Do It Yourself:
Simple Simulation Design with the Blender Game Engine

This is a tutorial for constructing a simple harmonic oscillator in the Blender Game Engine. It covers the basics of setting-up the environment, writing a short python script to control the position of the oscillator, and a bit of logic-brick wiring to hook it all together.

1. **Download and install the latest version of Blender**
   Blender is available at www.blender.com in the downloads section. It is available for Linux, Windows and OS X in both 64 and 32 bit versions (be sure to download the version that corresponds with your system architecture). To install in windows, follow the directions in the setup executable, for OS X drop the file in the .dmg volume to your applications folder. Linux does not require an install, just extract the gzip file and click on the “blender” executable inside the extracted file.

2. **Set-up the Blender Environment**

   When you start Blender, you will be greeted with the default screen:

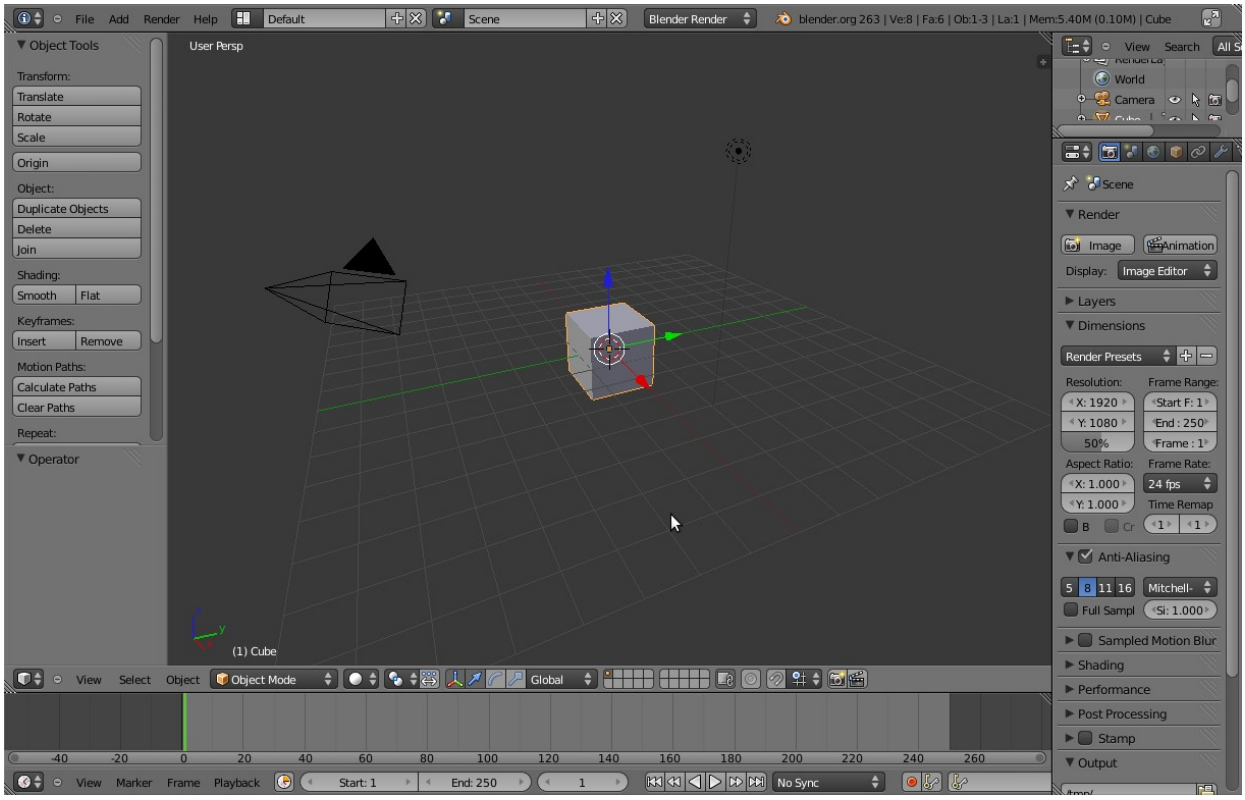

Figure 1: Screenshot of Blender's default layout. The central viewport shows the "default cube".

