**Overview**

**1. Setting up information needed for the experiment**

You set up everything you ever need, information about tasks. All the information are in structural arrays.

Example:

**Stimulus** is a struct array that contains information about the stimuli.

stimulus.scan

stimulus.plots

stimulus.noeye

…

stimulus.colors

…

stimulus.responseKeys

stimulus.calib

stimulus.theta\_

stimulus.thetas

stimulus.backgroundLab

…

stimulus.fixWidth

stimulus.targetWidth

**Allow to have different stimulus dot patches:**

stimulus.patchEcc

stimulus.patches.color

stimulus.patches.dots.dir

stimulus.patches.theta

stimulus.patches.ecc

stimulus.patches.xcenter

stimulus.patches.ycenter

**Define the order/index of different “segments” (seg) during a trial:**

stimulus.seg.iti = 1

stimulus.seg.fix = 2

stimulus.seg.cue = 3

stimulus.seg.isi = 4

stimulus.seg.stim = 5

stimulus.seg.delay = 6

stimulus.seg.resp = 7

stimulus.seg.feedback = 8

**dots** is a struct array about dots

dots.density

dots.speed

dots.maxAlive

dots.maxX

dots.maxY

dots.dotScale

dots.

**Task** is a struct array containing information about task.

You can have multiple tasks. To set up, use task{1} task{2} task{3} and etc. For each task, you can have multiple phases, for example task 1 has two phases, task{1}{1} and task{1}{2}; task 2 has three phases, task{2}{1}, task{2}{2}, task{2}{3}.

“Phase” here is referred to different phases of a task. For example, a task has two phases: in the first phase, randomly moving dots are shown. In phase 2, coherently moving dots are shown. Then you can set the phase-1 in task{1}{1} and phase-2 in task{1}{2}.

task{1}{1} = struct;

task{1}{1}.segmin = [0 inf 0.5 0.75 inf 1 4 0.75];

task{1}{1}.segmax = [2 inf 0.5 0.75 inf 1 4 0.75];

if stimulus.practice==1

task{1}{1}.segmin(stimulus.seg.cue) = 1;

task{1}{1}.segmax(stimulus.seg.cue) = 1;

task{1}{1}.segmin(stimulus.seg.isi) = 1;

task{1}{1}.segmax(stimulus.seg.isi) = 1;

task{1}{1}.segmin(stimulus.seg.resp) = 6;

task{1}{1}.segmax(stimulus.seg.resp) = 6;

task{1}{1}.segmin(stimulus.seg.feedback) = 1.5;

task{1}{1}.segmax(stimulus.seg.feedback) = 1.5;

elseif stimulus.practice==2

task{1}{1}.segmin(stimulus.seg.cue) = 1;

task{1}{1}.segmax(stimulus.seg.cue) = 1;

task{1}{1}.segmin(stimulus.seg.isi) = 1;

task{1}{1}.segmax(stimulus.seg.isi) = 1;

end

task{1}{1}.waitForBacktick = 1;

task{1}{1}.getResponse = zeros(1,length(task{1}{1}.segmin));

task{1}{1}.getResponse(stimulus.seg.resp) = 1;

% task{1}{1}.getResponse = [0 0 0 0 0 0 1 0]

% a vector containing 1s and 0s used as index for whether to detect a response

if stimulus.scan==1

task{1}{1}.numTrials = Inf; %why set number of trials to Inf if scan=1?

else

task{1}{1}.numTrials = 40;

end

**2. Initialize task**

Once they are set up, the next step is to "initialize" task by calling initTask. initTask calls back functions that draw stimuli and detect response.

**3. Enter the main task loop**

Then you enter the main task loop where you "update" status until the end of the experiment.

while (phaseNum <= length(task{1})) && ~myscreen.userHitEsc

% This function serves to run experimental tasks

[task{1}, myscreen, phaseNum] = updateTask(task{1},myscreen,phaseNum);

% flip screen

myscreen = tickScreen(myscreen,task);

end

initTask and updateTask are the most important functions for running the task. They seem to do a lot of heavy lifting and I have to admit after looking at them I still don’t have a clear sense what exactly they are doing.

