Holocube software introduction

The holocube software is a python module to display arbitrary, perspective-corrected, rapidly updating visual scenes. We have developed it primarily for behavior experiments in back-projection flight arenas (Fig. 1), but use it now in many other situations. It allows you to coordinate 3 dimensional immersive displays, precisely timed visual stimuli, closed loop visual feedback, and probably more things we haven't thought of.

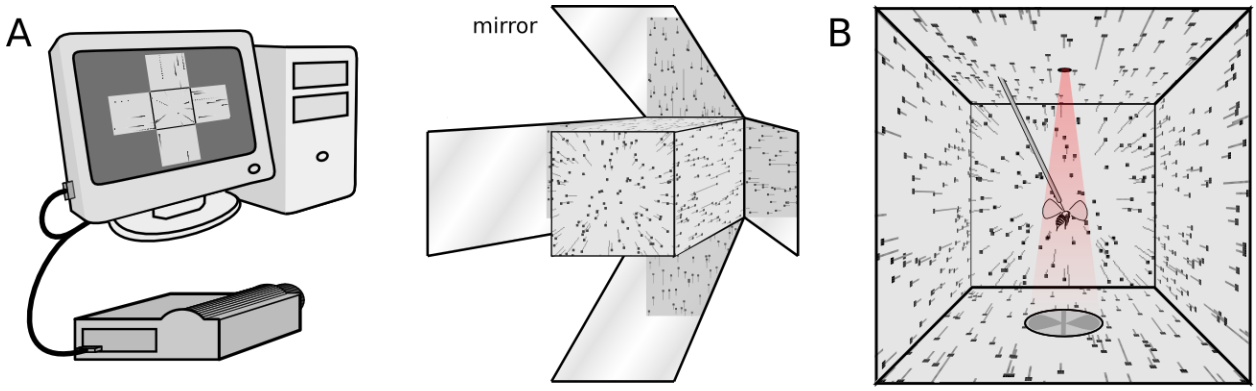

Figure 1: (A) the holocube software displays a moving dot field onto a back-projection flight arena, (B) so an insect tethered in the center views perspective-corrected optic flow.

Some terms here have specific meanings

Opengl is a programming interface for interacting with your computer's GPU, which we can access with python. Some important terms from opengl are *screen*, which refers to the physical display device, such as a computer screen or a projector, and *window*, which is the rectangle that opengl is rendering in. The *window* can take the whole *screen*, such as when we use the projector, or part of it, such as when we test a script on a laptop. A *viewport* is a rectangular area of the *window* that renders a view from a hypothetical camera in the virtual world, such as the projection onto the left facing side of the holocube arena (Fig. 2). So *screen*, *window*, and *viewport* are standard terms in opengl, and all of them are set up in the configuration file.

