Introduction

The hive system is a system for programming with connected components (nodes). It be used visually or from Python. It can be used with any library or game engine with Python support: currently it has bindings for the Blender Game Engine (BGE) and Panda3D.

The first chapter of the tutorial gives a gentle introduction to the hive system and it concepts. It is mostly at the GUI level, with a bit of Python mixed in.
The second chapter of the tutorial describes a case study: the drawing of 2D shapes.
- The first part is GUI-oriented, showing how to use various implementations for shape-drawing.
- The second part is Python-oriented, showing how the various implementations are created.

The third chapter of the tutorial is a detailed explanation of the various layers (visual and Python) in the hive system and how they are relate to each other.

Non-programmers are recommended to start with the first chapter of the tutorial together with the Tetris demo.
Programmers are encouraged to start with the third chapter of the tutorial.

Chapter 1: Hello World

Chapter 1, section A: Building a hello world hivemap using HiveGUI
Python or visual? Visual
Difficulty: Easy

What to know first
The hive system interacts with the rest of the world through a *top-level hive*. Top-level hives are: blenderhive for the Blender Game Engine, pandahive for Panda3D, and consolehive for command-line key presses. Inside a top-level hive, a *hivemap* is launched. The hivemap contains the nodal logic, and is visually editable with the *HiveGUI*.
The hive system has a *default empty project* that you can download. It contains an empty hivemap, and three Python scripts that launch the hivemap inside a different top-level hive.
In this tutorial chapter, we will modify the default project hivemap to print “Hello World” and a few other things. The finished code for the tutorial can be downloaded as well.

How to get started
- Follow the installation instructions (installation.pdf)
- Download the finished tutorial code at https://launchpad.net/hivesystem/+download. You will find the current chapter in “helloworld/”.
- *Check if you can run the finished tutorial code under the consolehive*. Go to “helloworld-A1” and start “defaultproject.py”.
Under Linux, this means typing “python defaultproject.py”.
Under Windows, double-clicking on the “defaultproject.py” icon should normally work. If it doesn't, make sure that “RClick => Open with” points to C:\PythonXX\python.exe. Alternatively, python 3.3 and later provides a nice “Python Launcher for Windows (console)”, you will find it in C:\Windows\py.exe. If all else fails, click on Start, run “cmd.exe”, use cd to go to the correct folder and type “C:\Python27\python.exe defaultproject.py” or “py defaultproject.py” or “py -2 defaultproject.py” or “py -3 defaultproject.py”.
In all cases, you should see a window appear with “Hello World!”. Press Esc to stop it.
- *Check if you can run the finished tutorial code under the BGE*. Go to “helloworld-A1” and open “defaultproject.blend” with Blender. If it isn't visible yet, enable the Blender system console with “Window => Toggle System Console”. Go to “Game => Start Game Engine”. After a few seconds, in the 3D window, the default Blender cube will turn white, indicating that the Blender Game Engine is running. Check the system console, you should see “Hello World” printed. Click on the 3D window and press Esc to stop it.
- *Or: Check if you can run the finished tutorial code under Panda3D*. Follow the installation instructions for Panda3D above. Go to “helloworld-A1” and start “defaultproject-panda.py”. You will see an empty gray Panda3D window, and “Hello World” printed in a text window. Click on the empty gray Panda3D window and press Esc to stop it.

Exploring the HiveGUI
Download the default empty hive project at https://launchpad.net/hivesystem/+download and unzip it. This will be used as the starting point for the tutorial.

Start the HiveGUI. Under Windows, you can find it under “Start => All programs => Hive system => hivegui”. In Linux, simply type “hivegui.py”. Go to the copied defaultproject folder and open “defaultproject.hivemap”.

The HiveGUI screen consists of three areas. In the middle, there is the canvas area that displays the nodes and their connections. On the right, the node parameters are shown. On the left, there is a tree-like listing of all nodes that you can drag onto the canvas.

Nodes are visual representations of *bees*. Bees come in many flavors, but the most important ones are *workers*, that can be connected, and *drones*, which are unconnected, standalone units. In general, a collection of bees is called a *hive*. A visual collection of nodes in a GUI is called a *hivemap*. Hives (and hivemaps) can also be embedded as bees inside other hives (and hivemaps).

The HiveGUI discovers the tree of available nodes at startup. It inspects Python modules and directories for bees and hivemap files. In the “Workers” area, you can see two trees: “bees” and “dragonfly”. “bees” will not be used in this tutorial chapter, but “dragonfly” is the hive system's standard node library. *If you are a Python programmer: dragonfly is a normal Python module, which you can import in the console. Its source code may be worthwhile for you to read. More details are given in Chapter 3.*
HiveGUI will also discover any custom workers, hives and hivemap that you create in the “workers/” folder. We will do this in section C.

Part A1: Hello World
In the Workers area, open the “dragonfly” tree, then the “std” tree, and click the “variable_str” label. Keep the left mouse button pressed, and drag the label onto the canvas area. This will create a new worker node “variable_str_1”. Click on it, and in the “Props” area on the right, click on the “Parameters” tab. Under “value”, enter "Hello world!" (no quotes). Then, under dragonfly tree, open “io” and drag the “display_str” label onto the canvas.
Now we have two workers, a dragonfly.std.variable_str worker called “variable_str_1”, and a dragonfly.io.display_str worker called “display_str_1”. If you like, you can rename the workers in the General tab.
Click on the green diamond on the right side of “variable_str_1”, keep the button pressed, and connect it to the green dot of “display_str_1”. In the pop-up menu, select “on change”. Normally, you can't connect diamonds and dots. To make the connection possible, the HiveGUI automatically creates a new dragonfly.std.sync worker for you. This will be explained below, but first save the hivemap, and run the hive inside Blender, Panda3D or the command line, as explained in the “How to get started” section. You will see “Hello World!” printed. Press Esc to exit. If you change the variable_str value to something else, you will see something else printed.
You can find the finished project in the “helloworld-A1” folder.

Push and pull
In the HiveGUI, *dots* represent *push* input (or output), and *diamonds* represent *pull* input (or output). It is very important to understand their difference. You can only connect push to push, and pull to pull. When you connected a pull to a push and clicked “on change”, a sync worker was automatically inserted. A sync worker activates itself every tick. Whenever it does, it *pulls* the current value of the

variable_str worker, and compares it to a stored value. If the current value has changed, it *pushes* it to the display_str worker, and then stores it. In this way, polling and notification can be achieved.

Part A2: responding to key presses

Click on variable_str_1, press Ctrl+C, and press Ctrl+V twice. This will create workers “variable_str_2” and “variable_str_3”.
Drag variable_str_2 and variable_str_3 to the left of variable_str_1.
If necessary, use the scroll wheel and the arrow keys to navigate the hive canvas
Click variable_str_2, click Parameters and change the value to "Bye world!"
Click variable_str_3, click Parameters and change the value to "Hello again world!"
Connect variable_str_2 to variable_str_1 and select Manual. This will auto-create a new worker called a *transistor*. It will be explained in the next section.
Create a worker *dragonfly.io.keyboardsensor_trigger* (dragonfly tree => io tree => keyboardsensor_trigger label; drag the label onto the canvas).
Select the new keyboardsensor_trigger_1, select "Parameters" and set the keycode to 1
Connect keyboardsensor_trigger_1 to transistor_1.trig (connect the red dot to the other red dot marked “trig” on the left side of the transistor).
Connect variable_str_3 to variable_str_1 and select Manual. This will create a second transistor.
Select keyboardsensor_trigger_1, press Ctrl+C and then Ctrl+V
Select the new keyboardsensor_trigger_2, set the keycode to 2, and connect it to transistor_2.trig
Save the hivemap, and run the hive. At startup, you will see "Hello world!" appear. Press 1 to see "Bye world"; pressing 1 repeatedly will have no effect. Press 2 to see "Hello again world"; pressing 2 repeatedly will have no effect.
You can find the finished project in the “helloworld-A2” folder.

The execution order

The hivemap illustrates the rules for execution order in the hive system. The following four rules can be observed:
Rule 1: Only one thing happens at a time. The workers are not executed simultaneously, but there is only one worker that has the *initiative*.
Rule 2: The workers pass the initiative to each other, via the push-push and pull-pull connections.
Rule 3: It matters *which connection* activates a worker. The variable_str_1 worker can do two very different things. When activated over its pull output connection (by sync_1), it simply returns its value. However, when it is activated over its push input connection (by transistor_1 or transistor_2), it updates its value instead.
Rule 4: Workers gain the initiative by detecting events. The keyboardsensor_trigger workers detect keypress events. The sync worker detects tick events (one tick event is fired on every frame).
In programming speak, the rules are as follows:

Rule 1: the hive system is single-threaded
Rule 2: workers call each other
Rule 3: every antenna and output is a different entry point for calls
Rule 4: workers are event-driven: they can register callbacks that trigger upon certain events

The execution sequence of the hivemap is then as follows:

On the first frame, the sync worker is activated by the tick event. It activates variable_str_1 on its pull output, which returns its value “Hello World!”. The sync worker compares it to its previously stored value, which is None. Since they are different, it pushes the value to the display_str worker, which displays it.
On a keypress event of key “1”, keyboardsensor_trigger_1 is activated. It activates transistor_1 on its “trig” antenna. A transistor takes pull input and pushes it out: therefore, variable_str_2 is activated on its pull output and returns its value “Bye World!”. The transistor pushes that value to variable_str_1 on its push antenna. Variable_str_1 then updates its value. No other things happen, but on the next tick event, the sync worker gets activated, detects a change, and pushes “Bye World!” to the display_str worker.
Upon the keypress of “2”, the same mechanism is activated, but with different workers and values.

Part A3: Controlling the execution order
Select the sync worker and press Delete to remove it.
Now again, click on the green diamond on the right side of “variable_str_1”, keep the button pressed, and connect it to the green dot of “display_str_1”. In the pop-up menu, this time, select “Manual”. This will create a new transistor, transistor_3.
Create a startsensor (dragonfly.sys.startsensor) and connect it to transistor_3. Then, connect both keyboard sensors to transistor_3. Save the hivemap and run the hive.
Press 1 to see "Bye world"; pressing 1 repeatedly will repeat the text
Press 2 to see "Hello again world"; pressing 2 repeatedly will repeat the text
Since there is no more sync worker, the hive now no longer detects tick events, so it works now more efficiently. All actions are now in response to the keypress events detected by the keyboard sensor (and the “start” event detected by the start sensor).
When the key “1” is pressed, two things happen: *first*, it activates transistor 1 (which pushes “Bye World”), and *then*, it activates transistor_3 (which prints it). Push output (a dot on the right side) is special, because this is the only case where the connection order matters. The hive system will *first* activate the connection that was created *first.* The HiveGUI indicates this by the *direction* of the connections. All output connections point to the right, but the first connection points the most upward, and the next one ever more downward. You can modify the connection order by hovering over the output dot, press TAB and use the 1,2,3 keys to indicate the position.
You can find the finished project in the “helloworld-A3” folder.

Part A4: Types and metaworkers
You may have noticed that the dots and diamonds have different colors: red and green. This is because they represent different *types*: red indicates *trigger*, and green indicates *str* (a Python string). You can only connect an output to an antenna of the same mode (push or pull) and type: the shape and color must be the same. If you hover over a worker antenna or output, you will see its mode and type indicated in the bottom left.
There are workers that can deal with many different types. This *dynamic typing* are an important feature of the Python language. For example, a worker that would take two inputs x and y, and adds them together with the Python expression “result = x + y”, could take two integers (adding 2 and 2 with result = 4) but it could also take two strings (adding “Hello” and “World” with result = “HelloWorld”).
To deal with such dynamically-typed cases, the hive system has so-called *meta-workers*. Meta-workers are nodes that take one or more meta-parameters, resulting in a normal worker. These meta-parameters

are usually the type name(s) of the data type(s) that the worker will process. We have used meta-workers already, namely *sync* and *transistor*, but they were auto-created with the correct type name already. If you select one of the transistors and click the “Parameters” tab, you will see a “Metaparams” field with “str” filled in under “type”.
Now we will create a few metaworkers explicitly. Select all variable_str workers and delete them. For each deleted worker, create a dragonfly.std.variable meta-worker. Under “Parameters=>Metaparams=>type”, fill in “str” and click “Create instance”. Under “value”, fill in the same value it had before, and re-create the connections. Do the same for the display_str worker, replacing it with a dragonfly.io.display meta-worker.
You can find the finished project in the “helloworld-A4” folder.

Chapter 1, section B: Building a hello world hive from Python
Python or visual? Python
Difficulty: Easy

In the “helloworld-B1” folder, you will find a “helloworld.py” that builds and runs a hive directly from Python, without using hivemaps. Versions for Blender (with .blend) and Panda3D are also there. The code consists of the “boilerplate” code, copied from defaultproject.py, and the hive logic, which is closely modelled after the hivemap in section A1 above. The boilerplate is explained extensively in chapter 3 and in the manual, so here we will focus on the logic.
In the first seven lines, the workers that we are going to use are imported. As mentioned earlier, *dragonfly* is a Python module that you can import; its sub-modules correspond to the tree structure that you can see in HiveGUI. In addition, we will import one feature from the bee module, namely bee.connect, which connects two workers. (*For Python programmers: the bee module is **not** a good place to browse the source code; look at dragonfly instead*).
In line 14-21, the actual hive logic is defined. As you can see, there is a very close correspondence between the hivemap in section A1 and the Python code here. In HiveGUI, you would go to workers and select dragonfly=>std=>variable_str, creating a worker named “variable_str_1” (which you can rename later if you like), and set its parameter as “Hello World!”. In Python, you simply define “import dragonfly.std” on top of your script; inside the hive class, you simply define “variable_str_1 = dragonfly.std.variable_str('Hello world!')”. Likewise, if you want to connect variable_str_1 to sync_1, you simply do “from bee import connect” and then “connect(variable_str_1, sync_1)”. As long as there is one valid way to connect two workers, the connection will be made. For more detail, see chapter 3.

