Expyriment graphics and timing

Programming Psychology Experiments (CORE-1)

Barbu Revencu & Maxime Cauté

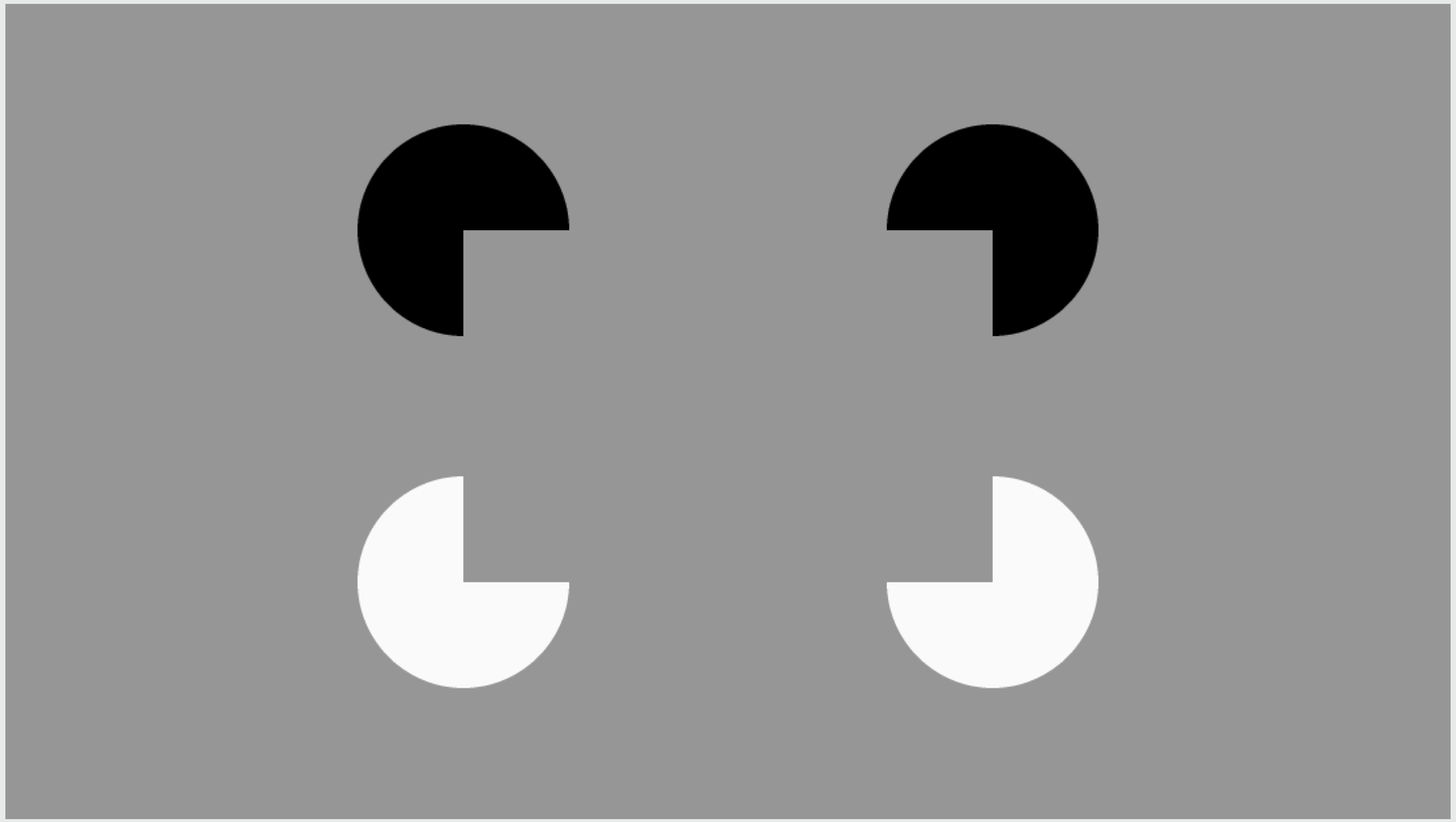
Session 4 | 1 October 2025

The plan for today

1. **Assignment** discussion
2. **Expyriment graphics and timing**: Present stimuli on-screen *when* and *for how long* you want them

Assignment 3 Discussion

Kanizsa display



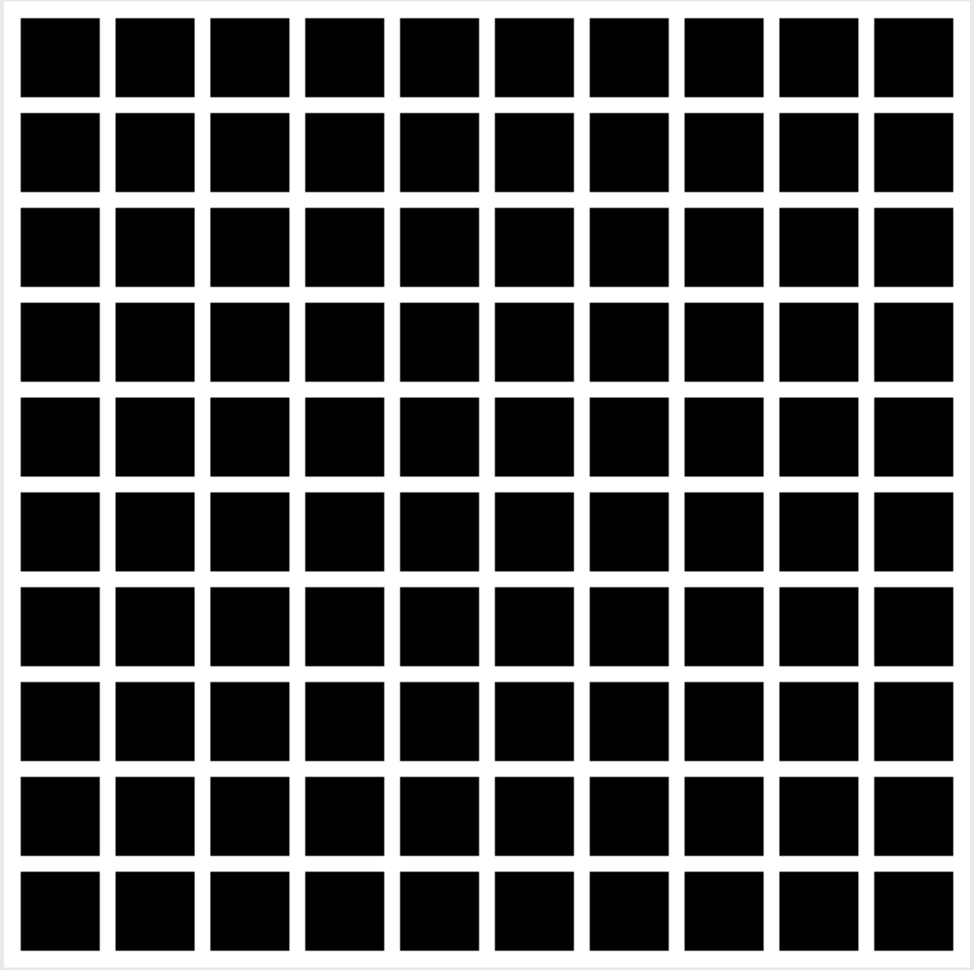
Submitted solutions: Excerpts

```
square1 = stimuli.Rectangle((rectangle_width*2, rectangle_height*2), position=(0, 0))

pos1 = (rectangle_width, rectangle_height)
pos2 = (-rectangle_width, rectangle_height)
pos3 = (-rectangle_width, -rectangle_height)
pos4 = (rectangle_width, -rectangle_height)

circle1 = stimuli.Circle(circle_size, colour="black", position=pos1)
circle2 = stimuli.Circle(circle_size, colour="black", position=pos2)
circle3 = stimuli.Circle(circle_size, colour="black", position=pos3)
circle4 = stimuli.Circle(circle_size, colour="black", position=pos4)

circle1.present(clear=True, update=False)
circle2.present(clear=False, update=False)
circle3.present(clear=False, update=False)
circle4.present(clear=False, update=False)
square1.present(clear=False, update=True)
```

