EEG and Unity Timestamps Alignment and Trigger Files for ERP Analysis

This notebook is intended to create the descriptive, streams information and timestamps as '.csv' files, and trigger files. - Descriptive files contain information about the recordings such as duration, sampling rate, names of streams collected, type, etc. - Stream information files contain all data gathered during recording with respective timestamps (e.g., head and eye tracking, object names) - The trigger files contain information about the time when an image was shown, the type of image, distance, rotation and block. These files are required to set the stimulus onset during the eeg data analysis.

[ ]:
```
import copy
import datetime
import itertools
import os
import matplotlib.pyplot as plt
from collections import OrderedDict
from matplotlib import patches
import matplotlib.gridspec as gridspec
import numpy as np
from IPython.display import display
from matplotlib.ticker import FormatStrFormatter
import pandas as pd
import pyxdf
import dataframe_image as dfi
import seaborn as sns
from scipy.signal import find_peaks
from operator import itemgetter
from tqdm.notebook import tqdm
```

[ ]:
```
# choosing the color palette
deep_pal = sns.color_palette('deep')
sns.palplot(deep_pal)
colors = dict(startMessage=deep_pal[6], fixCross=deep_pal[5], image=sns.
 ↪color_palette("hls", 10)[9], grayCanvas=deep_pal[4], endMessage=deep_pal[7])
# colors_cat = dict(object=deep_pal[0], face=deep_pal[1], body=deep_pal[3])
sns.set_style("white")
```

[ ]:
```
sns.color_palette("hls", 12)
```

Importing XDF Files

[ ]:
```
# path to data stored
r_path = "data"
# path to store trigger data
t_path = r_path + '/triggers'
# path to store eye-tracking data
e_path = r_path + '/eye_tracking'
# path to store subjects' full streams data
all_csv_path = r_path + '/streams_csv'

# Get some quick idea about the files
files = os.listdir(r_path)  # get all files from the folder "data"
files.sort()  # sort them alphabetically
recordings = {}
file_names = []
for i, file in enumerate(files):  # store and display all files
    if file.endswith('.xdf') and 'room1' in file:
        file_names.append(file)
```

```
        created = os.path.getmtime(f"data/{file}")  # creation timestamp
        created = datetime.datetime.fromtimestamp(created)  # translate as datetime
        created = created.strftime("%d.%m.%Y %H:%M")  # arrange it
        recordings[i] = {"file": file, "created": created}
        #check for a second recording for same subject
        if os.path.isfile(f"data/{file.replace('room1', 'room2')}"):
            print(f"2nd room exists for {file}")

# print(file_names)
files = [f.split(".")[0] for f in files]
print("Included:")
display(recordings)
```

Load data

[ ]:
```
# check streams for recording 0
file_to_use = file_names[2]
print(file_to_use)
streams, _ = pyxdf.load_xdf(f"data/{file_to_use}")
```

[ ]:
```
# stream channel names in recording 0
s_channels = {streams[i]["info"]["name"][0]: i for i in range(len(streams))}
s_channels
# Get an idea of stream 'ImageInfo'
# streams[7]
# streams[7]['info']
# streams[7]['time_series']
# streams[7]['time_stamps']
# length of time_stamps
# len(streams[2]['time_stamps'])
# print all source_ids in stream 7
# source_ids = [i['info']['source_id'] for i in streams]
# source_ids
```

1.2 Important Functions

[ ]:
```
def stract_eeg_data(streams, time_ref_stream='openvibeSignal'):
    s_channels = {streams[i]["info"]["name"][0]: i for i in range(len(streams))}
    eeg = s_channels[time_ref_stream]
    # choose 6144 samples which correspond to 6 sec of recording
    df_eeg = pd.DataFrame(streams[eeg]['time_series'][0:6144,0:10]).
 ↪rename(columns={0: "Ch1", 1: "Ch2", 2: "Ch3",3: "Ch4", 4: "Ch5",5: "Ch6", 6:␣
 ↪"Ch7", 7: "Ch8",8: "Ch9", 9: "Ch10"})
    df_eeg['timestamps'] = streams[eeg]['time_stamps'][0:6144] -␣
 ↪streams[eeg]['time_stamps'][0]
    # have all timestamps
    df_eeg_all = pd.DataFrame(streams[eeg]['time_series'][:,0:10]).
 ↪rename(columns={0: "Ch1", 1: "Ch2", 2: "Ch3",3: "Ch4", 4: "Ch5",5: "Ch6", 6:␣
 ↪"Ch7", 7: "Ch8",8: "Ch9", 9: "Ch10"})
    df_eeg_all['timestamps'] = streams[eeg]['time_stamps']
    return df_eeg, df_eeg_all
```

[ ]:
```
df_eeg, df_eeg_all = stract_eeg_data(streams)
df_eeg_all
```

[ ]:
```
# visualize eeg data only
def visualize_eeg(df):
    fig, ax = plt.subplots(nrows=10, figsize=(15, 6), sharex=True)
```

```python
    plt.rcParams.update({'font.size': 10})
    for i, value in enumerate(df.columns[0:10]):
        sns.lineplot(x= df['timestamps'] - df['timestamps'][0], y = df[value],␣
 ↪ax=ax[i], linewidth=0.5)
        ax[i].set(yticklabels=[])
        ax[i].set_xlim(0,None)
        ax[i].set_xlabel('Time (s)', fontsize=12)
        sns.despine(top=True, bottom=True)
    plt.show()
# plt.savefig("data/images/" + "eeg_6seconds" + ".pdf", format='pdf', dpi=1200)
```

