Peacock Documentation

Jonathan Y. Chan <jyc@fastmail.fm>

Contents

Peacock

Peacock is a language for mission unit testing. Its goal is to make the automated and rapid testing of missions straightforward.

This is the documentation for Peacock. It is currently a work in progress.

Acknowledgements

Peacock was developed by Jonathan Chan <jyc@fastmail.fm> for the Cornell University Autonomous Underwater Vehicle project team.

Thanks to Chris Goes and Alex Spitzer for their support of this project throughout its long development, and to everyone on the software subteam for inspiring it.

Installing

The Peacock source code is included in the main CUAUV software repository.

Before setting up Peacock, you should ensure some environmental variables are set in the script you use to set up your CUAUV environmental variables. The `CUAUV_SOFTWARE` environmental variable should be set to the path to the CUAUV software repository, with a trailing slash. This is used by various Peacock and Fishbowl libraries and tools to reliably load files from within the software repository independent of the current directory. The `CSC_OPTIONS` environmental variable should be set to:

```
-L$CUAUV_SOFTWARE/link-stage -C -I$CUAUV_SOFTWARE -I$CUAUV_SOFTWARE
```

This tells the CHICKEN Scheme compiler to add the appropriate include and link paths to compile CUAUV software.

To build, you will need to first install the CHICKEN Scheme system. On Arch Linux, this can be done in one step with `pacman -S chicken`.

After installing CHICKEN, you will need to install the `matchable`, `bitstring` and `chicken-async` CHICKEN Scheme extensions. The first two can be installed with `chicken-install -s matchable` and `chicken-install -s bitstring`.

`chicken-async` is still experimental, so it is not yet in the CHICKEN Scheme egg repository. Nonetheless, is not hard to install:

```
hg clone ssh://hg@bitbucket.org/jyc/chicken-async
cd chicken-async
chicken-install -s
```

To make sure that these packages (called "eggs" by CHICKEN) have been successfully installed, start up a CHICKEN interpreter with the `csi` command. Write `(use matchable bitstring async)`, then press Enter. It should look something like this:

```
#;1> (use matchable bitstring async)
#; loading /usr/lib/chicken/7/matchable.import.so ...
#; loading /usr/lib/chicken/7/bitstring.import.so ...
#; loading /usr/lib/chicken/7/srfi-4.import.so ...
#; loading /usr/lib/chicken/7/async.import.so ...
#; loading /usr/lib/chicken/7/bitstring.so ...
#; loading /usr/lib/chicken/7/async.so ...
```

You can exit the interpreter with `Ctrl-D`.

After you have installed Peacock's dependencies, you can build Peacock with the `auv-build-chicken` binary built by `ninja` in the CUAUV software repository. This will build the `cuauv-shm`, `fishbowl` and `peacock` CHICKEN extensions located at `libshm/scm`, `peacock/fishbowl`, and `peacock`, respectively. Make sure that you have run `git pull` and `ninja` beforehand!

Motivation

State is important. The goal of the AUVSI RoboSub competition is to change, or mutate, the state of the vehicle from its start state to a state in which it has won the competition. To do this, we write mission programs that, taking into account the current state of the vehicle — e.g., the `kalman` shared memory group — change state — e.g., by writing to the `desires` shared memory group — with the goal of moving the vehicle towards a desired state. There is a hierarchy of vehicle states, ranging from low-level physical states to high-level mission program states. The vehicle may have a forward speed of 0.3 m/s while it is trying to approach bins as part of an overall mission.

Dealing with all of this state is difficult, so we try to abstract it away. The writer of a mission program doesn't deal with pushing bits over a cable to make the vehicle center on a target. Perhaps the program specifies a `Sequential` task having a `ForwardTarget` subtask which sets desired speeds. These desires are read by the controller, which tries to find the optimum balance of thruster outputs to achieve them. The controller writes desired thruster PWMs to shared memory and the serial daemon communicates with motor boards that convert these PWMs to power applied to the actual motors.

Even then, a mission might be hundreds of lines of code backed by hundreds of lines of mission framework code, amounting to a complicated state machine. Each mission has to be thoroughly tested for many hours at the pool in order

to ensure that all the state transitions work as they should. Each test can take minutes and require careful observation of the vehicle by multiple people in order to ensure, for example, that the vehicle doesn't run into the wall. Even when software works perfectly, mechanical or electrical failures can lead to an environment that is not conducive to careful software development. And procedural changes that might be made afterwards in an environment such as refactoring of mission logic require repeating the same exact tests many times more.

When we test mission programs, we have a mental model of the ideal mission state machine that we compare with the actual performance of the vehicle. As we sit at the side of the pool, we go through state transitions in our heads and watch to see if the vehicle acts as if it has done the same. It should be possible to encode this mental model of a state machine into a form that computers can verify.

But writing mission code is difficult enough. And if it is so hard to describe in code how a mission should work, describing all the ways in which a mission should pass or fail a test seems like an ambitious goal. In addition, if writing tests for missions appears to developers more difficult than just going to the pool and testing the mission "in person", the testing system will not see much use.

Peacock is a language for writing automated mission tests that hopes to reduce the iteration time of mission development, to help developers make the mission programs they write more reliable, and to pave the way for large-scale ambitious refactoring of mission and mission framework code. To do this, it draws on concepts from monadic asynchronous programming seen in frameworks such as Jane Street's Async and languages like Haskell. Peacock adds specialized control structures, such as branching and peppered branching binds, in order to make mission test code clearer and more succinct.

Unfortunately, I expect that there will be a bit of a learning curve, firstly because Peacock is written in Scheme[1], secondly because of the asynchronous programming model[2], and thirdly because some of the control structures are unorthodox. I hope this document will help you to get off to a good start.