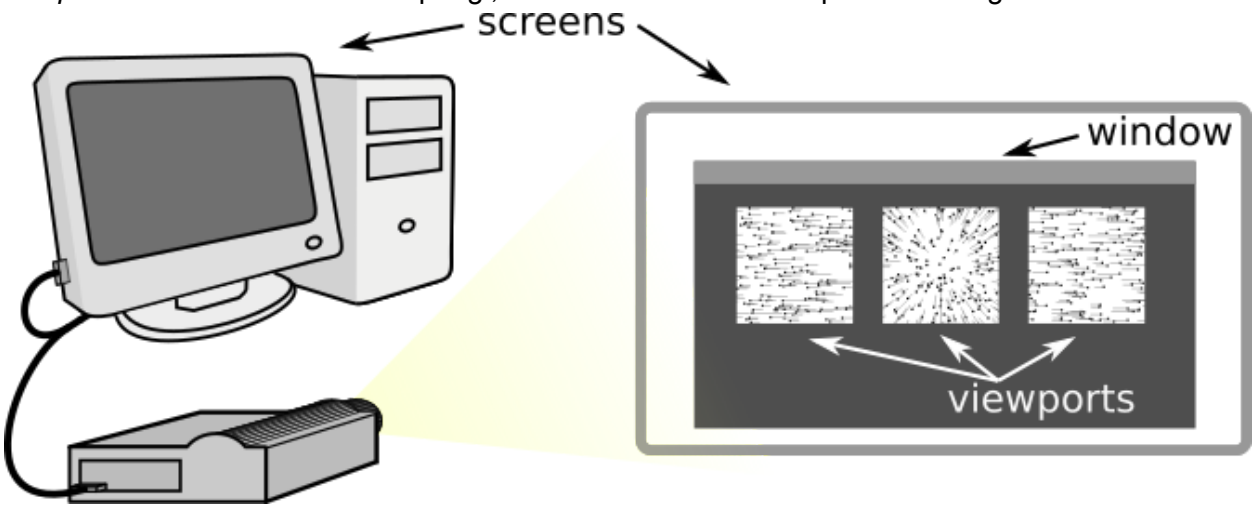

Figure 2. In opengl, screens, windows and viewports are different things.

For the holocube software, I settled on some terms that are somewhat arbitrary just to keep track of what different parts of a stimulus mean. I usually display a series of visual scenes that vary in some way, like a sinusoidal grating that, with each subsequent viewing, show a different spacing between the bars, or a moving field of dots that shows progressively more dots with each appearance. The goal is usually to change one variable and check for behavioral differences, like strength of steering responses. I call a series of visual stimuli an *experiment*, and a single *experiment* is usually coded with a python script in a single file. Each of the presentations in an *experiment*, like the dot field with 200 dots/steradian, or the one with 100 dots/steradian, is called a *test*. In between *tests* we often display something else, like a vertical bar that responds to steering feedback in closed loop, and this presentation is called a *rest* (a *rest* can display nothing, if you like). A presentation of all the different *tests*, interrupted by the *rests*, constitute a *trial* of an *experiment*. We often use just one fly per *trial* (we don't rep flies), meaning that one *trial* corresponds to a single fly responding to all the *tests* in an *experiment*. The *rest* is actually optional, and multiple *tests* are also optional---an *experiment* can be made of only a single *test*, if you like. So, *experiment*, *test*, *rest*, and *trial*, we use in a specific way just for this software, which you need to know in order to write experiment scripts.

Files follow a simple structure

Holocube experiments can run from any folder as long as it contains a run script, a configuration file, a folder with the experiment scripts, and a folder with the holocube module. If you want to install the module with python and distutils, you can use the setup script, then the holocube module folder isn't needed. The run script, `run.py`, is what you actually execute to start the display and experiments. It needs to import holocube from the `holocube` folder, read a configuration file from `viewport.config`, and read the experiment scripts you've written from `experiments` (Fig 3). Most of the names can be changed, and you can have multiple run scripts or configuration files to control different experimental setups. You will also add and delete experiment scripts, or use a set of experiment folders, to control which experiments are available when you run holocube.

One small note is that when you first run python files they generate bytecode, a process that only needs to happen the first time they are run. To avoid redoing this step, python generates bytecode files, which have the extension `.pyc`. So after you successfully run holocube, there will be more files with similar names in your directories, such as `windows.py` and `windows.pyc`. These are all taken care of automatically, so you never need to worry about them. If they are accidentally deleted, they will simply be remade on the next run.

