D6.6 VIRTUAL CHARACTER PLAYER DEMONSTRATION

Grant Agreement nr	856879
Project acronym	PRESENT
Project start date (duration)	September 1st 2019 (36 months)
Document due:	28/02/2022
Actual delivery date	04/03/2022
Leader	UPF
Reply to	josep.blat@upf.edu
Document status	Submission Version

Project funded by H2020 from the European Commission

Project ref. no.	856879
Project acronym	PRESENT
Project full title	**P**hotoreal **RE**altime **S**entient **ENT**ity
Document name	D6.6 - Virtual Character Player Demonstration
Security (distribution level)	PU
Contractual date of delivery	28/02/2022
Actual date of delivery	04/03/2022
Deliverable name	Virtual Character Player Demonstration
Type	Demonstration
Status & version	Submission Version
Number of pages	23
WP / Task responsible	UPF
Other contributors	UAu
Author(s)	Eva Valls, Josep Blat
EC Project Officer	Ms. Diana MJASCHKOVA-PASCUAL Diana.MJASCHKOVA-PASCUAL@ec.europa.eu
Abstract	Key aspects of the demonstrator of the proof of concept high-quality virtual character player with a very high degree of integrability undertaken mainly by UPF-GTI within the PRESENT project.
Keywords	Virtual character player, behaviour realizer, behaviour planner
Sent to peer reviewer	Yes
Peer review completed	Yes
Circulated to partners	No
Read by partners	No
Mgt. Board approval	No

Document History

Version and date	Reason for Change
1.0 15-01-2022	Document created by Eva Valls
1.1 15-02-2022	Version for internal review
1.2 04-03-2022	Revisions in response to review: final version submitted to Commission

Table of Contents

1 EXECUTIVE SUMMARY

This deliverable presents key aspects of the demonstrator of the proof of concept high-quality virtual character player with a very high degree of integrability undertaken mainly by UPF-GTI within the PRESENT project.

Integrability is grounded on a Behaviour Planner (BP) and a web-based Realiser based on the *Behaviour Markup Language* (BML) within the *SAIBA* multi-modal Behaviour Generation Framework coming from the research community on Embodied Conversational Agents (ECAs). This architecture, based on two independent modules which communicate via BML, is discussed in 4.1. It presents several advantages, such as, for instance, that the Realiser can directly receive BML input coming from other systems.

High quality of the interaction is based on the capabilities of supporting behaviours and intents of the BP and the development of the Realiser, offering a wide range of interactions and animation. Further details are given on the *4.2 Behaviour Realizer* section. Visual high quality has been achieved by including the better rendering capabilities developed in the first period of the project by UPF-GTI.

So that developers can create their own virtual assistants in the easiest possible way, making it accessible to the content creator, the BP, based on *Hybrid Behaviour Trees*, is exposed through interactive graphs supporting a pipeline that starts from editing, through debugging, until playing and publishing. Further details are available on the *4.3 Behaviour Planner* section.

UPF-GTI work supports PRESENT reference implementation and APIs, thus interfacing with Unreal Engine, the most widespread games engine nowadays. A Web Socket connection is used to allow the Planner and the Realizer to communicate with each other and with other applications. The protocol and the messages formatting is based on the description of the *D5.5 Protocols and APIs implementation* deliverable. The guidelines set out in section *4.4 Message protocol and format* below provide more detailed information.

The different sections of 4 cover the different aspects of the Virtual Character Player. Section 2 provides the essential background with respect to its role within PRESENT, section 3 provides a short introduction and includes terminology, and the final section 5 is devoted to conclusions.

2 BACKGROUND

This deliverable belongs to the task *WP6T5 Virtual Character Player* within WP6, and provides a bridge towards the more natural interaction of the (sentient) agent, by driving through the Behaviour Planner the behaviours of the agent, which result from deriving the behaviour, for instance, from the collected and analysed inputs to the agent by means of the *Social Signal Interpretation* framework of University of Augsburg leading to messages to be carried out by the agent, which are realised and rendered through a Realiser into an animated interactive character. This work is also integrated through the APIs of the reference implementation developed and implemented mainly by FS, which lead to animations of a higher quality character, implemented through Unreal Engine (which use as well the Motion Generator of higher quality provided by CM. On the other hand, the player is demonstrated through a couple of the use cases of the project, the Registration Authority Officer and the Virtual Clerk, and behaviours derived from the framework of UAu, at the current level of Proof of Concept of the agent. The player should provide a more accessible way to characters of the highest level of quality using widely available systems.

3 INTRODUCTION

UPF-GTI is in charge of creating a player for virtual characters that gathers as much data as possible from the user, makes it accessible to the content creator, and ensures that the virtual character displays natural reactions based on the messages received.

This deliverable describes the main elements developed and used to create the virtual character player, such as the different modules that make up the system and how to communicate between them. In addition to the demonstration of the performance of this player in different use cases.

3.1 Main objectives and goals

- Create a web based player that displays a virtual character that interacts with the user in a natural manner.
- Establish and develop a pipeline to customise the behaviour of the virtual character so the developers can create their own virtual character player in an easy way.

3.2 Terminology

- **BML:** Behaviour Markup Language (BML) is an XML description language used to control verbal and nonverbal behaviour of (humanoid) embodied conversational agents (ECAs). The standard defines the form and use of BML blocks, mechanisms for synchronisation, the basic rules for feedback about the processing of BML messages, plus a number of generic basic behaviours.

- **Behaviour Planner (BP):** It is responsible for deciding which multi-modal behaviours to choose for expressing the communicative intent (through speech, face expressions, gestures, etc) and for specifying proper synchronisation between the various modalities.

- **Behaviour Realizer (BR):** It is in charge of carrying out the actions specified in the BML message. This entails generating sound and motion (e.g., animation or robot movement) in such a way that the time constraints specified in the BML block are met.

- **Behaviour Manager (BM):** It is responsible for handling the combination of behaviours of different requests following the specified BML time constraints. That is, it determines when a behaviour of a new request that is sent before the realisation of previous requests has been completed has to be applied. How to combine the behaviours is specified in the composition attribute, which can be *merge*, *append* or *replace*.

- **Hybrid Behaviour Tree (HBT):** A BT is a system to determine which behaviour an AI should perform. HBT combines two types of workflow: vertically (used in BT) and horizontally.

