A Software Framework for Controlling Virtual Reality Avatars via a Brain-Computer Interface

Denis Porić, Alessandro Mulloni, Robert Leeb, Dieter Schmalstieg

Abstract:

This paper discusses the Avatar Control Framework which allows control of an avatar in Virtual Reality by a Brain-Computer Interface. The current Brain-Computer Interfaces are still experimental so a framework was needed to help with their testing. This framework allows for the creation of special scenarios which can give the BCI developers feedback. The Avatar Control Framework consists of a visualization and communications part. The visualization part uses motion classes to animate an avatar. The communications part uses a UDP connection to receive commands and relay them to the visualization part. The Avatar Control Framework is able to construct complex motions out of simple building blocks, play these on one or multiple avatars and is highly customizable.

1. Introduction

Since the emergence of computers in our everyday life, scientists and engineers have been trying to simplify the interaction between the computers and the users. This has been accomplished by introducing various interfaces which allow a simpler and faster interaction with the computer. Beside a plethora of interesting approaches, one of the more interesting is the Brain-Computer Interface (BCI). It theoretically allows a user to control a computer with his or her thoughts by monitoring the user's electroencephalogram (EEG) and executing some predetermined actions for a particular signal pattern. Because a BCI functions in that particular way, there are many possible areas of application. One of the more important areas would be helping handicapped or paralyzed people to lead more independent and self-sufficient lives. Another area of application would be in the entertainment industry. A BCI would allow unprecedented possibilities for interactive software and movies. Besides these there are many other areas of application which require a hands-free approach (e.g. surgeries, hazardous materials work etc.)

Most of the currently existing BCI are still in development and are still experimental, preventing their wider use. The same is true for the BCI that is being developed at the time of the writing of this document by the Laboratory of Brain-Computer Interfaces of the Graz University of Technology. The development and improvement of a BCI requires various feedback scenarios.

The issue with the feedback scenarios that the Graz BCI group currently face is that it is tedious to prepare a specific scenario and that the created scenario isn't very flexible. The

visualization of the BCI commands is important because it may allow the BCI group to speed up the learning process of the BCI. In order to do that the BCI group is in need for an avatar control framework (ACF) that can understand and visualize within virtual reality (VR) the specific commands the BCI sends which is flexible and is at the same time easy to use.

In this paper we will explain how this ACF is built up, show how it works and show the customization options that are available to the end user. We will also present an overview of the underlying dependencies and libraries which enable the framework to function. In the end we will give a short overview of the communications protocol the framework uses as well as its customization options.

2. The Avatar Control Framework

As shown in Figure 1 the ACF can be roughly divided into 2 parts. The communication part allows the framework to receive and process commands from the BCI as well as to send feedback to the BCI or the user. The other part is the visualization part which visualizes the commands which were received and processed by the communications part.

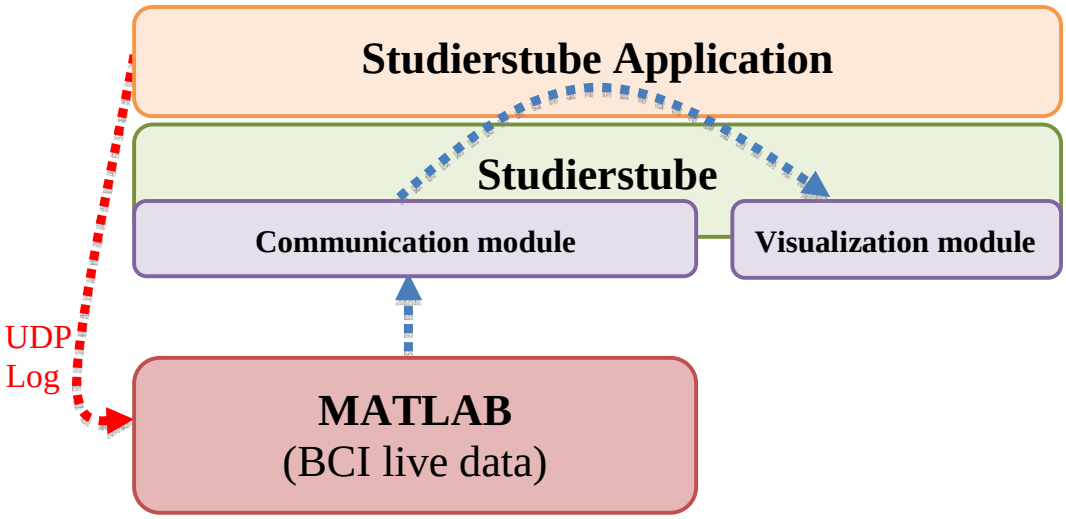

Figure 1: Graphical representation of the ACF

There are 3 libraries/frameworks on which the ACF depends. These are:

- Studierstube
- PIAVCA
- Adaptive Communication Environment (ACE)

Further reference can be found at the respective libraries API´s.

Studierstube is " a component based framework and provides a viewer for displaying 3D objects for augmented reality applications based on OpenInventor and Coin" [1] . It is being developed and maintained by the Graz University of Technology and the Vienna

University of Technology. Studierstube 4.2 was used during the development of the ACF. Additional information about Studierstube, the full documentation as well as the newest versions can be found at [3].

PIAVCA (Platform independent API for Virtual Characters and Avatars) is a platform independent animation engine which is based upon Cal3D [4]. It allows the blending, addition and subtraction of different animations thereby enabling the user to create large amounts of animations from a small list of original animations. The available documentations as well as a short introduction can be found at [2].

The Adaptive Communication Environment (ACE) "is a freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software" [5]. It is used within the ACF communications part to allow concurrent communication and execution of different threads and also to ensure that the ACF is platform independent. Further information can be found at [6].

2.1 Visualization

The visualization of the BCI commands uses Studierstube in conjunction with PIAVCA. Studierstube is needed to display the objects which are being manipulated with the BCI while the PIAVCA component is needed to manipulate the animation of the same objects.

2.1.1 The Avatar

The objects that are being manipulated with the BCI commands within the AFC are called avatars and are basically VR representations of objects or beings.
An avatar contains a skeleton that is defined by a list of joints. The joint list can vary in complexity and detail depending on the abstraction of the avatar compared to a real being or object. This skeleton can be used to move the avatar by using skeletal animations. The clear advantage of such an approach is that the skeletal animations can be reused for all avatars that use the same skeleton. A disadvantage however would be that it might be needed to recalculate all vertex positions on the fly.
The previously mentioned joints within the joint list have an orientation and position. The orientation saves the angle and rotation axis of the joint while the position determines the overall position of the joint within the avatar.
By manipulation of the joint orientations it is possible to move the limbs of the VR avatar and thereby enable it to execute specific movements.

2.1.2 Motions

A motion within the ACF consists of one or more successive manipulations of one or multiple joints. The motion moves a limb attached to the targeted joint to a user-specified angle on a user-selected axis by actually rotating the joint itself. The user inputs only one of the 4 possible movement directions: left, right, up and down. These directions are then used to determine the axis along which the chosen joint will rotate.

To be able to initiate a motion PIAVCA needs to know which avatar needs to move and the joint, angle and axis. When these values have been provided PIAVCA then builds a so called track. A track is a list of keyframes of the aforementioned variables that define in space and time the progress of the motion from its starting point to the endpoint. The more keyframes we have in that list the smoother the motion will be.

The naming of the motions depends on their purpose. If the motion is related to manipulating joint orientations then it is named after the joint it manipulates, the direction of the movement and the word "Motion" (e.g. HipLeftRightMotion). If it executes some specific action or manipulates the whole avatar then it is named after the action with the word "Motion" at the end (e.g. RotateAvatarMotion)

