Programming Tasks for Psychological Experiments using Presentation

Thomas E. Gladwin

Contents

I. Introduction

This book aims to provide the foundations of making typical types of experiments using Presentation. The aim is to teach maximally generalizable aspects of task programming; that is, instead of using special, program-specific functionality, I'll tend to use low-level functions present in any programming language as much as possible, in combination with basic Stimulus objects that should have rough analogues in any other system.

The sections explain the basic structure of a Presentation task and the foundations of task programming via a sequence of canonical examples that contain important elements of task programming. The sections are meant to be read ***while actually typing everything in and building and running the tutorial tasks*** as you go along. Just reading through the book almost certainly won't help develop skills or understanding. The text is designed to allow you to follow the evolution of a task program as you'd typically tinker with it; I'm consciously showing the scaffolding.

Warning: If you're reading this on something you can copy-paste from, you might be tempted to do so; but this could lead to a lot of extremely difficult to detect typos due to conversion to pure ASCII text, such as punctuation marks such as apostrophes or dashes being subtly messed up.

Note that the Getting Started and Tutorials sections in the Help function of the program (Help -> Documentation) provide a good "authorized" introduction, and should be the first place to look for help. If there's something in the code that's unclear or isn't working, look it up in the Help. A close second option is Google: there's a huge amount of information online, and your question has probably been asked and answered somewhere public.

II. Hello World!

1. Installing Presentation

Acquire a license code, network license, or dongle. Then download Presentation from the website of its developer, Neurobehavioral Systems (Google for “Neurobs”, since Googling for "Presentation" will not help you much) and install it.

2. Preparation

Before starting on a task, make a directory to contain the task files, and inside that make two subdirectories called Log and Stimulus. This is a magical ritual to be followed always, even if you don't plan to save log files or use stimulus files. For the example task in this section, set up a main directory this way called HelloWorld now.

3. The first two types of file: the Experiment file and the Scenario file

Presentation experiments consist of two parts: the Experiment (.exp) file, and the Scenario (.sce) file. The Scenario file is where the “programming part” of the task starts: this is the code implementing what stimuli appear, what’s done with responses, what gets saved, and so on. The Experiment file specifies the interface between the Scenario and the real world: which buttons are used, what the subject number is, which scenarios will be called in which order, where files will be saved, and so on.

4. The Experiment tabs

Open Presentation. The information in the Experiment file is shown in the tabs of the Presentation window. On the first tab, the Main tab, you can specify the name of the Experiment, the subject name (although I recommend always using subject *numbers* together with an Excel file matching subject numbers to names and demographic information; see below), and start up the Experiment when everything’s ready. You can save the Experiment file from this tab (the save button is bugged: you don’t or hardly see it press down). Name the Experiment “HelloWorld” and save it to a nice directory.

The second tab, Scenarios, is about files and directories. You have to specify the directories from which (if necessary) stimulus files will be read (the Stimulus subdirectory) and to which files will be saved (the Log subdirectory). We won’t use these directories yet, but they should be specified by default (it turns out that the act of specifying them, even unnecessarily, causes a psychological state of preparedness for writing code). Specify the Log and Stimulus directories via the arrow buttons in the center of the screen (create them first if they don’t exist yet). Here is also where you select the scenario files that will be run by the Experiment. We’ll return here later when there’s a scenario to add.

The third tab, Settings, contains interface information, and is where difficulties tend to exist. Here is where, primarily, the response buttons are defined, on the Response panel. To select buttons, choose the Keyboard device and select keys from the Buttons list via the Use button. Note that

different sets of buttons are specified per scenario in the experiment, so you have to be careful for which scenario you're defining buttons. (The "default scenario" keys are used when running scenarios directly when they're not specified in the experiment.) The number of response buttons defined on this tab must match the number of buttons specified in the associated scenario. Tip: you can choose keys from the Buttons list by pressing the key you want. Special keys are named as SPACE and so on (scroll down to check them). Other input devices are also selected here, including joysticks. How to define and use such devices is explained later.

Also on the Settings tab, on the Logfiles panel, is the default method of naming log files and how to provide the subject name. I usually fill that in on the Main tab, and prefix the subject name to the log file. For this, make sure the "Prefix subject name to scenario file name…" and the "Use entry on Main tab" options are selected. The Video panel, can be important to check when you're connected to dual monitors: the Adapter options might include a second monitor where the task can be presented. Finally, the Port tab is where port input and output devices can be defined, such as joysticks.

The final tab, Editor, is where you write scenarios.

5. Scenarios

Scenarios are the "real" task program: this is where you write the code that defines and presents stimuli, checks responses and saves data. In Presentation, code is divided into two main types: code that defines the "building blocks" of the task and code that presents and manipulates those building blocks. There's also a third type of code: the header.

Make a new scenario and save it as "HelloWorld.sce" in the task's directory. Then go to the Scenarios tab and use the arrow button put this scenario file in the Scenarios list. This lets you choose the scenario when running the experiment later, and lets you define keys for this scenario on the Response panel on the Settings tab.

An important general point about writing code is that Presentation is case-sensitive: that is, code in upper- and lower-case letters is **not** the same code. If a function is called "loop", for instance, then typing "Loop" won't be recognized.

6. The header

The first lines of a scenario specify some standard task variables: this is the header that precedes even the beginning of the SDL code.

```
# Header

scenario = "HelloWorld";

response_matching = simple_matching;

active_buttons = 0;

default_background_color = 50, 50, 50;
```

default_font_size = 48;

The first line, starting with #, is a comment (it turns red in the Presentation editor). The variables specify the following information. “Scenario” provides the name for the scenario. Response_matching in the vast majority of situations is just excess baggage. It tells Presentation how to classify stimuli and responses; simple_matching is the standard, as opposed to “legacy matching” which you’ll probably never use, and is assumed in the response-handling section below.

Active_buttons defines the number of response keys to be used; this **must** be equal to the number of keys specified for the scenario on the Settings tab! For now, we’re not using responses.

Default_background_color specifies the background screen color. The three numbers represent red, green and blue values, from 0 to 255. Black and white are 0, 0, 0 and 255, 255, 255, respectively; 0, 50, 0 would be dark green. The 50, 50, 50 gray background chosen here is more neutral than harsh white. Default_font_size is what it says, in pixels: when displaying text, this is its standard size.

Note that every line ends with a semi-colon “;”. Every command in Presentation has to end in a semi-colon; usually, this means that every line of code ends in a semi-colon. Also, Presentation is case-sensitive, i.e., whether letters are capitalized matter. Typing "Loop" instead of "loop", for instance, will cause an error.

7. Scenario Definition Language: SDL

SDL is the “building block” type of code: the Scenario Definition Language. After setting up the header, we define task objects using SDL: simple stimulus components (text or images), pictures in which basic stimuli are combines, and trials consisting of events in which pictures are presented. Here’s the SDL code for the most basic stimulus: a word to be presented onscreen. Note that the SDL code following the header starts with the “begin” command: this tells Presentation that the header has been specified and the SDL part is coming next. Type the following below the header code in the HelloWorld.sce scenario file in the Editor tab.

```
# SDL

begin;

text {

        caption = “Hello World!”;

} my_text;

picture {

        text my_text;

        x = 0;

        y = 0;

} my_picture;
```

```
trial {
        trial_type = fixed;
        trial_duration = 1000;
        stimulus_event {
                picture my_picture;
                time = 0;
        } my_event;
} my_trial;
```

The first line is another comment, to make it easy to see where the SDL code begins. Then three objects (object: a set of variables and methods that Presentation treats as a unit) are defined: a text object, which is used in a picture object, which is finally used in a trial object. Note the "grammar" Presentation expects when defining objects: the type of the object (text, picture or trial), then the specification of the properties and components of the object between accolades, and a semi-colon at the end. The component lines are also ended with semi-colons.

So, the text object has the property "caption", here "Hello World!". The picture object, my_picture, has the text object defined above, named my_text, as its only component (more complex picture objects are possible as explained later). The x and y lines specify where the text will appear on screen: if x and y are set to zero, the text will appear centrally.

Now the trial object. Trials have one fundamental property: the trial_type. **Fixed** means that the trial will end after the duration specified in trial_duration. (The trial ending means that presented stimuli are removed from the screen, and the task continues.) The other type of trial is **first_response**, which means that trial will end when a response is given (explained further below). The kind of component contained by trials is the stimulus_event. This contains a picture that determines what will be shown on screen when the stimulus event occurs, and a time that determines when the stimulus event will occur relative to the start of the trial.

At this point all the ingredients are prepared for the best code: PCL.

8. Presentation Control Language: PCL

PCL, Presentation Control Language, is used for manipulating and presenting the objects defined in SDL.

Make a new file called HelloWorld.pcl. Then, at the bottom of the scenario file, type

```
#PCL

begin_pcl;
```

include "HelloWorld.pcl";

This will load the contents of the .pcl file when Presentation runs HelloWorld.sce. Note that you don't necessarily have to put the rest of the PCL code in a separate file, it just makes things tidier. But with more complex tasks, it starts becoming very important to pay attention to how you organize code over different files.

Go to HelloWorld.pcl to enter the PCL code. At this point this is going to be very simple: just the single line

my_trial.present();

All that's going to happen is that the trial defined in SDL will be presented onscreen.

9. Running HelloWorld!

We're almost ready to run the HelloWorld scenario. Running tasks is done via the Main tab: go to Main and click the Run button. This opens a window that lets you select scenarios added to the experiment, currently only HelloWorld.sce. Pressing Run Scenario starts the scenario, following a Ready screen you go past by pressing Enter. Presentation should now show you the fruits of the labor so far: one second of "Hello World!"! After the task ends, Presentation automatically shows you a summary of what happened; we won't be using this so just click it away.

10. Exercises in SDL

Now to mess around with things. Go back to the .sce file and change some stuff, like the color, the caption, the location and the duration, and re-run the task to see what happens. If it all goes horribly wrong, Q is the panic button: this (Q)uits the task and returns to the Presentation interface. To repeatedly run the task a bit more efficiently, use the green arrow or F5 while in the Editor **while the tab for the .sce file is active**. This will give an error if you try to run the .pcl file directly (i.e., while you're on its tab), since Presentation doesn't know that it should first run the preceding SDL and other code in the .sce file.

11. Setting stimulus properties in PCL

In a real task, of course, you'll use PCL to change which stimuli are presented instead of changing things in SDL by hand. As will become clear, we can do very many things just using three property-setting commands. Return the SDL code to the original property values. Then go to the .pcl file and, before the my_trial.present(); line, type

my_text.set_caption("I am a PCL master");

my_text.redraw();

Run the task: now you've used PCL to set the caption! Note that until you redraw, the caption change isn't executed.