However, you will want to change the environment to “Game Logic”. To do this, you simply need to click on the button to the left of where it says “default” at the top of the Blender window. This will bring up a drop-down menu. Click on “game logic” to bring up the game logic layout. You should now see the following:

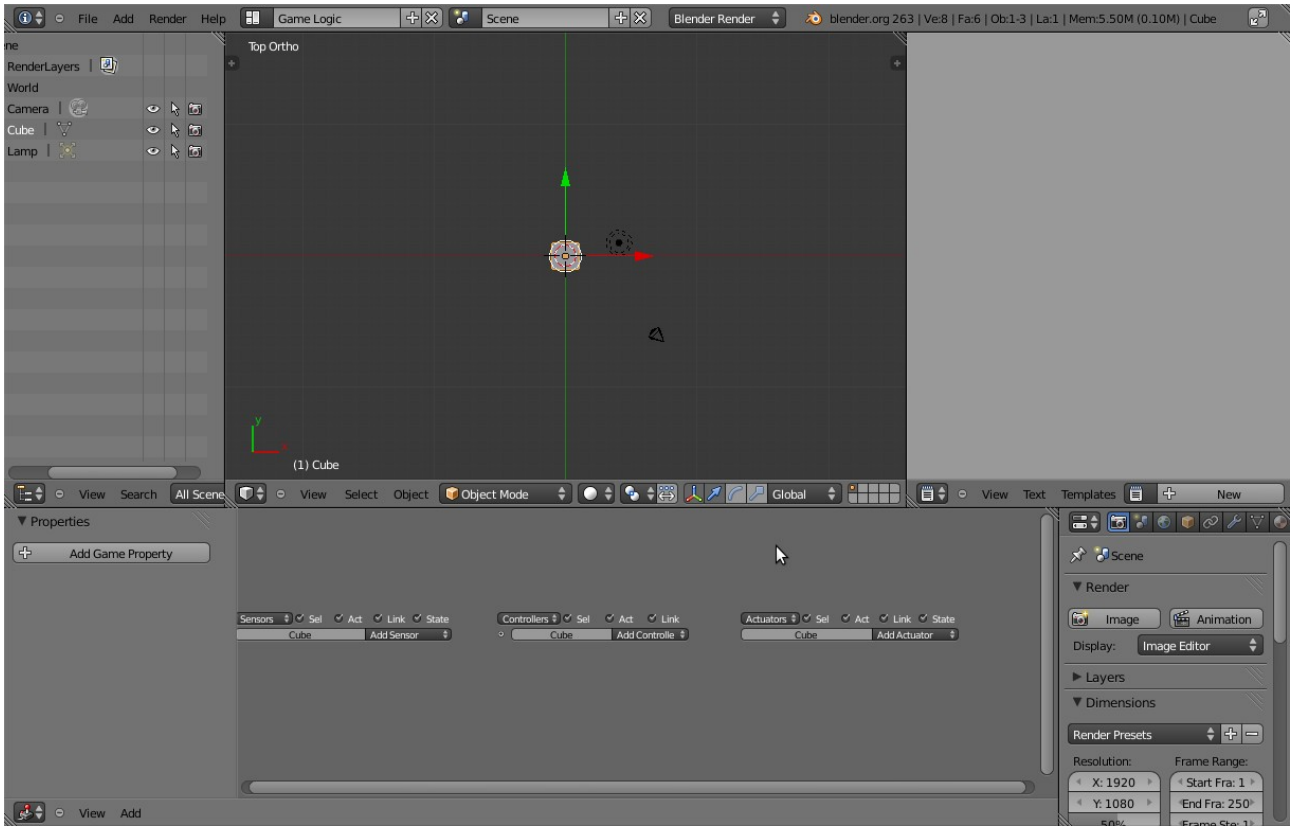

Figure 2: Screenshot of Blender's game logic window layout.

The game logic window layout consists of five default frames. The upper left frame is the “outliner” frame that lists all the objects in the current scene. The upper central frame is the 3D view port. The upper rightmost window is the text editor (this is where we will be doing our python programming). The bottom left window is the game logic window (where we will link together the object in the 3D view port and the python script). And, the bottom left window is the properties window. We need only concern ourselves with the game logic window, the 3D view port and the text edit window.

3. **Writing the Python Script**

   This is the most technical part of the entire tutorial. The goal of this tutorial is to get the default cube in the view port moving according to the analytic solution for a simple harmonic oscillator (e.g. a spring hooked to a weight). To do that, of course, we need to tell Blender what we want it to do. For this, we use the Python programming language.