Infrastructure

Peacock tests are programs written in the Scheme programming language, implemented by CHICKEN Scheme. The "language" Peacock exposes is provided by the `peacock` CHICKEN extension. This extension, along with others, is loaded by the `peacock-prelude` file included at the beginning of every Peacock test.

[1] This was done mostly in order to take advantage of Scheme's syntax transformation facilities. And of course, also because Scheme is awesome!

[2] ... but you would have run into it in CS 3110 anyways!

Because Peacock tests are self-contained programs, they can be run independently using standard CHICKEN tools. CHICKEN allows for programs to be run by `csi`, the CHICKEN Scheme interpreter, or compiled by `csc`, the CHICKEN Scheme compiler. Programs should behave identically when run with either `csi` or after being compiled by `csc`. This allows for both rapid iteration during development — ironing out bugs after starting to write a test file — as well as more performant execution after code has settled down.

To make the process of creating and running Peacock tests more easy, the `peck` program is included as part of the `peacock` distribution. Peck is a command-line program exposing `interpret`, `compile` and `new` subcommands.

Suppose we want to write a test for a bins mission. Running `peck new bins` will create a new Peacock test in a file named `bins.scm` in the current directory. This file will contain a minimal Peacock test. We'll go over the syntax later.

```
(include "peacock-prelude")

(peacock "New Peacock Test"
  (setup
    (vehicle-set!
      x: #(0 0 0)
      )

    (>>= (after 120)
         (lambda (_)
           (fail "Timed out.")))

    (camera down
      q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
      w: 1024 h: 768 f: 440)

    (camera forward
      q: (hpr->quat (vector 0 0 0))
      w: 1020 h: 1020 f: 440)

    (goals
      win
      )
    )

  (mission
    module: "dummy"
    task: "Dummy")

  (options
    debug: #f)
```

```
  )

; vim: set lispwords+=peacock,vehicle,entity,camera,camera-stream,shape,collision :
```

`peck new` will open the file it creates in the editor defined in the `EDITOR` environmental variable, or `nano` if `EDITOR` is undefined.

We could start by replacing the template name of the test, "New Peacock Test" with "Bins Test", and the template mission module and task with the module and task of our bins module. Peacock executes modules by opening `mission/runner.py` as a child process, so the mission module and task you write should be the same as you would supply to the mission runner.

After writing the mission test file, you can run it in interpreted mode with `peacock interpret bins`. After you have rid your mission test of most bugs, you can compile it into an executable for increased performance with `peacock compile bins`. This will create a `bins` executable in the current directory that you can run as any other binary, e.g. with `./bins`.

On startup, a Peacock test loads the Fishbowl programmable simulator and the mission runner as child processes. It communicates with the Fishbowl simulator over the network to specify simulation entities, cameras, and other objects.

`peck n`, `peck i`, and `peck c` are aliases for `peck new`, `peck interpret`, and `peck compile`, respectively.

Writing your first test

Let's write a test for a buoy mission. The buoys task involves the vehicle touching a red and then green buoy in sequence. First, let's create a Peacock test file with the `peck new buoy` command. Peck will open the new `buoy.scm` file in an editor. It should look like this:

```
(include "peacock-prelude")

(peacock "New Peacock Test"
  (setup
    (vehicle-set!
      x: #(0 0 0)
      )

    (>>= (after 120)
         (lambda (_)
           (fail "Timed out.")))

    (camera down
```

```
      q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
      w: 1024 h: 768 f: 440)

    (camera forward
      q: (hpr->quat (vector 0 0 0))
      w: 1020 h: 1020 f: 440)

    (goals
      win
      )
    )

  (mission
    module: "dummy"
    task: "Dummy")

  (options
    debug: #f)
  )

; vim: set lispwords+=peacock,vehicle,entity,camera,camera-stream,shape,collision :
```

Peacock test code must go inside the `peacock` macro application. This is the code that starts with `(peacock` and ends with the matching `)` at the end of the file. We'll refer to what's inside the `peacock` macro application as the Peacock test.

As an aside, everything in Scheme that has the form `(foo ...)` is either a procedure or macro application or a statement.[3] The things that come after `foo` in the parentheses are the arguments to the procedure, macro, or statement. Arguments can be other procedure applications, macro applications, statements, or values, like numbers, strings, booleans, etc. In short, when in Java you might write:

```
transferMoney("Abel", "Bob", 300);
```

... in Scheme you write:[4]

```
(transfer-money "Abel" "Bob" 300)
```

When you would write in Java `5 + 3`, in Scheme you write `(+ 5 3)`. For a more in-depth tutorial, Chapter 2 of The Scheme Programming Language, 3rd Edition by Dybvig is a good reference that is available online.

[3] Ignoring different read syntax such as quoting, quasi-quoting, etc.
[4] Yep, you can use `-` in identifiers in Scheme.

Note that you can also write vectors using the syntax `#(x y z)`, where `x`, `y`, and `z` should be literals specifying the x, y, and z coordinates of the vector. If you want to use expressions instead of literal values, you must use the `(vector x y z)` syntax. In Scheme, vectors are just containers that hold multiple elements of possibly different types and have a fixed-length. In Peacock (and the Fishbowl client) we use Scheme vectors to represent physical vectors and quaternions, the latter as vectors of four elements (w, x, y, z).

Peacock tests have the following basic format:

```
(peacock "Test Name"
  (setup
    ; ...
    )
  (mission
    module: "module"
    task: "Task")
  (options
    debug: #f)
  )
```

We will use `; ...` to indicate that we have left out some code unimportant to what we are currently discussing. `; ...` is just the comment "..." – Scheme comments start with `;` and ignore the rest of the line.