In other words, a worker is defined as “workername = workerclass(parameters)”. For meta-workers, you use “()” two times: once to get the worker class, once to get the worker: “workername = metaworkerclass(metaparams)(parameters)”. You can see it for sync_1. Also, in “helloworld-B2”, “dragonfly.std.variable_str” and “dragonfly.io.display_str” have been replaced by the metaworkers “dragonfly.std.variable('str')” and “dragonfly.io.display('str')”. In fact, that is how they were defined in the first place: “variable_str = variable('str')”. You can check it in dragonfly/std/variable.py, line 17. For more details, see chapter 3.

Chapter 1, section C: Creating your own workers using WorkerGUI
Python or visual? Visual
Difficulty: Easy

As you have seen in Sections A and B, a *hivemap* consists of workers and connections visually defined in HiveGUI, whereas a *hive* is the same thing defined in Python code. Likewise, there are visual *workermaps* and Python-code *workers*.
Up to now, we have used the workers of the dragonfly standard library. In this section, we will create our own workers. They will be created visually, in the form of a *workermap*, using the WorkerGUI. The created workers will be embedded inside the hivemap from section A.

Download the default empty hive project at https://launchpad.net/hivesystem/+download and unzip it. This will be used as the starting point for the tutorial.
Start the WorkerGUI. Under Windows, you can find it under "Start => All programs => Hive system => workergui". In Linux, simply type "workergui.py".

A worker is basically a mini-hive, and a workermap is basically a mini-hivemap. But there is an important difference: a hivemap consists (mostly) of worker nodes, and the HiveGUI discovers all workers from which you can choose by scanning the possible folders. On the other hand, a worker consists of a fixed "alphabet" of nodes that are called *segments*. Segments are created and connected into a workermap in the WorkerGUI.

Part C1: The mydisplay worker, using print
Here, we will create a worker that displays a string, replacing the *dragonfly.io.display_str* worker.
First, we will define the interface of the worker with the outside world. This is done using "antenna" and "output" segments. First, start WorkerGUI. Open *segments*, then *antenna*, and then drag *push_antenna_str* onto the canvas (*segments.antenna.push_antenna_str*). Go to Props => General and rename "Segment ID" to "string".
Now, create a *segments.variable.variable_str* and rename its segment ID to "v_string". Connect the antenna to v_string.
Then, create a "modifier" segment (*segments.modifier.modifier*) and rename its ID to "do_display". Go to Props => Parameters => code and type "print(self.v_string)" (no quotes).
Select the v_string segment, go to Props => General => Segment Profile and select "Input". A new output "on_update" will appear; connect it to "do_display". This means that whenever v_string gets updated (by the antenna), the code "print(self.v_string)" gets invoked.

Our workermap is now ready. Go to the defaultproject folder, then to the "workermaps" subfolder, and save it as "mydisplay.workermap".

There is another big difference between hives/hivemaps and workers/workermaps. The hive system can use hives directly, and it can also use hivemaps directly, as long as they are wrapped correctly (inside a so-called *hivemaphive* Python class, this is what defaultproject.py does). However, the hive system can only directly use workers, not workermaps. Direct loading of workermaps is not supported because custom code (such as "print(self.v_inp)") is much user-friendlier to debug inside a Python-code worker than inside a workermap. Therefore, our workermap must be converted to Python code.

Fortunately, the WorkerGUI can generate the Python code automatically. In the menu “Generate” (next to File), click on “Generate code”, or press Ctrl+G. A new window will appear that shows the generated code. Open your favorite Python editor (e.g. IDLE or WordPad), copy-paste the code, and save it in the “workers” folder under “mydisplay.py”.

Now, quit WorkerGUI and start HiveGUI (if it was still open, restart it). Open defaultproject.hivemap. On the left side, under “bees” and “dragonfly”, you will now see a third tree called “workers”, containing “mydisplay”. Create a “mydisplay” worker by dragging it onto the canvas. As you can see, it has an input pin (antenna) called “string”. Finally, create a “variable_str” worker and connect it to mydisplay.string just like in section A1.
When you run the hive, “Hello World!” gets pushed into mydisplay's “string” antenna, which updates the “v_string” variable segment. After receiving the update, the segment sends a signal to the “do_display” segment, which then executes its code. This prints the value of v_string on screen using the built-in Python print() function.
You can find the finished project in the “helloworld-C1” folder.

Part C2: The mydisplay worker, using a display socket
Hives can define their own functions for displaying text. Therefore, it is nicer to use this function instead of (directly) calling the Python print() function. To get access to it, we will define a *socket*. A socket is basically a named request to receive a certain Python object. By convention, the display function can be retrieved with a socket named “display”. For more information on sockets (and *plugins*, which is how you *provide* those objects), see chapter 3.

Open again the mydisplay workermap with WorkerGUI. Then, add a segments.custom_code.custom_class_code. Select it, and in Parameters=>Code, type the following piece of Python code:

```
def set_displayfunc(self, displayfunc):
  self.display = displayfunc
```

Then, add a segments.custom_code.custom_place_code. Select it, and in Parameters=>Code, type the following piece of Python code:

```
s = socket_single_required(self.set_displayfunc)
libcontext.socket("display", s)
```

Finally, change the code of do_display to "self.display(self.v_string)" .

Save the workermap, generate the code and save the code as workers/mydisplay.py.
You won't have to change the hivemap: it uses whatever worker is defined as “mydisplay”, and we haven't changed the name, only the contents.
You can find the finished project in the “helloworld-C2” folder.

Part C3: The mytext worker (variable_str)
Here, we will create a worker that holds a string, replacing the dragonfly.variable.variable_str worker.

Start WorkerGUI to create a new workermap.

The variable segment is exactly as in the mydisplay workermap above: create a *segments.variable.variable_str* and rename its segment ID to “v_string”.
Next, create and output segment *segments.output.output_str*, and rename it to "string". Connect “v_string” to “string”.
Finally, select the “v_string” segment, go to Props=>Parameters, and enable "is_parameter".
Save the workermap as workermaps/mytext.workermap
Press Ctrl+G and save as workers/mytext.py.

Open defaultproject.hivemap with HiveGUI and the new mytext worker should appear (If not, quit HiveGUI and restart it).
Replace “variable_str_1” with a new mytext worker: delete the old worker, create the new mytext worker, rebuild the connections, and in Props=>Parameters, set v_string to "Hello world!"
Finally, save the hivemap and run the hive.
You can find the finished project in the “helloworld-C3” folder.

Chapter 1, section D: Creating your own meta-workers
Python or visual? Python
Difficulty: Intermediate

Meta-workers are workers that can take meta-parameters that determine how the worker is built. Usually these meta-parameters describe the type(s) of the data that the worker accepts. We have used meta-workers already in section A4. Here we will build our own. Meta-workers, unlike normal workers, cannot be built in WorkerGUI, but only in Python.
The starting point is the project of section C3, which you can find in the “helloworld-C3” folder. Copy the project to a new place. Open defaultproject.hivemap with HiveGUI, remove all workers, and save it. You can also remove the workermap files in the “workermaps/” folder. Instead of the workermaps, we will take the generated Python code in the “workers/” directory as a starting point.

A metaworker must be a Python object with the following two attributes:
- A method __new__ that takes a number of arguments, and that returns a bee.worker class
- An attribute "metaguiparams" that must be a dictionary. The keys of the dictionary are the names of the parameters to __new__. Its values are the data types of these arguments, which can be any allowed hive system data type (such as "str", "float", "Coordinate", …) but most commonly it is "type", since their values will describe the names of the data types for which the worker will be built.

Open “workers/mydisplay.py” in a text editor. Go to the line "class mydisplay(bee.workers)", select it and all code below it, and indent it two levels (I use two spaces per level).

Then, add the following code:

```
class mydisplay(object):
  metaguiparams = {"vartype":"type"}
  def __new__(cls, vartype):
```

```
    class mydisplay(bee.worker):
      ... (here comes the existing code) ...
    return mydisplay
```

Finally, search for 'str' (with quotes!) and replace it with "vartype" (no quotes!):

```
  ...
  string = antenna('push', vartype)
  ...
  v_string = variable(vartype)
```

Save the Python file. Then, do exactly the same for mytext:

```
class mytext(object):
  metaguiparams = {"vartype":"type"}
  def __new__(cls, vartype):
    class mytext(bee.worker):
      ... (here comes the existing code) ...
    return mytext
```

and also replace 'str' with vartype.

Start HiveGUI and open the (empty) defaultproject.hivemap. Both the metaworkers should now appear under “workers”.
Create a mydisplay worker and a mytext worker, and for both, enter "str" under
Parameters=>Metaparams=>vartype
in mytext_1=>Props=>Parameters, set v_string to "Hello world!"
Connect the workers, save the hive and run it.
You can find the finished project in the “helloworld-D” folder.

Chapter 2: Drawing on a canvas

The hive system comes with a 2D canvas on which you can draw shapes (images, text, etc.). The canvas works under the BGE (blenderhive) and Panda3D (pandahive). In this chapter, as an example shape, the implementation and usage of a colored text box is explained.

The projects for this chapter are in the "canvas" folder. In "canvas0", you will see an empty canvas project. There is a "canvas-blender" hivemap, containing a single drone, the *canvas drone* (dragonfly.blenderhive.blendercanvas), that adds canvas functionality to the BGE. "canvas-blender.py" loads this hivemap under a blenderhive and runs it. Of course, this will only work if "canvas-blender.py" is run inside the BGE. "canvas.blend" is a .blend file where "canvas-blender.py" is defined as the main BGE script: starting the BGE after opening "canvas.blend" will start up the canvas-blender hive.
Likewise, there is a "canvas-panda" hivemap, loaded by "canvas-panda.py" using the Panda3D engine. It contains a canvas drone for Panda3D (dragonfly.pandahive.pandacanvas).

There are different ways to implement and use canvas shapes. "start1", "start2", "start3" and "start4" contain four different implementations of a colored text box. In part 1, the usage of each implementation inside HiveGUI is explained. In part 2, it is shown how each of the implementations works from a developer's perspective, showing how you can extend the canvas to draw your own custom shape classes.

Part 1: Drawing on a canvas, from a GUI perspective
Python or visual? Visual
Difficulty: Intermediate

Part 1, Section 1:

This section describes six usage examples for the first implementation of colored text boxes. The implementation is in the "start1/" project folder, which provides the starting point for the usage examples.

In this implementation, the functionality for drawing colored text boxes is provided by a separate drone (workers.coloredtextbox_blender or workers.coloredtextbox_panda) that is added to the hivemap, in addition to the canvas drone. Individual boxes are created using Spyder data models. The Spyder model contains not only the properties (color, text) and also the bounding box (size) of what we want to displayed. Therefore, it is displayed using the canvas "show" system, rather than the canvas "draw" system. More details on "show" vs. "draw" are in section 2 below.

The "canvas1/" folder contains the first usage example of this section: the static definition of a colored text box. With SpyderGUI, you can open the file "spydermaps/myspydermap.spydermap". A *spydermap* is a collection of structured data (Spyder objects) that are interpreted by the hive system as *configuration*. A spydermap needs to be interpreted by special hive called a *spyderhive*, which knows about colored text boxes. For each spydermap, you must select a spyderhive that will interpret it. As you can see, the interpreting spyderhive has been defined as

“spyderhives.myspyderframe.myspyderframe”. The spydermap contains a single ColoredTextBox Spyder object called “helloworld” (the “helloworld” node was created as spyderbees => attribute, with metaparameter “spydertype” set to “ColoredTextBox”). The ColoredTextBox object itself is defined as helloworld's “val” attribute: the box is red, 600x200 pixels, and contains the text “Hello World!”.
A copy of the spydermap has been added to the hivemap under the name “myspydermap”. Running “canvas1/canvas.blend” under the BGE (or canvas1/canvas-panda.py under Panda3D) will display the box. For more information on spydermaps, see chapter 3.

The “canvas1a/” folder shows he second usage example of this section: the dynamic definition of a colored text box. This is done not in a spydermap, but in the hivemap itself. The “coloredtextbox” variable (dragonfly.std.variable, “type” metaparameter = ”ColoredTextBox”) contains the textbox itself. It is connected to a transistor (dragonfly.std.transistor, “type” metaparameter = ”ColoredTextBox”) . Upon hive startup, the value of the variable is pushed by the transistor into a show1 worker (dragonfly.canvas.show1, “type” metaparameter = ”ColoredTextBox”), which shows the textbox.

To this, the “canvas1b/” folder adds a way to dynamically remove the textbox. The show1 worker generates and holds an identifier to the last object that was shown. When you press the SPACE key, this identifier gets pushed into the remove2 worker which removes it from the canvas.

The “canvas1c/” hivemap follows a different approach. You may have to zoom out to show all nodes in the hivemap. The hivemap displays two text boxes, and their identifiers are not generated, but predefined. The “show3” worker always displays a text box under the identifier “textbox1”, and the “remove3” worker always removes the “textbox1” object. Likewise, the identifier “textbox2” is used by “show3a” and “remove3a”.
Upon start, and upon pressing the SPACE key, the “toggle” worker toggles its state (which is initially False). If the new state is True (on), the “coloredtextbox” variable is shown under “textbox1”. If the new state is False (off), it gets removed again. Likewise, the RETURN key manages the visibility of the other textbox variable under the “textbox2” identifier.

