Initiation to Matlab 3 : introduction to the Psychtoolbox

The Psychtoolbox is a set of functions and Matlab executable (.mex file) written in C/C++ dedicated to conducting experiments in psychology. This toolbox is not developed by The Mathworks (Matlab's editor) but by researchers in psychology. It is available at the following URL

http://psychtoolbox.org/PsychtoolboxDownload.

It provides an easy interface to your computer graphic layer and allows to precisely control temporal aspects of the display update. Similarly, it facilitates the accurate collection of responses of the subject through different hardware (keyboard, mouse, eye tracker, ...).

Many demonstration scripts are provided with the toolbox. These scripts are available in the directory 'PsychDemos' located in the installation directory of the Psychtoolbox. To access the directory from Matlab command line:

```
>> cd([PsychtoolboxRoot,'/PsychDemos'])
```

These demo scripts are a good basis for the establishment of customized experiments. They show the implementation of the key functionality of the Psychtoolbox.

The psychtoolbox does not include in the Matlab environment a documentation as easily accessible as the one provided for basic Matlab functions. If an internet connection is available to the user, it is advisable to use the online help at the following address:

→ http://docs.psychtoolbox.org/Psychtoolbox

However, if no internet connection is available, it is possible to get help for the functions of the Psychtoolbox from the command line (eg. `>> help Screen` for a general description of the function `Screen`, `>> Screen('Flip?')` for detailed help of the option 'Flip').

1 The Screen function

The function `Screen` is at the basis of the Psychtoolbox. This is the function that allows the management of the display of stimuli. In this introduction, we will review some options for this function.

The syntax for the call to this function is always of the form:

```
Screen('function', pointer_to_the_window, mandatory and/or optional arguments)
```

1.1 Opening / Closing a window

Run the following script `first_screen.m` detailed below.

```matlab
%%%% first_screen.m
clear all
close all

%Initialise some variables that will be used to call 'OpenWindow'
whichScreen = 0; %allow to choose the display if there's more than one
wPtr = Screen('OpenWindow',whichScreen);
%Open a window and
%returns a pointer to the window
```
white = 255; %pixel value for white
black = 0; %pixel value for black
gray = (white+black) / 2; %pixel value for middle gray

Screen('FillRect', wPtr, gray); %fill the buffer image in the
%graphic card memory with grey

Screen('Flip', wPtr); %displays the content of the buffer
%in the window

WaitSecs(1); %pause for 1 sec

Screen('FillRect', wPtr, white);
Screen('Flip', wPtr);

WaitSecs(1);

Screen('FillRect', wPtr, black);
Screen('Flip', wPtr);

WaitSecs(1);

Screen('CloseAll'); %closes the window

At the beginning of the script, we called the 'OpenWindow' function, this function allows the Psychtoolbox
to take control of the display. For the operating system to get is hand back on the display, we have to call the
function Screen ('CloseAll'), as in the end of the example script.

1.2 Regain control when the computer hangs with an opened window

There is good chance that you will sometimes loose the control over the keyboard while you test your
program. This can happen if your program stops (probably due to an error in your program) before closing the
window with Screen ('CloseAll'). It can also happen if your script contains an infinite loop.

In case that happens to you, to regain control, first use the key combination Alt+Tab to bring the Matlab
command window to the foreground. This should allow the use of the keyboard on the command window. Then
press simulatenously Ctrl+C several times. This will stop the execution of your program if it has not already
stopped due to an error. Finally, type the statement sca and hit Return to evaluate it. sca is an alias to the
function Screen ('CloseAll'). The Psychtoolbox window should then close.

Test yourself with the following script.

To avoid to loose control each time there is an error in our script during the writing phase of our program
with the Psychtoolbox, the statements try and catch can be used as follows.
a = b % this statement yields an error
   Indeed, b is undefined
Screen('CloseAll'); % close the window

catch ERR
   % if an error occurs between the try / catch statements
   % the error informations are stored in the structure ERR
   % (ERR is an arbitrary variable name), and the following code
   % is then executed
   Screen('CloseAll');
   rethrow( ERR ); % now that the window is closed,
   % we propagate the error so that
   % the command window is closed,
   % the message error is nicely displayed
   % at the prompt
end

If an error occurs between the try and catch statements rather than to immediately stop the script, Matlab will execute the instructions between the following catch and end.