[ ]:
```python
visualize_eeg(df_eeg)
```

[ ]:
```python
def get_stream_timestamps(streams, streams_keep=['ImageInfo','Visual'],␣
 ↪time_ref_stream='openvibeSignal', use_manual_drift = False, useAllStreams =␣
 ↪False):
    s_channels = {streams[i]["info"]["name"][0]: i for i in range(len(streams))}
    times_egg_start_ts = streams[s_channels[time_ref_stream]]['time_stamps'][0]
    times_egg_end_ts = streams[s_channels[time_ref_stream]]['time_stamps'][-1]

    # Save user ID from Visual stream
    uid = streams[s_channels['Visual']]['info']['uid'][0]
    print(f"\nParticipant UID: {uid}")

    times_unity_min = []
    times_unity_max = []
    streams_used = []

    if useAllStreams:
        streams_keep = [streams[i]["info"]["name"][0] for i in range(len(streams))]

    for i in range(len(streams)):
        stream_name = streams[i]["info"]["name"][0]
        if stream_name in streams_keep and stream_name != time_ref_stream:
            if len(streams[i]['time_stamps']) > 0:
                times_unity_min.append(streams[i]['time_stamps'][0])
                times_unity_max.append(streams[i]['time_stamps'][-1])
                streams_used.append(stream_name)
    # select earliest timestamp
    times_unity_start_ts = np.min(times_unity_min)
    # select latest timestamp
    times_unity_end_ts = np.max(times_unity_max)

    # The following code calculates 3 different issue
    # issue A: delayed start of unity streams vs. eeg streams
    # issue B: initial drift between time stamps of unity and egg streams
    # issue C: linearly increasing drift between time stamps of unity and egg␣
 ↪streams over time

    # Read drifts from recording notes
    if use_manual_drift and os.path.isfile(path = r_path + '/custom_drifts.csv'):
        cdrifts = pd.read_csv(r_path + '/custom_drifts.csv')
    else:
        cdrifts = pd.DataFrame(columns=['uid'])

    # 0/ Custom drifts are defined for given participant taken from participant␣
 ↪notes
```

```
    if len(cdrifts[cdrifts['uid'] == uid]) == 1:
        print(f"!! Custom drifts from participant notes found")
        start_diff = cdrifts[cdrifts['uid'] == uid]['start_drift'].iloc[0]
        end_diff = cdrifts[cdrifts['uid'] == uid]['end_drift'].iloc[0]

    # 1/ normal case without issue A present
    else:
        start_diff = times_unity_start_ts - times_egg_start_ts # start difference between unity and eeg # issue B
        end_diff = times_unity_end_ts - times_egg_end_ts # end difference between unity and eeg

    dynamic_shift = end_diff - start_diff # drift increase over time # issue C
    delayed_start_shift = 0

    # 2/ In case drift is larger at the start vs. at the end, we deal with a case of delayed recording start, i.e. issue A is present
    # we need to calculate the dalay in recording and correct for it to know the dynamic shift
    ##
    if start_diff > end_diff:
        # assume normalized drift in ms per one minute of duration of -1.25256887748983
        normalized_drift_over_time = -1.25256887748983
        duration = (times_egg_start_ts - times_egg_end_ts) / 60
        dynamic_shift = min(end_diff, duration * normalized_drift_over_time / 1000) # issue C, calculated based on assumed normalized drift in ms per minute of duration
        start_diff = end_diff - dynamic_shift # issue B, calculated based that issue C is now known
        delayed_start_shift = (times_unity_start_ts - times_egg_start_ts) - start_diff # issue A, backed out using corrected B, it is going to be zero for case 0/
    ##
    print(f"Streams to keep: {streams_used} and reference stream ['{time_ref_stream}']")
    print(f"{time_ref_stream} start time is {times_egg_start_ts} and end time is {times_egg_end_ts}")
    print(f"Unity start time is {times_unity_start_ts} and end time is {times_unity_end_ts}")
    if delayed_start_shift != 0: # report that issue A is present
        print(f" !! Unity delayed recording start by {delayed_start_shift} seconds detected")
    i = 0
    for _, ch_name in enumerate(streams_used):
        print(f" -- Unity {ch_name} start time is {times_unity_min[i]} and end time is {times_unity_max[i]}")
        i += 1
    print(f" -- All Unity streams start time difference is {round(1000 * (np.max(times_unity_min) - np.min(times_unity_min)), 1)} (milliseconds)")
    print(f" -- All Unity streams end time difference is {round(1000 * (np.max(times_unity_max) - np.min(times_unity_max)), 1)} (milliseconds)")
    print(f"Starting drift between Unity and {time_ref_stream} is {1000*start_diff} (milliseconds)")
    if delayed_start_shift != 0: # report that correct value for issue B had to be calculated using assumption
```

```
        print(f" !! Making corrections using a normalized drift in ms per one␣
↪minute of duration at -1.25256887748983")
    print(f"Ending drift between Unity and {time_ref_stream} is {1000*end_diff}␣
↪(milliseconds)")
    print(f"Additional drift over time is {round(1000*dynamic_shift,2)}␣
↪(milliseconds)")

    return times_egg_start_ts, times_egg_end_ts, times_unity_start_ts,␣
↪times_unity_end_ts, start_diff, end_diff, dynamic_shift, delayed_start_shift
```

[ ]:
```
get_stream_timestamps(streams, streams_keep=['ImageInfo','Visual'],␣
↪useAllStreams=False)
```

[ ]:
```
(90275.76089333574 - 90275.54623195817) * 1000
```

[ ]:
```
def get_continuous_time_periods(time_stamps_input):
    # duration = (time_stamps_input[-1] - time_stamps_input[0])/60
    # print(f'Recording duration: {duration} minutes')
    time_stamps_shift = time_stamps_input[1:-1] - time_stamps_input[0:-2]
    time_stamps_shift_pd = pd.array(time_stamps_shift)
    cutoff = np.mean(time_stamps_shift) + 2*np.std(time_stamps_shift)
    #cutoff = 30
    last = 0
    chunks = []
    for i, val in enumerate(time_stamps_shift):
        if val > cutoff:
            chunks.append([time_stamps_input[last:i+1]])
            last = i+1
    chunks.append([time_stamps_input[last:(np.size(time_stamps_input))]])
    return chunks

def calcualte_fps(time_stamp_chunk):
    frame_count = np.size(time_stamp_chunk)
    chunck_duration = (np.max(time_stamp_chunk) - np.min(time_stamp_chunk) )/ 60
    # if chunck_duration > 1:
    fps = frame_count / (chunck_duration * 60)
    return fps

total_size = 0
time_stamps = streams[s_channels['Visual']]['time_stamps']
for chunk in get_continuous_time_periods(time_stamps):
    print(calcualte_fps(chunk))
    total_size += np.size(chunk)
    print(np.size(chunk))
    print(chunk)

print(f"Total input array size {np.shape(time_stamps)} vs. summed chunk sizes␣
↪{total_size}")
print(f"first entry {time_stamps[0]} and last entry {time_stamps[-1]}")
```

[ ]:
```
def describe_recordings(streams, file_name=''):
    ch_keys = ["name","type",␣
↪"channel_count","channel_format","nominal_srate","desc","effective_srate","hostname",␣
↪"created_at"]
    s_channels = {streams[i]["info"]["name"][0]: i for i in range(len(streams))}
    # Save user ID from Visual stream
    uid = streams[s_channels['Visual']]['info']['uid'][0]
```

```
    # Get the streams timestamps
    times_egg_start_ts, times_egg_end_ts, times_unity_start_ts, times_unity_end_ts, start_diff, end_diff, dynamic_shift, delayed_start_shift = get_stream_timestamps(streams)
    # Calculate the total duration of the recording
    duration = (times_egg_end_ts - times_egg_start_ts) / 60
    # Calculate the mean FPS for the Unity stream (it should be 90 fps)
    fps = np.mean([calcualte_fps(chunk) for chunk in get_continuous_time_periods(streams[s_channels['Visual']]['time_stamps'])])
    # print(fps)

    df = pd.DataFrame(columns=np.concatenate([['uid','file_name','duration (min)','unity_avg_fps','unity_recording_delayed_start(s)',
                                               'eeg_unity_ts_delta_start(s)','eeg_unity_ts_delta_end(s)',
                                               'drift_over_time(s)','stream_id'], ch_keys]))
    id_o = []
    for id, stream in enumerate(streams):
        id_o.append(id)
        items = itemgetter(*ch_keys)(dict(stream['info']))
        items = [(', '.join(list(item[0].keys())) if 'desc' in ch_keys[i] else (
                    item[0] if isinstance(item, list) else item)) for i, item in enumerate(items)]
        df = df.append(pd.Series(items, index=ch_keys), ignore_index=True)

    df['uid'] = np.resize(uid,len(streams))
    df['file_name'] = np.resize(file_name,len(streams))
    df['duration (min)'] = np.resize(duration,len(streams))
    df['unity_avg_fps'] = np.resize(fps,len(streams))
    df['unity_recording_delayed_start(s)'] = np.resize(delayed_start_shift, len(streams)) # issue A
    df['eeg_unity_ts_delta_start(s)'] = np.resize(start_diff, len(streams)) # issue B
    df['eeg_unity_ts_delta_end(s)'] = np.resize(end_diff, len(streams))
    df['drift_over_time(s)'] = np.resize(dynamic_shift,len(streams)) # issue C
    df['stream_id'] = id_o

    return df
```

[ ]:
```
desc_streams = describe_recordings(streams, file_name=file_to_use)
# dfi.export(desc_streams,'data/description.png')
desc_streams
```

2 Create dataframes from streams

Important: Initially, we normalized Unity streams to start from zero by substracting every timestamp in the "Visual" stream from the first timestamp of the EEG 'openvibeSignal' stream. However, after we noticed a delay in the Unity streams with respect to the EEG data, we decided to assume that time Zero (i.e., the time we started recording) is the first timestamp in each stream (Unity and EEG). Thus, we substracted the timestamps in the "Visual" stream by the first timestamp in it. We found out that the dalay is due to a "high" negative "DRIFT" in the OpenVibe Acquisition Server (OVAS) system with which we recorded the EEG data.