Submitted solutions: Excerpts

```
rectangle = stimuli.Rectangle((w, h), position=(0, 0))

half_w, half_h = w // 2, h //2
for x in (-half_w, half_w):
    for y in (-half_h, half_h):
        positions.append((x, y))
        colors.append(C_WHITE if y < 0 else C_BLACK)

circles = [stimuli.Circle(radius, position=pos, colour=color)
            for pos, color in zip(positions, colors)]

for i, circle in enumerate(circles):
    circle.present(clear=(i == 0), update=False)
rectangle.present(clear=True, update=True)
```

Hermann grid



Submitted solutions: Outline

```
""" HELPER FUNCTIONS “"""
def create_grid, run_trial...

”"” GLOBAL SETTINGS """
exp = design.Experiment(name="Kanizsa rectangle")
control.initialize(exp)
n_rows, n_cols = 10, 12
square_side_length, space_between_squares = 50, 10
square_color, background_color = C_BLACK, C_WHITE
max_width, max_height = exp.screen.size

""" CREATE STIMULI """
squares = create_grid(n_rows, n_cols, square_side_length,
    space_between_squares, square_color, max_width, max_height)

""" RUN EXPERIMENT """
run_trial(squares, background_color)
```

Submitted solutions: `create-grid`

```
def create_grid(..., max_width, max_height):
    # Compute necessary width and height
    W = n_cols * square_side_length + (n_cols - 1) * space_between_squares
    H = n_rows * square_side_length + (n_rows - 1) * space_between_squares

    # Raise an error if target rows and columns don't fit on the screen
    if W > max_width or H > max_height:
        raise ValueError("The target grid does not fit on the screen")

    ...
```

Submitted solutions: `create-grid`

```
def create_grid(size, color, space, ...):
    for row in range(n_rows):
        for col in range(n_cols):
            pos_x = col * (size + space) - (W - size) // 2
            pos_y = row * (size + space) - (H - size) // 2
            squares.append(stimuli.Rectangle(
                size=(size, size), position=(pos_x, pos_y), colour=color)
            )
```

Submitted solutions: `create-grid` alt

```
def create_grid(size, color, space):
    grid_surface = stimuli.Rectangle(size=(W, H), colour=color))

    for row in range(n_rows):
        y = row * (H / n_rows) - (H / 2)
        stims.append(stimuli.Line(
            start_point=(W / 2 + 1, y) end_point=(- W / 2 - 1, y),
            colour=background_colour, line_width=space)

    for col in range(n_cols):
        x = col * (W / n_cols) - (W / 2)
        stims.append(stimuli.Line(
            start_point=(x, H / 2 + 1), end_point=(x, -H / 2 - 1),
            colour=background_colour, line_width=space)
```

Expyriment graphics

Rendering

When you define a stimulus (e.g., `Circle`, `TextLine`), it doesn't yet have a pixel-level (visual) representation

Instead, it starts off as a set of parameters (points, color, size) that provides **a recipe for how it should be drawn** at presentation time

To present the stimulus on-screen, expyriment has to **turn this recipe into a pixel image**

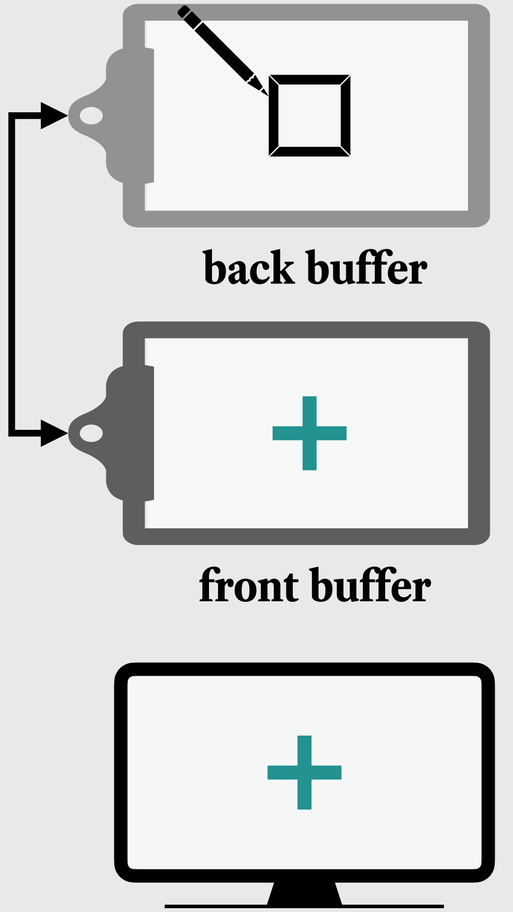
The double buffer

Graphical processing unit (GPU) renders visual stimuli on an invisible sketchpad: the **back buffer**

Monitor is linked to a second sketchpad: the **front buffer**

The monitor **reads out** the contents of the front buffer and **displays** them (every 16.67 ms, if refresh rate = 60 Hz)