To set the duration of the trial:

my_trial.set_duration(3000);

And the text stays onscreen for 3 instead of 1 second.

Finally, we're going to change the color of text; put this between the set caption and the redraw command.

my_text.set_font_color(255, 0, 0);

The input to the set_font_color function defines a color via red, green, blue values, on a scale from 0 to 255, as with the defaut_background_color line in the header. So now, the text should appear in red.

Congratulations! This was step 1 of Presentation programming. You should know what the header, SDL and PCL are, and how to conjure text onto the screen and manipulate it via PCL code.

III. Emotional Stroop

12. Introduction and preparation

This section introduces the building blocks for programming tasks: loops, stimulus sets, response management and if-then statements. These will all be illustrated in our second tutorial task: an Emotional Stroop task (MacLeod and MacDonald, 2000). The naming of this kind of task as any kind of Stroop task had been criticized, since it's not like the classical Stroop task where we have a direct conflict between an automatic dominant response (reading a word) and a weaker task-relevant response (naming the print color of the word). The Emotional Stroop just has the same task structure: you're supposed to respond to one feature of the stimuli, but you experience interference due to some other feature that should be irrelevant. However, this is interference is not, as in the classical Stroop, an overlearned contradictory response but probably something more like distraction: emotional stimuli appear to automatically evoke processes that are not necessarily good for task performance (Reeck and Egner, 2011).

Make a new directory called EmoStroop, and create the files EmoStroop.exp, EmoStroop.sce and EmoStroop.pcl via Presentation (you could use Save as… if HelloWorld is still open; then remember to change the Experiment Name on the Main tab, and the files and directories on the Settings tab; click scenarios and press Delete to remove them). You can copy almost all the .sce code from HelloWorld.sce; just change the scenario name in the header and the final line:

include "EmoStroop.pcl";

All the PCL code in the next sections belongs in EmoStroop.pcl.

13. Loops

In HelloWorld, the scenario involved a single stimulus presentation. Let's present *ten* stimulus-presentation trials!

```
# Variable declarations

int iTrial;



# Task

loop iTrial = 1 until iTrial > 10 begin

        my_text.set_caption("Iteration number " + string(iTrial));

        my_text.redraw();

        my_trial.present();

        iTrial = iTrial + 1;

end;
```

The first command after the #comment declares a variable called iTrial. PCL requires every variable (i.e., a name coupled to a value that can be changed by the code) to be declared before being used; this means telling presentation that the specified name is a variable of a certain type (and not for instance a typing error). Here, the name of the variable is iTrial (which stand for "trial index"), and its type is "int". This means that iTrials contains integer (whole) numbers: 0, 1, -1, 2, -2, etc.

This variable is subsequently used as a *loop counter* in the loop. Note the syntax (grammar): the loop command line says that iTrial is the loop counter, that it starts at value 1, and that the loop will end when iTrial exceeds 10 at the start of an *iteration*. An iteration is one execution of the code inside the loop (between the Loop and the End lines), performed for one value of the loop counter.

What happens in our loop is that iTrial is set to 1, and an iteration is executed for this value. In the set_caption line, the integer value iTrial is converted into a *string*: a sequence of letters. Note that the number 1, as encoded by Presentation, is a very different thing than the sequence of letters "o", "n", "e". The number 1 can be added to 2 via arithmetic; the string "one" can be appended to another string to make a sentence. That's what happens on the first line of the iteration.

The final line of the iteration is very important: this increments the value of iTrial by one, so the next iteration will use the next value and, finally, the loop will stop and the code will continue past it. An easy mistake to make is to forget to increment the loop counter, which means the loop keeps going for ever (or until you press Q to quit the task).

Run the program: you should see a message telling you which iteration is being performed!

Is it boringly slow? It will be after a while, so change the trial duration! Add

```
my_trial.set_duration(250);
```

just above the my_trial.present(); line.

Note the indentation (the tabs shifting the content of the loop to the right). This is **essential** to writing readable code. Whether someone correctly indents their code (or how quickly they learn to do so) is the single most accurate prediction of their value as a scientist, and a human being.

Now to fix something horrible in the code: a hard-coded value. The loop will run for ten iterations. What if you want it to be 5 or 20? What if the code is getting complex or you have multiple loops? Would you want to try to remember and go find every place in the code where you'd need to change something? What if you want the number of iterations to be determined via code? Luckily we can change this easily: we'll use a variable to determine the number of iteration, and give it its value (or "initialize it") right after the variable declarations. That way any parameter used in the scenario can be found quickly and changed if required.

We need to declare an integer variable called nTrials (number of trials), set it to 10, and replace the hard-coded 10 in the loop line with it.

```
# Variable declarations

int iTrial;

int nTrials;


# Variable initialization

nTrials = 10;


# Task

loop iTrial = 1 until iTrial > nTrials begin

        my_text.set_caption("Iteration number " + string(iTrial));

        my_text.redraw();

        my_trial.present();

        iTrial = iTrial + 1;

end;
```

Much better ☺

Exercises: set nTrials to 3 to check if it works. Then set nTrials to 100 but increment iTrial by 17; then increment iTrial by itself: iTrial = iTrial + iTrial. Try the version iTrial = iTrial * 2, where the star is the multiplication function.

14. Blocks

To create a block design, place the trial-loop inside an outer loop, making the code look like this (added code in bold):

```
# Variable declarations
int iBlock;
int nBlocks;
int iTrial;
int nTrials;



# Variable initialization
nBlocks = 4;
nTrials = 10;


# Task
loop iBlock = 1 until iBlock > nBlocks begin
    loop iTrial = 1 until iTrial > nTrials begin
        my_text.set_caption("Block number " + string(iBlock) + ", iteration number " + string(iTrial));
        my_text.redraw();
        my_trial.present();
        iTrial = iTrial + 1;
    end;
    iBlock = iBlock + 1;
end;
```

Now we have blocks of trial-loops: *nBlocks* of blocks, containing *nTrials* of trials. If you want to change the number of blocks or trials, that can be done simply by changing the controlling variable at

the start of the code. This is how you ideally want things to be: most of the code never needs to be changed, only the values of variables located near the top of the PCL file.

Try it out!

Now to add an introduction screen to each block. On the line below the block-level Loop command add

```
loop iBlock = 1 until iBlock > nBlocks begin
    my_text.set_caption("Block " + string(iBlock) + " of " + string(nBlocks));
    my_text.redraw();
    my_trial.set_duration(1000);
    my_trial.present();
```

Hopefully this will prevent subjects getting too temporally disoriented and befouling themselves halfway through the task (always a risk in lab research).

15. Stimulus sets: arrays

The next step is to add some (deep and) meaningful stimuli. To do that, first we need *arrays*.

Arrays are a special kind of variable: *numbered lists* of values, rather than a single value (note that a "value" can be either a number or a string). Let's make an array of words. Add this to the declarations:

```
int nWords = 10;
array<string> words[nWords];
```

Note the syntax for defining an array: the type is specified between angular brackets, and the number of elements is given in square brackets. Note also that we have to initialize nWords in the declaration section! We're making this exception because we're using the value of nWords to declare the words array.

Then add this to the initializations:

```
words[1] = "LAMP";
words[2] = "CHAIR";
words[3] = "TEA CUP";
words[4] = "PLATE";
```

words[5] = "SPOON";

words[6] = "DEATH";

words[7] = "SEX";

words[8] = "POO MAN";

words[9] = "LOVE";

words[10] = "HATE";

The "indexing" of the array, that is, how to tell Presentation which element of the list you want to do something with, uses square brackets. The index starts from one (other languages might count from zero).

Now we're going to use these words as stimuli. In the trial, set the caption as follows:

my_text.set_caption(words[iTrial]);

Try it! You should see your list of words appear in order.

However, danger lurks. What if you'd have more than ten trials? At iteration 11, you'd get a error message and the task would crash to the Presentation interface: there *is* no words[11] – the words only go up to ten. To deal with this, and very many other problems, elegantly, we use the modulo function.

16. The wonderful modulo function

The modulo is a mathematical function that works like this. Take two whole numbers, A and B, and divide A by B. In other words, determine how many times B can go into A. For instance, take 13 divided by 4:

[1 2 3 4] [5 6 7 8] [9 10 11 12] 13

So 4 goes into 13 three full times (4 * 3 = 12), which leaves a *remainder* of 1. So we could write 13 = k * 4 + 1, where k is 3 but we don't really care about that – k is just the biggest whole number such that k * 4 doesn't exceed 13. We care about the *modulo*: the remainder that doesn't fit the division. In the preceding example we say "13 mod 4 = 1", in other words: the remainder after dividing 13 by 4 as well as possible is 1. Check whether this is clear: 7 mod 4 = 3, 10 mod 2 = 0, 112 mod 25 = 12.

Additional explanation: the modulo comes from the ancient Greeks, when they only believed in integers – zero and positive numbers. We can play a game in this world of integers: take any number A and write it like this :

*A = k * B + r*

Remember, all the numbers are integers! Now, we can call B the "Base" of the modulo. k is how many times B can fit into A; note that this might mean that k equals zero! This happens when A is smaller than B. r is the remainder: the modulo. Again: r can't be negative. This means that we can say the k is

*the biggest number allowable – if k*B becomes bigger than A, r would have to be negative, and we're living in a world of positive numbers.*

One application of the modulo function is to determine whether a number is odd or even: even numbers are "modulo zero" when dividing by two; odd numbers are modulo 1. We'll use this a lot. But here, what the modulo lets us do is make sure we don't exceed the limits of our array.

In the PCL code, declare a new integer variable "iWord" and set nTrials = 20. Now we'll change the trial loop as follows:

```
loop iTrial = 1 until iTrial > nTrials begin
    iWord = 1 + mod(iTrial – 1, nWords);
    my_text.set_caption(words[iWord]);
    my_text.redraw();
    my_trial.present();
    iTrial = iTrial + 1;
end;
```

So what we're doing is using iWord is the index for the words array, instead of the untrustworthy iTrial directly. The clumsiness with the plus 1, minus 1 is due to Presentation indexing from 1 and not zero. Consider the first iteration: iTrial is set to 1. We could, at this point, just let iWord be iTrial mod nWords, that is, it remains 1. Why not do it that way? Consider what happens when iTrial reaches nWords (here, ten). Then we get 10 mod 10, or *zero*. But this would give an error, since Presentation indexes from one! We have to make sure that iWord is at least 1, and then add the modulo to that. So, on iteration 1, we get 1 + (1 – 1) mod 10 = 1 + 0 mod 10 = 1 + 0 = 1, which gets us the first trial as desired. At trial 11, the array index "loops back": 1 + (11 – 1) mod 10 = 1 + 10 mod 10 = 1 + 0 = 1 and so iWord is 1 again. For trials 12 to 20, iWord rises back up to 10. If we continued adding trials, iWord would simply loop back every tenth trial. If we increase or decrease the number of words later on, this would still work seamlessly. The modulo function thus keeps the index range within acceptable bounds.

