GEARS User Manual

László Szécsi, Ágota Kacsó, Günther Zeck, and Péter Hantz

March 31, 2017

Chapter 1

Quick and dirty user's guide

1.1 Installation

Choose a computer with at least a medium-power nVidia graphics card. It must support OpenGL 3.4. A core count of 500 with a base clock of 1000 MHz, or equivalent, is recommended for usual video processing and filtering tasks at 60Hz. Stimuli computing an 2D image from a procedurally defined 3D geometry demand even more power.

Download the GearsSetup.exe file from www.gears.vision (Download and Installation link). Run the .exe file. During installation, you can choose the directory where the program should be placed.

Launch by starting Gears.exe. The shortcut icon should appear in the Start menu and on the desktop as well. A black-background screen with red text and symbols will emerge.

1.2 Configuring GEARS

Right click the topmost item (Sequences") in the upper left box. A context menu with two items appears. Click Configure". A dialog window with several tabs in a horizontal row on top appears.

1.2.1 Display tab

This is for specifying general parameters of the display device and the size of the illuminated area, which depend on the optical system—for example, retina stimulation is usually implemented by projecting through a microscope objective. All values set by this point are written in the AppData.py file, which is located at ¡installation folder¿/Project/Sequences".

- Stimulus monitor Usually, besides your first display device, a second display device, like a projector, is also connected to your computer. You can set up the devices to duplicate the same image in your operating or windowing system. However, you can also extend your desktop, displaying different parts of it on different devices. In this latter case, GEARS offers two options. First, you can display the experiment stimulus on one device only, while other one serves for displaying stimulus settings. Second, you may use multi-monitor mode to tile the stimulus over two devices.

  The Stimulus monitor" setting lets you specify which device should be the primary stimulus display device when you only use one of your devices for stimulus display. It is possible to

change the device or switch to multi-monitor mode during stimulus display, by pressing the TAB key.

- Refresh rate To properly adjust this setting, consult the manual of your stimulus display device (e.g. the projector). Most of them accept 60Hz, but some peak models allow even 500 Hz. This value is set from the driver of the display device, and automatically matched between the video card and the display device. The synchronization of video card frames with displayed images (VSYNC) should be enables in the video card device settings.

  This is not the only issue you have to consider: High refresh rates demand high graphics card computational power. Your graphics card should have enough capacity to compute the consecutive frames at the chosen refresh rate, otherwise frame drops might occur. Note that the software warns you if frame dropping emerges.

- Illuminated field physical width and height (in millimeters) Here you can set set, in micrometers, the physical size of the illuminated area or the screen. These values will have importance in setting the physical sizes of shapes in stimuli, as well as the physical speed of moving stimuli.

- Force square field For certain tasks, it is convenient to set a strictly squared illumination field. By default, this is set to False.

- Field vertical and horizontal resolution Here you can define the width and height of your Stimulus monitor in pixels. If the image is projected in a microscope, display resolutions higher than 1024x768 are seldom demanded, because the physical resolution is limited by the microscope objective. Note that higher display resolution demand higher graphics card computational power.

- Field left and top margin These serves for letting a black margin around the illuminated (patterned) area. For example, a small square checkerboard stimulus can be located at different positions within the field. Normally you do not have to modify these values, just leave them on zero.

1.2.2 Ports tab

Here one can set the virtual serial communication (COM) ports for 4 pivotal signals which are provided by GEARS and should be recorded and stored, along with the physiological data, by the recording electronics. Some of these signals enable, during data analysis, the correlation between given stimuli and the physiological response.

If you click the setting, a dropdown with automatically detected COM ports will appear. One USB port can implement two COM ports by an appropriate simple hardware [1]. Every COM port can emit signals on two pins, called BREAK and RTS.

Note that most projectors implement the pictures with a constant (usually about 80 ms) delay (see the Supplementary Material of the Frontiers of Neuroscience paper presenting GEARS). This value has to be measured and considered in the data analysis software.

- Stimulus sync switches to TTL high (5V) at determining moments of the stimulus like the emergence of a pattern, and back to low when it vanishes. Switching points can be set in the Python stimulus sequence scripts (see later).

[1] http://gears.vision/PDF/USB

- Measurement stop switches briefly to TTL high (5V) at the end of the experiment (also defined in the Python stimulus sequence script) and can be used to command certain hardware, like MChS MEA, to stop recording.
- Per frame spikes provide brief (less than 5 ms) TTL high (5V) pulses when a new frame is sent to the Stimulus screen device (e.g. the projector).
- Experiment sync is a redundant signal which can be set to switch to TTL high (5V) in a certain period of the stimulus. It is advised to be switched to TTL high (5V) after the uniform (usually black or gray) starting frames in the stimulus, which are introduced for accomodation purpose.

1.2.3 FFT tab

This tab is designated for setting the resolution for computing Fast Fourier Transforms.

1.2.4 Electrode grid tab

For electrophysiology recording with multielectrode arrays (MEA), is often a demand to center certain stimuli on a given electrode. Therefore, electrode coordinates have to be be accessible by the software, and electrodes are referred in a straightforward manner (columns by letters AB, B, C, ... and rows by numbers 1, 2, 3, ..., so for example, an electrode can be referred to as C12). Special stimuli are provided (0, Utility stimulus set) to center the illuminated area on the middle of the multielectrode array.

The parameters are designed to specify the region filled with electrodes as the union of two square areas. As electrode lattices, square grids can be specified. The necessary parameters are depicted in figure **??**.

