

Another useful feature is actually simply a keyboard shortcut. If we hold down shift while rotating the mouse wheel, the canvas zooms in and out. Shift also turns the mouse into a “grabbing hand” that can be used to pull the canvas around instead of using the scroll bars.

We should not fail to mention Save. It is important to save your work frequently so as to not lose data. The design is saved in a file with a “.xml” extension at the path that you set in the “Save as” dialog. The Control-S shortcut can be used as well.

In case you forget to save your design, Picto includes an auto-save feature that allows you to restore a recent version of your work when you restart in the event of an application or system crash. If you end up restoring a file in this way, don’t forget to save it right away once it is opened. Note that Picto won’t let you restore a file if you don’t save it, close the window and ignore the automatic dialog that reminds you to save your work before closing. The auto-save feature is for “crash” situations only.

The tabbed area underneath the editor canvas includes three sections. The importance of the first of these, the About Element tab, cannot be overstated. This tab shows documentation about every type of element in Picto and brings it up automatically when that element type is selected on the canvas or clicked in the Element Library. The documentation there is not copied in this manual since it is simply much more accessible as an integrated part of the development environment. The Scripting Info tab includes the names, script properties, and script functions of elements that are in scope of the currently selected element if that element can include scripts. The Error List tab is used with the “syntax checker” button on the right side of the search box. When that button is pressed, the design syntax is checked for errors and those errors appear in the Error List tab.

Lastly, you will notice that some elements of your design include a thin blue outline around their central icon area as well as blue highlights in their start bars. The blue color indicates the presence of a script. On the start bar, the blue highlighted region shows which script is defined (top: entry, middle: frame, bottom: exit). If you hover the mouse over these highlights a tooltip will pop up with their script contents. The blue outline around the icon area indicates that there are scripts being used somewhere inside that element or its descendants. When designing a complex experiment, this is useful to focus in on the locations where logical checks are occurring and data is being changed.

## Testing

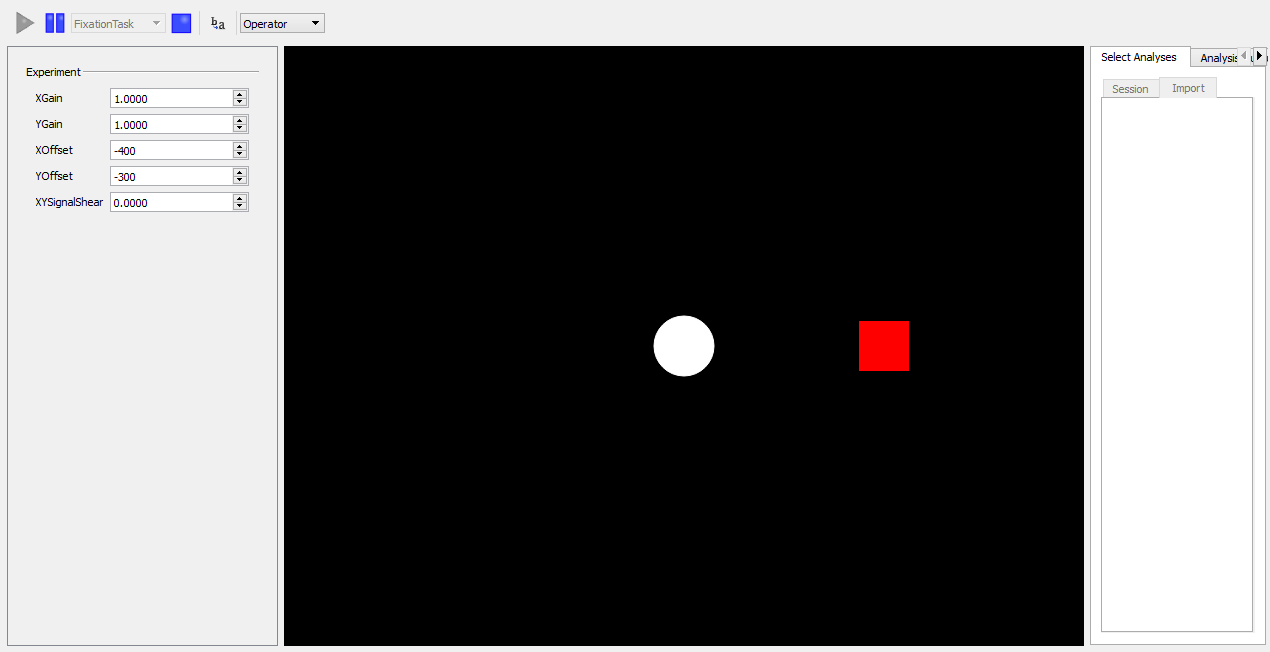


Figure 15 - Test Viewer running sample task

The best way to learn your way around the Test Viewer is to use it. Let’s go back to our sample design from the “[A Sample Design](#_A_Sample_Design)” section. We can now test it in the Test Viewer. Open the Test Viewer, find the Task name drop down on the toolbar, select our FixationTask and press play. If you didn’t miss any steps in building your task or add typos to scripts, everything should work as expected. Notice though that after we don’t fixate on the central circle, it sticks around during the ITI. This is a bug in our design. Since we set the fixateGraphic visible during the Fixation state and then return to the ITI without leaving the Trial level or setting the fixateGraphic invisible again, it stays visible until we complete a trial successfully. Let’s fix this by adding:

fixateGraphic.visible = false;

in the Fixation state’s Failed EntryScript and then test the experiment again. Now you should see things working like you would expect.

Let’s try introducing a script runtime bug to see how those are handled in the Test Viewer. Go back to the script that we just added and change “false” to “ffalse”, then run the Task again and don’t fixate on the fixateGraphic. When the script is called you will see a debugger window pop up.

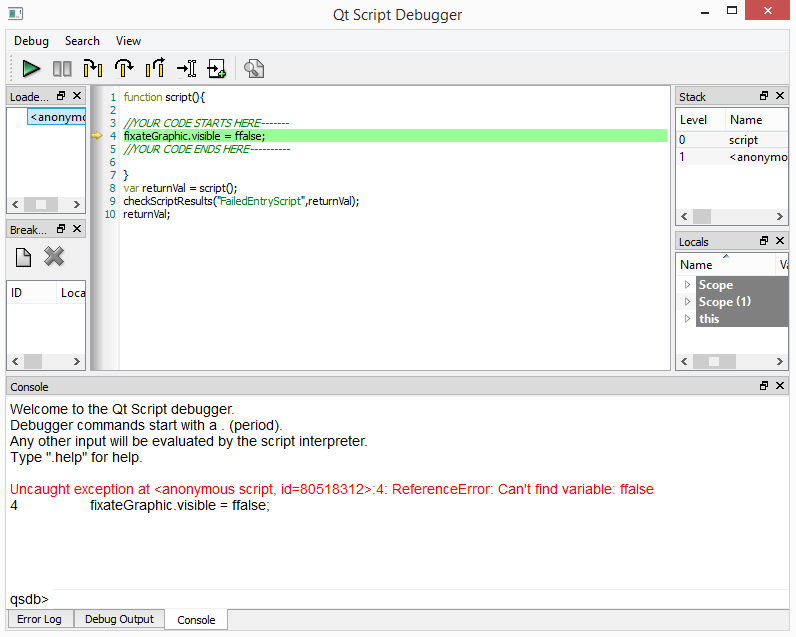


Figure 16 - Picto Run-Time Debugger

This debugger is very useful. It allows for stepping through detailed code. The command line at the bottom of the window allows us to run JavaScript just as if it were being called as part of the design. Another trick is to just add:

debugger;

to one of you scripts. As soon as the Test Viewer sees a script that includes this line, the debugger will pop-up and allow you to step through script code.

Keep in mind when using the debugger that there are still a number of improvements that need to be made. The most apparent issue is that a different debugger will pop up for each error in a different script. In some cases you may need to close multiple debugger windows in order to completely stop a test run. Also, it is important to close all debugger windows before leaving the Test Viewer to avoid strange effects in other viewers.

