PiavcaPython: User guide.

This is a brief introduction aimed at helping people get going with Programming PIAVCA using the Python API. The Python API has essentially the same functionality as the C++ API so this guide is really just a rewrite of the C++ guide. The full API documentation describes in full detail all the classes and methods in PIAVCA. It should be possible to use all of these from Python. Translating a C++ function to a python one should be fairly straightforward, they should be called pretty much identically with a few type covertions: tstring and std::string become a python string, StringVector becomes a list of strings. All C++ classes in PIAVCA become python classes. The API has been generated automatically by the SWIG interface generator, and though I've used it myself extensively, there is no guarantee that I've tested it all and therefore that every function works. Mail me if there is a problem.

Getting and Installing PIAVCA

How to get PIAVCA running heavily depends on which platform you use and will be dealt with in platform specific guides.

Getting the Python API working in your application

Getting the python API to work from a C++ application is fairly straightforward. You need to include the `PiavcaPythonInterface` and include "`PiavcaPythonApp.h`"

Then you need to do when using PIAVCA is to set up the `Piavca::Core` object. This object manages the global aspects of PIAVCA and is needed to create Avatars and Motions. The `Piavca::Core` object has to be created in a platform specific way (this is the only code that has to be) and will be explained in the platform guide an example is:

```
Piavca::Core *core = new Piavca::TaraPiavcaCore(g_pGraph);
```

The final piece of set up is to initialise the python interface by calling `InitPiavcaPython`. At the end of the program you need to call `EndPiavcaPython` to finalise the interface. Here is a short example:

```
#include "PIAVCA/PiavcaAPI/PiavcaCore.h"
#include "PIAVCA/PiavcaTaraImp/PiavcaTaracore.h"
#include "PIAVCA/PiavcaPythonInterface/PiavcaOPythonApp.h"
void main(char **argv, int argc)
{
   // set up underlying scene graph here
   // next line sets up PIAVCA
   Piavca::Core *core = new Piavca::TaraPiavcaCore(g_pGraph);
   // This next line sets up the python interface
   Piavca::InitPiavcaPython(g_pPiavcaCore);
   // main loop here
   while(something)
   {
       // do stuff
   };
   Piavca::EndPiavcaPython(g_pPiavcaCore);
}
```

Now you are ready to start writing Python code. `InitPiavcaPython` will search the `PYTHONPATH` for a file called `init_python.py` and run it, so all you have to do is put your code there. To access PIAVCA functions include `Piavca.py`. From this file it will try to run a function called `PiavcaStartUp` with the core object as an argument. When `EndPiavcaPython` is called it will call a function called `PiavcaEnd`. A function `PrintOutput` can also be provided, this is used to deal with output sent to `cout` from the C++ code. Therefore the `init_python.py` file needs to look something like:

```
import Piavca
def PiavcaStartUp(corePtr):
  # do some initialisation code here
  # probably create a callbacks (see later)
def PiavcaEnd():
  # do finalisation code here
def PrintOutput(str):
  # deal with stuff that Piavca sends to cout
  print str
```

Avatars

Once the core is created we can start to create and use Avatars. To create an avatar you have to pass in the ID of the avatar. This is a text string which is interpreted in different ways on different platforms, normally as the name of the root node of the avatar in the scene graph or as a file name containing the avatar geometry. Other optional parameters include a flag `bailOnMissedJoints` which causes the creation to fail if not all joints are present in the avatar (see the Joints section later), if so an invalid avatar is return and can be tested using the `isValid()` (any attempt to use an invalid avatar will fail and probably throw an exception, I can't promise it won't just crash though). You can also pass in a start position and orientation for the avatar. Here is an example (`Vec` and `Quat` are vector and quaternion classes for positions and rotations see the reference manual form more details):

```
import Piavca
jill = Piavca.Avatar("Jill")
jack = Piavca.Avatar("Jack", 1,
            Piavca.Vec(10.0, 0.0, 0.0),
            Piavca.Quat(M_PI/2.0, Piavca.Vec.YAxis()))
```

Once an avatar has been created a pointer is stored in the PIAVCACore object and can be retrieved by name:

```
jill = Piavca.Core.getCore().getAvatar("Jill")
```

OK so now we've got avatars we want to do something with them. There are three main methods of controlling an avatar, directly manipulating the joints, applying a motion and using callbacks. The three are described in the next sections.

Joints

An avatar in PIAVCA consists of a hierarchical collection of body parts called Joints. These Joints can rotate relative to each other. There is also a special Joint root which specifies the position and orientation of the whole avatar. Each Joint is identified by an integer ID, these identifiers are the same across all avatars so if joint 7 is the left elbow of Jill it is also the left elbow of Bill (this feature is very useful when we come to Motions). There is a mapping between strings representing Joint names and IDs that is handled by the core object:

```
l_elbow = core.getJointId("Left_Elbow")
```

There are two special IDs `root_position_id` and `root_orientation_id` for the position and orientation of the root.