   In the toolbar below the text editor window, press the + button in order to create a new blank Python program. Next to the + button, click in the text entry field and give your program a name ending in .py (for example myprogram.py).

   Now, to get programming. The first thing that we want to do is to expand Python's collection of functions that it can use. We want it to know about the Blender Game Engine functions. Also, we want to expand Python's capacity for mathematical functions (we are doing a physics simulation, after all). So, the first two lines of the program will be:

   ```
   import bge
   import math
   ```

Next, you must use the game logic class in the BGE library to set-up a controller for the default cube. If you do not know what that means, do not worry, all you have to know is that you need the next two lines of code are controllers that let you change properties of the default cube using Python (including the position).

```
cont = bge.logic.getCurrentController()
own = cont.owner
```

Before the simulation can be run, you need to give it a set of initial conditions. All that really needs to be set is the time at which the simulation starts and the path to which an output file should be written.

```
if 'init' not in own:
        own['t'] = 0.0
        own['output']= open('output.txt','a')
        own['init'] = False
```

The first line above is a simple initialization conditional. As long as the default cube does not have a property called 'init', the statement will be true and the code below it will be executed. However, the last line creates the 'init' property and thus the code within the conditional will be skipped for every future run-through of the code.

The next line sets the global variable 't' to 0.0. The third line creates an output file to write data to (or, if the file already exists, will simply append the data to the end of the file). The “own[]” around the variables defines the variables as “global variables” (but we will not go into this distinction here).

It must be noted here that Python has a unique perk. Unlike many other programming languages, Python uses white spice in order to denote that lines of code belong to a statement (as opposed to curly brackets in C, for instance). The editor in Blender will add these spaces automatically after writing the statement and pressing return. However, it is useful to note that this whitespace is equal to a single tab space.

Now that the controller is set-up and the initial conditions defined, the actual equation of motion must be written. The analytic equation for a simple harmonic oscillator is a simple oscillating solution (sin or cos). The line of code for this is

```
own.position.x = math.sin(own['t'])
```

Simple. The own.position.x statement refers to the x-position property of the default cube and, of course, math.sin(own['t']) is our equation of motion.

Next, you will want to actually write the position and time out to a file so that it can be plotted and analyzed later.

```
lineout = "%02f\t%02f\n" % (own['t'],own.position.x)
own['output'].write(str(lineout))
```

The top line may look like gibberish, but it is actually a formatting statement. %02f prepares for a double float, \t denotes a tab, and \n denotes a new line. The second line of code above writes the formatted data to the output file as a string.

Last, all that needs to be done is to update the time.

own['t'] = own['t']+.1

And that is it for the programming portion! Your Blender window should now look like this