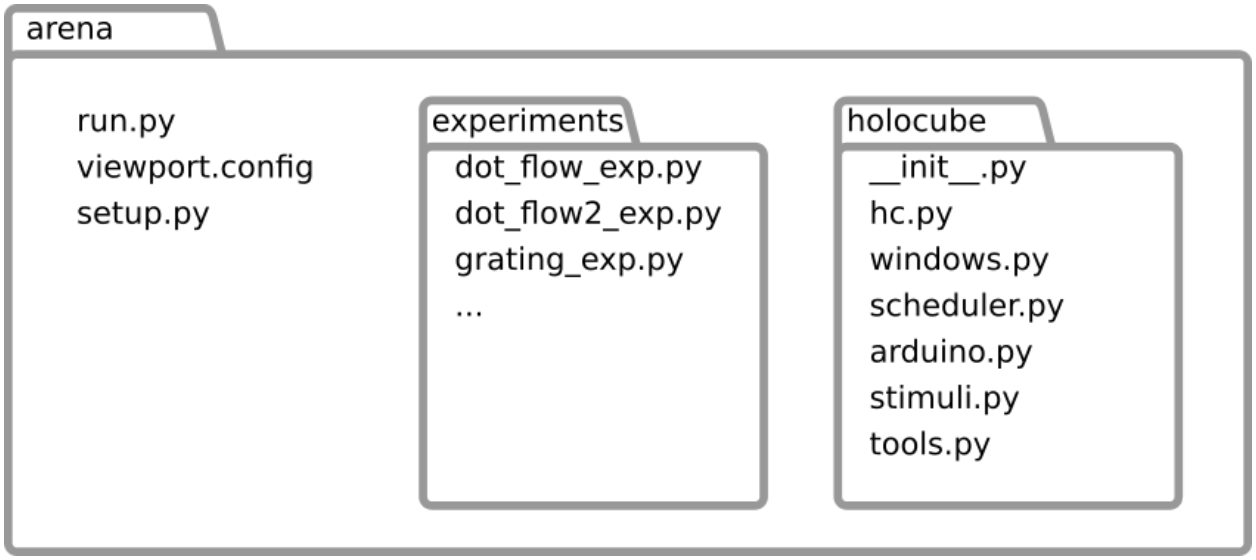

Figure 3. Folder structure showing the simplest files needed to run holocube scripts.

Run script is short

```
#! /usr/bin/env python
# run.py
import pyglet
import holocube.hc as hc

hc.window.start(config_file='viewport.config')
hc.arduino.start('/dev/ttyACM0')
hc.scheduler.start(hc.window, randomize=True, default_rest_time=.1)
hc.scheduler.load_dir('experiments', suffix=('exp.py', 'rest.py'))
print('ready')

pyglet.app.run()
```

Holocube is launched with a run script, usually named run.py, but it could be named anything at all. It has only a few essential lines of code. The first line, with `#!` (called shebang) looks like a comment, and it is a comment from python's point of view, but when executing a script without knowing the language, this first line details which command to use (the default python, in this case). So leave this as the first line of the script. It won't matter if you run it right from python, but if you want to run by double clicking, this has to be in place.

Imports

First we import `pyglet`, which allows python to interact with the GPU, and the holocube module itself, which I always abbreviate as `hc`:

```
import pyglet
```

```
import holocube.hc as hc
```

Now we can use the `hc.` prefix to reference all the functions and objects the module created.

Window

`hc` initializes a few different objects (`window`, `scheduler`, and `arduino`), which we now need to start running by calling their `.start` methods. We will always start the display, which requires the name of a config file. This file only needs to be generated once, usually by running the config scripts when you build a new arena. I'll cover the config file generation separately, but the file is specific to a display, and you don't need to redo it unless you move something or bump the projector.

```
hc.window.start(config_file='viewport.config')
```

Arduino

Then, if you are running a rig that gets input from an arduino, start that. An example is the wing beat analyzer, which feeds its output into an arduino and gives real time input to the computer. This is used in scripts that generate feedback, such as the bar tracking between tests. The start function requires a device filename, which is a string, either the path to an attached arduino (`'/dev/ttyACM0'`), or `'dummy'` if you want to simulate an attached arduino which generates random inputs. `'dummy'` is useful for writing experiment scripts when you are not actually on the rig with the arduino attached. We'll assume you don't have an arduino yet for this tutorial.

```
hc.arduino.start('dummy')
```

Scheduler

Then start the scheduler, which manages all the tests of all the experiments, frame by frame to send each to the window display manager. The only parameter it actually needs is the window it will control, `hc.window`. The others are optional, such as whether the order of tests in an experiment will be random, how long is the rest time between tests if the experiment script doesn't specify (irrelevant if it does specify).

```
hc.scheduler.start(hc.window, randomize=True, default_rest_time=.1)
```