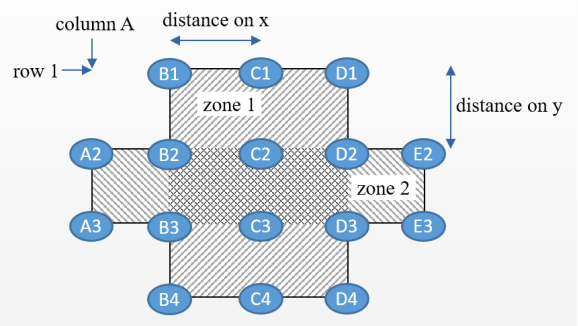

Figure 1.1: Parameters specifying electrode grid geometry. The top row, bottom row, leftmost column, and rightmost column of zone 1 are set to 2, 3, A, and E, respectively. The settings for zone 2 are, in the same order, 1, 4, B, and D. The locations of row 1 and column A, as well as the electrode distances along the horizontal and vertical axes are given in physical units in real space.

1.3 Hierarchy of configuration files

The configuration set by selecting Configure" from the context menu of the topmost item Sequences" applies globally to all stimulus sequences by default. However, right clicking individual folders

containing stimulus sequence scripts also brings up the configuration dialog discussed above. This allows editing config.py" configuration files in the respective folders. These override the global settings for the sequences located in the folder and its subfolders, recursively.

In the configuration dialog, the default values taken from the higher order configuration are always indicated. If a setting is overridden, both the previous and the new value is displayed in a warning message (e.g. 30 Hz overriding 60 Hz specified in AppData.py").

Factory defaults are set in the AppData.py script.

1.4 Stimulus Sequence Scripting

In order to modify existing, or write new Stimulus Sequence Scripts, go to the box with the title Sequences" on the starting screen of GEARS.

By clicking on any of the folders like Spots, FullFields, MovingShapes, etc., a column of subfolders will appear. Clicking on a subfolder, a list of Stimulus Sequence Scripts, files encoding a light stimulus sequence will emerge. These files have a .pyx extension. They are located in the AppData folder of your computer. You can browse this location by right clicking a folder or Stimulus Sequence Script, and choosing menu option Explore in folder".

Creating folders and subfolders is possible using file system tools like Windows Explorer. Folders containing Stimulus Sequence Scripts are automatically displayed in the Sequences" box on the starting screen, when the program is started. New stimulus sequence scripts can most easily be created by copying and renaming existing ones, or with any text editor. Note that folders are only displayed if they contain a .pyx file.

Left-clicking on a .pyx file drives to the Stimulus Overview Screen, where the synopsis of the stimulus and the GEARS settings are presented.

Right-clicking on a .pyx file, two options are offered. Open sequence" does the same as a left click, opening the Stimulus Overview Screen. Edit script" drives to the Script Editor Window.

This contains a Script Editor and a Stimulus Movie subwindow. The Script Editor contains the Python code of the selected stimulus sequence. In order to remind the user on the parameters of individual stimuli and their meaning, by left-clicking on the parameter list of any stimulus or SBC, a popup window jumps up, which presents this information. The Stimulus Movie subwindow shows what the selected stimulus looks like. The continuously running movie can be stopped and played from an arbitraty stage by the actual and by an accelerated speed. Modifications can be saved under the same or under a different name by clicking on the Save bar above the Stimulus Movie.

1.4.1 Handling color spaces

Many stimuli and SBCs take colors as parameters. These can be specified in various ways:

- A single scalar indicates a greyscale intensity, where 0 is black, while 1 is white.
- A triplet of values (e.g. `(0.3, 0.5, 1.0)`) is interpreted as RGB color coordinates.
- String color names (e.g. `'red'`) are aliases for basic RGB colors.
- Colors can be specified in other widely used color spaces (e.g. `HLS(0.5, 1.0, 0.5)`).
- Custom color matching functions can also be used (e.g. `MyColor(red=0.5, rest='dontcare')`, where MyColor is a custom matching explained below).

Channels of different display devices may emit colors having different spectra. Therefore, the very same color coordinates could produce physically different colors while using different display devices. In order to overcome this problem, emission spectra of different color channels can be recorded. The spectral sensitivity curves of the biological light sensors should also be provided. From these, GEARS assembles a transformation matrix, having as elements the overlaps between the respective emission and sensitity spectra.

The user may desire to stimulate a certain biological light sensor, or multiple sensors.

Let us assume first that we wish to stimulate a certain sensor. In this case, an adequate combination of channel intensities is set up. Using a single channel may be sufficient occasionally, but, as dynamic ranges of devices are limited for every channel, it is likely that multiple channels have to be combined to get the desired intensity for the sensor in question. There are usually multiple combinations of channel intensities that achieve the same goal. GEARS will select the one with minimum norm, i.e. the one using the least overall power.

The case when the user intends to stimulate two or several sensors is similar, if the problem remain underdetermined. When there are more sensors than display channels, in particular, the problem can be overdetermined, and an exact solution may not exist. GEARS computes a solution with minimum error in the least squares sense.

Let us assume that this is implemented with some kind of ordinary stimulus, e.g. a fullfield.

When a color parameter is expected, he can specify the desired sensor stimulations to the light sensors. Using the above matrix, GEARS computes a minimum-norm solution to set the color channel intensities to match (or, if that is not possible, approximate) the desired spectrum.

One example is just compensation of the display device differences to achive the same color sensation, when there are both three color channels and three types of biological sensors. In this case, an exact solution for color sensations within the capabilites of the display device are possible.