The **back and front buffer can be swapped** by a switch



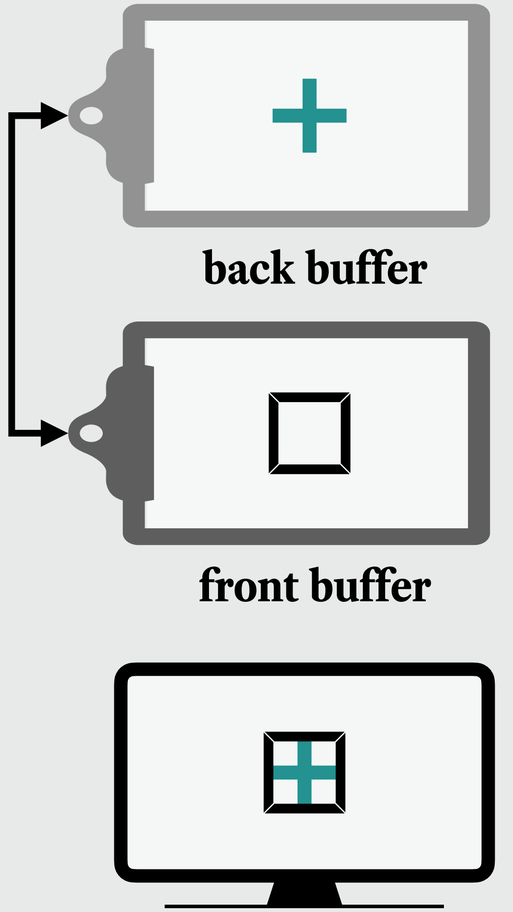
The double buffer

Graphical processing unit (GPU) renders visual stimuli on an invisible sketchpad: the **back buffer**

Monitor is linked to a second sketchpad: the **front buffer**

The monitor **reads out** the contents of the front buffer and **displays** them (every 16.67 ms, if refresh rate = 60 Hz)

The **back and front buffer can be swapped** by a switch



Revisiting `stimulus.present()`

`present()` takes two Boolean parameters: `clear` and `update`

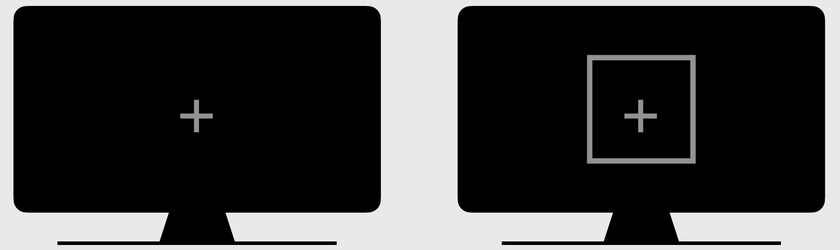
- `clear`: Should I erase what's currently on the back buffer?
- `update`: Do I swap the back and the front buffer?

By default, **both parameters are set to `True`**

What does this code do?

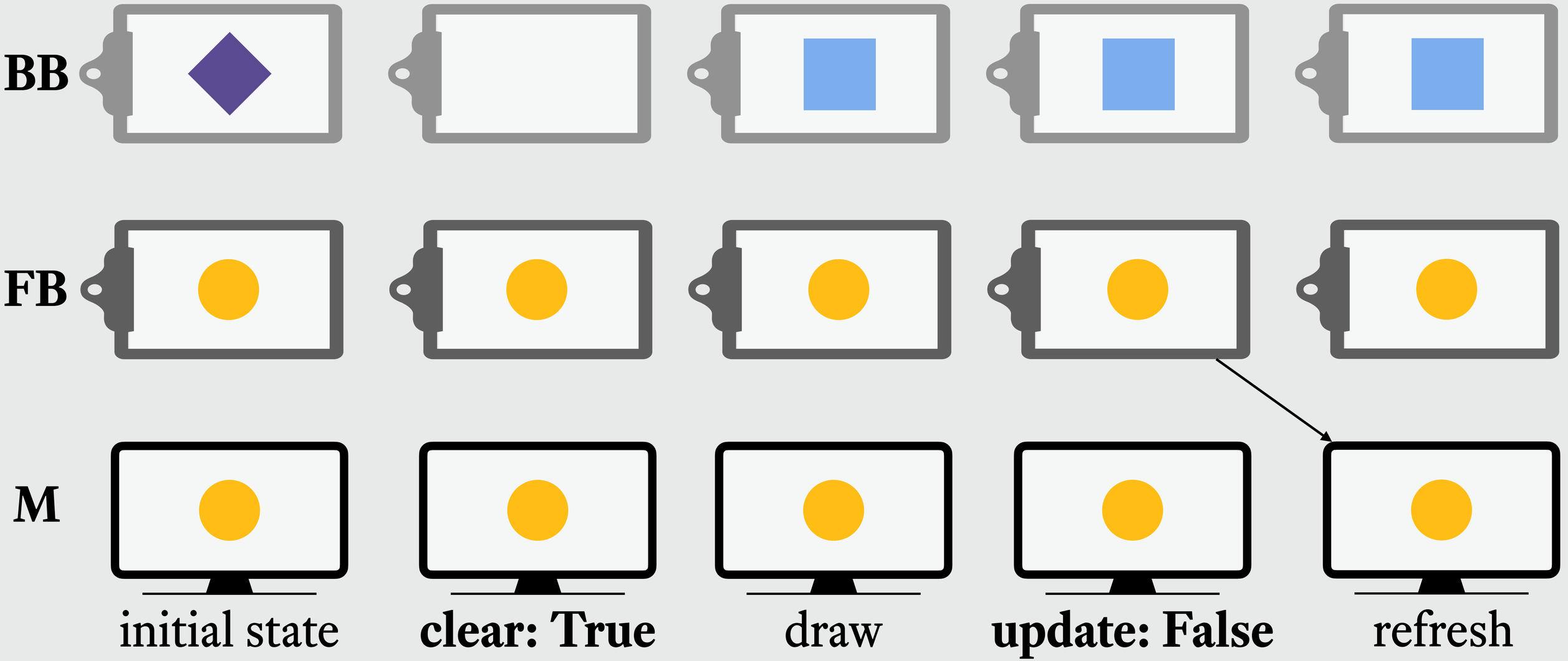
```
exp = design.Experiment()
control.initialize(exp)

fixation = stimuli.FixCross()
square = stimuli.Rectangle(size=(l, l), line_width=5)

control.start(subject_id=1)
fixation.present(clear=True, update=True)
exp.clock.wait(500)

square.present(clear=False, update=True)
exp.keyboard.wait()

control.end()
```