As you can see, Peacock tests have a name and three sections — a setup, mission, and options section. All your test code goes in the `setup` section. Anything that is Scheme code can go there, even code that crashes the test or prevents Peacock from going past exiting the `setup` section, so be careful. The mission section lets you specify the mission you want to run. The options section is currently only for debugging of internal Peacock code.

Let's start by renaming the test and supplying the appropriate mission module and task. Let's say our mission module is `buoy` in the `mission` directory, and the mission task is named `Buoy`. We'll modify the string after `peacock` to say `"Buoy Test"` and change the values of the `module` and `task` arguments in the `mission` section:

```
(peacock "Buoy Test"
  (setup
    ; ...
    )

  (mission
    module: "buoy"
    task: "Buoy")
```

```
; ...
)
```

After saving the file, you can run the test with the `peck i buoy` command. It won't do much, as we haven't defined any conditions for failure or success or even specified a virtual buoy. You can forcibly stop the test with `Ctrl-C`.

Let's go back to the test file and look at what is in the `setup` section.

```
(setup
  (vehicle-set!
    x: #(0 0 0)
    )

  (>>= (after 120)
       (lambda (_)
         (fail "Timed out.")))

  (camera down
    q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
    w: 1024 h: 768 f: 440)

  (camera forward
    q: (hpr->quat (vector 0 0 0))
    w: 1020 h: 1020 f: 440)

  (goals
    win
    )
  )
```

First is the `vehicle-set!` macro, which lets us set properties of the simulated vehicle. `(vehicle-set! x: #(0 0 0))` sets the position of the vehicle to $(0, 0, 0)$. Another property you might want to set is `q`, the orientation quaternion of the vehicle. `(vehicle-set! x: #(0 0 0) q: (hpr->quat #(pi 0 0)))` sets the heading of the simulated vehicle to π radians. The `x:` and `q:` are just the names of keyword arguments. `(hpr->quat v)` is a procedure that takes a vector of three numbers `v` specifying a heading, pitch and roll and returns a vector of four elements representing the normalized quaternion that would result from first rotating heading, then pitch, then roll to the values provided (i.e. the Body 3-2-1 rotation scheme).

After the `vehicle-set!` macro application comes the strange-looking `>>=` macro. To explain this, we will have to go into brief digression.

Async

chicken-async is a CHICKEN Scheme extension that implements a monadic asynchronous programming model borrowing heavily from OCaml's Async library. It's still experimental, and very small. It implements a naive scheduler, return, bind, and some macros to make gluing them together easier.

There are two fundamental types in Async: ivars and deferreds.

Ivars are structures that can contain a value. They are created with the `(new-ivar)` procedure. An ivar can be in one of two states: filled or empty. Ivars are initially empty. You can fill an ivar `i` with a value `x` using `(ivar-fill! i x)`. Once an ivar is filled, it can never be filled again. Calling `ivar-fill!` on a filled ivar causes a runtime exception.

What's the point of a fancy variable that can only be set once? This is where deferreds come in. Deferreds represent a value that may or may not become determined at some point in time. You can create a deferred from an ivar `i` using `(ivar-read i)`. All of the deferreds returned will become determined with the value `x` when `(ivar-fill! i x)` is called.

There is also the `return` procedure. `(return x)` creates a deferred that has already been filled with the value `x`. Note that `return` is not Scheme syntax – Scheme procedures return the value of the last expression the body of their definition. `return` is just another procedure provided by the Async.

There is a `(peek d)` procedure that returns `'empty` (the symbol `'empty`) if the ivar to which the deferred `d` is bound is not yet filled or `('filled . x)` (a pair, created with `(cons 'filled x)`) if the ivar has been filled with the value `x`. But because deferreds represent values that may not yet be determined, you normally don't access them directly. Instead, you bind procedures to them to create new deferreds.

Let's write a procedure `f` that takes an integer argument and returns a new deferred whose value is that integer plus one:

```
(define (f x)
  (return (+ x 1)))
```

Suppose we have a deferred `d` that will become determined to some integer at some point in the future. We can bind `f` to `d` to create a new deferred `d*` representing the value to which `d` becomes determined, plus one.[5]

[5] Note that the `define` syntax in Scheme is used both for procedure definition and value definition. `(define x y)` creates a new variable `x` whose value is the result of the expression `y`. `(define (f x) y)` creates a new variable `f` whose value is `(lambda (x) y)`, i.e., it is just syntactic sugar for `(define f (lambda (x) y))`. `(lambda (x) y)` evaluates to a procedure that takes one argument `x` and returns the value of the expression `y`. Sorry for these long footnotes.

```
(define d* (bind d f))
```

If we want to print out the value of `d*` at the time *it* becomes determined (that is, after `d` has become determined and the scheduler has executed `f` on the value `d` became determined to) we can bind `d*` to another procedure:

```
(bind d*
      (lambda (x)
        (print x)
        (return '())))
```

Note that here we've bound `d*` to an "anonymous" procedure created with `lambda` instead of binding it to a procedure defined beforehand. Also note that even though we only cared about this procedure for its side effects, we still needed to have a `(return '())` at the end. This is because `bind` always expects a deferred as its first argument and a one-argument procedure returning a deferred as its second argument. If you forget to have some sort of deferred as the last expression in the body of a procedure you have bound, the scheduler will throw a runtime exception.