Everything will work now, but there are no experiments to run. To load experiments we execute the scheduler function `.load_dir` and specify the directory that has experiments in it ('experiments', 'tests', 'my_experiments'...). Normally, we specify experiment files by ending them with 'exp.py', such as 'spatial_grating_exp.py', or 'dot_flow_speeds_exp.py', but really we can use any ending we want, if we specify it here as the suffix parameter. If you choose something besides 'exp.py', such as 'test.py', the scheduler will only load scripts that end with that suffix. We can also specify rest scripts that play when no experiment is running, between experiments or between the tests of an experiment doesn't specify its own rest script. The default suffix is 'rest.py'. If you want to use the defaults, you can omit all the arguments.

```
hc.scheduler.load_dir('experiments', suffix=('exp.py', 'rest.py'))
```

Running it all

You can do other things, (I usually `print('ready')`, so I know the experiments have loaded), but that is all the setup required. To actually set it all in motion, the last line of the run script is:

```
pyglet.app.run()
```

An experiment script

```
# pt_flow_exp.py
import holocube.hc as hc
import numpy as n

# 3 seconds per test
num_frames = 360

# make a dot field
pts = hc.stim.Points(hc.window, 10000, dims=[(-5, 5), (-5, 5), (-5,
5)], color=1, pt_size=4)

# experiment: add this experiment to the scheduler
exp_starts = [[hc.window.set_far, 3]]
exp_ends = [[hc.window.set_far, 1]]
hc.scheduler.add_exp(name=None, starts=exp_starts, ends=exp_ends)

# test1: add a test to the experiment
starts = [[pts.switch, True]]
middles = [[hc.window.inc_slip, 0.01]]
ends = [[pts.switch, False],
        [hc.window.reset_pos]]
hc.scheduler.add_test(num_frames, starts, middles, ends)

# test2: add another, similar test to the experiment
middles = [ [hc.window.inc_slip, -0.01] ]
hc.scheduler.add_test(num_frames, starts, middles, ends)

# add the rest
rest_frames = 240
bar = hc.stim.Bars(hc.window)

starts = [[bar.switch, True]]
middles = [[hc.window.inc_yaw, hc.arduino.lmr]]
ends = [[bar.switch, False],
        [hc.window.reset_rot]]
```

```
hc.scheduler.add_rest(rest_frames, starts, middles, ends)
```

Once a run.py script and configuration file are working, you can use them forever with the same rig, and change the experiments you run just by putting different files in your 'experiments' folder (or whatever you want to name it). The scheduler will try to load all your experiments, but it requires a few elements in each script. First, the file itself has to be named `something_exp.py`, where *something* can be anything. I prefer precise, descriptive names that use underscores but no spaces. I hate filenames with spaces, they only cause trouble. To make my life easier, I map SHIFT SPACE to underscore (_) on my computer so it is easy for me to type with underscores instead of spaces. This is a great idea, but optional for you.

Imports

Experiment scripts must import `hc.` to get access to the `window`, `arduino`, `scheduler`, and another module that I didn't discuss in the run script, the `stimuli`.

```
import holocube.hc as hc
```

It's not actually required, but it is also a good idea to import `numpy`, for all the useful functions we might need from it. It's already imported with all the other modules running, so the import doesn't tax resources, just lets us use its functions in our experiment script.

```
import numpy as n
```

Test length

I always put the number of frames that each test will use up top here. You can just specify it later when you call the functions, but this makes it easy to change. Each test could in theory use a different number of frames, but I've never tried this, and I certainly won't here in a simple example. The frames on many displays will run at 60 frames per second, but on our projectors, they run at 120 fps (we get 360 fps by using the three color channels, taken care of behind the scenes in the stim module)

```
# 3 seconds per test
num_frames = 360
```

This number could also be `n.inf`, which is an infinite test that will not exit except by aborting the experiment with the BACKSPACE key. This is frequently useful, especially if you assign key press controls that change the scene.