We can thus make sure the Screen (‘CloseAll’) statement gets evaluated when an error occurs and gain control on the keyboard and the command window. The command rethrow(ERR) will propagate the error that occurred between the try and catch, so we will find the trace of the alleged error on the command window, facilitating the debugging.

1.3 Visual stimuli with simple shapes
Screen('DrawLine', windowPtr [,color], fromH, fromV, toH, toV [,penWidth]);
Screen('DrawArc',windowPtr,[color],[rect],startAngle,arcAngle)
Screen('FrameArc',windowPtr,[color],[rect],startAngle,arcAngle [,penWidth] [,penHeight] [,penMode])
Screen('FillArc',windowPtr,[color],[rect],startAngle,arcAngle)
Screen('FillRect', windowPtr [,color] [,rect] );
Screen('FrameRect', windowPtr [,color] [,rect] [,penWidth] );
Screen('FillOval', windowPtr [,color] [,rect] [,perfectUpToMaxDiameter]);
Screen('FrameOval', windowPtr [,color] [,rect] [,penWidth] [,penHeight] [,penMode]);
Screen('FramePoly', windowPtr [,color] [,pointList] [,penWidth]);
Screen('FillPoly', windowPtr [,color], pointList [, isConvex]);

Here is the list of options on the Screen function for displaying simple shapes. As a reminder, you can access documentation regarding any of these options for the command line (eg : >>Screen (‘DrawLine?’)).

The basic unit on the screen is the pixel. A point on the screen is defined by its coordinates (x, y) pixels on a 2 dimensions frame. The origin (0,0) of this reference frame is in the upper-left of the screen, the horizontal axis X is directed to the right and the vertical Y axis downwards (see Fig. Fig. 1-(a))

![Figure 1](image)

**Figure 1** – (a) screen coordinate system. (b) a point (two coordinates - horizontal, vertical). (c) a line (two points : start, end). (d) a rectangle (two points : upper left corner lower right corner)

The **rect** parameter, useful for most shapes, defines the rectangle on the screen where you want to place the shape. It is a row vector of four elements. The first and second elements are the coordinates of the upper left
corner. The 3rd and 4th the coordinates of the lower right corner. For example, the rectangle of Figure 1-(d)
would be \texttt{rect}_1d = [2,3,7,7].

The \texttt{color} parameter specifies the desired color of the shape. It is defined by a row vector of 3 components
RGB (Red Green Blue) with R, G, B coded from 0 to 255 (eg \texttt{blue} = [0,0,255]). It’s easy to find correspondences
between color and RGB components on the internet (search google for ‘RGB color picker’). There is a special
case to set the colors in grayscale. In this case, simply set the color as an integer between 0 (black) to 255
(white).

\textbf{Preparation of a complex stimulus and screen updating} : The call to the above functions to draw simple
shape does not update the screen. It will only change a buffer image inside the memory of the graphics card.
We therefore can combine several shapes before displaying the result on the screen. To update the screen with
the image present in the memory of the graphics card, we call the \texttt{Flip} function.

**Exercice :** Create a script that opens a window and display the following shapes on a white background :
— a full yellow rectangle with its upper left corner at (30,50) and lower right corner at (130,150)
— a green circle in the middle of the screen with a diameter of 100 pixels. The size in pixels of the
screen is obtained from the second return value of the function \texttt{Screen(’OpenWindow’,...)} (see
the function help >>\texttt{Screen(’OpenWindow?’)})
— a line connecting the center of the rectangle to the center of the circle
Wait for two seconds before closing the window.
Modify the script so as to display the same primitive but this time on a black background.

\subsection*{1.4 Showing images (MakeTexture and DrawTexture)}

The \texttt{Screen(’MakeTexture’, ...,)} function converts a 2D image matrix into an OpenGL texture and
returns a pointer to that texture. This pointer can then be used to update the graphic card image buffer using
the \texttt{Screen(’DrawTexture’,...,)} function.

\begin{verbatim}
texturePointer=Screen(’MakeTexture’, windowPointer, imageMatrix, ...)
Screen(’DrawTexture’, windowPointer, texturePointer [,sourceRect] [,destinationRect],...)
\end{verbatim}

where \texttt{sourceRect} is an optional argument, a rect defining a subpart of picture to present, if not specified or
kept empty ([ ] ) it defaults to using the whole picture. \texttt{destinationRect} is a rect that can be used to define a
subpart of the screen to present the image in (defaults to center of screen without scaling), if the ratio between
the source and the destination rects is not the same, the image will be distorted to fit the destination rect.