17. If – then statements and choosing font colors

Together with loops, if - then statements do almost all of the work involved with so-called "flow control", i.e. what happens when. While loops control how often a chunk of code is executed, if – then statements let us choose the conditions under which code is executed.

Here we'll use an if – then (or more precisely an if, else, then) statement to determine the color of the stimulus word. For now, let's make the color alternate every trial. Declare three new variables: redValue, greenValue and blueValue. Then add the following code to the trial loop:

```
loop iTrial = 1 until iTrial > nTrials begin
```

```
		if (mod(iTrial - 1, 3) == 0) then
			redValue = 255;
			greenValue = 0;
			blueValue = 0;
		elseif (mod(iTrial - 1, 3) == 1) then
			redValue = 0;
			greenValue = 255;
			blueValue = 0;
		else
			redValue = 0;
			greenValue = 0;
			blueValue =255;
		end;
		my_text.set_font_color(redValue, greenValue, blueValue);
		iWord = 1 + mod(iTrial – 1, nWords);
		my_text.set_caption(words[iWord]);
		my_text.redraw();
		my_trial.present();
		iTrial = iTrial + 1;
	end;
```

The font color will loop through red, green and blue. Check out the syntax and try out some different colors and modulo functions.

*Note the **double** equals-signs. The single-equals sign in Presentation means assignment of a value; double means a logical comparison. Mixing them up will generate errors.*

What if you want the color to change every other trial? So that you'd get red – red – green – green – blue – blue? This can still be done via the modulo: you just divide (iTrial – 1) by 2. Since iTrial is an integer variable, that is, a whole number, dividing the sequence of iTrial values from 1 to 10 would result in (iTrial – 1) / 2 values of 0, 0, 1, 1, 2, 2, 3, 3, 4, 4. For instance, if iTrial is 6, then (iTrial – 1) / 2 is 5 / 2 is 2, because the integer-division in Presentation rounds down.

Note that the font_color will bleed over into block instructions! To avoid this, set the font color back to white there:

```
loop iBlock = 1 until iBlock > nBlocks begin
    my_text.set_font_color(255, 255, 255);
    my_text.set_caption("Block " + string(iBlock) + " of " + string(nBlocks));
    my_text.redraw();
    my_trial.present();
    . . .
```

(You can leave the trial coloring however you prefer for now.)

18. Randomization of stimulus lists

Of course presenting lists of words in the same order may not entail ideal task design. The primary Presentation function for randomization in the context of stimulus sets is shuffle(). Shuffle randomly permutes, i.e. changes the order of, the elements of an array, or a subset of an array. So we have the same elements, just in a different order.

To illustrate shuffle(), add the following line to the initialization code, below the list of words.

```
words.shuffle();
```

Run the task a few times. You'll see that the sequence of the ten words is now random. But the sequence itself is simply repeated every ten trials. Not good enough!

Remove the words.shuffle(); line from the initialization code.

We could add it at the start of the block iteration code, but then the sequence would repeat within a block, if there were more than ten trials. Well, we could add the shuffle to the trial code, providing a random-with-replacement randomization. Add words.shuffle() to the trial loop and see what happens. Looks random, hopefully!

Random-with-replacement might not be desirable however: maybe we want all the words to be presented before the list starts again in a new order. It's also somewhat computationally inelegant to shuffle an array every single trial – for big arrays and short trial durations, this could conceivably cause problems with timing. Further, if we shuffle every trial, indeed if we shuffle the words list at all, the index iWord becomes meaningless. It used to be the case that we knew that the first five words (the words with index 1 to 5) were all low-arousal, but after shuffling that information would be lost.

To do it right, we need the good old modulo function and a trick using a new array.

19. Helper arrays for better randomization

Instead of shuffling the word list, we're going to shuffle a "*helper*" array: shuffledIndices. Add the declaration

array<int> shuffledIndices[nWords];

Then add the following at the end of the initialization code:

loop iWord = 1 until iWord > nWords begin

shuffledIndices[iWord] = iWord;

iWord = iWord + 1;

end;

shuffledIndices.shuffle();

We made an array of the number 1 to 10 (or whatever we set nWords to be). At first, the value was exactly the same as the index, but the values were shuffled. Aha! We can shuffle the helper array and use it to address the words list, without messing up the order of the words!

To do so, add the following code in the trial loop, where iWord is specified:

iWord = 1 + mod(iTrial – 1, nWords);

if (mod(iTrial – 1, nWords) == 0) then

shuffledIndices.shuffle();

end;

iWord = shuffledIndices[iWord];

So we find an iWord as before, running through the values 1 to nWords and then looping back. However, in the final line, we don't use this value to get a word from the word list, but an index from shuffledIndices. This changes iWord to a random index without interfering with the word list! The if – then statement causes shuffledIndex to be shuffled on the first trial and whenever the full list has been run through: every tenth trial in our case.

We can take it one step better: *ninja shuffling*. As it is, the above method of randomization allows repetitions of words: the last word to be presented before a shuffle could be the first word after the shuffle. We can avoid this by shuffling subsets: replace the shuffle line in the new code by

shuffledIndices.shuffle(1, nWords - 1);

shuffledIndices.shuffle(2, nWords);

Shuffle can be given two numbers as input: these specify the index ranges within which to shuffle. So now we shuffle the word list twice. First, we shuffle everything except the last word. Second, we

shuffle everything except the first word. Thus, the dangerous previously-last word can never reach the new first position, but all the other words can get to every position. This keeps the whole list churning and avoids immediate repetitions. By changing the range of the "buffer zones" at the start and end of the list to 3 and nWords – 2, repetitions with only one different stimulus in between would also be avoided.

A further consideration when using randomization is the random_seed. Computer randomization functions actually use deterministic sequences of number: the sequence just “looks random” in some well-defined way. "Random" really means "unpredictable": if you know the current number in the sequence, there's no (obvious) general rule that predicts the next number. The random_seed says where you start on the sequence: so, same random seed, same “random numbers” that will follow. This can be a good thing. By setting the random seed to the subject number, you can in principle reconstruct all the random variables that happened during the task. This could help save a dataset if something goes wrong and an essential independent variable isn’t saved. To do this, immediately after the begin_pcl; line in the .sce file, add

set_random_seed(int(logfile.subject()));

Note the use of logfile.subject(): logfile is an automatically generated variable that contains information including the subject name you specified. If you use a subject number, as I’d strongly suggest, the integer conversion of the subject name will be a number, and will hence provide a unique random seed. This means the randomization will vary over subjects, but you can reconstruct it for a given subject. **VERY IMPORTANT:** if you do this, make sure the subject name is only a number: no letters as prefix or postfix, no messing about with preceding zeros: just the number. If the input to the int() function has any alphanumeric junk, as in "pp001" instead of just "1", it will return the value 0, and *everything* depending on randomization will be the same for all your subjects...

20. Responses

So we have all kinds of things in place: high- and low-arousal words, presented in varying colors, in nicely randomized order. Time to add an actual task!

In this case, this will involve pressing buttons. Let’s say that the task is to press “A” for red words and “D” for blue words.

First, time to visit the Settings tab and its Response panel. If you added EmoStroop.sce to the Experiment, you’ll see if in the Scenarios list on the Response panel. Select the scenario, then select Keyboard under Devices, and then specify (select and Use) two response buttons, A and D. Pressing the button after clicking something in the Buttons list will skip to buttons starting with that letter. Note that the order of buttons is determined by the order of selection. If you select A and then D, then pressing A will be coded as 1 in code, and D as 2. Test the newly defined active buttons via Test. Note that Q is used by default to (Q)uit a running task.

Now, in the header, change the active_buttons line as follows:

active_buttons = 2;

button_codes = 1, 2;

We now have two buttons specified in Settings, and that has to match the active_buttons in the header (why this isn't automatic beats me; perhaps it's some kind of test). The button_codes have to be specified for each button. In general this will just be 1, 2, . . ., number-of-buttons, but you could use this to swap button codes without reselecting them in Settings.

So we have buttons defined, but we want people´s button presses to be *judged,* harshly but fairly. So we need to access what color the stimulus was, and therefore which button is correct. For this it´s best to change how we're setting stimulus color: we'll use an explicit variable, which gets a random value, instead of having things vary along with the trial index.

Declare a new variable,

```
int colorCode;
```

and replace the code specifying the red, green and blue values in the trial loop with this:

```
redValue = 0;

greenValue = 0;

blueValue = 0;

colorCode = random(0, 1);

if (colorCode == 0) then

        redValue = 255;

else

        blueValue = 255;

end;
```

So we start off with all the color values at zero. Then we have a variable, colorCode, that determines whether the stimulus will be turned red (colorCode 0) or blue (colorCode 1). The colorCode value uses the random() function: this gives an integer value in the range provided as its input.

Try it out: now color are changing from blue to red randomly.

Now we need to do two more things: make the trial wait for and detect responses, and decide whether the response was correct.

The first thing means we're going to manipulate the trial_type. So far, this was fixed: the trial just appears and disappears. Now we need to make the trial_type **first_response** for trials, so it ends only when a response is given, and set it back to **fixed** for the block instructions. Add the following code:

```
loop iBlock = 1 until iBlock > nBlocks begin

        my_trial.set_type(fixed);

        my_trial.set_duration(1000);
```

```
        my_text.set_caption("Block " + string(iBlock) + " of " + string(nBlocks));
        my_text.redraw();
        my_trial.present();

        loop iTrial = 1 until iTrial > nTrials begin
            . . .
            my_trial.set_type(first_response);
            my_trial.set_duration(forever);
            my_trial.present();
            iTrial = iTrial + 1;
        end;
        iBlock = iBlock + 1;
    end;
```

Run the task: the words should stay on screen until you press one of the selected keys (A or D), but the block introduction should end after one second. If you set a lower duration then "forever" (a built-in Presentation value), then the trial could end before a response is given. This might need to be handled via special code. But for now, we know that if we reach code that comes after the trial has been presented, the subject has provided a response.