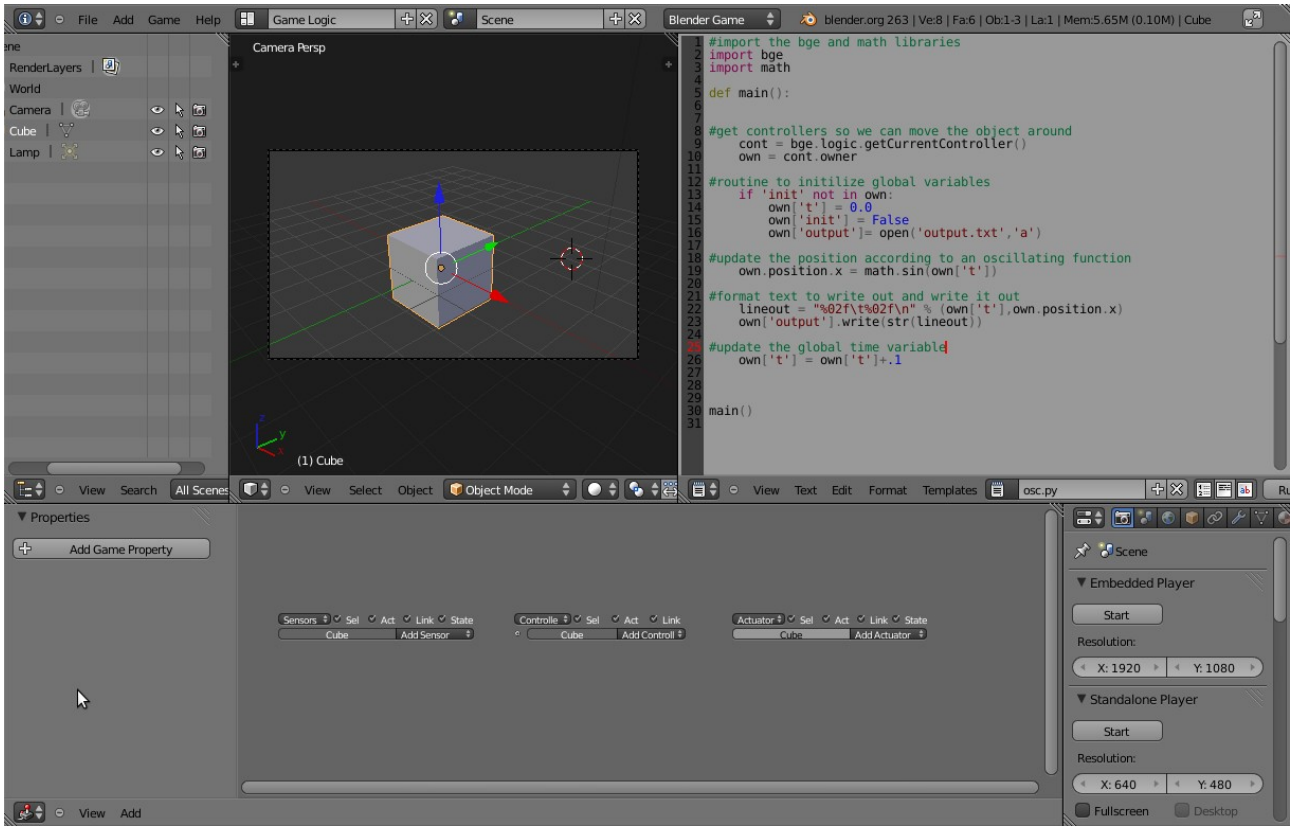

Figure 3: Blender with Python code

Note that the # symbol denotes a “comment”. Lines that start with # will not be evaluated by the interpreter. These are generally used as notes to the programmer.

4. **Add an “Always” Sensor**
   Now, to link the Python code to the default cube, you need to refer to the game logic window at the bottom of the screen. You must add a “sensor” called an “Always sensor”. Click on the drop-down menu that says “Add Sensor” in the game logic window. Select “Always”, then click the set of three dots (pulse) as shown below