Goal

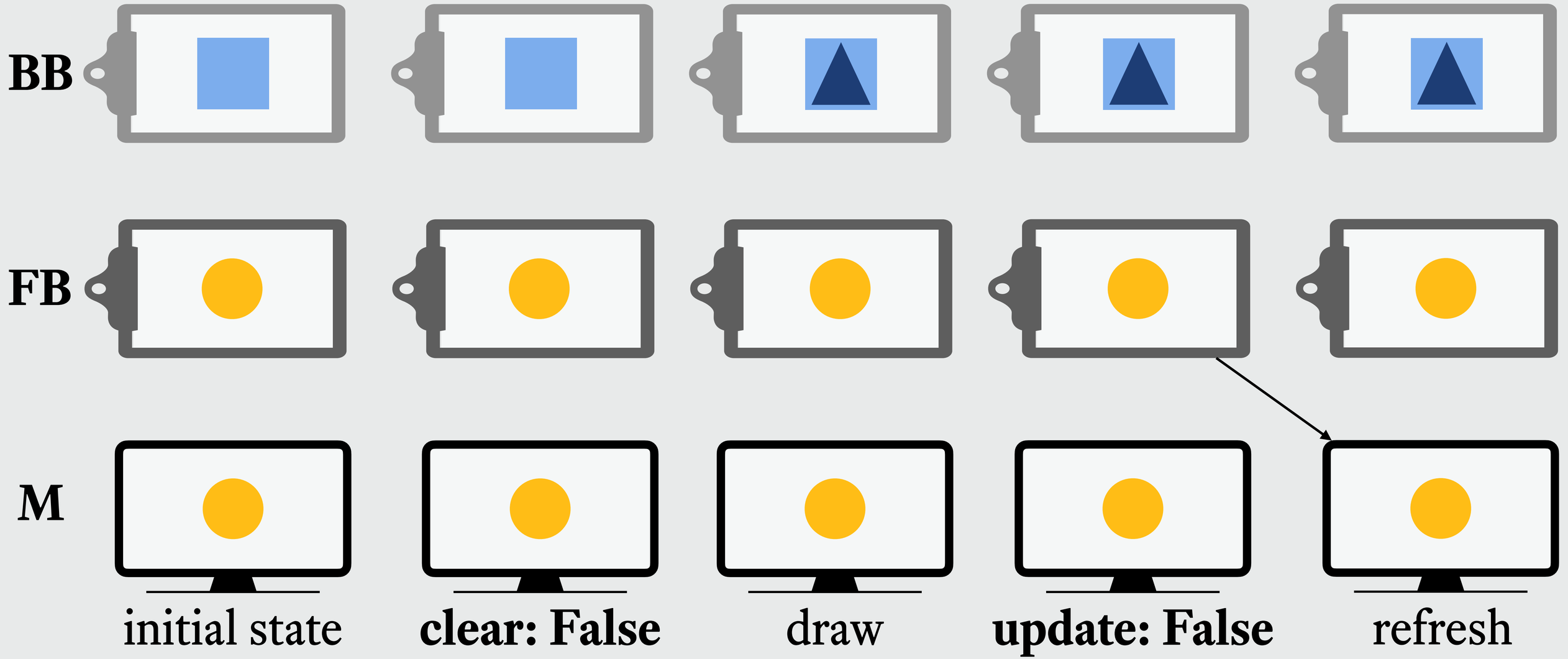


Reality

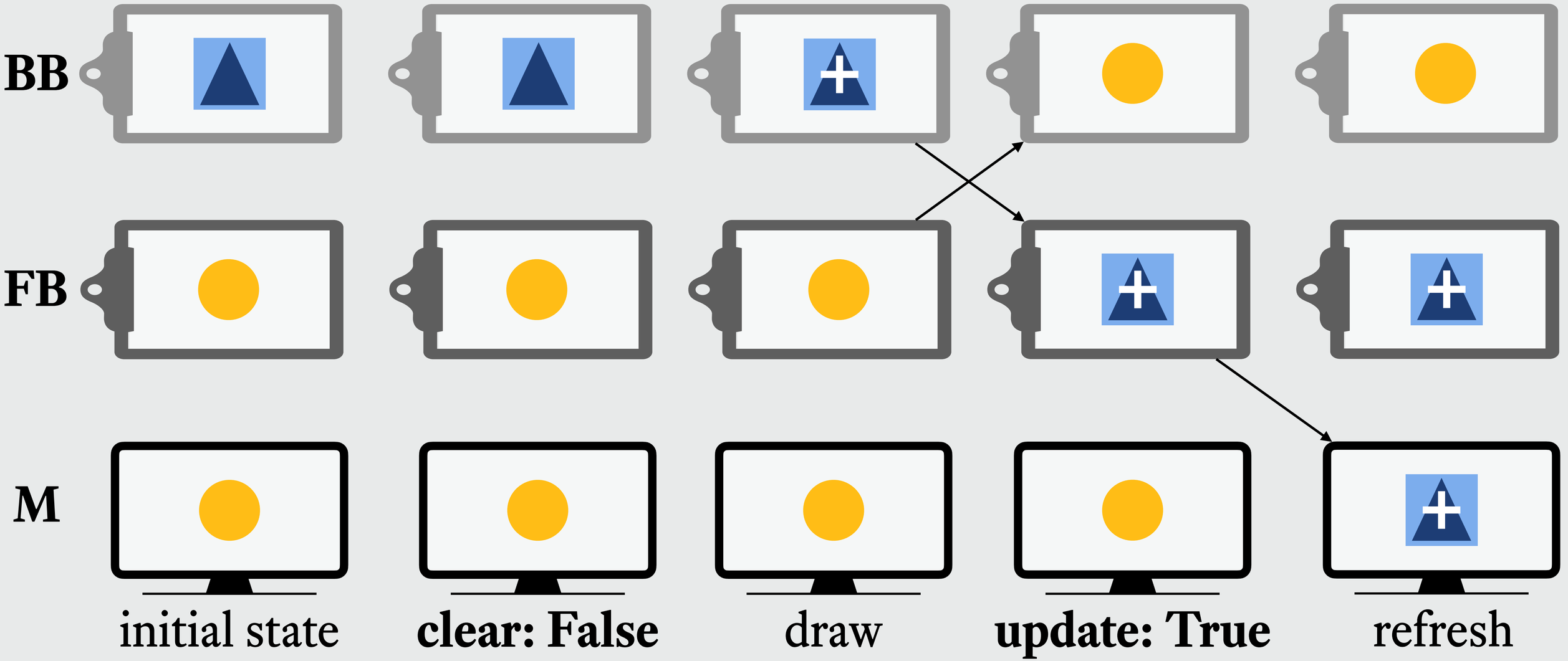
sq.present(clear=T, update=F)