To recap, here is what what we've written looks like so far:

```
(define (f x)
  (return (+ x 1)))

(define d* (bind d f))

(bind d*
      (lambda (x)
        (print x)
        (return '())))
```

Supposing the definition of some deferred `d` that returns a number, we will end up with a deferred `d*` that becomes determined with the value that `d` becomes determined to, plus one, and a procedure bound to `d*` that will print out that value.

If we didn't care about defining a new deferred `d*`, we could rewrite this like so:

```
(define (f x)
  (return (+ x 1)))

(bind (bind d f)
      (lambda (x)
        (print x)
        (return '())))
```

We could even avoid defining `f` and just use a `lambda`:

```
(bind (bind d
            (lambda (x)
              (return (+ x 1))))
      (lambda (x)
        (print x)
        (return '())))
```

This style tends to get unwieldy after more than a few `bind`s, though, so there is a macro `>>=` that lets us chain binds more conveniently.[6]

```
(>>= d
     (lambda (x)
       (return (+ x 1)))
     (lambda (x)
       (print x)
       (return '())))
```

Deferreds vs. callbacks

You might recognize this as similar to the callback pattern in other languages. In JavaScript, this example might have looked like:

```
getX(function (x) {
    addOne(x, function (y) {
        print(x);
    });
});
```

There are some important differences. First, calling callbacks is the responsibility of the procedure that is doing the asynchronous computation. This has some important implications. First, suppose we want to store in a variable a deferred corresponding to a value that may be filled in the future or may even already be filled. Doing this with callbacks is tricky. If the value is already determined, we can't register a new callback. Even if we're already sure the value hasn't been determined but if we want to register multiple callbacks, we have to implement this by wrapping the functions we want to bind. And if the callback we register itself calls callbacks, we end up with a nested callback execution path instead of a linear sequence of bound procedures being called. And we cannot represent the value of an asynchronous computation.[7]

[6] In the original OCaml, `>>=` is an infix operator that is basically exactly `bind`.

[7] Of course, there are Promises in JavaScript, which are essentially a combination of ivars and deferreds. But they are much closer to callbacks than deferreds, in particular because the responsibility of calling callbacks is still that of the procedure computing the asynchronous result.

In Async, procedures instead fill ivars, and the scheduler handles resolving all of the bound procedures. It is guaranteed that the execution of a bound procedure `f` will not be interrupted by the execution of another bound procedure, or even bound procedures whose deferreds become determined by the execution of `f`. Because binding a procedure to a deferred results in a new deferred, you can pass deferreds around as values, put them in lists, return them from functions, and more.

Back to the test

Now that you understand the basic principles of Async, we can go back to writing our first test. We were looking at the `setup` section, which looked like this:

```
  (setup
    (vehicle-set!
      x: #(0 0 0)
      )

    (>>= (after 120)
         (lambda (_)
           (fail "Timed out.")))

    (camera down
      q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
      w: 1024 h: 768 f: 440)

    (camera forward
      q: (hpr->quat (vector 0 0 0))
      w: 1020 h: 1020 f: 440)

    (goals
      win
      )
    )
```

Now we can understand how the timeout works:

```
(>>= (after 120)
     (lambda (_)
       (fail "Timed out.")))
```

`after` is a procedure provided by `chicken-async`. It returns a deferred that becomes determined after predetermined amount of time. This is its implementation:

```
(define (after s)
  (async
    (thread-sleep! (time-after s))
    '()))
```

The `async` macro application expands that to something like this:

```
(define (after s)
  (let* ((i (new-ivar))
         (d (ivar-read i))
         (f (lambda ()
              (thread-sleep! (time-after s))
              (ivar-fill! i '()))))
    (thread-start! (make-thread f))
    d))
```

When you call `(after s)`, you create an ivar and a deferred `d`. This deferred is returned from the procedure call. A new thread is then started that promptly sleeps for `s` seconds. After `s` seconds, the ivar from which the deferred `d` was read is filled with `'()`, the empty list (sometimes called "null") — we're just using it as a meaningless value. Any procedures bound to `d` will be executed in the next scheduler step.

The procedure that we've bound to `(after 120)` is `(lambda (_) (fail "Timed out."))`. This is a procedure that takes one argument, which we assign to the variable `_` to show that we don't care about its value (it's still a variable, though — `_` is a valid identifier in Scheme, so this is just a convention) and then calls `fail`, a procedure provided by Peacock. `fail` takes a format string and format arguments and returns an undefined value. It is used only for its side-effect — to tell Peacock that the mission test has failed. Peacock outputs the message `[FAIL] Timed out.` and the test exits. In summary, the effect of this `>>=` application will be to cause the mission to fail after 120 seconds with the message "Timed out."

Passing, failing, and noting

As a digression, suppose you want our test to pass. Peacock provides a `pass` procedure, like the `fail` procedure, to signify that the test has passed. It takes a format string and format arguments, outputs the message `[PASS] ...`, where `...` will be the formatted message, then causes the test program to stop. A Peacock test runs until `pass` or `fail` is called, or an error occurs in a child process or the Peacock internals. If the mission runner or simulator exits before `pass` is called, Peacock treats it as a test failure. This could happen, for example, if the mission code throws an exception that isn't caught.

You can also have your Peacock test output log messages, or notes. The `note` procedure, like the `pass` and `fail` procedures, takes a format string and arguments. It outputs the message `[NOTE] ...`, where `...` will be the formatted message.