- **JSON:** Standard format for data-interchange. It is used to represent simple data structures and objects (associated lists). This format derives from Javascript language.

- **NodeJS:** It is a programming environment for developing web applications, focusing on web servers. The programming language is Javascript and the architecture is event oriented with asynchronous input and output.

- **WebGLStudio:** Platform to create interactive 3D scenes directly from the browser. It allows you to edit the scene visually, code your behaviours, edit the shaders, and all directly from within the app.

- **Iframe**: HTML element used to embed another document within the current HTML document. In this deliverable it is used to refer to a HTML document with the scene published by WebGLStudio.

4 Virtual Character Player

This chapter presents the key aspects of the virtual character player demonstrator, and discusses the affordances it allows.

4.1 System architecture

The system is based on the SAIBA multi-modal Behaviour Generation Framework and the use of the *Behaviour Markup Language* (BML) [1] So, the system can be split into two main different modules: the *Behaviour Planner* (BP) and the *Behaviour Realizer* (BR). The main idea behind the first module is to generate actions or intents for the virtual characters, taking into account internal and contextual information and using a degree of intelligence. These actions or behaviours are formulated in BML. Regarding the second module, the BR, it receives, interprets and executes the behaviours generated in the BP, to play and render the virtual characters' animations shown on the screen. Both modules have been implemented by UPF-GTI, and they can be independently used, as discussed in later sections.

UPF-GTI Behaviour Planner is achieved through a variant of the *LiteGraph*[2] graphs represented as *Behaviour Trees*. A first version of this extension was implemented for a previous project, SAUCE[3], which has been continuously improved and extended throughout the PRESENT project. The first BP approach[4] aimed to control background virtual characters, where the quantity was more important than the quality. The current BP approach aims to control an Embodied Conversational Agent (ECA), which requires more precision, as well as control over dialogues and emotions in a synchronised manner, which is a more complex undertaking than the initial one for SAUCE.

UPF-GTI Behaviour Realizer is developed in *WebGLStudio*[5], its own open-source platform for creating interactive 3D scenes directly in the browser. This module is in charge of controlling the body animation, facial expressions, eye gaze and blink, the agent and contextual attributes given the actions that the character has to perform. So, it is the Virtual Character Player.

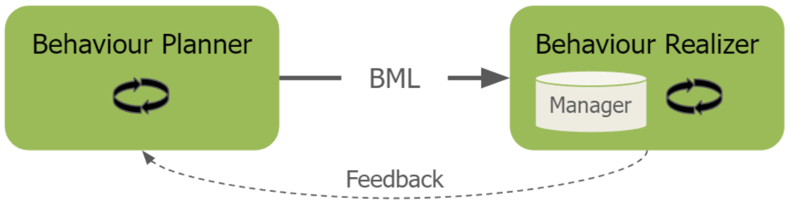


Figure 1. System workflow

4.2 Behaviour Realizer

As mentioned above, the Behaviour Realizer is in charge of interpreting and executing the behaviours generated in the Behaviour Planner, that is, of applying the actions on the character. Although it could be carried out on any platform, UPF-GTI has created a web based realizer on its 3D open source editor, WebGLStudio, for the Virtual Character Player.

4.2.1 Web based

One of the advantages of having a web based behaviour realizer is that it is accessible for anyone, on any device without having to install anything. It also gives the opportunity to explore the limitations of the web and improve the low-quality assets usually related to this field.

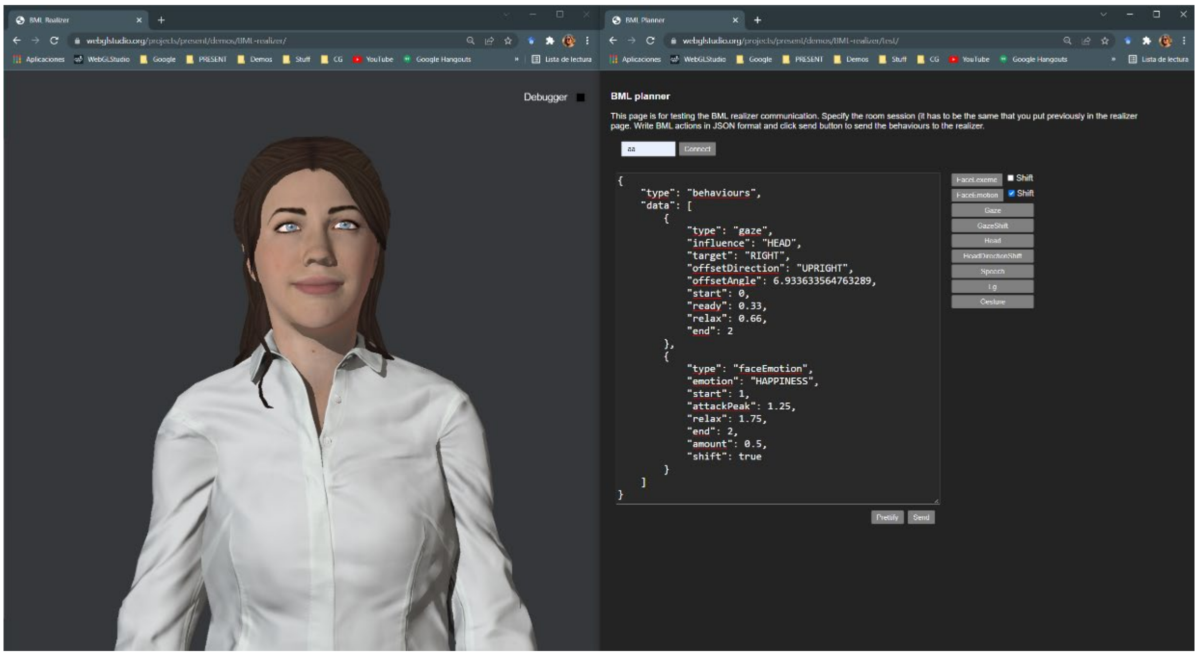
The set of actions which the realizer has to perform has been grouped into what we call *intentions*, a series of co-related actions organised in time through temporal marks. These intention actions may

[1] Kopp, Stefan, Brigitte Krenn, Stacy Marsella, Andrew Marshall, Catherine Pelachaud, Hannes Pirker, Kristinn Thórisson, and Hannes Vilhjálmsson. 2006. *Towards a Common Framework for Multimodal Generation: The Behavior Markup Language*.
[2] Agenjo, Javier. 2013. "LiteGraph." GitHub. https://github.com/jagenjo/litegraph.js.
[3] European Union's Horizon 2020 Research and Innovation Programme. n.d. "Smart Assets for re-Use in Creative Environments." Sauce. https://www.sauceproject.eu/.
[4] UPF-GTI and David Moreno. 2019. "HBTreeJS: Library for decision making using hybrid behaviour trees." GitHub. https://github.com/upf-gti/HBTreeJS.
[5] UPF-GTI and Javier Agenjo. 2013. WebGLStudio. https://webglstudio.org/.

include verbal and non-verbal speech, pose and emotion shifts among others following the BML standard, and also non-behaviour actions to support other aspects of the use cases.