Another example would be the stimulation of organisms with a higher number of bilogical light sensors. In this case, a least power approximation is calculated by GEARS.

1.4.2 How to drive external auxiliary hardware by GEARS

Chapter 2

A second look

GEARS is a hybrid C++/Python solution. The C++ Python extension `Gears.pyd` is responsible for rendering and GPU resource management. This includes rendering various plots, tone mapping, and filterings. The Python application `GearsPy.py` works closely with the extension providing it with a user interface, event management, and, most importantly, scripting for stimulus and stimulus sequence design.

GEARS is an experiment design framework meant for power users who are able to compose their experiments programmatically. There are several application layers corresponding to different levels of programming expertise that allow for varied levels of freedom in extending the framework. We call these user roles the following:

Experiment conductor is able to launch predefined stimulus sequences through the graphical user interface.

Stimulus sequence assembler is able to edit simple Python code that defines experiments as lists of predefined stimuli, with the help of a visually aided script editor. He is also able to configure measurement device parameters like display gamma curve, field size on the retina, etc.

Stimulus composer is able to write simple Python code that defines new stimuli using preexisiting components for random number generation, particle systems, shape, pattern, motion, intensity modulation, spatial and temporal filterings, gamma compensation, and sync signals.

Component programmer is able to define new behaviors in the above aspects that can be used in composing stimuli. This requires Python and GLSL programming on the level of being able to formulate mathematical definitions as simple GLSL functions.

Component system architect is able to define a whole new system of components extending on or superseeding the existing one. This requires some understanding of how GLSL shaders work.

GUI developer is an expert Python programmer who can reprogram the open source Python components to display information differently or respond to new user events. Any such changes are under the GPL licence — as opposed to the above listed scripting activities, the results of which need not be open source.

GPU developer is an expert in C++, OpenGL, and GPU programming capable of changing or enhancing the code for the C++ extension. Any such changes are under the GPL licence.

In the following, we describe and illustrate the operation of the application level-by-level.

Chapter 3

Performing experiments

3.1 The sequence browser

Once the program has been launched, the *stimulus sequence browser window* appears. The available stimulus sequences can be navigated and selected from a tree view. The hierarchy corresponds to the folder hierarchy within the `Project/Sequences` folder. Python script files with the extension `.pyx` are listed (Section 4.1 provides a more detailed discussion of these sequence script files). The mouse and cursor keys can be used to navigate and select a sequence. Additionaly, it is encouraged that all sequence and folder names start with a unique digit (or character). The digits (or characters) along the sequence path provide a unique numerical (or alphabetical) ID for the sequence. The number keys (or any character keys) can be used to select subitems in the tree. Thus, entering the numerical (or alphabetical) ID of the sequence instantly selects the sequence. You can return to the root of the tree by pressing space, backspace, or del.

Depending on how some special sequences (e.g. ones using the Till Photonics Polychrome device) are implemented, they may attempt to connect to external instruments as soon as the sequence is selected in the sequence browser. Make sure you have those devices already turned on.

The script defining the sequence may have errors. When this happens, either a Python error message is displayed (for syntax problems), or a higher level error message is shown, pinpointing the erroneous parameters.

3.2 Measurement device configuration

Right-clicking any item in the sequence browser brings up a context menu. The `Explore folder` or `Explore in folder` menu options open up Windows Explorer, where script files can be cloned, renamed, or deleted easily. Sequence script files offer the `Edit script` option that opens them in the *visually aided script editor* (described in detail in section **??**). Folders, on the other hand, offer the option `Configure`. This opens up a dialog window with settings that apply to all sequences in the folder. Subfolders inherit the settings of their parent folders, but local configurations can override them. Thus, every setting may use the inherited default (with the value and its source displayed in the configuration dialog), or assume a new value (with the new value displayed alongside the overridden value and its source). Settings are saved as `config.py` Python script files in the corresponding folders. Whenever a sequence is opened, all `config.py` scripts found along its folder path are executed, beginning from the root folder `Project/Sequences`.

3.2.1 Display configuration

The `Display` tab lists configuration options for the projector system. The *field* is the area illuminated on the retina. The most important settings are the logical resolution of the field given in pixels, and its physical size on the retina, given in micrometers. For applications other than retina stimulation, a physical field size must still be given. All locations, sizes, or velocities specified in script are primarily expected in physical units (although components may offer other options, or new components are at liberty to deviate from this scheme). The logical resolution is automatically set to the detected display device resolution. You may want to override this if only a part of the display device is used for stimulus projection, e.g. when a square field is used. The `Force square field` checkbox is an easy way to make the program ignore the greater logical dimensions of the field and use the lesser for both dimensions.

The monitor to be used for projection on multi-monitor systems can be chosen for a drop-down list of detected monitors. This is the monitor on which the stimulus appears by default, but by pressing `tab` while executing a stimulus, the stimulus window can be moved to other monitors or extended to cover multiple monitors.

The physical refresh rate of the display device can also be given. Although this setting is auto-detected, driver-reported values are not supposed to be accurate—they are just approximations intended for the identification of possible display modes. E.g. 60 Hz of 59 Hz is usually reported when the exact physical refresh rate is the standard 59.94 Hz. Therefore, this setting must be configured manually—typically in `Project/Sequences/config.py` to affect all sequences. The physical refresh rate is used everywhere in the program to convert between durations given in seconds and the number of corresponding frames.

3.2.2 Signal port configuration

The `Ports` tab allows selection of RTS or BREAK pins of available COM ports for signal output. These ports are typically implemented in hardware using USB to RS232 converters with BNC connectors on the RS232 side, limiting outgoing voltage to TTL levels.

