Tutorial on using the Psychology Experiment Building Language (PEBL) in the Laboratory, the Field, and the Classroom

Shane T. Mueller, Ph.D.

Applied Research Associates

smueller@ara.com

smueller@obereed.net

shanem@mtu.edu

Held at the 2011 Meeting of the Cognitive Science Society

1pm-4pm, Wednesday, July 20, 2011

Boston, MA

Cognitive Science 2011 Tutorial on PEBL

Held at the 2011 Meeting of the Cognitive Science Society

1pm-4pm, Wednesday, July 20, 2011

Boston, MA

http://cognitivesciencesociety.org/conference2011/tutorials.html

About: A half-day tutorial workshop will be held on PEBL as part of the Cognitive Science Meeting. To attend, you must register for the entire conference. You do not, however, need to register specifically for the tutorial, or pay extra fees for the tutorial.

Tutorial on using the Psychology Experiment Building Language (PEBL) in the Laboratory, the Field, and the Classroom

Presented by: Shane T. Mueller, Ph.D.

Applied Research Associates

Downloads

Prior to the tutorial, download the following. Copies of these files will also be available the day of the tutorial.

For Windows: http://sourceforge.net/projects/pebl/files/special/PEBL%20Tutorial%20Win32.zip/download

For OSX: http://sourceforge.net/projects/pebl/files/special/pebl_tutorial_osx1.zip/download

For Linux: Ubuntu/Debian .deb file for PEBL 0.12 available at:

http://sourceforge.net/projects/pebl/files/pebl/0.12/pebl-0.12-linux-2.6-intel.deb/download

Background:

Laboratory and field research in cognitive science often uses computer-based tools to design experiments and collect data. Although researchers are typically happy to exchange the data obtained from such studies, sharing the actual software used to collect the data is more difficult. This partly stems from the widespread use of special-purpose proprietary software tools to collect data, which prevents exchange and review of the actual experiment specification (without purchasing costly licenses) and can at times prevent researchers from accessing their own past experiments. Often, without a good understanding of the exact experiment, the data can be of little use. Furthermore, such systems are typically tied to a particular operating system platform, reducing the ability to exchange, modify, and evaluate details of an experiment. This is an obstacle to scientific progress, where it is critical to be able to share, evaluate, modify, and test the paradigms that we use to develop and test theory.

The Psychology Experiment Building Language (PEBL; Mueller, 2003; 2010) was developed to overcome these obstacles. It is a cross-platform Free (GPL) software tool for designing and running computer-based laboratory research that has been in development for the past eight years, and is used in research laboratories around the world. PEBL incorporates a set of approximately 50 standard laboratory tests which are increasingly becoming the standard non-commercial versions of classic neuropsychology tests (including, for example, tests such as the Wisconsin Card Sort, Iowa Gambling Task, Tower of London task, and pursuit rotor task). PEBL is a non-commercial community-supported project that aims to enable experiments to be shared as easily as data.

Objectives:

This half-day tutorial will provide experiential training for using PEBL in laboratory and field research, and in the classroom. It will begin with the basics of using PEBL and its standardized tests, with a discussion of the growing body of published literature using PEBL. Basics of the language will be covered to enable attendees to understand the workings of existing experiments and to modify them. Following this, we will develop one or more experiments 'from scratch' to highlight the important basic steps in experimental design. We will conclude with discussions and examples using PEBL outside the laboratory, in field research, questionnaires, and classroom settings. As an outcome of this tutorial, attendees will have the ability to incorporate a powerful tool into their teaching and research repertoire, which both makes research easier to conduct and enables improved and open scientific exchange and standardization.

Qualifications:

Dr. Mueller is the originator and developer of the PEBL, and has implemented each of the tests in the PEBL Test Battery. He is a cognitive scientist studying human performance and decision making in the field and the laboratory, and developed PEBL to reduce the complexity of developing experiments in these settings. He is a Senior Research Scientist at Applied Research Associates, Inc., in Dayton, OH.

Need/Justification for Tutorial:

Developing new experiments in PEBL is fairly simple, but can require guidance and direct feedback to understand its methods and intent, which is best handled in a tutorial/workshop setting. PEBL has the potential to be useful for laboratory researchers and instructors within the cognitive science community. These especially include empirical laboratory research psychologists and linguists, which constitute a large proportion of the membership of the cognitive science society, and the attendees of the conference.

Audience:

The target audience is primarily researchers in psychology and related disciplines who are involved in empirical work doing laboratory data collection. It will be targeted to two main groups: (1) graduate students, post-doctoral researchers, and faculty researchers who design and conduct laboratory experiments; and (2) instructors who wish to enhance classroom instruction using experiment demonstrations.

Special Requirements:

Participants will need to bring a laptop (Windows, Linux, or OSX), and should download/install the PEBL software and PEBL User Manual at http://pebl.sourceforge.net prior to the tutorial.

Details about the workshop can be found at: https://sourceforge.net/apps/mediawiki/pebl/index.php?title=CogSci2011_Tutorial#Tutorial_on_PEBL

About PEBL

- PEBL is Free psychology software for creating experiments .
- PEBL allows researchers to design their own experiments or use ready-made ones
- PEBL enables the exchange of experiments without license or fee
- PEBL offers a simple programming language tailor-made for creating and conducting many standard experiments. It is Free software, licensed under the GPL, with both the compiled executables and source code available without charge.
- PEBL is designed to be easily used on multiple computing platforms, and compiles natively under Win32, Linux, and Macintosh Operating Systems. PEBL is used by researchers around the world, and its most recent version (PEBL 0.11, released in August 2010) has been downloaded more than 5,000 times.

PEBL Website:

http://pebl.sourceforge.net

Syllabus

1. Introduction, Background, Goals
2. Launching and running PEBL on different platforms.
    - Using the PEBL launcher.
3. The PEBL Language, Syntax, and Functions
    - Language and Syntax
    - Input and Output
    - Display
    - Response collection
    - Randomization and Experimental Design
    - Graphics
4. Using and modifying tests from the PEBL Test Battery
    - Review of available tests
    - Modifying test example 1.
5. Developing New Tests
    - The PEBL Programming Philosophy
    - Development Strategy
    - Create Stimulus
    - Experiment Template
6. Beyond the Laboratory
    - Small-scale applications for data coding, recording field data
    - Demonstration: PEBL in the classroom
    - Demonstration: Creating Surveys and Questionnaires using a spreadsheet.
    - Open Tests, Open Norms

Launching and Running PEBL

The following launch instructions are valid for the PEBL 0.12 beta used for the tutorial. The final release may have slightly different methods.

Download

Prior to the tutorial, download the following: (Copies of these files will also be available the day of the tutorial.)

For Windows: http://sourceforge.net/projects/pebl/files/special/PEBL%20Tutorial%20Win32.zip/download

For OSX: http://sourceforge.net/projects/pebl/files/special/pebl_tutorial_osx1.zip/download

For Linux: Ubuntu/Debian .deb file for PEBL 0.12 available at: http://sourceforge.net/projects/pebl/files/pebl/0.12/pebl-0.12-linux-2.6-intel.deb/download Linux users should also download the OSX archive to get access to the battery and launcher.

Starting the PEBL Launcher

On Windows

The beta release of PEBL does not have an installer. It is simply a .zip archive that you unzip. If you have downloaded and installed a previous version of PEBL, it will not impact that set-up (and of course, it will not run if you use those menu shortcuts).

Once you unzip PEBL, you can run the launcher by double-clicking on the launch.bat file in the unzipped directory. When you launch a script, it will actually create a file called tmp.bat that will itself launch your experiment. You can rename this and adapt it in the future to circumvent the launcher altogether.

The tools/ directory contains a few additional handy tools: a data file merger, and a copy of notepad2, a decent programmer's editor.

On OSX

On OSX, PEBL is distributed as a .zip file. it contains an application bundle (in the tools/ directory), as well as a some additional directories and files.

Although the PEBL system is distributed in a application bundle, it will not do much if you double-click on it. You must copy the PEBL_OSX app to Applications in order to use the launcher. Then, to run the launcher, use the launch.command file, which will open the launcher. This launcher expects the application to be installed in Applications.

You can alternately drag .pbl files onto either one of the dropscript files in the .zip archive.

The tools/ directory contains a copy of 'flip' a command-line tool that lets you get rid of windows-style linebreaks (the 0.12 beta sometimes has trouble with these), and an open source programmer's text editor that can be used for editing PEBL scripts. A handy way to run PEBL on osx is via the commmand line. Read the documentation in the .zip file for more details.

On Linux

We created a .deb file of PEBL 0.12 beta, suitable for use on ubuntu and other systems. Linux users should also download either the OSX or Windows bundle so they get a copy of the newest PEBL Test Battery and launcher.

The PEBL Language, Syntax, and Functions

Language and Syntax

The basic syntax of PEBL is intended to be forgiving and to encourage readable experiments. To write a PEBL script, you basically need to open a text editor, and save the file with a .pbl extension. No special authoring tools are necessary to create an experiment. On the other hand, no special authoring tools are available, which can make writing your first experiment a little challenging.