This realizer has two main components: the Behaviour Manager and the Behaviour Realizer. The first one is in charge of handling the combination of behaviours of different requests following the specified time constraints. That is, it determines when a behaviour corresponding to a new request that is sent before the realisation of previous requests has been completed has to be applied. How to combine the behaviours is specified in the composition attribute. The behaviour manager, by default, merges the behaviours, but they can also be appended or replaced. The Behaviour Realizer is in charge of carrying out the actions after the manager has triggered them, such as making the virtual character say the speech received using lip sync in the specified time.

The Virtual Character Player, which is the default version of this realizer published on the web, is connected to a server. This way, any developer can use it, by sending the behaviours through a Web Socket connection. He or she just has to follow this link, connect the service that he or she uses to plan the behaviours to the server and put the same room both in the Virtual Character Player and in his or her service. It is also possible to to send messages through this page to test the actions and the messages format. The code of the player is to be published on GitHub, but for the moment is in private mode while the documentation is being finalised. The performance of the behaviours can be seen in this video.



BML planner

This page is for testing the BML realizer communication. Specify the room session (it has to be the same that you put previously in the realizer page. Write BML actions in JSON format and click send button to send the behaviours to the realizer.

```
{
    "type": "behaviours",
    "data": [
        {
            "type": "gaze",
            "influence": "HEAD",
            "target": "RIGHT",
            "offsetDirection": "UPRIGHT",
            "offsetAngle": 6.933633564763289,
            "start": 0,
            "ready": 0.33,
            "relax": 0.66,
            "end": 2
        },
        {
            "type": "faceEmotion",
            "emotion": "HAPPINESS",
            "start": 1,
            "attackPeak": 1.25,
            "relax": 1.75,
            "end": 2,
            "amount": 0.5,
            "shift": true
        }
    ]
}
```

Figure 2. *Left*: Default Virtual Character Player. *Right*: Web page for testing behaviours and messages formatting.

4.2.2 Use cases tested

The Virtual Character Player has been mainly tested and demonstrated with the Registration Authority Officer (RAO) use case. The behaviour logic had been done using the Behaviour Planner application and publishing the scene. One can interact with her through this link or see the demonstration on this video. For this case some non-behaviour custom actions had to be added to the realizer to support InfoCert API requests.

Furthermore, it is used for UPF-GTI's own use case, the Virtual Clerk (VC), with some modifications. A demonstration of the interaction can be seen in this video. It is accessible through this link for the realizer and this other for typing. You must open the realizer and put the room first, and then put the same room on the typing page. The interaction starts when you press the start button. The code is available on GitHub.

Not only has the player been used for the RAO and VC, but the University of Augsburg has been also using it for testing while waiting for the integration of the Motion Generation module into the reference implementation. A demonstration can be seen in the video.

4.3 Behaviour Planner

As previously mentioned, in order to design and execute the behaviours that have to be applied in the Virtual Character Player, UPF-GTI has developed a web application to provide the functionality of the so-called Behaviour Planner. The implementation is built on top of the previous work in the SAUCE project, where *decision trees* are used to decide the behaviour of an agent. The trees are displayed as a graph in a web editor to allow the visual design of complex behaviour using a node based programming approach. This web application allows users to create new behaviour trees and test them using the built-in chat (that simulates user input speech) within a sample scene created using a 3D web editor, WebGLStudio. The application is published here and the code is available on UPF-GTI GitHub.

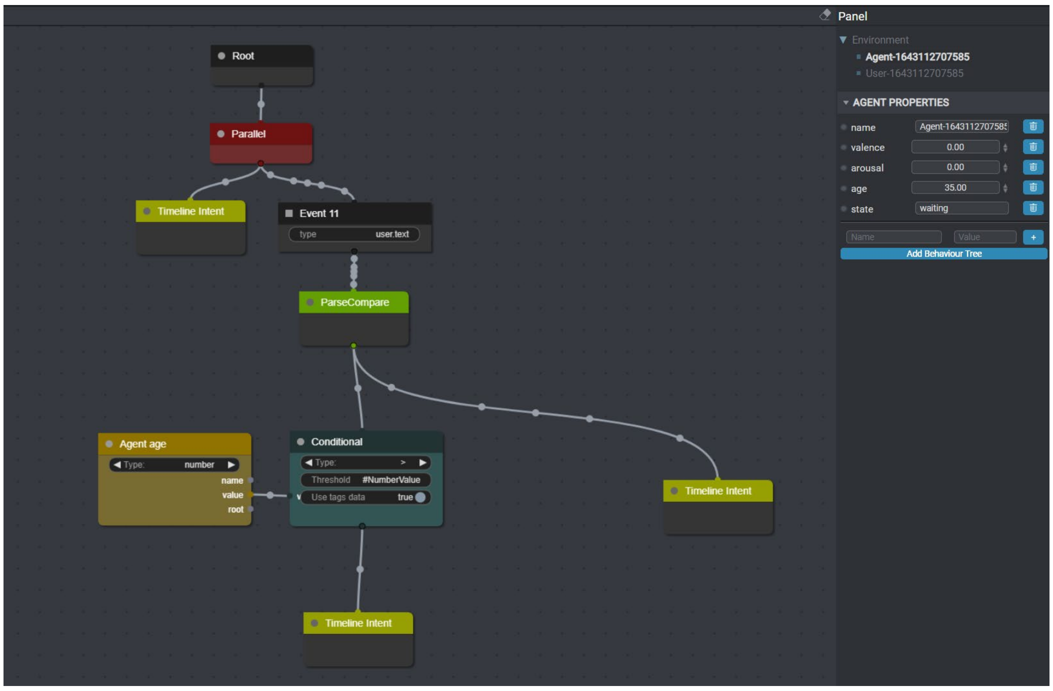
Depending on the user inputs (i.e. speech, emotion, gestures) and the context, the tree decides which actions the agent may take and/or what the application should do. This set of actions has been grouped into the *intentions*, or series of co-related actions organised on time through temporal marks already indicated. These intention actions may include verbal and non-verbal speech, pose and emotion shifts among others following the BML standard (or, more precisely, interoperability specification). The context can contain the information received from the user, other services or previous behaviours of the agent.