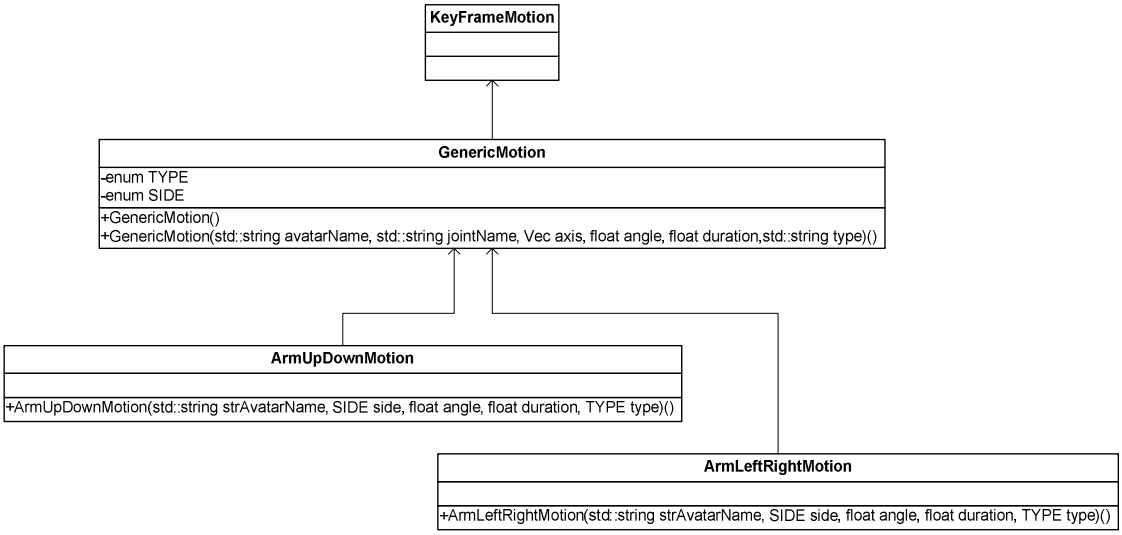

Figure 2: GenericMotion class diagram with 2 example motions

2.1.2.1 Joint related motions

The requirement to know the avatar name and the joint name that needs to be moved would lead a naive implementation to an enormous amount of replicated code. To avert this, we inherit all specialized classes in the ACF from a parent class called GenericMotion as shown in Figure 2. This class contains the true functionality and is able to use direct PIAVCA methods since it is inherited from a PIAVCA motion class itself. Because all other ACF motion classes inherit the GenericMotion class, they are able to call the constructor of GenericMotion as in Figure 3, allowing it to initiate the motion without having redundant code in the framework.

```
LimbDirectionMotion::LimbDirectionMotion(std::string strAvatarName, SIDE side, float angle, float duration,TYPE type)
        : GenericMotion(strAvatarName,
                        (side==LEFT  ? LeftJoint" : "RightJoint"),
                        (side==LEFT ? Vec(0,1,0) : Vec(0,-1,0)),
                        angle, duration,
                        (type==REL ? "relative" : "absolute"))
{
}
```

Figure 3: A generalized example of how the motions call GenericMotion

Another issue with the joint name was that the user would have to know the correct form of it which PIAVCA reads from the avatar input file. This would have complicated the use of the ACF because it can be tedious to find the joint name. In order to avert this the ACF contains 10 motion classes which only need a generalized joint name and which does not have to be the same as the PIAVCA name.

The following generalized joint names are currently used:

- Arm
- Elbow
- Wrist
- Hip
- Knee
- Ankle

These names are converted to the internal PIAVCA joint names by the 10 previously mentioned classes. These classes then call the GenericMotion constructor which executes the motion. They also decide if the left or right joint will be moved and what motion type will be used according to the parameters received from the communications part of the framework.

2.1.2.2 Motion types

There are 2 motion types within the ACF: the relative and absolute type. The relative motion moves a joint by incrementing or decrementing its current rotation angle while the absolute motion type moves the joint to an absolute angle with no considerations of the current rotation angle.

PIAVCA resets the joints orientation after conducting a motion to its default position and at the joint initialization its direction and orientation is random. Therefore the ACF contains a special register which stores the joints orientation data and is updated each time a joint moves. That register allows the ACF to perform absolute and relative motions.

Currently all of the joint-related motions support both types. The miscellaneous motions however do not support the relative type.

2.1.2.3 Miscellaneous motions

Besides the joint-related motions there are 3 additional motions. Two of these control the location and the orientation of the avatar within 3D space while the third one resets joints to their predefined orientation.

The ACF supports 2 ways of navigating within the 3D environment. The first is moving the avatar and animating it to give the illusion that he walks while the other one is sliding the avatar to its destination.