Everything PEBL does must be embedded within one or more functions in the text file. A function is created with the define keyword. Every function must start with a Capital letter, but afterwards the name is not case sensitive. The first thing you need when creating an experiment is a Start function. When PEBL runs, it looks for a function called Start. A function also can return a value using the return keyword, which is shown below:

```
define Start(p)
{
  #The # character is a comment character.
  #Anything on its line following it will be ignored.
  return 100
}
```

See demo01.pbl

Anytime you want to keep track of something, you use variable. Variables always must start with lower case (after that they are case-insensitive). You can assign them using the two-character '<-' command.

```
define Start(p)
{
  #The # character is a comment character.
  #Anything on its line following it will be ignored.
  a <- 100
  b <- "One hundred"
  c <- a +" = " +  b           #Creates a string '100 = One hundred'
  return c
}
```

See demo02.pbl

Display

This is a legal PEBL script but it really doesn't do much. Usually, an experiment will need to display something for a subject to react to. To do this, we need to first create a window to put stimuli on. PEBL does not actually need a window to work, although without a window, it is really only good for data processing. Windows are created with the MakeWindow() function, which must be assigned to a variable.

```
define Start(p)
{
   #Create a window with a grey background:
   win <- MakeWindow()
   Draw()   #You need to draw the window for anybody to see it:
   WaitForAnyKeyPress()  ##Simple input function--waits for a key to be pressed.
}
```

See demo03.pbl

This program doesn't do much. You will also need to create stimuli, instructions, images, etc. to make your experiment. There are literally dozens of such objects in PEBL you can use to create the graphical objects of an experiment. One of the simplest is a text label. A text label simply holds some text you want to display, at a particular location on the screen, with a particular font. To create a simple label, you can use the EasyLabel() function (there are other ways to make a label that give more control over some of the details, but this is the easiest). The following create a label centered at point 100, 100 (from the upper left), in fontsize 18. Now, you use the win variable to add the label to the screen.

See demo04.pbl

```
define Start(p)
{
   win <- MakeWindow("black")  #You can set the color of the window at startup
   label <- EasyLabel("one two three", 100,100, win,18)
   Draw()
   WaitForAnyKeyPress()
}
```

A label has a number of properties that can be set or changed in order to create your experiment. A property is accessed by appending its name with a . after the variable name. If you update a property, it won't change on the screen until you issue a 'Draw()' command. A few basic ones include label.x, label.y, and label.text. Here is an example of how to use them.

See demo05.pbl

```
define Start(p)
{
   win <- MakeWindow("black")  #You can set the color of the window at startup
   label <- EasyLabel("one two three", 100,100, win,18)
   Draw()
   Wait(500)  ##This waits 500 ms before continuing

   #Do a mini-animation
   label.x <- 120
   Draw()
   Wait(100)
   label.x <- 140
   Draw()
   Wait(100)
   label.x <- 160
   Draw()

   label.text <- "Done"
   Draw()
   WaitForAnyKeyPress()

}
```

Output

When you create an experiment, you typically want to record information to a data file somewhere. The simplest way to do this is to use the Print() function. The Print function will send whatever is inside the function to 'standard out' stream, which is either the command line (on linux or OSX) or a file called stdout.txt (on Windows, or when using the launcher on any platform). There are two related print commands, Print() and Print_(). Print_() prints an expression without a carriage return at the end, allowing you to compose more complex lines piecemeal.

Print() will print information about just about anything. Usually, you will be interested in printing text, which must be delineated by quotation marks. However, multiple expressions can be combined using the '+' symbol. Here are some examples:

```
define Start(p)
{
  Print("Mary had a little lamb")
  Print_("One two three " + 4 + " " + 5 + " " + 6)
  Print(".")  #Adds a . and linebreak to line

}
```

See demo06.pbl

This can be an easy way to record output, and is especially useful for debugging, but is not a great way to save data, because the same file will get overwritten the next time you run the program. Thus, for more permanent recording, use the FilePrint() and FilePrint_() functions. To use them, you need to first open a file using the FileOpenWrite() or FileOpenAppend() function (to overwrite or append to another file), then FilePrint works like Print, but takes two arguments, the file object and the text:

```
define Start(p)
{
  file <- FileOpenAppend("datalog.txt")
  FilePrint(file,"testing one")
  FilePrint(file,"testing two")
  FilePrint(file,"testing"+ "one"+"two"+"three")
}
```

See demo07.pbl

Response collection

An experiment usually needs to collect responses as well. There are a number of means to do this in PEBL, including accessing hardware, joysticks, networks, and so on. We will start by focusing on keyboard input, and eventually also look at mouse input. A number of functions exist to test and wait for keyboard input. One is WaitForListKeyPress(). To use it, you need to understand what a list is. A list is a series of data points, and is defined and displayed using [ ] characters. WaitForListKeyPress() takes as an argument a list of keyboard keys, and waits until one of them has been pressed, then returns the value that has been pressed.

Here is a short program that combines many of the lessons learned so far, illustrating response collection:

```
define Start(p)
{
  win <- MakeWindow()
  stim <- EasyLabel("Rate on a scale of 1 to 7 how good you feel today",
    400,300,win,18)
  Draw()
  resp <- WaitForListKeyPress(["1","2","3","4","5","6","7"])
  Print("response was: " + resp)
}
```

See demo08.pbl

Another important aspect of collecting a response is the time of the response. PEBL lets you poll the state a realtime clock that reports the number of milliseconds since PEBL started. You do this via the GetTime() function. You can use this to measure the absolute time a response occurred at, and thus to determine how long a response took. Notice that the time2-time1 is in parentheses, so PEBL isn't confused about appending text versus a real mathematical operation.

```
define Start(p)
{
   win <- MakeWindow()
   stim <- EasyLabel("Rate on a scale of 1 to 7 how good you feel today",
```

See demo09.pbl

```
                    400,300,win,18)
   Draw()
   time1 <- GetTime()
   resp <- WaitForListKeyPress(["1","2","3","4","5","6","7"])
   time2 <- GetTime()
   Print("response was: " + resp + ", which took " + (time2-time1)+" ms.")
}
```

Iteration

This is starting to look like an experiment! In a typical experiment, you'd probably want to run the same stimulus-response sequence multiple times. To do this in PEBL, you use one of the iteration methods. The simplest is the loop() command. If you have a list, you use the loop command to iterate over each element of the list, executing the code within the {} for each value.

See demo10.pbl

```
define Start(p)
{
   items <- [1,2,3,4,5,6,7,8,9,10]
   runningsum <- 0
   loop(i,items)
   {
     runningsum <- runningsum + i
     Print(i + " " + runningsum)
   }
}
```

The loop function gives you a lot of power and flexibility to create an experiment. Notice that on each iteration, the variable i is bound to a different value from the list items. However, you can still make things accrue over time and depend on what iteration you are on, like the runningsum variable does.

You'll probably want to put together all of the stimulus presentation and response collection into a function. This will also allow you to encapsulate, reuse, and replace the display code, and make the overall program logic easier to understand. To do this, define another function, specifying the arguments you want to pass into the function:

See demo11.pbl

```
define Start(p)
{
  win <- MakeWindow()
  label1 <- EasyLabel("Which do you prefer? Use left or right shift key.",
                      400,200,win,18)

  label2 <- EasyLabel("", 400,300,win,18)
  stim <- ["Red vs. Blue","Ford vs. Chevrolet",
           "Asparagus vs. Broccoli","Brick vs. Mortar",
           "Facebook vs. Twitter"]

  loop(i,stim)
   {
     resp <- Trial(label2,i)
     Print(i + "," + resp)
   }

   label1.text <- ""
   label2.text <- "Thank you for you opinions."
   Draw()
   WaitForAnyKeyPress()
}
```

```
define Trial(label,stimulus)
{
  label.text <- stimulus
  Draw()
  resp <- WaitForListKeyPress(["<lshift>","<rshift>"])
  label.text <- ""
  Draw()
  return resp
}
```

Now, we can easily replace the sequence of events in trial to make a different trial type, and it is fairly easy to see what the high-level sequence of events in the main experiment are.

Conditional Logic

Suppose you have two trial types you want to intermix. You need a way of deciding whether to run one block of code versus another. You do this with the if() argument. the if command evaluates whether an argument is true or false (actually, non-zero or zero), and only runs the associated block of code if it is true. You can append an else or an elseif() block to the end of an if block to get more complex conditional logic. To illustrate how to use this, we'll use a nested list and the Transpose function. The Transpose() function takes a nested list (a list-of-lists), and makes a new nested list, where the first sublist contains the first element of all the lists, the second sublist contains the second, and so on:

```
  trans <- Transpose([[1,2],[3,4],[5,6]])
  ##trans is [[1,3,5],[2,4,6]]
```

See demo12.pbl

```
define Start(p)
{
   win <- MakeWindow()
   label1 <- EasyLabel("Type the letter you see.", 400,200,win,18)
   label2 <- EasyLabel("", 400,300,win,18)
   #Type will specify whether we use lowercase or uppercase
   type <- [1,0, 1,1, 0,0, 0,1, 1,0]
   stim <- ["a","b","c","d","e","f","g","h","i","j"]
   trials <- Transpose([type,stim])

   trial <- 1
   loop(i,trials)
   {
     resp <- Trial(label2,i)
     Print(trial + "," + First(i) + "," +Second(i)+","+ resp)
     trial <- trial + 1
   }

  label1.text <- ""
  label2.text <- "Thank you."
  Draw()
  WaitForAnyKeyPress()
}

define Trial(label,stimulus)
{
  if(First(stimulus)==1)
   {
     lab <- Uppercase(Second(stimulus))
   }else {
     lab <- Second(stimulus)
   }
  label.text <- lab
  Draw()
  resp <- WaitForListKeyPress(["a","b","c","d","e","f","g","h","i","j"])
  label.text <- ""
  Draw()
  return resp
}
```

Together, the information on this page gives the capability to create simple experiments. Try one on your own:

Exercise

Using this final basic experiment, modify it to do one or more of the following:

- Add correct/incorrect feedback
- Collect response time
- Add a third independent variable that changes the x or y position of the label.
- Instead of manipulating upper/lower case, add <<< >>> or some other flanker characters.