The “canvas1d/” hivemap is a bit more complex usage case. Initially, it shows a textbox in the same way as “canvas1a/”. However, pressing the SPACE key not only removes the last textbox, it also shows a new one at a random position. This is achieved by modifying the Spyder ColoredTextBox object at runtime, using nodal logic.
The value of the ColoredTextBox object is not stored in a variable, as previously, but instead in a “coloredtextbox” *block worker* (dragonfly.blocks.block, “spydertype” metaparameter = ”ColoredTextBox”). This block worker is linked to a *setter* (dragonfly.blocks.setter, “spydertype” metaparameter = ”ColoredTextBox”) that allows the modification of ColoredTextBox elements. In the setter's Block section (on the right side, Props=>Block tab), two entries have been defined: “box.x” and “box.y”. This leads to the creation of two push antennas on the setter, both of them taking integers.

When the SPACE key is pressed, first it removes the “textbox1” textbox, using the remove3 worker. Then, it pushes a value into the box.x antenna of the setter. The value is obtained from randomx, a dragonfly.random.randint worker. When randomx is asked for a value, it pulls in a (min, max) range tuple and returns a random integer from that range. The range tuple is built by a *weaver* (weaver_1;

dragonfly.std.weaver, “type” metaparameter = "('int', 'int')") from the “minx” and “maxx” variables, as (50, 500). After box.x is updated, the same is done for box.y. Finally, the show3 worker shows the updated ColoredTextBox as a new textbox under “textbox1”.

Finally, “canvas1e/” is an advanced usage case. When you look at the hivemap, you will see a worker called “create”. This is a simple *connector* (dragonfly.std.pushconnector, “type” metaparameter = “trigger”) : whenever it receives a trigger on “inp”, it sends a trigger on “outp”. The “create” connector is triggered by either the start event or the press of a SPACE key, and does three things:

1. It generates a random value for box.x and box.y in the same way as the previous section
2. It shows the randomized textbox on screen
3. It triggers transistor_3

Transistor_3 takes the identifier of the new textbox. It first pushes it into a *selector*, a worker that holds identifier and keeps one of them selected. Then it displays (prints) the identifier, adding “Created:” in front.
The delete key triggers a *filter* (dragonfly.logic.filter, “type” metaparameter = “trigger”) called “is_empty”. The filter checks if the selector is empty: if it is, it does nothing. If the selector is not empty, it does two things:

1. It triggers transistor_2. Transistor_2 takes the currently selected textbox identifier, removes it from the canvas, and displays it, adding “Deleted:” in front.
2. It triggers the “unregister” antenna of the selector, which removes the currently selected identifier from the selector, and selects the previous identifier (if any).

Repeated presses of SPACE will create more and more text boxes. Pressing DEL will remove them again, starting with the last one that was created. This order is called LIFO (last-in,first-out). To use FIFO (first-in, first-out) instead, connect transistor_3 to “register” instead of “register_and_select”.

Part 1, Section 2:

This section describes the second implementation. The starting point for this implementation is the “start2/” project folder.
The second implementation is very similar to the first. The only difference is that in the first implementation, there was a canvas drone, plus an additional drone (coloredtextbox_blender, coloredtextbox_panda) to draw ColoredTextBoxes. The second implementation uses a single canvas drone (mycanvas) that can *also* draw the ColoredTextBoxes. Therefore, the hivemap contains only one drone instead of two.
For the rest, the implementations are exactly the same. The “canvas2/” folder is the same as “canvas1/”, and “canvas2a/” is the same as “canvas1a/”. The other usage cases of section 1 can be ported in the same way.

Part 1, Section 3:

This section describes three usage examples for the third implementation. The starting point for this implementation is the “start3/” project folder.

The difference with the first implementation is that *at runtime*, text boxes are not represented as a

Spyder model (ColoredTextBox) but as a Python class (designated in the HiveGUI as type ("object", "coloredtextbox")). For *configuration*, the implementation still contains a ColoredTextBox Spyder model that is internally converted to Python.
This means that *static configuration* is still possible in a visual way. This is show in the "canvas3/" folder, which contains the exact same hivemap and spydermap as "canvas1/" . However, the *dynamic* use of text boxes cannot be done visually, and the other usage cases of section 1 cannot be ported straightaway. "canvas3a/" is a port of "canvas1a/", showing a colored text box dynamically upon startup. The hivemap is explained below.
The value of the text box is contained in the v_coloredtextbox variable. This variable contains the actual Python object: the type of the variable is set to ("object", "coloredtextbox"). However, the HiveGUI has no mechanism to visually edit an arbitrary Python object, nor can the hivemap format store an arbitrary Python object. Therefore, the object must be constructed from Python code at hive startup, and then inserted into the variable. The construction is done by a *pyattribute* (Workers => bees => pyattribute) and the insertion is done by a *wasp* (Workers => bees => wasp). Both of them are executed at runtime when the hive is built.
The pyattribute is called "coloredtextbox". In the "Parameters" tab, you can see two fields. The "code" field contains the Python code, importing the coloredtextbox Python class and constructing a Python object as "v = coloredtextbox(...)". When the hive is built, this code is executed. The "code_variable" field defines "v" as the variable in the code that holds the Python object. The wasp then inserts this Python object into the "value" parameter of the "v_coloredtextbox" variable. The rest of the hivemap works just as "canvas1a/", except that the type of all meta-workers is now ("object","coloredtextbox") (the Python class) instead of ColoredTextBox (the Spyder model).

Note for Python programmers: the pyattribute defines an actual "coloredtextbox" attribute of the hivemap. If you edit canvas-blender.py or canvas-panda.py and go to the semifinal line (just above "main.run()"), you can retrieve a bee.attribute object by "attr=main.hivemaphive.hivemap.coloredtextbox". To get its actual value, you can call the bee.attribute. Therefore, "print(attr())" will yield "<coloredtextbox.coloredtextbox object at 0x.....>", and "print(attr().textcolor)" will yield "(1.0, 0.0, 0.0, 1.0)"

Part 1, Section 4:

This section describes three usage examples for the fourth implementation. The starting point for this implementation is the "start4/" project folder.

The difference with the first implementation is that *at runtime*, text boxes are represented as a different Spyder model: ColoredText instead of ColoredTextBox. A ColoredText does not contain the bounding box (position and size), and therefore it is used with the canvas "draw" system instead of the "show" system. However, like section 3, a ColoredTextBox Spyder model is added on top, so *static configuration* with spydermaps works the same (the "canvas4/" project is the same as "canvas1/"). The different usage of ColoredText and the "draw" system is shown in "canvas4a/" and "canvas4b/".
In general, because of the decoupling of drawn objects and their bounding boxes, the "draw" system is more powerful and flexible, but also more difficult to use.

In "canvas4a/", the ColoredText is drawn dynamically, but its bounding box is configured statically, in

a spydermap. In “myspydermap.spydermap”, opened with SpyderGUI, you can see two nodes: “canvas”, which is a *parameter* (spyderbees.parameter), and “canvasslot_helloworld”, which contains a CanvasSlot Spyder object. The “canvas” parameter means that the canvas drone must be passed as a parameter to the spydermap. The canvas slot contains the name of the canvas (“canvas”, the same name as the parameter), the name of the bounding box (“slotname”; set to “id_helloworld”), the type of the object that should be in the bounding box (“slottype”; set to “ColoredText”) and the bounding box parameters.
In the hivemap (canvas-blender.hivemap and canvas-panda.hivemap), the spydermap is embedded, and the canvas drone is indeed passed as a parameter to it (using a wasp). The spydermap configures the canvas so that the bounding box is added and provided under the name “id_helloworld”.
The ColoredText is defined inside the “coloredtext” variable, and the drawing is performed by the draw3 worker. It draws the colored text inside the bounding box defined as “id_helloworld”, which is provided by the canvas.

There is one twist: in the hive configuration phase, the spydermap configures the canvas with new bounding boxes, and *then* the canvas declares these bounding boxes as plugins. This is one of the rare cases where the configuration order matters. In the hive system, the configuration order is determined by the *names of the bees*, in alphabetic order. Therefore, the spydermap is named “aaa_myspydermap”, so that the spydermap is configured *before* the canvas drone is. Don't worry, if you defined the wrong name order so that new bounding boxes are added too late, an error message will be printed.

In “canvas4b/”, static configuration by spydermaps is not used, and the bounding box is defined dynamically. This is achieved using the “reserve” worker (dragonfly.canvas.reserve). The “reserve” worker takes a tuple of two arguments, an identifier and a bounding box object. The bounding box is a Python object (“box2d” Python class, the hive system type name is (“object”, “box2d”)). We could define the bounding box object as Python code, with a pyattribute, like in section 3. However, it is possible to edit the bounding box visually, using an *attribute (*bee.attribute) . An attribute works the same as in the SpyderGUI, namely by selecting a Spyder type and editing it. In our case, the attribute node is named “box1” and its Spyder type is “Box2D”. The Box2D Spyder model is special because it contains a Python method “sting()”, which returns a box2d Python object. The wasp “wasp_1” inserts box1 into the “v_box1” variable, but the wasp has the parameter “sting” enabled. This means that before inserting, the sting() method is called on the Spyder object, and the result (a Python box2d object) is inserted into “v_box1”.
Then, the weaver packs the identifier “id_helloworld” and the box2d into a tuple. Upon startup, this tuple is pushed into the “reserve” worker. This makes the “id_helloworld” bounding box available for use by the “draw3” worker to draw the ColoredText. To enable the dynamic definition of the bounding box, the “static” parameter of the draw3 worker is disabled.

The “canvas4b/” hivemap shows an alternative way to draw a ColoredText using the draw1 worker. While draw3 uses a ColoredText with a fixed bounding box identifier, draw1 takes a tuple of a ColoredText and a bounding box object. The bounding box is defined in the same way as for draw3: visually (Box2D) in an attribute (box2), and inserted by a wasp (wasp_2) into a variable (v_box2) using the sting option. A weaver then weaves the ColoredText and the bounding box into a tuple that is pushed into the draw1 worker.
Running the hive will display the ColoredText in both bounding boxes at startup.

Part 2: Drawing on a canvas, from a developer's perspective

Python or visual? Python
Difficulty: Intermediate

Here, canvas drawing is explained from a developer's perspective. In short, it is shown how you proceed from a default starting point (“canvas0/”) to four different implementations (“start1/”, “start2/”, “start3/”, “start4/”) of a colored text box. First, it is explained what different design choices were made in these implementations, so that you can make the correct choices for drawing your own shapes.

Choice 1. Will shapes be drawn statically or dynamically? Will they be created once and never updated? Or will they be dynamically created, updated and/or removed?
Choice 2. At runtime, what should be the primary representation of my shape: a Spyder model or a Python class?

The answers to these choices are related. When shape are created only statically, they will all be defined in spydermaps. In that case, you can easily define your shape as a Python class, and use this representation internally. For the sole purpose of visual editing, you define then a Spyder model with a method that converts it to a Python object.
However, if your shapes are created dynamically, you should try to define your shape in Spyder, defining the drawing functions for the Spyder model instead of for a Python class. This is much friendlier for a user who uses HiveGUI: because of their rigid structure, Spyder models are inherently visual and editable in HiveGUI. Also, block workers (dragonfly.blocks) can be used in hives for arbitrary access and manipulation of Spyder objects, whereas a custom-written worker is required in the case of a Python class (an example of such a custom worker can be seen in dragonfly.grid.bgridcontrol).
However, if you are using some library to manage your shape objects, you may be forced to use a Python class as the primary representation (or to use a Python library class directly). Also, if your users prefer to define their hives in Python code instead of in HiveGUI, a Python representation may be easier for them. However, to make it easier for HiveGUI users, you should in any case define a Spyder model with a sting() method that converts into the Python class. This allows those users to define variable values visually, and not having to use pyattributes (they are evil!).
Of the implementations, only implementation 3 uses internally a Python representation of the colored text box shape. All of the other implementations use a Spyder model (ColoredTextBox or ColoredText) as the primary representation.

Choice 3: Should my shape contain bounding box information?
For a shape that appears in a fixed area, such as a skill bar, it may be best to include the position and size in the data model. On the other hand, if a shape wanders around a lot, or if the same area is holding different shapes in succession, it may be better to decouple the bounding box from the shape.
The hive system canvas has two different systems for managing shapes: the “show” system for shapes with bounding boxes, and the “draw” system for shapes without. In general, the “show” system is easier to use. Of the implementations, only implementation 4 uses the “draw” system, the other use the “show” system.

Choice 4: Should I define a separate drone for my shape?
In general, to enable the canvas, users must add a canvas drone to their main hive or hivemap. The hive system gives you the choice. You can either subclass the standard canvas drone, and tell your users to include that one instead of the normal canvas drone. Or, you can define a new canvas drone, that your users must add in addition to the standard canvas drone, if they want to use your shape.
Only implementation 2 subclasses the standard canvas drone. The other implementations use a separate canvas drone to implement colored text boxes.