A stimulus to display

Next we make a visual stimulus that holocube will display during the tests. This will generally be from the stimuli.py module, which we can refer to as `hc.stim` (from the import at the start of

the script). For this example, I'll use `hc.stim.Points`, which makes a set of dots, randomly placed in 3D.

```
pts = hc.stim.Points(hc.window, 10000, dims=[(-5, 5), (-5, 5), (-5, 5)], color=1, pt_size=4)
```

The first argument is our window where the points will display, followed by the number of points, which I'm setting to 10000. Then, dims is the x, y, and z ranges over which they are scattered, -5 to 5 in each case, creating a cube filled with 10000 dots. Then the color (1 is white), and pixel size. Now I can refer to `pts` in the code and manipulate this set of dots. Even though it is created and in the correct window, it starts off as invisible, until we switch it to visible.

Add an experiment

To make an experiment with these `pts`, we have to add an experiment to the scheduler, and at least one test to that experiment. The function is `hc.scheduler.add_exp`, and it takes 3 arguments, starting with the name. Experiments have starts, a set of commands that initiate everything, middles, the tests and rests they display, and ends, which set everything back to however we want to leave it. In many cases we can ignore these arguments and not use them, but I'll set one up for demonstration:

```
exp_starts = [[hc.window.set_far, 3]]
exp_ends = [[hc.window.set_far, 1]]
hc.scheduler.add_exp(name=None, starts=exp_starts, ends=exp_ends)
```

The name is set to `None`, which means the experiment name will be named after the file (for example 'dot_flow_exp.py'), but you can choose a different name if you like. The argument exp_starts takes a list of functions to execute, and the arguments those functions take. Here we have a function, `hc.window.set_far`, which takes the argument `3`, and sets the maximum distance for objects that the viewport can render. In other words, how far away can these points be before we don't draw them on the screen anymore. If they are closer than 3 units away from the virtual camera, they are drawn. Any farther, and they are invisible. You can set any finite value for this. The function is executed one time when you press the key to start your experiment. When the experiment is over, or aborted, the `exp_ends` execute, and resets the far view back to default of `1`.

Add tests

The middle part of running an experiment is displaying the tests, one by one, interspersed with rests. This requires calling the function `hc.scheduler.add_test`, which takes 4 arguments.

```
starts = [[pts.switch, True]]
middles = [[hc.window.inc_slip, 0.01]]
ends = [[pts.switch, False],
        [hc.window.reset_pos]]
hc.scheduler.add_test(num_frames, starts, middles, ends)
```

The first argument is just the number of frames for the test, which we set as a variable at the top of the script. The second argument is the start functions, a list of all the functions and their arguments to execute at the beginning of this test---analogous to the `starts` argument of the `add_exp function`. These are executed exactly once when the test begins, before the first frame. Here we just use one, the `.switch` function of our `pts` object, which takes the argument `True` to indicate switching the points to visible. Next is the `middles` (not an argument for add_exp), which is a list of functions and their arguments executed every frame (so `num_frames`, or 360 times in this case). Again we just use a single function for this, `hc.window.inc_slip`, which advances the viewpoint left by .01 units every frame. Finally `ends` has all the functions and arguments executed when the test is finished or the experiment is aborted, just once. There are two functions here, one that switches the points to invisible, and another that resets the viewpoint position back to the center.

Now let's add a second test to our experiment:

```
middles = [[hc.window.inc_slip, -0.01]]
hc.scheduler.add_test(num_frames, starts, middles, ends)
```

Here we changed the `middles` variable to a negative slip, left `starts` and `ends` the same, and called `hc.scheduler.add_test` again, generating a second test for the experiment.

Add rest

Now we have two tests, but what happens in between their display? Nothing has to, but if we want a pause, maybe to let the subject reset its behavioral state, we can add a rest period. We don't have to do this in the experiment script, we can set it up as a default, but it can be better to put the code here so you know exactly what happened when you examine it later on. It follows the same pattern as adding tests, except we add only one rest, with `hc.scheduler.add_rest`.

```
rest_frames = 240
bar = hc.stim.Bars(hc.window)

starts = [[bar.switch, True]]
middles = [[hc.window.inc_yaw, hc.arduino.lmr]]
ends = [[bar.switch, False],
        [hc.window.reset_rot]]
hc.scheduler.add_rest(rest_frames, starts, middles, ends)
```