The \texttt{imread} function that comes with Matlab creates an image matrix from an image file (eg, BMP, JPEG).

The sample script below demonstrates the use of 4 textures to display images. We use here images that are
shipped with the Psychtoolbox distribution in the subfolder PsychHardware/EyelinkToolbox/EyelinkDemos/SR-
ResearchDemo/AntiSaccade.

\begin{verbatim}
% ex_texture.m
clear all
close all
basedir = fullfile(PsychtoolboxRoot,'PsychHardware','EyelinkToolbox','EyelinkDemos','SR-ResearchDemo','AntiSaccade');
imMatrix_V1 = imread(fullfile(basedir, ‘img51.jpg’ ));
imMatrix_V2 = imread(fullfile(basedir, ‘img52.jpg’ ));
imMatrix_R1 = imread(fullfile(basedir, ‘img101.jpg’ ));
imMatrix_R2 = imread(fullfile(basedir, ‘img102.jpg’ ));

% initialise some variables that will be used to call ’OpenWindow’
whichScreen = 0; % allow to choose the display if there’s more than one

try
    wPtr = Screen(’OpenWindow’, whichScreen);
end
\end{verbatim}
white = 255; %pixel value for white
black = 0; %pixel value for black
gray = (white + black) / 2; %pixel value for middle gray

%create an openGL texture for each image
imTexture_V1 = Screen('MakeTexture', wPtr, imMatrix_V1);
imTexture_V2 = Screen('MakeTexture', wPtr, imMatrix_V2);
imTexture_R1 = Screen('MakeTexture', wPtr, imMatrix_R1);
imTexture_R2 = Screen('MakeTexture', wPtr, imMatrix_R2);

Screen('DrawTexture', wPtr, imTexture_V1); % Fill the buffer with the first texture
Screen('Flip', wPtr); % Update the display with the buffer content
WaitSecs(1);
Screen('DrawTexture', wPtr, imTexture_V2);
Screen('Flip', wPtr);
WaitSecs(1);
Screen('DrawTexture', wPtr, imTexture_R1);
Screen('Flip', wPtr);
WaitSecs(1);
Screen('DrawTexture', wPtr, imTexture_R2);
Screen('Flip', wPtr);
WaitSecs(1);
Screen('CloseAll'); % Close the window

catch ERR
    Screen('CloseAll');
    rethrow(ERR);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Exercice : Complete the following script in order to display the 3 images (konijntjes1024x768.jpg, konijntjes1024x768blur.jpg, konijntjes1024x768gray.jpg) on the same screen as in the figure Fig. 2.

The size in pixels of the screen is obtained from the second return value of the function Screen('OpenWindow',...) (see the function help >>Screen('OpenWindow?')), also the size of an image can be obtained using the function size() on an image matrix. For example, if IM is an image matrix, the horizontal size is given by size(IM,2), and the vertical is given by size(IM,1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all
close all
basedir = fullfile(PsychtoolboxRoot, 'PsychDemos');
imMatrix_sharp = imread(fullfile(basedir, 'konijntjes1024x768.jpg'));
imMatrix_blur = imread(fullfile(basedir, 'konijntjes1024x768blur.jpg'));
imMatrix_gray = imread(fullfile(basedir, 'konijntjes1024x768gray.jpg'));

% Initialise some variables that will be used to call 'OpenWindow'
whichScreen = 0; % Allow to choose the display if there's more than one

try
    [wPtr, wRect] = Screen('OpenWindow', whichScreen);
    % Open a window and
    % Returns a pointer to the window
white = 255; % Pixel value for white
black = 0; % Pixel value for black
gray = (white + black) / 2; % Pixel value for middle gray

% Create an openGL texture for each image
imTexture_sharp = Screen('MakeTexture', wPtr, imMatrix_sharp);
imTexture_blur = Screen('MakeTexture', wPtr, imMatrix_blur);
imTexture_gray = Screen('MakeTexture', wPtr, imMatrix_gray);

XXX ADD YOUR CODE HERE XXX
% wait for a key stroke
while ~KbCheck
    Screen('CloseAll');
end

Screen('CloseAll');
catch ERR
    Screen('CloseAll');
    rethrow(ERR);
end

Figure 2 – Placement souhaitée des 3 images.