[ ]:
```
```

```python
# Function to create DF from streams
def get_streams_data(streams, streams_keep=['ImageInfo','Visual'],␣
↪time_ref_stream='openvibeSignal', use_manual_drift = False,␣
↪use_startdiff_correction = True):
    """
    :param streams: streams after loading from .xdf file
    :param streams_keep: str. of the stream names to keep
    :param time_ref_stream: str. name of the eeg  signal used used for time␣
↪correction and uid creation
    :return: df containing the time_stamps and stream_data as columns for each␣
↪stream to keep
    """
    data = pd.DataFrame()
    # Save user ID from Visual stream
    s_channels = {streams[i]["info"]["name"][0]: i for i in range(len(streams))}
    uid = streams[s_channels['Visual']]['info']['uid'][0]
    times_egg_start_ts, _, times_unity_start_ts, _, start_diff, _, dynamic_shift, _␣
↪= get_stream_timestamps(streams, streams_keep, time_ref_stream, use_manual_drift)
    # data['uid'] = np.resize(uid,len(data))
    # Calculate if a shift is needed per stream when time stamps differ
    stream_ts_info = {}
    for i, ch_name in enumerate(streams_keep):
        u = s_channels[ch_name]
        if round(1000 * streams[u]['time_stamps'][0], 0) == round(1000 *␣
↪times_unity_start_ts, 0):
            print(f"Shift channel '{ch_name}' by -1")
            stream_ts_info[u] = -1
        else:
            stream_ts_info[u] = 0

    if not use_startdiff_correction:
        start_diff = 0

    for i, ch_name in enumerate(streams_keep):
        # get all current streams with their positions on the recording
        # example: {'ImagesOrder': 0, 'ValidationError': 1, 'HeadTracking': 2}
        # s_channels = {streams[i]["info"]["name"][0]: i for i in␣
↪range(len(streams))}
        u = s_channels[ch_name]
        # save the recording UID and append to df
        data['uid'] = np.resize(uid,len(data))
        # check the type and length of data arrays and get only 1 value of the array
        stream_data = streams[u]['time_series']
        time_stamps = streams[u]['time_stamps'][-stream_ts_info[u]:]

        # linear de-drift of unity timestamps
        print(f"\nDedrifting channel '{ch_name}'...")
        ts_chunks = get_continuous_time_periods(time_stamps)
        fps_avg = np.mean([calcualte_fps(chunk) for chunk in ts_chunks])
        ts_complete = np.array([])
        ts_correction = np.array([])
        for i, chunk in enumerate(ts_chunks):
            chunk = np.resize(chunk, np.size(chunk))
            if i > 0:
                missing_chunk_length = int(round(fps_avg *␣
↪(ts_chunks[i][0][0]-ts_chunks[i-1][0][-1]), 0))
                np_zeros = np.zeros(missing_chunk_length, dtype=int)
```

```
                print(f"Missing chunk {i} length {missing_chunk_length}")
                #print(f"Appending filler for missing chuck of size {np.
↪size(np_zeros)}")
                ts_complete = np.append(ts_complete, np_zeros, axis=0)
                ts_correction = np.append(ts_correction, np_zeros, axis=0)
            ts_complete = np.append(ts_complete, chunk, axis=0)
            ts_correction = np.append(ts_correction, np.ones(np.size(chunk),␣
↪dtype=int), axis=0)
            #print(f"Appending original chunk of size {np.size(chunk)}")
        #ts_complete = ts_complete.transpose()
        drift_correction_np = np.linspace(0, dynamic_shift, len(ts_complete))
        # linearly correct the timestamps from start to end given the drift␣
↪increase over time 'dynamic_shift'
        time_stamps_dedrifted = ts_correction * (ts_complete - drift_correction_np)
        time_stamps_dedrifted = time_stamps_dedrifted[time_stamps_dedrifted>0]
        print(f"Size of original timestamps {np.shape(time_stamps)}")
        print(f"Size of ts_complete {np.shape(ts_complete)}")
        print(f"Size of dedrifted timestamps {np.shape(time_stamps_dedrifted)}")

        delta_drift_correction = abs(time_stamps_dedrifted - (time_stamps - np.
↪linspace(0, dynamic_shift, len(time_stamps_dedrifted))))
        print(f"Max time stamp correction improvement vs. simple linear drift␣
↪correction = {round(1000 * np.max(delta_drift_correction), 3)} ms")

        # double check keys on each stream to make sure they are all appended to df
        # print(f"Stream {ch_name} keys: {streams[u]['info']['desc'][0].keys()}")
        # check stram_data is of kind np.array()
        if isinstance(stream_data, (list, pd.core.series.Series, np.ndarray)):
            # access all stream names in dictionary's 'info' description
            for i, key in enumerate(streams[u]['info']['desc'][0].keys()):
                # save each dict key as column to df
                stream_data = pd.DataFrame(streams[u]['time_series'])[i]
                data[f"{key}_{ch_name}"] = stream_data.shift(periods =␣
↪stream_ts_info[u])

        # get timestamps and attach them as column to df
        data = pd.concat([data, pd.DataFrame(time_stamps,␣
↪columns=[f"time_stamps_{ch_name}"]),
                              pd.DataFrame(time_stamps_dedrifted - start_diff,␣
↪columns=[f"corrected_tstamps_{ch_name}"]),
                              pd.DataFrame(time_stamps_dedrifted - start_diff -␣
↪times_egg_start_ts, columns=[f"normalized_tstamps_{ch_name}"])], axis=1)

    return data[:-1] # exclude the last row since it probably contains NaN
```

2.1 Eye-tracking independent Unity streams to DF

E-tracking independent streams refer to those streams that are constantly sending samples into the system after hitting the play button. These streams inform us about the type of objects in the scene and the state of the game at all times.