There is the option to open a WebSocket connection when the Behaviour Planner executes a behaviour. The web application uses a WebSocket connection with a server that acts as a broker to be able to receive and send data between applications. The description of the protocol and the format of the messages can be found in *4.4 Message protocol and format*.

In the following subsections we discuss in more detail the different aspects of the Behaviour Planner implementation, and its pipeline/workflow.

4.3.1 Graph system

The implementation of the graph system that aims to control the virtual agent's behaviour is called HBTree, which comes from Hybrid Behaviour Trees (HBT), embedded inside the Behaviour Planner. These are a hybrid approach of common Behaviour Trees. They are denoted as hybrid due to their capability to combine two workflows: one which is the common flow of a behaviour tree from top to bottom depending on the conditions of the nodes, and a second flow which is traversal to the tree, and allows to dynamically modify, through different nodes, the properties of some tree nodes. In Figure 3 this is visually represented through an example.

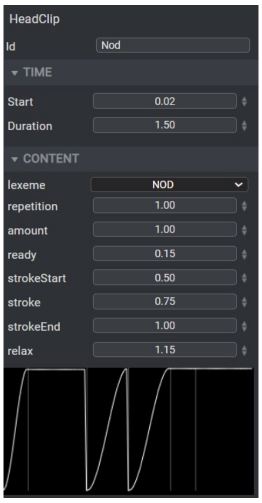


name	Agent-1643112707585
valence	0.00
arousal	0.00
age	35.00
state	waiting

Figure 3. HBT to decide what agent has to say depending on the user's age. If it's bigger than the agent's age (it's predefined as a property), the agent will say, for example "I'm younger than you". Otherwise the agent will say "I'm older than you".

Although new node types can be easily created, the library provides the necessary nodes for planning the agent's behaviour. The default ones are the following:

- **Root/Subroot**: Starting node of the tree/subtree execution.
- **Selector**: Executes child nodes from left to right until one succeeds (or at least does not fail).
- **Sequencer**: Executes child nodes from left to right until one fails. As its name indicates, it is useful for a sequence of actions/conditions.
- **Parallel**: Executes all child nodes in parallel.
- **Conditional** / **BoolConditional**: This node takes a value from the left inputs and compares it with the one set in the inner widgets. If the condition is passed, the execution continues along this branch. If not, the execution comes back to the parent node.
- **Timeline Intent**: This is the most complex node. It allows users to generate verbal and non-verbal behaviours at specific times and with specific duration. Users can generate different kinds of actions, such as facial expressions, gaze control, speech, gesture, etc. And place them at the time the user thinks it fits best, and with a custom duration. These actions follow the BML specifications (see Figure 4 next).


HeadClip

Id: Nod

TIME

Start: 0.02

Duration: 1.50

CONTENT

lexeme: NOD

repetition: 1.00

amount: 1.00

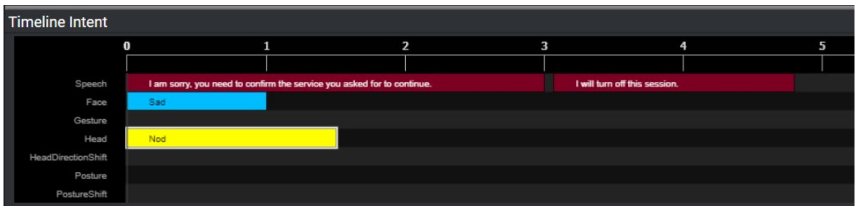
ready: 0.15

strokeStart: 0.50

stroke: 0.75

strokeEnd: 1.00

relax: 1.15


Timeline Intent

Speech: I am sorry, you need to confirm the service you asked for to continue. | I will turn off this session.

Face: Sad

Head: Nod

Figure 4. *Left*: Timeline shown on double click in the Timeline Intent node. Users can place the actions at the time the user thinks it fits best, and with a custom duration. *Right*: Each of these colourful rectangles (which represent different actions) can be double clicked, and the specific inspector for each type will open on the right inspector

- **ParseCompare**: Natural language processing node, where a set of phrases with tags or entities can be defined to be identified in the text passed through the branches. If the text passes the condition/ contains the text, tags or entities which are in the node, it continues with its children. If not, it goes back to the parent. It uses a library called *compromise*[6] to perform the natural language processing. Check the available entities there. It can be searched on the inspector putting # on the textbox.

- **SetProperty**: Puts a chosen property to a certain value.

- **Event**: This node is executed when an event of a given type occurs. Useful to capture when a message is received, for example, user text (user.text). The key of the message has to be specified as a property. It checks the message protocol.

- **TriggerNode/TriggerSubtree**: This node acts as a "bridge" between the current status, and another important part of the graph that might be in previous layers. Useful to create cycles in case several responses lead to dead ends and there is the need to go back to another stage.

- **CustomRequest**: This is used in case the developer needs something from the scene (hosted in the realizer). It can also be used to indicate that the developer has to carry out some other action not related to the character's behaviour (such as https requests).

- **HttpRequest**: (only works on the client side) This node allows making calls to external services/APIs. The inspector of the node allows the users to build the http message, with headers and body of the message. There are some templates to create the body with the required info (api-version, ids, texts…)

- **HttpResponse**: (only works the on client side) It parses the response of a HttpRequest node (this means that both are connected vertically) and detects if the code is the one set in the embedded widget (200, 201, 400)

[6] Kelly, Spencer, and Alex Corvi. 2011. "Compromise: modest natural-language processing." GitHub. https://github.com/spencermountain/compromise/.