Part 2, Section 1:
This implementation uses the "show" system, has the ColoredTextBox Spyder model as the primary representation, and defines a separate additional canvas drone to draw colored text boxes. The implementation can be found in "start1/", and proceeds in three steps:

step 1. Defining a ColoredTextBox Spyder model
The ColoredTextBox Spyder model is defined in spydermodels/ColoredTextBox.spy. As you can see, it is just a description of the data layout of ColoredTextBox. More information about Spyder can be found in chapter 3 and at http://spyder.science.uu.nl. To make the data model automatically imported by the GUIs and the main scripts, the line "from .ColoredTextBox import ColoredTextBox" must be added to spydermodels/__init__.py.

step 2. Specifying a drone that can render a ColoredTextBox
The ColoredTextBox canvas drone is defined in workers/coloredtextbox_blender.py for the BGE, and in workers/coloredtextbox_panda.py for Panda3D.
The hive system provides a helper function "build_canvasdrone" to define your own canvas drones for shape rendering. You can import it from your top level hive (blenderhive or pandahive), and you invoke it as:

```
X = build_canvasdrone (
  wrappedclass = … ,
  classname = "X",
  drawshow = "show" / “draw”,
  drawshowtype = "ColoredTextBox",
  baseclass = dragonfly.canvas.canvasdrone
)
```

In this case, X is replaced with "coloredtextbox_blender" / "coloredtextbox_panda", and drawshow = "show".
Note that the generated canvasdrone manages the rendering of *all* ColoredTextBox objects. In contrast, instances of the supplied "wrappedclass" will each manage the rendering of a *single* ColoredTextBox object. The requirements of this wrappedclass depend on the system ("draw" or "show") and the rendering backend (BGE or Panda3D). For the "show" system, they are described below:

Blender
When running under the BGE, the hive system canvas describes its scene hierarchy using instances of *bglnode* (dragonfly.blenderhive.bglnode). Each bglnode has a draw() function that does a OpenGL matrix transform and then calls the draw() function of its children. Therefore, if an object wants to be

drawn, it must have a draw() method and must possess a handle to a bglnode that is already in the scene, and then add itself to that bglnode's children.
The wrappedclass's __init__ method must accept four arguments (in addition to *self* , of course). The first argument will be the instance of the canvasdrone generated by build_canvasdrone. The second argument is the ColoredTextBox object that is to be rendered. The third argument is the identifier for the object (auto-generated in the case of the show1 worker, or user-defined in the case of show2 or show3). Finally, the fourth argument “parameters” may contain additional parameters for the rendering of an object. They are not used in this chapter.
The first argument, the canvasdrone, can be accessed to link the drawing code to the scene. To get an overview of its methods, look in the dragonfly.blenderhive.build_canvasdrone to see the class “buildclass”. Most importantly, the method *_get_parent_bglnode(self, identifier, boundingbox)* generates and returns a new bglnode that is already linked to the scene. The wrappedclass instance must add itself to this bglnode's children, in order to have its draw() method invoked.
The wrappedclass's draw() method (no arguments) should contain the OpenGL (bgl) commands to render the ColoredTextBox.
The wrappedclass's update() method must accept three arguments: a ColoredTextBox, an identifier, and a parameters object. They correspond to argument 2-4 of the __init__ function, and they may indeed be identical to it.
Finally, the wrappedclass must have a remove() method that removes the shape from the scene. This is done simply by invoking “removeNode()” on the bglnode.
A demonstration of these required “wrappedclass” methods is provided not only by coloredtextbox_blender.py but also by several classes in /dragonfly/blenderhive/.

Panda3D
When running under Panda3D, the hive system canvas describes its scene hierarchy using instances of Panda3D NodePaths. As usual in Panda3D, to render an object, it must reparent to an existing NodePath that is already in the scene.
The wrappedclass's __init__ method must accept four arguments (in addition to *self* of course). These are the same arguments as for the BGE. The first argument will be the instance of the canvasdrone generated by build_canvasdrone. The second argument is the ColoredTextBox object that is to be rendered. The third argument is the identifier for the object (auto-generated in the case of show1, or user-defined in the case of show3). Finally, the fourth argument “parameters” may contain additional parameters for the rendering of an object. They are not used in this chapter.
The first argument, the canvasdrone, can be accessed to link the drawing code to the scene. To get an overview of its methods, look in the dragonfly.pandahive.build_canvasdrone to see the class “buildclass”. Most importantly, the method *_get_parent_nodepath(identifier, boundingbox)* generates and returns a new NodePath that is already linked to the scene. It is up to the wrappedclass to parent new Panda3D objects to this NodePath.
The wrappedclass's update() method must accept three arguments: a ColoredTextBox, an identifier, and a parameters object. They correspond to argument 2-4 of the __init__ function, and they may indeed be identical to it.
Finally, the wrappedclass must have a remove() method that removes the shape from the scene. As usual in Panda3D, this is done simply by invoking “removeNode()” on the NodePath.
A demonstration of these required “wrappedclass” methods is provided not only by coloredtextbox_panda.py but also by several classes in /dragonfly/pandahive/.

Note that for both Blender and Panda3D, the coordinate system is defined so that (0,0) is the bottom left of the designated bounding box, and (1,-1) is the top right.

step 3. Adding support for static configuration
Together, the ColoredTextBox Spyder model and the coloredtextbox_blender canvasdrone are sufficient for the *dynamic* rendering of colored text boxes in HiveGUI, using the dragonfly.canvas workers, for example in the "canvas1a/" folder. The *static* configuration of ColoredTextBoxes, with SpyderGUI, requires one additional layer: a spyderhive. The file "spyderhives/myspyderframe.py" contains the definition of this spyderhive.

The purpose of the spyderhive is to add a make_bee() method to ColorTextBox, leading to the rendering of the ColoredTextBox at hive init time. To do so, the make_bee() method must return an instance of a drone that contains a special plugin (libcontext.plugin). The *name* of the plugin must be ("canvas", "show", "init", "ColoredTextBox"). The *content* of the plugin must be a *canvasargs* instance (dragonfly.canvas.canvasargs). A canvasargs contains any combination of: shape object (in our case: ColoredTextBox), identifier, and parameters (i.e. arguments 2-4 of the wrappedclass __init__, or all arguments of wrappedclass update()). A canvasargs can be constructed for example as "canvasargs(obj = …, identifier = …)" , or simply as "canvasargs(obj)". Finally, a dummydrone (bee.drone.dummydrone) can be constructed to provide a wrapper around the plugin. An instance of this dummydrone is what the make_bee() method returns.
More information on plugins is in chapter 3.

Part 2, Section 2:
This implementation uses the "show" system, has the ColoredTextBox Spyder model as the primary representation, and subclasses the standard canvas drone to draw colored text boxes. The implementation can be found in "start2/".

The difference between this implementation and the first one is that there is only one canvasdrone. Instead of a standard canvasdrone "blenderhive.blendercanvas" plus a specialized (ColoredTextBox-specific) canvasdrone "coloredtextbox_blender", there is now only the canvasdrone "mycanvas_blender". As you can see, the implementations of the drones (start1/workers/coloredtextbox_blender.py versus start2/workers/mycanvas_blender.py) are nearly the same: the only difference is build_canvasdrone takes "object" instead of "canvasdrone" as baseclass, and the result is now *embedded* in a class derived from blenderhive.blendercanvas. For the rest, the implementation is identical to section 1.
The same is true for the Panda3D version (start2/workers/mycanvas_panda.py)

Part 2, Section 3:
This implementation uses the "show" system, has the coloredtextbox Python class as the primary representation, and defines a separate additional canvas drone to draw colored text boxes. The implementation can be found in "start3/", and can be divided into three steps:

step 1. Defining a coloredtextbox Python class
The Python class is defined in "start3/coloredtextbox.py". As you can see, it just holds the data to

describe a colored text box, very similar to the Spyder model that the other implementations use.

step 2. Specifying a drone that can render a coloredtextbox
This is very similar to the the first implementation. In both cases, the implementation is in workers/coloredtextbox_blender.py. The most obvious difference is the *name* of the class for which we are defining the renderer (the “drawshowtype” parameter). If it had been a Spyder class, the name would have been equal to the class name: “ColoredTextBox”; the HiveGUI recognizes the Spyder capitalization pattern (“camelcase”), and everything goes fine. But for a Python class, you must pick a name of the form (“object”, <something>), and <something> is up to you. Here, (“object”, “coloredtextbox”) is chosen. This means that (“object”, “coloredtextbox”) must be the type meta-parameter of canvas workers that show/remove coloredtextbox objects, as well as for other workers (variables, transistors) that hold and manipulate them.

step 3. Adding support for static configuration
The steps above are sufficient for the *dynamic* rendering of colored text boxes in HiveGUI, for example in the “canvas3a/” folder. However, to allow *static* configuration in the SpyderGUI, a ColoredTextBox Spyder model must be defined on top of the “coloredtextbox” Python class, with a converter to generate a “coloredtextbox” out of it. The Spyder model is in “spydermodels/coltextbox/ColoredTextBox.spy”, and the converter in “spydermodels/coltextbox/to_coloredtextbox.py”. A spyderhive very similar to “start1/” is defined in “start3/spyderhives/myspyderframe.py”.
The converter has also been specified as the ColoredTextBox string() method, allowing visual ColoredTextBox definitions in HiveGUI that are inserted into (“object”, “coloredtextbox”) variables by wasps that have their “sting” option enabled.

Part 2, Section 4:
This implementation uses the “draw” system, has the ColoredText Spyder model as the primary representation, and defines a separate additional canvas drone to draw colored texts inside a bounding box. The implementation can be found in “start4/”.
The main difference with “start1/” stems from the different requirements for the build_canvasdrone “wrappedclass” when using the draw system. For the BGE, the wrappedclass is no longer responsible for adding itself to any kind of bglnode. It must still implement the exact same methods receiving the exact same arguments, but for ColoredText, the canvasdrone argument is simply ignored, and the “remove” method does nothing. For Panda3D, those methods now all receive an extra argument “parentnode” (after “identifier”), a NodePath to which they should reparent. The canvasdrone takes care of creating and removing this “parentnode”: as for the BGE case, the wrappedclass's “remove” method does nothing.

For the sake of static configuration in the spydermap, a ColoredTextBox Spyder model has been defined also. The layout is a bit different than in the “start1/” implementation: it consists of a ColoredText plus a bounding box (Box2D). When a ColoredTextBox is being processed, the ColoredText is drawn using the bounding box. This can be seen in the spyderhive (spyderhives/myspyderframe.py) to the “start1/” implementation.

Chapter 3: The hive system layers
Python or visual? Both
Difficulty: Advanced

Introduction

The hive system is a configuration system. It allows one person to design and set up a system in the form of a node library and data structures. This system can be configured by a second person into an actual application: creating instances of nodes, connecting nodes, and setting up parameters, using the Hive GUI and/or Python. A well-designed (sub-)system can be connected with and embedded inside any other system (or Python script) in any way the second person chooses.
I will call the first person “the programmer”, because design such a system requires strong skills in logic and software design. Even so, the required amount of Python syntax is usually very limited, so “visual programmer” may be a more accurate term. I will call the second person “the editor”, because configuring a program using the hive system requires no programming skills, although it does require technical skills and good understanding what the nodes do. The hive system is very flexible and gives fine control over the individual components of a program. The ultimate goal is to give this fine control to both the programmer and the editor, keeping a visual form where possible, while reusing existing components and minimizing the amount of Python syntax that is needed.

This chapter is aimed at both the programmer and the (expert) editor. For the editor, it will bring the necessary insight into the hive system so that they can use existing system setups effectively. For the programmer, it will show the necessary steps to set up a system, and how different design choices can be made and implemented. Note that the hive system should be able fulfill the needs of the visual programmer, but node libraries are still in development: for most tasks, there are not nearly enough nodes finished to bring total control to the editor. Therefore, I hope that this chapter will make it easier for programmers to design new nodes and data structures to remedy this, allowing editors to perform a large variety of tasks.

In the hive system, high-level interfaces are automatically generated from lower-level definitions. For example, there is no need to develop your own GUI for data editing or file formats for resource definition: these things will be auto-generated as a visual layer inside the HiveGUI. Therefore, the main focus of this chapter is about the many different representation layers (both visual and textual) in the hive system: how they work, and how the high level layers translate into lower-level text representations, and finally into pure Python code.

As an example, a third-person application is chosen, where the user controls a character. When the user performs some action (e.g. a key press), an animation for the character will be shown and/or a sound will be played.
When discussing this example, the following assumptions are made:

- The goal is to give full control of the program to an editor who is using the HiveGUI. While it is perfectly possible to control the hive system through text files (both Python source files and Spyder data), a GUI is preferable. In the GUI, the editor must be able to define a dictionary of actions, each with an associated name, animation and sound. The editor must also have control

over what triggers the actions (change the key press, or change it into something else entirely). Finally, another programmer (or editor with sufficient skill) must be able to easily change the nature of these actions, by breaking the coupling between sound and animation, or by adding additional features to the action (e.g. changes in the camera, the environment, etc.).

- What that the program exactly does (showing animations, playing sounds) is irrelevant here. There are already hive system examples that show these things. For our purpose, we simply assume that there are somewhere some functions that can do these things; the tutorial chapter just shows simple text messages such as “Playing sound: x.wav”.
- Some people prefer to learn things top-down, seeing a GUI example where the layers are step-by-step explained and broken down to their underlying Python instructions, while others prefer a bottom-up approach, starting with some Python code that is gradually wrapped in nodes and data structures. This tutorial chapter will do both: one section that explains the system top-down, then another one explaining it bottom-up.

This tutorial chapter is not gentle. If you just want to *use* the hive system and its existing nodes, and don't care to *understand* it, you may be better off with the Hello World chapter or the Tetris demonstration. Also, especially if you are an editor without programming skills, you may want to skip part of this tutorial chapter: it is organized in 20 layers, and non-trivial Python syntax is used heavily beyond layer 12 or so. It is all up to you.

Now let's get started.

Designing the system

When you are doing multiple things (playing sounds **and** animations), and these things are somehow coupled, there are three sane ways for a programmer to design the system, to be configured by an editor person. Each approach has advantages and disadvantages:

- You can create “play” components (nodes) that process the actual data (names of sound files, direct handles to animation objects), and store the sound data and animation data in two different dictionaries. Each action has a unique identifier key, which is used to retrieve the sound data from the sound dictionary, and then, with the same identifier key, the animation data from the animation dictionary, feeding each piece of data to its component.
  The advantages of this approach are that you can easily decouple sound and animation, and you can modify both dictionaries at runtime. The disadvantage is that there is nothing representing an action at runtime: just the shared identifier indicates that a sound and animation belong together. The editor can still define actions as action objects at the configuration stage, as long as they are first decomposed into sounds and animations when the hive starts.
- You can create the components as above, but manage the data in a single dictionary, storing action objects that contain both sound and animation data.
  The advantages are that there is a single representation, the action object, at all stages of the program, and that action objects can still easily be modified in the dictionary. The disadvantage is that is now much harder to decouple the sound and animation. In addition, the editor will also need to extract the sound and animation data portions at runtime before feeding them to the play

nodes, which adds some overhead in the form of additional nodes.