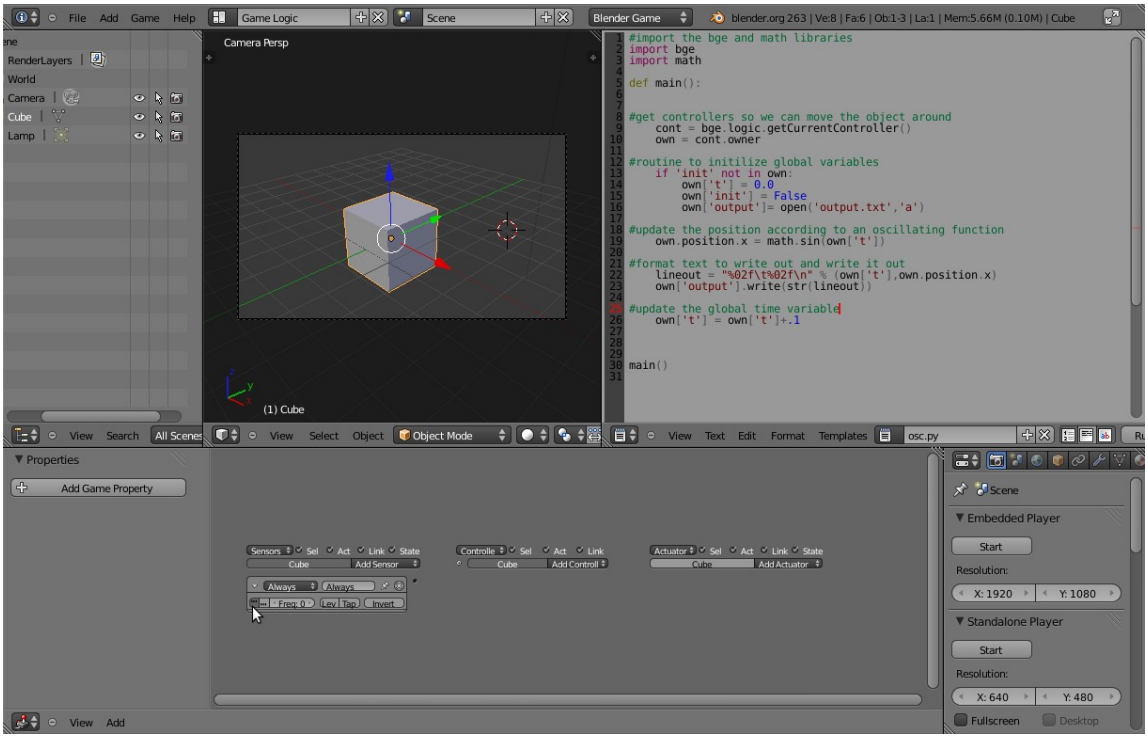

Figure 4: Adding an always sensor

By clicking the “pulse” button, you are telling Blender to run whatever is attached to the “Always Sensor” over and over again (as opposed to once).

5. **Connecting it all up!**
   Now you must connect the “Always Sensor” to the Python program you wrote. To do this, you need to add a “Controller”. Simply click “Add Controller” in the center of the game logic window. Select “Python” from the drop-down menu. Then, click in the text entry box in the new “brick” that pops up and select your program. Finally, grab the knob on the right of the “Always Sensor” and drag it to the knob on the “Python Controller” you just created. That is it! Your Blender window should now look like this:

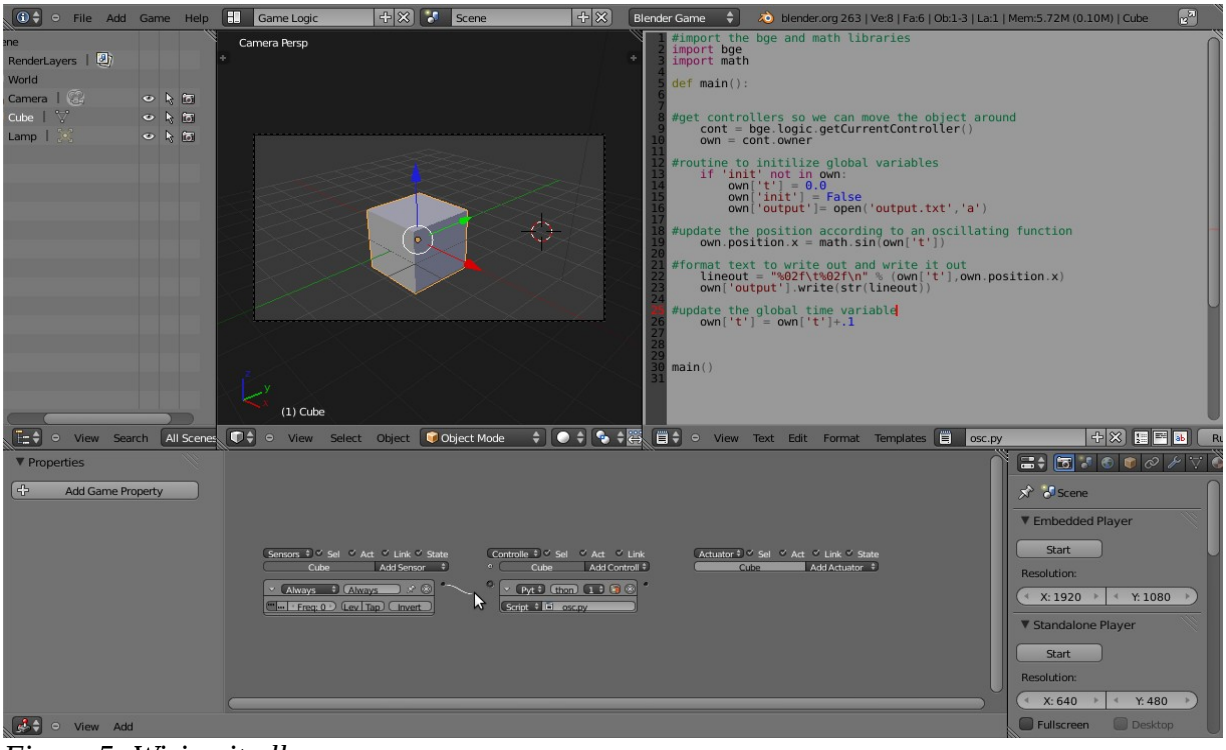

Figure 5: Wiring it all up

6. **Run it!**
   Now, you just need to run your simulation. Click inside the 3D port viewer and press the P key.