4.3.2 Application modules

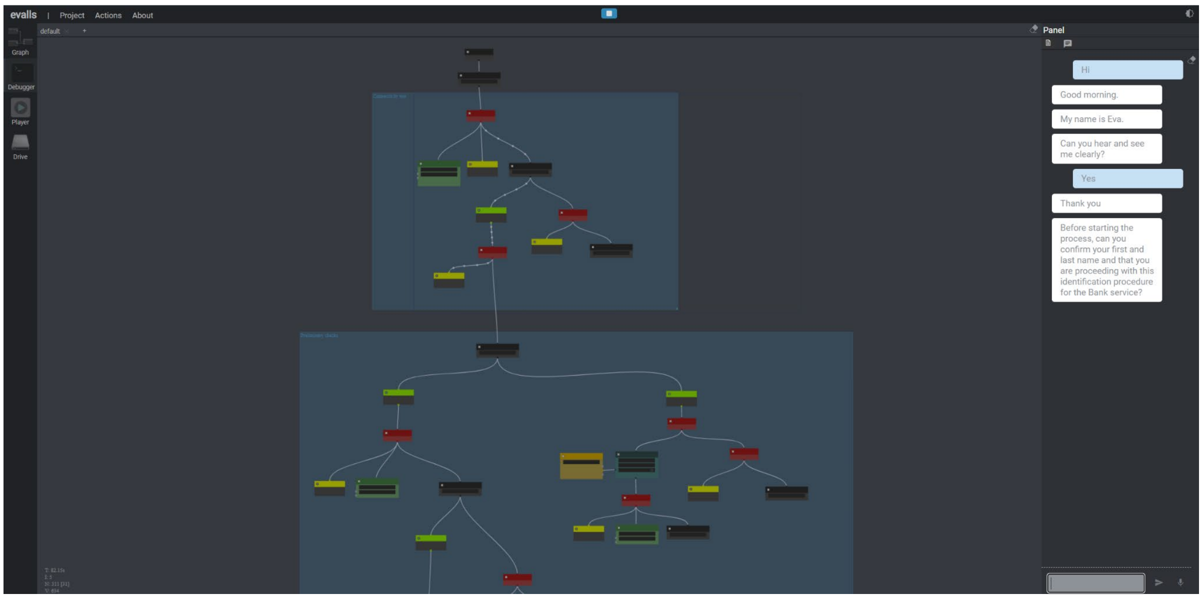
The Behaviour Planner application has different modules: Graph, Debugger, Player, Drive and Publisher.

4.3.2.1 Graph Editor

In the Graph Editor module, the developer can create the graph and edit the properties of the different elements. Each graph must start with a *Root* node. Sub-graphs can also be created, starting with a *SubRoot* node. The subgraph will be executed if the main execution passes through the leaf node *TriggerSubtree* with the id of the corresponding *SubRoot* node. A node can be searched by double-clicking on the 2D canvas, so that a search box appears. This can also be performed by right-clicking on the canvas and searching by type.

4.3.2.2 Debugger

In this module, the graph is just as in the Graph module, but this time the right inspector is a chat, where the developer can debug and test specific use cases and see in real time the execution of the graph (highlighting the path followed). The user can write on the text box or speak by pressing the microphone button (the text will appear on the text box when the speech ends).



Hi

Good morning.

My name is Eva.

Can you hear and see me clearly?

Yes

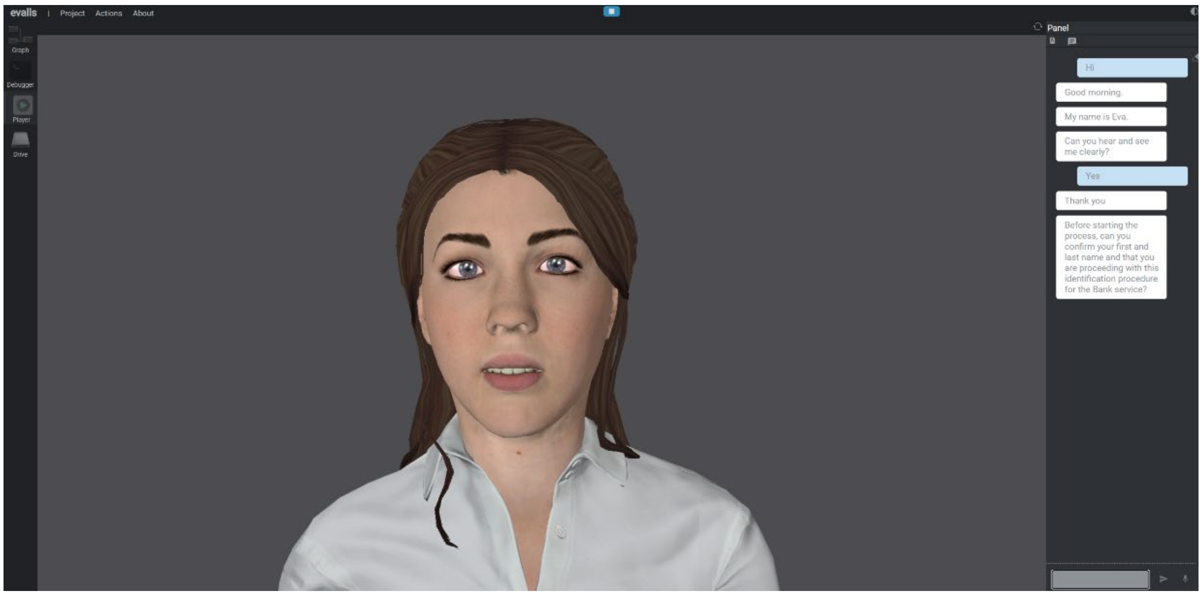
Thank you

Before starting the process, can you confirm your first and last name and that you are proceeding with this identification procedure for the Bank service?

Figure 5. Debugger module view.

4.3.2.3 Player

If the user wants to test not only the dialogue but also the non-verbal behaviour, the Player module provides a default scene to do so. The player is just an iframe containing a scene created in WebGLStudio. This iframe can be changed by going to *Actions→Change Iframe Scene*, but the new one has to be created also using the mentioned editor.



Hi

Good morning.

My name is Eva.

Can you hear and see me clearly?

Yes

Thank you

Before starting the process, can you confirm your first and last name and that you are proceeding with this identification procedure for the Bank service?