So how do we get at the response data associated with a trial? In Presentation there's a stimulus_manager for this, that's automatically available. After a first_response type trial is presented, information about the response that ended it is sent to stimulus_manager, and can be taken from it in the form of "stimulus_data", a complex object containing methods to recall response information. Declare a stimulus_data variable and some response variables,

```
    stimulus_data my_data;
    int RT;
    int acc;
    int resp_button;
    int correct_button;
```

Then in the trial loop, before the my_trial.present() line, add

```
    if (colorCode == 0) then
```

```
        correct_button = 1;
else
        correct_button = 2;
end;
my_event.set_target_button(correct_button);
my_trial.present();
```

This determines what the correct response is, and tells Presentation that this is the my_event stimulus event's correct response via the set_target_button command. (We won't use this now, but it's a good habit just in case.) Check the if – then statement: colorCode is red and button 1 is A, so this means that subjects should press A when they see a red word, and press D when they see a blue word.

Immediately after my_trial.present(), add

```
my_data = stimulus_manager.last_stimulus_data();
RT = my_data.reaction_time();
resp_button = my_data.button();
if (resp_button == correct_button) then
        acc = 1;
else
        acc = 0;
end;
```

So, following the execution and ending of the trial, the last_stimulus_data object is copied to my_data: this contains the information about the most recent response to a stimulus event. We can then use the response_time() and button() methods of my_data to get the reaction time and button code, and then use the button code to determine accuracy.

We want to see if this works of course. To do so, we'll add a feedback screen. Following the above code, add

```
# Feedback
if (acc == 0) then
        my_text.set_font_color(255, 0, 0);
```

```
        my_text.set_caption("Wrong!");
        my_text.redraw();
    else
        my_text.set_font_color(0, 255, 0);
        my_text.set_caption("Correct!");
        my_text.redraw();
    end;
    my_trial.set_type(fixed);
    my_trial.set_duration(500);
    my_trial.present();
```

And this seems like a good time to an intertrial interval, since it's very similar and comes at this point in the code:

```
    # ITI
    my_text.set_font_color(255, 255, 255);
    my_text.set_caption("+");
    my_text.redraw();
    my_trial.set_type(fixed);
    my_trial.set_duration(500);
    my_trial.present();
```

Try it out!

21. No hard coding!

In the current version, we´ve hard-coded the correct response to red to be A, and to blue to be D. In general, that should be counterbalanced: we don't want an arbitrary but systematic relationship lurking in the design as a confound.

It's actually a small step to deal with that at this point. What defines a correct response to the various colors? That little bit of code determining correct_button:

If (colorCode == 0) then

correct_button = 1;

else

correct_button = 2;

end;

There's a "zero" hard coded in that if statement – the colorCode for red. We can "abstract" the code by saying: the correct_button is 1 if the colorCode is the colorCode-such-that-the-correct-button-is-1. This makes more sense than it might seem ☺ Declare a new variable

int button_1_colorCode;

Initialize it as follows:

if (random(0, 1) == 1) then

button_1_colorCode = 0;

else

button_1_colorCode = 1;

end;

and finally change the correct_button code:

if (colorCode == button_1_colorCode) then

correct_button = 1;

else

correct_button = 2;

end;

Now the color requiring a button 1 (A) press will be varied randomly. You can try it out, but you'll have to figure out what the correct stimulus – response mapping is by trial and error. Now we really need instructions. Let's make them, and have them adapt to the mapping automatically.

At the start of the block loop, change the code as follows:

loop iBlock = 1 until iBlock > nBlocks begin

my_text.set_font_color(255, 255, 255);

my_trial.set_type(first_response);

my_trial.set_duration(forever);

if (button_1_colorCode == 0) then

my_text.set_caption("Press A for red and D for blue.\nPress either key to continue.");

else

my_text.set_caption("Press A for blue and D for red. \nPress either key to continue.");

end;

my_text.redraw();

my_trial.present();

. . .

We want subjects to really read the instructions now, so the trial_type is first_response. Depending on the colorCode associated with button 1, subjects get different instructions. Note the \n code in the caption: this makes Presentation print the following text on the next line.

22. Saving data

So the task is hopefully up and running happily. Maybe it would be nice to actually save some data!

Presentation lets you write data to a text file. Add the following code at the very top of the .pcl file:

include "openOutput.pcl";

And the following code at the very bottom:

ofile1.close();

Make a file called openOutput.pcl, saved in the same directory as the .sce and other .pcl file, and put this code in it:

string logpath = logfile_directory;

string fn = logpath + "EmoStroop_" + logfile.subject() + ".txt";

bool fe0 = file_exists(fn);

output_file ofile1 = new output_file;

ofile1.open_append(fn);

if (!fe0) then

ofile1.print("Block\t");

ofile1.print("Trial\t");

```
        ofile1.print("Color\t");
        ofile1.print("Word\t");
        ofile1.print("RT\t");
        ofile1.print("acc\t");
        ofile1.print("button\n");
    end;
```

The code recalls the specified Log directory that Presentation automatically stores in logfile_directory, makes a filename string built up from: the directory, a task identifier (here, EmoStroop) and a subject number, checks whether the file exists, opens it and gives it a label (ofile1) and, if it didn't exist already, writes a line of variable names. The \n code is like typing an Enter, making a new line; the \t code prints a tab spacing. The open_append command tells Presentation to add new lines to the file, instead of overwriting the file; so appending avoids accidental loss of data. But it might mean you have to disentangle subjects that accidentally got the same subject number. Luckily, the files to be created are simple text files and hence easy to manipulate.

The close() at the end if the code tells the operating system that the file is no longer being used; leaving files open can make Windows refuse to use them.

Now go back to the main .pcl file and go to the end of the trial loop, just before the iTrial incrementing line. Here, we can write to the file the variables we want to save. In general, save absolutely EVERYTHING, including a timestamp. Here, making sure the variables match the variable names printed to the first line:

```
        . . .
        ofile1.print(string(iBlock) + "\t");
        ofile1.print(string(iTrial) + "\t");
        ofile1.print(string(colorCode) + "\t");
        ofile1.print(string(iWord) + "\n");
        ofile1.print(string(RT) + "\t");
        ofile1.print(string(acc) + "\t");
        ofile1.print(string(resp_button) + "\n");
        iTrial = iTrial + 1;
    end;
```

Note that we have to convert the values to strings.

That's it! Try it out, use WordPad (not NotePad, which doesn't handle line breaks correctly) to open the file with saved data that should have been created in the Log directory, and bask in the knowledge of having programmed a task!

For convenience, here's the full task code. In EmoStroop.sce (note that this part stays simple):

```
# Header
scenario = "HelloWorld";
response_matching = simple_matching;
active_buttons = 2;
button_codes = 1, 2;
default_background_color = 50, 50, 50;
default_font_size = 48;

# SDL
begin;
text {
	caption = "Hello World!";
} my_text;
picture {
	text my_text;
	x = 0;
	y = 0;
} my_picture;
trial {
	trial_type = fixed;
	trial_duration = 1000;
	stimulus_event {
		picture my_picture;
```

```
                time = 0;
        } my_event;
} my_trial;
#PCL
begin_pcl;
set_random_seed(int(logfile.subject()));
include “EmoStroop.pcl”;
```

In EmoStroop.pcl (hopefully you, dear reader, have typed all this in over the course of the tutorial since otherwise it’s going to look incomprehensible; but otherwise, note that everything is built up from the simple chunks explained above):

```
include "openOutput.pcl";


# Variable declarations
int iBlock;
int nBlocks;
int iTrial;
int nTrials;
int nWords = 10;
array<string> words[nWords];
int iWord;
int redValue;
int greenValue;
int blueValue;
array<int> shuffledIndices[nWords];
int colorCode;
stimulus_data my_data;
int RT;
int acc;
int resp_button;
int correct_button;
int button_1_colorCode;
```

```
# Variable initialization

nBlocks = 4;

nTrials = 10;

words[1] = “LAMP”;

words[2] = “CHAIR”;

words[3] = “TEA CUP”;

words[4] = “PLATE”;

words[5] = “SPOON”;

words[6] = “DEATH”;

words[7] = “SEX”;

words[8] = “POO MAN”;

words[9] = “LOVE”;

words[10] = “HATE”;

loop iWord = 1 until iWord > nWords begin

        shuffledIndices[iWord] = iWord;

        iWord = iWord + 1;

end;

shuffledIndices.shuffle();

if (random(0, 1) == 1) then

        button_1_colorCode = 0;

else

        button_1_colorCode = 1;

end;


# Task

loop iBlock = 1 until iBlock > nBlocks begin

        my_text.set_font_color(255, 255, 255);

        my_trial.set_type(first_response);

        my_trial.set_duration(forever);

        if (button_1_colorCode == 0) then

                my_text.set_caption(“Press A for red and D for blue.\nPress either key to continue.”);

        else
```

```
		my_text.set_caption("Press A for blue and D for red. \nPress either key to continue.");
end;
my_text.redraw();
my_trial.present();

loop iTrial = 1 until iTrial > nTrials begin
		# Determine font color
		redValue = 0;
		greenValue = 0;
		blueValue = 0;
		colorCode = random(0, 1);
		if (colorCode == 0) then
			redValue = 255;
		else
			blueValue = 255;
		end;
		my_text.set_font_color(redValue, greenValue, blueValue);
		# Choose word
		iWord = 1 + mod(iTrial – 1, nWords);
		if (mod(iTrial – 1, nWords) == 0) then
			shuffledIndices.shuffle(1, nWords / 2);
			shuffledIndices.shuffle(nWords / 2 + 1, nWords);
			shuffledIndices.shuffle(2, nWords - 1);
		end;
		iWord = shuffledIndices[iWord];
		my_text.set_caption(words[iWord]);
		# Redraw stimulus and set up trial
		my_text.redraw();
		my_trial.set_type(first_response);
		my_trial.set_duration(forever);
		if (colorCode == button_1_colorCode) then
			correct_button = 1;
		else
```

```
        correct_button = 2;
end;
my_event.set_target_button(correct_button);
# Present trial
my_trial.present();
# Response collection
my_data = stimulus_manager.last_stimulus_data();
RT = my_data.reaction_time();
resp_button = my_data.button();
if (resp_button == correct_button) then
        acc = 1;
else
        acc = 0;
end;
 # Feedback
if (acc == 0) then
        my_text.set_font_color(255, 0, 0);
        my_text.set_caption("Wrong!");
        my_text.redraw();
else
        my_text.set_font_color(0, 255, 0);
        my_text.set_caption("Correct!");
        my_text.redraw();
end;
my_trial.set_type(fixed);
my_trial.set_duration(500);
my_trial.present();
# ITI
my_text.set_font_color(255, 255, 255);
my_text.set_caption("+");
my_text.redraw();
my_trial.set_type(fixed);
my_trial.set_duration(500);
```

```
                my_trial.present();
                # Save data
                ofile1.print(string(iBlock) + "\t");
                ofile1.print(string(iTrial) + "\t");
                ofile1.print(string(colorCode) + "\t");
                ofile1.print(string(iWord) + "\t");
                ofile1.print(string(RT) + "\t");
                ofile1.print(string(acc) + "\t");
                ofile1.print(string(resp_button) + "\n");
                # Increment iTrial counter
                iTrial = iTrial + 1;
        end;
        # Increment iBlock counter
        iBlock = iBlock + 1;
end;
ofile1.close();
```