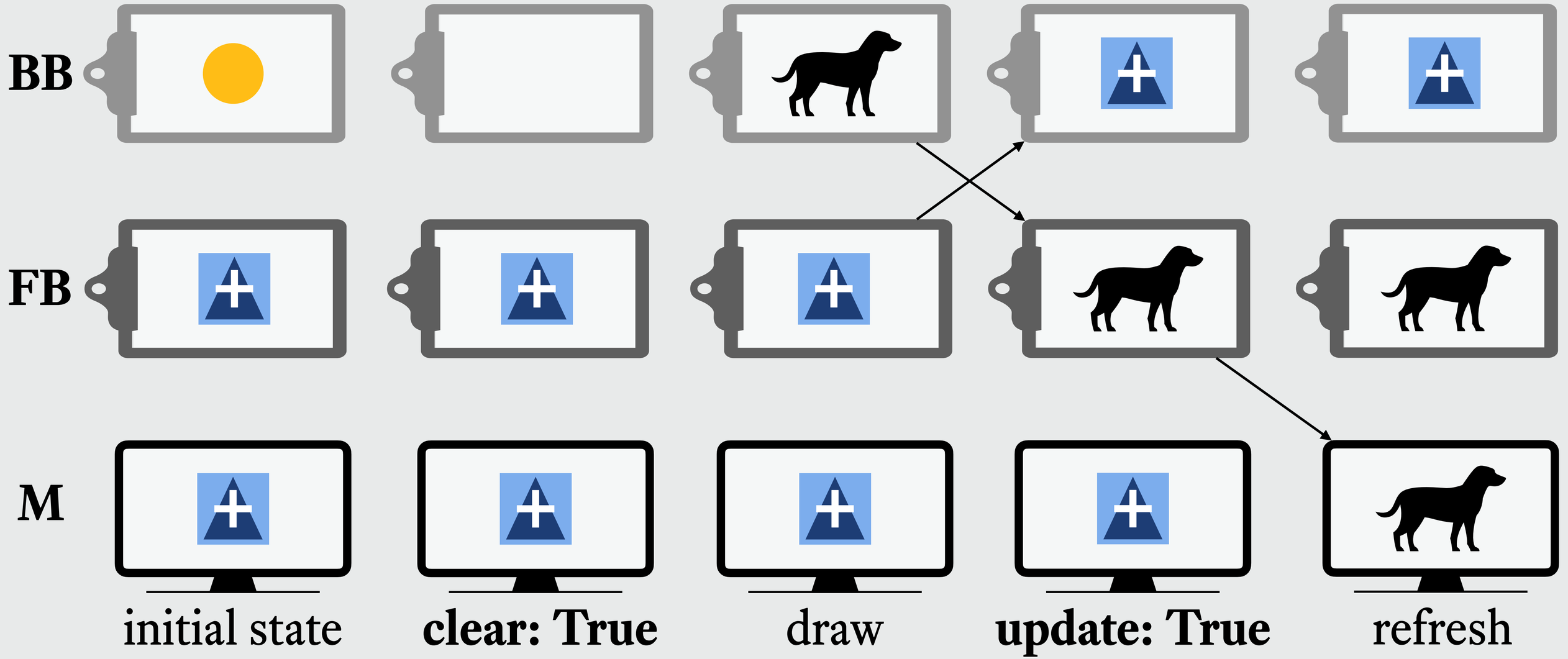
```
tri.present(clear=F, update=F)
```



`fix.present(clear=F, update=T)`



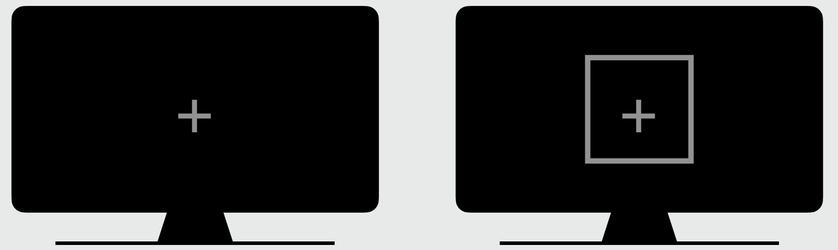
dog.present(clear=T, update=T)



Exercise 1: Fix `square_fixation.py`

```
...

control.start(subject_id=1)
fixation.present(clear=True, update=True)
exp.clock.wait(500)


square.present(clear=False, update=True)
exp.keyboard.wait()

control.end()
```

Goal



Reality

Overview

If you want to draw n stimuli on-screen, you must call `present()` with 3 different clear–update combinations:

S_1: `stim_first.present(clear=True, update=False)`

$S_2, \ldots, S_{n-1}$: `stim.present(clear=False, update=False)`

S_n: `stim_last.present(clear=False, update=True)`

This is a problem for several reasons, so let's see how we can avoid doing this by hand

Solution 1: A `drawing` function

```
# First attempt

def draw(stims):
    """Draw a sequence of stimuli in order and show on-screen."""

    # Clear the back buffer and draw the first stimulus
    stims[0].present(clear=True, update=False)

    # Continue drawing the middle stimuli
    for stim in stims[:-1]:
        stim.present(clear=False, update=False)

    # Add the last stimulus and swap the buffers
    stims[-1].present(clear=False, update=True)
```

Solution 1: A `drawing` function

```
# First edge case: Empty list

def draw(stims):
    """Draw a sequence of stimuli in order and show on-screen."""

    if not stims:
        raise ValueError("Stimuli list must be nonempty.")

    # Clear the back buffer and draw the first stimulus
    stims[0].present(clear=True, update=False)

    # Continue drawing the middle stimuli
    for stim in stims[:-1]:
        stim.present(clear=False, update=False)

    # Add the last stimulus and swap the buffers
    stims[-1].present(clear=False, update=True)
```

Solution 1: A `drawing` function

```
# Second edge case: 1-element list

def draw(stims):
    """Draw a sequence of stimuli in order and show on-screen."""

    # If only 1 stimulus, draw and update at the same time
    if len(stims) == 1:
        stims[0].present(clear=True, update=True)

    else:
        ...
```

Solutions 2 & 3: Simpler `drawing`

```python
def draw(stims):

    # Clear first, update last
    for i, stim in enumerate(stims):
        stim.present(clear=(i == 0), update=(i == len(stims) - 1))
```

```python
def draw(stims):

    # Clear the back buffer
    exp.screen.clear()

    # Draw to back buffer
    for stim in stims:
        stim.present(clear=False, update=False)

    # Swap the buffers
    exp.screen.update()
```

Can you spot an important difference between these functions?