- You can avoid the explicit managing of dictionaries by the editor; the “play” components maintain their own internal dictionaries, and the editor simply configures them. At runtime, the components are accessed simply with identifiers that causes the associated animation/sound to be played.
  The advantage is that this makes the system rather easy to use. Also, because of the way the hive system works, it is now possible to separate the place where the actions are configured from the place where they are played. The disadvantage is that the dictionaries are now more or less hidden from the editor, reducing the editor's control over the system: it is now much harder to modify a dictionary at runtime, or to maintain alternative dictionaries for different sets of actions, unless additional nodes are written to accomplish these tasks.

The first approach is known as an entity system. The second and third approach are object-oriented approaches, with the second approach being data-centric (grouping all data of an action into a single object), and the third approach being more code/component-oriented (grouping the sound code together with its dictionary into a single “jukebox” component, and the same for animation).

I will not argue that one approach is best in the general case. In my opinion, it depends on the situation, on the style of the programmer and of the editor. The hive system has the philosophy of giving you the choice. This tutorial chapter contains implementations of all three approaches (systems) and they will be simultaneously broken down, layer by layer.

The directory “layer1” contains a top-level, visual implementation using the HiveGUI.
Likewise, the “layer20” folder contains a pure-Python implementation that does not use the hive system at all. The next section will describe the progress from top to bottom, from layer 1 to layer 20; and the final section will do the reverse, traveling from bottom to top, from layer 20 to layer 1.

Top-down

In the hive system, all components are called “bees”, which come in many flavors. For example, workers are bees that can be connected (nodes); drones are bees that are not connected explicitly, they communicate implicitly through sockets and plugins via Python. Hives are container classes for bees: all bees are embedded inside hives. Hives are hierarchical: hives may themselves also be embedded inside other hives. Hives can be defined directly in Python, or in visual form (called a hivemap) that is edited in the HiveGUI. The main hive is invoked from a start-up script, embedding it inside a platform-specific top-level hive (Blender Game Engine, Panda3D or the command line terminal). There are other visual representations besides hivemaps: workermaps define a custom worker (node) that can be embedded and connected within a hive, and spydermaps contain Spyder data, parametric data objects that are used to configure and define resources for the various components.

Layer 1-6 are based on the very-high-level (visual) layers of the hive system

Layer 1
In the directory “layer1”, you can see a top-level, visual implementation using the HiveGUI. It is based on the defaultproject folder. The stubs for playing sound and animation (they simply print text

messages) are added as a new Python file called “somelibrary.py”. The main hivemap has been renamed to “layers.hivemap”. The chosen top-level hive is the consolehive, which can read key presses from the terminal. Therefore, the main script “layers.py” is based on “defaultproject.py”; the default scripts for the other top-level hives (“defaultproject-blender.py” and “defaultproject-panda.py”) have been deleted. As you can see, the changes in “layers.py” from the default project are absolutely minimal: importing “somelibrary.py” and changing the name of the hivemap, that is all.

The three design approaches described in the previous section are implemented as three systems called action1, action2 and action3. They are implemented using three hivemaps in the “hivemaps” sub-folder. The three hivemaps use custom workers to play the sound. These custom workers have also been created visually, using the WorkerGUI: their visual forms, workermaps, are stored in the “workermaps” sub-folder: play_X for action1 and action2, and action3_play_X for action3.
The configuration of the action dictionaries is done using custom data models to describe an action. The hive system's data modelling framework is called Spyder, and you can find the models in the “spydermodels” sub-folder.
The actual configuration data is defined as spydermaps (in the “spydermaps” sub-folder), visual description of Spyder data, edited using the SpyderGUI. The Spyder objects are embedded inside the spydermap, which performs some processing operations on them. These operations are defined by the spyderhive on which the spydermap is based. The action2 spydermap, deriving from spyderdicthive, simply puts the Spyder action objects in a dictionary. The action1 and action2 spydermaps, deriving from “spyderhives/action1hive” and “spyderhives/action2hive”, split the action objects into two dictionary items (sound and animation). This operation is defined by the custom spyderhives. In case of action3, each dictionary is contained inside a separate bee called a drone.

The “action1”, “action2” and “action3” systems are used in the main hivemap “layers.hivemap”. The “layers.hivemap” contains a copy of each of the three systems. These systems are connectible. In the hive system, workers are connected by declaring antennas (inputs) or outputs. However, antennas and outputs can also be declared for hives. In other words, because each of the hivemaps has antennas defined, they are treated by HiveGUI like a worker: you can add additional (independent) copies of action1,action2 or action3 by selecting its name on the “Workers” panel on the left, under the “hivemaps” menu tab, and connect them. For more information on using the HiveGUI, see the “Hello World” tutorial.
Action2 is an integrated system, that just has an “actionplay” antenna. For action1 and action3, sound and animation are invoked separately. If you don't want this, you can use a wrapper that integrates the two into a single antenna: such a wrapper has been created for action1 as the “action1a” hivemap.

In “layers.hivemap”, the “W” key is bound to the “walk” action of action1, and “TAB” to the “jump” action. The “R” and “SPACE” keys are bound to action2, and “S” and “C” to action3. You start up the hive by executing “layers.py”, press the keys to invoke the actions, and press Escape to end the hive.

This is not the place to explain how the connections in hivemaps and workermaps trigger at runtime. This is explained in the Hello World tutorial, in the “Worker segments” paragraph below, and in the bottom-up section (“Push versus pull connections”) of this chapter. A complete overview with examples is given on page 29-33 of the Hive manual.

Translating the visual layers into text

The “layer1” representation above is almost completely at the visual level, using hivemaps, spydermaps and workermaps. Only the drones, the data models and the somelibrary.py stubs are non-visual. However, a visual representation is just data, and it has to be translated into text, as Python instructions, before they can be executed. Therefore, when running the layer 1 main script, all of the hivemaps, spydermaps and workermaps are translated into a textual representation: in the same order, these representations are hives, spyderhives and workers.

For hivemaps and spydermaps, the visual-to-text translation is internally performed by the hive system when the main hive script is started. However, this is not so for the workermaps. The hive system cannot read workermaps directly, but only in their text form, as workers. The visual-to-text translation is performed by the WorkerGUI: when you press the Generate button, a window containing the Python code pops up. In “layer1”, this has been done for each workermap, and the generated Python code has been saved as .py file in the “workers” subfolder. This is what is imported when the hive system starts. The reason for this semi-automatic translation step is that custom workers typically contain custom snippets of Python code. If these snippets contain a mistake, this is reported by Python with a line number etc., making debugging much easier from the Python representation than from the workermap. In the future, when debugging is integrated with HiveGUI/WorkerGUI, it may no longer be necessary to do it like this.

For hivemaps, spydermaps and workermaps, you have the choice to define it either visually, or directly as text within Python. Now, to explain how the hive system works, we will slowly replace the visual representations with Python code; then, we will be change the code from Python-using-the-hive-system to stand-alone Python, that does not use the hive system at all.

Layer 2

In layer 2, the workermaps/ directory has simply been deleted, leaving just the generated code in the workers/ directory. Comparing the workers and the layer1 workermaps shows the following features:

- In Python, creating a worker is done by subclassing bee.worker and defining segments inside the class definition.
- Some Python-level segments (bee.segments.*) correspond to the segments in the WorkerGUI. At the Python level, a segment is of the form:
  identifier = segmenttype(meta-parameters)(parameters)
  The Python-level identifiers correspond to the identifiers in WorkerGUI, and the segments (variable, transistor, buffer) are the same as the segment types (segments.variable, segments.transistor) in the WorkerGUI. The parameters and meta-parameters also correspond to what is editable in WorkerGUI.
- Other Python-level segments correspond to connection lines in the WorkerGUI. “connect” segments correspond to connections between “inp” and “outp”. “trigger” segments define connections between “on_output”, “on_update”, “output” and “update”; while “pretrigger” connects to “pre_output/pre_update” instead of “on_output/on_update”.
- The “custom_code” WorkerGUI segments do not exist in the Python class, their contents are literally inserted into the code.

All of these features we will also see later, when converting hivemaps to hives.

Worker segments
Here is a short overview of the segments:

- State-holding segments: *buffer* and *variable*. These hold a value that can be accessed and modified from Python (accessible as self.<segmentname>). They can also receive values from other segments, and send values to them. A connection can be either in *push* mode, where the initiative lies with the sending segment, or in *pull* mode, where the initiative lies with the receiving segment. Buffers can be either push or pull, sending and receiving in the same mode. Variables receive push input and give pull output. State-holding segments emit triggers when they receive input or are asked for output.
  For more information on the push/pull execution model, see the bottom-up section below, the Hello World tutorial, and/or the Hive manual, page 29-30.
- *startvalue*: this segment creates a start value for a state-holding segment. In WorkerGUI, this is an editable attribute of those segments.
- *parameter*: this segment exposes a state-holding segment as a parameter to build the worker. In WorkerGUI, this is an editable flag of those segments. When the worker is embedded inside a hivemap, the value becomes editable in HiveGUI.
- *transistor*: when triggered, converts pull input into push output.
- *connect, trigger, pretrigger*: connects segments to each other.
- *triggerfunc*: this segment creates a method to trigger segments (mostly transistors and buffers) from Python. The triggerfunc name is the name of the method. In WorkerGUI, this is an editable attribute of those segments.
- *modifier*: when triggered, executes arbitrary Python code. A modifier is a Python method, having access to the worker object via the “self” argument.
- *antenna* and *output*: Declaring these segments exposes a trigger or a state-holding segment to the outside world. Note that the antenna and output themselves do not possess state, and do not exist at runtime; they are merely entry points for connections to the outside world.
- *weaver* and *unweaver*: Rarely used. A weaver combines multiple push values into a tuple. An unweaver extracts multiple pull values from a tuple.
- *operator*: rarely used. Converts push input to push output by executing arbitrary Python code, without accessing the worker object.

For more information on segments and their execution, see the Hive manual, page 29-30.
For more information on using the WorkerGUI, see the Hello World tutorial.

Layer 3
Now we will convert one of the systems, action3, from hivemap to hive. In the layer3 folder, you will see that the action3 hivemap has been removed from the hivemaps/ folder, and that a Python implementation of the action3 hive is in the workers/ folder.

The code is very straightforward, following closely the layout of the hivemap. Again, the pattern is the same as for creating a worker: defining a new class from a base class and throwing components in there, and connecting them. Instead of segments, we are throwing workers, drones and configuration hives into the class. As a base class, we choose not bee.worker but bee.frame, a variant of bee.hive that shares all sockets and plugins (which basically means, all drones) with and from the parent.

Note that the *configuration* of action3 is still in visual form (the action3spydermap), and is embedded inside the action3 hive. Actually, locating and loading and embedding the spydermap is almost half of the code (line 3-17 and line 34-37).

For the HiveGUI, there is no distinction between a (sub-)hive, with its antennas/outputs, and a worker (likewise, a sub-hive without antennas/outputs is treated pretty much like a drone; see the Hive manual page 22,38 and 59 for more details on sub-hives) . Therefore, the Python code for our hive is put in the workers/ folder to be discovered by HiveGUI. In “layers.hivemap”, using the HiveGUI, you can simply replace the action3 hivemap with the action3 'worker'.

Layer 4
Now we are converting the main hivemap, “layers.hivemap”, into a Python-defined hive (“layershive.py”). The main script “layers.py” is adapted to embed the main hive instead of the main hivemap.
So, instead of a top-level hive embedding a main *hivemap* embedding a hive and two hivemaps, we have the same top-level hive embedding a main *hive* embedding a hive and two hivemaps. Therefore, the syntax to load and embed a hivemap inside a hive is now moved from “layers.py” to “layershive.py”, embedding the two remaining hivemaps “action1” and “action2”. As you can see, you can easily make a connection from a hive into a hivemap, but the syntax is a bit different and verbose. For the rest, “layershive.py” is a straightforward port of “layers.hivemap”.

Layer 5
Now it is time to shift the focus to the configuration of the system. Configuration in the hive system is done using Spyder objects, which are structured data objects: in our case, the Spyder objects define the sound files and animations that are to be played. In the “spydermodels” directory, you will find the “characteraction.spy” file with the (very simple) Spyder data models that we are using in the spyderhives: CharacterAction, CharacterActionItem, AnimationItem and SoundItem.
Configuring hives in Python is done by adding Spyder objects to spyderhives. Spyderhives are collections of zero or more Spyder objects, plus machinery to interpret these objects. The visual equivalent of spyderhives are spydermaps: they can contain Spyder objects, but no machinery. Therefore, a spydermap is always based upon an underlying spyderhive that provides the machinery.

In layer 5, the configuration for action1 and action2 is converted from a spydermap to a spyderhive. Previously, the Spyder data for action1 was defined in “action1spydermap”, based on the spyderhive “action1hive” (this is shown in the big combobox at the top of the SpyderGUI window). Now, there is instead a new spyderhive “action1conf”, based on the same “action1hive”, adding the same data, but now in Python. (An alternative would have been not to bother with a new spyderhive, instead adding the Spyder data simply to the existing “action1hive” spyderhive)
Since you can't embed a spyderhive directly in a hivemap, there is now an empty “action1conf” spydermap, based on the spyderhive of the same name. This “wrapper” spydermap is what is now embedded in the action1 hivemap in layer 5.

Configuration of the action2 system
The conversion in layer 5 from spydermap to spyderhive of the action2 data is done in the same way as

described for action1 above. It is a simple system to configure: therefore, it is now explained in more detail how the configuration of the action2 system works. The action1 system is (a little bit) more complex, and uses some additional features of Spyder, as explained later.