3.2.3 Fast Fourier Transformation dimensions

For frequency space processing, image frames and filter kernels are transformed using FFT (Fast Fourier Transformation). The resolution of the transformed image influences image quality: low resolution can cause loss of high-frequency detail and ringing artifacts. Higher resolution, on the other hand, requires more processing power, which may impede real-time performance and cause frame drops. Therefore, FFT resolution should be selected with respect to the requirements of the application and the available processing power. Note that, as another option, the frame refresh interval may also be increased to allow for more processing time between displayed frames.

3.2.4 Electrode grid configuration

Some stimuli and other components rely on information about the configuration of the MEA (multi-electrode array) used to measure retina response. Shapes may be positioned over specific electrodes, or all of them. For this, the MEA geometry can be specified as a union of two regular rectangular grids. The grid constant and the positions of row 1 and column A must be given in physical units on the field, defining an infinite regular grid. Then, two zones (possibly overlapping) of $n \times m$ electrodes can be defined. This is enough to model MEAs with a regular electrode arrangement, with the corner electrodes possibly missing.

3.3 The launcher window

After clicking on a sequence, the *launcher window* appears, which allows you to review sequence properties and the timeline of stimuli. Editing, by design, is performed through editing the Python scripts, described later in this document. The sequence timeline shows the stimulus intensity as a function of time, and the signals emitted on the digital channels for syncronization. The stimulus intensity corresponds to the actual base brightness displayed—for most stimuli. (This is, however, customizable: for example, when fading between two images, the weighting factor is plotted.) Zooming in the timeline is possible with the mouse wheel or the slider underneath. Panning is possible by dragging with the mouse or with the scrollbar underneath. Clicking a stimulus selects it, and details are displayed in the lower half of the screen. This includes the stimulus timeline, which operates similarly to the sequence timeline, but displays graphs for a single stimulus item. Where meaningful, further tabs appear concerning spatial filtering, temporal filtering, random number management, or tone mapping.

3.3.1 Spatial filtering

If the currently selected stimulus employs spatial filtering, the spatial filter kernel is displayed on the `Spatial` tab. There is a 2D intensity-coded plot and a cross-section profile plot. The minimum and maximum values appearing on the plot, as well as the plot domain can be adjusted using the spinboxes. These do not influence the actual stimulus in any way, they are just there to tune the plot if required.

3.3.2 Temporal filtering

If the currently selected stimulus employs temporal filtering, the temporal filter kernel is displayed on the `Temporal` tab.

3.3.3 Random number generation

If any stimulus of the current sequence employs pseudo-random numbers, these can be exported to a file. For every frame of the sequence where random numbers are used, this is a 2D array of random numbers in column-major order. The file will appear in the `Project/Randoms` folder, in a subfolder with the same subpath as the sequence. The file name contains the sequence name as well as the date of the export. The file is a simple text file intended for software that is used to analyze sequence results. Note that it is also possible to replicate the functionality of the pseudo-random number generator implemented in the stimulus, and generate the same random numbers procedurally. The file may or may not contain comments. Comments might be useful for human insight, no comments might help importing the data. The random numbers may be exported as integers, as normalized (in $[0, 1]$) floating point values (e.g. a greyscale random chessboard), or just zeros and ones (e.g. for a binary random chessboard). For colorful stimuli, it might be required to create multiple random numbers per 2D array element (e.g. 3 channels for an RGB random chessboard stimulus). These settings are inherent to specific sequences and in most cases, already set by the sequence script, but they can be overridden here if necessary.

Random numbers are exported by clicking the `Save randoms` button. This will launch the sequence at a possibly elevated frame rate, while retrieving the random numbers used and saving them to a file. Let the sequence run through to have all the randoms exported.

3.3.4 Histogram measurement and tone mapping

On the `Tone mapping` tab the stimulus tone mapping can be launched. This will run the selected stimulus without displaying the result, but displaying a histogram of pixel intensities instead. The minimum, maximum intensities, the mean and variance are measured and displayed. These values can be used to fine tune tone mapping (see 5.3) for the stimulus.

For a comprehensive description of all sequences bundled with the software, see Chapter **??**.

Chapter 4

Sequence assembly

4.1 Sequence scripts

Sequences are defined by .pyx files containing Python code. These are organized into a folder hierarchy under `Project/Sequences` for categorization and easier navigation when selecting them for execution. Thus, you create a new sequence by simply adding a new file to this hierarchy. Sequences are uniquely indentified by their complete path and filename relative to the `Project/Sequences` root folder, but in order to allow shorter unambiguous references to them, the convention is to number them. The single-digit numbers before the first underscores—in the folder names making up the relative path—read together make a unique sequence ID number. It is also advisable to avoid similarly named sequences in different folders. Note that it is possible to use any character allowed in file names instead of the numbers, resulting in non-numerical sequence codes.

4.1.1 Sequence script entry point

A `.pyx` file must contain a function with the header

```
def create():
```

that returns a Python object derived from class `DefaultSequence`. Class `DefaultSequence` is defined in `Project/Sequences/DefaultSequence.py`, and is derived from `Gears.Sequence`, where `Gears` is the C++ extension module. `DefaultSequence` sets sequence configuration defaults stored in `config.py` files along the sequence script path (described earlier in section 3.2), and offers helper functions for further configuration.

When you configure the software to your own measurement setup, including display device resolution, display device gamma curve, electrode array setup, and optical projection ratios, you should use the configuration UI available in the context menu of the sequence browser, as described in section 3.2.