Solution 4: Drawing on a `canvas`

```
# Generate an empty, transparent surface on which you can draw stimuli
canvas = stimuli.Canvas(size=exp.screen.size)

def draw(stims, canvas):

    # Clear the canvas
    canvas.clear_surface()

    # Draw to canvas
    for stim in stims:
        stim.plot(canvas)

    # Swap the buffers
    canvas.present(clear=True, update=True) # or just canvas.present()
```

Summary

All three methods are **valid ways of drawing stimuli** on-screen

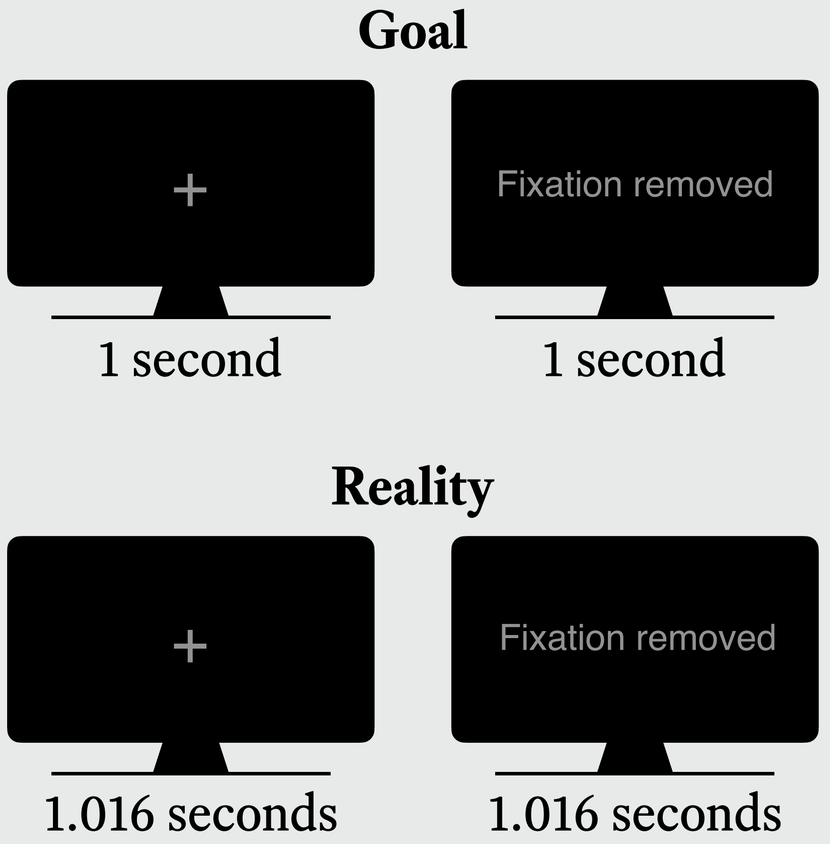
While the details vary, the logic in all cases is the same—before any stimulus presentation, you need to go through 3 steps:

1. **Clear the surface** (back buffer or canvas)
2. **Draw the stimuli**
3. **Push the stimuli to the front buffer**

Expyriment timing

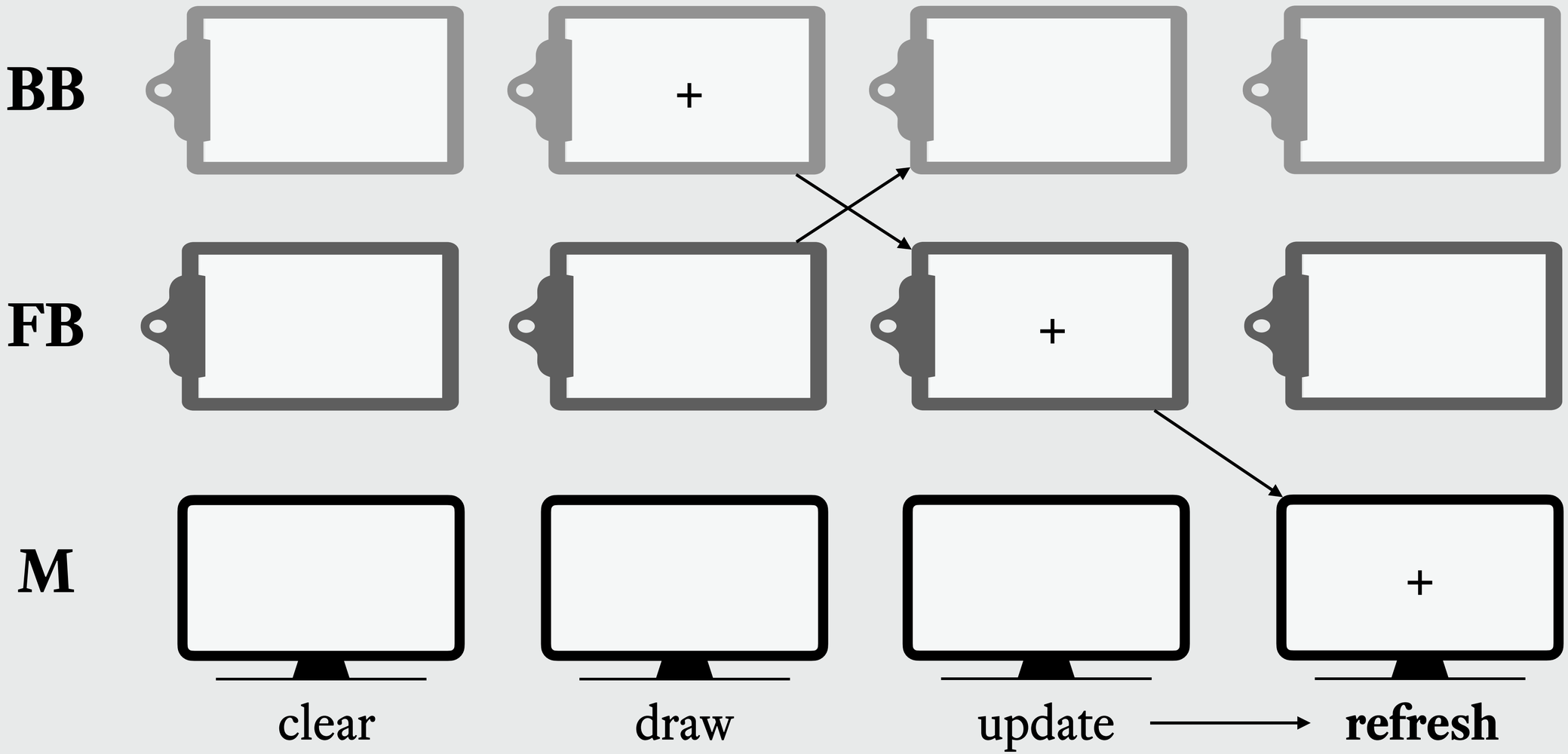
What does this code do?

```
fixation = stimuli.FixCross()
text = stimuli.TextLine('Fixation removed')

fixation.present()
exp.clock.wait(1000)

text.present()
exp.clock.wait(1000)
```