1.5 Precise management of visual stimuli onset

As you can see from the above examples, the actual updating of the display is done by calling the 'Flip' function. This function can be called with different options and can return important information about time management.

\[ \text{[VBLTimestamp StimulusOnsetTime]} = \text{Screen}('\text{Flip}', \text{windowPtr [, when]}) \]

The optional input argument 'when' allows to specify the timestamp at which we want to update the display. The call to the 'Flip' function will block the execution of the program until the effective update of the display. The return value 'StimulusOnsetTime' provides an accurate estimate of the time at which the update actually took place on the screen.

However, one must be aware of the limitations of the screen in terms of refresh rate. A screen with a refresh rate of 100Hz displays an image every 10ms. For a screen at 60Hz, it falls to one frame every 16.67ms.
white = 255; %pixel value for white
black = 0; %pixel value for black
gray = (white+black) / 2; %pixel value for middle gray

% create a openGL texture for each image
imTexture_V1 = Screen('MakeTexture', wPtr, imMatrix_V1);
imTexture_V2 = Screen('MakeTexture', wPtr, imMatrix_V2);
imTexture_R1 = Screen('MakeTexture', wPtr, imMatrix_R1);
imTexture_R2 = Screen('MakeTexture', wPtr, imMatrix_R2);

Screen('FillRect', wPtr, gray); % Fill the buffer in grey
[VBL, stimOnsetTime] = Screen('Flip', wPtr); % Immediately update the display and store the timestamp of the effective update in stimOnsetTime

Screen('DrawTexture', wPtr, imTexture_V1); % Fill the buffer with the first texture
[VBL, stimOnsetTime] = Screen('Flip', wPtr, stimOnsetTime+duree_stim); % Update the display 500 ms after the last 'Flip'

Screen('DrawTexture', wPtr, imTexture_V2);
[VBL, stimOnsetTime] = Screen('Flip', wPtr, stimOnsetTime+duree_stim);

Screen('DrawTexture', wPtr, imTexture_R1);
[VBL, stimOnsetTime] = Screen('Flip', wPtr, stimOnsetTime+duree_stim);

Screen('DrawTexture', wPtr, imTexture_R2);
Screen('Flip', wPtr, stimOnsetTime+duree_stim);

WaitSecs(0.5);
Screen('CloseAll'); % close the windows

try
    Screen('CloseAll');
    rethrow(ERR);
end

2 Interacting with the subject

In most psychology experiments, the subject’s responses are obtained by using the keyboard or mouse. In addition to the key pressed or the mouse position, the response time (time between the onset of the stimulus and the subject’s response) is often a variable that researchers in psychology wish to measure accurately. Functions to access the keyboard and mouse stated offered by Psychtoolbox allow a temporal precision in the order of the millisecond.

2.1 Handle keyboard events

Keyboard response collection is performed through the function KbCheck. This function is non-blocking (it does not block program execution waiting for an event in the keyboard). It returns ”instantaneously” the state of the keyboard keys as follows:

[keyIsDown, secs, keyCode] = KbCheck

The return value 'KeyIsDown’ is a boolean variable that is set to 1 when any key is pressed. ‘sec’ returns the computer time of the keyboard status check. And ’keyCode’ is an array of 256 Boolean. Each boolean represents the state of a key, a 1 means that the key is pressed and 0 the key is released.

To find out which element of boolean array 'keyCode' corresponds to a button, you can run the following statement in the Command window and press the desired key while it runs:

>> for i = 1:10, WaitSecs(0.5); [keyIsDown, secs, keyCode] = KbCheck; find(keyCode), end

To watch and wait for an event on the mouse or keyboard, you must set up a scanning loop. For example, to wait for any key stroke:

>> for i = 1:10, WaitSecs(0.5); [keyIsDown, secs, keyCode] = KbCheck; if keyCode(i), end
while ~KbCheck
end

To wait for a specific key stroke, here the left Ctrl key:

[keyIsDown, secs, keyCode] = KbCheck;
while ~keyCode( 38 ) %element 38 of keyCode vector is the CtrlLeft key on my system but may vary from one system to another
[keyIsDown, secs, keyCode] = KbCheck;
end

2.2 Handle mouse events

The mouse position and the status of its buttons can be obtained using the 'GetMouse' function:

[x, y, buttons] = GetMouse

The return values 'x' and 'y' are the position of the cursor in screen coordinates (pixels). 'buttons' is an array of N Boolean (0/1 values) where N is the number of mouse buttons. Each array element is a mouse button. A boolean to (1) means that the button is pressed and is (0) otherwise.