Now go back and remove our script error. Go back to the Test Viewer, go to the drop down that says “Operator” and change it to “Test Subject” then run the Task. You will notice that the green outline bounding the fixation window will disappear. This is showing you the view that the Test Subject will see in a real experiment.

The last important piece of the Test Viewer is the parameter frame. Currently you will notice some parameters in that frame like XGain, YGain, etc that are used to change eye calibration values. Go back to our Task design, select the fixateGraphic and check UIEnabled. Go back to the Test Viewer and you will notice that our parameter frame now includes a number of new parameters that can be used to change the fixateGraphic’s properties. These parameters work by changing the “init values” of the properties. This means that you can change them whenever you want but they will only go into effect when their values are copied to the property “run values” at the time that the fixateGraphic enters scope (ie. when Trial starts). Start the Task now, let it run and then change a parameter like the fixateGraphic color. You will notice that the change won’t go into effect until you use the mouse to fixate on the graphic and allow Trial to end and begin again.

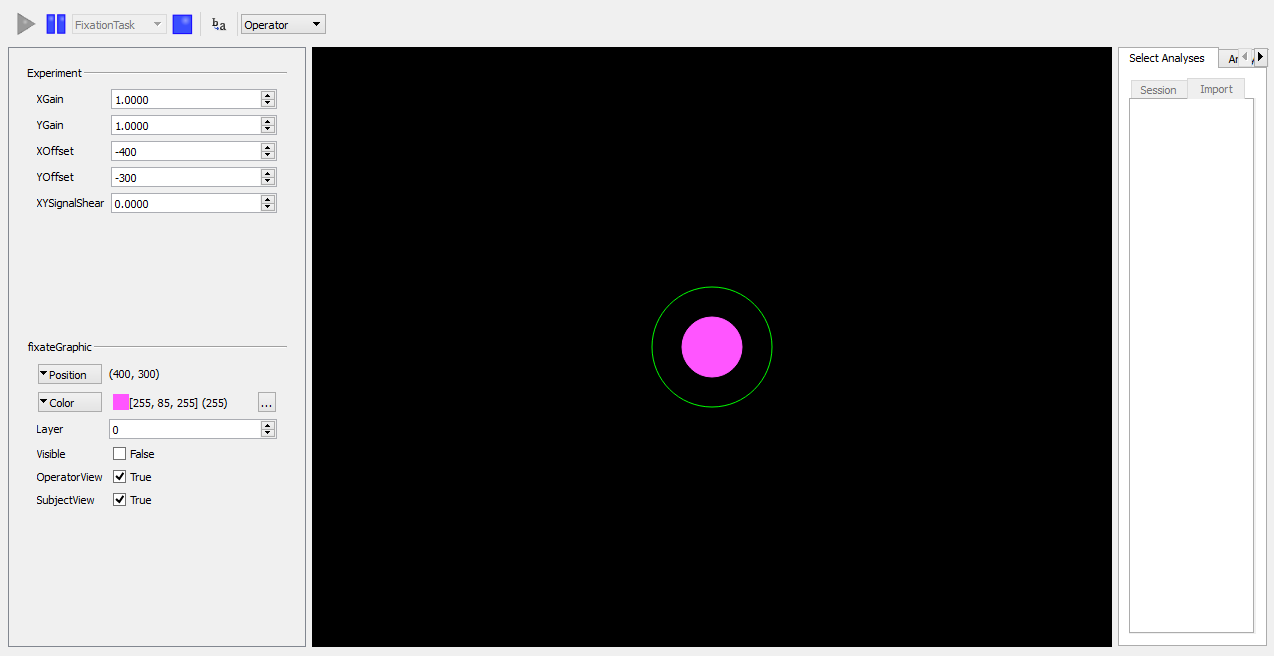


Figure 17 - Test Viewer with custom parameters

The fields on the right side of the screen are part of analysis and will be discussed in the [Analysis Testing](#_Testing) section. The pause button causes the Task to pause, but only if a Pause Point element is included in the design (we did not add it in our sample). The idea is that we don’t want to allow the task to pause in any random place since that can introduce invalid timing into a session. We want to be able to define the place at which a pause occurs, such as just before an inter-trial interval. We add the Pause Point element to the design at the desired location and then whenever pause is pressed execution continues until it reaches that point. Execution then waits there until the play button is pressed again and execution resumes.