Two things are new here. First, we generated a bar from `hc.stim.Bars`, similar to generating the point field before. We are just using default values, but you can specify many details. Second, the `middles` function list uses `hc.window.inc_yaw`, similar to the `hc.window.inc_slip` above, but changing rotation instead of forward position. However, the

argument to `inc_yaw` isn't a number, but a function, `hc.arduino.lmr`. What this does is specifies that rather than use a single number for the yaw increment, query the function from the `arduino` module (`lmr` for **l**eft **m**inus **r**ight wing beat amplitude), which gives an instantaneous measure of steering effort, and then this number determines the yaw. This generates a closed-loop steering experience, where the fly, in this case, seems to control the angle of the bar by steering left and right. Flies like controlling bars. In both tests and rests, arguments to the functions that appear in `middles` (which are called `num_frames` times, instead of once) can take a single argument, like `.01`, a function, like `hc.arduino.lmr`, or an array of values, like `n.linspace(0, 2, 100)`. A single value is simply used for each frame, a function is evaluated right when the frame is rendered, and an array (from the numpy module which we imported at the beginning) is a sequence of values which are subbed in with each frame, looping back around if we reach the end.

An experiment script with a loop

The last experiment was fine, and illustrated everything we need to understand about adding experiments and tests, but it would be tedious to add many tests this way, specifying all the parameters and repeating code. However, we can easily add a series of tests by iterating over a loop, a code structure that evaluates multiple times, sometimes assigning a different value to a variable with each pass.

```
# pt_flow_series_exp.py
import holocube.hc as hc
import numpy as n

# 1 second per test
num_frames = 120
num_tests = 11

# make a dot field
pts = hc.stim.Points(hc.window, 5000, dims=[(-20, 20), (-5, 5), (-5,
5)], color=1, pt_size=4)

# experiment: add this experiment to the scheduler
exp_starts = [[hc.window.set_far, 3]]
exp_ends = [[hc.window.set_far, 1]]
hc.scheduler.add_exp(name=None, starts=exp_starts, ends=exp_ends)

# generate a series of velocities between -.1 and .1
velocities = n.linspace(-.1, .1, num_tests)

# add a test for each velocity with a loop
for velocity in velocities:
     starts = [[pts.switch, True]]
```

```
     middles = [[hc.window.inc_slip, velocity]] #notice velocity
here
     ends = [[pts.switch, False],
             [hc.window.reset_pos]]
     hc.scheduler.add_test(num_frames, starts, middles, ends)

# add the rest
rest_frames = 120
bar = hc.stim.Bars(hc.window)

starts = [[bar.switch, True]]
middles = [[hc.window.inc_yaw, hc.arduino.lmr]]
ends = [[bar.switch, False],
     [hc.window.reset_rot]]
hc.scheduler.add_rest(rest_frames, starts, middles, ends)
```

This code is identical to the previous section except we have assigned a new variable up top:

```
num_tests = 11
```

And used it to make an array of that length, which takes 11 values that increment from -0.1 to +0.1:

```
velocities = n.linspace(-.1, .1, num_tests)
```

So `velocities` is a sequence of values:

```
array([-0.1, -0.08, -0.06, -0.04, -0.02,  0.,  0.02,  0.04,  0.06,
0.08,  0.1])
```

And when we generate a loop:

```
for velocity in velocities:
    starts = [ [pts.switch, True] ]
    middles = [ [hc.window.inc_slip, velocity] ] #notice *velocity*
    ends = [ [pts.switch, False], [hc.window.reset_pos] ]
    hc.scheduler.add_test(num_frames, starts, middles, ends)
```

The variable `velocity` takes on, one by one, each value in `velocities`, and we add a new test with that argument given to `hc.window.inc_slip` In the `middles` list. For python, everything that is indented after a line with a colon is part of the same block, commands that are executed together, so the four space indent is important syntax, not just style.