Randomization and Experimental Design

So far, the basic functions we have covered in PEBL are not too different that what you will find in any computer language. Of course, the need to display and collect responses is central to PEBL, and so these are easier to use that in many programming languages. But where PEBL really saves time is its large library of functions that help perform standard tasks typically involved in designing experiments. These include randomization, counterbalancing, and layout, stimulus collection, and a rich graphical object library.

Random Numbers and Randomization

PEBL allows you to generate psuedo-random numbers, and will allow you to do randomization based on this random number generator. You should realize that these numbers are not truly random. They are a deterministic sequence that has properties of random numbers. By default, the element of the series is set at the beginning of each experiment by consulting the clock time. But, for complex randomization and debugging purposes, you can choose to set it by some other number, using the SeedRNG() function.

The Random() function generates a decimal number between 0 and 1. Also, RandomNormal, RandomDiscrete, RandomUniform, RandomExponential, RandomLogistic, RandomBinomial, and RandomBernoulli are available for use.

For many experimental purposes you can use one of these random functions directly to help reduce predictability of your task. For example, you may add random delays between trials by sampling a value with the RandomUniform() and using the Wait() function to wait that many ms. Or you may change the x,y coordinate of a target in a bivariate normal distribution using two samples from the RandomNormal() distribution.

But oftentimes, sampling a stimulus class or other independent variable on each trial is not the right way to go, because you'll end up with an unbalanced design and potentially a confound. In these cases, Shuffle() is your friend. Shuffle mixes up the order of a list, and returns the new list for you to operate with. Together with a number of other list manipulation functions like SubList, RepeatList, Flatten, and Repeat, this can be very powerful.

Of course, both are sometimes needed. Suppose you want to do some type of visual search task, where you are searching for either a T or an E in a field of Ls, Is, and Fs. You want the same number of T and E files, but you really don't care too much where each of the other points are in the field. Here is a short demo experiment that does this.

See demo13.pbl

```
define Start(p)
{
  gWin <- MakeWindow("black")  ##Variables that start with a g are global!
  gSleepEasy <- 1

  targs <- ["T","O"]
  foils <- ["L","I","F","E"]
  numFoils <- 50
  numstim  <- 5  #number of T or F trials.

  stimsequence <- Shuffle(RepeatList(targs,numstim))  #Create a list of Ts and Es
  trial <- 1
  loop(i,stimsequence)
    {
      out <- Trial(i,foils,numfoils)
      Print(trial + " " + out)
      trial <- trial + 1
    }
}

define Trial(stim,foils,numfoils)
{
  baseX <- 400
  baseY <- 300
  sd <- 80      ##standard distribution of spread

  ##Create a list of foils:
  foilStim <- SampleNWithReplacement(foils,numfoils)

  ##Add them to the center, in a random normal bivariate distribution
   foils  <- []
   loop(i, foilStim)
    {
      tmp <- EasyLabel(i,RandomNormal(baseX,sd), RandomNormal(baseY, sd),gWin,22)
      foils <- Append(foils,tmp)  ## We need to hold onto each foil
                                  ## or it will disappear.
     #Don't draw here because we want to display them all at once.
    }
   targ  <- EasyLabel(stim,RandomNormal(baseX,sd),
                      RandomNormal(baseY, sd),gWin,22)
   footer <- EasyLabel("Press O or T when you see an O or T",baseX, 550,gWin,30)
   Draw()
   time1 <- GetTime()
   resp <- WaitForListKeyPress(["o","t"])
   time2 <- GetTime()
   tx <- targ.x
   ty <- targ.y

   loop(i,foils)
    {
      RemoveObject(i,gWin)  ##Remove the foils
    }

    corr <- (Uppercase(resp) == Uppercase(stim))
    if(corr)
```

```
      {
       footer.text <- "Correct"
      } else {
       footer.text <- "Incorrect"
      }
     Draw()
     Wait(500)
     RemoveObject(footer,gWin)
     RemoveObject(targ,gWin)
     Draw()
    return [resp, corr, tx, ty, (time2-time1)]
}
```

This shows how we may want to control the stimulus sequence sampling using Shuffle, but individual response locations using a Random() function.

Experimental Design

Simple experimental design can usually be done using Shuffle, Transpose, Repeat, and related functions. However, a number of prepackaged functions are available that can help do this more easily. These include:

- Sequence <start> <end> <step>
- ChooseN <list> <n>
- Sample <list>
- SampleN <list> <n>
- SampleNWithReplacement <list> <n>
- DesignLatinSquare <list1> <list2>
- LatinSquare <list>
- DesignGrecoLatinSquare <list1> <list2> <list3>
- DesignBalancedSampling <list> <number>
    - This is an approximate partial factorial design
- DesignFullCounterbalance <list1> <list2>
- CrossFactorWithoutDuplicates <list>
    - This makes a set of comparisons of all levels of list with each other level, excluding x-x matches.

Exercise

- Step 1. Add more than two levels to the stimulus (instead of O and T, choose several other letters)
- Step 2. Choose another independent variable with two or more levels to add to the visual search task. We already have stimulus identity. We could add upper/lower case, fontsize (either of stimulus or foils), standard deviation of normal distribution, stimulus color, or other similar factors. Implement a version of the task that presents all combinations of both factors, creating the stimulus sequence with Repeat, Transpose, Shuffle, Flatten, etc.
- Step 3. Create the design using LatinSquare or DesignLatinSquare

Graphics

PEBL has a number of ways to create and display stimuli. These include text, images, and a bunch of shapes. Suppose we want to modify the visual search task created in Randomization and Experimental Design to:

1. Present instructions
2. Draw a black rectangle to specify the background of the region that stimuli will appear

3. Use red and blue circles and squares instead of letters as stimuli.
4. Restrict stimuli so they don't overlap

The following lessons will cover those four goals.

Adding Instructions

Instructions are best displayed in a TextBox, rather than a Label, because labels are just a single line of text. To make a text box, you must specify a font, a background color, and a horizontal and vertical size in pixels. We will use a shortcut function, EasyTextBox(), to do this. We will start by making a textbox with no text in it:

See demo14.pbl

```
define Start(p)
{
 win <- MakeWindow()  ##assume an 800 x 600
 inst <- EasyTextBox("",100,50,win,22,600,200)
 inst.text <- "These are the instructions. Press any key to begin."
 Draw()
 WaitForAnyKeyPress()
 Hide(inst)
 Draw()
 Wait(500)  #wait a little bit before closing
}
```

A textbox can be very useful for displaying text and getting free-form text entry. In the above example, we showed instructions, then used Hide() to make the text box disappear so we can continue. You could alternately get rid of it by using RemoveObject(), or by assigning something else to the inst variable, but by Hideing it, you can use it again later for inter-block instructions or debriefing.

Adding a rectangular field

Let's say we want all stimuli to appear within a box, and that box should be black. One reason to do this is if you are running the experiment on multiple different computers with different screens; you can make sure everybody is focusing on the same region even if they have a wider or taller screen.

The Rectangle(), Square() and Polygon() and Canvas objects are all possibilities here. I'll use Rectangle(). When you create a Rectangle, it does not automatically appear on the screen--you have to use the AddObject() function to add it to the drawing context you want to put it on (i.e., the screen). You need to do this because you can often add objects to other objects to create a compound object, and so it won't automatically appear on the screen. Furthermore, you can pre-load images and other objects and then just add them to the screen when you need them.

When multiple objects are added to a window, they get drawn in the order in which they are added. An object added later will be drawn above earlier objects. Thus, if you want to create a background object and add it to the window before you add anything else.

The Rectangle() function also require you to specify a color. Colors are special objects that are created, either by name or RGB coordinates. There are close to 800 distinct color names you can choos e from, which are shown in the back of the manual.

See demo15.pbl

```
define Start(p)
{

  win <- MakeWindow()

  blue <- MakeColor("navyblue")
  black <- MakeColor("black")

  ##To make a rectangle, specify centerx, centery, width, height,color,
  ##and whether it should be filled:
  rect <- Rectangle(400,300,700,500,black,1)
  circ1 <- Circle(200,200,30,blue,1)
  circ2 <- Circle(200,200,35,blue,0)

  ##now, add all the objects in the order you want:
   AddObject(rect,win)
   AddObject(circ1,win)
   AddObject(circ2,win)

   Draw()
   WaitForAnyKeyPress()
}
```

Using Shapes for Stimuli

Instead of creating the stimuli from letters, lets make this a conjunction search; the field of distractors will red circles and blue squares, and the target will be a red square or a blue circle. To do this, we'll modify Trial() a little. We'll start by modifying the arguments:

```
define Trial(stim, foils, numfoils)
{
```

We'll use the following code to represent stimuli of different shape/color configurations:

- 1= red circle
- 2= red square
- 3= blue circle
- 4= blue square

Then, we'll make a function called MakeStim() that takes the code and returns a stimulus of the appropriate color:

See demo16.pbl

```
define MakeStim(code)
{
  if(code < 3)
  {
    color <- MakeColor("red")
  } else {
    color <- MakeColor("blue")
  }
 if(code == 1 or code == 3)
  {
   stim <- Circle(0,0,10,color,1)
  } else {
   stim <- Square(0,0,10,color,1)
  }
  return stim
}
```