4.1.2 Sequence script entry point

Thus, in the `.pyx` file defining a new sequence, the function `create` should return a `DefaultSequence`
.

Subclassing DefaultSequence

Creating subclasses of `DefaultSequence` is possible but not encouraged. This is because the motivation for subclassing would be creating custom sequence configurations, but this functionality is better served by the per-folder config scripts, editable via the UI. If you do create a subclass of `DefaultSequence`, and your `create` method returns an instance of it, the subclass definition must appear somewhere in the chain of parent folders of the `.pyx` file. `.py` scripts from these folders are automatically imported.)

A sequence without stimuli is useless. Thus, stimuli must be added, which is done with the `setAgenda` method of the sequence object. This returns the sequence object itself, thus it is fine to write:

```
def create():
  return DefaultSequence
      ('Description of the sequence')
      .setAgenda( agenda )
```

assuming `agenda` is a Python list of stimulus objects .

Stimulus objects must derive from `Gears.Stimulus`, but they typically derive from its more specialized and convenient subclasses

- `Project.Components.Stimulus.Generic`,
- `Project.Components.Stimulus.SinglePatch`, or
- `Project.Components.Stimulus.SingleShape`.

These elements of the component-based *Stimulus Building Component* system are discussed later in **??**. There are numerous stimulus types available in `Project.Components.Stimulus`. In order to use these, the `.pyx` file must import this Python package with:

```
from Project.Components import *
```

thus every `.pyx` file typically starts with this line. Using the stimuli `Stimulus.Blank` and `Stimulus.Fullfield`, a stimulus list for a simple sequence with 1 second of darkness followed by 2 seconds of brightness could look like this:

```
from Project.Components import *
def create():
  agenda = [
    Stimulus.Blank( duration_s = 1 ),
    Stimulus.Fullfield( duration_s = 2 ),
    ]
  return DefaultSequence('Sunrise')
      .setAgenda( agenda )
```

Of course Python programming can be used here, so repeating darkness/brightness three times could be done like:

```
from Project.Components import *
def create():
```

```
  # create empty list
  agenda = []
  for i in range(0,3) :
    # append to list
    agenda += [
      Stimulus.Blank( duration_s = 1 ),
      Stimulus.Fullfield( duration_s = 2 ),
      ]
  return DefaultSequence('Three days')
      .setAgenda( agenda )
```

For purposes of controlling the measurement, when the measurement should start and when it should end can be signalled through digital channels connected on COM ports. These can be configured as described in section 3.2.2. When assembling the sequence, we can insert signal events into the agenda list. In the following script, `StartMeasurement` raises the `sequence sync` singal, then `EndMeasurement` clears it, and does a spike on the `measurement stop` channel, appending some darkness to the sequence start and sequence end. Note that the `EndMeasurement` point is also where aborting the measurement would jump, so anything after `EndMeasurement` is still performed after the user presses `Esc`.

```
from Project.Components import *
def create():
  agenda = [
    Stimulus.Blank( duration_s = 1 ),
    StartMeasurement(),
    ]
  for i in range(0,3) :
    agenda += [
      Stimulus.Blank( duration_s = 1 ),
      Stimulus.Fullfield( duration_s = 2 ),
      ]
  agenda += [
      EndMeasurement(),
      Stimulus.Blank( duration_s = 1 ),
      ]
  return DefaultSequence('Full simple sequence')
      .setAgenda( agenda )
```

This is basically what a full sequence definition looks like. Typically, what varies is the number, type, and parametrization of stimuli. The duration is always a parameter, except when computed from other parameters like for shapes crossing the screen. The duration can be specified in seconds (parameter `duration_s`), but as a stimulus must last for an integer number of displayed frames, the actual duration may be slightly different. Parameter `duration` can be used to set the number of frames displayed precisely. Different stimuli accept different parameters, which are documented in the stimulus classes themselves, and displayed interactively while editing the scripts from within GEARS.

If a lot of stimuli share some parameter, then it may not be the best practice to write that parameter out for every stimulus instance. Instead, you could set the value of a local variable, and pass the variable as a parameter to the stimuli. However, for best readability, it is recommended

to derive your own subclass of the stimulus class. The `boot` method of this subclass should call the superclass' similar method with proper parametrization. For example, if you want a number of rectangles crossing the field with the same size and speed, but different directions, you can extend `Stimulus.CrossingRect` as

```
class FastDirectionSelectiveCrossing (Stimulus.CrossingRect) :
  def boot(self, direction='east'):
    super().boot(
      direction = direction,
      size_um = (1000, 500),
      velocity = 1200,
      )
```

and now `FastDirectionSelectiveCrossing` can be used in the stimulus agenda list.

If you ever want to change the velocity for all instances, you only have to modify it in class `FastDirectionSelectiveCrossing`.

4.2 Interactivity

Most stimulus parameters accept interactive controls (instances of `Interactive.MouseMotion`, `Interactive.MouseWheel`, `Interactive.MouseColor`, or other controls) instead of values. By design, these controls are not supposed to be visible, so that they do not interfere with the stimulus projection. Therefore, they do not have a graphical representation. Instead, values can be changed by moving the mouse or the mouse wheel while holding a key associated with the control. As an example, the color of the simple fullfield stimulus can be made interactive as follows:

```
from Project.Components import *
def create():
  agenda = [
    Stimulus.Blank( duration_s = 1 ),
    Stimulus.Fullfield(
        duration_s = 20,
        color = Interactive.MouseColor(
                        initialValue = 'white',
                        label = 'bright color',
                        key = 'W',
                )
        ),
    ]
  return DefaultSequence('Sunrise')
      .setAgenda( agenda )
```

Instead of specifying a concrete value, a control is created. While pressing key 'W', the user may scroll the mouse wheel to change light intensity, or move the mouse to change hue and saturation. On pressing the space bar, information about controls is shown in red (or hidden). The key, the control label, and the current value are displayed. `Interactive.MouseWheel` can be used for scalar values, while `Interactive.MouseMove` for 2D value pairs like rectangle size, motion velocity, or shape position.