# Analysis Design

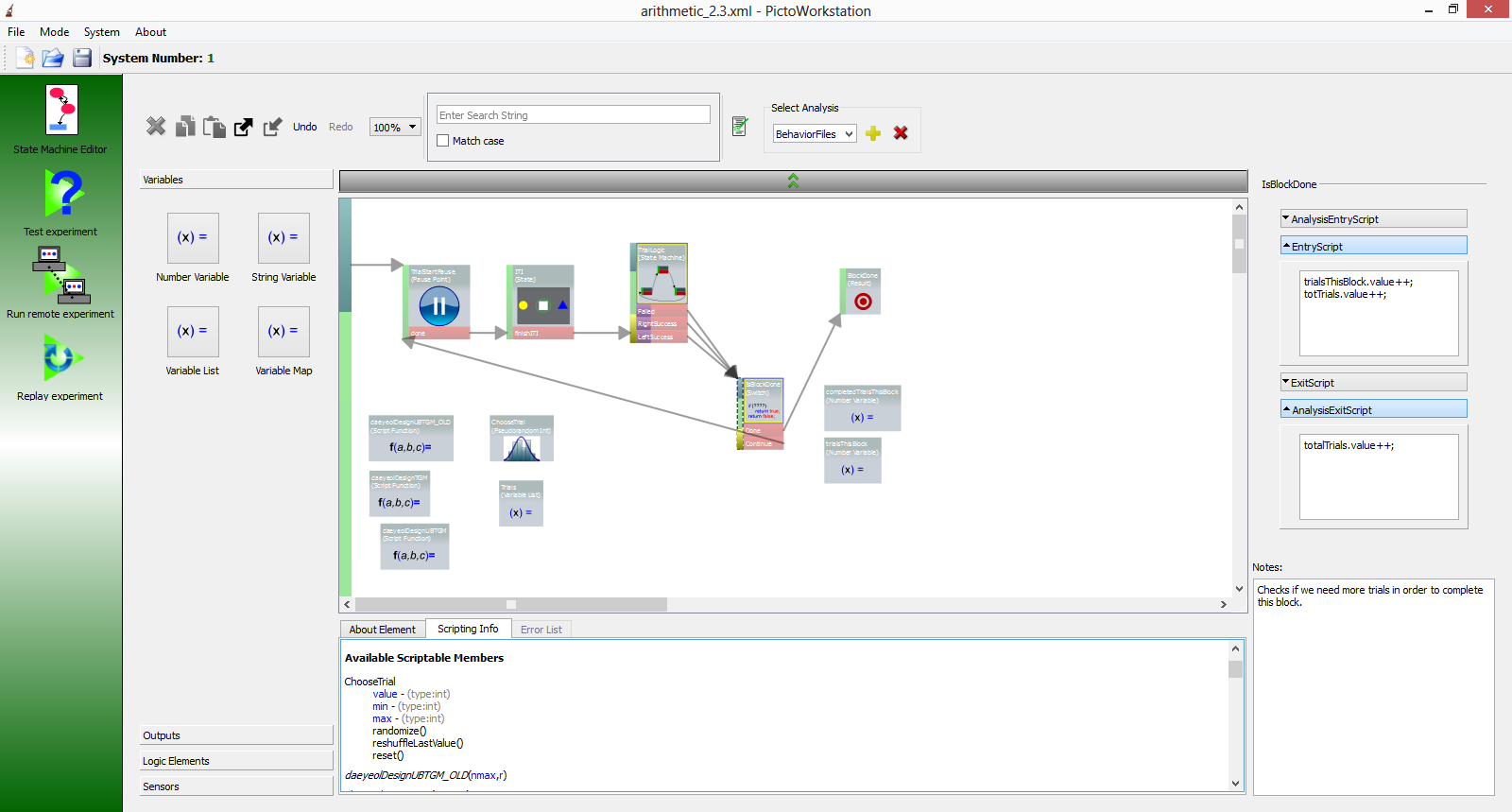


Figure 18 - State Machine Editor in analysis design mode

## Development

### A Sample Analysis

Let’s go back to our sample design and add some analysis code. Imagine that we want to find out what the test subject’s last eye position is before the Trial ends as well as the color of the fixateGraphic (in case we changed it with the runtime parameter). The first thing that we need to do is add an analysis to our design. Press the yellow “+” button on the toolbar and change the name in the dropdown to “TestAnalysis”. A number of things happens when you do this. First of all, all the “experimental design elements” are now partially greyed out. You will notice that you can click on the elements and see their properties, but you cannot change their properties or even move the element icons around in the canvas. When editing an analysis, the experimental design is strictly read only. This is because the analysis is not actually a part of the design. The analysis is a layer that lies on top of the design. We will see later that even if you run a session and record the data without having defined an analysis, an analysis can later be designed and imported into the session to analyze it. For this reason it is crucial that the analysis be a completely separate entity from the experiment design. Notice also that the dropdown where we changed the name to “TestAnalysis” now includes two elements, “None” and “TestAnalysis”. By switching back and forth between these two, we can switch from experiment design mode to analysis design mode. If we use the yellow “+” again, we can also add an additional analysis layer to the design and switch from one analysis layer to another to experimental design mode by using this drop down. If we want to remove an analysis layer from the design, we just select it with the dropdown, then use the red “X” next to the yellow “+”.

Now that we have created our analysis layer, let’s start building it up. We can start with a File Output element which will be used to output our analysis data to file for analysis in matlab or elsewhere. Go back up to the top level of the state machine (ie. the level that includes the “Trial” element). We will add the File Output element here since we don’t want it to enter and leave scope every time a new Trial stops/starts. You can find the File Output element in the Element Library under Outputs. Add it to the canvas, call it “outFile” and set its FileSuffix to “.test.txt”. We will now be able to use this element’s script functions to add text to the file. When our Task is done running, we will be able to save the output file to disk. It will be named according to the name of the current Task Run and the run time. The “.text.txt” suffix will be appended to the end of the name.

While we are on this level, let’s add a Signal Data element. We will use this to extract the test subject’s eye position. Find the Signal Data element under “Sensors” and call it “posSig”. The Top level of the state machine should now look like this:

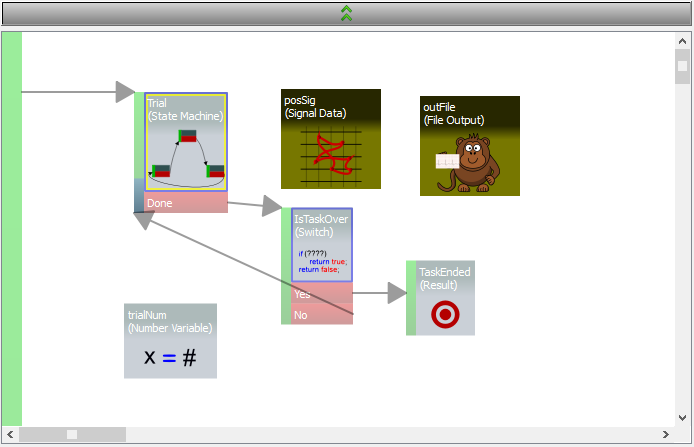


Figure 19 - Top level of sample analysis

Now, let’s go back down into the Trial element. First let’s add a String Variable to our analysis at the Trial level. When our analysis is active, any time Trial starts, the String Variable will be emptied and we can use it to store a single line of analysis data over the course of each trial. Find the String Variable under Variables, add it to the Trial canvas and call it “outputLine”.

Now we will add an analysis script to our design to extract experimental data and write it to the outputLine. Click on the green bar of the Stimulus element. You will notice that there are now six script options instead of three. The regular Entry, Frame and Exit Scripts are read-only now, but there are additional Analysis Entry, Frame, Exit Scripts that can be edited. Add the following code to the AnalysisExitScript:

outputLine.value += fixateGraphic.color;

outputLine.value += ","+posSig.getLatestValue();

This code takes a string representing the current color and adds it to the outputLine string. It then appends a comma and adds the latest position value to the string. The value returned from Signal Data’s getLatestValue() function is a JavaScript array (see About Element tab documentation). In javascript, when we add an array to a string it is automatically converted to a comma separated array, so we now have a string with three comma separated values: color, xPos, yPos.

Take notice at this point of the Stimulus element Start Bar. You now see that it is highlighted with a blue rectangle on top and a yellow rectangle on the bottom. In Picto, blue signifies “experimental design” and yellow signifies “analysis design”. The experimental EntryScript is signified by the blue rectangle and the AnalysisExitScript is signified by the yellow rectangle. You will also notice that all of the analysis elements that we have added to the canvas have a yellow tint. This helps differentiate them from experimental elements. If you check back up one level and look at the icon for the Trial element, you will notice that it now has both blue and yellow outlines. This indicates that there are both experimental and analysis scripts inside.

Now that we have our row of analysis data, let’s write it to the output file. Back inside the Trial element, select the start bar on the left side of the canvas and add the following code to the AnaysisExitScript.

outFile.writeLine(trialNum.value + "," + outputLine.value);

This code writes the trial number followed by a comma, followed by the outputLine with the analysis data that we generated. Since the AnalysisExitScript runs after the experimental ExitScript (as indicated by their order in the Property frame), the first trial number in our output will be 1, not 0. Our final analysis design at the level of the Trial element looks like this:

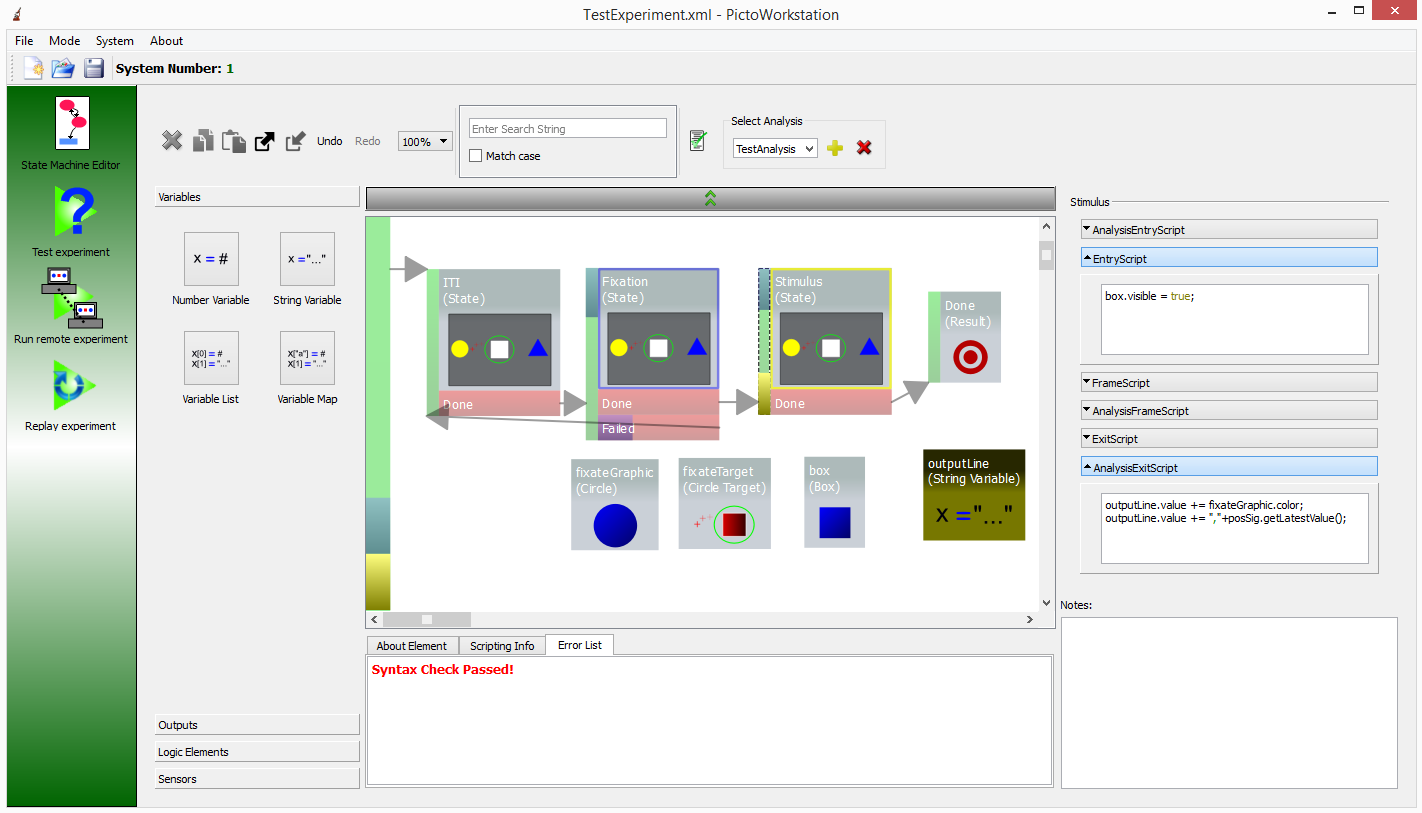


Figure 20 - Completed sample analysis at Trial level

Before we jump into testing our analysis, let’s go through a brief overview of some of the most important State Machine Editor analysis elements and tools.

### Analysis Elements

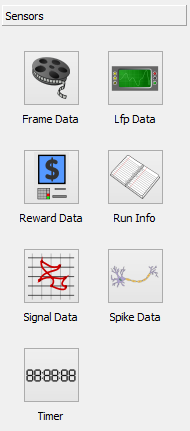


Figure 21 - Analysis Sensors from Element Library

In this section we will present a quick list of some of the most important analysis design elements. This list is by no means exhaustive and descriptions are broad. In particular, many analysis elements, like the analysis variables, are direct analogs to their experimental counterparts. For complete documentation of the analysis elements, the About Element tab in the bottom-center of the State Machine Editor should be used. This tab includes a context dependent documentation frame. Whenever you select an Element in the canvas or press its button in the Element Library, the frame will be updated with detailed documentation of that element’s function, properties and scripting interface.

* File Output – This is used to write analysis data to an output file. Simple script functions are used to write to the object. When included in an active analysis, an Output File Viewer will appear during test runs in the Test Viewer and playback in the Replay Viewer. The Output File Viewer provides an option to save the output file to disk so that it can be used in Matlab or other analysis.
* Frame Data – Used to extract data about precise frame presentation times over defined time ranges.
* LFP Data – Used to extract data about LFP values over defined time ranges.
* Reward Data – Used to extract data about precise reward supply times over defined time ranges.
* Run Info – Used to extract notes, name and other information saved in the Workstation during a Task run.
* Signal Data – Used to extract signal values and timing over defined time ranges for a set signal (ie. Position, Diameter).
* Spike Data – Used to extract neural spike timing and other data over defined time ranges.
* Analysis Scripts – These are added by selecting the Start Bar of elements in the state machine and editing their values in the Property Frame. They have access to all script functions and properties in the experimental and analysis designs. As control flow moves through the elements to which these scripts are attached they extract data from the state machine, operate on it and write it to an output element (currently File Output is the only output element, but in the future we hope to develop others, ie. plots, rasters, etc). These scripts are not allowed to change any data in the experimental design. If they do so a runtime error will be triggered in the Test Viewer.

### Analysis Tools

There are a number of tools that are available for use specifically with analyses, some of which only appear when designing in analysis mode. We go through them here.

The “Select Analysis” section of the State Machine Editor is used to create analyses, delete them, switch the active analysis and restore experimental design editing. The yellow “+” button creates a new analysis. The analysis is created with a default name that can be adjusted in the dropdown box. The red “X” button deletes the currently selected analysis from the design file. The dropdown box lets us select the analysis that we want to edit or move back to editing the experimental design itself.

When we are editing an analysis, two new tools appear in the toolbar, export and import. Export and import are similar to copy and paste in that they allow us to read elements from one location and reproduce them in another. The difference is that with analysis elements, we want to be able to extract only the analysis elements from multiple levels of the state machine at once and then add them to another state machine at all of the same levels and locations. This is useful when we build a task as a modified copy of another previously developed task. In that case, we can export some previously developed analysis from the old task and import it into the new one. If some of the names of elements have changed, we receive a warning and the specific analysis elements that cannot be imported are skipped. Even so, we are able to reuse a lot of the old code without multiple manual copy and paste actions on multiple levels. To export/import, select the element that contains analysis elements that you want to export and press the export button (), next select the element at which you want to import the analysis elements and press the import button (). Export / Import is typically used at the task level for exporting all of the analysis elements of one task and importing them into another.

Note that copy / paste is available for copying individual or multiple analysis elements that appear on the same state machine level, it is simply less efficient for element reproduction on multiple levels.

## Testing

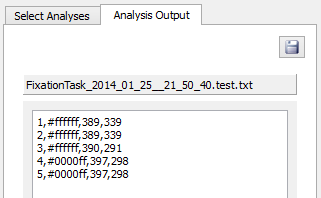


Figure 22 - Sample analysis output

Let’s get back to our sample analysis and try it out. Open the Test Viewer and select the FixationTask in the dropdown. On the right side of the main window, you will see a tabbed frame with “Select Analyses” and “Analysis Output” tabs. In the “Select Analyses” tab, select the “Session” sub-tab and you will notice a checkbox next to the name of the analysis that we created: “TestAnalysis”. This area shows us all of the analyses that have been defined in the current design and lets us select which analyses we want to activate during the current test run. Multiple analyses may be activated at the same time. Check the box next to “TestAnalysis” to activate that analysis and press the play button to run the task. Go through a couple of trials then use the parameter that we created to change the color of the fixateGraphic and continue running trials until the task ends after the fifth trial.

Now that we have finished the task, let’s look at our analysis results. In the tabbed frame on the right side of the window, select the “Analysis Output” tab. In this tab you will notice a single section titled FixationTask\_[*time when run started*].test.txt. This section includes all of the output that was written to our File Output element. If we wrote data to multiple File Output elements, all of their output data would appear here in multiple sections, titled according to the suffixes that we put into their FileSuffix property. In our case the section includes 5 lines, each of which contains the trial number followed by a string starting with “#” and two other numbers. The “#...” string is just the string encoded color of the fixateGraphic that we wrote to the File Output. In Picto, “stringified” colors are printed as a “#” followed by 6 hexadecimal digits with the first two representing the red component of the color (between 0-255 or 0-ff in hex), the second two representing the green component and the third two representing the blue component. The last two values are the x,y position values (in pixels) that we wrote to the File Output. You should notice that the color value changes at the trial after you changed the fixateGraphic color since the change only went into effect when control flow next reentered the Trial element. The x,y position values depend on where your mouse was when the trial ended. You can rerun the task and move your mouse around during the stimulus period to see that those values change accordingly.