And in openOutput.pcl:

```
string logpath = logfile_directory;
string fn = logpath + "EmoStroop_" + logfile.subject() + ".txt";
bool fe0 = file_exists(fn);
output_file ofile1 = new output_file;
ofile1.open_append(fn);
if (!fe0) then
        ofile1.print("Block\t");
        ofile1.print("Trial\t");
        ofile1.print("Color\t");
        ofile1.print("Word\t");
        ofile1.print("RT\t");
        ofile1.print("acc\t");
        ofile1.print("button\n");
end;
```

IV. Approach - Avoidance Task

This tutorial will have you building an Approach - Avoidance Task. These tasks measure biases in the fundamental action tendencies of approach and avoidance. For example, people with a fear of spiders will be faster to perform avoidance than approach responses in relation to spider pictures, relative to control pictures (Rinck and Becker, 2003). Interestingly, the task can be adapted for use as a training, and this seems to have real clinical applications in, e.g., addiction (Wiers et al., 2013) and anxiety (MacLeod and Mathews, 2012).

Make a new experiment called PushPull, with a scenario file and pcl files as in the previous tutorial. Pick a JPEG picture and save it to PushPull's Stimulus subdirectory as example.jpg.

23. Pictures

To use pictures, we need to introduce some new SDL code. The following code goes in PushPull.sce.

```
# Header

scenario = "PushPull";

response_matching = simple_matching;

active_buttons = 0;

default_background_color = 50, 50, 50;

default_font_size = 48;


# SDL

begin;

bitmap {

        filename = "example.jpg";

        preload = true;

} my_bitmap;

picture {

        bitmap my_bitmap;

        x = 0;

        y = 0;

} my_picture;

trial {

        trial_type = fixed;
```

```
        trial_duration = 1000;
        stimulus_event {
            picture my_picture;
            time = 0;
        } my_event;
} my_trial;
#PCL
begin_pcl;
set_random_seed(int(logfile.subject()));
include "PushPull.pcl";
```

PushPull.pce contains only the line

```
my_trial.present();
```

Run it to see the picture appear on screen; that's all there is to it in most cases! Instead of words, you can present pictures; you can make arrays of filenames, and select randomly from them; you can use if-them statements to select from different arrays of picture sets, and to determine correct responses to them. Note that usually you'll also want to use text stimuli, for block instructions for instance, so the SDL code will contain both text and bitmap pictures and trials. Pictures can also combine multiple graphics objects: this can be accessed as the pictures indexed picture parts, starting from index 1.

24. Animation

Animation involves moving, scaling or changing the content of a bitmap. Replace PushPull's PCL code with the following code for examples to show the basic functions involved with moving and scaling:

```
int iAnimStep;
int nAnimSteps;
double my_scale_factor;

nAnimSteps = 20;

# Moving horizontally
```

```
loop iAnimStep = 1 until iAnimStep > nAnimSteps begin
	my_picture.set_part_x(1, 15 * iAnimStep);
	my_trial.set_duration(100);
	my_trial.present();
	iAnimStep = iAnimStep + 1;
end;

# Moving vertically
loop iAnimStep = 1 until iAnimStep > nAnimSteps begin
	my_picture.set_part_y(1, 15 * iAnimStep);
	my_trial.set_duration(100);
	my_trial.present();
	iAnimStep = iAnimStep + 1;
end;

my_picture.set_part_x(1, 0);
my_picture.set_part_y(1, 0);

# Scaling
loop iAnimStep = 1 until iAnimStep > nAnimSteps begin
	my_scale_factor = 0.5 + double(iAnimStep) / double(nAnimSteps);
	my_bitmap.set_load_size(0.0, 0.0, my_scale_factor);
	my_bitmap.load();
	my_trial.set_duration(10);
	my_trial.present();
	iAnimStep = iAnimStep + 1;
end;
```

Check out the Presentation help function for details about the set_part_x and set_part_y functions. The important thing is that you can multiple elements to a picture, which are indexed in order. This index can be used to manipulate one element of a picture via functions like set_part_x.

Note the new variable type: **double**s. These contain digital representations of real numbers, such as 1.5, 32.003, and so on. If a function requires a double value, you can't use an integer variable as input; the integer-type value must be explicitly converted (or “cast”, in jargon) to a double-type value, as is done for my_scale_factor. The reverse casting is also common, but this usually loses information: an integer variable can't hold the fractional part of a number like 3.5. A common mistake is trying to assign integer numbers to doubles like this:

```
double example = 10;
```

This HAS to be written with a decimal point:

```
double example = 10.0;
```

Now, perhaps you noticed that the scaling animation was a little jerky; it was also very slow, although you'll only notice the difference later. This is because loading a bitmap takes significant time and so the animation timing is thrown off, since the bitmap was loaded in every animation step. Let's deal with this.

25. Using planes instead of bitmaps

Bitmaps are a very limited way to present visual stimuli. The next level of Presentation programming uses textures and planes. Instead of the flat matrix of colors that is a bitmap, a plane is a representation of a 2D object that can be manipulated in 3D space. A texture can be pasted onto the plane, and will be rotated and moved and skewed along with it. So, for zooming effects like in the Approach Avoidance Task, instead of having to reload or store a series of static bitmaps with different size, we can create a plane and scale it whenever and however we want.

So let's replace the my_bitmap object in the SDL code of the PushPull task, here:

```
bitmap {
        filename = "example.jpg";
preload = true;
} my_bitmap;
```

with this:

```
texture {
        filename = "example.jpg";
} my_texture;


plane {
```

```
        width = 400.0; height = 600.0;

        mesh_texture = my_texture;

        color = 0.0, 0.0, 0.0;

        emissive = 1.0, 1.0, 1.0;

} my_plane;
```

So this gets the texture from the file, and stick it onto the plane. We also need to replace the bitmap in the my_picture object, as follows:

```
picture {

        plane my_plane;

        x = 0;

        y = 0;

        z= 0;

} my_picture;
```

Now my_picture contains a lovely plane instead a stupid bitmap. So: we can go to the PCL code in PushPull.pcl and use the plane's capabilities. First, though, we need to go to the code for moving the picture around and replace the lines with the set_part_x and set_part_y bitmap-moving functions with the plane's equivalent, set_3dpart_xyz, as follows:

```
# Moving horizontally

loop iAnimStep = 1 until iAnimStep > nAnimSteps begin

        my_picture.set_3dpart_xyz(1, 15.0 * double(iAnimStep), 0.0, 0.0);

        my_trial.set_duration(100);

        my_trial.present();

        iAnimStep = iAnimStep + 1;

end;


# Moving vertically

loop iAnimStep = 1 until iAnimStep > nAnimSteps begin

        my_picture.set_3dpart_xyz(1, 0.0, 15.0 * double(iAnimStep), 0.0);

        my_trial.set_duration(100);
```

```
        my_trial.present();
        iAnimStep = iAnimStep + 1;
    end;

    my_picture.set_3dpart_xyz(1, 0.0, 0.0, 0.0);
    my_picture.set_3dpart_xyz(1, 0.0, 0.0, 0.0);
```

This is the old code that did the scaling:

```
    loop iAnimStep = 1 until iAnimStep > nAnimSteps begin
        my_scale_factor = 0.5 + double(iAnimStep) / double(nAnimSteps);
        my_bitmap.set_load_size(0.0, 0.0, my_scale_factor);
        my_bitmap.load();
        my_trial.set_duration(100);
        my_trial.present();
        iAnimStep = iAnimStep + 1;
    end;
```

Instead of loading a new bitmap at each iteration, we simply rescale the plane, as follows:

```
    # Scaling
    loop iAnimStep = 1 until iAnimStep > nAnimSteps begin
        my_scale_factor = 0.5 + double(iAnimStep) / double(nAnimSteps);
        my_plane.set_size(my_scale_factor * 500.0, my_scale_factor * 500.0);
        my_trial.set_duration(10);
        my_trial.present();
        iAnimStep = iAnimStep + 1;
    end;
```

Note how much faster the scaling movement is than in the previous version!

26. Putting it together

Now let's turn PushPull into the basis of a real Approach Avoidance Task. Typically, subjects are instructed to pull or pull pictures based on whether they're rotated clockwise or counterclockwise. Rotation is another nice thing we can do planes! Delete the PCL code in PushPull. We'll start with a simple design:

```
# Design control variables

int nTrials = 10;


loop int iTrial = 1 until iTrial > nTrials begin


    iTrial = iTrial + 1;

end;
```

Now let's add comments in the trial loop that will indicate what code we're going to add:

```
loop int iTrial = 1 until iTrial > nTrials begin

    # Set the picture to its default size and rotation


    # 1 s inter-trial interval


    # Randomly determine whether this trial requires a pull or a push movement


    # Push is clockwise, pull is counter-clockwise.

    # Rotate the picture accordingly;.


    # Wait for the correct response: A is push, D is pull.


    # Zoom the picture once the correct response is given.


    iTrial = iTrial + 1;

end;
```

Let's fill stuff in! First, we set up the picture to its starting size and unrotated state, and leave it on-screen for 1000 ms:

```
# Set the picture to its default size and rotation

double starting_size = 400.0;

my_plane.set_size(starting_size, starting_size);

my_picture.set_3dpart_rot(1, 0.0, 0.0, 0.0);


# 1 s inter-trial interval

my_picture.present();

wait_interval(1000);
```

The function wait_interval() is an easy way to control timing without having to set up and use a trial. Here we just put up the picture onscreen and then wait 1 s.

Now it gets interesting: the random rotation. Let's make a variable that can be -1 or 1:

```
# Randomly determine whether this trial requires a pull or a push movement

int pull = -1 + 2 * random(0, 1);
```

Note that random(0, 1) can be 0 or 1; so 2 * random(0, 1) can be 0 or 2; so -1 + 2 * random(0, 1) can be -1 or 1, as desired.