[ ]:
```
df_img_info = get_streams_data(streams, streams_keep=['ImageInfo','Visual'],␣
↪time_ref_stream='openvibeSignal', use_manual_drift=False,␣
↪use_startdiff_correction=True)
```

[ ]:
```
df_img_info
```

```
[ ]: # Depending on when the recording was started, there may be datasets containing␣
     ↪'startMessage' in the 'ImageInfo' stream since this stream was set to send␣
     ↪samples to LabRecorder at all times.
     # d_selected = df_img_info.loc[~df_img_info['imageName_ImageInfo'].
     ↪isin(['startMessage'])].head(5400)
     # d_selected
```

```
[ ]: # running_number = -1
     def shift_increment(shift):
         """
         :param shift: bool
         :return: a running number for every new shift
         """
         global running_number
         if shift:
             running_number += 1
             return running_number
         else:
             return None

     # rename the columns in the dataframe for better visualization
     def rename_displays(df):
         global running_number
         running_number = -1
         # identify the shifts in the imageInfo stream
         df['shift'] = df['imageName_ImageInfo'].shift(1) != df['imageName_ImageInfo']
         # fill an increasing number everytime a shift occurs
         df['shiftID'] = df.apply(lambda x: shift_increment(x['shift']), axis=1).
     ↪fillna(method='ffill').astype(int)

         df['displayStatusNames'] = df.apply(lambda x: 'image' if 'img' in␣
     ↪x['imageName_ImageInfo'].lower()
                                             else ('fixCross' if 'fixation' in␣
     ↪x['imageName_ImageInfo'].lower()
                                             else ('grayCanvas' if 'gray' in␣
     ↪x['imageName_ImageInfo'].lower()
                                             else ('startMessage' if 'start' in␣
     ↪x['imageName_ImageInfo'].lower()
                                             else ('endMessage' if 'end' in␣
     ↪x['imageName_ImageInfo'].lower()
                                                   else '')))), axis=1)
         return df
```

```
[ ]: df_renamed2 = rename_displays(df_img_info)
     df_renamed2
```

```
[ ]: # TODO: plot the display time duration distribution
     # fig, ax = plt.subplots(nrows=1, figsize=(20, 2))
     # sns.histplot(data=df_renamed['imageName_ImageInfo'])
```

```
[ ]: # visualize the first 16 shifts (i.e., canvas, cross, image) from the Unity streams␣
     ↪together with the first seconds of the EEG channels
     def visualize_unity_eeg(unity_df, eeg_df, id='', display_start=0, display_end=16,␣
     ↪save=True):
         # select the first 16 shifts in the data
         unity_df2 = unity_df[unity_df['shiftID'].between(display_start,display_end)]
```

```
    # create axis subplots with specific ratios
    fig, (a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10) = plt.subplots(11, 1, figsize=(30,18),gridspec_kw={'height_ratios': [1, 0.2, 0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2]})
    plt.rcParams.update({'font.size': 16})
    fig.suptitle("Unity and EEG through time. ID: " + id, fontsize=36, y=0.99, fontweight='bold')
    # colors = dict(startMessage=deep_pal[6], fixCross=deep_pal[4], image=deep_pal[0], grayCanvas=deep_pal[1], endMessage=deep_pal[9])
    ########TODO: increase the dots' transparency to appear stronger in the .pdf img ################
    sns.scatterplot(x=unity_df2['normalized_tstamps_ImageInfo'], y=unity_df2['displayStatus_Visual'], data=unity_df2, hue='displayStatusNames', palette=colors, ax=a0)
    a0.set_ylim(-0.5, 2.2)
    a0.set_xlim(unity_df2['normalized_tstamps_ImageInfo'].min(),unity_df2['normalized_tstamps_ImageInfo'].max())
    a0.set_yticks([-1.5, -1, 0, 1, 2])
    # a0.set_xticks([0,0.5, 1, 1.5, 2, 2.5,3,3.5,4,4.5,5,5.5,6])
    a0.locator_params(axis='both', nbins=33)
    plt.legend(bbox_to_anchor=(1, 1), loc=2, borderaxespad=.0)
    a0.set_title("Unity markers. Sample: " + str(round(unity_df2['normalized_tstamps_ImageInfo'].max())) + ' s', fontsize=26)
    a0.set_xlabel('', fontsize=16)
    a0.set_ylabel('Unity markers', fontsize=22)
    sns.move_legend(a0, "upper left", bbox_to_anchor=(1, 1))
    sns.despine(top=True)
    fig.subplots_adjust(hspace=1.5)
    # Axis list to iterate through
    axis_list = [a1, a2,a3,a4,a5,a6,a7,a8,a9,a10]

    # normalize eeg time_stamps to start from zero
    eeg_df['timestamps'] = eeg_df['timestamps'] - eeg_df['timestamps'][0]
    eeg_df = eeg_df[eeg_df['timestamps'].between(unity_df2['normalized_tstamps_ImageInfo'].min(),unity_df2['normalized_tstamps_ImageInfo'].max())]

    for i, value in enumerate(eeg_df.columns[0:10]):
        sns.lineplot(x= eeg_df['timestamps'], y = eeg_df[value], ax=axis_list[i], linewidth=0.5)
        axis_list[i].set(yticklabels=[])
        axis_list[i].set_xlim(eeg_df['timestamps'].min(),eeg_df['timestamps'].max())
        axis_list[i].spines['bottom'].set_visible(False)
        axis_list[i].set_xlabel('', fontsize=1)
        if axis_list[i] == axis_list[-1]:
            axis_list[-1].set_xlabel('Time (s)', fontsize=22)
            axis_list[-1].spines['bottom'].set_visible(True)
            #axis_list[-1].set_xticks()
        else:
            axis_list[i].set_xticks([])
    a1.set_title("EEG first 10 channels. Sample: " + str(round(eeg_df['timestamps'].max())) + ' s', fontsize=24)
    # plt.tight_layout()
    if save:
        plt.savefig("data/images/" + "displayStatus_" + str(round(unity_df2['normalized_tstamps_ImageInfo'].max())) + '_'+ id + ".pdf", format='pdf',bbox_inches='tight', dpi=1200)
```

```
    del unity_df2
```

```
[ ]: # df_unity_vis = df_dis_renamed[df_dis_renamed['shiftID'] < 16]
visualize_unity_eeg(df_renamed2, df_eeg_all, id='1', display_start=16,␣
 ↪display_end=32, save=False)
```

```
[ ]: # quick statistics to know if all rows have same length
df_eeg.describe()
# df['HON_HitObjectNames'].isnull().sum()
# df['HON_HitObjectNames'].unique()
```