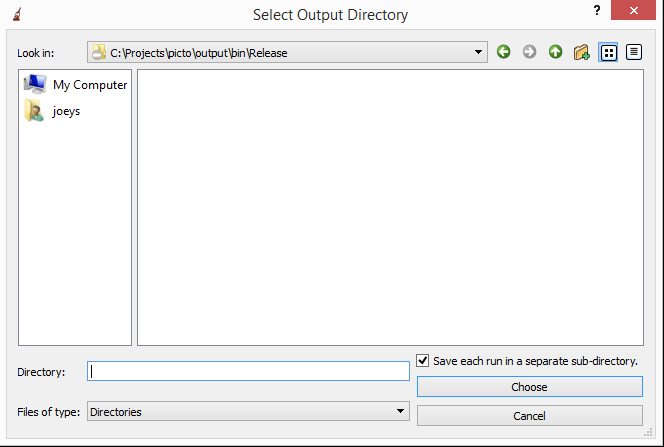


Figure 23 - The analysis output file path selection dialog

For this sample task the output data is not particularly detailed or useful; however, in a real task you will likely print out more complex data sets and want to look at the data more closely in Matlab or a similar application. Press the save button inside the Analysis Output tab and a dialog will pop up where you can select a path in which to save the Analysis Output file. In our case we only used one File Output object, but in a real analysis you may use many. The path selected in this dialog is the path at which all of your File Outputs’ files will be saved. The dialog includes a “save each run in a separate sub-directory” checkbox. When this is checked, all output files will be saved in a new directory that is titled according to the current task name and run time. Select a directory and press “Choose”. Navigate to the path that you chose in the Windows File Explorer, open the file in notepad or any text editor and you will see your analysis output data.

That is all there is to the mechanics of analysis testing. Remember that just like in the case of testing the task itself, bugs in analysis code will trigger a runtime error and cause the script debugger window to pop-up. “debugger;” can also be used to force the script debugger to pop-up at a particular analysis script line. It should also be noted that some analysis data source objects, like the Signal Data element that we used, allow you to extract future data over a range of times from an analysis script. When we are analyzing a previously saved session in the Replay Viewer, this can be very useful. In the Test Viewer, however, we cannot predict the future and so the data returned from this type of function is not valid. In cases where functions of this type return an array of values, the array will be empty. In cases where a function returns a single future value (ie. getNextValue()), the value will not be meaningful. This makes debugging the results of these types of functions difficult, so be very careful when using them.

# Running an Experiment

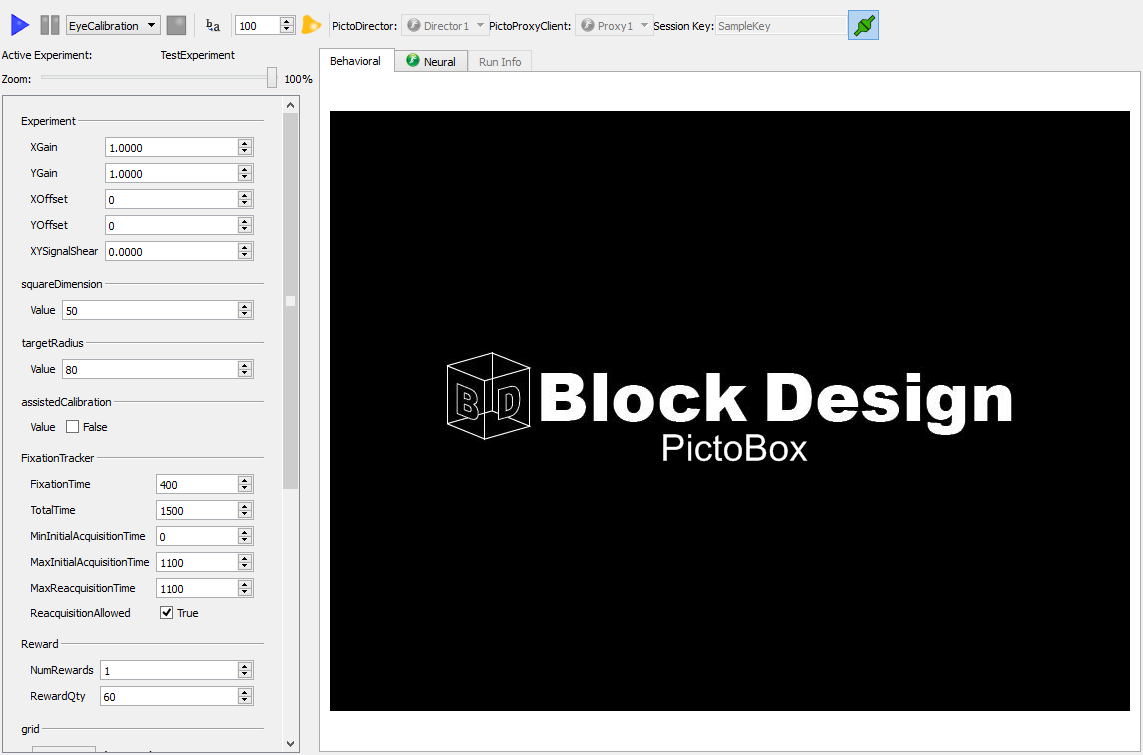


Figure 24 - Remove Viewer in a stopped session

In general, the process of running an experimental session in Picto is very similar to that of testing the design. The Remote Viewer, used to run experimental sessions, contains many of the same controls at the Test Viewer: Play, Pause, Stop, the task selection dropdown and the design specific left-side parameters are all the same. Running a session does require some “first time” setup of other applications in the Picto system, however, and there are certain additional things to keep in mind during a session run. In this section we will present a quick walk through of a standard experimental session with and without neural recording.

## Behavioral Sessions



Figure 25 - Setting up the Remote Viewer for a behavioral session

In order to start an experimental session, we need to be sure that our Picto server and director are up and running. If we were gathering neural data we would need to make sure that our Proxy was running as well; however, in this sub-section we will be describing a purely behavioral session. For a description of the Server / Director setup process, see the [Configuration](#_Configuration_1) section.

Starting a behavioral session is simple. Run the PictoWorkstation application on any computer on the Server’s local network. Open the design file that you want to run in the experiment, and go to the Remote Viewer window. In that window’s toolbar there are dropdowns for selecting a director and proxy to run the experiment. These dropdowns will contain the names of all of the directors/proxies that are on the local network with the same system number as the server. Currently, we recommend using only one director/proxy combination with each Picto server. In the future, a single server will be able to handle multiple simultaneous sessions. Select the desired director from the dropdown and leave the “No Proxy” setting in the PictoProxyClient dropdown. You will also need to enter a “Session Key” to start the session. This key is not secure and you should definitely not use any sort of secret password. The purpose of the key is simply so that no one else that wants to monitor your experiment from a different computer will be able to accidentally change parameters in your session. Only workstations with the correct key will be able to make changes to a running session. Enter a session key now and press the  connection icon.

In a few seconds, the connection icon should close and turn green () indicating that you are attached to an active session. You will notice that tools for selecting a task, starting it, providing rewards and zooming are now enabled. Typically, we begin a session by running the EyeCalibration task that is included in every new design. To start that task, select its name in the dropdown and press play just as you would in the Test Viewer. After a few seconds, the task will start on the director, and you will be able to monitor it in real time in your workstation.

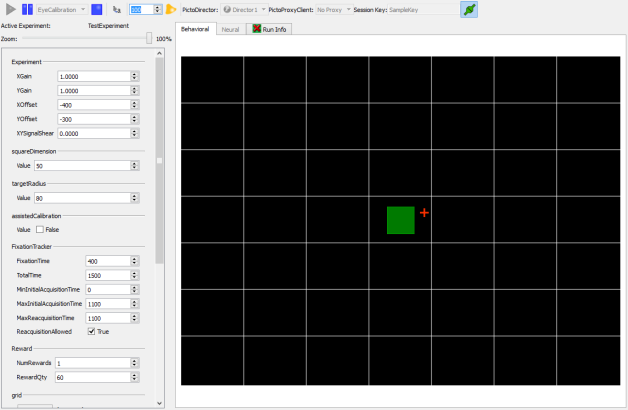


Figure 26 - EyeCalibration task in remote viewer

Eye calibration is something of an art, and the best way to learn to do it is to watch someone who has experience with it. The eye calibration task contains a large grid that is visible only to the operator. The task uses an Operator Click element to capture clicks in the main window of the workstation and use them to control the position of a Box Graphic. When the operator clicks somewhere in the grid, it causes the yellow box graphic to appear on the test subject’s otherwise empty display. Inevitably, the test subject looks at it, and the operator changes position calibration values based on the offset between where the position signal ended up and the position of the box that the subject actually looked at. Note that eye calibration parameters are an exception to the normal parameter rule. In general, workstation parameters only go into effect at the next time that their parent element enters scope. In the case of the eye calibration parameters, effects are instantaneous. The standard eye position calibration procedure is as follows:

1. Use the zoom slider at the top left to zoom way out of the main window and lower the x,y gain until the red crosshairs representing the subject’s eye position are visible and moving around somewhere around the main grid area.
2. Click in the center of the grid to make a yellow box appear. The eye position will jump to some spot and fixate briefly as the subject looks at the box.
3. In order to encourage the subject to look at the dot, provide a manual reward by using the “bell” button at the top of the workstation window, pressing F1, or using the reward trigger attached to a Pictobox.
4. Change the x, y offset values to move the position that the subject looked at to the center of the screen.
5. Repeat steps 2-4 until the eye position successfully moves to the central dot whenever you display it. You will notice that once the eye position enters the fixation area, the task will reward the test subject automatically.
6. Next, click somewhere off center and see where the position signal goes.
7. If the position signal goes in the wrong direction, change the +/- sign of the gain signal for the appropriate x/y dimension. If the position signal does not go far enough or goes too far, increase or decrease the gain in the appropriate x/y dimension.
8. Periodically, go through steps 2-4 again to make sure that the position signal does not drift off center with the increased x/y gain.
9. Don’t forget to reward the test subject manually until calibration is good enough that the reward is automatic.
10. Repeat steps 6-10 until the test subject is able to successfully fixate wherever you click on the main window.

Typically, the first eye day’s calibration will be the most difficult. After the first day’s calibration, assuming that the test subject and the camera used for eye tracking are placed in the same locations each day, you will have a good sense of the appropriate offset / gain factor magnitudes and day by day calibration will be a matter of “tweaking” values rather than an all-out search.

Once the eye calibration process is complete, stop the EyeCalibration task, select your experimental task from the dropdown and run it. Eye calibration values will be maintained from one task to the next, so the test subject should be able to start right away. If you notice any eye position drifting during your task run, you can always tweak the calibration parameters accordingly.

During the course of the run, parameters from the left hand frame can be adjusted, rewards can be provided, and the task can be paused and restarted. A “Run Info” tab is provided as a spot to save notes about the run, name it and mark it as “saved”.

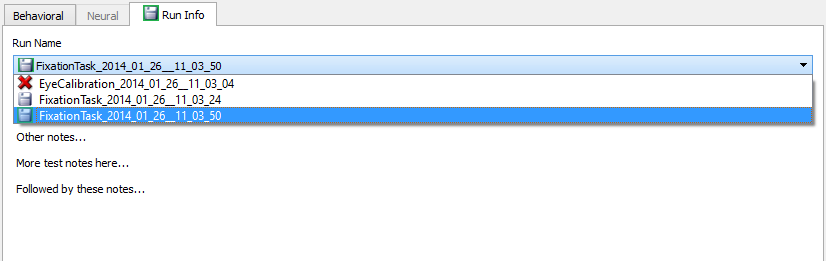


Figure 27 - Run Info tab

The contents of this tab are modified by selecting the appropriate run (all runs from the current session are listed), and pressing the large “Modify” button on the bottom of the frame. This will make all of the fields for the selected run editable. The name can be changed, notes can be added, and the “saved” value can be changed by clicking the top left button to change its icon from saved () to unsaved (). Values are stored when the “Apply” button at the bottom of the screen is pressed.

When the session is complete, close it by making sure that the session is in a stopped state, pressing the green connection button to disconnect and responding yes to the dropdown that asks whether you would like to end the session or just stop watching it.

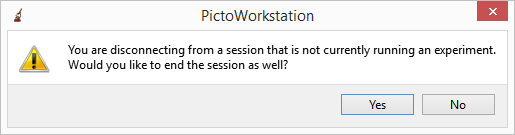


Figure 28 - End session dialog

The idea behind this dialog is that in actuality Picto sessions run entirely on the Director, Server and Proxy. The workstation is required only to create the session and start a task. Once the task is running, the workstation can freely disconnect or close down and the task will continue unaffected on the Director. For this reason, we don’t end the session whenever the workstation disconnects. We wait until there is no task running, and only then if a workstation disconnects we provide an option to end the session.

The last thing that we should go over with regard to running a session is joining it from multiple computers. So far, we have gone over the process of starting, running and ending a session from a single workstation, but as we alluded to in the previous paragraph, the workstation itself is not actually an integral part of a session. In actuality, the same connection button used to create a session on a selected director can be used to join another workstation to the session currently running on that director. Essentially, the meaning of the button is: Join me to the session running on that director, and if there isn’t one, make one. If a workstation connects to a director’s session without typing in the correct Session Key, it can watch what is happening in the session, but cannot control it. If a workstation does include the correct Session Key, it plays exactly the same role as the workstation that started the session. In fact, any changes made on one (start, stop, pause, parameter values, reward quantity, run info, etc.) will be automatically propagated to the other.

### Pictobox



Figure 29 - Pictobox front panel

A Pictobox is not required in order to use Picto, but if it is not used a complex system involving a PC with a DAQ card and a breakout box plus relays for reward control, etc will have to be built. Pictobox provides all of this in one simple package, so running Picto experiments on a Pictobox is highly recommended. Detailed information on building, configuring and using a Pictobox is provided in the PictoboxDocumentation document; however, we provide a short overview of day to day use here from a user perspective.

If the Pictobox is configured correctly, you should be able to simply turn it on using the Power button, wait a minute or so for it to boot, and start using it once you see information appear in the LCD display. The main display includes several pieces of information. On the top left, you will find the name of the Director application running on the Pictobox. On the top right an icon indicates the current status of the Director application. Possible status values are: *Director not running, No server connection, Server connected, Session Stopped, Session Running, Session Paused and Session Ending*. The bottom left “R:?” value shows the current duration in milliseconds of the default reward. The value here is the reward duration that will be used when a reward button on the Pictobox or the reward icon () from the workstation is pressed. Note that rewards that are incorporated into a task design are controlled by the properties of those reward elements. The “F:?” value in the middle of the bottom row is the default duration in seconds for a reward flush. When the flush button on the box is pressed, a flush will continue until this time is up or the flush button is pressed again, whichever comes first. The “C:?” button indicates the current reward channel. Currently Pictoboxes only support C:1. Turning the rotary dial next to the LCD scrolls the window down to show the Pictobox’s current IP address. Pressing the rotary dial moves the display to a menu mode where the various settings, name, reward duration, flush duration, and reward controller can be changed. From this menu you can return to the main settings page. Alternatively, control returns to the main settings page after a few seconds without activity.

The Pictobox panel includes reward and flush buttons as well as a jack where a push button with phono style plug can be inserted and used to trigger a reward.

For any more detailed information than what was described here, please refer to the PictoboxDocumentation document.