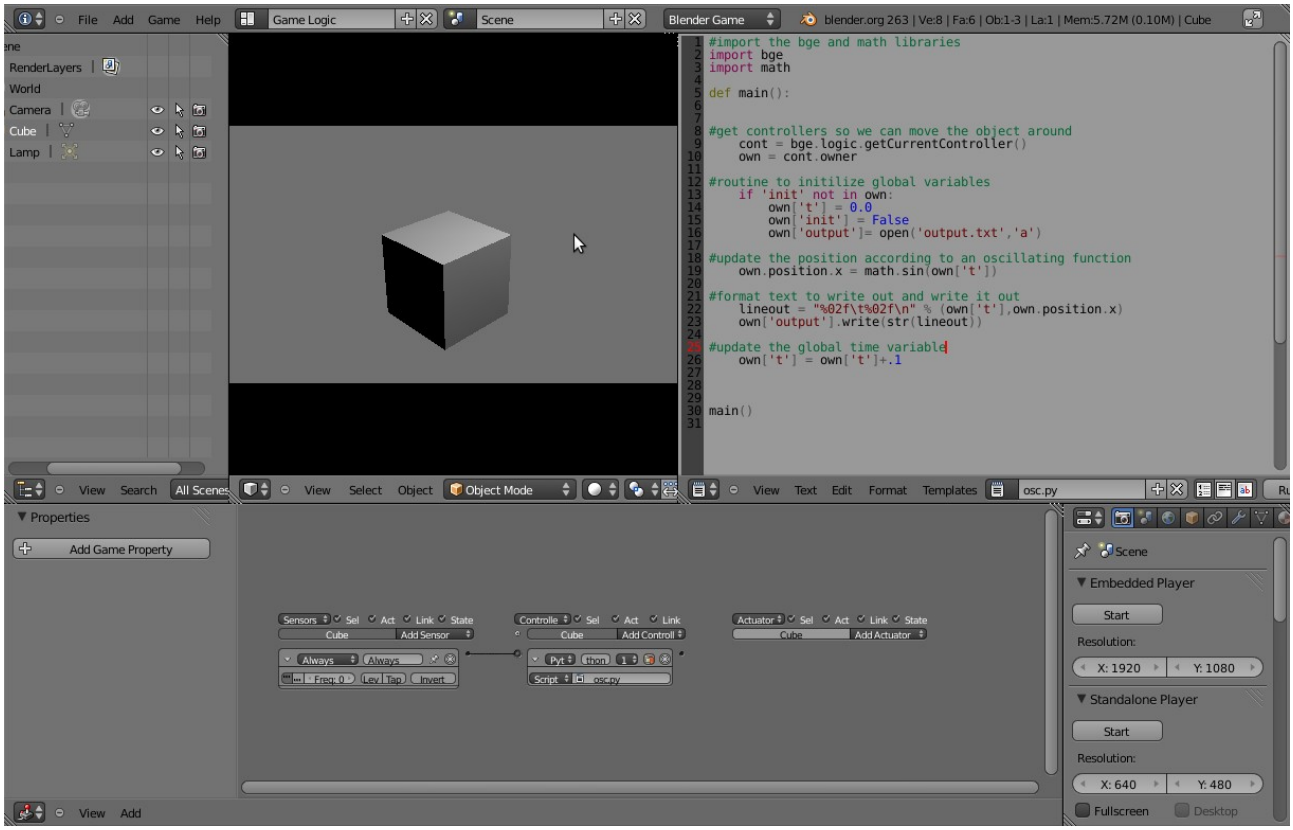

Figure 6: The simulation running

Now that you have it running, try playing with the Python code. What happens if you use a cos rather than a sin? What if you multiply the right side of the equation of motion by some number? Have fun.

If you have any comments or questions, please contact Trevor Tomesh at t.tomesh@worc.ac.uk

```
#import the bge and math libraries
import bge
import math



#get controllers so we can move the object around
cont = bge.logic.getCurrentController()
own = cont.owner

#routine to initilize global variables
if 'init' not in own:
        own['t'] = 0.0
        own['output']= open('output.txt','a')
        own['init'] = False


#update the position according to an oscillating function
own.position.x = math.sin(own['t'])

#format text to write out and write it out
lineout = "%02f\t%02f\n" % (own['t'],own.position.x)
own['output'].write(str(lineout))

#update the global time variable
own['t'] = own['t']+.1
```