All changes to interactive parameters are logged into a file for later enaluation. The file appears in the `Project/Log` folder with the same path as the sequence script. The file name includes the sequence name, the execution date end time, and is postfixed with `_interactions`. It lists all changes of interactive parameters with the time they occured, the name of the parameter, and the new value.

Note that interactive controls are only accepted if the stimulus supports interactivity for the given parameter—which typically means that the components of the stimulus concerned with the parameter also do. Section **??** deals with the topic of how to author such components. Existing components and stimuli support interactivity for almost all parameters. This is indicated in the parameter documentation (the attribute string states 'or Interactive' or just '(I)').

4.3 Forced choice and response

For most use cases, sequences are just defined by a list of stimuli. However, psychophysical experiments require querying for and recording responses from users. This can be accomplished by inserting `Response` objects into the stimulus list when defining a sequence. `Response.Start` defines a point in the sequence where a message and a set of options should appear on the screen. The user may respond by pressing a key or clicking on one of the options with the mouse.

Chapter 5

Stimulus composition

GPU programming background

Computer graphics cards are programmed through various specialized Application Program Interfaces (APIs). GEARS employs OpenGL, which provides an abstraction of the GPU hardware components, implementing a sort of virtual computer, constructed as a *pipeline* for rendering images.
The pipeline has multiple *stages*, which can be fixed-function and shader-programmable. Fixed-function stages can be customized by setting so-called *render states*. Special programs, called *shaders*, define the operation of programmable stages. Shaders can be written in OpenGL's own shading language, GLSL.
Pipeline stages perform tasks like filling the GPU memory buffers with arrays of vertices of a shape, projecting them into screen space, assembling triangles which are rasterized, and finally, assigning color to the pixels and sending the information to the target graphic buffer.
A drawing sequence that uses a certain configuration of shaders and render states is called a *pass*. The output of a pass is not necessarily a visible image. It can also be an intermediary dataset, which is streamed for further processing to a later pass. Multiple passes can be organized into a chain, referred to as a *workflow* here.
Modern GPU programming libraries enable low level access to the parallel hardware. This allows for efficient compute-intensive rendering, but requires deep understanding of the GPU. GEARS has been created with the aim to maintain efficient rendering, without demanding advanced knowledge in computer graphics.

Figure 5.1 shows the GPU workflow for rendering a single frame of a stimulus. The nucleus of the workflow is called *core drawing*, which is a set of *passes* designed to produce an intermediate version of the stimulus image, which may be further processed before being displayed on the screen. The core drawing may be preceded by operations like video decoding, random number generation, and/or 3D OpenGL rendering, and can be followed by post-processing steps like spatial and temporal filtering, or gamma compensation.

5.0.1 Workflow organization in GEARS

In GEARS, all stimuli are constructed as a combination of previously defined *stimulus building components* (SBCs), which may fall into the following type categories:

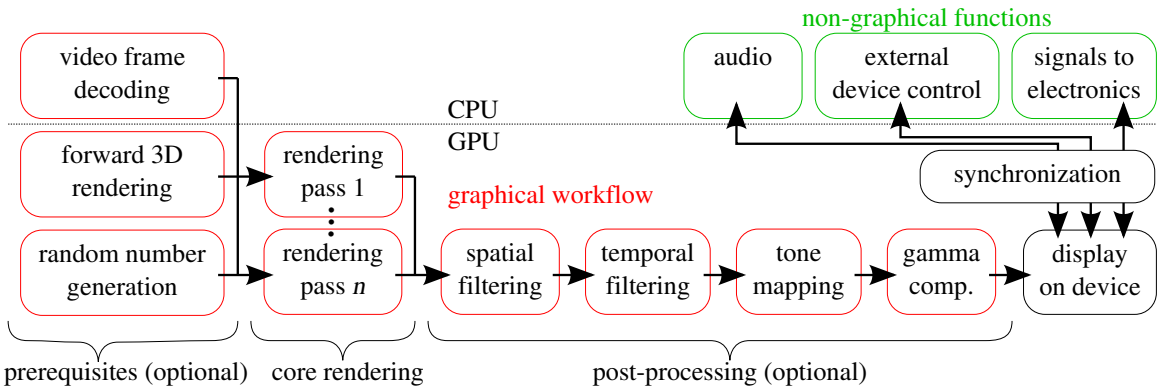

Figure 5.1: Structure of a single stimulus frame.

- PRNG or *pseudo-random number generator*,
- particle system for physics simulation tasks,
- forward for custom 3D rendering,
- PIF or *primitive image function*, the atomic visual building blocks of stimuli (includes shapes, patterns, images from files, video frames, etc.),
- compositor that combines multiple PIFs,
- polygon mask for freeforms and many-shape stimuli,
- motion for shape and pattern animation,
- modulation for intensity change over time,
- warp for shape and pattern distortion,
- time warp for temporal distortion,
- temporal filtering,
- spatial filtering,
- tone mapping,
- gamma compensation,
- interactive for parameters controlled by the user in real time,
- signal for synchronization with measurement electronics and external device, and
- audio.