## Behavioral / Neural Sessions

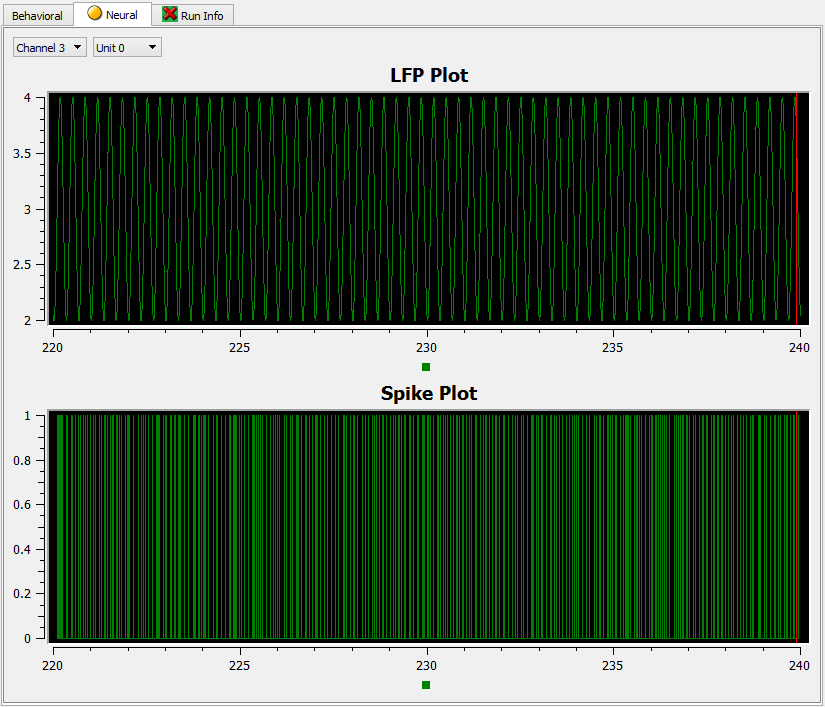


Figure 30 - Neural plots in the Remote Viewer

Running a session with neural data is only slightly different from running a session without it. First of all, make sure that your proxy application is running on the neural system. For instructions on setting up the proxy application, see [Configuration](#_Configuration_1). Before starting your session, make sure that the neural system (currently Plexon and TDT are supported) is open and recording data. Currently, the Proxy application must be able to connect to the neural system at the time that the session starts and that means that the neural system must be in recording mode when the connection button is first pressed in the workstation’s Remote Viewer.

Once the Proxy application is running and the neural system is recording. Start your session by selecting the appropriate director and proxy from the Remote Viewer dropdowns, selecting a Session Key and pressing the connection button. Once the session begins, notice that the “Neural” tab that was not active in a “behavioral only” session is activate here. This tab contains an indicator light that flashes when neural data is coming in. It is a good idea to glance at this at least once toward the beginning of a session to be sure that everything was set up correctly.

If you open the “Neural” tab, you will find two plots and two dropdowns. The dropdowns are used to select the channel and spike unit whose data appear in the plots. The upper plot shows LFP data for the selected channel and the lower plot shows spike timing for the selected channel and unit combination. The plots show neural data over a window of 20 seconds. Each includes a red line which only moves during the course of a task run and indicates the current session time. You will notice that in some cases neural data lags behind the red line by up to a few seconds. There is some built in periodic lag of about 500ms in the LFP system since the proxy sends LFP data to the server in 500ms batches, but any lag beyond this is a result of delays in the neural system software’s providing of data. In testing, we have found that the TDT system provides data with slightly greater lag than the Plexon system.

Beyond this, the only difference to keep in mind when running sessions with neural data is that the size of the session file will be significantly larger. Session file sizes of several hundred megabytes are normal when neural data is recorded.

# Session Playback and Analysis

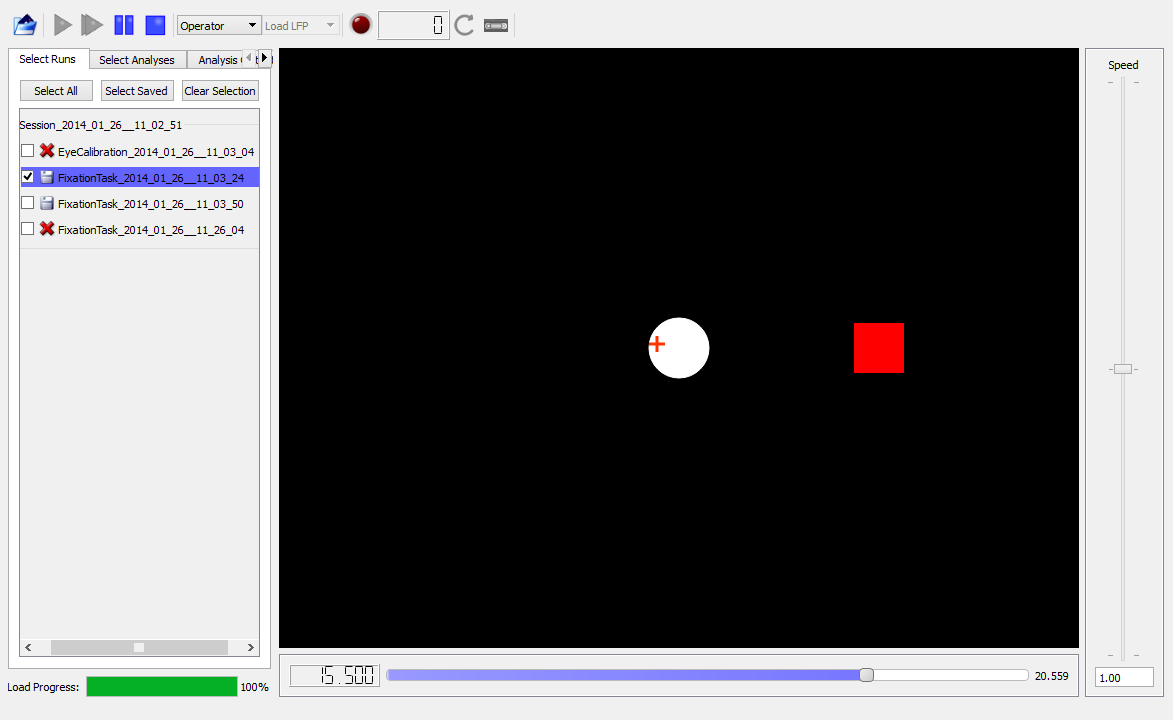


Figure 31 - The Replay Viewer

Since analyses in Picto are designed as layers on top of an experimental design, the Replay Viewer performs double duty, playing back a session and analyzing it in the process. In this section we will start by going through the process of playing back a session file, then continue on to explain how to analyze the session at the same time.

## Playback

In order to playback a session, the first thing you will need is a session file. Session files end with a .sqlite extension and can be found on the server that was used to run the session. They are saved in a directory called “sessions” that is located at the same path as the Picto bin directory. Session subdirectories are named according to the name of the director that was used to run a session. Within each of those directories, subdirectories are named according to the name of the experiment that was used for the session. Inside those directories you will find the session files. Since session files include the design that was used to run them, you can use any session file at all to go through this section. If a task was run during the session, then you should be able to play it back.

Once you have the session file, press the “Open Session” button () that appears next to the play button, navigate to the session file, and open it. Do not confuse the “Open Session” button with the “Open Design” button () that is at the top left of the workstation’s outer application window. These buttons have different purposes. “Open Session” selects a session for playback. “Open Design” selects a design for the purpose of editing it, running a new session, or importing analyses into a session that is being played back. When your session has been opened, the “Select Runs” tab at the left side of the window will include all the runs, saved and unsaved that are included in the session. Select one of these and press play. Once play is pressed, the session’s data will load (this can take a few seconds to a few minutes depending on the size of the session) and playback will begin.

During playback, there are a number of controls available for use. They are listed below:

Playback Speed Slider – The large slider on the right side of the display can be used to control playback speed at anything from 0.1 times to 100 times normal playback speed (note that the maximum playback speed may be limited by your computer’s processor capability).

Time Slider – The time slider underneath the display shows the current playback time. It can be moved to jump forward or backward to a different time. The blue highlight inside the slider requires some explanation. Since the Replay Viewer is used for analysis, anytime that we jump forward or backward, we need to make sure that analysis will remain valid. When jumping forward, this means that we must actually fast forward, playing through everything in between the current time and the new time quickly in order to make sure that all analysis code is called correctly on the way to the new position. When jumping backward in time, since we have no way to figure out how to automatically undo all of the changes that were triggered in analysis, we just restart playback and fast forward to the set time. The blue highlight indicates the current “fast forward” position on the way to a new playback position. During normal playback, it simply goes up to the current playback position. During jumps, it quickly moves forward until it meets the playback position at which point playback continues.

Play, Pause and Stop – These buttons function as you would expect. Play starts / resumes playback. Stop stops playback and brings the time slider back to zero. Pause pauses playback at the current position until play is pressed again.