2.2 Eye-tracking streams into DataFrame

Given all eye-tracking related streams have a different length than the “ImageInfo” stream, we create a separate DF using the same function we created for the “ImageInfo” stream data.

```
[ ]: # inspect the eye_tracking data
df_eye = get_streams_data(streams, streams_keep=['HitObjectNames',␣
 ↪'HitPositionOnObjects','HeadTracking','EyeTrackingWorld','EyeTrackingLocal'])
df_eye
```

```
[ ]: # check no NaN in dataset
df_eye.isnull().sum()
```

```
[ ]: df_eye_start = df_eye.head(2700)
df_eye_start
```

```
[ ]: def visualize_eye(df, id='', save=True):
    # take a sample for 30 seconds of recording (30s x 90fps = 2700 datapoints)
    df_eye_start = df.head(2700)

    plt.rcParams.update({'font.size': 14})
    # rename the columns of interest
    eye_selected =␣
 ↪df_eye_start[["normalized_tstamps_EyeTrackingWorld","HTdirectionX_HeadTracking",␣
 ↪"HTdirectionY_HeadTracking","HTdirectionZ_HeadTracking","ETWdirectionX_EyeTrackingWorld",
 ↪"ETWdirectionY_EyeTrackingWorld","ETWdirectionZ_EyeTrackingWorld"]].
 ↪rename(columns={"HTdirectionX_HeadTracking": "HTdirectionX",␣
 ↪"HTdirectionY_HeadTracking": "HTdirectionY","HTdirectionZ_HeadTracking":
 ↪"HTdirectionZ","ETWdirectionX_EyeTrackingWorld": "ETdirectionX",␣
 ↪"ETWdirectionY_EyeTrackingWorld":␣
 ↪"ETdirectionY","ETWdirectionZ_EyeTrackingWorld":"ETdirectionZ"})
    # covert the dataframe to be tidy (long and not wide format)
    df_eye_tidy = eye_selected.melt('normalized_tstamps_EyeTrackingWorld',␣
 ↪var_name="cols",value_name='vals')

    # create the subplots
    fig, ax = plt.subplots(nrows=3, figsize=(20, 8), sharex=True)
    fig.suptitle("Head and eye axis' directions through time. Sample␣
 ↪"+str(round(df_eye_tidy['normalized_tstamps_EyeTrackingWorld'].max()))+" s. ID:␣
 ↪" + id, fontsize=22, y=0.93, fontweight='bold')
    x = sns.lineplot(x="normalized_tstamps_EyeTrackingWorld", y='vals', hue='cols',␣
 ↪data=df_eye_tidy[df_eye_tidy['cols'].str.contains('X')], style='cols', ax=ax[0],␣
 ↪linewidth=3)
    ax[0].set_ylim(-1,1)
    ax[0].set_ylabel("X coordinate", fontsize=16)
    ax[0].set_xlim(0,df_eye_tidy['normalized_tstamps_EyeTrackingWorld'].max())
    x.legend_.set_title(None)
```

```
    y = sns.lineplot(x="normalized_tstamps_EyeTrackingWorld", y='vals', hue='cols',␣
↪data=df_eye_tidy[df_eye_tidy['cols'].str.contains('Y')], style='cols', ax=ax[1],␣
↪linewidth=3)
    ax[1].set_ylim(-1,1)
    ax[1].set_xlim(0,df_eye_tidy['normalized_tstamps_EyeTrackingWorld'].max())
    ax[1].set_ylabel("Y coordinate", fontsize=16)
    y.legend_.set_title(None)
    z = sns.lineplot(x="normalized_tstamps_EyeTrackingWorld", y='vals', hue='cols',␣
↪data=df_eye_tidy[df_eye_tidy['cols'].str.contains('Z')], style='cols', ax=ax[2],␣
↪linewidth=3)
    ax[2].set_ylim(-1,1)
    ax[2].set_xlim(0,df_eye_tidy['normalized_tstamps_EyeTrackingWorld'].max())
    z.legend_.set_title(None)
    ax[2].set_ylabel("Z coordinate", fontsize=16)
    ax[2].set_xlabel("Time (s)", fontsize=16)
    ax[2].locator_params(axis='x', nbins=30)
    plt.tight_layout()
    sns.despine(top=True, bottom=False)
    if save:
        plt.savefig("data/images/" + "axis_directions_" +␣
↪str(round(df_eye_tidy['normalized_tstamps_EyeTrackingWorld'].max())) + '_'+ id +␣
↪".pdf", format='pdf',bbox_inches='tight', dpi=1200)

    del df_eye_start, eye_selected, df_eye_tidy
```

Eye-tracking analysis for data recording 46: The starting time here corresponds to the time unity started collecting samples from the 'Visual' stream which is constantly sending samples to LabRecorder. When visualizing the eye tracking data we can notice that the fist eye data samples collected start at around 60 seconds from the time the LabRecorder was started. Previous to that, there is no data being collected which is how the experiment was designed. This is corroborated by the fact that the initial 'Display status name" in the infoName stream corresponds to the 'startMessage" (see image "display status" image).

```
[ ]: visualize_eye(df=df_eye, id='1', save=False)
```

2.3 Join ImageInfo DataFrame with eye DataFrame

So far we have created two different dataframes: - ImageInfo df: contains all Unity streams that send data into the system independent of eye tracking (e.g, 'ImageInfo', 'Visual') - Eye df: containing all Unity streams info that depend on eye-tracking (e.g., 'HitOnObjectNames') For this reason, both df have different length, the first one having the largest amount of datapoints a.k.a. rows. To be able to separate the images that were actually seen from those that were not, we join both DataFrames based on the timestamps. – We detect when the first eye-tracking timestamp matches a timestamp un the ImageInfo df, so we join them from then on.

```
[ ]: def merge_img_eye_streams_data(img_df, eye_df):
    # create a common timestamp column for each df from the time-related columns to␣
↪be merged
    img_df['merged_ts'] = img_df["time_stamps_ImageInfo"]
    #print(df_renamed2.columns)
    eye_df['merged_ts'] = eye_df["time_stamps_HitObjectNames"]
    # drop shared column
    eye_df2 = eye_df.drop('uid', axis=1)
    # merge both df based on the common column, specifying the tolerance will not␣
↪allow times to be to far apart
    merged_eye_img_df = pd.merge_asof(img_df, eye_df2, on='merged_ts', tolerance=0.
↪01)
```

```
    merged_eye_img_df

    ### CALCULATE VALID CENTERED FIXATIONS ###
    grouped_unique = merged_eye_img_df.
↪groupby(['blockNumber_ImageInfo','shiftID','imageName_ImageInfo','HON_HitObjectNames'])['
↪nunique().reset_index().rename(columns={'normalized_tstamps_ImageInfo':␣
↪'centered_fixations'})
    grouped_unique = grouped_unique[(grouped_unique['imageName_ImageInfo'] !=␣
↪'grayCanvas') & (grouped_unique['imageName_ImageInfo'] != 'fixationCross') &␣
↪(grouped_unique['imageName_ImageInfo'] != 'startMessage')&␣
↪(grouped_unique['imageName_ImageInfo'] != 'endMessage')]
    sumpergroup = grouped_unique.
↪groupby(['blockNumber_ImageInfo','shiftID','imageName_ImageInfo']).
↪sum('HON_HitObjectNames').reset_index().rename(columns={'centered_fixations':␣
↪'total_fixations'})
    merge = pd.merge(grouped_unique, sumpergroup,␣
↪on=['blockNumber_ImageInfo','shiftID','imageName_ImageInfo'], how='left')
    # valid fixation when the total hit on fixation collider > 70 %
    merge['valid_fixation'] = merge.apply(lambda x: True if x['centered_fixations']/
↪x['total_fixations'] > 0.7 else False, axis=1)
    merge = merge[merge['HON_HitObjectNames'] == 'FixationCollider']
    merge.drop('HON_HitObjectNames', axis=1, inplace=True)
    merge

    df_eye_img2 = pd.merge(merged_eye_img_df, merge,␣
↪on=['blockNumber_ImageInfo','shiftID','imageName_ImageInfo'], how='left')

    del eye_df2, merged_eye_img_df, grouped_unique, sumpergroup, merge
    return df_eye_img2
```

[ ]:
```
merged_img_eye_df = merge_img_eye_streams_data(df_renamed2,df_eye)
merged_img_eye_df
```

3 Creating Trigger Files

- For each initial time an image was shown, we want to keep the type of object it was (i.e., face, object, body) as a separate column.
- Additional triggers contain the rotation and distance the specific object was with respect to the player at the time the free-viewing walk took place.
- **Note:** We want the triggers only once to denote the initial time the image was shown.