Of the four data models in “characteraction.spy”, CharacterAction is the one that is used in the action2 system. As you can see in characteraction.spy, a CharacterAction simply consists of two strings, one for sound and one for animation.
In layer 4, CharacterActions are placed in the “action2spyderdict” spydermap. This spydermap is very simple: it just defines the CharacterActions and their identifiers (the names of the nodes). As any spydermap, it is based on a spyderhive. However, the configuration for action2 is so simple that it doesn't need special machinery, and is based on one of the vanilla spyderhives that come with the hive system: the *spyderdicthive*. (Again, the fact that “action2spyderdict” is based on the spyderdicthive is shown on top of the SpyderGUI.) The spyderdicthive is so simple that it just takes the defined Spyder objects (in this case, the CharacterActions), and their identifiers, and inserts each CharacterAction under its identifier into a provided dictionary. The action2spyderdict is embedded inside the action2 hivemap, which provides this dictionary: “actiondict”, a CharacterAction-containing dictionary worker (dragonfly.logic.dictionary; in the metaparams tab, you can see that *type = CharacterAction*). The embedding action2 hivemap also defines a special bee, a so-called *wasp*, which inserts “actiondict” as a parameter to “action2spyderdict”. Therefore, the action2spyderdict fills up the “actiondict” dictionary with CharacterAction objects. More information on wasps is in the Canvas tutorial. At run-time, a CharacterAction is retrieved from “actiondict” inside the action2 hivemap. The hive system comes with a category of workers, *block workers (*dragonfly.blocks), to manipulate Spyder objects at run-time, and the action2 hivemap uses them to access the CharacterAction attributes, feeding them to the play workers. This how the action2 system works.
In layer 5, the data has simply been moved from the spydermap to a spyderhive, and nothing else has changed.
The next step is to analyze the action1 system in the same way. But before that, a general explanation of Spyder is needed.

Spyder
Spyder is a generic framework for data modelling in Python. It is the oldest part of the hive system, its development started around 2006, and its documentation can be found at spyder.science.uu.nl. A number of scientific web servers have been built with Spyder, most notably the HADDOCK server (haddock.science.uu.nl).
The central idea of Spyder is that you describe the structure of your data model just once: the data type of its members, validation rules, perhaps some formatting options. Then, useful things are being generated automatically: a Python class to hold the data, a parser to load and save the data, a HTML form, a Qt/Gtk GUI, a model-view-controller chain, anything that can make use of a structured description of the data model to manipulate it.
Spyder models are hierarchical (you can embed them inside other Spyder models), self-contained, and they are easy to manipulate in Python (you can build them from a keywords, a list, a dict, a string and/or from other Spyder objects). Also, Spyder has a feature called ATC (acquisition-through-conversion), which means that you can define a method (e.g. “spam”) for a Spyder model A, and a converter from another Spyder model B to A, and that model B acquires the method through conversion: B.spam() is automatically expanded to B.convert(A).spam(). ATC is heavily used in

spyderhives, including in our example, as we will see below.
In the spydermodels directory, you will find the characteraction.spy file containing the Spyder data models that we are using in the spyderhives: CharacterAction, CharacterActionItem, AnimationItem and SoundItem. In Python, a .spy file can be imported in the normal way (as if it was a .py file, with "import characteraction") but only after you do "import spyder". Importing Spyder also creates a module "Spyder" containing all data models: Spyder.CharacterAction, Spyder.AnimationItem, etc, plus all pre-defined data models that are bundled with Spyder and the hive system. For more general information on using Spyder, see spyder.science.uu.nl.

Configuration of the action1 system
The action2 system has only a single dictionary, "actiondict", to store action data. However, when you look in the action1 hivemap, you see that it has not one but two dictionaries: "animdict" and "sounddict". These dictionaries do not contain Spyder objects, but simple strings. Using wasps, both of these dictionary workers are passed as parameters to the action2 configuration spyderhive.

In general, a spyderhive works as follows: on every Spyder object inside the spyderhive (or the spydermap), the make_bee() method is invoked, and the result of that method is added to a conventional hive. The Spyder object identifiers (names of the nodes) are ignored, which is why they are a bland "attribute_1", "attribute_2" in the action1 spydermap.
For action1, the type of Spyder object that we are adding is CharacterActionItem, consisting of a string called "identifier", and a CharacterAction called "action" (see characteraction.spy). Since a CharacterActionItem does not by itself have a make_bee() method, we must provide one. This is the "machinery" discussed above, which the underlying spyderhive (action1hive) must provide.
Looking in the action1hive.py source code, we see that make_bee() methods are indeed provided... for SoundItem and AnimationItem! These data models describe individual sounds and animations. In their make_bee() methods, an init bee is created. An init bee is a proxy for something that does not yet exist, in this case the "sounddict" or "animdict" parameter. When the hive is initialized (main.init() in layers.py), the operation on the init bee (inserting a new item into the dictionary) is performed on the sounddict/animdict instead.
But how does the hive system invoke make_bee() on a CharacterActionItem if CharacterActionItem doesn't have the method? By exploiting Spyder's ATC feature. In characteraction.spy, it is defined how to split a CharacterActionItem into a SoundItem and an AnimationItem. More precisely, a CharacterActionItem is converted into an ObjectList of two elements. ObjectList is a generic list for Spyder objects, and it has a special converter defined so that ObjectList.spam() becomes [ObjectList[0].spam(). ObjectList[1].spam(), …]. In summary, CharacterActionItem.make_bee() gets automatically expanded into CharacterActionItem.convert(ObjectList).make_bee(), which gets expanded into [AnimationItem.make_bee(), SoundItem.make_bee()]. The resulting list of two init bees gets added to the hive.
Doing it this way means that sound and animation are decoupled. In addition to CharacterActionItems, you can also add individual SoundItems and AnimationItems to the action1 spyderhive or spydermap (also for the action3 system), and the spyderhive will know how to deal with them.
For more examples on spyderhives and ATC, see page 45-49 of the Hive manual.

What is a hivemap exactly?
The hivemap is the data format of the HiveGUI, as simple as that. But if you open it in a text editor

(Wordpad or Emacs or something), you can see that it *also* Spyder. A hivemap file is a so-called .web file, a Spyder object written to a file. The content of a .web file looks a bit like XML or JSON, but it is also valid Python syntax to build a Spyder object in Python.
The fact that hivemaps are Spyder objects is even more obvious when you look at the code in "layershive.py":

```
action1hivemap = Spyder.Hivemap.fromfile(action1hivemapfile)

from bee.spyderhive.hivemaphive import hivemapframe
class action1hivemaphive(hivemapframe):
  hivemap = action1hivemap
```

In other words, not only is a hivemap a Spyder object, it is placed in a subclass of a spyderhive.
Hivemap hives are just a special case of spyderhives! The HiveGUI is just a special-case editor for the Spyder.Hivemap data model, and hivemaphives or hivemapframes are just spyderhives that give a make_bee() method to Spyder.Hivemap, a make_bee() method that returns an entire hive instead of a single bee.
For more information, see the Hive manual page 57-59. Note that some details about the HiveGUI are quite a bit out of date.

Layer 6
In layer 6, the action1 and action2 hivemaps are replaced by hives. The hives (action1.py and action2.py) are in the workers/ directory.
Since hives (unlike hivemaps) can embed spyderhives directly, there is no more need for the empty actionconf spydermaps. Also, since passing parameters/arguments to a bee is easy in Python, there is no more need for any wasps. Finally, the hive code in "layershive.py" is now a bit simplified: inside a hive, it is easier to embed and connect to a hive than to a hivemap.

Layer 7-9 are based on the high-level layers (hives, spyderhives) of the hive system

Layer 7
In layer 7, we get rid of the last visual representation: the action3 spydermap. Now, the system is completely defined in Python, without any maps edited with the HiveGUI, WorkerGUI or SpyderGUI.

In terms of configuration, the action3 system works very much like action1: here, too, there is a Spyder action object that gets split into action and animation strings that are added to two dictionaries. However, here the dictionaries are not in a worker, but they are maintained by the components for playing sound and animation. In the hive system, such components are implemented in the form of drones. Drones are coded in Python, they do not form explicit connections with workers, hives or other drones, but they can communicate in the form of sockets and plugins. In workers/action3drones.py, you can see the implementation of the two action3 drones, "animationmanager" and "soundmanager". Each drone has a method "add_animation" / "add_sound", and a method "play". The "play" method is exposed through a plugin with a name ("play", "sound"/"animation") . A worker that wants to play a sound or animation can do so by declaring a corresponding socket with the same name, this is done by the workers in workers/action3_play_sound.py and workers/action3_play_animation.py. The

add_animation/add_sound methods are invoked by the init bees in spyderhives/action3hive.py, so that every CharacterAction object in the spyderhive is translated into a call to animationmanager.add_animation() plus a call to soundmanager.add_sound().

Layer 8
Layer 8 is just a reorganization of the files. Since we are no longer using the HiveGUI or any other GUI, there is no reason to maintain the defaultproject directory structure. The hivemaps/, spydermaps/ and workermaps/ directories have been removed. The spyderhives/ directory, containing all configuration, has been renamed to “conf”. The Python files in workers/, as well as characteraction.spy, have been moved to the main directory. The main files “layers.py” and “layershive.py” have been adapted accordingly.

Layer 9 and 10
In layer 9, the configuration via Spyder objects performed using the action3 spyderhives is performed manually. The spyderhives conf/action3hive.py and conf/action3conf.py have been removed, and instead of this, a conventional hive is created containing the init bees required to fill the sound/animation dictionaries.
In layer 10, all configuration via Spyder objects is eliminated. All action1 and action2 spyderhives have been replaced with conventional hives containing init bees. The CharacterActionItem, SoundItem and AnimationItem data models have been deleted. The only remaining Spyder model is CharacterAction, because CharacterAction objects are in the action2 action dictionary at runtime.

Layer 10-13 are based on the middle layers (workers, drones) of the hive system

Layer 11 and 12
In layer 11, the entire conf/ directory has been removed. The configuration hives action1hive, action2hive and action3hive in the conf/ directory (translated from the spyderhives/spydermaps) have been integrated into the worker-like hives of the same name (translated from the hivemaps) in the main directory. All files are now in the main directory.

In layer 12, all three systems action1, action2 and action3 have been translated from worker-like hives to workers. The workers to play sound and animation (in actionworkers.py) have been merged into these workers. The action3 drones have been moved from the action3 hive into the main hive.

The action1 worker initializes and fills its sound and dictionaries at place() time, there are no more init bees. For the rest, its code is the sum of the former play workers (play_animation and play_sound in actionworkers.py). It imports somelibrary to get the functions to play sound and animation, and has two antennas to do so. The main difference with the former play workers is that these antennas do not take filename strings, but instead identifiers, which are mapped to strings using the dictionaries.

The action2 worker also initializes and fills its action dictionary at place() time, there are no more init bees. It has a single antenna, taking an identifier, which is mapped to a CharacterAction object using the action dictionary, and then the somelibrary functions are invoked on the attributes of the CharacterAction object.

The action3 worker defines sockets to interface with the soundmanager and animationmanager drones, to get their play methods. It also defines sockets to get access to their dictionaries, or at least, to get access to the methods adding a sound/animation item to their dictionaries. These sockets are not yet filled at place() time. Therefore, the dictionaries cannot be filled in the same function, and a special _init method is created that fills the dictionary. Using a (“bee”, “init”) plugin, this _init function is registered with the top-level hive, so that it is invoked at main.init() (the same mechanism is used by init bees).

Layer 13
In layer 13, the main hive (layershive.py) has been reduced to a single worker (mainworker.py). The action1/2/3 workers, the connections between those workers and the main hive, and the action3 drones have all been moved to the top-level hive (in layers.py). Note that the top-level hive is now the only hive.

Because of this reduction, the main hive can no longer use the keyboard sensor workers from the dragonfly library, and has to tap directly into the event system.

Events
All hives have event streams. All top-level hives, including the console hive, have in their evin (event-in) stream an event handler that receives key presses. They also emit a single “start” event upon main.run() . Previously, we were using dragonfly.io.keyboardsensor_trigger and dragonfly.sys.startsensor workers to catch these events, but since workers cannot be embedded inside other workers, we now have to register directly with the event handler (basically replicating the hooks from those dragonfly workers' source code). The way to do so is to define a matching pattern and register a listener under that matching pattern as a plugin named (“evin”, “listener”).
For the “start” event, we want a callback to be triggered when exactly that event fires. Therefore, the pattern is (“trigger”, <callback>, “start”). Key press events have the form (“keyboard”, “keypressed”, <key>). Therefore, if we want to catch all keys, we want all events starting with the leader sequence (“keyboard”, “keypressed”), chopping off that leader sequence, and feeding the remaining part of the event (containing the name of the key) to the callback function. This makes the pattern (“leader”, <callback>, (“keyboard”, “keypressed”)). If we were interested in the keypresses for a single key, say the SPACE key, the pattern would have been (“trigger”, <callback>, (“keyboard”, “keypressed”, “SPACE”)).
For more information on events, see page 23 of the Hive manual.

Layer 14-18 are based on the low-level layers (drones, sockets/plugins) of the hive system

Layer 14
Until now, we have been relying on the console hive to detect key presses for us. In layer 14, key presses are detected manually, allowing us to abandon the console hive for the simplest top-level hive, the inithive. To do so, the difference between top-level hives and normal hives will be explained.
All hives have the following methods: getinstance(), build(), place() and close(). If a hive is embedded inside another hive, calling parent.getinstance() will also instantiate the children, parent.build() will also build the children, and the same for place() and close(). If a top-level hive needs to implement more methods, it has to do so via a super-drone called an app. Normally, if you embed a drone in a hive

you can call a method from Python simply as hive.drone.method(). If you declare that the drone serves as the hive's app, you can simply call hive.method() instead. For more information on apps, see page 26 of the Hive manual.