What do I need to specify in order to run my task?

Stimulus:

Dot size

Dot starting location: x,y

Dot speed

Dot destination (where it will pass through the edge of the aperture): x,y

Dot trajectory

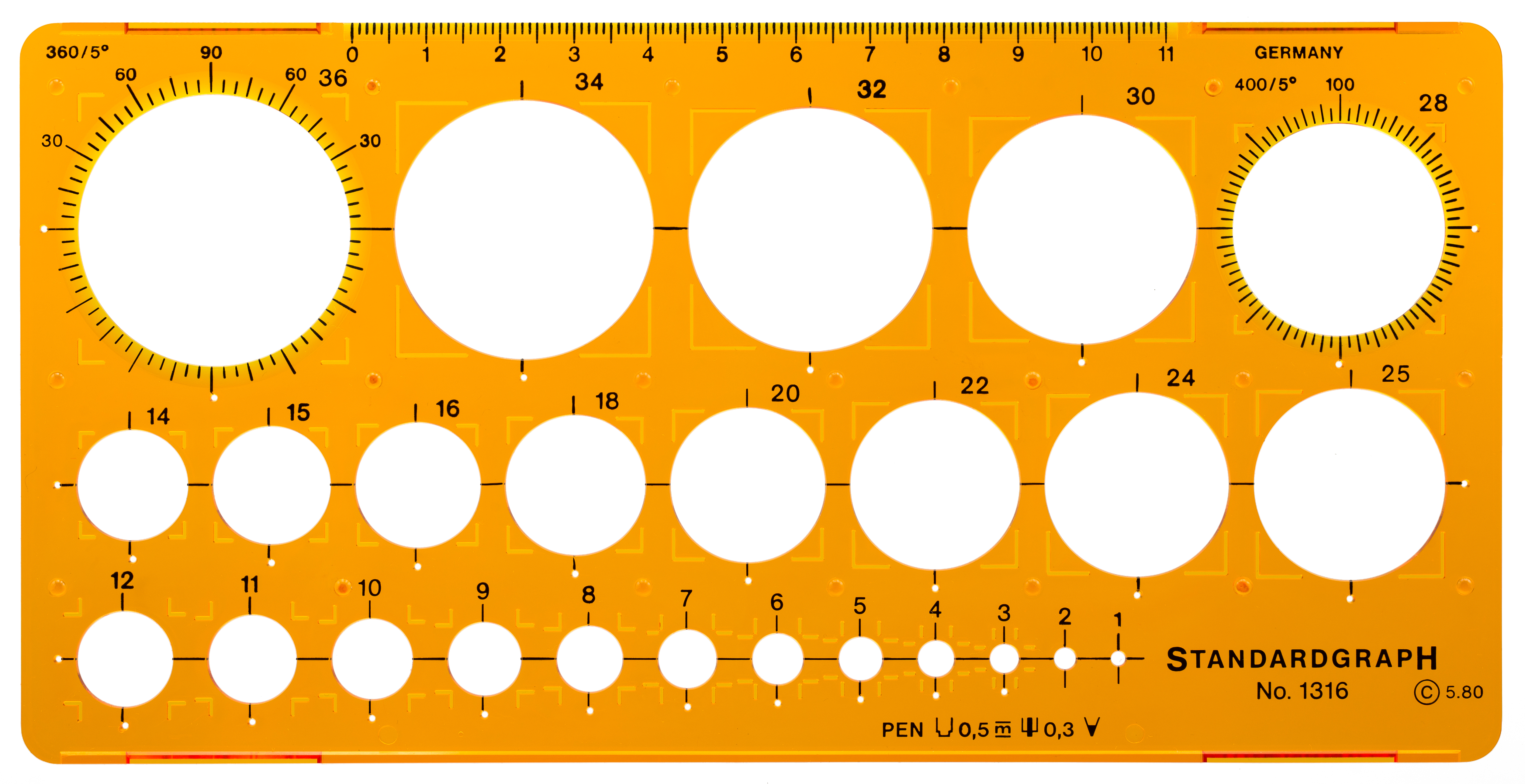
How to calculate trajectory: calculate distance between start and destination. Calculate distance in x. get theta using theta = acos(x\_dist/dist). Define radius – how much dot moves in between frames – as radius\_move. Next frame location: x\_next = x\_now + radius\_move\*cos(theta); y\_next = y\_now + radius\_move\*sin(theta).