Component classes derive from `Project.Components.Component`, and they are found in the Python module `Project.Components`, in submodules named after the component types. `Project.Components.Stimulus` (which contains stimulus classes) and `Project.Components.Pass` (which contains pass classes that make up the *core drawing* part of a stimulus) are also submodules of `Project.Components`, but they do not derive from `Project.Components.Component`.

Not all of these component types are relevant for every stimulus, and all of them have meaningful defaults. In this chapter, we discuss the process of stimulus composition in a didactic way—for just a reference, the user should rely on in-code annotations (also displayed during editing by the visual scripting interface). Thus, the role of SBC types is introduced along with the simplest stimulus constructs they are typically used for, from the simple to the more complex.

5.1 Single shape stimuli

It is very often the case that a stimulus displays a single shape, with some kind of motion. A foreground and background color or pattern may have to be applied. For these stimuli, GEARS offers the `Project.Stimulus.SingleShape` class, which accepts all types of components listed above. However, it does not require the user to author the composition of PIFs, but offers intuitive `shape`, `pattern`, and `background` parameters instead. These are automatically composed using linear interpolation between `pattern` and `background`, with `shape` as interpolant. Parameters `modulation`, `motion`, `patternMotion`, and `backgroundMotion` are also applied.

Here is an example of a very simple stimulus class:

```
import Gears as gears # access C++ extension
from .. import * # access all components
from .SingleShape import * # access base class

class SmallWhiteSpot (SingleShape) :
  def boot(
    self,
    *,
    duration_s : 'Stimulus time [s].'
               = 0
    ):
      super().boot(
        name = 'UnitWhiteSpot',
        duration_s = duration_s,
        shape = Pif.Spot(
          radius = 50,
          ),
        )
```

Only the method `boot` needs to be overridden, and the superclass paremetrized appropriately. It is advisable to use only keyword arguments with default values and string annotations, so that these can be provided to the sequence assembler in the visual script editor. This simple example class only exposes one parameter: `duration_s`, the duration of the stimulus in seconds. It leaves all the other parameters of `Stimulus.SingleShape` at their default, values, except for the name (which is used in the display of the sequence overview), and the `shape`. The `shape` paremeter takes a PIF SBC from the `Project.Components.Pif` module (or from `Project.Components.Composition`, but

we discuss that later). PIFs are various parametrizable shape and pattern classes. One of them is the `Pif.Spot` SBC, which defines a disc or an annulus. Here we only set the `radius` parameter, leaving `innerRadius` as the default `0` (a full disc, not an annulus), and `filterRadius_um` also at its default (no softening of the shape edges for antialiasing). Note that the parameters `pattern` and `background` of `Stimulus.SingleShape` have default values `Pif.Solid(color='white')` and `Pif.Solid(color='black')`, respectively, so the result will be a white disc of radius 50μm. Note that if we used `Pif.Solid` as the shape, we would get a fullfield stimulus.

If we wish to add motion, e.g. make the disc move across the screen, we need to set up a `Motion.Crossing` component and pass it as the `motion` parameter to `super().boot()`. Similarly, intensity oscillation can be added using e.g. `Modulation.Cosine` for `modulation`.

It is of course a good idea to expose as many parameters of underlying components in a stimulus class as possible—with appropriate annotation, and default values given. This way, the stimulus class becomes more versatile, but the sequence assembler can just ignore parameters not relevant for her goals. Another useful strategy is to provide a simplified or more intelligent interface than that of the underlying components. The class `Stimulus.Spot` provides examples of both strategies:

```
import Gears as gears
from .. import *
from .SingleShape import *

class Spot(SingleShape) :
  def boot(
    self,
    *,
    duration : 'Stimulus time in frames (unless superseded by duration_s).'
                  = 1,
    duration_s : 'Stimulus time in seconds (takes precendece over duration given in frames).'
                  = 0,
    name : 'Stimulus name to display in sequence overview plot.'
                  = 'Spot',
    radius : 'Disc radius [um].'
                  = 150,
    innerRadius : 'Annulus hole radius [um]. Zero for solid disc.'
                  = 0,
    position : 'Spot center position as an (x,y) pair [(um, um)]. (0,0) is the field center.'
                  = (0, 0),
    color : 'Spot color as an (r,g,b) triplet or a color name.'
                  = 'white',
    background : 'Background color as an (r,g,b) triplet or a color name.'
                  = 'black',
    spatialFilter : 'Spatial filter component. (Spatial.*)'
                  = Spatial.Nop(),
    temporalFilter : 'Temporal filter component. (Temporal.*)'
                  = Temporal.Nop(),
    motion : "Motion component. (Motion.*) If specified, parameter 'position' is ignored."
                  = None,
    modulation : "Intensity modulation component. (Modulation.*) If specified, parameter 'color' is ignored."
                  = None
    ):
      if name == 'Spot' and type(position[0]) == str :
        name = '{r}{c}'.format( r=position[0], c=position[1])
      elif name == 'Spot' :
        name = 'Spot {r} um'.format(r = radius)
      if modulation == None :
        modulation = Modulation.Linear(
                      brightColor=color,
                      )
      if motion == None :
        motion = Motion.Linear(
                      startPosition=position,
                      )
      super().boot(name=name, duration=duration, duration_s=duration_s,
        modulation = modulation,
```

```
        shape = Pif.Spot(
                radius = radius,
                innerRadius = innerRadius,
                ),
        background = Pif.Solid(
                color = background,
                ),
        shapeMotion = motion,
        gamma = Gamma.Linear(),
        spatialFilter = spatialFilter,
        temporalFilter = temporalFilter,
        )
```

Note, first, how stimulus parameters like `spatialFilter` are exposed, but provide a default no-filtering setting, allowing the sequence assembler to ignore the parameter unless a spot with spatial filtering is required. Secondly, `color` and `position` are simpler ways to specify the `pattern` and `motion` parameters of `Stimulus.SingleShape`. The stimulus name is generated from the parameters to be something informative, if it is not explicitly given.