The following script provides an example of using KbCheck and GetMouse functions.

```
try
    whichScreen = 0;
    wPtr = Screen('OpenWindow', whichScreen);

    white = 255; %pixel value for white
    black = 0; %pixel value for black
    gray = (white+black) / 2; %pixel value for middle gray
    nb_trial = 5;

    for i = 1:nb_trial
        Screen('FillRect', wPtr , gray);
        DrawFormattedText(wPtr, ...
                        'Measure of the time difference between a key stroke and a mouse click
                        Start by hitting a key' ...
                        ,[...]
                        ,'center', 'center', black);
        Screen('Flip', wPtr);

        %wait for a key stroke
        while ~KbCheck
            %we store the timestamp of the key stroke event
            tkeyboard = GetSecs;
        end

        Screen('FillRect', wPtr , white);
        DrawFormattedText(wPtr, 'Waiting for a mouse click' ,'center', 'center', black);
        Screen('Flip', wPtr);

        %wait for a mouse click
        [x, y, buttons] = GetMouse;
        while ~sum(buttons)
            [x, y, buttons] = GetMouse;
        end

        tmouse = GetSecs; %we store the timestamp of the mouse event

        Screen('FillRect', wPtr , black);
        DrawFormattedText(wPtr, 'Please release the keyboard and the mouse' ,[...], 'center', 'center', white);
        Screen('Flip', wPtr);

        %wait for the subject to release the mouse and the keyboard
        [x, y, buttons] = GetMouse;
        while KbCheck | sum(buttons)
            [x, y, buttons] = GetMouse;
        end

        deltaT(i) = tmouse-tkeyboard;
```
Exercice : Program a task to evaluate the Muller-Lyer Illusion effect.

The Muller-Lyer illusion is an optical illusion. One of its manifestation is for a line segment to appear longer or smaller depending if it is surrounded with inward or outwards arrows (see Fig 3).

One can imagine the following task to evaluate the Muller-Lyer Illusion effect.
- Display one horizontal line of fixed known length in pixels (for example 300 px) with arrows pointing outwards at both ends on the center of the screen.
- Display a second horizontal line of random length (but close to the first line, for example from 260 to 340 px) under the first one with inwards arrows
- Using 2 keyboard keys of your choice to record subject responses, ask the subject to say whether he sees the bottom line as longer or smaller in comparison to the top line.

Implement the program for this task for 50 trials by varying randomly the size of the second arrow from -40 to +40 pixels relative to the first arrow by step of 10 pixels. We want to keep record of the value of the size difference and the response of the subject for each trial.

**Figure 3** – Illustration of the Muller-Lyer illusion. The line in the bottom arrow seems to be longer than the one from the top arrow.
3 Animations

The illusion of motion is obtained by changing rapidly the position of what is displayed on the screen. Typically, a screen is refreshed at 60 Hz (ie 60 times per second, some hardware can perform at higher rate), thus we can display a new image or change stimuli positions every 16.67 ms.

The higher the distance between positions of the stimuli in 2 successive frames, the faster the motion will appear.

```matlab
% animation1.m
clear all
close all

try

whichScreen = 0;
[wPtr, wRect] = Screen('OpenWindow', whichScreen);

white = 255; % pixel value for white
black = 0; % pixel value for black
gray = (white+black) / 2; % pixel value for middle gray
red = [255,0,0];

Vx = 5; % velocity on X axis in pixel / frame
Px = wRect(3)/2; % initial position of the circle on X axis
Py = wRect(4)/2; % initial position of the circle on Y axis
R = 50; % radius of the circle

while ~KbCheck

% compute new position
Px = Px + Vx;

% if the circle goes outside the right side of the screen
% we change its position in order to make it appear on
% the left side
if Px-R > wRect(3)
    Px = -R;
end

% draw circle in new position
Screen('FillOval', wPtr, red, [Px-R, Py-R, Px+R, Py+R]);
Screen('Flip', wPtr);
end

Screen('CloseAll');

catch ERR

Screen('CloseAll');
rethrow(ERR);
end
```