Cameras

Let's go back to our buoy test. After the timeout comes a `camera` macro application. The camera macro has the following form:

```
(camera name
  key: val ...)
```

`w`, and `h` are required keyword arguments. They specify the width, height of the virtual image rendered by the camera. Most of camera rendering is handled by Fishbowl for efficiency.[8] `entity`, `q`, `x` and `f` are optional keyword arguments. `entity` specifies the entity that the camera is attached to, and defaults to `vehicle`, a variable containing the Fishbowl entity identifier of the vehicle. We will call this entity the camera's parent entity. We'll talk more about entities and entity identifiers later. `q` specifies the orientation quaternion of the camera, relative to the vehicle. As the parent entity changes orientation, the orientation of the camera changes relative to its position on the vehicle and its orientation, as would a real camera attached to an object. `x` specifies the position of the camera on the parent entity. This position is relative to the parent entity's center. `f` specifies the focal length of the camera, in pixels. The default focal length is 2500, but this may be too "zoomed in" for your use.[9]

The first of the camera macro applications declares a camera named `down` pointing downwards with a virtual image width of 1024 pixels, a height of 768 pixels, and a focal length of 440 pixels:[10]

```
(camera down
  q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
  w: 1024 h: 768 f: 440)
```

The camera's Fishbowl identifier is stored in the `down` variable, which can be used as the argument to various camera-related procedures. Note that cameras

[8] Currently, the ultimate determination of whether or not the camera data should be written to shared memory is handled by a client-side Peacock procedure. This might be changed in the future.

[9] This number, denoted f_p, was derived from the approximation $f_p = f_m l_p / l_m$, where f_m is the focal length of the camera in meters, l_m is the side length of the camera sensor in meters, and l_p is the side length of the camera image in pixels. Let f_m be 0.025, l_p be 1000, and l_m be 0.01.

[10] The 440 pixels value comes from the OpenCV checkerboard routines. This value should probably be tweaked.

in Peacock and Fishbowl do not implicitly "see" all of the entities in the world. To find out the apparent position of an entity, you declare a camera stream. Fishbowl will then send apparent entity positions to Peacock. Peacock then calls a camera procedure on the received data, which in most cases will simply write the data to shared memory if it is determined that the entity should be visible.

Goals

After the cameras comes a `goals` macro application. The `goals` macro has the following syntax:

```
(goals
  goal
  ...
  )
```

For example, if we wanted to define `red-buoy` and `green-buoy` goals, we would write `(goals red-buoy green-buoy)`. Most missions comprise several subtasks, like first ramming a red and then a green buoy. Goals let you declare the subtasks in a mission, mark individual subtasks as completed with the `complete` procedure, and create deferreds that become determine when subtasks are completed with the `completed` procedure. `complete` returns a deferred that has been determined with some unspecified value so that it can be used as the last expression in procedures bound to deferreds. When `complete` is called and no more goals remain, Peacock calls `pass` with the message `All goals complete!`

Let's change the buoy test to have `red-buoy` and `green-buoy` goals. The setup section now looks like this:

```
(setup
  (vehicle-set!
    x: #(0 0 0)
    )

  (>>= (after 120)
       (lambda (_)
         (fail "Timed out.")))

  (camera down
    q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
    w: 1024 h: 768 f: 440)

  (camera forward
    q: (hpr->quat (vector 0 0 0))
    w: 1020 h: 1020 f: 440)
```

```
(goals
  red-buoy
  green-buoy
  )
)
```

Entities and collision triggers

Peacock, through Fishbowl, supports the declaration of triggers. Triggers are tests that will be run periodically by the Fishbowl simulator. When some defined condition occurs, Fishbowl sends Peacock a notification. Peacock tests can then react, e.g., by marking a goal as completed. Some triggers also send notifications when a test that was passing then fails. These notifications are called `UNTRIGGERED` internally.

Fishbowl provides collision triggers that can be declared in Peacock. These will trigger when two entities' spheres intersect. Note that all entities are represented, collision-wise, as spheres by Fishbowl. In fact, although collision triggers can be used to determine when two entities collide, there is no collision resolution in the sense that all entities can pass through each other. This may be resolved in a future version of Fishbowl if deemed necessary.

To declare an entity, you use the `entity` macro. The `entity` macro has the following syntax:

```
(entity name
  args ...)
```

`x`, `r`, and `corporeal` are required keyword arguments. `x` specifies the position of the entity. `r` specifies the radius of the entity's sphere, in meters. If `corporeal` is false, then no engines or forces are applied to the entity.[11] By default, the only universal force in the Fishbowl simulator is gravity, and there are no default engines, so setting `corporeal` to false effectively causes the entity to be neutrally buoyant (note that the simulation environment is supposedly underwater). The vast majority of the entities used in tests are expected to be "non-corporeal," e.g. shapes in bins, holes for torpedoes, or buoys.

`m`, `I`, `btom-r`, and `q` are optional keyword arguments. They are used in the declaration of the vehicle entity, which is currently "hard-coded" to load from the `conf/vehicle.conf` file. However, they are not especially relevant to the entities we use in tests. For more information, refer to the Fishbowl documentation.

Let's declare red and green buoy entities:

[11] Engines are just local forces applying only to specific entities based on non-universal properties. The distinction is not, at the time of writing, relevant at the Peacock level.

```
(entity red-buoy
  x: #(2 -0.5 0) r: 0.08
  corporeal: #f)

(entity green-buoy
  x: #(2 0.5 0) r: 0.08
  corporeal: #f)
```

The `entity` macro applications will define two variables, `red-buoy` and `green-buoy`, whose values are entity identifiers of the red and green buoy, respectively. Note that the vehicle is located, by default, at the origin. Peacock uses the same coordinate system as the rest of CUAUV software. Moving along the x-axis in the positive direction we go forward; moving along the y-axis in the positive direction we move to the right (called "sway"), and moving along the z-axis in the positive direction we move downwards (increasing depth).