[ ]:
```
def create_triggers(df):
    # save the names of the object, body or face shown in the image
    df['ob_names'] =  df.apply(lambda x: x["imageName_ImageInfo"].split(".")[5] if␣
↪len(x["imageName_ImageInfo"].split(".")) > 7 else '', axis=1)
    # create the triggers to mark the start of image displaying
    # check when there is a change from image, canvas, fixationCross
    df['shift'] = df['imageName_ImageInfo'].shift(1) != df['imageName_ImageInfo']
    # save the starting time (aka 'latency') when image is displayed
    df['latency'] = df.apply(lambda x: x['normalized_tstamps_ImageInfo'] if␣
↪len(x["imageName_ImageInfo"].split(".")) > 7 and x['shift'] else '', axis=1)
    # save the type of image displaying (face, object, body)
    df['type'] = df.apply(lambda x: 'face' if x['shift'] and 'face' in␣
↪x['imageName_ImageInfo'].lower()
                                     else ('body' if x['shift'] and 'npc' in␣
↪x['imageName_ImageInfo'].lower()
```

```
                                                    else ('object' if x['shift'] and␣
↪'rotation' in x['imageName_ImageInfo'].lower()
                                                    and 'face|npc' not in␣
↪x['imageName_ImageInfo'].lower()
                                                    else '')), axis=1)
    # save if it is valid or invalid fixation (that is, if person was looking to␣
↪the center
    df['valid'] = df.apply(lambda x: 1 if x['valid_fixation'] == True else 0,␣
↪axis=1)
    # define the triggers for rotation, distance, and block
    df['rotation'] = df.apply(lambda x: x["imageName_ImageInfo"].split(".")[7] if␣
↪len(x["imageName_ImageInfo"].split(".")) > 7 and x['shift'] else '', axis=1)
    df['distance'] = df.apply(lambda x: x["imageName_ImageInfo"].split(".")[9] if␣
↪len(x["imageName_ImageInfo"].split(".")) > 7 and x['shift'] else '', axis=1)
    df['block'] = df.apply(lambda x: str(x["blockNumber_ImageInfo"]) if␣
↪len(x["imageName_ImageInfo"].split(".")) > 7 and x['shift'] else '', axis=1)
    # select the trigger columns and non empty rows
    df_sel = df[['latency','type','valid','rotation','distance','block']]
    df_triggers = df_sel[df_sel['latency'] != '']
    # save first uid for later usage
    uid = df['uid'][0]
    del df_sel
    return df, df_triggers, uid
```

[ ]:
```
df, df_triggers, uid = create_triggers(merged_img_eye_df)
df_triggers
```

[ ]:
```
df
```

[ ]:
```
def valid_images(df):
    img_unique = df[~df['imageName_ImageInfo'].str.
↪contains('fixation|grayCan|Message')]
    total_images = img_unique.
↪groupby('blockNumber_ImageInfo')['imageName_ImageInfo'].nunique().reset_index().
↪rename(columns={"blockNumber_ImageInfo": "block_number", "imageName_ImageInfo":␣
↪"total_img"})

    # convert column to numeric to drop the duplicates correctly
    total_images['block_number'] = pd.to_numeric(total_images['block_number'])

    img_validity_array = img_unique[img_unique['shift']].
↪groupby(['blockNumber_ImageInfo','valid'])['time_stamps_ImageInfo'].nunique().
↪reset_index()
    # img_validity_array
    valid = img_validity_array[img_validity_array['valid'] ==␣
↪1][['blockNumber_ImageInfo', 'time_stamps_ImageInfo']].
↪rename(columns={'blockNumber_ImageInfo': 'block_number', 'time_stamps_ImageInfo':
↪ 'valid'}).astype('int64')
    invalid = img_validity_array[img_validity_array['valid'] ==␣
↪0][['blockNumber_ImageInfo', 'time_stamps_ImageInfo']].
↪rename(columns={'blockNumber_ImageInfo': 'block_number', 'time_stamps_ImageInfo':
↪ 'invalid'}).astype('int64')
    total_images = pd.merge(total_images, valid, how='left', on=['block_number'])
    total_images = pd.merge(total_images, invalid, how='left', on=['block_number'])
    del img_validity_array, valid, invalid

    return total_images
```

```
[ ]: valid_images(df)
```