We can rotate the plane using its built-in function:

```
# Push is clockwise, pull is counter-clockwise.

# Rotate the picture accordingly.

my_picture.set_3dpart_rot(1, 0.0, 0.0, double(pull) * 45.0);
```

Remember planes live in 3D space. We're rotating the picture around its z-axis, the line sticking out of the screen. If pull is 1, then we rotate +45 degrees, i.e., clockwise. If pull is -1, we rotate -45 degrees, i.e., counter-clockwise. Note that we have to cast the integer pull to a double to match the required input to the set_3dpart_rot function. Sneak a look at things now by temporarily adding the line my_trial.present().

Now to add responses. Go to the Settings tab, and then the Response panel, to add the A and D keys, in that order; then set active_buttons = 2 in the SDL header. Now we can add the PCL code for responding, within a loop that continues until the correct response is given. Remember Presentation has a little magic monkey called the response_manager that takes care of things for us.

```
# Wait for the correct response: A is push, D is pull.
```

```
int acc = 0;
loop until acc == 1 begin
	my_trial.set_type(first_response);
	my_trial.set_duration(forever);
	my_trial.present();
	int my_response = response_manager.last_response();
	if pull == -1 && my_response == 1 then
		acc = 1;
	end;
	if pull == 1 && my_response == 2 then
		acc = 1;
	end;
end;
```

We're being a bit lazy by hard-coding the responses here, to keep things simple. Note that this only works because the trial won't end until a response is given: otherwise, my_response could still contain whatever response was given on a previous trial. After getting the correct response, we'll zoom in or out depending on the response:

```
# Zoom the picture once the correct response is given.
double smallest_size = 10.0;
double biggest_size = 800.0;
double current_size = starting_size;

double size_step_per_ms = 2.0;
int time_step_in_ms = 10;

loop until current_size < smallest_size || current_size > biggest_size begin
	my_plane.set_size(current_size, current_size);
	my_picture.present();
	wait_interval(time_step_in_ms);
```

```
        current_size = current_size + double(pull) * double(time_step_in_ms) *
size_step_per_ms;

    end;
```

So we're making an animation loop to scale the plane, like above. Now I've defined the smallest and largest size the picture should get. I've also defined the number of pixels the plane will grow or shrink per ms. So, since the interval being waited is 10, the plane will change by 10 * 2 pixels per iteration of the loop. Recall that pull is -1 or 1, so current_size grows or shrinks depending on its value.

So these are the bare bones of an Approach - Avoidance Task! But let's go a little further. Real Approach - Avoidance Tasks explore effects of the nature of the picture being pulled or pushed. Find yourself a picture you find attractive, and one you find aversive, and put them in the Stimulus directory. Now change the texture and place portion of the SDL code:

```
    texture {

        filename = "attractive.jpg";

    } attractive_texture;



    texture {

        filename = "aversive.jpg";

    } aversive_texture;



    plane {

        width = 400.0; height = 600.0;

        mesh_texture = attractive_texture;

        color = 0.0, 0.0, 0.0;

        emissive = 1.0, 1.0, 1.0;

    } my_plane;
```

Now we'll randomize the texture attached to the plane per trial, by adjusting the first part of the trial-loop code:

```
    # Set the picture to its default size, rotation, and texture

    double starting_size = 400.0;

    my_plane.set_size(starting_size, starting_size);

    my_picture.set_3dpart_rot(1, 0.0, 0.0, 0.0);
```

```
int attractive = random(0, 1);

if attractive == 0 then

        my_plane.set_texture(aversive_texture);

else

        my_plane.set_texture(attractive_texture);

end;
```

So now there's a factorial design for the trial conditions: the picture is aversive or attractive, and the correct response is push or pull. One would expect that aversive pictures are easier to push than to pull, and attractive pictures are easier to pull than to push. Let's make this testable by saving the data. At the top of the PCL file, put in the code for opening an output file:

```
# Open file for saving

string logpath = logfile_directory;

string fn = logpath + "PushPull_" + logfile.subject() + ".txt";

bool fe0 = file_exists(fn);

output_file ofile1 = new output_file;
```

and close it at the bottom of the PCL file:

```
ofile1.close();
```

At the end of the trial-loop, write the important variables to the output file:

```
ofile1.print(string(clock.time()) + "\t");

ofile1.print(string(attractive) + "\t");

ofile1.print(string(pull) + "\t");

ofile1.print(string(RT) + "\t");

ofile1.print(string(errors) + "\n");
```

We have two variables we still need to define: reaction time RT, and how often the subject pressed the wrong button before hitting the correct response. Let's create those now, in the code for acquiring responses:

```
# Wait for the correct response: A is push, D is pull.

int acc = 0;

int errors = 0;
```

```
int RT;
int stopwatch_start = clock.time();
loop until acc == 1 begin

        my_trial.set_type(first_response);
        my_trial.set_duration(forever);
        my_trial.present();

        int stopwatch_stop = clock.time();

        int my_response = response_manager.last_response();
        if pull == -1 && my_response == 1 then
                acc = 1;
        end;
        if pull == 1 && my_response == 2 then
                acc = 1;
        end;

        if acc == 0 then
                errors = errors + 1;
        end;
        RT = stopwatch_stop - stopwatch_start;
end;
```

Note how we get the reaction time: a stopwatch is started immediately before the loop, and stopped immediately after the trial ends, which happens when a subject responds. If the response was incorrect, errors is incremented by one, and the loop starts again; but, importantly, the stopwatch_start time remains the same. Thus the RT gives the time taken to reach a correct response.

Currently, the task contains some arbitrary confounds, and we have some hard-coded filth in there. Let's adjust that, and while doing so add instructions. First, add a variable for randomizing the rotation - response mapping to the design control variables:

```
# Design control variables

int nTrials = 10;

int rotationRandomizer = -1 + 2 * random(0, 1);
```

We'll use the rotationRandomizer to adjust how the rotation maps to push versus pull stimuli:

```
# Push and pull are clockwise and counter-clockwise, depending on the randomizer.

# Rotate the picture accordingly.

my_picture.set_3dpart_rot(1, 0.0, 0.0, double(rotationRandomizer * pull) * 45);
```

So now, if pushPullRandomizer is -1, the mapping from push / pull responses to the direction of rotation will be reversed.

Now we'll randomize the button to push / pull response mapping:

```
# Design control variables

int nTrials = 10;

int rotationRandomizer = -1 + 2 * random(0, 1);

array<int> pushPullButton[2];

pushPullButton[1] = 1;

pushPullButton[2] = 2;

pushPullButton.shuffle();
```

and in the code for evaluating responses:

```
int my_response = response_manager.last_response();

if pull == -1 && my_response == pushPullButton[1] then

    acc = 1;

end;

if pull == 1 && my_response == pushPullButton[2] then

    acc = 1;

end;
```

Now of course you don't know what's what without learning from trial and error. But we can use the randomizing variables to set up the correct instructions. Add this code just above the trial loop (but after the variable definitions of course):

```
# Instructions

string instructions = "Instructions\n\n";

if rotationRandomizer == 1 then

        instructions = instructions + "Pull counterclockwise,\npush clockwise.\n\n";

else

        instructions = instructions + "Push counterclockwise,\npull clockwise.\n\n";

end;

if pushPullButton[1] == 1 then

        instructions = instructions + "Press A to push, press D to pull\n\n"

else

        instructions = instructions + "Press D to push, press A to pull\n\n"

end;

instructions = instructions + "Press either key to start";

my_text.set_caption(instructions);

my_text.redraw();

my_text_trial.present();
```

Try it out!

Now to finish the task up, we'll adjust it to draw from a set of attractive and a set of aversive stimuli. Find two more pictures to add to the Stimulus directory. Call the three attractive pictures attractive1.jpg, attractive2.jpg, and attractive3.jpg, and the three aversive pictures aversive1.jpg and so on. Now, we're going to use only one texture variable, and load the texture from the appropriate file during the inter-trial interval. In SDL, change the code as follows:

```
begin;

texture {

        filename = "attractive1.jpg";

} my_texture;
```

```
plane {
        width = 400.0; height = 600.0;
        mesh_texture = my_texture;
        color = 0.0, 0.0, 0.0;
        emissive = 1.0, 1.0, 1.0;
} my_plane;
```

In PCL, we're going to load the texture during the intertrial interval. Since this is a slow operation, we're going to make sure the interval isn't longer than we want it to be by using clock.time(). Change the start of the trial loop code as follows:

```
# Set the picture to its default size and rotation.
double starting_size = 400.0;
my_plane.set_size(starting_size, starting_size);
my_picture.set_3dpart_rot(1, 0.0, 0.0, 0.0);

# Randomize whether the picture will be aversive or attractive.
int attractive = random(0, 1);
string base_filename;
if attractive == 0 then
        base_filename = "aversive";
else
        base_filename = "attractive";
end;
int stimulus_ID = random(1, 3);
string texture_filename = base_filename + string(stimulus_ID) + ".jpg";

# 1 s inter-trial interval
int loading_time_start = clock.time();
my_texture.set_filename(texture_filename);
```

my_texture.load();

int loading_time_end = clock.time();

int time_lost = loading_time_end - loading_time_start;

my_picture.present();

wait_until(loading_time_start + 1000);

So: we set up the filename based on the stimulus type - aversive or attractive - and pick one of the stimuli in that set at random. Around performing the my_texture.load() command, we put clock.time() commands so we know how long the operation took. We then wait_until the time point one second after the time point before loading started, so it doesn't matter how long that took: the task will simply fill in the remaining time.

At this point you pretty much have a fully working Approach - Avoidance Task (albeit with way too few trials, no practice block, and not all the variables being saved). You could think of some stimulus categories for which you hypothesize that people might have an approach or avoidance bias, find nice exemplar stimuli, and go for it.

V. Conditioning Task

This section will use a very basic task to illustrate two useful capabilities of Presentation: templates and layered graphics. The task will involve the appearance of angry, axe wielding men at predictable times. Such stimuli could evoke *action tendencies* - patterns of physiological and cognitive changes that - unless somehow rendered dysfunctional - support responding in a way optimized for survival (Frijda, 1988).

Set up a task directory ImplLearn as usual, with subdirectories Stimulus and Log. Open Presentation and save the new experiment as ImplLearn.exp in the ImplLearn directory; in the Editor, create a ImplLearn.sce file saved to the CAPT directory; add the subdirectories and scenario in the Scenarios tab. Add the space bar key as the only response in the Response panel in the Settings tab: click Keyboard under devices, the click on the Buttons list and press S a few times until the selection jumps to SPACE. Click Use to add it to the active buttons.