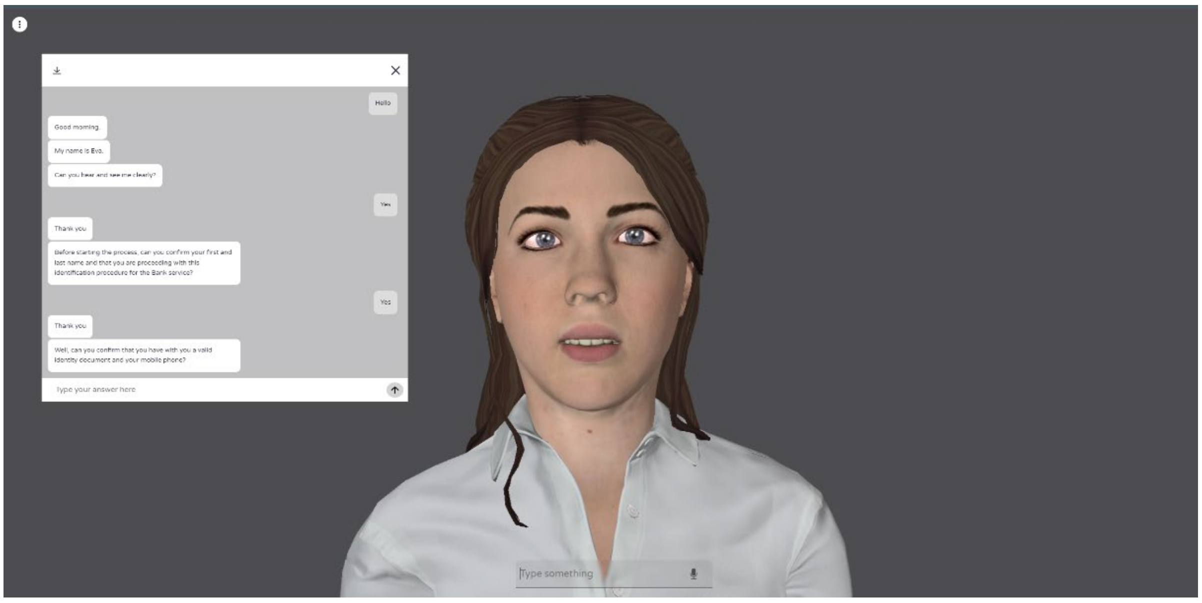
Figure 6. Player module view.

4.3.2.4 Drive

The user can register in the application and save his/her projects. So, this module is basically a File System where stored environments and graphs can be easily found and loaded into the current session.

4.3.2.5 Publisher

When the behaviour graph is complete and the user is satisfied with the results, the player can be published as an external web application. The published environment will have the same scene used for the tests, that is, it will be a WebGLStudio scene using the default Behaviour Realizer, explained in *4.2 Behaviour Realizer* section. The user can interact with the agent either by voice (by holding down the space key or the microphone button) or by text (by typing on the screen). It also has a chat function that can be activated or deactivated to keep track of the conversation. Therefore, this new application is the virtual character player and everyone can access it if they have the link without the need to have the BP application open.



Hello

Good morning.

My name is Eva.

Can you hear and see me clearly?

Yes

Thank you

Before starting the process, can you confirm your first and last name and that you are proceeding with this identification procedure for the Bank service?

Yes

Thank you

Well, can you confirm that you have with you a valid identity document and your mobile phone?

Figure 7. Virtual Character Player. Published behaviour planner with WebGLStudio scene.

4.3.3 BP with other applications

Because in the context of PRESENT the work of most of the partners is included in the reference implementation, developed in UE, the Behaviour Planner is also prepared to be integrated in this offline realizer following the *D5.5 Protocols and APIs implementation* document. And it can be used not only in the reference implementation, but also in any application to create new character players. There are two ways to do that: through WebSocket connections or including the BP library in the application.

4.3.3.1 WebSocket connection

The BP application is running at *wss://webglstudio.org/port/9003/ws/*, so external applications can send or receive data using a web socket connection. The other application only has to connect to the host and to the same room (specified on environment properties). The room is used for session persistence, that is, BP routes all messages originating from a specific client in a specific room to a single backend web server as well as other clients connected to the same room. More detailed information is described in the *4.4 Message protocol and format* section.

4.3.3.2 Stand alone library

The logic of the BP application is developed as a stand alone library called *behaviour-planner.js*, so that it can be easily integrated into any existing application and the graph system can be executed without the need for the editor. This library is built in NodeJS, so it works both on the server and in the web browser. At runtime, the library outputs BML blocks containing a number of behaviour elements with synchronisation. The planner also waits for messages from the Behaviour Realizer to inform it of the progress of the realisation as well as what is happening in the environment (e.g. user response). It is used to inform the planner (and possibly other processes) of the progress of the realisation process. This library can be found on UPF-GTI GitHub.

4.3.3.3 Within reference implementation

As described in *D5.5 Protocols and APIs implementation*, a WebSocket connection is used to execute a behavioural planning running in the BP application in the reference implementation. But it can be also executed inside UNREAL adding a NodeJS plugin[7] to run the BP library. To do that it is necessary to add the library code, its dependencies, the main file which uses the library and the JSON file exported from the BP application in PRESENT/Content/Scripts/. The plugin code compiled has to be put in PRESENT/Plugins/External.

4.4 Message protocol and format

The developer has to connect and communicate with the Virtual Character Player using the following message formatting and the BML actions types.

Each message must be in JSON format with ***type*** and ***data*** values:

- ***type***: Type of the message. The values can be "info", "session", "custom_action" and "behaviours". For the behaviours (bml actions), the value always has to be "behaviours".

- ***data***: Content of the message. For "behaviours", it is an array of behaviours. Each behaviour is an object with its specified attributes. For "custom_request", it can have different parameters, explained below.

First of all, the user has to connect to the server application creating a WebSocket connection. The host to which the user must connect is *wss://webglstudio.org/port/9003/ws/.* Once connected with the server application, he or she has to connect to the same session as the realizer application sending a message to the server specifying the room (the same put in the Virtual Character Player). The message has to looks like this:

[7] "Embed node.js as an unreal plugin." 2019. GitHub. https://github.com/getnamo/nodejs-ue4.

```
var msg = {
                type: "session",
                data: { token: "my room", action: "connect"}
}
ws.send(JSON.stringify(msg))
```

The room is used for session persistence, that is, the server routes all messages originating from a specific client in a specific room to a single backend web server as well as other clients connected to the same room. This way the user establishes a private server session between the Virtual Character Player and his/her service/application.