The consolehive and other top-level hives have an app that defines two important methods: init() initializes startvalue segments, init bees, and everything else that registers a (“bee”, “init”) plugin. The run() method fires up a separate thread (therefore, non-blocking) that constantly listens for keypresses, creates events, and sends them to a scheduler. The run() method then enters an infinite loop: on every iteration, a “tick” event is generated, and then all keyboard events since the last tick are flushed from the scheduler into the event handler. After that, the top-level hive is free to do other things, for example to render a frame of animation. The loop goes on until an exit actuator worker has been activated. Every top-level hive contains a keyboard sensor for Escape connected to an exit actuator, so pressing Escape usually does the job.

All of this is overkill for our current example. We are not listening for any other events than key presses (and the start event), and our program does nothing else than responding to key presses (it would of course be a different story if we were actually playing some sound or animation!). Therefore, it is fine if the program blocks its execution until the next key has been pressed. Therefore, for a clearer demonstration, layer 14 contains a simplified implementation of the console hive's key detection loop, not using any event streams, simply invoking the mainworker's keypress method directly. The scripts getch.py and keycodes.py are taken from dragonfly, and they are invoked in a simple mainloop that blocks until the next key press, and stops at ESCAPE. The mainworker.start method (previously triggering upon the “start” event) is now invoked manually from layers.py. The top-level hive has changed from consolehive to the inithive, which has a much simpler app that only defines an init() method.
Layer 14 no longer uses any functionality from dragonfly.

Layer 15
In layer 15, all four workers (mainworker, action1worker, action2worker and action3worker) have been replaced by drones. All antenna segments on the action workers have been replaced by plugins that expose the play methods. All outputs on the mainworker have been replaced by sockets that catch these methods, and all segment triggers on the mainworker have been replaced by invocations of these methods. In the hive system, worker antennas and outputs *are* finally translated to sockets and plugins, but not exactly in the way described above. For a more precise explanation, see page 31 of the Hive manual.
Since there are no more startvalue segments (or any segments at all), there is no more need for the init() function, and the inithive class has been replaced by the vanilla bee.hive class. Previously, the action3 worker was relying on the init() function to fill the dictionary. Now, the action3 drone fills the dictionary the first time that one of its playing methods is invoked.

Layer 16
In layer 16, the sockets and plugins have been adapted. Instead of exposing their methods, the drone objects themselves are exposed as plugins, and captured via sockets. The methods are then invoked on the captured child objects. This is a more traditional way of sharing parent-child functionality in object-oriented programming.

Layer 17 and 18
In layer 17 we are no longer using bee.hive or bee.drone, or anything else from the bee module. All drones (including the animationmanager and the soundmanager) are now “components” that are just plain Python classes, although they still expose their functionality as libcontext sockets and plugins in their place method.
The bee.hive baseclass of has been replaced by a generic “pseudo-hive”, a Python class that implements the most important methods of a hive: build(), place() and close(). Since the hive does not contain subhives, and none of the components take parameters upon construction, there is no need for a getinstance() method. The absence of antennas, outputs, connections, init/configuration bees, attributes and parameters (we have used all of these features in higher-level layers!) further simplifies the code.

In layer 18, the generic (pseudo-)hive as a base class for mainhive has been abandoned. There is now a single specific mainclass that performs build(), place(), close() for the specific components that are embedded inside it.

Layer 19 and 20

In layer 19, libcontext sockets and plugins are no longer used. The mainclass now contains a specific code section “#connecting the components”, in its __init__ function, providing the maincomponent with references to the action1/2/3 components, and the action3 components with references to the manager methods.
Finally, in layer 20, the CharacterAction Spyder model is replaced with a minimal Python class with the same attributes. No more features of the hive system are used.

Bottom-up

Layer 20
Layer 20 contains a pure-Python object-oriented implementation that does not use the hive system at all. Each of the three design choices for actions has been implemented separately, as action1, action2 and action3.
Action1 (action1component in action1.py) has two embedded dictionaries, one for sound and one for animations, which are filled at start-up. Users can play sounds and animations independently. To do so, they have to specify an identifier (dict key), and the corresponding sound or animation is retrieved from the dictionary, and sent to a play function. The play functions are imported from “somelibrary.py”, which provides them as stubs (a simple text message is printed).
Action2 (action2component in action2.py) also imports the play functions from “somelibrary”, but there is only a single action dict. The dict contains CharacterAction objects, a simple class that is defined in CharacterAction.py, with attributes for the sound file and the animation.
In the action3 system, the sound and animation dicts are stored on separate soundmanager and animationmanager components. These components have methods for adding new sounds/animations, as well as playing them. The action3 component requires references to these methods (specified using the set_XXXfunc methods). The first time that a sound/animation is played, action3 configures the managers by filling their dictionaries (with values for “swim” and “crouch”), this is done in the special _init (not __init__ !!) method.

The three systems action1/2/3 are invoked from the maincomponent. The maincomponent receives keypresses in its keypress() method, and then plays an animation on of the three systems depending on which key was pressed.
The keypresses are provided by a simple mainloop function in the main script, layer.py, blocking the execution of the program until the next keypress. Pressing the Escape key breaks the mainloop. The actual keypress detection is done by getch.py, a platform-independent script inspired by the Python cookbook, lifted from the hive system source code.
All components are embedded inside a main class (mainclass in “layers.py”). In its __init__ function, first all components are created. Then, all connections between the main component and the action1/2/3 components (and between the action3 component and the managers) are established.

Layer 19
In layer 19, the CharacterAction class is replaced by a Spyder data model of the same name. Spyder is the data modelling framework of the hive system. It allows you to describe the attributes of a static data structure in a .spy file, and a Python class is generated automatically.
For now, the Spyder model does the same as the previous CharacterAction Python class, but the additional features that Spyder provides, such as saving/loading from a file, data conversion and GUI form generation, are necessary to create visual layers and configuration layers in the hive system later on.
The definition of the data model is in characteraction.spy. After importing the “spyder” module, .spy files can be simply imported in Python in the normal way.
To get information on a specific Spyder model, for example AxisSystem, simply execute “import spyder; from Spyder import AxisSystem; help(AxisSystem)”. Note the capitalization: first “import **s**pyder”, then “from **S**pyder import ...”.
For more information on Spyder, see the “Spyder” paragraph in the top-down section above.

Layer 14-18 are based on the low-level layers (drones, sockets/plugins) of the hive system

Layer 18 and 17
In layer 18, the low-level module of the hive system is introduced: libcontext. All components have now a place() method where they provide or request functionality, via sockets and plugins. The action3 soundmanager component in “action3components.py” declares that it can provide the functionality for playing sound, via a plugin called (“play”, “sound”). The name of this plugin is arbitrary and completely up to the programmer. Then, the action3 component declares that it needs this functionality by declaring a (“play”, “sound”) socket. The action3 component also has sockets for adding sound, playing animation adding animation (the latter two being provided by the animationmanager component). Conversely, the action3 components provides itself as a “action3” plugin, and is used by the maincomponent, which has “action1”, “action2” and “action3” sockets.
Plugins and sockets are always declared inside a *context*, which has a name. When the context is closed, all plugins are automatically connected to sockets of the same name within the context. Our system currently has only a single context “main” (self._contextname in mainclass). Sockets and plugins in different contexts do not see each other, unless they are explicitly shared or exported (which is beyond this tutorial). For more information on libcontext, see the Hive manual page 16-18.

The advantages of using sockets and plugins is are that connections are made implicitly: the "connections" code section of the mainclass in "layers.py" is gone. Also, *functionality* is connected, not specific objects. You can decide to merge the sound manager and the animation manager into a single component and with libcontext you can just do so, you don't have to change any connection code anywhere else.

In addition, it allows us to generalize the mainclass. This is shown in layer 17. Here, the mainclass is split into two parts:
First, a generic part, the "pseudohive" class, that iterates over all components and instantiates them. Its build() method creates the libcontext.context; its place() method selects the context, and invokes the place() method of all components (declaring the sockets and plugins); and its close() method closes the context.
Second, a specific part, the "mainhive" class, that derives from "pseudohive", and that only defines the names and classes of the components.

Layer 16
The layer 17 pseudohive class is a truly generic way of composing a composite class (a hive) out of components (the hive system calls them bees). Because the connections form automatically, there is no manual section of connection code, like there is in layer 19. There is no manual code section of any kind: just add the components and then do instantiate => build => place => close.
The pseudohive class is simple enough, but still has a few limitations:

- The components cannot have parameters (in the sense of: arguments to their __init__ function). (We could add them, by adding argument lists/dicts (args and kwargs) to the component class type specification that is currently in the components dict)
- There is only one hive. You cannot embed (pseudo)hives hierarchically inside other (pseudo)hives
- The use of sockets and plugins *forces* all connections to be implicit. It is Python Zen that explicit is better than implicit. If you could add explicit connection objects to the component mix, then you would have the choice.
- The syntax of the connection dict is just ugly.

All of these limitations have been removed in layer 16. The mainhive class now derives from bee.hive, and specifying components is as simple as "componentname = componentclass(parameters)" inside the class definition. As before, independent copies are instantiated every time. To make this magic work, all components now have bee.drone as their parent class, and are therefore now called "drones". For the rest, their code and functionality is exactly the same: in hive system terminology, drones simply are bees (components/nodes) that communicate/connect exclusively through sockets and plugins, like they already did in layer 17 and 18.

For more information on drones and hives, see page 18-22 of the Hive manual. Unlike the pseudohive, I do **not** recommend studying the bee.hive source code in the bee module; removing all the above limitations also makes the bee.hive code extremely hairy.

Layer 15

In layer 15, the sockets and plugins have been slightly adapted. The action1/2/3 drones no longer supply as plugin a reference to themselves, but rather their functionalities: (“action1”, “play”, “animation”) and (“action1”, “play”, “sound”). You are now free to do something like splitting the action1 drone into two drones, one for animation and one for sound. You could also add more drones like (“action1”, “play”, “video”) or something like that.

Layer 14
In layer 14, the drones have been replaced by another type of hive system bees: *workers*. Maindrone has become mainworker, action1drone has become action1worker, etc. They are now explicitly connected with each other inside the mainhive, using bee.connect. The only drones that are still drones are the action3 manager drones. They will stay drones until the very end: singleton “manager” components such as these are best represented by a drone.

It is not so easy to explain in a few words how workers work and how they relate to drones. Workers can connect implicitly, via sockets and plugins, just like drones can, but they also support explicit connections. These come in two flavors: bee.connect is to connect workers to each other, inside a hive (like the connections between mainworker and action1worker), wheres bee.segments.connect is to make connections *within* a worker. Workers themselves are miniature hives, consisting of connected miniature elements (i.e. nodes/components/mini-workers/you name them...) called *segments*.

A quick overview of all worker segments is given in the top-down section above. For a thorough explanation of workers and segments, you really should read page 29-33 of the Hive manual, and the summary on page 40. Alternatively, you could gloss over workers for now and see if it makes more sense once we are at the visual level (workermaps).

One thing that workers, unlike drones, do not have, is a user-defined constructor (__init__ function). Therefore, the action1/2/3 configuration code (the code that fills the sound/animation/action dictionaries) has been moved to the place() method for action1 and action2, and to a separate _init() method for action3. (This is because action3 cannot perform the configuration before it has received a reference to the manager functionalities “add_sound” and “add_animation”, and it won't have them until the hive is closed. This problem will be solved more elegantly with init bees in layer 11).

The action3 _init() method, and workers in general (more specifically, workers with “startvalue” segments) depend on an initialization step. The inithive provides this step: when its init() function is called, all functions registered as (“bee”, “init”) plugins are called (in our case, just the action3 _init function, you can see the registration in action3worker.place). Therefore, in layers.py, the base class for the mainhive has been changed from bee.hive to bee.inithive, and the init() method is invoked after close().

Push versus pull connections
Another important thing to understand about workers is the difference between push and pull connections. Connections are made via entry points called antennas and outputs: an antenna is an entry point for the upstream (sending) bee, whereas an output connects to the downstream (receiving) bee. Push connections are like function calls: the upstream bee sends data to the downstream bee (or no data, in case of a trigger). Pull connections are like requests or attribute accesses: the downstream bee

asks the upstream bee for a value.
Pull antennas should have exactly one connection (they must know where to get their data from), while pull outputs can have zero, one or many connections (a sign can be read by any number of people). Likewise, a push antenna can have any number of connections (a satellite dish can receive any number TV channels). In neither case does it matter in which order the connections have been established. However, push outputs can also have any number of connections, but the order does matter here: the first connected downstream bee receives the signal first. This may or may not affect the final execution result of your hive or hivemap, so be aware of this.
Finally, segments and connections in the hive system are statically typed. As a Python programmer, you may not like this, but it is the only way to auto-generate visual editable representations. Deal with it.

Layer 10-13 are based on the middle layers (workers, drones) of the hive system

Layer 13
Up to now, keypresses have been detected manually, blocking the program until a new key has been pressed. In general, it would be rather annoying to worry about this all the time, especially when we want non-blocking keyboard detection (so the hive can do other things in between key presses, such as rendering, or responding to mouse clicks). For this, and many other functionalities, the hive system comes with a standard library called *dragonfly*, which contains hives, workers and drones to take care of common tasks. So, from now on, we will use dragonfly and approach events like keypresses from a higher level; but the manual key detection from layer 14-18 serves as a reminder that these things can be bypassed, you always have the option to do such things yourself. In general, the dragonfly source code is a good place to study (***much*** better than the hairy code in bee and libcontext) and provides many examples of how features can be presented in a node-based API.

In layer 13, the mainloop and all keyboard detection code has been removed. Instead, the mainhive's baseclass has now been changed from inithive to dragonfly's consolehive. The consolehive listens for keypresses in a non-blocking way, converts them to keypress events, and flushes them to the event handler on every tick. More details are given in the top-down section under “layer14”.
The dragonfly consolehive is a top-level hive, meant to embed a hive running in a command-line terminal, providing keyboard events. The dragonfly library contains several other top-level hives: the commandhive supports command-line strings entered via the terminal, the blenderhive is to run a hive under the Blender Game Engine, and the pandahive provides similar bindings to the Panda3D game engine.