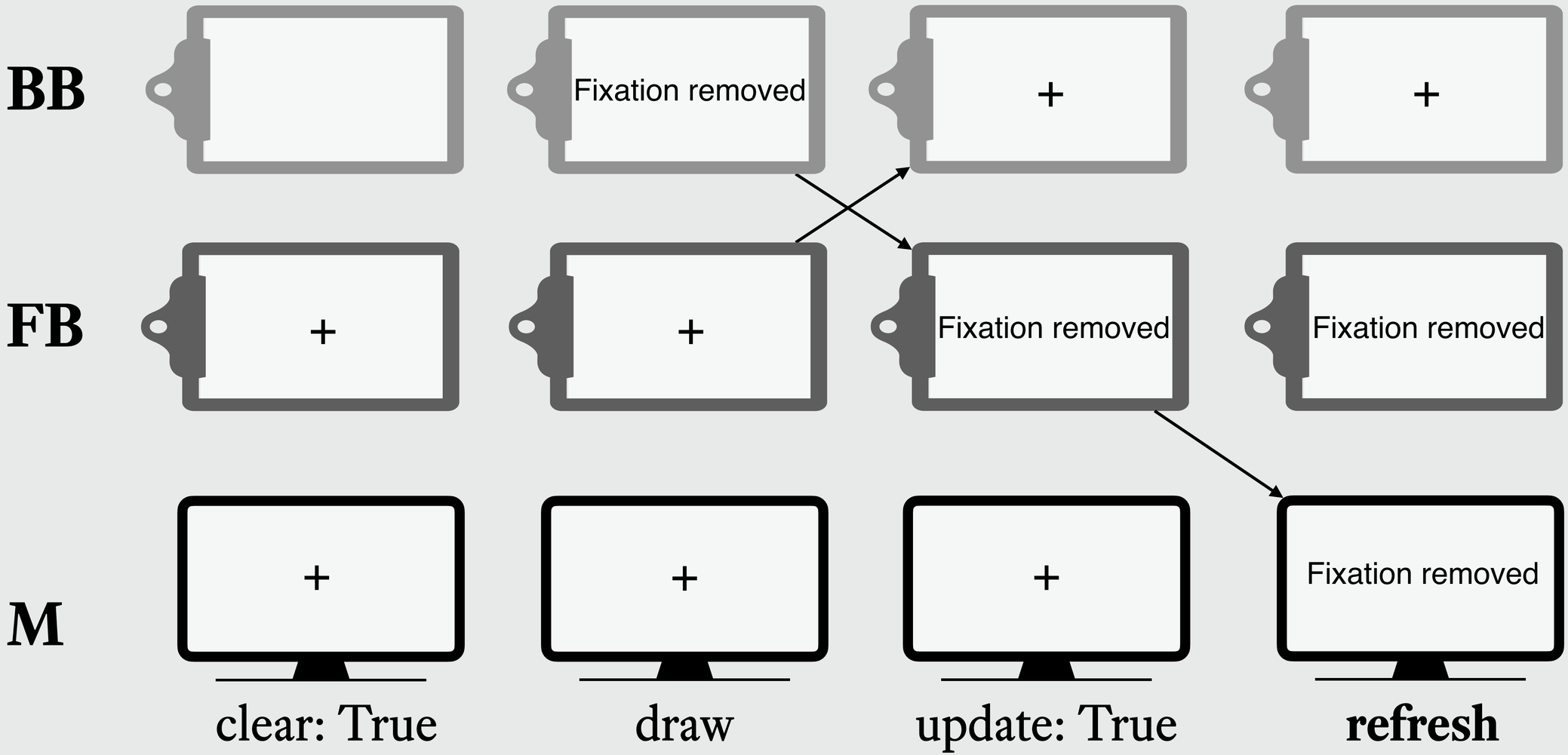
Run **`timing.py`** in Assignments/Exercises

`fixation.present()`



How long does this step take?

text.present()



Solving the puzzle

`fixation.present()`: When called, it will take a variable amount of time to return (0–16.67 ms), depending on **when in the screen refresh cycle** it was called

> Suppose screen was 5 ms away from next refresh when `present()` was called
> This means that at t = 5 ms, the fixation cross will be shown on-screen

`exp.clock.wait(1000)`: Halt program execution for 1 second

> Nothing funky will happen here, so at the end t = 1,005 ms

`text.present()`: This is **not instantaneous**, so the text will be presented at the next refresh rate, which is 1,005 ms + 16.67 ms = 1,021.67 ms

Addressing the problem (1)

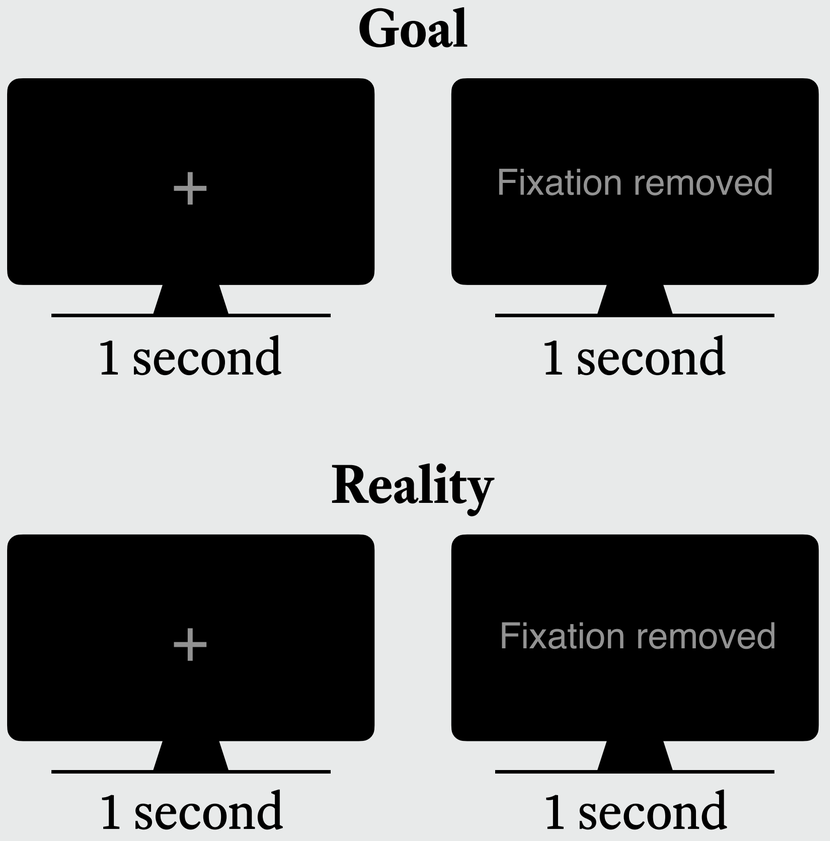
Time the drawing of the stimuli, then subtract this value from the duration you want the stimuli to be present on-screen

