Piavca: User guide.

This is a brief introduction aimed at helping people get going with Programming PIAVCA. This guide covers platform independent aspects of the PIAVCA C++ API. There will be separate guides for each platform and for the Python/TCL/ActiveX APIs. For more detail on any of these subjects see the reference manual.d

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.

Note of tstring

As PIAVCA is platform independent it does not specify whether ACSII or UNICODE characters should be used. To get around this problem we use the `_T()` and `_TCHAR` macros (which are standard but PIAVCA defines them if it is run on a system where they are undefined). `_TCHAR` is either defined to be a `char` or `wchar_t` depending on whether `_UNICODE` is defined. `_T()` is used to convert a string literal to the appropriate type, if you wrap the literal in `_T("text")` it will be converted to an appropriate type. To extend this to STL strings we define `tstring` which is an STL string for `_TCHARS`. All string handling in PIAVCA is done through `tstrings`.

Intialisation

The first thing 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);
```

There is only ever one core object (it is a singleton if you're a Design Patterns fan), and this can be obtained with a static member of the `Core` class:

```
Piavca::Core *core = Piavca::Core::getCore();
```

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):

```
Piavca::Avatar jill = new Piavca::Avatar(_T("Jill"));
Piavca::Avatar jack = new Piavca::Avatar(_T("Jack"), true,
            Vec(10.0, 0.0, 0.0), Quat(M_PI/2.0, Vec::YAxis()));
```

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

```
Avatar jill = Core::getCore()->getAvatar(_T("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:

```
int l_elbow = core->getJointId(_T("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. The position and orientation of joints can be handled in a number of coordinate systems, in general it is most useful to always deal with the orientation of the joint in its own local coordinate system (`JOINTLOCAL_COORD`) and with position in either the coordinate system of the avatar or the global coordinate system (`LOCAL_COORD, WORLD_COORD`). Here are some examples:

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

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:

```
for(int i = jill.begin(); i < jill.end; i = jill.next(i))
{
   // do something
}
```

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:

```
TrackMotion *mot = new TrackMotion(_T("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:

```
getCore()->loadMotion(_T("walk"), _T("walk_cycle.bvh"), TRANS_NEG_X);
Motion *mot = getCore()->getMotion(_T("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:

```
Motion *walk = getCore()->getMotion(_T("walk"));
Motion *run = getCore()->getMotion(_T("run"));
Motion *blend = new BlendBetween(walk, run, 0.5);//param 3: blend factor
Motion *loop = new LoopMotion(blend);
jill.loadMotion(loop);
jill.playMotion();
```

Roll your own Motion

To achieve effects that are not possible with keyframe animation and the built-in manipulators users can create their own subclasses of Motion. This could be to implement a manipulator with different features to the built-in ones or to implement a procedural animation scheme. In order to create a new motion type the following virtual functions need to be defined:

- `getMotionLength()` returns the length of the motion
- `isNull(int trackId)` returns whether a given track is valid for the motions
- `getTrackType(int trackId)` returns an enum defining the type of the track (`FLOAT_TYPE`, `VEC_TYPE`, or `QUAT_TYPE`)
- `getFloatValueAtTime(int trackId, float time)`, `getVecValueAtTime(int trackId, float time)`, `getQuatValueAtTime(int trackId, float time)` get the value of float, Vec and Quat tracks, respectively at a given time (these will implement the core functionality of the motion).

If you are implementing a motion manipulator you can sub-class from `MotionFilter` or `TwoMotionCombiner` which supply default implementations of the first three methods (check the reference manuals to make sure these implementations do what you want. The following is a quick example of a new motion manipulator which raises the root position of the motion by a given amount.