Exercises:
- Observe the effect of the velocity Vx by increasing and decreasing its value. Change the velocity Vx to a negative value. What happens? Fix it.
- Modify the code to add a vertical component to the motion.
- Modify the code to control the motion of the circle using the keyboard arrows.
- Create a script so that the circle follows the mouse.
- Create a script that displays and updates an simplified analog clock (see Fig. 4) using the following lines of code to get the current time:
  ```matlab
time = now;
hour = str2num( datestr( time , 'HH' ) );
minute = str2num( datestr( time , 'MM' ) );
second = str2num( datestr( time , 'SS' ) );
```
4 Programmation of a clone of the game Snake/Nibbles

You most certainly had already the occasion to play to a variation of the game Snake. The game consists in guiding a snake in order to collect food items. The snake moves continuously at constant speed, the player can only change the direction the snake is following. With each food item collected, the snake grows a bit. The game is over when the snake hits the edge of the screen or when he hits an obstacle (a wall or himself by biting his tail). We’re going to program a simplified version of this cult game without obstacle and with very limited graphics (see Figure 5).

4.1 Game modelisation

In order to simplify the screen space, we’re going to subdivide the screen into a matrix of blocks describing the possible positions of the items of the game (items to collect, snake’s head and its tail). The number of blocks on the horizontal and vertical axis are parameters that we’re going to set in our code (for ex $N_x = 100$ et $N_y = 75$). As you can see on Figure 6, knowing the number of blocks and the screen size, we can compute the rectangle of a block located at column $i$ and row $j$ with the following formula:

$$rectBloc = [(i-1) \times \text{scrSizeX}/N_x, i \times \text{scrSizeX}/N_x, (j-1) \times \text{scrSizeY}/N_y, j \times \text{scrSizeY}/N_y];$$
Exercise : Write a script that opens a window making sure to store its size in a variable, and that on a matrix of blocks of size \(N_x=100, N_y=75\) fills the block \((i=2,j=2)\) in black, the block \((i=50,j=37)\) in red et \((i=99,j=74)\) in blue.

### 4.1.1 Direction of the snake

The snake can follow 4 directions (up, right, down, left). We’re going to use a variable named `direction` who can take the value 1,2,3 or 4 to represent the direction in the order previously stated.

At the beginning of the game, we draw the initial direction randomly

```matlab
direction = randi(4);
```

The player can change the direction during the game using the arrow keys on the keyboard. Here is an example of how to scan the keyboard for 0.5 s.

```matlab
tStart = GetSecs;
while GetSecs - tStart < 0.5
    [keyIsDown, secs, keyCode] = KbCheck;
    if keyCode(UpArrowCode)
        direction = 1;
    elseif keyCode(RightArrowCode)
        direction = 2;
    elseif keyCode(DownArrowCode)
        direction = 3;
    elseif keyCode(LeftArrowCode)
        direction = 4;
    end
end
```

To use this code, you need to initialise the variables `*ArrowCode` to the codes corresponding to the arrow keys of the keyboard.
4.1.2 Target

Positions of the objects and the snake can now simply be described as the positions \( i, j \) of the corresponding blocks. We define a variable \( \text{target} \) row vector of 2 elements to represent the position of the target. The position of the target will be initialized randomly at the beginning and will also be reset randomly to a new position after the snake eats it.

\[
\text{target} = [ \text{randi}(\text{Nx}) , \text{randi}(\text{Ny}) ];
\]

4.1.3 Snake

The snake has a variable length and can spread on several blocks. The snake positions will be represented by a matrix \( \text{snake} \) of size (\( Ns \times 2 \)) where \( Ns \) is the snake’s length. Each line of \( \text{snake} \) contains the coordinates \( i, j \) of a block constituting the snake. At the game start \( Ns=1 \), and will increase with each collected food item. We will make sure to keep the snake’s head at the first line of the variable \( \text{snake} \).

The snake will be placed at the center of the screen at the game start.

\[
\text{snake} = [ \text{round}(\text{Nx}/2) , \text{round}(\text{Ny}/2) ];
\]

In order to move the snake, we first need to grow the \( \text{snake} \) matrix by adding the new snake’s head position as the first line of the matrix. To do that, we use matrix concatenation. For example, to place the head at the left of its previous position :

\[
\text{snake} = [ [ \text{snake}(1,1)-1 , \text{snake}(1,2) ] ; \text{snake} ]; \% \text{deplacement de la t\^ete vers la gauche}
\]

At this stage, the snake has grown by 1 block, if the snake has just eaten an item we’re good. Otherwise, we need to pull out the last block to simulate the drag of the tail. We can do this by assigning the empty matrix [] to the last line of variable \( \text{snake} \) as follow :

\[
\text{snake}(\text{end},:) = [] ;
\]

4.2 Game loop

Contrary to a simple program or script that processes some data and then stops, a video game can run indefinitely. Also contrary to a program with a graphical user interface who waits for an action from the user to perform the requested task, the game environment often evolves continuously even if the player do not interact. For example, in our game the snake will move continuously without having to wait for the gamers input.