The solution may not be very elegant but it works

```
t0 = exp.clock.time
draw(stims)
dt = exp.clock.time - t0
exp.clock.wait(1000 - dt)
```

Exercise 2: Fix `timing_puzzle.py`

```
fixation = stimuli.FixCross()
text = stimuli.TextLine('Fixation removed')

t0 = exp.clock.time
fixation.present()
dt = exp.clock.time - t0
exp.clock.wait(1000 - dt)

t0 = exp.clock.time
text.present()
dt = exp.clock.time - t0
exp.clock.wait(1000 - dt)
```



Addressing the problem (2)

To have the stimuli ready for fast presentation, preload them to memory in advance by using **`stimulus.preload()`** for each stimulus after defining them

This **reduces the time** it takes `expyriment` to draw the stimuli to the buffer

Use a `preloading` function

```
def load(stims):
    for stim in stims:
        stim.preload()
```

Note. If using the canvas method, you must take care of preloading every single time

```
def draw(stims, canvas):
    canvas.clear_surface() # Clearing the canvas leads to unloading
    canvas.preload()

    for stim in stims:
        stim.plot(canvas)

    canvas.present()
```

Limitations

Timing accuracy in expyriment works properly **only in full-screen mode**

More serious limitation: There's **no way to check timing accuracy with high precision** using only the presentation computer

- This is why labs use **photodiodes to measure** when and for how long a stimulus was onscreen
- In the absence of that, **we should still do our best** to code scripts properly, taking into consideration all available information

Exercise 3

Write a script that contains three functions:

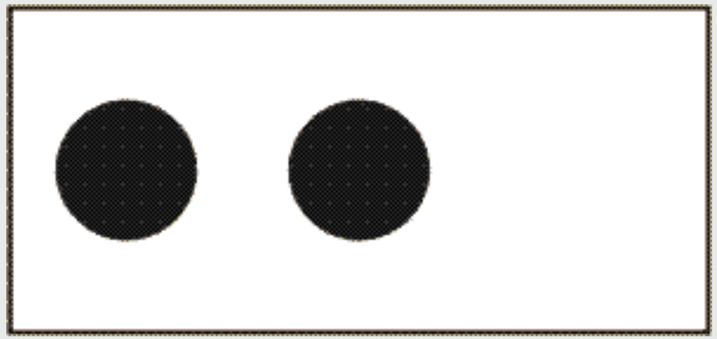
1. **`preloading:`** preload all the stimuli passed as input
2. **`draw:`** draw a list of (preloaded) stimuli on-screen
3. **`present_for:`** draw and keep stimuli on-screen for time *t* in ms (be mindful of **edge cases**!)

Hint: Have **`draw`** return its execution time & use *that* inside **`display`**

Exercise 4

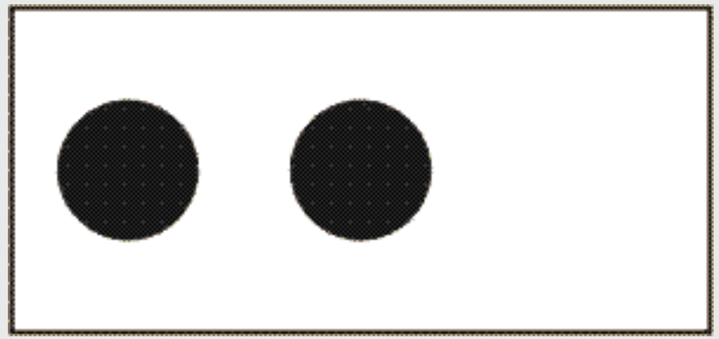
The **Ternus illusion** illustrates a very thorny problem in CogSci: **How do our cognitive systems keep track of object identity over time?**

Element motion



ISI < 50 ms

Group motion



ISI > 50 ms

Exercise 4

Create a customizable Ternus display similar to the one **here** with parameters for **radius**, **ISI** (the amount of time between the circle displays) and **color tags** (for tracking the circles)—you can set the **duration of the circle displays to 200 ms**

When run, the program should present **three Ternus displays** in succession:

1. Element motion without color tags (low ISI)
2. Group motion without color tags (high ISI)
3. Element motion with color tags (high ISI)

Exercise 4: Caveats

Make sure it works on yourself! With some ISIs, you will get bistable perception—try squinting or fixating at the central circles to flip the identity interpretation

Don't forget to preserve the structure of expyriment scripts: Global settings → Stimuli generation → Trial run

Write functions: `present_for, make_circles, add_tags`

Aim for compact code (< 80–90 lines)

Exercise 4: Hints

Pay attention to how you implement the **blank screen duration** (What *should* happen when ISI = 0? What will happen when ISI = 5?)

In the loop for the display (e.g., `while True`), add a command that checks for user input and exits the loop if SPACE was pressed

```
from expyriment.misc.constants import K_SPACE # top of script

if exp.keyboard.check(K_SPACE): # inside the loop
    break
```

Exercise 4: Hints

Since a frame is 16.67 ms, best to **use frames instead of times**

- 1 second = 60 frames; 500 ms = 30 frames; 100 ms = 6 frames …

Take **frame input** and **convert to milliseconds** internally

This will help you **avoid rounding errors** or **passing in meaningless commands** (such as present for 12 milliseconds)

Exercise 4: Hints

As we've seen with `canvases`, expyriment lets you plot stimuli onto the surface of other stimuli

You can add tags by plotting small circles on the surface of the big ones, but be mindful of two aspects:

1. The big circles must be preloaded *after* the plotting of the tags
2. The positions of the tags must be set *relative* to the circles on top of which they're plotted (easier than it sounds)

Push your work to GitHub

Homework: Leftover exercises