All top-level hives contain a scheduler for events. You are not really supposed to trigger workers (or other bees) manually from external Python code; instead, workers (sensors) that should be triggered should register a listener for certain events. Specifically, our mainworker should register a listener for keypress events. In addition, the mainworker prints “START” at the beginning of the program, so it should register a listening for the “start” event as well. This is exactly what is done by the mainworker in its place() function, using the (“evin”, “listener”) plugin. More information on events, event listeners and their syntax is in the “Events” paragraph in the top-down section.

Layer 12
Layer 12 re-implements the mainworker as a sub-hive, a hive embedded inside another hive. The hive

class chosen is bee.frame, which automatically imports/exports all plugins and sockets from/to its parent hive. “Mainworker.py“ is now renamed“layershive.py”, containing a “layershive” hive that is embedded inside the top-level hive (“mainhive” in “layers.py”). Instead of being a monolithic worker, the main “layershive” now consists of dragonfly workers that interface with the event system, print the “START” message, and activate the action workers, which are now embedded inside this hive. The action3 manager drones have also been moved from the top-level hive into this hive.
Many of these dragonfly workers (dragonfly.std.transistor, dragonfly.std.variable) are analogous to segments. These workers are actually meta-workers: a metaworker takes a type argument (“str”, “int” etc.), which creates a worker class, and this worker class is then initialized, with zero or more arguments. Creating your own meta-workers from Python is explained in the Hello World tutorial.
Finally, the connection points to the action1/2/3 workers are exposed using bee.output bees (not to be confused with bee.segments.output). This allows the new layershive to be treated like a worker: in the top-level hive in “layers.py”, connections are made between the layershive and the action workers, just like for the mainworker before.

Layer 11 and 10
In layer 11, the same is done for the action systems, re-implementing action1worker as action1hive, action2worker as action2hive, and action3worker as action3hive.
Most of the functionality, previously specified as Python code, is now delegated to dragonfly workers. The functionality consists of four parts: 1. the connection points, 2. the retrieval of the action/sound/animation from the dictionary, 3. the playing of the sound/animation, and 4. the configuration of the dictionaries. For action2, there is also 5. accessing the attributes from the CharacterAction Spyder model.
Part 1 is done the same way as for the mainworker/layershive above. Part 2 is also straightforward. Instead of Python dicts, the sounds, animations and actions are now stored in dragonfly.logic.dictionary workers.
Part 3, the actual playing, is the only part that is still implemented as a worker. There is now “actionworkers.py” with custom workers “play_sound” and “play_animation” that take a sound file / animation and pass it on to “somelibrary.py”. These are used by action1 and action2. Action3 uses a different set of workers (also in “actionworkers.py”), that take an *identifier* for a sound or animation and passes it on to the sound/animationmanager (which retrieves the sound/animation from its own internal dictionary and plays it).
Part 4, the configuration, is an extremely important part that will be discussed below. However, first part 5 is explained, how the action2 system accesses Spyder attributes, using block workers.

Block workers
In action2, dragonfly block workers are used to access the attributes of the CharacterAction Spyder object from the dictionary. As you can see, “get_action” is a dragonfly.blocks.getter worker that provides a pull output for every attribute of the Spyder model. Since CharacterAction has a Spyder.String attribute “soundfile”, it is available on “get_action” as a (“pull”, “String”) output.
Elementary Spyder types (String, Integer, Float, Bool) are subclasses of basic Python types (str, int, float, bool), and there is a special rule in the hive system that allows arbitrary connections between them. Therefore, it is allowed to connect get_action.soundfile, which is (“pull, “String”) , to transistor_3.inp, which is (“pull”, “str”).
An alternative syntax would be the tuple-form: *connect((get_action, “soundfile”), transistor_3.inp)*.

The tuple-form works even for nested data models, such as Spyder.AxisSystem (whose “origin” attribute is a Spyder.Coordinate, whose “x” attribute is a Spyder.Float) . For example, you can do:

```
getter = dragonfly.blocks.getter(“AxisSystem”)
t = dragonfly.std.transistor(“float”)()
connect((getter, “origin”, “x”), t.inp)
```

Configuration in the hive system

Replacing a worker with a hive has some important advantages. It allows you to work at a higher level, simply connecting prefab components (dragonfly workers) that take care of lower-level details (such as listening for keyboard events). It also brings some difficulties: unlike workers, hives do not lend themselves well for creating custom Python methods that can be hooked up with the system. For configuration the action/sound/animation dicts, up to now, we have been doing exactly that: defining a couple of initialization commands inside a custom method and hooking it up with the rest of the system inside a worker. However, to do it within a hive, we will need init bees.

Init bees (and their sisters, configure bees) are a way to proxy another bee that does not yet exist. For example, in the action1 hive, the “adict” init bee is a proxy for the “animdict” dragonfly.logic.dictionary worker. Within Python class definitions, all code is executed immediately. However, during the class definition, the animdict worker does not yet exist. To see why this is so, have a look again at layer 16 and 17 to remember that “animdict = ...” is just syntactic sugar around building a dictionary of component *classes*; the components are only *instanced* when the hive class is instanced later on. Therefore, what the init bee (“adict”) does, is to “buffer up” the operations on it (the addition of new dictionary items), and to execute them on the proxied bee (“animdict”) at initialization time, i.e. when init() is invoked on the hive. The same mechanism is used for the sounddict and for the action2 actiondict. In the action3 system, init bees are used to invoke the add_sound/animation methods of the manager drones.

Configure bees work exactly the same as init bees. The only difference between them is when the proxy commands are executed. Configure bees are executed upon place(). Therefore, they are used to do configuration that may lead to the placement of additional sockets and plugins. In contrast, init bees are executed at init(), when the hive has already been closed. Therefore, init bees can use functionality that has just been acquired through a socket, which isn't there yet at place() time. In many situations, either kind of bee can be used.

Configure bees are explained in the Hive manual page 43-44. This explanation is also valid for init bees. Also, when debugging, it is nice to know that the Python exception traceback for init bees and configure bees is rerouted, showing the trace as if the proxy commands were executed immediately.

With init bees and configure bees, the hive system stays true to its philosophy that a hive is *nothing more* than its components and their connections, and that a hive contains no other Python code than what is needed to assemble and connect the components. Even top-level functionalities like the consolehive inputhandler for key presses, or its eventhandler, are implemented with drones; you could subclass consolehive and swap out those drones, if you want.

Layer 10 splits each of the action hives into a main part (action1/2/3hive) and a configuration part (action1/2/3conf). A new directory “conf” has been created, and all action1/2/3conf has been moved to

there.
The configuration hive just contains the init bees. In action1 and action2, the configuration hive is embedded inside the main action hive, receiving the dictionary workers as parameters. In action3, the init bees operate not on the dictionary workers, but on the manager drones, and these are sent as parameters instead.

We have now reached an important turning point. Until now, we have been slowly replacing custom Python syntax with hive system syntax, first with low-level libcontext, then with drones and workers, and finally with hives consisting of prefab dragonfly workers. The focus has been on explaining the mechanics of all these hive system features.
We are now at the point that all non-hive system Python syntax has been reduced to a few trivial snippets: somelibrary.py, the do_play() and set_play() methods of the actionworkers, and the play() and add() methods of the manager drones. This is as far as the hive system can go.
From now on, the focus will no longer be on mechanics, but on configuration. First we will expand the usage of Spyder in the hive classes, since Spyder is the primary method of configuration in the hive system. Then, we will gradually move out of Python and up to the visual level. As you will see, these two things will be basically the same thing: the visual representations are just GUI-editable Spyder data layers that *configure* the system.
But let's start simple, by getting rid of the init bees.

Layer 7-9 are based on the high-level layers (hives, spyderhives) of the hive system

Layer 9 and 8
In layer 9, the action1 and action2 files in the conf directory have been re-implemented. Looking at action2conf, it looks quite different from layer 10. The init bees have gone, the Spyder objects are directly defined inside the class definition, and the base class has changed from bee.frame to bee.spyderhive.spyderdicthive. What is going on?

The concept behind spyderhives is simple. They allow Spyder objects to be directly placed inside hives, as if they were bees. At build time, the spyderhive then performs some operation on them. The spyderdicthive, which is used by action2conf, treats the Spyder objects as dumb data. The spyderdicthive relies on a parameter called “dictionary”, creates an init bee around it, and then adds each Spyder object to the dictionary, using its attribute name as key and the Spyder object as value. In other words, it boils down exactly to the code in layer 10.

For action1conf, the story is a bit more complicated. “characteraction.spy”, the file that defines our Spyder model CharacterAction, has been expanded with three more models: CharacterActionItem, SoundItem and AnimationItem. It is also stated that a CharacterActionItem can be converted to an ObjectList containing a SoundItem and an AnimationItem. ObjectList is just the generic Spyder list container class. Now, action1conf uses a subclass of the standard spyderhive, not the spyderdicthive. Standard spyderhives ignore the attribute name of a Spyder object in the class definition, and just look at the Spyder object itself. They don't have a standard mechanism to process the Spyder object, but simply calls a method make_bee() on it, and the resulting bee is put inside a hive. The action1conf derives from conf/action1hive, which derives from spyderframe. The conf/action1hive defines make_bee() methods for SoundItem and AnimationItem, which return exactly the same init bees as in

layer 10.
The mechanism is explained in detail in the top-down section above, under the paragraph "Configuration of the action1 system". In layer 8, the same reimplementation is done for action3conf.

Layer 7
From here, the Python hive code will be ported to the visual level. To prepare for this, layer 7 rearranges the files into the same directory layout as used by the visual defaultproject. The "layers.py" main script now looks exactly like "defaultproject.py", with the few changes documented. The "conf/" directory has been renamed to "spyderhives/". The action1/2/3 hives, since they are worker-like (with antennas and outputs), have been moved to the new workers/ directory, together with the drones and the actual workers. In addition, three directories that will later contain the visual representations have been created: hivemaps/, workermaps/ and spydermaps/.
These directories (as well as the main directory) contain a "hivegui.conf" file, containing the paths and the Python modules that the HiveGUI (and the SpyderGUI) must import at startup when editing a file in this directory. This is because the HiveGUI does not use listings of possible workers (nodes) or data models: they are being discovered at run-time from the imported Python modules, scanning for bee.worker and bee.hive classes and generating their visual forms from the class definitions.

Layer 1-6 are based on the very-high-level (visual) layers of the hive system

Layer 6 is the first that uses a visual representation: the spydermap. The Spyder objects in action3conf have been deleted, instead a single large Spyder.Spydermap is loaded from the file "spydermaps/action3spydermap.spydermap" and put into a special spyderhive called a spydermaphive (to be more exact: a spydermapframe; all frames share plugins and sockets with the parent, and we do want that). You can open the action3spydermap in a text editor and you will see that it really just contains a text representation of a Spyder object (this is called a .web file); if you wanted, you could specify it inline inside the spydermaphive class definition instead, as we did previously with the spyderhives.
A spydermaphive defines a make_bee() method for Spyder.Spydermap which returns a spyderhive; the base class of the spyderhive is defined as the Spydermap's "spyderhive" attribute. It then places inside the Spyderhive all Spyder objects under "objectdata", with the class attribute names given in "names". As you can see, these are exactly the Spyder objects that we have been defining since layer 8, so it all boils down to what it was before.
Except that a spydermap is editable in a GUI. When you type "spydergui.py action3spydermap.spydermap", you see the Spyder objects appear as spyderbee nodes of which you can edit all attributes, and you create more of them also. You can also rearrange the visual positions of the spyderbee nodes, but this won't have any effect at runtime, since the spydermaphive ignores them.

Layer 5
In layer 5, the same transformation to a visual representation is made for the action1 and action2 systems: the hives are transformed to *hivemaps*. A hivemap is another Spyder model with a special spyderhive, the hivemaphive, that gives it a make_bee() method. As you can guess, the hivemaphive inspects the Spyder.Hivemap object for the names and parameters of the bees (workers, drones and subhives), imports them, puts them inside a hive, and returns the hive. And it is of course editable in a GUI, the HiveGUI.

The only annoying thing is that the action1/2 hives had embedded spyderhives inside, and you can't embed spyderhives directly in a hivemap. Therefore, empty spydermaps, based on the action1/2 spyderhives (wrapping them, basically), have been created and added to the hivemaps.
Also, the syntax to connect a hive (layershive) to a hivemap is a little verbose (as you can see in the adapted layershive.py)

Layer 4 - 2
In layer 4, the data is moved out of the spyderhive into a spydermap for action1 and action2, just as layer 6 did for action3.

In layer 3, the main hive “layershive” is transformed to a hivemap. The transformation is pretty straightforward, but it does use one trick: if you need to pass a bee as parameter to another bee (such as passing the dragonfly.logic.dictionary or manager drones to the three action systems) in a hivemap, you must use a so-called wasp, which does exactly that. Wasps are better explained in the Canvas tutorial.

There are a few more of these tricks if you need to pepper your hivemap with some Python programming syntax. To define arbitrary non-GUI-editable attributes, you can use a pyattribute: the text in a pyattribute is literally interpreted inside the class as a Python expression. The same is done for all parameters to drones.

In layer 2, the last hive (action3) is converted from hive to hivemap.

Layer 1
The hive system contains one more visual representation: the workermap, which is edited with the WorkerGUI. However, there is no special spyderhive for the workermap. Instead, the transformation from workermap to worker is done by a Python code generator inside the WorkerGUI. This is because a Python representation (with numbered lines that you can look up) is much easier to debug from than a data representation. In fact, since layer 11, the Python code in workers/ has been generated by the WorkerGUI. The details of this are described in the top-down section above.