Operator/Test Subject Dropdown – This sets the current viewer in the display. When set to operator we see what the operator saw at the current playback time. When set to test subject we see what the test subject saw at the current playback time (with the exception of the position cross hairs).

Session Recording System – The session recording system () includes a “Toggle Record” light / button a recording time indicator, a “Restart Record” button and a “Save Recording” button. When the toggle record button is pressed, everything that appears in the main display will be recorded to the video file. If playback is sped up, the video file will includes sped up video. If the Operator/Test Subject button is changed, the change will be reflected in video. Recording can be paused and resumed by pressing the record button repeatedly. The restart record button erases the video recorded so far so that recording can start over. The save recording button opens a “save as” dialog that allows you to save the recorded file to disk.

## Analysis

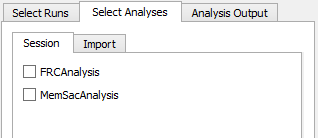


Figure 32 - Select Analyses tab in Replay Viewer

Using the Replay Viewer for analyzing a session is not much more complicated than normal playback. In order to analyze the session, the first thing you will need is a session with an analysis defined in its design. Actually, the Replay Viewer can be used to import new analyses into sessions that were recorded without one, but for our purposes right now, you should use one that has an analysis built in. Load the session and select one of the runs from the Select Runs tab. To analyze the selected run, go to the Select Analyses tab. You will find two sub-tabs there, Session and Import. The session tab includes all of the analyses that were saved as part of the design inside the session file. The import tab lists all of the analyses that are part of the design file that is currently opened in the workstation (ie. the one that you would be editing if you switched to the State Edit Viewer). The idea here is that in addition to running analyses that you designed along with the experiment, you can also open the design that was used in the session in the state edit viewer, add a new analysis or edit the original one, then select the analysis in this import tab to use the new analysis to analyze the old session. In our case we will just use the analysis stored with the session, so go to the session sub-tab and check the checkbox next to the desired analysis. Now press play to play the session and analyze it in the process.

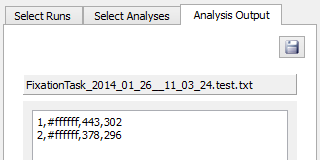


Figure 33 - Analysis output in the Replay Viewer

After the session data is loaded, playback will start. If this is a long run, you can just drag the time slider to the end to analyze the run quickly. When it is done, go to the Analysis Output tab and you will see one or many (depending on the analysis code) File Output widgets with analysis output data. To save the data to file press the save button in the tab. The “save as” widget that pops up works exactly the same as the one in the [Analysis Testing](#_Testing) section. See there for details about its function.

While the process outlined above is sufficient for analyzing session data, there are a number of shortcuts that can be used to make the process faster. First of all, the process of loading session data when play is pressed can take a significant amount of time for large sessions, particularly when LFP data is stored. If LFP data is not needed for the analysis, the “Set LFP Requirements” drop down can be used before pressing play to disable LFP analysis and significantly speed up the load process. Also, if your purpose is only to analyze sessions and you don’t need to actually watch the session playback, you can use the “Run to End” button () next to the play button. This button will cause the viewer to playback the run with analysis at maximum speed without displaying it on screen. When this button is used, a pop-up comes up asking if you want to automatically save runs when the analysis is complete. If you choose yes, you will be able to select a save location, and then just let playback run without needing to babysit the application waiting for analysis to end so that you can save it.

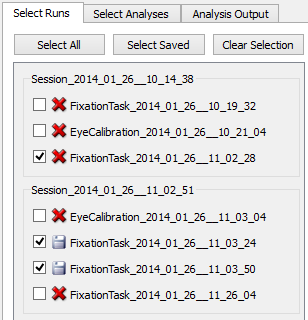


Figure 34 - Run selection for batch analysis

Lastly, it is frequently useful to be able to analyze multiple runs at the same time as a batch. This can be done from one or multiple session files at the same time. To load multiple sessions at once, simply select multiple session from the Open Session dialog using control-click. The sessions with all of their runs will show up in the Select Runs tab. Check the checkboxes next to all of the runs that you want to analyze, or use a shortcut selection button like “Select Saved”. Since local analyses may be different in different session files, the Select Analysis tab only lets you use imported analyses for batch analysis. If you know that all of the session files use the same analysis, you can just load one of the session’s designs using the workstation’s “Open Design” function then use it for importing. In batch analysis, you will want to use “Run to End” so that you can set a save location once and send everything there. Press that button, select the save location and let the analysis begin.

Note that in some cases due to RAM restrictions, Picto may have trouble analyzing multiple large sessions as a batch. In cases like this, the sessions that were not be successfully analyzed will be highlighted red at the end of the batch run. A dialog will pop-up and provide the option to select failed analyses and try them again.

# Upgrading Software

To upgrade Picto applications, you need only update the PictoServer application.  Once the server is upgraded, all Picto applications on the same system as that server will find out about the upgrade when they next connect to the server and upgrade automatically.

To upgrade the server, first run the shortcut that you made when installing the Server (see [Configuration](#_Configuration_1)).  Press the "Remove PictoServer Service/Daemon" button.  When it is finished, close the application.  Replace the contents of the C:\projects\picto\bin folder with the new Picto binaries.  Restart the Server shortcut.  Press Install PictoServer Service/Daemon.  When that is done, press "Start PictoServer Service/Daemon".  That completes the upgrade process.  In a few seconds, or when they are next run, attached Picto applications will upgrade automatically.

# Troubleshooting

As in any major software package, there will be times when unexpected things will occur. This section includes a number of steps to take when problems come up.

Server Connectivity Issues – If either the director or proxy seem to be experiencing strange lags or jumpiness, if workstation commands are taking an unusually long time or task presentation in the Remote Viewer seems jumpy, the problem may be on the server. As the central hub, connectivity with the server is vital for things to run smoothly. The server connection may be spotty due to problems with its host computer or if it enters a strange state.

If you have recently installed the server application or the computer that is running it, make sure that you are using the correct network drivers for your network card. Windows’ default network drivers can lead to very slow network performance that doesn’t meet the server’s requirements. Installing proper network drivers has been enough to solve problems like this in the past.

If nothing has changed recently about the server application’s host computer, the application may have entered a strange state. This has not been known to happen frequently, but if it does you should try to perform a “Full Server Reset”. The procedure is as follows.

1. Find the PictoServer shortcut that is created as part of the Server installation (see [Configuration](#_Configuration_1)).  The current convention is for this shortcut to appear on the desktop of the computer running the server.
2. Run the shortcut and you will see a dialog and a DOS window in the background.  Press the "Remove PictoServer Service/Daemon" button on the dialog.  When the "Install PictoServer Service/Daemon" is enabled, the operation is complete.
3. Close the application.
4. Delete the contents of the C:\projects\picto\config folder.
5. Run the PictoServer shortcut again.  This time, press "Install PictoServer Service/Daemon".
6. When "Start PictoServer Service/Daemon" is enabled, press it.  Wait for "Stop PictoServer Service/Daemon" to be enabled, indicating that the service has been started, then close the application.

Large Session Analysis Problems – Sometimes Picto can have problems analyzing large session files due to RAM restrictions. The Replay Viewer is designed to detect these issues, inform the user and provide an option to rerun failed analyses. In some cases, even the rerun analysis fails. In cases like this, restart the Workstation application and try the analysis again. For particularly large datasets, fundamental RAM limitations of 32-bit programs may be the issue. For this reason, we have developed a 64-bit version of the Picto Workstation.

Jumpy Eye Position Signal – We have yet to come across a jumpy eye position that is due to a problem in Picto. We have encountered this type of problem in the past, and it led to questions about whether the issue was with Picto so we are including a discussion here. In our experience, jumpy eye position signals were due to problems in setting up the eye tracker. In particular, if the angle between the light and the eye tracker camera leads to significant glare in the camera, it can cause significant errors in the camera tracking software.

Other Issues – Other issues are bound to come up at some point. For those eventualities, it is always a good idea to try the “magic restart”. When in doubt, restart the computer. It is amazing how many problems end up being solved by a good old restart.