Any PIF can work as shape, pattern, or background. PIFs include a multitude of methods to compute an image, so the `Stimulus.SingleShape` scheme covers ground ranging from oscillating spots or fullfields through random flickering checkerboards to displaying videos, possibly with filterings. It is also possible to use compositors to combine PIFs for the `shape`, `pattern`, and `background` parameters, but then it could make sense to use `Stimulus.SinglePass` instead.

5.1.1 Fade stimuli

Fading from one image to another is often required. The `Stimulus.Fade` class makes this easy. `Stimulus.Fade` is very similar to `Stimulus.SingleShape` discussed above. The key difference is that modulation affects the shape, and not the complete stimulus. As the shape dictates whether the foreground or the background is seen, modulating the shape means fading between the foreground and background patterns. These patterns can be any PIF, including images loaded from files. Indeed, `Stimulus.ImageFade` is a specialized version which works with file names, hiding PIFs.

5.1.2 Single pass stimuli

Stimuli may involve multiple shapes, or a combination of shapes, patterns, and masks different from the shape-pattern-background scheme of single shape stimuli. These can still be rendered on the GPU in a single pass (not counting passes for random number generation, particle systems, or filtering), with a properly assembled custom shader. GEARS generates such a shader from PIF SBCs, composited by compositor SBCs, modified by motion, modulation, warp, and time warp SBCs. The `Project.Stimulus.SinglePass` class exposes this functionality to the stimulus composer.

`Stimulus.SinglePass` takes a `Pass` in parameter `spass`.

5.2 Stimulus Building Components

SBC types for computing prerequisites are

- PRNG SBCs are pseudo-random number generators, which produce a 2D array of random numbers en masse, in every frame. The numbers are computed from previous number arrays.

Note that PRNG SBCs can implement functionality in addition to basic random number generation, e.g. the array of randoms can be shifted between frames, actually generating new numbers only at the edge. PRNG SBCs that use the same set of randoms in every frame are also possible.

- Particle system SBCs are very similar to PRNGs in that they maintain a 2D array of values, updating it in every frame. They are intended for simulation purposes, e.g. for Browninan motion or Ising model simulation. Particle systems often require random numbers from appropriate PRNGs.

- Forward rendering SBCs are responsible for 3D-rendered content.

GEARS uploads images and video frames (decoded by FFMPEG [**?**]) to the GPU automatically, according to the configuration of later passes.

Core drawing passes can be composed of

- image SBCs, which set up the pointwise rendering of images, giving the pixel color as a function of 2D position and time. This is the most populous group of SBCs, with a wide range of functionality. Some of these SBCs generate patterns or shapes (e.g. a sine grid or a disc) procedurally, while being extensively parametrizable. Custom-made new components can easily be implemented by providing the desired formulae defining the pattern. Further image SBCs display video frames, or images either loaded from files, or rendered by forward rendering SBCs. Another subset uses random numbers for producing band-limited or white noise, or randomly positioned and animated shapes.

- Members of the image composition SBC group can be used to combine multiple image SBCs. They are implemented by function composition: image functions are substituted into composition functions, yielding a new image. Building a full operation tree is possible.

- *Polygon mask* SBCs allow the definition a binary image by providing the control vertices of its contour(s). This is helpful for drawing free-form shapes, where a formulaic definition is not feasible. Such shapes can be used, among others, for masking in the pointwise rendering scheme.

Several types of SBCs modify raw image SBCs:

- motion SBCs enable changing, as a function of time, the scale, orientation, and position, e.g. for shifting shapes with different orientations,

- modulation SBCs vary the intensity or contrast as a function of time,

- warp SBCs can be used to distort renderings, e.g. for toroidal or spherical screens, or tile them indefinitely,

- time warp SBCs distort or reverse the time scale, and they can introduce periodicity.

Modified image SBCs accept further modifiers, allowing them to be chained up indefinitely.

The image resulting from core drawing can be subject to various image processing operations. SBC types performing these tasks are the following:

- spatial filtering SBCs compute a convolution with a filter kernel, either in the spatial or the frequency domain,

- temporal filtering SBCs convolve the image stream with a temporal filter kernel, or channel it through a linear data processing system,
- tone mapping SBCs renorm the pixel intensity values, which may exceed the dynamic range of the display device after a filtering operation, and
- gamma compensation SBCs cancel the nonlinearity introduced by the light projecting hardware.

Additionally, non-graphical tasks performed alongside stimulus rendering are defined by further SBCs:

- audio SBCs play various audio file formats, and
- signal SBCs provide synchronization, start and stop signals for the recording electronics.

All component types have several different implementations. Those can be used to create a stimulus by parametrizing a `Stimulus.Generic` stimulus with all those components. However, a number of specialized classes hiding this mechanism have already been created, and new ones are certainly possible.

Creating new stimulus types from existing components requires some Python programming skills, but it amounts to not much more than mechanically passing parameters to the components. This is the user level most users are expected to master: building new stimulus types from a library of existing components.

5.3 Tone mapping