A grating stimulus with a loop

```
# grating_series_exp.py
import holocube.hc as hc
import numpy as n

# 1 second per test
num_frames = 120
num_tests = 11

# make a dot field
pts = hc.stim.Points(hc.window, 5000, dims=[(-20, 20), (-5, 5), (-5,
5)], color=1, pt_size=4)
square = n.array([[-1,1,1,-1],[-1,-1,1,1], [-1,-1,-1,-1]])
grating = hc.stim.Movable_grating(hc.window, square, sf=5, tf=3, o=0,
sd=.35)


# experiment: add this experiment to the scheduler
exp_starts = [[hc.window.set_far, 3]]
exp_ends = [[hc.window.set_far, 1]]
hc.scheduler.add_exp(name=None, starts=exp_starts, ends=exp_ends)

# generate a series of angle positions between -90 and 90
angles = n.linspace(-90, 90, num_tests)

# add a test for each velocity with a loop
for angle in angles:
     starts = [[pts.switch, True],
          [grating.set_ry, angle],
          [grating.switch, True]]
     middles = [[grating.next_frame]]
     ends = [[pts.switch, False],
          [grating.switch, False]]
     hc.scheduler.add_test(num_frames, starts, middles, ends)

# add the rest
rest_frames = 120
bar = hc.stim.Bars(hc.window)

starts = [[bar.switch, True]]
middles = [[hc.window.inc_yaw, hc.arduino.lmr]]
ends = [[bar.switch, False],
     [hc.window.reset_rot]]
hc.scheduler.add_rest(rest_frames, starts, middles, ends)
```

Next is a stimulus that adds a moving grating to the points in the background. These are extremely useful for vision experiments because they can isolate vertical, horizontal, and temporal frequencies, and have well-defined contrasts. The code listing above is similar to the previous dot flow experiment, but we make an additional object, a grating:

```
region = n.array([[-1,1,1,-1],[-1,-1,1,1], [-1,-1,-1,-1]])
grating = hc.stim.Movable_grating(hc.window, region, sf=5, tf=3, o=0,
sd=.35)
```

Which we simply move to a set of different angles (-90 to 90 degrees), perhaps to look for regional effects on behavior.

Like `stim.Points`, `stim.Movable_grating` takes the window it will display in as its first argument, `hc.window`. Gratings are powerful objects in holocube, and their other arguments require just a bit of explanation.

First, the variable `region` is a list of x, y and z coordinates of the three dimensional quadrilateral in which to display the grating. Here we are using a simple square floating 1 unit away from the center of the coordinate system. It is fine to move the grating to a different distance, or move the viewpoint somewhere else, but this will change visible spatial frequency, orientation, and contrast (which is always the case when you move things around in space). Temporal frequency is not affected (since we move through time in a more predictable way). After this are the parameters that describe the grating, spatial frequency, temporal frequency, orientation, contrast (which defaults to 1.0 since we don't specify it here), and sd, meaning the standard deviation of the gaussian window that frames the grating. If sd is `None` the grating will fill the region, if it is smaller than 0.35 it will taper off more quickly.

Importantly, as long as the grating is 1 unit away, upright, and facing the viewpoint directly, the values for spatial frequency are correct in cycles/radian. Although it might look distorted on the screen, it is actually accommodating for the perspective of the virtual camera, so that from the perspective of an insect in the arena, the spatial frequency is always correct. This won't work, however, for large extended region (like a really large square), so we have other structures that can wrap around and keep the spatial frequency constant (`stim.Quad_image`)

Finally, for this experiment, we implement a series of tests that move the grating to different angles:

```
for angle in angles:
      starts = [[pts.switch, True],
            [grating.set_ry, angle],
            [grating.switch, True]]
      middles = [[grating.next_frame]]
      ends = [[pts.switch, False],
```

```
            [grating.switch, False]]
    hc.scheduler.add_test(num_frames, starts, middles, ends)
```

This loop simply plays the moving grating that we defined in several positions on the screen. This is useful to look for regional response differences. However, we probably want to also examine responses to different spatial and temporal frequencies. The advantage to using `stim.Movable_grating` is that the intensive computations to calculate grating movement are done in advance, so holocube only has to play the stimulus during an experiment.

An experiment for spatial frequency tuning curves