Now we need some stimulus files and graphics software that has to be more advanced than the standard Paint. A good option (with a bad name) is Paint.NET, which is free for download at http://www.getpaint.net/. Choose a picture that will serve as a background, called background.jpg; something filled with lots of detail and colors. Next, use Paint.NET to make two simple stick-figure drawings. Make one angry guy holding an axe, and one happy guy holding a flower; it doesn't matter how silly they look, but use a thick brush width to draw the lines. **Save these figures in the PNG format.**

27. Layers

Start off the scenario file like this:

 # Header

```
scenario = "ImplLearn";
response_matching = simple_matching;
active_buttons = 1;
default_background_color = 50, 50, 50;
default_font_size = 48;

begin;

texture {
        filename = "background.jpg";
} texture_background;

plane {
        width = 800.0; height = 600.0;
        mesh_texture = texture_background;
        color = 0.0, 0.0, 0.0;
        emissive = 1.0, 1.0, 1.0;
} plane_background;

texture {
        filename = "happy.png";
} texture_figure;

plane {
        width = 200.0; height = 200.0;
        mesh_texture = texture_figure;
        color = 0.0, 0.0, 0.0;
```

```
        emissive = 1.0, 1.0, 1.0;
} plane_figure;

picture {
        plane plane_background;
        x = 0;
        y = 0;
        z = 0;
        plane plane_figure;
        x = 0;
        y = 0;
        z = 0;
} my_picture;

trial {
        trial_type = fixed;
        trial_duration = 5000;
        stimulus_event {
                picture my_picture;
                time = 0;
        } my_event;
} my_trial;

begin_pcl;

my_trial.present();
```

If you now run this, you'll see the happy guy superimposed on top of the background. But he'll be in his little white box: ugly! However, PNG files can be made transparent. Go to Paint.NET and open

happy.png. Select the Magic Wand tool and click somewhere on the white of the figure. This will select around the happy guy. Press delete: now the white has been replaced by a checkerboard pattern indicating transparency. Save the figure and see what the trial looks like now: the little guy will be standing in the background. Using layers of transparent stimuli like this allows complex stimuli to be built up flexibly.

28. Templates

We can clean up the SDL code by dividing it over files, called templates. In the ImplLearn task, we could put all the graphical objects in their own template file as follows. First make a new file called graphics_SDL.tem, saved to the task directory, and put all the graphics code in it:

```
texture {
        filename = "background.jpg";
} texture_background;


plane {
        width = 800.0; height = 600.0;
        mesh_texture = texture_background;
        color = 0.0, 0.0, 0.0;
        emissive = 1.0, 1.0, 1.0;
} plane_background;


texture {
        filename = "happy.png";
} texture_figure;


plane {
        width = 200.0; height = 200.0;
        mesh_texture = texture_figure;
        color = 0.0, 0.0, 0.0;
        emissive = 1.0, 1.0, 1.0;
} plane_figure;
```

```
picture {
        plane plane_background;
        x = 0;
        y = 0;
        z = 0;
        plane plane_figure;
        x = 0;
        y = 0;
        z = 0;
} my_picture;
```

In the scenario file, this code can then be replaced by the line

```
TEMPLATE "graphics.tem";
```

This keeps the SDL code a lot tidier and when tasks get more complex it's very helpful to have the code organized over files, although this does requires systematicity.

Let's put the little guys in a task. Replace the end of the scenario file with

```
include "ImplLearn.pcl";
```

and create that PCL file. We're going to create a repeating sequence of happy and angry guys. The task is to press the space bar as quickly as possible when the angry guy appears. The sequence is defined as follows:

```
array<int> happy_sequence[6];
loop int n = 1 until n > happy_sequence.count() begin
        int stim_type = (n - 1) % 3;
        if stim_type == 0 then
                happy_sequence[n] = 0;
        else
                happy_sequence[n] = 1;
        end;
        n = n + 1;
```

```
end;

happy_sequence.shuffle();
```

This will create a random sequence consisting of two zeros and four ones. We then make an infinite trial loop, which runs over the happy - angry sequence, using the modulus trick:

```
int iTrial = 1;

loop until false begin

    int index = 1 + (iTrial - 1) % happy_sequence.count();

    int happy = happy_sequence[index];



    iTrial = iTrial + 1;

end;
```

Depending on the happy variable, we'll load the angry or happy guy into the texture on the plane following the line defining it, and then show him using the trial:

```
string filename;

if happy == 0 then

    filename = "angry.png";

else

    filename = "happy.png";

end;

texture_figure.set_filename(filename);

texture_figure.load();


my_trial.set_duration(1000);

my_trial.present();
```

We want to have some inter-trial interval, where we only see the background. Go to the handy graphics.tem, and at the bottom add:

```
picture {

    plane plane_background;
```

```
        x = 0;

        y = 0;

        z = 0;

    } my_picture_only_background;
```

In the PCL file, add the following code to show just the background after the trial presentation:

```
    my_picture_only_background.present();

    wait_interval(1000);
```

29. Go - Nogo responses

Now add response evaluation code around trial presentation: we want to have subjects only respond when the angry guy appears. We can do that by checking the number of logged responses in the response_manager before the trial and after the trial.

```
    int nResponsesPre = response_manager.total_response_count();

    my_trial.set_duration(1000);

    my_trial.present();

    int nResponsesPost = response_manager.total_response_count();

    int acc = 1;

    if nResponsesPost > nResponsesPre then

        if happy == 0 then

            acc = 1;

        else

            acc = 0;

        end;

    else

        if happy == 0 then

            acc = 0;

        else

            acc = 1;

        end;

    end;
```

Make sure you can follow the go - nogo logic for calculating accuracy. Let's give feedback on errors: you can be too late to respond to the angry guy, or you can give a false positive to the happy guy. We first make a basic text-presenting trial in the SDL:

```
trial {
    trial_type = fixed;
    trial_duration = 250;
    stimulus_event {
        picture {
            text {
                caption = "Default";
            } my_text;
            x = 0;
            y = 0;
        } my_text_picture;
    } my_text_event;
} my_text_trial;
```

Now in PCL we add, after the response evaluation:

```
if acc == 0 then
    if happy == 0 then
        my_text.set_caption("Too late!");
    else
        my_text.set_caption("Incorrect response!");
    end;
    my_text.set_font_color(255, 0, 0);
    my_text.redraw();
    my_text_trial.present();
end;
```

Run the task! If you play it long enough, you should learn when the angry guy appears and be able to respond more quickly to him.

30. Titration

Of course, why would you bother to respond quickly, there's not much time pressure. So let's add some. Make a new variable response_window at the top of the PCL file:

```
int response_window = 1000;
```

Use this variable to determine the trial_duration:

```
my_trial.set_duration(response_window);
```

Now, every time you respond quickly enough to the angry guy, we'll shorten the response window by a certain percent, by adding this code after the code for feedback presentation:

```
if acc == 1 && happy == 0 then
    response_window = int(0.5 * double(response_window));
end;
```

If you start missing responses on angry guy, we'll loosen up the window:

```
if acc == 0 && happy == 0 then
    response_window = int(2.0 * double(response_window));
end;
```

This process of making the task more or less difficult is called titration. This version of titration will keep performance fluctuating around 50% accuracy. Note that the steps in this titration are far bigger than usual: something like 10% would be more reasonable outside illustrative purposes.

A subject performing this task for a while would be expected to learn, possibly not consciously, when the angry guy appears, since the sequence repeats. A fun thing to do would be to shuffle the happy - angry sequence after a while and see what happens. The task would also lend itself to psychophysiological measurements, to study preparatory states preceding the implicitly expected negative stimulus.

Now, let’s make some more use of the ability to layer visual stimuli. We’ll change the task into a conditioning task, in which the background predicts the likelihood of the axeman appearing. First, make a copy of the previous task’s directory, and open the Experiment file in the new directory. Make sure the Log, Stimulus, and scenario files have correctly updated to refer to the files in the new directory. Now find a second background picture and add it to the Stimulus directory as background2.jpg. Rename the first picture as background1.jpg. Now adjust the stimulus-related code in graphics.tem so we have two backgrounds:

```
texture {
    filename = "background1.jpg";
} texture_background1;
```

```
plane {
        width = 800.0; height = 600.0;
        mesh_texture = texture_background1;
        color = 0.0, 0.0, 0.0;
        emissive = 1.0, 1.0, 1.0;
} plane_background1;

texture {
        filename = "background2.jpg";
} texture_background2;

plane {
        width = 800.0; height = 600.0;
        mesh_texture = texture_background2;
        color = 0.0, 0.0, 0.0;
        emissive = 1.0, 1.0, 1.0;
} plane_background2;
texture {
        filename = "happy.png";
} texture_figure;
plane {
        width = 200.0; height = 200.0;
        mesh_texture = texture_figure;
        color = 0.0, 0.0, 0.0;
        emissive = 1.0, 1.0, 1.0;
} plane_figure;
```

```
picture {
        plane plane_background1;
        x = 0;
        y = 0;
        z = 0;
        plane plane_figure;
        x = 0;
        y = 0;
        z = 0;
} my_picture;

picture {
        plane plane_background1;
        x = 0;
        y = 0;
        z = 0;
} my_picture_only_background;
```

Note that there's a judgment call here: we could also create one background plane, and reload the appropriate file in PCL code, instead of the current approach of using two planes created at the start of the experiment.

Now to change the PCL code, in ImplLearn.pcl, to change the conditions under which the axeman appears. Remove the following code, which we needed for sequential version:

```
array<int> happy_sequence[6];
loop int n = 1 until n > happy_sequence.count() begin
        int stim_type = (n - 1) % 3;
        if stim_type == 0 then
                happy_sequence[n] = 0;
        else
```

```
            happy_sequence[n] = 1;
        end;
        n = n + 1;
    end;
    happy_sequence.shuffle();
```

as well as

```
    int index = 1 + (iTrial - 1) % happy_sequence.count();
    int happy = happy_sequence[index];
```

Now, as the first step in the loop, we randomly select a background, and slide it into the background position in the layered stimuli:

```
    int current_background = random(1, 2);
    if current_background == 1 then
        my_picture_only_background.set_3dpart(1, plane_background1);
        my_picture.set_3dpart(1, plane_background1);
    else
        my_picture_only_background.set_3dpart(1, plane_background2);
        my_picture.set_3dpart(1, plane_background2);
    end;
```

Although it doesn't matter here, note that Presentation indexes normal and 3D picture parts of pictures separately! So if you also used bitmaps in the pictures, it wouldn't affect the index of planes.