4 Complete visualization

Let's visualise Unity and EEG streams in one single plot

```
[ ]: def visualize_unity_eeg_all(unity_df, eeg_df,time_fragment='', id='',␣
     ↪display_start=0,display_end=16, vis_end=False, shifts_from_end=16, save=True):
         # when visualizing the end of the recording
         if vis_end:
             # find the last shift in the dataframe
             display_end = unity_df['shiftID'].iloc[-1]
             # calculate amount of shift to display from the end
             display_start = display_end - shifts_from_end
             # keep only data in dataframe that is between desired start and end to be␣
     ↪visualized
             unity_df_sel = unity_df[unity_df['shiftID'].
     ↪between(display_start,display_end)]
             # replace ending display nummber '99' for -0.5 to allow plot in y-axis
             unity_df2 = unity_df_sel.replace({'displayStatus_Visual':99},-1)
         else:
             # select the first 16 shifts in the data
             unity_df2 = unity_df[unity_df['shiftID'].between(display_start,display_end)]

         start_time = unity_df2['normalized_tstamps_ImageInfo'].min()
         end_time = unity_df2['normalized_tstamps_ImageInfo'].max()

         # set the color palette
         # colors = dict(startMessage=deep_pal[6], fixCross=deep_pal[4],␣
     ↪image=deep_pal[0], grayCanvas=deep_pal[1])

         fig = plt.figure(figsize=(26, 22))
         plt.rcParams.update({'font.size': 18})
         if vis_end:
             fig.suptitle("Unity and EEG streams at end of recording time. ID: " + id,␣
     ↪fontsize=32, y=0.94, fontweight='bold')
         else:
             if display_start <= 0:
                 fig.suptitle("Unity and EEG streams at start of recording time. ID: " +␣
     ↪id, fontsize=32, y=0.94, fontweight='bold')
             else:
                 fig.suptitle("Unity and EEG streams through recording time. ID: " + id,␣
     ↪fontsize=32, y=0.94, fontweight='bold')
         # make outer gridspec of 3 rows and 1 column
         outer = fig.add_gridspec(3, 1, hspace = .2, height_ratios = [8, 4, 16])

         # make nested gridspecs
         inner_grid1 = outer[0, 0].subgridspec(3, 1, hspace=0.05)
         inner_grid2 = fig.add_subplot(outer[1, 0])
         inner_grid3 = outer[2, 0].subgridspec(10, 1, hspace=0)

         ### Visualize EYE data ###
```

```
    eye_selected =␣
↪unity_df[["normalized_tstamps_EyeTrackingWorld","HTdirectionX_HeadTracking",␣
↪"HTdirectionY_HeadTracking","HTdirectionZ_HeadTracking","ETWdirectionX_EyeTrackingWorld",
↪"ETWdirectionY_EyeTrackingWorld","ETWdirectionZ_EyeTrackingWorld"]].
↪rename(columns={"HTdirectionX_HeadTracking": "HTdirectionX",␣
↪"HTdirectionY_HeadTracking": "HTdirectionY","HTdirectionZ_HeadTracking":
↪"HTdirectionZ","ETWdirectionX_EyeTrackingWorld": "ETdirectionX",␣
↪"ETWdirectionY_EyeTrackingWorld":␣
↪"ETdirectionY","ETWdirectionZ_EyeTrackingWorld":"ETdirectionZ"})
    # covert the dataframe to be tidy (long and not wide format)
    df_eye_tidy = eye_selected.melt('normalized_tstamps_EyeTrackingWorld',␣
↪var_name="cols",value_name='vals')

    def eyeviz(axx, dimension):
        dim = sns.lineplot(x="normalized_tstamps_EyeTrackingWorld", y='vals',␣
↪hue='cols', data=df_eye_tidy[df_eye_tidy['cols'].str.contains(dimension)],␣
↪style='cols', ax=axx, linewidth=4)
        if dimension == 'X':
            axx.set_title("Eye-tracking. Sample: " +␣
↪str(round(end_time-start_time)) + ' s', fontsize=28)
        if dimension == 'Z':
            axx.set_xlabel('', fontsize=22)
            # axx.set_ylim(-1,1.1) # set y-axis limits if needed
        else:
            axx.set_xticks([])
            axx.set(xlabel=None)
        # axx.spines.top.set_visible(False) # set top, left, right lines visible
        axx.set_ylabel(dimension + " axis", fontsize=22)
        axx.set_xlim(start_time-0.013,end_time)
        ## change legends' lines size
        # obtain the handles and labels from the figure
        handles, labels = axx.get_legend_handles_labels()
        # copy the handles
        handles = [copy.copy(ha) for ha in handles ]
        # set the linewidths to the copies
        [ha.set_linewidth(4) for ha in handles ]
        # put the copies into the legend
        axx.legend(title='HT vs. ET', title_fontsize='20', handles=handles,␣
↪labels=labels,bbox_to_anchor=(1.01, 1), loc=2, borderaxespad=.05)

    # plot head vs. eye tracking axis in its own gridspecc
    for i, ax in enumerate(inner_grid1.subplots()):
        eyeviz(ax, chr(88 + i)) # Ord 88 = 'X', Ord 89 = 'Y' etc.

    def unityviz(axx):
        hue_lab = unity_df2['displayStatusNames'].unique().tolist()
        # choose the hue labels based on start or end plotting
        if 'startMessage' in hue_lab:
            hue_order = ['grayCanvas','fixCross','image', 'startMessage']
        else: hue_order =['grayCanvas','fixCross','image', 'endMessage']
        # visualize displayStatuses
        sns.scatterplot(x=unity_df2['normalized_tstamps_ImageInfo'],␣
↪y=unity_df2['displayStatus_Visual'], data=unity_df2, hue='displayStatusNames',␣
↪palette=colors, s=110, hue_order=hue_order,ax=axx)
        axx.set_ylim(-1.2, 2.2)
        axx.set_xlim(start_time - 0.013,end_time)
        axx.set_yticks([-1, 0, 1, 2])
```

```
        axx.locator_params(axis='both', nbins=33)
        axx.legend(title='Display status',title_fontsize='22',bbox_to_anchor=(1.01, 1), loc=2, borderaxespad=.05)
        #axx.legend(bbox_to_anchor=(1, 1), loc=2, borderaxespad=.0)
        axx.set_title("Unity markers. Sample: " + str(round(end_time - start_time)) + ' s', fontsize=28)
        axx.set_xlabel('', fontsize=16)
        axx.set_ylabel('Unity markers', fontsize=22)

    # visualize Unity displays statuses
    unityviz(inner_grid2)

    # normalize eeg time_stamps to start from zero
    eeg_df['timestamps'] = eeg_df['timestamps'] - eeg_df['timestamps'][0]
    eeg_df = eeg_df[eeg_df['timestamps'].between(start_time,end_time)]
    # select latencies to set stimulus onset
    labels = unity_df2[unity_df2['type'] !='']
    list_labels = labels['type'].values.tolist()
    # select valid gazes >70%
    valid_lab = labels['valid'].replace([1,0],['valid','invalid']).values.tolist()

    latencies = unity_df2[unity_df2['latency'] !='']
    list_lat = latencies['latency'].values.tolist()

    def eegviz(axx, i, first = False, last = False):
        g = sns.lineplot(x= eeg_df['timestamps'], y = eeg_df[eeg_df.columns[i]],ax=axx, linewidth=0.1)
        axx.set(yticklabels=[])
        axx.set_xlim(eeg_df['timestamps'].min(),eeg_df['timestamps'].max())
        axx.spines.top.set_visible(False)
        axx.set_ylabel('Ch'+str(i+1), fontsize=20)
        for lat_i, lat_value in enumerate(list_lat):
            # create a vertical line with diff. color per category indicating the stimulus onset
            if list_labels[lat_i] == 'face':
                g.vlines(x=lat_value, ymin=eeg_df[eeg_df.columns[i]].min(),ymax=eeg_df[eeg_df.columns[i]].max(), colors="#D0465A",label=list_labels[lat_i], ls='solid', lw=4)
            if list_labels[lat_i] == 'body':
                g.vlines(x=lat_value, ymin=eeg_df[eeg_df.columns[i]].min(),ymax=eeg_df[eeg_df.columns[i]].max(), colors="#C99756",label=list_labels[lat_i], ls='dashed', lw=4)
            if list_labels[lat_i] == 'object':
                g.vlines(x=lat_value, ymin=eeg_df[eeg_df.columns[i]].min(),ymax=eeg_df[eeg_df.columns[i]].max(), colors="#046378",label=list_labels[lat_i], ls='dotted', lw=4)

        if last:
            axx.set_xlabel('Time (s)', fontsize=24)
        else:
            if first:
                # annotate valid/invalid triggers
                for lat_i, lat_value in enumerate(list_lat):
                    y_loc = round(eeg_df[eeg_df.columns[i]].min() + eeg_df[eeg_df.columns[i]].max()) / 2
                    axx.text(lat_value,y_loc, str(valid_lab[lat_i]), bbox=dict(boxstyle="round", edgecolor='black', fc="0.94"))
```

```
                # include legend
                axx.legend(title='Category',title_fontsize='22',bbox_to_anchor=(1.
↪01, 1), loc=2, borderaxespad=.05)
                axx.set_title("EEG with triggers for first 10 channels. Sample: " +␣
↪str(round(end_time-start_time)) + ' s', fontsize=28)
                axx.spines.top.set_visible(True)
                # do not repeat label names in legend
                handles, labels = axx.get_legend_handles_labels()
                by_label = OrderedDict(zip(labels, handles))
                axx.legend(by_label.values(), by_label.
↪keys(),title='Category',fontsize='20',title_fontsize='22', bbox_to_anchor=(1.01,␣
↪1), loc=2, borderaxespad=.05)
            axx.set_xticks([])
            axx.set_xlabel('', fontsize=22)
            axx.spines.bottom.set_visible(False)

    # plot every column from eeg df into a single gridspec
    for i, ax in enumerate(inner_grid3.subplots()):
        eegviz(ax, i, i == 0, i == len(eeg_df.columns) - 2)
    # save image
    if save and vis_end:
        plt.savefig("data/images/" + "displayStatusNEW_end_from_"+␣
↪str(round(start_time)) +"s_to_"+ str(round(end_time)) + 's_'+ id + ".pdf",␣
↪format='pdf',bbox_inches='tight', dpi=1200)
    else:
        if save and display_start <= 0:
            plt.savefig("data/images/" + "displayStatusNEW_start_from_"+␣
↪str(round(start_time)) +"s_to_"+ str(round(end_time)) + 's_'+ id + ".pdf",␣
↪format='pdf',bbox_inches='tight', dpi=1200)
        else:
            plt.savefig("data/images/" + "displayStatusNEW_through_time_from_"+␣
↪str(round(start_time)) +"s_to_"+ str(round(end_time)) + 's_'+ id + ".pdf",␣
↪format='pdf',bbox_inches='tight', dpi=1200)

    del unity_df2, latencies, labels, list_labels, valid_lab
```

[ ]:
```
visualize_unity_eeg_all(df, df_eeg_all, id=uid, display_start=0, display_end=16,␣
↪vis_end=False, shifts_from_end=15, save=False)
```

[ ]:
```
import gc
gc.collect()
```

5 General pipeline

Here we extract all streams info for all recordings. - Extract streams info - Create streams, eye, and eeg graphics for every recording - Create trigger files per recording - Save data in respective folders as .csv and .pdf