I want the number of dots on the screen at any given moment to be the same. So when a dot goes out, a new dot has to fill in.

key functions:

> stencils (繪圖模板; 在這裏的意思跟aperture類似) related

circle stencils



Example:

mglOpen;

mglScreenCoordinates;

%Draw an oval stencil

mglStencilCreateBegin(1);

mglFillOval(300,400,[100 100]);

mglStencilCreateEnd;

mglClearScreen;

notes\_afcom: ~line 266.

> initDots

% initialize dots according to the specifications above.

% mostly on locations of these cue dots.

**initScreen**

myscreen = initScreen;

* This call will handle opening up of the screen with appropriate parameters. If you want to add or change screen parameters, like screen size, call mglEditScreenParams.

**initTask**

[task{1}{phaseNum}, myscreen] = initTask(task{1}{phaseNum},myscreen,@startSegmentCallback,@screenUpdateCallback,@getResponseCallback,@startTrialCallback,@endTrialCallback,[]);

* initTask is a mgl function that initializes task – a structural array containing important information about the experimental task – for running the task.

* inputs are task and myscreen, both are struct array, and function handles through function call-back @function\_name.
* When you call back a function using @function\_name, it assigns function handle, a value, to the called back function. What initTask does is to write this function handle to task (struct array) – one of the output variables of initTask. The “task” array will then be used by updateTask.

An example:

% set function handles

if exist('startSegmentCallback','var') && ~isempty(startSegmentCallback)

task.callback.startSegment = startSegmentCallback;

end

The call-back functions will later be evaluated in updateTask.

* Below are the call-back functions. They are subfunctions in the main task-running function file (e.g. taskTemplate, testExperiment, afcom):

startSegmentCallback: does nothing.

screenUpdateCallback: does all the drawing and get response

getResponseCallback: if task.thistrial.gotResponse==0 (didn't get a response?), then task = jumpSegment(task) (jump to the next segment).

startTrialCallback: set mouse position, get mouse position, cur dots direction, distractors, targetAngle, distractorAngle ...

endTrialCallback: if task.thistrial.dead, return, end.

**updateTask**

[task{1}, myscreen, phaseNum] = updateTask(task{1},myscreen,phaseNum);

What updateTask does is to use the function handle defined in initTask to call the function.

An example:

% call segment start callback

[task{tnum} myscreen] = feval(task{tnum}.callback.startSegment,task{tnum},myscreen);

> set random state: what does it mean? What is it for?

it's like you draw everything (in the back) and when you flip it you show what's drawn.

This makes sense. But you still need to detect response. It must be somewhere in updateTask.

> **mglClose** would close the screen (with about 15 sec delay)

> **mglClearScreen** Sets the background color and clears the buffer

> **mglGetParam**

> **mglStencilCreateBegin**: begin drawing to stencil.

e.g.

mglOpen; %open an mgl window

mglScreenCoordinates;

%Draw an oval stencil

mglStencilCreateBegin(1);

mglFillOval(300,400,[100 100]);

mglStencilCreateEnd;

mglClearScreen;

% now draw some dots, masked by the oval stencil

mglStencilSelect(1);

mglPoints2(rand(1,5000)\*500,rand(1,5000)\*500);

mglFlush; %swap front and back buffer (waits for one frame tick)

mglStencilSelect(0);

Drawing: first open a screen. begin drawing to a stencil (mglStencilCreateBegin)

draw an oval stencil. draw some dots masked by the oval stencil (use mglStencilSelect).

swap butter (mglFlush).