Now we determine the chance of the little guy being happy or angry based on the background picture. Under background1, he's going to be happy 80% of the time; under background2, there's an 80% chance he'll come at you with his axe.

```
    int chance_of_angry;
    if current_background == 1 then
        chance_of_angry = 20;
    else
        chance_of_angry = 80;
    end;
```

We'll use this chance to determine the actual value of happy by throwing our handy virtual 100-sided die:

```
int die = random(0, 99);

if die < chance_of_angry then

        happy = 0;

else

        happy = 1;

end;
```

Finally, we now have to show the background-only picture before the background + little guy picture. Move the lines for that up, so that they come right after the above code for selecting the happy / angry stimulus:

```
my_picture_only_background.present();

wait_interval(1000);
```

Since that move stole our inter-trival interval, let's quickly add a simple fixation cross to appear first:

```
my_text.set_caption("+");

my_text.set_font_color(255, 255, 255);

my_text.redraw();

my_text_trial.set_duration(750);

my_text_trial.present();


my_picture_only_background.present();

wait_interval(1000);
```

Since we're recycling our my_text_trial and changing its duration, we have to set its duration to what we wanted, in the chunk of code where we use it for feedback:

```
if acc == 0 then

        if happy == 0 then

                my_text.set_caption("Too late!");

        else

                my_text.set_caption("Incorrect resposne!");
```

```
        end;

        my_text.set_font_color(255, 0, 0);

        my_text.redraw();

        my_text_trial.set_duration(750);

        my_text_trial.present();

end;
```

The task now should condition subjects to associate the second background with the chance that the angry guy appears. Of course, in a real experiment which of the two backgrounds is the one which predicts danger would be counterbalanced over subjects. Make this change as an exercise, as was done for the push / pull button in the Approach Avoidance Task for example.

A general theoretical question is how to conceive of what's going on with tasks like this: should you describe things behavioristically, i.e. very carefully and precisely in terms of objective task characteristics and response patterns, or focus more on models of information processing, cognition and the function of physiological states for action tendencies? The current version of the task could, for instance, be seen as providing information to an adaptive system that determines which preparatory state to activate as a response to a context cue.

VI. Visual Working Memory Task

In this task, I'll illustrate how to use some trigonometry and line graphics to flexibly generate stimuli for a visual working memory task. Working memory can be defined (more or less as by, e.g., Kane and Engle, 2003) as short-term storage of information plus executive functions operating on that information, or as the representation of information in a way that it can be operated upon. In true working memory tasks, the executive function aspect of working memory is manipulated; here, we'll mostly be looking at short-term storage, although followed by a comparison operation.

Set up a VisWM directory as usual. We're not going to need much SDL code this time. Start off with this, in VisWM.sce:

```
begin;


picture {

        text {

                caption = "+";

        } my_text;

        x = 0; y = 0;

} my_picture;
```

```
begin_pcl;

include "subfuncs.pcl";

loop until false begin

end;
```

We're going to start off with some structure now, by using **subfunctions,** defined in their own file, subfuncs.pcl. Subfunctions are re-usable chunks of code, which can be given input and can return output. We're going to use a subfunction to generate the points of an object that we want the subject to hold in short-term memory. The tricky thing in Presentation is that subfunctions have access to variables defined in the code before their definition. Unlike other programming languages, they're not encapsulated, that is, they code within a function isn't isolated from the variables outside it. This can be confusing, but sometimes handy since it avoids having to pass a lot of input arguments. So, immediately before the *include "subfuncs.pcl";* line, we define two arrays, for X and Y coordinates.

```
array<double> x[8];
array<double> y[8];
```

In subfuncs.pcl, here comes the subfunction for generating the spokes of a random shape, technically called a Wonky Shape:

```
sub generatePoints begin
    int nPoints = xvec.count();
    double stepAngle = 2.0 * pi_value / double(nPoints);
    loop int iPoint = 0 until iPoint >= nPoints begin
        double radius = 0.25 + 0.3 * random();
        double angle = double(iPoint) * stepAngle;
        double this_x = radius * cos(angle);
        double this_y = radius * sin(angle);
        xvec[iPoint + 1] = this_x;
        yvec[iPoint + 1] = this_y;
```

```
            iPoint = iPoint + 1;
        end;
    end;
```

What this does, is loop around a circle. For every point, we have a given angle, but the radius is random. We can then draw lines connecting the points, to get a wonky polygon that subjects shouldn't have any verbal label for. We call this subfunction at the start of every trial:

```
    loop until false begin
        generatePoints();
    end;
```

31. Line graphics

But now we want to draw the thing. Before the line including the subfunction file, add these variables for line graphics:

```
    int screenHeight = display_device.height();
    double screenScaler = double(screenHeight) / 2.0;
    line_graphic my_line_graphic = new line_graphic;
    my_line_graphic.redraw();
    my_picture.add_part(my_line_graphic, 0, 0);
```

Note that we can get the size of the screen from the display_device monkey hanging around automatically. The line_graphic object is something like a bitmap or a plane, but consist of lines that can be used to draw a vectorized shape. The redraw() is a little silly here: Presentation throws up an error if it encounters a picture with an undrawn line_graphic. We'll do something useful with the line_graphic in a new subfunction in subfuncs.pcl:

```
    sub draw_wonky_shape(int r, int g, int b) begin
        my_line_graphic.clear();
        loop int n = 1 until n > xvec.count() begin
            my_line_graphic.set_next_line_color(r, g, b, 255);
            double screen_x_1;
            double screen_y_1;
            if n > 1 then
                screen_x_1 = screenScaler * xvec[n - 1];
                screen_y_1 = screenScaler * yvec[n - 1];
```

```
            else
                screen_x_1 = screenScaler * xvec[xvec.count()];
                screen_y_1 = screenScaler * yvec[yvec.count()];
            end;
            double screen_x_2 = screenScaler * xvec[n];
            double screen_y_2 = screenScaler * yvec[n];
            my_line_graphic.add_line(screen_x_1, screen_y_1, screen_x_2, screen_y_2);
            n = n + 1;
        end;
        my_line_graphic.redraw();
    end;
```

Note that although I called this function draw_... it doesn't put anything on screen yet: for that the picture has to be presented. We can call the draw function in the trial code to generate a sequence of random shapes:

```
    loop until false begin
        generatePoints();
        draw_wonky_shape(255, 255, 255);
        my_picture.present();
        wait_interval(500);
    end;
```

Have a look what happens when you run this: you should see how the calculations in generatePoints result in a random sequence of polygons.

32. Rotation of points of a polygon

To make a task of this, we could show a Wonky Shape, then present a delay period, and then show a second shape that's either the same shape but rotated, or a different shape.

Let's add a fixation cross to the start of the trial code; note we have to clear and redraw the line graphic to get rid of it.

```
    # Clear lines and present fixation cross
    my_line_graphic.clear();
    my_line_graphic.redraw();
```

```
my_picture.present();

wait_interval(2500);
```

After this we already have the code that generates and presents the initial shape. After that, we add this delay period:

```
# Delay period

my_line_graphic.clear();

my_line_graphic.redraw();

my_picture.present();

wait_interval(1500);
```

And finally, we decide whether to rotate or replace the shape:

```
# Rotate or replace stimulus

int rotate = random(0, 1);

update_points(rotate);

draw_wonky_shape(0, 255, 0);
```

We called a new subfunction here - update_points - that we need to define (in subfuncs.pcl) as follows:

```
sub rotate begin

    double rotation_angle = (0.25 + 1.5 * random()) * pi_value;

    int nPoints = xvec.count();

    loop int iPoint = 0 until iPoint >= nPoints begin

        double this_x = xvec[iPoint + 1];

        double this_y = yvec[iPoint + 1];


        double radius = sqrt(pow(this_x, 2.0) + pow(this_y, 2.0));

        double angle = arctan2(this_y, this_x);


        angle = angle + rotation_angle;
```

```
            this_x = radius * cos(angle);
            this_y = radius * sin(angle);
            xvec[iPoint + 1] = this_x;
            yvec[iPoint + 1] = this_y;

            iPoint = iPoint + 1;
        end;
    end;

    sub update_points(int rotate) begin
        if rotate == 0 then
            generatePoints();
        else
            rotate();
        end;
    end;
```

We have to define rotate before update_points, because it has to be present before being called in update points. The rotate function specifies a random rotation, rotation_angle, that's set up not to be too close to zero degrees or a full circle, which would give almost the same figure. Then we go round the points, and use trigonometry to reverse-engineer the radius and angle from the x and y coordinates. We can then easily add the rotation_angle to the original angles and recalculate the new x and y positions. (For the connoisseurs - yes, this could be done via matrix algebra, but I think this way might be more informative.)

Finally, we just need to add response code: let's agree to press F for different shapes and J for rotated shapes. Add the keys in that order to the Response panel, and add *active_buttons = 2;* to the top of the scenario file. Then add a basic text trial to the SDL code:

```
    trial {
        trial_type = first_response;
        trial_duration = forever;
        stimulus_event {
            picture {
```

```
                    text {
                        caption = "Default";
                    } my_text_response;
                    x = 0;
                    y = 0;
                } my_text_picture;
            } my_text_event;
        } my_text_trial;
```

Then add the response code after the presentation of the rotated or different stimulus, followed by feedback so you know when you get it wrong.

```
        # Response acquisition
        my_text_response.set_caption("Different (F) or Rotated (J)?");
        my_text_response.redraw();
        my_text_trial.present();

        # Response evaluation
        int acc;
        int resp_button = response_manager.last_response();
        if resp_button == (rotate + 1) then
            acc = 1;
        else
            acc = 0;
        end;

        # Feedback
        if acc == 0 then
            my_text_response.set_caption("Incorrect.\n\nPress a key to continue.");
            my_text_response.redraw();
```

```
        my_text_trial.present();

    end;
```

And there we have it! The task consists of the classic trio of encoding of the shape into working memory; its maintenance; and its recall for comparison to a rotated or different shape. Something to add could be a varying duration of the maintenance period, or the addiction of a secondary task or distracting stimuli. This would get into interactions between emotional stimuli and the "hot" processes they evoke, and the more "cold" working memory functions taxed by an abstract working memory task such as this (Gladwin and Figner, 2014).

VII. Conclusion

Hopefully you've reached this conclusion after actually following the process of building up and, sometimes, breaking down and rebuilding the programs; if so, you'll have an idea of how to proceed through the steps from an idea for a task through implementing chunks of the task to the final program. The building blocks of the tasks developed in this book should be applicable, with some mixing and matching, to a very broad set of psychological tasks. Although this book was concerned with Presentation, figuring out how to build the analogues of these tasks might be of use when learning a different language such as E-Prime or Python as well.

Googling the title of this book should find you a webpage for the book, including errata and the complete versions of the canonical tasks worked through here.