The ResetJoints class is used to reset joints to their resting position. It can reset the joints to that position or it can store a desired resting position for each joint for future use.

2.2 Communication

The communication part as seen in Figure 1 has the task to receive commands from the BCI, to translate them to a format that is understandable to the framework, to send the translated commands for execution and to send feedback back to the BCI. In order to be able to do all of the above tasks in parallel it uses the ACE framework components for concurrent computing.

This part of the ACF runs in a separate thread mainly to prevent program lockups caused by the receiving of data and also to make sure that currently running motions are not interrupted by the newly received motions.

The basic principle of the communications part is as follows. First it checks if any UDP packets were received. These packets are nothing more then basic strings which are terminated with a semi-colon. Then it parses the command out of the packet and executes the command. When the appropriate action based on the received command has been executed the framework sends feedback back to the BCI about what actions have been performed.

2.2.1 UDP Protocol

The UDP Protocol consists of a string which can be divided into a command part and a data part. The command part tells the framework what to do while the data part provides the necessary parameters for the execution of the command.
The generalized syntax for the UDP protocol is:

```
command.parameter1 parameter2 … parameterN;
```

The command keyword can be one of the following 5:

- motion
- python
- movement
- reset
- playMotion

Each of these command keywords must be separated from the data part of the packet by a period (.). If they are not separated with a period the framework will ignore that command and read a new packet from the socket. It is very important to point out that every packet apart from playMotion that must be terminated by a period, has to be terminated by a semi-colon (;) or the aforementioned parser will not be able to parse and execute the command.

2.2.1.1 Motion command

The motion command tells the ACF that the following data part describes a motion. The framework builds a python command string from the received UDP packet and pushes it onto the motions stack. When the playMotion command is received all the motions from the stack are popped from it, combined and executed.
Syntax:

```
motion.avatarName limb bodySide direction motionType angle duration;
```

Explanation of the keywords:

- limb – tells the program which joint to use. This variable can only be one of the 6 generalized joint names (Arm, Elbow, Wrist, Hip, Knee, and Ankle). If other generalized joint names are needed then their appropriate motion classes will need to be constructed first and the serverThread updates as explained in section 3.
- bodySide – is used to determine which limb will be moved. It can be either L or R representing left and right
- direction – the direction we want to move the chosen limb. It can be U, D, L or R (up, down, left and right)
- motionType – determines how to move the limb. It can be either R (for relative) or A (for absolute). It is important to know that a relative motion increments or decrements the current rotational angle depending if the angle parameter is positive or negative while an absolute motion rotates the limb to the angle parameter directly
- angle – the angle to which the limb will be moved to. Because of uncontrolled problems in the implementation of PIAVCA, using negative angles is not encouraged as these may not postion the joint into the desired postition
- duration – defines how long the motion will play. The shorter the duration parameter the faster the motion will be animated. The duration parameter also influences how long the communications part of the ACF will be put to sleep

Example:

```
motion.bill Arm L U A 1.2 1;
```

This command moves the left arm of the avatar bill in an upward direction to an angle on 1.2 radians within 1 second.

2.2.1.2 python command

The python command tells the parser within the communications part of the ACF that the following data part is a direct python command written as a string. The parser retrieves the name of the avatar for which the python command is meant for and then retrieves and sends the whole python command for execution.

Syntax:

```
python.avatarName pythonInstruction;
```

Explanation of the keywords:

- avatarName – name of the target avatar
- pythonInstruction – the python command written as a string

It is important to point out that if the python command is written incorrectly an error will show up within the Studierstube command window

Example:

```
python.bill avatar.playMotion(motion1, core.getTime());
```

where “motion1” is the string

```
Piavca_stb.ArmUpDownMotion(''bill'',Piavca_stb.ArmUpDownMotion.LEFT,1.2,2,Piavc
a_stb. ArmUpDownMotion.REL)
```

This example orders the avatar named bill to play the motion called motion1

2.2.1.3 movement command

The movement command tells the ACF that the following data part describes a motion within the 3D environment. The protocol is a bit different here then because the first parameter within the data part tells the parser if the avatar needs to be moved or rotated. When the parser knows which motion it has to execute exactly it builds the corresponding python command string and saves it on the motions stack.
Syntax:

```
movement.motion avatarName direction length;
```