If the connection in a session succeeds, a message like the following one will be received:

```
{
                type: "info",
                data: { "Info: connected to a session with token "my room"}
}
```

4.4.1.1 BML actions

Although the actions that the realizer allows are based on the BML specification, they are not exactly the same. Here are the descriptions of each behaviour related to a specific tag:

FACE BEHAVIOURS

At the moment this only supports back-to-back lexemes, i.e. one lexeme starts when the other finishes, not at the same time or within the same time range as another lexeme is running.

- <faceLexeme>: This behaviour shows a (partial) face expression from a predefined lexicon.
- <faceEmotion>: This behaviour shows a set of face expressions through single emotion.

GAZE BEHAVIOURS

- <gaze>: This behaviour causes the character to temporarily direct its gaze to the requested target.
- <gazeShift>: This behaviour causes the character to direct its gaze to the requested target. This changes the default state of the ECA: after completing this behaviour, the new target is the default gaze direction of the ECA.

HEAD BEHAVIOURS

- <head>: Movement of the head, recalled from a gesticon by requesting the corresponding lexeme.
- <headDirectionShift>: Orient the head towards a target referenced by the target attribute. Like gazeShift with head as influence.

SPEECH BEHAVIOURS

- <speech>: Utterance to be spoken by a character. Realisation of the speech element generates both speech audio (or text) and speech movement.
- <lg>: Utterance to be spoken by a character. It gives the url of the audio (and text) for speech audio and movement.

GESTURE BEHAVIOURS

- <gesture>: Coordinated movement with arms and hands, recalled from a gesticon by requesting the corresponding lexeme.

By default, the realizer updates the blink and saccades automatically, but they can be triggered at any moment using the Gaze and Face Lexeme actions.

Further information about the description of each action and its possible values are provided in an Annex, as chapter *7*.

5 CONCLUSION

This deliverable has presented key aspects of the demonstrator of the proof of concept high-quality virtual character player with a very high degree of integrability.

The system architecture uses two modules, a Planner and a Realiser, both based on the BML interoperability specification; they can be used independently, through WebSocket connections implemented, and the Planner integrated into applications, and, in particular, in the PRESENT reference implementation, thus supporting UE as an offline realiser.

The Planner uses Hybrid Behaviour Trees, which are displayed as interactive and editable graphs, and supporting a full workflow, to allow content creators to easily create and test behaviours.

The virtual character player proof of concept has already been tested and demonstrated on the RAO and VC use cases, and to support University of Augsburg research. The proof of concept is also available through web links. It will be provided in GitHub with full documentation.

6 ANNEX

6.1 BML action attributes

Here are the descriptions of each behaviour related to a specific BML tag.

6.1.1 FACE BEHAVIOURS

Shared face attributes:

Attribute	Type	Use	Default	Description
amount	float	optional	0.5	A float value between 0..1 to indicate the amount to which the expression should be shown on the face, 0 meaning ’not at all’ and 1 meaning ’maximum, highly exaggerated’
shift	boolean	optional	false	A boolean value to indicate if the specified compound face expression has to become the new BASE state of the ECAs face. (<faceShift>)

Shared face sync attributes:

Attribute	Description
start	Beginning of face expression
attackPeak	Maximum expression achieved
relax	Decay phase starts, not for <faceShift> behaviours
end	Face expression ended, not for <faceShift> behaviours

6.1.1.1 *FaceLexeme*

Attribute	Type	Use	Default	Description
type	typeID	require		Value: *faceLexeme*
lexeme	openSetItem	require		Member of a set of lexemes (See table below)

LIP_CORNER_DEPRESSOR
LIP_CORNER_DEPRESSOR_LEFT
LIP_CORNER_DEPRESSOR_RIGHT
LIP_CORNER_PULLER
LIP_CORNER_PULLER_LEFT
LIP_CORNER_PULLER_RIGHT
PRESS_LIPS
MOUTH_OPEN
LOWER_LIP_DEPRESSOR
CHIN_RAISER
LIP_PUCKERER
TONGUE_SHOW
LIP_STRECHER
LIP_FUNNELER
LIP_TIGHTENER
LIP_PRESSOR
BROW_LOWERER
BROW_LOWERER_LEFT
LOWER_RIGHT_BROW
LOWER_BROWS
INNER_BROW_RAISER
OUTER_BROW_RAISER
RAISE_LEFT_BROW
RAISE_RIGHT_BROW
RAISE_BROWS
UPPER_LID_RAISER
CHEEK_RAISER
LID_TIGHTENER
EYES_CLOSED
BLINK

WINK
NOSE_WRINKLER
UPPER_LIP_RAISER
DIMPLER
DIMPLER_LEFT
DIMPLER_RIGHT
JAW_DROP
MOUTH_STRETCH

Example of message:

```
      {
                  type: "behaviours",
                  data: [{
                        type: "faceLexeme",
                  lexeme: "LIP_CORNER_DEPRESSOR",
                  start: 0,
                  attackPeak: 0.25,
                  relax: 0.75,
                  end: 1,
                  amount: 0.5,
                  shift: false
            }]
      }
```

6.1.1.2 FaceEmotion

Attribute	Type	Use	Default	Description
type	typeID	require		Value: *faceEmotion*
emotion	openSetItem	require		Member of a set of emotions [HAPPINESS, SADNESS, ANGER, SURPRISE, CONTEMPT, DISGUST, FEAR, NEUTRAL]

Example of message:

```
      {
                  type: "behaviours",
                  data: [{
                        type: "faceEmotion",
                  emotion: "ANGER",
                  start: 0,
                  attackPeak: 0.25,
                  relax: 0.75,
                  end: 1,
                  amount: 0.5,
                  shift: true
                  }]
      }
```

6.1.2 GAZE BEHAVIOURS

Shared gaze attributes:

Attribute	Type	Use	Default	Description
target	openSetItem	require	CAMERA	A reference towards a target instance that represents the target direction of the gaze [CAMERA, RIGHT, LEFT, UP, DOWN, UPRIGHT, UPLEFT, DOWNLEFT, DOWNRIGHT]
influence	openSetItem	optional		Determines what parts of the body to move to affect the gaze direction. [EYES, HEAD, NECK.]
offsetAngle	angle	optional	0.0	Adds an angle degrees offset to gaze direction relative to the target in the direction specified in the *offsetDirection*. Range recommended: [-25,25]
offsetDirection	direction	optional	RIGHT	Direction of the offsetDirection angle [RIGHT, LEFT, UP, DOWN, UPRIGHT, UPLEFT, DOWNLEFT, DOWNRIGHT]

6.1.2.1 Gaze

This behaviour causes the character to temporarily direct its gaze to the requested target. The influence parameter is read as follows:

gaze attributes:

Attribute	Type	Use	Default	Description
type	typeID	require		Value: *gaze*

gaze sync attributes:

Attribute	Description
start	gaze starts to move to new target
ready	gaze target acquired
relax	gaze starts returning to default direction
end	gaze returned to default direction

Example of the message:

```
    {
        type: "behaviours",
        data: [{
                type: "gaze",
            influence: "EYES",
            target:"UPRIGHT",
            offsetDirection: "LEFT",
            offsetAngle: 24.5,
            start: 0,
            ready: 0.33,
            relax: 0.66,
            end: 2.0
    }]
    }
```

6.1.2.2 GazeShift

This behaviour causes the character to direct its gaze to the requested target. This changes the default state of the ECA: after completing this behaviour, the new target is the default gaze direction of the ECA.

gazeShift attributes:

Attribute	**Type**	**Use**	**Default**	**Description**
type	typeID	require		Value: *gazeShift*

gazeShift sync attributes:

Attribute	**Description**
start	gaze starts to move to new target
end	gaze target acquired

Example of the message:

```
        {
                type: "behaviours",
                data: [{
                                type: "gazeShift",
                        influence: "EYES",
                        target:"UPRIGHT",
                        offsetDirection: "LEFT",
                        offsetAngle: 24.5,
                        start: 0,
                        end: 2.0
                }]
        }
```

6.1.3 HEAD BEHAVIOURS

6.1.3.1 Head

Movement of the head, recalled from a gesticon by requesting the corresponding lexeme.

head attributes:

Attribute	**Type**	**Use**	**Default**	**Description**
type	typeID	require		Value: *head*
lexeme	openSetItem	require		Refers to an animation or a controller to realise this particular head behaviour. Minimum set offered by all

				realizers: [NOD, SHAKE, TILT]
repetition	int	optional	1	Number of times the basic head motion is repeated.
amount	float	optional	0.0	How intense is the head nod? 0 means immeasurable small; 0.5 means ”moderate”; 1 means maximally large

head sync attributes:

Attribute	**Description**
start	start of the preparation phase
ready	end of the preparation phase
startStroke	start of the stroke
stroke	stroke of the head motion. Note that the meaning of this sync point becomes undefined if *repetition* > 1
strokeEnd	end of stroke
relax	start of retraction phase
end	end of the head motion

Example of the message:

```
      {
            type: "behaviours",
            data: [{
                        type:"head",
                  lexeme: "NOD",
                  repetition:1,
                  start: 0,
                  ready: 0.3,
                  strokeStart: 0.3,
                  stroke: 1,
                  strokeEnd: 1.6,
                  relax: 1.6,
                  end: 2.0
            }]
      }
```

6.1.3.2 HeadDirectionShift

Orient the head towards a target referenced by the target attribute. Like gazeShift with head as influence.

headDirectionShift attributes:

Attribut e	**Type**	**Use**	**Default**	**Description**
type	typeID	require		Value: *headDirectionShift*

target	targetID	require		Target towards which the head is oriented [CAMERA, RIGHT, LEFT, UP, DOWN, UPRIGHT, UPLEFT, DOWNLEFT, DOWNRIGHT]

headDirectionShift sync attributes:

Attribute	**Description**
start	Beginning of motion
end	Reached desired direction; set this direction as new BASE state

Example of the message:

```
        {
                type: "behaviours",
                data: [{
                                type: "headDirectionShift",
                        target: "DOWNLEFT",
                        offsetDirection:"LEFT",
                        offsetAngle: 10.0,
                        start: 0,
                        end: 2.0
                }]
        }
```

6.1.4 SPEECH BEHAVIOURS

Shared speech sync attributes:

Attribute	**Description**
start	Start of the speech
end	End of the speech

6.1.4.1 Speech

speech attributes:

Attribute	**Type**	**Use**	**Default**	**Description**
type	typeID	require		Value: *speech*
text	string	require		Attribute that contains the text to be spoken.

Example of the message:

```
{
        type: "behaviours",
        data: [{
                type: "speech",
                text: "Hi. How are you?",
                start: 0,
                end: 2.0
        }]
}
```

6.1.4.2 Audio

audio attributes:

Attribute	Type	Use	Default	Description
type	typeID	require		Value: *lg*
url	string	require		Audio url that has to be reproduced. The audio has to be public, that is, it has to be reproduced by putting the link on the browser.

Example of the message:

```
{
        type: "behaviours",
        data: [{
                type: "lg",
                url:
                "https://webglstudio.org/projects/present/audios/RAO/How%
                20old%20are%20you.wav",
                start: 0,
                end: 2.0
        }]
}
```

6.1.5 GESTURE BEHAVIOURS

6.1.5.1 Gesture

Coordinated movement with arms and hands, recalled from a gesticon by requesting the corresponding lexeme.

gesture attributes:

Attribute	Type	Use	Default	Description
lexeme	openSetItem	require		Refers to an animation or a controller to realise this particular gesture. [WAVE, PRESENT]
repetition	int	optional	1	Number of times the basic head motion is repeated.

gesture sync attributes:

Attribute	Description
start	beginning of gesture
ready	end of gesture preparation phase
startStroke	start of the stroke
stroke	gesture stroke
strokeEnd	end of stroke
relax	start of retraction phase
end	end of gesture

Example of the message:

```
{
```

```
            type: "behaviours",
                  data: [{
                        type:"gesture",
                  lexeme: "WAVE",
                  repetition:0,
                  start: 0,
                  ready: 0.3,
                  strokeStart: 0.3,
                  stroke: 1,
                  strokeEnd: 1.6,
                  relax: 1.6,
                  end: 2.0,
                  amount:1
            }]
      }
```