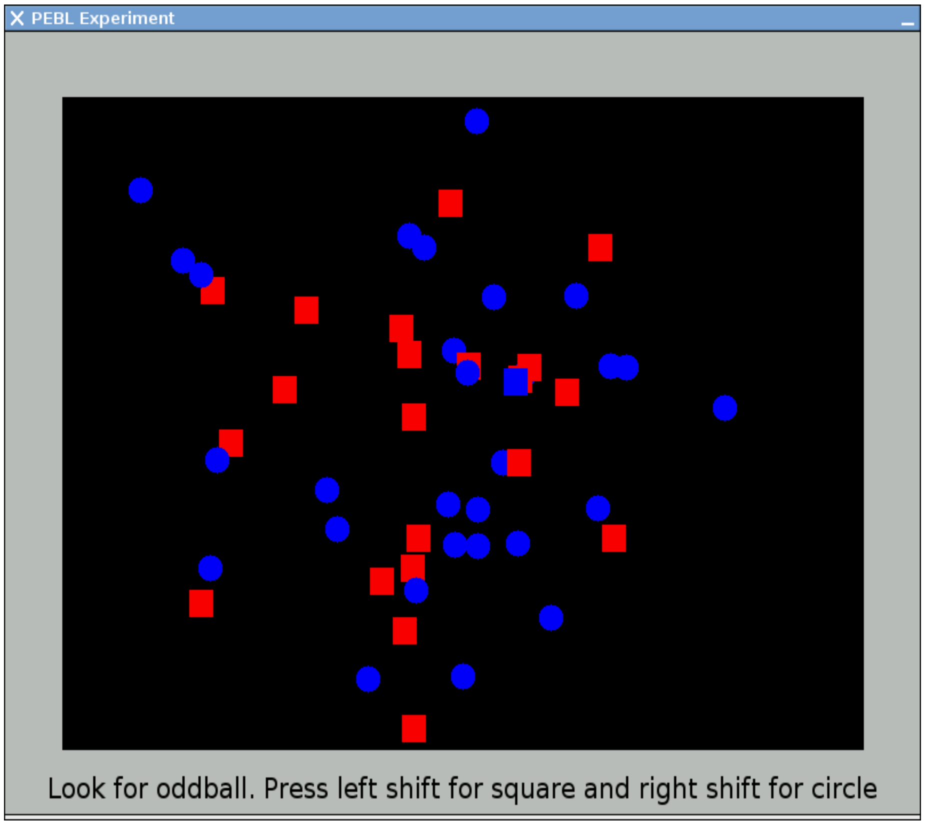
This way, we can choose a stimulus, choose the foils, and the number of foils, and have a lot of flexibility. Continuing on in the main Trial() function, we use the same code until we get to the loop that creates the foils. Here, we call the MakeStim() function instead of creating a label. Note that inside the MakeStim() function, we could create arbitrarily complicated stimuli, or even load images with the LoadImage() function. Because the basics of adding stimuli to a window, moving them around, showing and hiding will apply equally to all graphical objects, you can easily replace this.

The only other change is to figure out whether a response is correct, and fix a few things in the Start() function, taking care to add the black rectangle:

```
define Start(p)
{
  gWin <- MakeWindow()  ##Variables that start with a g are global!
  gSleepEasy <- 1
  rect <- Rectangle(400,300,700,500,MakeColor("black"),1)
  AddObject(rect,gWin)


  targs <- [1,4]
  foils <- [2,3]
  numFoils <- 50
  numstim  <- 5  #number of T or F trials.

  stimsequence <- Shuffle(RepeatList(targs,numstim))  #Create a list of Ts and Es
  trial <- 1
  loop(i,stimsequence)
    {
      out <- Trial(i,foils,numfoils)
      Print(trial + " " + out)
      trial <- trial + 1
    }
}
```

```
define Trial(stim,foils,numfoils)
{
  baseX <- 400
  baseY <- 300
  sd <- 120      ##standard distribution of spread

  ##Create a list of foils:
  foilStim <- SampleNWithReplacement(foils,numfoils)


  ##Add them to the center, in a random normal bivariate distribution
   foils  <- []
   loop(i, foilStim)
    {
      tmp <- Makestim(i)
      AddObject(tmp,gWin)
      Move(tmp, RandomNormal(baseX,sd), RandomNormal(baseY, sd))
      foils <- Append(foils,tmp)  ## We need to hold onto each foil
                                  ## or it will disappear.
    }
   targ  <- MakeStim(stim)
   AddObject(targ,gWin)
   Move(targ,RandomNormal(baseX,sd), RandomNormal(baseY, sd))

   footer <- EasyLabel("Look for oddball.”+
    “Press left shift for square and right shift for circle",baseX, 580,gWin,22)
   Draw()
   time1 <- GetTime()
   resp <- WaitForListKeyPress(["<lshift>","<rshift>"])
   time2 <- GetTime()
   tx <- targ.x
   ty <- targ.y

     corr <- ((stim==2 or stim==4) and resp == "<rshift>") or
               ((stim==1 or stim==3) and resp == "<lshift>")
   loop(i,foils)
    {
      RemoveObject(i,gWin)  ##Remove the foils
    }
    Draw()
    Wait(500)
    RemoveObject(footer,gWin)
    RemoveObject(targ,gWin)
    Draw()
   return [resp, corr, tx, ty, (time2-time1)]
}
```

The result looks like this:



Look for oddball. Press left shift for square and right shift for circle

Using an improved Layout Function

The normal distribution we happened to choose can sometimes spill off the inner frame, and they can easily overlap, at times even obscuring the target stimulus. PEBL has a very nice layout function called NonOverlapLayout(<xmin,xmax,ymin,ymax,tol,num>), which will produce a set of X,Y coordinates that are randomly dispersed in a space, but will have a minimum distance between each coordinate. Of course you can specify a layout that is impossible, like trying to cram 1000 points in a 10 x 10 grid and forcing a minimum spacing of five between each point. The function will typically fail gracefully in these situations, and produce a pretty good layout with minimal overlaps.

See demo17.pbl

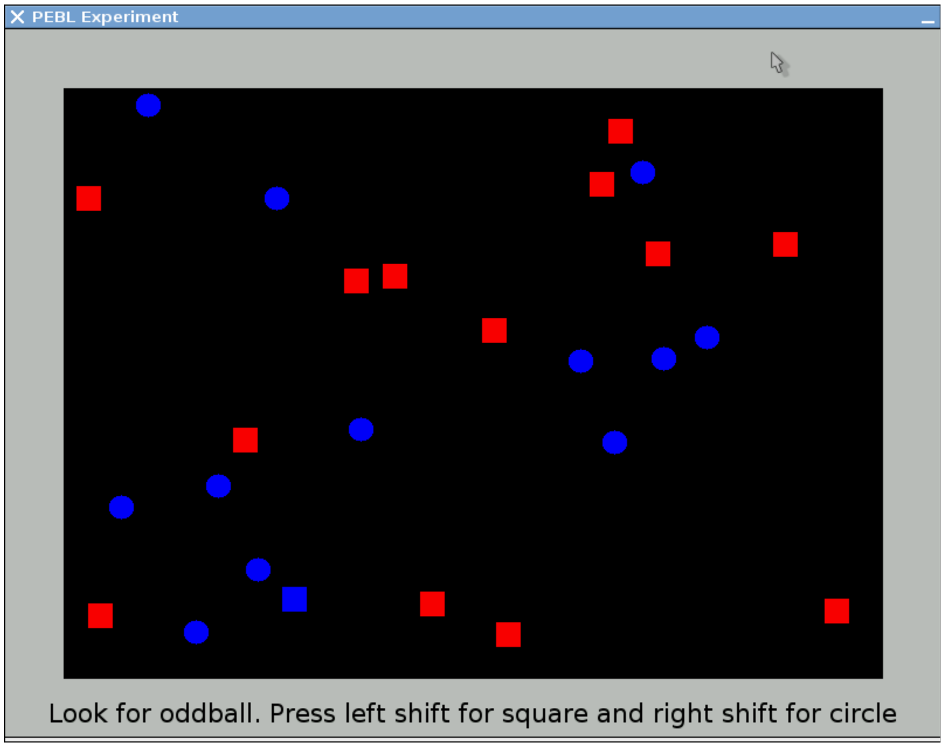
We'll use NonOverlapLayout with an x and y range that are 11 pixels smaller than the black rectangle, so that when our stimuli are presented, they won't go outside the box. The basic strategy will be to create all the stimuli as we did before, but instead of adding them to the window and scattering them around, make a second pass with the layout points. The following code replaces part of the Trial function:

```
define Trial(stim,foils,numfoils)
{
  baseX <- 400
  baseY <- 300

  ##Create a list of foils:
  foilStim <- SampleNWithReplacement(foils,numfoils)
```

```
##Add them to the center, in a random normal bivariate distribution
foils  <- []
loop(i, foilStim)
 {
   tmp <- Makestim(i)
   AddObject(tmp,gWin)
   foils <- Append(foils,tmp)  ## We need to hold onto each foil
                               ## or it will disappear.
 }
targ  <- MakeStim(stim)

numpoints <- numfoils+1
points <- NonOverlapLayout(60,740,60,540, 21,numpoints)

targpoint <- First(points)
foilpoints <- SubList(points,2,Length(points))
loop(i,Transpose([foilpoints,foils]))
 {
  pos <- First(i)
  shape <- Second(i)
  shape.x <- First(pos)
  shape.y <- Second(pos)
 }
 AddObject(targ,gWin)
 Move(targ,First(targpoint),Second(targpoint))



footer <- EasyLabel("Look for oddball. Press left shift for square “ +
                     “and right shift for circle",baseX, 580,gWin,22)
Draw()
time1 <- GetTime()
resp <- WaitForListKeyPress(["<lshift>","<rshift>"])
time2 <- GetTime()
tx <- targ.x
ty <- targ.y


  corr <- ((stim==2 or stim==4) and resp == "<lshift>") or
            ((stim==1 or stim==3) and resp == "<rshift>")

loop(i,foils)
 {
   RemoveObject(i,gWin)  ##Remove the foils
 }

 if(corr)
  {
   footer.text <- "Correct"
  }else{
   footer.text <- "Incorrect"
  }
 Draw()
 Wait(500)
 RemoveObject(footer,gWin)
```

```
    RemoveObject(targ,gWin)
    Draw()
   return [resp, corr, tx, ty, (time2-time1)]
}
```

The result is:



Look for oddball. Press left shift for square and right shift for circle

.