Explanation of the keywords:

- motion – differentiates between walk rotate motions. It can be either walk, slide or rotate
- avatarName – the name of the avatar that is about to be moved or rotated
- direction – if the motion keyword is walk or slide, then direction is a PIAVCA vector written as Vec(x,y,z). It is important to point out that if we want to move the avatar on a horizontal plane then y must be 0 while x and z can vary.
  If the keyword is rotate then direction is either LEFT or RIGHT
- length – duration motion plays (it also indicates the angle if the rotate subcommand is chosen)

Example

```
movement.walk bill Vec(0,0,1) 10;
```

This command will move the avatar bill in a horizontal direction for 10 seconds

2.2.1.4 reset command

The reset command tells the ACF that either one or multiple joints within the data part need to be either reset to their resting orientation or that a new resetting orientation has to be saved for the joints. This command is executed directly when the parser assembles the appropriate python command in string form.

Syntax:

```
reset.avatarName joints resetFlag duration desiredRotation;
```

- avatarName – name of the targeted avatar
- joints – is a string which describes which joints will be targeted by the command. It contains pairs of generalized joint names and indicators on which body side the joints are located separated by commas (e.g. (Arm L,Hip R,)). It is important to point out that the last name-indicator pair must also be terminated by a comma and that no blank spaces must exist between the comma and either the indicator or joint name. If we want to either reset or save a new reset orientation for all joints in a given avatar then we simply write (All) as the joints parameter
- resetFlag - is a simple string which is either true or false and determines if the joint will be reset to its reset orientation or if a new reset orientation will be saved for the target joint. resetFlag has to value true if we wish to reset the limb or limbs while it is false if we wish to set a new reset orientation
- duration – is a numerical value which tells the ACF how long the reset is going to take. Recommended value: 0
- desiredOrientation - is a quaternion which is composed of an angle and PIAVCA Vector. Its syntax is equal to the C++ syntax: Quat(angle, Piavca_stb.Vec(x,y,z)). Quat() commands the PIAVCA vector to create a new quaternion, angle is our desired angle in radian and Piavca_stb.Vec(x, y, z) is our desired rotation axis

Example:

```
reset.bill (Arm L,Hip R,Wrist R,) true 0 Quat(1,Piavca_stb.Vec(0,1,0));
```

This command resets the left upper arm joint and the right hip and wrist joint to a predetermined rotation. Note that the Quat(…) parameter has no function here.

```
reset.bill (Arm L,Hip R,Wrist R,) false 0 Quat(1,Piavca_stb.Vec(0,1,0));
```

This command changes the current reset orientation to the one defined in the Quat(…) parameter.

Note that the reset command should be called after the playMotion command.

2.2.1.5 playMotion command

The playMotion command tells the ACF parser that it needs to lookup the motions stack and play the motions contained within. Since playing single motions one behind the other would defeat the purpose of this project the ACF uses one of the main features of PIAVCA. It takes all the motions in the motions stack, adds them to a single complex motion and sends them to be played. Before the parser returns to the UDP receiving socket to get a new UDP packet it is put to sleep for the sum of the durations of tall the combined motions. This approach ensures that all motions will be correctly played and that they will not be interrupted before they are complete.

Syntax:

```
playMotion.
```

Example:

```
motion.bill Arm L U A 0.75 1;
motion.bill Elbow L U A 1 1;
motion.bill Wrist L D A 1 1;
playMotion.
```

Figure 4: The result of executing the example detailed in section 2.2.1.5

These commands will make the avatar bill move its left arm and elbow upward and its wristdown simultaneously. The resulting pose will be the same as the one given in Figure 4. It is important to point out that contrary to the rest of the UDP packets this one needs to be terminated like a command by a period and not a semi-colon.