Joints cannot be directly accessed (their representation is platform dependent). Instead all operations on joints are performed by passing the ID to a method of the avatar class. You can set and get the orientation of the joint and the position and orientation of the root of the avatar, and you can get the position of a joint. Here are some examples:

```
jill.setRootPosition(Piavca.Vec(0.0, 0.0, 5.3))
jill.setJointOrientation(l_elbow, Piavca.Quat(M_PI/4, Piavca.Vec.XAxis())
headOri = jill.getJointOrientation(core.getJointId("Head"))
headPos  = jill->getJointBasePosition(core.getJointId("Head"))
```

Not all avatars necessarily contain all possible joints so a given joint ID might not exist for an avatar, a method `isNull()` is provided to test if a joint exists. To help iterate over joints three methods are provided, `begin()` returns the first valid ID, `next()`, when passed an ID returns the next valid one and `end()` gives the last one + 1. This means that the standard way to iterate over joints is this:

```
i = jill.begin()
while (i != jill.end()):
   # do something
   i++
```

Motions

Motions are probably the key feature of PIAVCA, they represent an animation sequence independently of an avatar. A motion has a number of tracks each of which correspond to a joint of an avatar, the motion can be queried for the value of a track at a given time. In order to animate an avatar a motion can be applied to that avatar by first loading it into the avatar and then calling `playMotion` to start it playing:

```
jill.loadMotion(mot);
jill.playMotion();
```

The joint IDs of a track is identical to that of the corresponding joint so when a motion is applied to an avatar tracks are matched to joints by ID. After the values of the motion are queried at each frame and the joints are put into the resulting posture, thus animating the avatar.
Tracks have three types depending on what type of data they hold, float, vector (Vec) or quaternion (Quat). Currently Quats are used for joint orientations, Vecs for the root position, float tracks are there for possible future expansion (for example for facial animation).
The Motion class itself is an abstract data type, specifying the interface for motions. A motion can be any type of object that can be represented as a sequence of value over time and therefore support a number of different styles of animation: keyframe animation, procedural animation and real time tracking. Adding any of these sort of animation style involves creating a subclass of the Motion object. To use Motions there are three possible methods: using the built-in keyframe animation object; using a built in Motion manipulator object to transform an existing motion, or creating a new sub type of Motion for a new method of generating motion. The next few sections describe these methods:

Keyframe Animation

PIAVCA has a built in class for keyframe animation, `TrackMotion`. Keyframe data can be loaded from a BVH file and then used:

```
mot = Piavca.TrackMotion("walk.bvh")
```

```
jill.loadMotion(mot)
jill.playMotion()
```

Motion can also be loaded into the core object so that it can be used by multiple avatars later (motions in the Core are identified by a name). As BVH files can be based on different coordiate systems or on different avatar rest postures from PIAVCA, flags to can be passed to the motion creation process to do various corrections (see the reference manual for details). For example:

```
Piavca.Core.getCore().loadMotion("walk", "walk_cycle.bvh"), TRANS_NEG_X)
mot = Piavca.Core.getCore()->getMotion("walk")
jill.loadMotion(mot)
jill.playMotion()
```

Motion manipulators

Motion manipulators are a subclass of Motion that generate new motions by transforming existing motions. PIAVCA contains a number of built in manipulators and others can be created (see next section). There are two types of built in manipulators, MotionFilters generate new motion based on a single motion, they include: `LoopMotion` which runs a motion in a loop; `TurnMotion` which makes the motion turn through a given angles, and `ScaleMotion` which scales the root position of the motion (useful for correcting foot slip). TwoMotionCombiners, combine two pre-existing motions, they include: `BlendBetween` which interpolates between two motions, `MotionAdder`, which adds two motions, performing them simultaneously, and `SequentialBlend`, which performs one motion after an other, smoothly blending into eachother. See the reference manual for more details.

Motion Manipulators are used by passing the motions to be manipulated into the constructor of the manipulator and then loading the manipulator object into an avatar. Manipulators can be passed into other manipulators to create complex effects:

```
walk = Piavca.Core.getCore()->getMotion("walk")
run = Piavca.Core.getCore().getMotion("run")
blend = Piavca.BlendBetween(walk, run, 0.5) #param 3: blend factor
loop = Piavca.LoopMotion(blend)
jill.loadMotion(loop)
jill.playMotion()
```

Roll your own Motion

Currently it is not possible to create new types of motion from Python. This would require creating a subclass of the motion class but python subclasses of C++ classes cannot be used by C++ so a motion created in Python cannot be used. A work around might be provided in the future.

Callbacks

The final method of animating avatars is to use a callback object. PIAVCA defines two types of callback object `TimeCallback` and `AvatarTimeCallback`. The python interface provides two subtypes of these: `PyTimeCallback` and `PyAvatarCallback`. Both are activated every frame, the first is a global callback which is loaded into the core and the second is loaded into an individual avatar. A new python callback is created by defining a class that contains two methods `callbackInit` and callbackMethod. The first provides initialisation code and the second is called every frame. For `PyTimeCallback` both methods take a `Piavca.Core` as an argument and `callbackMethod` takes the current time, for `PyAvatarCallback` both take an avatar instead of the core. Here is an example of a trivial callback class:

```
class PrintTime:
  def __init__(self):
      pass:
  def callbackInit(self, core):
      self.numAvatars = core.numAvatars();
      print "I'm just going to print out the time every frame"
  def callbackMethod(self, core, time):
      print "time is ", time
      if numAvatars < core.numAvatars():
            print "ooh look they've added a new avatar"
            numAvatars = core.numAvatars()
```

To register a callback you must first create a `PyTimeCallback` object by passing in this new class and a name as arguments. As the callback object is going to be passed back to C++ you need to tell Python that the memory management for it will be done by C++ not Python. This is done by setting the `thisown` flag to zero.

```
callback = PrintTime()
callbackObj = Piavca.PyTimeCallback(callback, "Print Time")
callbackObj.thisown = 0
```

Once you have a callback object you register it either with the core or with an avatar:

```
core.registerCallback(callbackObj)
avatar.registerCallback(callbackObj)
```

That's all folks

This is the end of the brief overview and tutorial of PIAVCAPython, for more information about individual features you can look at the reference manual.

Marco Gillies (m.gillies@ucl.ac.uk)

7/7/04