Using and modifying tests from the PEBL Test Battery

Review of Available Tests

There are currently about 60 tests available in the PEBL Test Battery (Follow link for a partial list with details.) The file launcher allows you to browse all the tests, and has brief descriptions and screenshots of each test.

- Many are work-a-likes of prominent test paradigms.
- Some tests in the battery are found nowhere else.
- Many tests in the battery are the only free version widely available
- Many have sensible defaults that can be used out-of-the-box to conduct a study
- Most contain parameters that can be tweaked to suit the needs of an experimenter
- All can be modified, shared, improved, and used without license costs.
- Most tests can be readily translated into other languages.

Modifying A Test: Some Typical Requirements

What follows are a few typical ways in which users have wanted to modify the PEBL Tests

Changing Screen Size

By default, PEBL will run in a screen mode of 800x600. Alternate screen modes can be selected at start-up, and additionally the horizontal and vertical resolution can be hard-coded into the script. The problem is that every computer lab has a different monitor set-up, with different video cards and different available resolutions. 800x600 is pretty widely accepted, but on many laptops and widescreen monitors, it leads to letterboxing on the sides, or worse yet, stretching of the aspect ratio. You need to be able to adapt a script to your computer set-up.

When PEBL launches, if a screen resolution was given by the launcher, it will load those values into the global variables gVideoWidth and gVideoHeight. However, these values are changeable. You can set them prior to creating a window, to specify a new resolution. If you want a 1440x900 display, do:

```
gVideoWidth <- 1440
gVideoHeight <- 900
```

See demo18.pbl

at the beginning of the Start() function.

These will impact the display if changed before the MakeWindow() function is called. When called, the MakeWindow() function checks to see whether the screen resolution is supported, and if not, chooses the 'best' one, which is probably the current resolution. After that, if you print out the values of gVideoWidth and gVideoHeight, it will be the actual values, which may differ from the ones you requested.

You may need to do this to get a decent full-screen mode experiment. It does not mean that the experiment will show up well in the new resolution. You may then need to change font sizes, stimulus locations, and other things to make everything fit right. Typically, stimuli and objects are fit onto the screen relative to the screen size: the center is [gVideWidth/2, gVideoHeight/2]; or they might be placed 50 pixels from the top or bottom, etc., and so small changes in the screen resolution may have essentially no impact.

Changing stimulus size

Sometimes the display is too small. This can be common when running an fMRI experiment, where the screen must be reflected or projected onto a tiny viewport. One way to do this is to change the font size of the text label you want to use. If the label was created using EasyLabel, the fontface is the last number in the list. If it was created with Makelabel, you need to find the font object, and change it there. You can always create a new font object and set the font of the label to that object, like this:

See demo19.pbl

```
lab  <- EasyLabel("one",gVideoWidth/2,gVideoHeight/2,gWin,25)
newfont <- MakeFont(gPEBLBaseFont,0,55,MakeColor("white"),MakeColor("black"),0)
lab.font <- newfont
```

Another strategy is to stretch the target object using the .zoomx and .zoomy properties. This will not always be a good idea, because you can see scaling artifacts, but it is a really easy fix:

```
lab.zoomX <- 1.5  #Increase size by 1.5 times
lab.zoomY <- 1.5
```

Adding or Changing Instructions

It is usually pretty easy to find where the instructions are located so you can change them if you want. Just look for the text of the instructions. Typically, these will be in a textbox, which will be created and displayed near the beginning of an experiment. Sometimes the textbox is created in a function called something like Init() which will be called near the beginning of the experiment and will take care of a bunch of typical bookkeeping things. Other times, the instructions will be located in a function called GetStrings(), which reads in different instructions for different languages. Still other times, it will be read in from a text file (again for translation reasons), and should be located in a file in the translations\ directory.

Oftentimes, for multi-screen instructions, a textbox is used. Assuming the textbox tb has already been created, the sequence of steps might look like:

See demo20.pbl

```
tb.text <- "These are the first set of instructions"
Draw()
WaitForAnyKeyPress()
tb.text <- "These are the second set of instructions"
Draw()
WaitForAnyKeyPress()
Hide(tb)
Draw()
```

If you want instruction panes to be triggered with a mouse buttonpress, you can use WaitForDownClick() instead of WaitForAnyKeyPress(). In such cases, just add three additional lines to the sequence.

A slightly simpler way to do this is to use the MessageBox function. It displays the given text and has an OK button on the bottom:

```
MessageBox("These are the first instructions.",win)
MessageBox("These are the second instructions.",win)
```

Add a line like this anywhere you want to add further instruction.

Collecting and Aggregating Additional Data

It is often convenient to aggregate and report summary statistics over conditions for each participant, either for feedback or for saving to an aggregate file that gets analyzed. The easiest way to do this is to create a set of global list variables that collect the data. Consider the following snippet of code:

See demo21.pbl

```
gRTS <- []
 gAcc <- []
 conds <- Shuffle([1,1,1,2,2,2,3,3,3,4,4,4])
 loop(i,conds)
 {
   itext <- i+""
   label.text <- itext
   Draw()
   time1 <- GetTime()
   resp <- WaitForListKeyPress(["1","2","3","4"])
   time2 <- GetTime()

    label.text  <- ""
    Draw()
    gRTs <- Append(gRTS, (time2-time1))
    gAcc <- Append(gAcc, resp == itext)
    Wait(200)
 }
```

In the example above, gRTS and gAcc did not need to be global, but for a more complex experiment that moved the logic inside the loop into another function (maybe called Trial()), the same code will work because the global variables will be available inside the Trial() function. At the end of this sequence, there would be a sequence of response times (in ms) and 1/0 numerals indicating correct or incorrect. You can create aggregate values of accuracy and RT for each condition using the Match and Filter functions (or use SummaryStats).

```
 aggRT <- []
 aggAcc <- []
 loop(i, [1,2,3,4])
  {
    match <- Match(conds,i)
    subRT <- Filter(gRTs,match)
    subAcc <- Filter(gAcc,match)
    aggRT <- Append(aggRT, Mean(subRT))
    aggAcc <- Append(aggAcc,Mean(subAcc))
  }
##Now, aggRT and aggACC contain mean RT and accuracy for each stimulus condition.
```

Saving Pooled Data Files

You may wish to save the aggregated values to a pooled data file for later summary statistics, even if a detailed trial-by-trial data log were saved. Suppose you wanted to do this for aggRT and aggACC. You can wait until the end of the session to do this. Let's start be seeing if the data file actually exists, so that if it does not, we will add a header to it:

See demo22.pbl

```
 if(not FileExists("pooled-data.csv"))
  {
    pooled <- FileOpenAppend("pooled-data.csv")
    FilePrint(pooled,"subnum,timestamp,acc1,acc2,acc3,acc4,rt1,rt2,rt3,rt4")
  } else {
    pooled <- FileOpenAppend("pooled-data.csv")
  }
  aggAcc <- [1,2,3,4]
  aggRT <- [200,300,400,600]

 FilePrint_(pooled,gSubNum+","+TimeStamp())
  loop(i,aggAcc)
  {
     FilePrint_(pooled,","+i)
  }
  loop(i,aggRT)
  {
     FilePrint_(pooled,","+i)
  }
  FilePrint(pooled,"") ##add end-of-line
  FileClose(pooled)
}
```

Translating text

It is relatively simple to translate most tests. Most tests have no built-in translation, and in these case you simply need to open the .pbl file in a text editor, look for the text that wants to be translated (almost always enclosed in quotation marks), and translate it. Foreign scripts that use extended letter sets need to be saved in UTF-8 format (not generic unicode or UTF-16), but should usually work.

Some tests are set up for more automatic translation. PEBL allows you to specify a two-character language code at launch, which will be bound to the value of gLanguage. This lets the script author do different things based on the language of choice. Furthermore for CJK language (chinese, japanese, korean), it will select a different default font. In some of the more popular tests, there is a function near the end called GetStrings which assigns a set of global variables different values depending on the specified language, allowing one to present a translated version of the test. At other times, these strings are read in from another file, usually stored in a translations/ directory. If the test works by reading in strings from a file, all you need to do is make a copy of the english version of the file, rename it replacing the '-en' code with your own two-letter code, and translate that file. Be sure to observe hard returns; each line in that file is read in as a different variable.

If you translate a file, be sure to send a copy to the pebl-list, so it can be incorporated in future versions of the test battery.

Developing New Tests

The previous lessons covered many of the details of creating and modifying an experiment, from the nuts-and-bolts perspective. Here, we will discuss some higher-level strategies for developing a test.

The PEBL Programming Philosophy

PEBL's basic notion is that it should be a programming language targeted toward developing, sharing, and modifying experiments. At the heart of most experiments is a sequence of events that occur repeatedly. In contrast, developing more complex desktop applications, such as a word processor or image editor, has entirely different requirements. A typical GUI program differs from an experiment in many ways:

- Has no start and no end
- Lets the user explore the different functions, so they can compose arbitrarily complex sequences of operations
- Is expected to interact with other software while in use, or at least allow the user to switch between applications
- Is expected to run for hours, and end only when the user wants it to
- Looks and feels like the rest of the software on your system.