3. Guidelines for reusing and extending the ACF

The ACF is dependant on correct naming of the joints. The issue with the naming is that identical joints can be named differently on different avatar models. Therefore the ACF uses the PIAVCA feature to use a text file named JointNames.txt which contains all possible aliases of a given joint. Its format is as follows:

```
"joint name1" "alias1" "alias2" "alias3"
"joint name2" "alias1"
```

Since the ACF has great customization potential here are some guidelines to ensure that the customization process is easy and fast:

When introducing a new joint:

- Create a new motion class which inherits from GenericMotion and name it according to the naming conventions described in 2.1.2 Motions
- Include the new motion class in Stb_piavca.i
- Edit or append the new joint to the existing parser within the serverThread() method located in the MyApp class so it can be recognized by the parser

When introducing a new animation or command:

- Modify the existing parser within the serverThread() method located in the MyApp class

When changing the IP and port of the communications part:

- Modify the existing port and IP number within the serverThread() method found in the MyApp class

4. Results

The Avatar Control Framework was tested with a simulated BCI which was represented by a Matlab script. This script sent UDP packets containing various commands to the ACF running on the same computer. In this phase the ACF is capable of playing all of the required motions correctly. It is also capable of executing relative and absolute motions, moving and rotating one or multiple avatars within the 3D environment and combining an unlimited number of motions. Some examples can be seen in Figure 5.

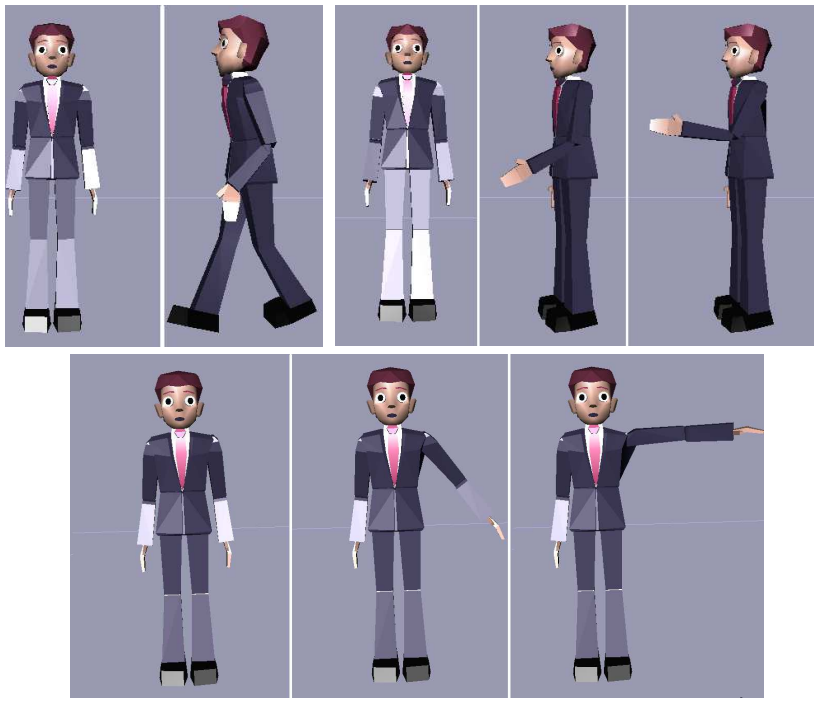

Figure 5: Examples showing a few motion combination possibilities: rotation and walk motion combined (upper left), rotation and movement of the arm and elbow joints combined (upper right), simple hand motion (lower)

The most important restriction currently is that non humanoid avatars are not supported. This however can be easily changed by some minor customization. Another minor issue is that there are no limits on moving the joints which can lead to some awkward poses.

5. Acknowledgements

I would like to thank my supervisor Alessandro Mulloni for his excellent tips and help given during the development of the framework. I would also like to thank Robert Leeb and Prof. Dieter Schmalstieg for clearly stating the requirements and comments on the development of the Framework.

6. References

[1] Studierstube 4 Subchapter Viewer by Antonio Rella
http://studierstube.icg.tu-graz.ac.at/doc/pdf/Stb4Viewer.pdf

[2] PIAVCA Documentation
http://www.cs.ucl.ac.uk/staff/m.gillies/piavca/

[3] The Studierstube Augmented Reality Project
http://studierstube.icg.tu-graz.ac.at/

[4] Cal3D - 3d character animation library Project homepage
https://gna.org/projects/cal3d/

[5] ACE API Overview
http://www.cse.wustl.edu/~schmidt/ACE-overview.html

[6] ACE API Documentation
http://www.cs.wustl.edu/~schmidt/ACE.html