```
class MotionRaiser : public MotionFilter
{
  float height;
public:
  MotionRaiser(Motion *mot, float height):MotionFilter(mot), height(height){};
  getFloatValueAtTime(int trackId, float time)
  {
      return mot->getFloatValueAtTime(trackId, time);
  };
  getVecValueAtTime(int trackId, float time)
  {
      if(trackId == root_position_id)
            return mot->getVecValueAtTime(trackId, time) + height*Vec::YAxis();
      else
            return mot->getVecValueAtTime(trackId, time);
  };
  getQuatValueAtTime(int trackId, float time)
  {
      return mot->getQuatValueAtTime(trackId, time);
  };
};
```

The following example gives an outline of how a procedural animation Motion might be implemented. This example adds noise to the rest position of an avatar to generate a feeling of movement (alla Ken Perlin). A function `PerlinNoise` is assumed to exists that generates coherent quaternion noise from an floating point number. Here we use time as the input to `PerlinNoise` but add an offset to make the different noise functions (I assume a function `RandomOffset` to generate these). The constructors takes a vector of joint IDs which are the ones that are to have the random

noise applied to them.

```
class Perlinizer : public Motions
{
  //! true for joints that should have noise applied
  std::vector<bool> jointsToApply;
  //! the offset for a given jointLOCAL
  std::vector<float> offsets;
public:
  //! initialise the two arrays to the number of joints and then
  // fill in the values for the joints that the noise should be applied to
  Perlinizer(std::vector<int> joints)
      : jointsToApply(getCore()->getMaxJointId()+1, false),
        offsets(getCore()->getMaxJointId()+1, 0.0)
  {
      for(int i = 0; i < joints.size(); i++)
      {
            jointsToApply[i] = true;
            offsets[i] = RandomOffset();
      }
  };
  float getMotionLength(){return -1;} //! negative == no end
  bool isNull(int trackId){return true;} //! valid for all motions
  track_type getTrackType(int trackId)
  {
      if(trackId == root_position_id) return VEC_TYPE;
      else return QUAT_TYPE;
  };
  //! float a Vec tracks have no noise applied (you don't want
  // to have the root position vary randomly) so just return zero values
  float getFloatValueAtTime(int trackId, float time){return 0.0;};
  Vec getVecValueAtTime(int trackId, float time){return Vec();};
  // this does the business
  Quat getQuatValueAtTime(int trackId, float time)
  {
      if(jointsToApply[trackId])
            return PerlinNoise(time+offsets[trackId]);
      else
            return Quat(); //returns zero rotation
  };
};
std::vector<int> v;
v.push_back(getCore()->getJointId(_T("neck")));
v.push_back(getCore()->getJointId(_T("r_elbow")));
```

```
v.push_back(getCore()->getJointId(_T("l_elbow")));
v.push_back(getCore()->getJointId(_T("pelvis")));
Motion *mot = Perlinizer(v);
jill.loadMotion(mot);
jill.playMotion();
```

Callbacks

The final method of animating avatars is to use a callback object. PIAVCA defines two types of callback object `TimeCallback` and `AvatarTimeCallback`. 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 callback is created by subclassing either of these type, they both have two methods which should be overridden, `init` which is called when the callback is registered and `timeStep` which is called every frame after the callback has been registered. Both of these methods are passed either a pointer to the core in the case of TimeCallbacks or a pointer to the avatar for an AvatarTimeCallback. The follow gives an example of a trivial callback:

```
class PrintTime : TimeCallback
{
  int numAvatars;
public:
  PrintTime(tstring name):TimeCallback(name){};

  void init(Core *core)
  {
      numAvatars = core->numAvatars();
      std::cout << "I'm just going to print out the time every frame\n";
  };

  void timeStep(Core *core, float time)
  {
      std::cout << "time is " << time << std::endl;
      if(numAvatars < core->numAvatars())
      {
            std::cout << "ooh look, they've just added a new avatar\n";
            numAvatars = core->numAvatars();
      }
  }
}
```

You create and register a callback like this:

```
TimeCallback *cb = PrintTime("timer");
getCore()->registerTimeCallback(cb);
AvatarTimeCallback *acb = SomeAvatarCallback("someName");
jill.registerCallback(acb);
```

That's all folks

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