These are all either somewhat or completely at odds with the needs of experimental design, and tools designed to support those types of programs are often targeted to the wrong task. For example, something like Visual Basic can be very easy to make a GUI application for, but because of its event-based organization, it can be frustrating and confusing to create a linear experiment. This is partly because code that is associated with some particular response can be hidden away inside a button.

Almost every real experiments require:

- Display elements
- Instructions
- Response collection (both time and responses)
- Randomization and experimental design
- Logging of data

A number of experiment programming tools exist that are tightly restrict the organization of your experiment. These might be very object-oriented, with where an experiment is an object whose properties you adapt (setting properties for instructions, the stimulus, a randomization, the SOA, data responses, etc.). PEBL does not restrict you to such a limited vocabulary, for several reasons. First, using such a scheme abstracts the notion of a sequence of events, making it difficult to understand what is going on, and difficult to debug and verify the experiment is correct. Second, the model begins to fail on moderately complex designs, adaptive designs, and experiments that do not fit easily into the trial-block-randomization notion. Third, although object-based experiment designs may have the ability to be written tersely, the results I've seen are typically more verbose than an equivalent PEBL experiment. The first iterations of nearly all of the tests in the PEBL Test Battery were each written in 200-300 lines of PEBL code, which is just a few screens. This tends to increase as more options were added, instructions were expanded, translations were added, and real-time data scoring was expanded, but the core functionality is really pretty simple.

Of course, PEBL currently offers a survey experiment that can be controlled via a spreadsheet, and in future versions, we may attempt to create a simpler data-based experiment that can be more easily controlled in similar ways.

Development Strategy

The high-level nuts and bolts of writing an experiment are largely similar for many experiments. To begin creating a new experiment, start at the level of a 'trial'. It is usually smartest to have randomization of critical variables be done outside the Trial level, and the proper condition and stimulus be passed into a Trial() function. This gives more better control over randomizing the whole study, rather than just a single trial at a time. As described in earlier lessons, if you want 80/100 trials to be in one condition, it is better to create a list with 80 1s and 20 2s, then Shuffle() it and loop over it, than to call Trial 100 times and have trial randomly determine (with 80% probability) whether it should be in that condition.

Create Trial Function

So, work on a Trial function first. Identify the inputs. In most cases it is also easiest to create labels or images during the trial, rather than creating them ahead of time and simply hiding or showing them during the trial. Of course, after you write the trial function, you can determine if there is a problem loading stimuli and pre-cache if necessary. Here is a bare-bones trial function, called by a barebones Start function:

See demo23.pbl

```
define Start(p)
{
  gWin <- MakeWindow()
  Print(Trial(1))
  Print(Trial(2))

}
define Trial(cond)
{
   time <- GetTime()

  return [cond, time]
}
```

Create Stimulus

Of course, the cond variable holds something important about the stimulus you want to create. It might be the name of an image file, or a word or number label, or a complex list that contain a set of stimulus locations and colors, and so on. But don't be hesitant to create an image or a label at the beginning of each trial, then use AddObject and RemoveObject to manage it. Let's suppose in this experiment, cond is a text label that is the root for an image you have created and stored in the 'stim' subdirectory.

See demo24.pbl

```
define Trial(cond)
{
   time <- GetTime()

   stim <- MakeImage("stim/"+cond+".png")
   Move(stim,gVideoWidth/2,gVideoHeight/2)
   AddObject(stim,gWin)

   Draw()
   Wait(1500)
   RemoveObject(stim,gWin)
   return [cond, time]
}
```

In fact, unless you make stim a global variable (by starting it with a lower-case g), it will automatically unload from the window at the end of the trial, so the RemoveObject line is not really necessary. If instead you want the stimulus to be a text label, create a label based on the information in cond:

```
  stim <- EasyLabel(cond,gVideoWidth/2,gVideoheight/2,gWin,44)

 ##No need to move and add this stimulus)
 Or perhaps a shape, whose color or size or position is controlled by cond:

  stim <- Circle(gVideoWidth/2,gVideoHeight/2, cond,MakeColor("blue"),1)
```

This step can, of course, be arbitrarily complex, might require creating additional functions that actually create the stimulus, might require additional variables passed into the Trial() function, and so on.

Collect Response

There are many ways to collect response. In this example, we will stick with a multi-option response collection function called WaitForListKeyPress(). Currently, the response is not time-tagged, and so you need to record start and stop times yourself using the GetTime() function.

See demo25.pbl

```
define Trial(cond)
{
   time <- GetTime()
#
   stim <- MakeImage("stim/"+cond+".png")
   Move(stim,gVideoWidth/2,gVideoHeight/2)
   AddObject(stim,gWin)
#
   Draw()
   time1 <- GetTime()
   resp <- WaitForListKeyPress(["<lshift>","<rshift>"])
   time2 <- GetTime()

   RemoveObject(stim,gWin)
   return [cond, resp, time, (time2-time1)]
}
```

Scoring response

It is usually easiest in the long run to score trials within the trial, rather than doing it later using excel or a stats package. In the above example, maybe we want a response of left-shift for conditions 1 and 2, and a response of right-shift for conditions 3 and 4. It is often easiest to write a function named something like ScoreTrial that takes the response and the condition and returns a 0 or 1 depending on whether the trial is correct. ScoreTrial in this case could look something like:

See demo26.pbl

```
define ScoreTrial(resp,cond)
{
   if(cond==1 or cond==2)
    {
      corr <-  (resp == "<lshift>")
    } else {
      corr <-  (resp == "<rshift>")
    }
 return corr
}
```

You can also use a trick like this:

```
define ScoreTrial(resp,cond)
{
   map <- ["<lshift>","<lshift>","<rshift>","<rshift>"]
   corresp  <- Nth(map, ToInteger(cond))
   corr <- corresp == resp
   return corr
}
```

Using a scoretrial function, the trial function would look like this:

```
define Trial(cond)
{
   time <- GetTime()
#
   stim <- MakeImage("stim/"+cond+".png")
   Move(stim,gVideoWidth/2,gVideoHeight/2)
   AddObject(stim,gWin)
#
   Draw()
   time1 <- GetTime()
   resp <- WaitForListKeyPress(["<lshift>","<rshift>"])
   time2 <- GetTime()
   corr <- ScoreTrial(resp,cond)
   RemoveObject(stim,gWin)
   return [cond, resp, corr, time, (time2-time1)]
}
```

Experiment-Level

Once you have created a credible Trial function, then start working on higher-level aspects like randomization, instructions, data output, These happen in most experiments, and a basic template is here:

See demo27.pbl

```
define Start(p)
{
  gWin <- MakeWindow("black")

  ##Open a data file:
  gFileOut <- FileOpenAppend("exp1-"+gSubNum+".csv")
  FilePrint(gFileOut,"subnum,trial,time,cond,noresp,corr,rt")
  gSleepEasy <- 1  ##avoid busy-waits, if you want

  #Display instructions:

   MessageBox("Instructions here",gWin)

  #Create and randomize stimuli:
  conds <- Shuffle(RepeatList([1,2,3,4],4))

  trialnum <- 1  ##Keep track of a trial number
  loop(i, conds)
  {
    time1 <- GetTime()  #Keep track of when the trial started
    out <- Trial(i) ##Do the trial!
    FilePrint(gFileOut,gSubNum+","+i+","+time1+","+out)
    trialnum <- trialnum+ 1
  }

  MessageBox("Thank you for participating in the study.",gWin)
}
```

Experiment Template

Together, these steps are needed for many experiments. Below is a basic template that can be adapted as needed. Here, a little bit of content is filled in so it is an actual experiment, which can be replaced to suit your needs.

See demo28.pbl

```
define Start(p)
{
  gWin <- MakeWindow("black")
  ##Open a data file:
  gFileOut <- FileOpenAppend("exp1-"+gSubNum+".csv")
  FilePrint(gFileOut,"subnum,trial,time,cond,noresp,corr,rt")
  gSleepEasy <- 1

  #Display instructions:
   MessageBox("In this experiment, you are to hit the left or right shift key when
you see a stimulus with arrows pointing either direction. For arrows pointing in
both directions, make no response.

          Left Shift         Do Not Respond       Right Shift
           <  <  <              <  +  >               > > >

Press any key to continue.",gWin)

  #Create and randomize stimuli:
  conds <- Shuffle(RepeatList([1,2,3],5))

  trialnum <- 1  ##Keep track of a trial number
  loop(i, conds)
  {
    time1 <- GetTime()  #Keep track of when the trial started
    out <- Trial(i) ##Do the trial!
    FilePrint(gFileOut,gSubNum+","+i+","+time1+","+First(out)+","+
              Second(out)+","+Third(out)+","+Fourth(out))
    trialnum <- trialnum+ 1
  }

  MessageBox("Thank you for participating in the study.",gWin)
}


##Let's do a basic 3-alternative forced choice. stimulus 1 requires left,
## stimulus 2 requires right, stimulus 3 requires no response
define Trial(cond)
{
   stimuli <- ["< < <","> > >","< + >"]
   stim <- Nth(stimuli,cond)

   lab <- EasyLabel("+",gVideoWidth/2,gVideoHeight/2,gWin,50)
   Draw()
   Wait(500)
   lab.text <- stim
   Draw()
   time0 <- GetTime()
   resp <- WaitForListKeyPressWithTimeOut(["<lshift>","<rshift>"],1000,1)
   time1 <- GetTime()

   score <- ScoreTrial(resp,cond)
   corr <- First(score)
   noresp <- Second(score)

   if(corr)   ##Give feedback
    {
       lab.text <- "CORRECT"
    } else {
       lab.text <- "INCORRECT"
    }
   Draw()
   Wait(400)

  RemoveObject(lab,gWin)
  return [cond, noresp,corr, (time1-time0)]
}

define ScoreTrial(resp,cond)
{
   #Score the trial:
   if(IsList(resp))
    {
      noresp <- 1
    } else {
      noresp <- 0
    }

  if(cond==1)
   {
     corr <- (resp == "<lshift>")
   } elseif(cond==2)
   {
     corr <- (resp == "<rshift>")
   } else {
     corr <- noresp
   }

   return [corr,noresp]
}
```
This took about 80 lines of code. To adapt it to your own experiment you basically need to change the instructions, create a condition sequence you care about, and adapt the Trial() function to display the stimulus, accept the correct response, score the response, and return the relevant data to save out to the data file.