[ ]:
```
import gc
gc.collect()

# sort data files alphabetically
files_s = os.listdir(r_path)
files_s.sort()
# path to save .csv with total images per block
total_img_file = os.path.join(t_path, 'total_unique_images_per_user.csv')
desc_file_streams = os.path.join(t_path, 'desc_files_streams.csv')
```

```python
# progress bar format definitons
m_format = "{desc}:{bar}{percentage:3.0f}% {n_fmt}/{total_fmt} in {elapsed_s:.2f}s"
s_format = ("{desc}:{bar}{percentage:3.0f}% {n_fmt}/{total_fmt}{postfix} in {elapsed_s:.2f}s")
# main progress bar
main_bar = tqdm(
    files_s,
    #os.listdir(r_path),
    desc="Processed",
    dynamic_ncols=True,
    mininterval=0.001,
    bar_format=m_format,
)
# for k in main_bar:
for file in main_bar:
    if file.lower().endswith('.xdf'):
        # Skip if not a first part of a recording
        if 'room2' in file:
            continue
        pbar = tqdm(
        range(7),
        mininterval=0.001,
        maxinterval=1,
        bar_format=s_format,)

        pbar.set_postfix(file=file)
        #### 1. Load the XDF file ####
        postfix = {"step": "1. Load the XDF file", "file": file}
        # set flag if a second recording for same subject
        file2 = file.replace('room1', 'room2')
        second_rec = True if os.path.isfile(os.path.join(r_path, file2)) else False

        pbar.set_postfix(postfix)
        streams, _ = pyxdf.load_xdf(os.path.join(r_path, file))
        # read streams from second file if it exists
        if second_rec:
            streams2, _ = pyxdf.load_xdf(os.path.join(r_path, file2))

        pbar.update(1)
        #### 2. Store selected stream info (only useful info)
        postfix = {"step": "2. Store selected streams info", "file": file}
        pbar.set_postfix(postfix)
        # IMG_INFO: store trigger-related stream data into df and use the EEG stream for first timestamp reference
        df_img_inf = get_streams_data(streams, streams_keep=['ImageInfo','Visual'], time_ref_stream='openvibeSignal', use_startdiff_correction=True,use_manual_drift=False)
        if second_rec:
            # store the df for second recording if subject has to recordings
            df_img_inf2 = get_streams_data(streams2, streams_keep=['ImageInfo','Visual'], time_ref_stream='openvibeSignal', use_startdiff_correction=True,use_manual_drift=False)
            # combine both df into one
            df_img_inf = pd.concat([df_img_inf,df_img_inf2], ignore_index=True)

        # drop nan values
```

```
        df_img_inf = df_img_inf.dropna().reset_index(drop=True)
        df_img_info_renamed = rename_displays(df_img_inf)

        # EYE_DATA: save eye_tracking related data
        df_eye = get_streams_data(streams, streams_keep=['HitObjectNames',␣
↪'HitPositionOnObjects','HeadTracking','EyeTrackingWorld','EyeTrackingLocal'],␣
↪time_ref_stream='openvibeSignal',␣
↪use_startdiff_correction=True,use_manual_drift=False)
        if second_rec:
            # save eye df for second recording if subject has to recordings
            df_eye2 = get_streams_data(streams2, streams_keep=['HitObjectNames',␣
↪'HitPositionOnObjects','HeadTracking','EyeTrackingWorld','EyeTrackingLocal'],␣
↪time_ref_stream='openvibeSignal',␣
↪use_startdiff_correction=True,use_manual_drift=False)
            # combine both df into one
            df_eye = pd.concat([df_eye,df_eye2], ignore_index=True)
        pbar.update(1)

        ### 2.1 merge Unity imageInfo and Eye dataframes
        # We want to have only one df with Unity img_info and eye tracking data
        merged_img_eye_df = merge_img_eye_streams_data(df_img_info_renamed, df_eye)

        #### 3. Create triggers ####
        postfix = {"step": "3. Creating triggers from ImageInfo", "file": file}
        pbar.set_postfix(postfix)
        df_merged, df_triggers, uid = create_triggers(merged_img_eye_df)  # df,␣
↪df_triggers, uid = create_triggers(merged_img_eye_df)
        pbar.update(1)

        #### 4. Create and save visualizations ####
        postfix = {"step": "4.  Creating and saving visualizations", "file": file}
        pbar.set_postfix(postfix)
        # get some seconds of the eeg data from streams, not all
        _, df_eeg_all = stract_eeg_data(streams)
        # take the first 30 seconds of recording (30s x 90fps) and rename names in␣
↪'displayStatusNames' column
        ###### df_renamed = rename_displays(df)

        # visualize the first 30 seconds of recording
        visualize_unity_eeg_all(df_merged, df_eeg_all, time_fragment='start',␣
↪id=uid, display_start=0, display_end=16, vis_end=False, save=True)
        plt.close()
        # visualize 5 shifts (display statuses) from shift 100 to 105
        visualize_unity_eeg_all(df_merged, df_eeg_all, time_fragment='', id=uid,␣
↪display_start=100, display_end=105, vis_end=False, save=True)
        plt.close()
        visualize_unity_eeg_all(df_merged, df_eeg_all, time_fragment='end', id=uid,␣
↪vis_end=True, shifts_from_end=15, save=True)
        plt.close()
        pbar.update(1)

        #### 5. Save total number of images per block, per uid  ####
        postfix = {"step": "5 Saving total number of images per block, per uid",␣
↪"file": file}
        pbar.set_postfix(postfix)
        # convert column to numeric to drop the duplicates correctly
        #total_images['block_number'] = pd.to_numeric(total_images['block_number'])
```

```python
        total_images = valid_images(df_merged)

        # save total images for all participants
        if not os.path.isdir(t_path):
            os.mkdir(t_path)
        if not os.path.isdir(e_path):
            os.mkdir(e_path)
        desc_streams = describe_recordings(streams, file_name=file)

        #### 6. Create streams description per recording  ####
        postfix = {"step": "6. Creating triggers from ImageInfo", "file": file}
        pbar.set_postfix(postfix)
        if os.path.exists(desc_file_streams):
            df_desc_files = pd.read_csv(desc_file_streams)
            df_desc_files = pd.concat([df_desc_files,desc_streams],␣
↪ignore_index=True)
            df_desc_files.reset_index(drop=True, inplace=True)
            df_desc_files.drop_duplicates(inplace=True, ignore_index=True)
        else:
            df_desc_files = desc_streams
        df_desc_files.to_csv(desc_file_streams, index=False)
        pbar.update(1)

        total_images['uid'] = uid # add colum to total images csv with uid
        if os.path.exists(total_img_file):
            df_total_unique_img = pd.read_csv(total_img_file)
            df_total_unique_img = pd.concat([df_total_unique_img,total_images],␣
↪ignore_index=True)
            df_total_unique_img.reset_index(drop=True, inplace=True)
            df_total_unique_img.drop_duplicates(inplace=True, ignore_index=True)

        else:
            df_total_unique_img = total_images
        df_total_unique_img.to_csv(total_img_file, index=False)
        pbar.update(1)
        #### 7. Saving triggers, ET, and full_streams_csv files ####
        postfix = {"step": "7. Saving triggers and et files", "file": file}
        pbar.set_postfix(postfix)
        df_triggers.to_csv(os.path.join(t_path, 'trigger_file_' + uid +'.csv'),␣
↪index=False)
        df_eye.to_csv(os.path.join(e_path, 'et_' + uid + '.csv'), index=False)
        merged_img_eye_df.to_csv(os.path.join(all_csv_path, 'all_streams_' + uid +␣
↪'.csv'), index=False) # make sure the folder exists
        pbar.update(1)
        pbar.set_postfix(file=file)
        pbar.close()
```