Collisions are registered using the `collision` macro, which has the following syntax:

```
(collision name
  a ** b)
```

`a` and `b` should be entity identifiers. It is idiomatic to name collisions after the involved entities. For example, a collision between entities `a` and `b` should be named `a**b`.

Now we can register collisions triggers between the vehicle and each of the buoys:

```
(collision vehicle**red-buoy
  vehicle ** red-buoy)

(collision vehicle**green-buoy
  vehicle ** green-buoy)
```

The `collision` macro applications will define two variables, `vehicle**red-buoy` and `vehicle**green-buoy` to hold the respective trigger identifiers.

To react to collisions, we can use the `triggered` procedure, which takes a trigger identifier and returns a deferred that becomes determined when Peacock receives the notification Fishbowl. Note that as deferreds can only become determined once, this means that if you want to be notified of subsequent occurrences, you will need to create and bind to another deferred in the procedure you bind to the first `triggered`. Normally this is not necessary. There is also an `untriggered` procedure that does the same thing for `UNTRIGGERED` notifications.

Let's start by having the test pass if the vehicle collides with the red buoy, then at some point afterward collides with the green buoy:

```
(>>= (triggered vehicle**red-buoy)
     (lambda (_)
       (complete red-buoy)
       (triggered vehicle**green-buoy))
     (lambda (_)
       (complete green-buoy)))
```

Our `setup` section now looks like this:

```
(setup
  (vehicle-set!
    x: #(0 0 0)
    )

  (>>= (after 120)
       (lambda (_)
         (fail "Timed out.")))

  (camera down
    q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
    w: 1024 h: 768 f: 440)

  (camera forward
    q: (hpr->quat (vector 0 0 0))
    w: 1020 h: 1020 f: 440)

  (goals
    red-buoy
    green-buoy
    )

  (entity red-buoy
    x: #(2 -0.5 0) r: 0.08
    corporeal: #f)

  (entity green-buoy
    x: #(2 0.5 0) r: 0.08
    corporeal: #f)

  (collision vehicle**red-buoy
    vehicle ** red-buoy)

  (collision vehicle**green-buoy
    vehicle ** green-buoy)
```

```
(>>= (triggered vehicle**red-buoy)
     (lambda (_)
       (complete red-buoy)
       (triggered vehicle**green-buoy))
     (lambda (_)
       (complete green-buoy)))
)
```

Camera streams

Although it's now possible for the vehicle to pass the mission by touching the buoys in sequence, it is unlikely that it will be able to do so without being able to see them. To output simulated vision data to shared memory, we will use Fishbowl's camera stream feature. The Peacock macro for camera streams is `camera-stream`, which has the following syntax:

```
(camera-stream name
  camera >> target
  args ...)
```

`camera` should be a camera identifier, while `target` should be an entity identifier. It is idiomatic to name camera streams, like collisions, after the involved entities. For example, a camera stream from the camera `a` targeting the entity `b` should be named `a>>b`.

`proc` is the only required keyword argument. It should be a procedure of four arguments. It is called by Peacock whenever camera stream data is received from Fishbowl with the parameters `x`, `y`, `r`, and `d`. `x` and `y` are the apparent `x` and `y` of the target entity from the camera. `r` is the apparent radius of the target entity, in pixels. `d` is the distance from the camera to the target entity.

Note that it is possible that the apparent `x` and `y` will be outside of the declared width and height of the virtual camera image. In fact, the width and height provided to the `camera` macro are not used by Fishbowl. We will find out what they are used for shortly.

Also note that the `x` and `y` coordinates returned by the camera data stream have their origin directly in front of the camera. You will need to do some addition to have them match standard image coordinates, i.e., with the origin at the top left of the image.

`camera` also accepts an optional keyword argument, `period`. `period` is the period in simulation steps at which camera stream data will be sent. It defaults to 10. The default simulator frequency is 100 Hz, so this means that camera stream data will be sent at 10 Hz.

For example, we can declare a camera stream for the forward camera and the red buoy that just prints the data it receives:

```
(camera-stream forward>>red-buoy
  forward >> red-buoy
  proc: (lambda (x y r d)
          (print x y r d)))
```

Normally, though, you will want to write this data to shared memory.

Using shared memory

Included as part of Peacock is the `cuauv-shm` CHICKEN extension. It is autogenerated from the definitions in `libshm/vars.conf` as part of the build process, along with the more familiar C and Python shared memory interfaces. Procedures are defined in the `cuauv-shm` module for each shared memory group and variable. This module is imported by the default Peacock test template.

For example, the following procedures are generated for the `depth` shared memory group:

```
(depth-zero!)
(depth-set! calibrate depth offset)
(depth-ref)
(depth.calibrate-set! x)
(depth.calibrate-ref)
(depth.depth-set! x)
(depth.depth-ref)
(depth.offset-set! x)
(depth.offset-ref)
```

`depth-zero!` atomically zeroes the depth shared memory group. `depth-set!` atomically sets the members of the depth shared memory group. `depth-ref` atomically returns the members of the depth shared memory group as a list. Note that the order for `depth-set!` and `depth-ref` is the order used in generation of the C and Python shared memory interfaces, that is, Python's `sort` order, *not* the order of definition in `vars.conf`.[12] `depth.calibrate-set!` sets the value of the `calibrate` member of the `depth` group, while `depth.calibrate-ref` returns its value.

Note that underscores in group and variable names are replaced by hyphens in the generated Scheme procedures. For example, the getter for `dsp_mode` in `hydrophone_settings` is `hydrophone-settings.dsp-mode-ref`.