Exercise

Adapt the above experiment to create a new experiment of your own design.

Beyond the Laboratory

Small-scale applications for data coding, recording field data

I've used PEBL many times to create small-scale logging programs to code or analyze field data.

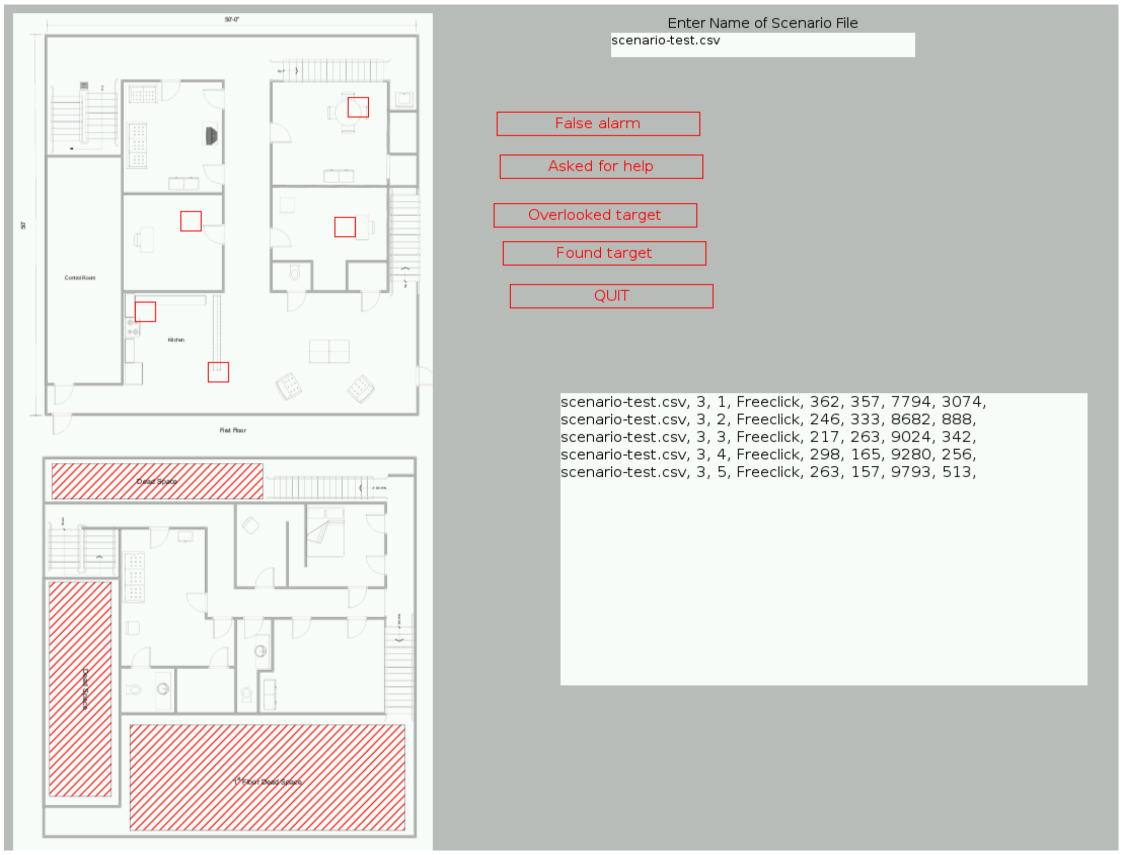
Examples:

Search Logger

In a study I did, we had people searching a building for suspicious materials. The building was instrumented with cameras allowing us to record and view the location of the searchers, but we wanted a detailed record of individual search paths. I created the logger application (below) to do this:

The logger read in an image that was a map of the building, and a .csv file containing information about the placement of labeled buttons and map hotzones (locations of doors and hidden objects). The logger then had to follow the searcher around, clicking on their current location with the mouse, and clicking buttons to log specific events. Whenever the mouse was clicked, a record was made of that along with a timestamp, so that the events could later be registered with the original data stream.

Such a logger could be used for data coding of a number of complex field tasks to do real-time or archival analysis of activities and communications.

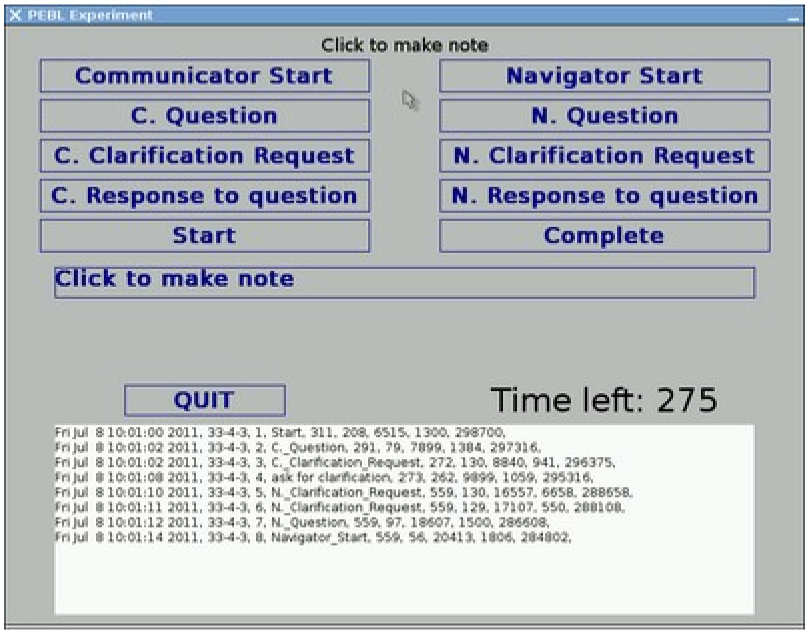


```
scenario-test.csv, 3, 1, Freeclick, 362, 357, 7794, 3074,
scenario-test.csv, 3, 2, Freeclick, 246, 333, 8682, 888,
scenario-test.csv, 3, 3, Freeclick, 217, 263, 9024, 342,
scenario-test.csv, 3, 4, Freeclick, 298, 165, 9280, 256,
scenario-test.csv, 3, 5, Freeclick, 263, 157, 9793, 513,
```

Communication Tool

Another study I did had people trying to communicate a route on a map. The actual map drawing was done using PEBL, but we also wanted to do some fast and dirty communication analysis: coding turns in speech, and coding speech according to a few specific categories. A screenshot is here:

See demo29.pbl



```
Fri Jul  8 10:01:00 2011, 33-4-3, 1, Start, 311, 208, 6515, 1300, 298700,
Fri Jul  8 10:01:02 2011, 33-4-3, 2, C._Question, 291, 79, 7899, 1384, 297316,
Fri Jul  8 10:01:02 2011, 33-4-3, 3, C._Clarification_Request, 272, 130, 8840, 941, 296375,
Fri Jul  8 10:01:08 2011, 33-4-3, 4, ask for clarification, 273, 262, 9899, 1059, 295316,
Fri Jul  8 10:01:10 2011, 33-4-3, 5, N._Clarification_Request, 559, 130, 16557, 6658, 288658,
Fri Jul  8 10:01:11 2011, 33-4-3, 6, N._Clarification_Request, 559, 129, 17107, 550, 288108,
Fri Jul  8 10:01:12 2011, 33-4-3, 7, N._Question, 559, 97, 18607, 1500, 286608,
Fri Jul  8 10:01:14 2011, 33-4-3, 8, Navigator_Start, 559, 56, 20413, 1806, 284802,
```

The basic code for this logger is below. The nice thing about it is that both this logger and the one above are almost completely controlled by a .csv file, that looks like this: Each column specifies the name, type , and x,y position of a button. Here, all we had was buttons, but others are possible, such as the map in the tool above

```
Start, button, 200,210,
Complete, button, 600,210,
Communicator Start, button, 200, 50,
C. Question, button, 200, 90,
C. Clarification Request, button, 200, 130,
C. Response to question, button, 200, 170,
Navigator Start, button, 600, 50,
N. Question, button, 600, 90,
N. Clarification Request, button, 600, 130,
N. Response to question, button, 600, 170,
```

The code for a fairly generic version is here:

See demo30.pbl

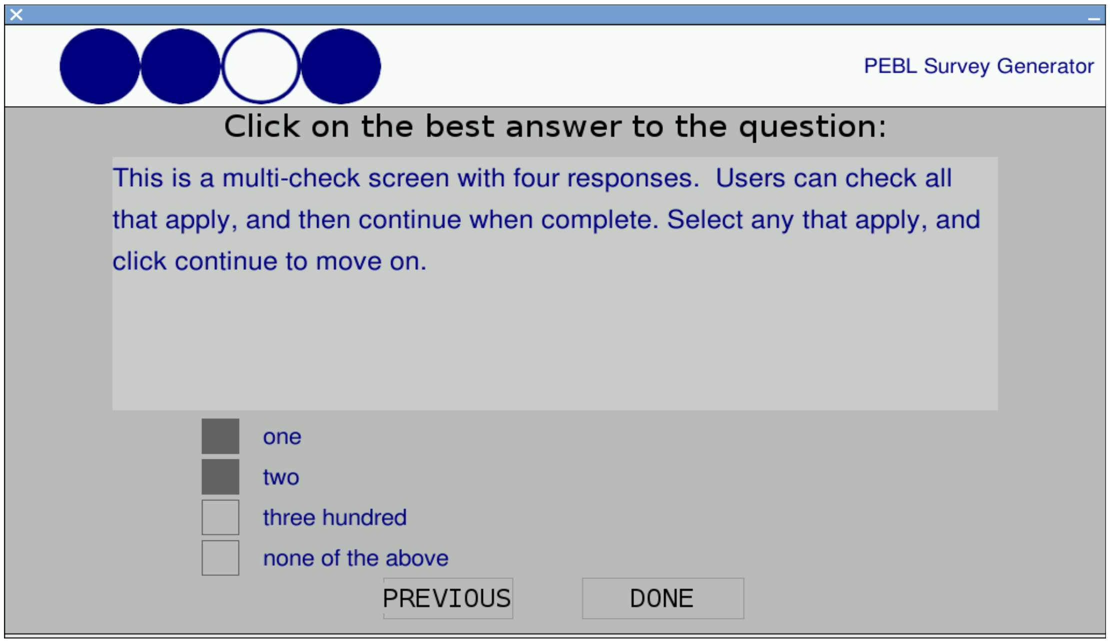
```
define Start(p)
{
  gWin <- MakeWindow()
  gSleepEasy <- 1
  scename <- "labels.csv"

......
```

Developing Surveys

There are many simple-to-use survey tools available, especially ones that can be designed and run on-line. PEBL has a task in its battery that fills some niches these survey generators miss, including:

- Field studies where internet access in unavailable
- Demographics surveys or personality questionairres that are given within a study
- Capabilities where you want real-time scoring of responses which determine experimental conditions or manipulations
- Limited 'kiosk' mode, where you restrict ability to access computer.

PEBL's survey generator offers a number of different screen types. Below is a multi-select option, where the participant chooses as many options as apply.

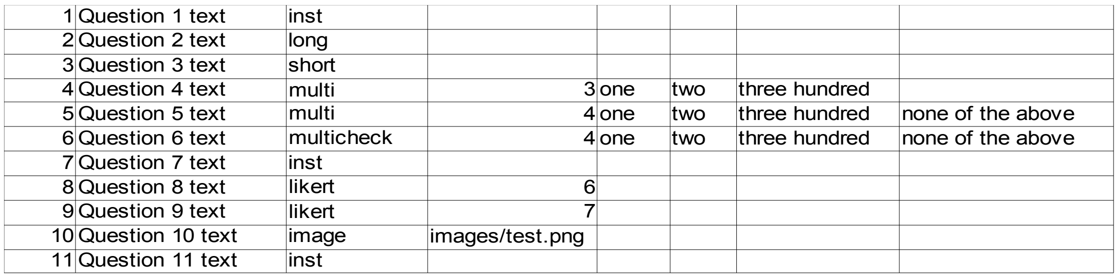

PEBL Survey Generator

Click on the best answer to the question:

This is a multi-check screen with four responses. Users can check all that apply, and then continue when complete. Select any that apply, and click continue to move on.

- one
- two
- three hundred
- none of the above

Like the tools above, the survey generator is controlled by editing a .csv file. This can typically be done using a spreadsheet program such as excel., but is also easy to do with a text editor.


1	Question 1 text	inst						
2	Question 2 text	long						
3	Question 3 text	short						
4	Question 4 text	multi		3	one	two	three hundred	
5	Question 5 text	multi		4	one	two	three hundred	none of the above
6	Question 6 text	multicheck		4	one	two	three hundred	none of the above
7	Question 7 text	inst						
8	Question 8 text	likert		6				
9	Question 9 text	likert		7				
10	Question 10 text	image	images/test.png					
11	Question 11 text	inst						

The survey is found in battery/survey, and the questions in questions.csv

.

Touchscreen Input

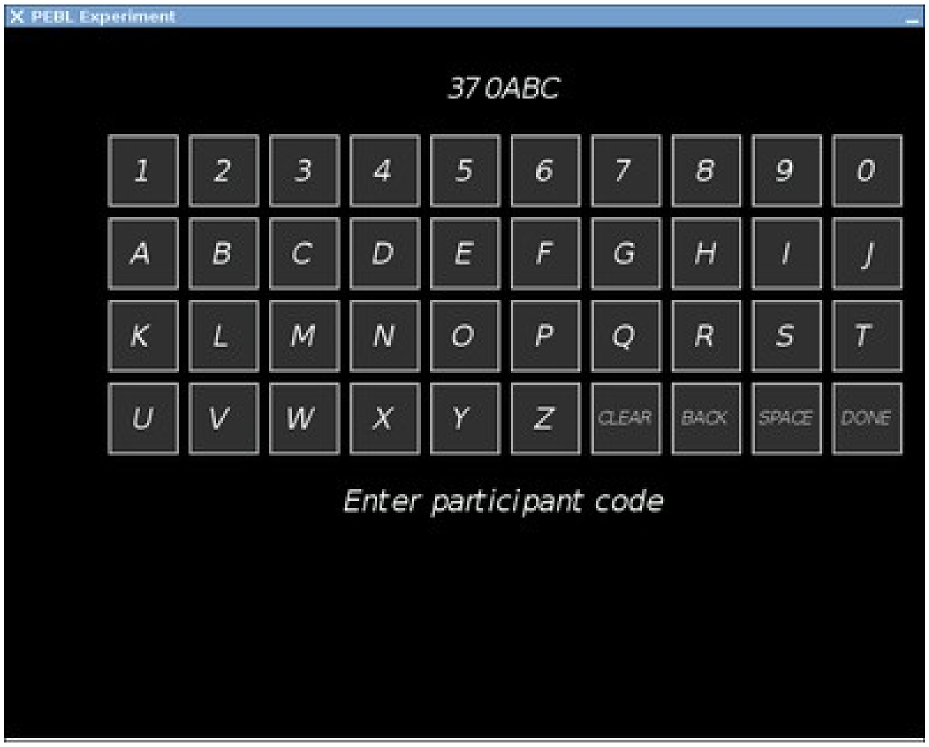
With the rise of netbooks and tablets, entire new opportunities for field research have opened up. PEBL can support a lot of this, hosting surveys and questionnaires. However, one thing you might like to do is get rid of any need for a keyboard. This especially impacts entering things like subject codes or conditions.

In the demo/vkeyboard.pbl file, there is a function that creates a virtual keyboard. You can copy this function into any experiment you want to use, and then modify particular functions to use it. If there is a pre-canned function that uses keyboard input, you can redefine the function in your experiment, and PEBL will use the new one instead of the standard one. So, consider GetSubNum. If VKeyboardInput is in your .pbl file, you can create a new GetSubNum that looks like this:

See demo31.pbl

```
define GetSubNum(win)
{
    out <- VKeyboardInput(win,"Enter participant code","keyboard")
    return out
}
```

Your experiment will use this new function instead of the standard one that requires a keyboard. It will display a screen that looks like this:



Summary and Conclusions

Past, Present, and Future of PEBL

PEBL was originally designed as a simple cross-platform programming language targeted to developing psychology experiments. Two critical aspects of this are that it is both cross-platform and open source. This is critical for scientific archival and exchange; you can count on your experiments being available to yourself and to others, without the need to re-acquire licenses, and even if they use a different computing platform. These are the core values of PEBL, and are essential to its success.

As PEBL became more mature, other aspects have become important to the community. For example, we began distributing the PEBL Test Battery as a way to help promote its use and create easy-to-modify experiment templates. These became more mature, and with contributions, translations, and experiments from researchers around the world, it has grown into the largest completely-free behavioral test battery available. It has become an important tool for clinicians around the world, as it provides a free alternative to proprietary tests that can cost hundreds of dollars each or incur license fees for every application. This user group is generally not made of researchers, but rather practicing clinicians at poorly-funded organizations (which characterizes most mental health organizations both within and outside the U.S.)

Currently, PEBL is typically downloaded 1500-2000 times per month by researchers around the world, and has over the years been downloaded nearly 70,000 times. There are nearly 40 published articles and reports that have cited or used PEBL, and there are likely more that have used it but do not cite it explicitly. PEBL is approaching and in many ways exceeding the capability of many commercial research products. We publish an occasional web journal called "PEBL Technical Report Series", which enables PEBL-based research that would otherwise go unreported to have a platform. This is especially important for publicizing norm studies, which are often challenging to publish in traditional journals because they can appear to editors to be be only of "incremental" scientific merit.

In the future, PEBL has some important improvements on its roadmap, including:

- Improved audio input and playback
- Video playback capabilities
- Better interfaces to response devices and neuroscience hardware
- Versions for IOS and Android
- A targeted distribution for classrooms

So, now that the PEBL tutorial is concluded, welcome to the PEBL community! I hope you find ways to use it in your own research, and can find ways to repay the benefits, contributing back to the PEBL community, and help grow a vital open research community.