We are therefore going to use a while loop from which we exit only if the player wants to quit the game (by hitting the ESCAPE key for example) or if the game is lost (the snake goes outside of the screen or has bitten its tail). In this loop, at each iteration, we are going to check if the player wants to change the direction, compute the new position of the items in the environment and update the screen. In our game, the apparent motion of the snake is performed by periodically shifting the snake position from one block. As we have seen before in the part concerning animations, the speed of the snake depends on the time interval between two consecutive positions, we are going to specify this interval in the variable \( \text{updateInterval} \) and use it to limit the iteration rate of the game loop. Also it will be used to set the difficulty of the game.

<table>
<thead>
<tr>
<th>Game program description</th>
</tr>
</thead>
<tbody>
<tr>
<td>— Initialize game parameters (update interval for ex ( \text{updateInterval}=0.1 ), block matrix size ( \text{Nx}, \text{Ny} ))</td>
</tr>
<tr>
<td>— Initialize the game state variables (target position, snake, score)</td>
</tr>
<tr>
<td>— Show a welcome screen and wait for key press from the player to notify that he’s ready</td>
</tr>
<tr>
<td>— While that game is not finished (game loop)</td>
</tr>
<tr>
<td>— Draw game objects (target and snake) at their current position</td>
</tr>
<tr>
<td>— Scan the keyboard state for a duration of ( \text{updateInterval} ) seconds, update the ( \text{direction} ) variable if an arrow is pressed</td>
</tr>
<tr>
<td>— Update the ( \text{snake} ) variable depending on the direction, i.e. grow the matrix to add a new first line with the new position of the head</td>
</tr>
<tr>
<td>— Check if the target has been eaten by the snake</td>
</tr>
<tr>
<td>— If yes, compute a new position for the target and increment the score</td>
</tr>
</tbody>
</table>
— If no, we shorten the snake’s tail by deleting the last line of the matrix `snake`
— Check that the snake does not collide with an obstacle at his new position. If it’s the case, the game is lost
— Show a game over screen

Now you just have to program all of that.

5 Going further

The psychtoolbox also includes some abstractions to the portaudio library that can play sound stimuli with precise scheduled timing (if the hardware and the driver allows it) and movies. We’re not going to detail these two aspects here but if you’re interested you can have a look at the demos that are shipped with the Psychtoolbox.

For the audio aspect, the portaudio backend is accessed through the PsychPortAudio() function and is demonstrated in the following scripts:

— BasicSoundOutputDemo - Demonstrate basic usage of PsychPortAudio() for sound playback.
— BasicSoundInputDemo - Demonstrate basic usage of PsychPortAudio() for sound capture.
— SimpleSoundScheduleDemo - Simple demo for basic use of sound schedules with PsychPortAudio.
— SimpleVoiceTriggerDemo - Demo of a simple voice trigger with PsychPortAudio.
— BasicSoundFeedbackDemo - Demonstrates a audio feedback loop via PsychPortAudio(). See DelayedSoundFeedbackDemo for a more research grade approach.
— BasicSoundScheduleDemo - Demonstrate basic usage of sound schedules and buffers with PsychPortAudio.
— DelayedSoundFeedbackDemo - Demonstrates a audio feedback loop via PsychPortAudio() with exactly controlled latency.

For movie playback, the gstreamer backend is accessed through the Screen function and is demonstrated in the following scripts:

— SimpleMovieDemo - Most simplistic demo on how to play a movie.
— PlayMoviesDemo - Show simple playback of one movie with sound at a time.
— PlayDualMoviesDemo - Same as PlayMoviesDemo, but play two movies in parallel.
— PlayMoviesWithoutGapDemo2 - Play one movie while opening another one to reduce gaps between movies.
— DetectionRTInVideoDemo - How to collect reaction times in response to detection of some event in a presented movie file. Takes care to get timing right.
— LoadMovieIntoTexturesDemo - Quickly load a movie into a stack of textures for quick playback with arbitrary speed and order.

To view the source code, enter `>>edit DemoName` at the prompt.