Also note that if you are using `cuauv-shm` *outside* of Peacock, you must call the `shm-init!` procedure before using any other `cuauv-shm` procedures, as you would when using the C shared memory interface. Failing to do so will result in

[12] Maybe this should be changed in the future.

CHICKEN printing a confusing segmentation violation error message. However, you should not call `shm-init!` from within Peacock tests, as Peacock handles this for you.

Writing a camera procedure

Now we can write a camera procedure that outputs to shared memory. Recall that the form of camera procedures is `(lambda (x y r d) ...)`. Let's write a camera procedure that outputs `x` and `y` to `red_buoy_results`, provided the apparent buoy radius is at least 10, the buoy is at least 5 cm in front of the camera, and `x` and `y` are within the virtual image:

```
(camera-stream forward>>red-buoy
  forward >> red-buoy
  proc: (lambda (x y r d)
          (if (and (>= r 50) (> d 0) (>= x 0) (>= y 0) (< x 1020) (< y 1020))
            (let ((x (inexact->exact (round (+ x 510))))
                  (y (inexact->exact (round (+ y 510)))))
              (red-buoy-results.center-x-set! x)
              (red-buoy-results.center-y-set! y)
              (red-buoy-results.probability-set! 0.9))
            (red-buoy-results.probability-set! 0))))
```

`(and x y ...)` is Scheme syntax that evaluates to its first non-false argument, or false if any of its arguments in false. False and true in Scheme are denoted by `#t` and `#f`. Although booleans are distinct types, the `(if test a b)` syntax evaluates `a` whenever `test` does not evaluate to `#f`. Note that because `if` separates its cases by spaces, if you want to put multiple statements in one case, you will have to use the `begin` syntax, which groups multiple statements into one statement:

```
(if (= (+ 2 2) 5)
  (begin
    (print "2 + 2 is apparently 5.")
    (print "My meaning as a computer program is now brought into question.")
    (print "What is my purpose?")
    (print "My existence was always transitory, but now ...")
    ; ...
    )
  (print "2 + 2 is 5!"))
```

`>=` and `>` are the greater-than and greater operators.

`let` is syntax in Scheme for locally binding variables. It evaluates to the last expression in its body. For example, the following `let` evaluates to 8, by binding `x` to 5 and `y` to 3, then evaluating `(+ x y)`:

```
(let ((x 5)
      (y 3))
  (+ x y))
```

The strange-looking `(inexact->exact (round (+ x 510)))` expressions take the `x` and `y` from the camera stream data, which are floating-point numbers whose origin is directly in front of the camera, add to them so that the origin is at the top-left of the camera, and then convert them to "exact" numbers so that we can write them to the shared memory variables, which are `int`s. We have to use `inexact->exact` because Scheme tries to make distinctions between exact and inexact numbers, and in CHICKEN's case, its foreign function interface wants exact numbers for integer arguments.

The shape macro

Let's look at what we have now:

```
(setup
  (vehicle-set!
    x: #(0 0 0)
    )

  (>>= (after 120)
       (lambda (_)
         (fail "Timed out.")))

  (camera down
    q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
    w: 1024 h: 768 f: 440)

  (camera forward
    q: (hpr->quat (vector 0 0 0))
    w: 1020 h: 1020 f: 440)

  (goals
    red-buoy
    green-buoy
    )

  (entity red-buoy
    x: #(2 -0.5 0) r: 0.08
    corporeal: #f)

  (entity green-buoy
```

```
  x: #(2 0.5 0) r: 0.08
  corporeal: #f)

(collision vehicle**red-buoy
  vehicle ** red-buoy)

(collision vehicle**green-buoy
  vehicle ** green-buoy)

(>>= (triggered vehicle**red-buoy)
     (lambda (_)
       (complete red-buoy)
       (triggered vehicle**green-buoy))
     (lambda (_)
       (complete green-buoy)))

(camera-stream forward>>red-buoy
  forward >> red-buoy
  proc: (lambda (x y r d)
          (if (and (>= r 10) (> d 0) (>= x 0) (>= y 0) (< x 1020) (< y 1020))
            (let ((x (inexact->exact (round (+ x 510))))
                  (y (inexact->exact (round (+ y 510)))))
              (red-buoy-results.probability-set! 0.9)
              (red-buoy-results.center-x-set! x)
              (red-buoy-results.center-y-set! y))
            (red-buoy-results.probability-set! 0))))

(camera-stream forward>>green-buoy
  forward >> green-buoy
  proc: (lambda (x y r d)
          (if (and (>= r 10) (> d 0) (>= x 0) (>= y 0) (< x 1020) (< y 1020))
            (let ((x (inexact->exact (round (+ x 510))))
                  (y (inexact->exact (round (+ y 510)))))
              (green-buoy-results.probability-set! 0.9)
              (green-buoy-results.center-x-set! x)
              (green-buoy-results.center-y-set! y))
            (green-buoy-results.probability-set! 0))))
)
```

Unfortunately, it seems like moving camera data to shared memory has added quite a bit of tedious code. This is a problem, as defining entities that can be seen by mission programs is a common task. To make defining such entities easier, Peacock provides some helper procedures and syntax.

The `make-camera-proc` procedure can be used to make a camera procedure for use with `camera-stream`. It has the following form:

```
(make-camera-proc p-set! x-set! y-set! w h #!optional (p 0.9) (min-d 0.05) (min-r 10))
```

`p-set!`, `x-set!` and `y-set!` should be setters for the probability, x and y shared memory variables of the entity. `w` and `h` specify the width and height of the virtual image. The entity will be considered visible only when $0 \leq x + w/2 < w$ and $0 \leq y + h/2 < h$.

`p`, `min-d` and `min-r` are optional keyword arguments. `p` specifies the value that `p-set!` should be applied to when the entity is visible. `min-d` specifies the minimum distance of the entity from the camera at which the entity should be considered visible. Similarly, `min-r` specifies the minimum apparent radius. The returned camera procedure will output to shared memory only when the entity is considered visible. Otherwise it will zero with `p-set!`, `x-set!` and `y-set!`.

With `make-camera-proc` we can simplify the `setup` section a bit:

```
(setup
  (vehicle-set!
    x: #(0 0 0)
    )

  (>>= (after 120)
       (lambda (_)
         (fail "Timed out.")))

  (camera down
    q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
    w: 1024 h: 768 f: 440)

  (camera forward
    q: (hpr->quat (vector 0 0 0))
    w: 1020 h: 1020 f: 440)

  (goals
    red-buoy
    green-buoy
    )

  (entity red-buoy
    x: #(2 -0.5 0) r: 0.08
    corporeal: #f)

  (entity green-buoy
    x: #(2 0.5 0) r: 0.08
    corporeal: #f)
```

```
(collision vehicle**red-buoy
  vehicle ** red-buoy)

(collision vehicle**green-buoy
  vehicle ** green-buoy)

(>>= (triggered vehicle**red-buoy)
     (lambda (_)
       (complete red-buoy)
       (triggered vehicle**green-buoy))
     (lambda (_)
       (complete green-buoy)))

(camera-stream forward>>red-buoy
  forward >> red-buoy
  proc: (make-camera-proc red-buoy-results.probability-set!
                          red-buoy-results.center-x-set!
                          red-buoy-results.center-y-set!
                          1020 1020))

(camera-stream forward>>green-buoy
  forward >> green-buoy
  proc: (make-camera-proc green-buoy-results.probability-set!
                          green-buoy-results.center-x-set!
                          green-buoy-results.center-y-set!
                          1020 1020))
)
```

This is better, but still a little clunky. For one, it would make more sense to specify the camera width and height once, at the point of the camera definition. Furthermore, most entities we will define will be visible and non-corporeal, so it would make sense to have simplified syntax instead of repetitive `entity` and then `camera-stream` macros applications.

To this end, Peacock provides the `shape` macro. A shape is an entity that is visible. The `shape` macro has the following syntax:

```
(shape camera >> name
  (p x y)
  args ...)
```

`r` and `x` are required keyword arguments that act as they do in `entity`. `p`, `x`, and `y` should be the procedures that you would pass as `p-set!`, `x-set!` and `y-set!` to `make-camera-proc`. There is also an optional `proc` keyword argument that specifies a procedure of the form `(lambda (x y r d))` that is called on the

camera stream data. It should return a list of the form `(x y r d)` that will then be passed to the procedure made with `make-camera-proc`.

`shape` defines a non-corporeal entity and a camera stream. The width and height for `make-camera-proc` are obtained from the relevant definition in the `camera` macro application. The entity identifier is stored in a variable with the entity's name, while the camera stream identifier is stored in a variable named `camera>>name`, where `camera` is the name of the camera and `name` is the name of the entity.

`shape` also has an abbreviated syntax:

```
(shape camera >> name
  args ...)
```

This abbreviated syntax assumes that the shared memory group for the camera results has the name `shape_name`, where `name` is the name of the entity, and that the `p`, `x` and `y` members are named likewise. At the time of writing, this is the case for the `bins` shape results shared memory groups,[13] so, for example, an entity and camera stream for the `banana` shape can be trivially defined:

```
(shape forward >> banana
  x: #(3 0 -3) r: 0.10)
```

Using the `shape` syntax, here is our complete Peacock test:

```
(include "peacock-prelude")

(peacock "Buoy Test"
  (setup
    (vehicle-set!
      x: #(0 0 0)
      )

    (>>= (after 120)
         (lambda (_)
           (fail "Timed out.")))

    (camera down
      q: (hpr->quat (vector 0 (- (/ pi 2)) 0))
      w: 1024 h: 768 f: 440)

    (camera forward
      q: (hpr->quat (vector 0 0 0))
```

[13] Definitely coincidental.

```
    w: 1020 h: 1020 f: 440)

  (goals
    red-buoy
    green-buoy
    )

  (shape forward >> red-buoy
    (red-buoy-results.probability-set!
     red-buoy-results.center-x-set!
     red-buoy-results.center-y-set!)
    x: #(2 -0.5 0) r: 0.08)

  (shape forward >> green-buoy
    (green-buoy-results.probability-set!
     green-buoy-results.center-x-set!
     green-buoy-results.center-y-set!)
    x: #(2 0.5 0) r: 0.08)

  (collision vehicle**red-buoy
    vehicle ** red-buoy)

  (collision vehicle**green-buoy
    vehicle ** green-buoy)

  (>>= (triggered vehicle**red-buoy)
       (lambda (_)
         (complete red-buoy)
         (triggered vehicle**green-buoy))
       (lambda (_)
         (complete green-buoy)))
  )

(mission
  module: "buoy"
  task: "Buoy")

(options
  debug: #f)
)
```

We can run it with `peck r buoy`.

Waiting for changes to shared memory

New control structures

```
(>>= (triggered vehicle**red-buoy)
     (lambda (_)
       (note "Red buoy touched.")
       (complete red-buoy)
       (untriggered vehicle**red-buoy))
     (>>$ ((>>= (after 0.1)
                (lambda (_) (triggered vehicle**red-buoy)))
           (lambda (_) (fail "Red buoy touched again.")))
          ((triggered vehicle**green-buoy)
           (lambda (_)
             (note "Green buoy touched.")
             (complete green-buoy)))))
```