NTNU
Norwegian University of
Science and Technology

Multimodal Behaviour Generation Frameworks in Virtual Heritage Applications
A Virtual Museum at Sverresborg

Michael James Stokes

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Torbjørn Hallgren, IDI
Co-supervisor: Jo Skjermo, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

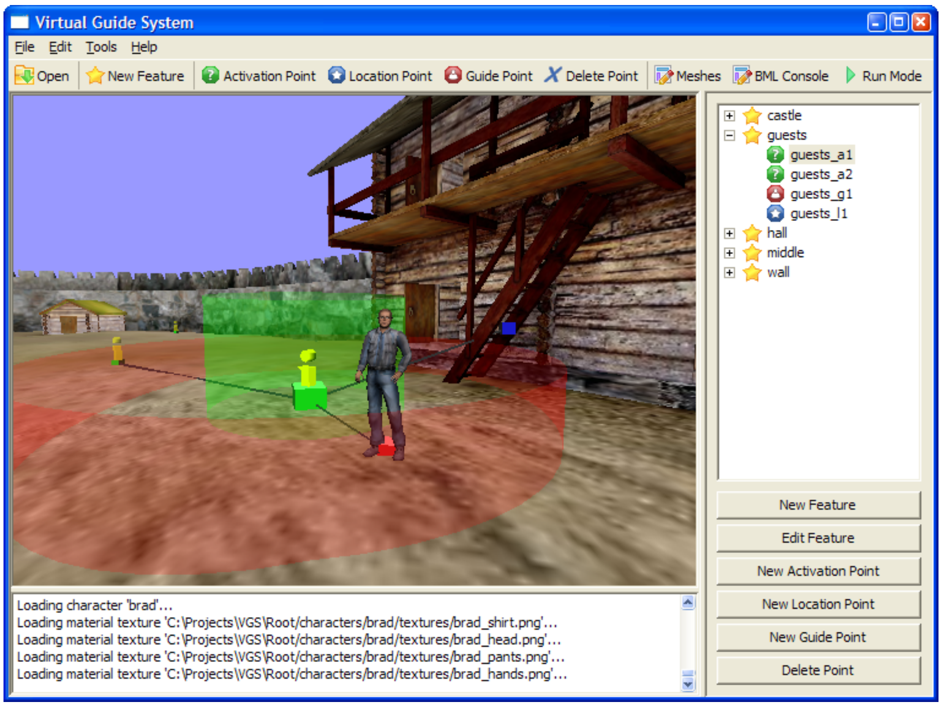
Problem Description

Investigate markup languages that describe human behaviour for the purpose of animating virtual human agents. Consider these methods in the context of virtual heritage applications.

Assignment given: 15. January 2009
Supervisor: Torbjørn Hallgren, IDI

Abstract

This masters thesis proposes that multimodal behaviour generation frameworks are an appropriate way to increase the believability of animated characters in virtual heritage applications. To investigate this proposal, an existing virtual museum guide application developed by the author is extended by integrating the Behavioural Markup Language (BML), and the open-source BML realiser SmartBody. The architectural and implementation decisions involved in this process are catalogued and discussed. The integration of BML and SmartBody results in a dramatic improvement in the quality of character animation in the application, as well as greater flexibility and extensibility, including the ability to create scripted sequences of behaviour for multiple characters in the virtual museum. The successful integration confirms that multimodal behaviour generation frameworks have a place in virtual heritage applications.



```
Loading character 'brad'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_shirt.png'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_head.png'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_pants.png'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_hands.png'...
```

Contents

1 Introduction

A virtual heritage application uses virtual reality technologies to allow a user to see and experience an environment of cultural significance that they would otherwise be unable to enjoy. The environment may no longer exist physically, or it may be impractical for the user to visit it in person. Virtualising cultural heritage allows it to be delivered anywhere and at any time.

This report is the latest in a series of virtual heritage projects undertaken at the Department of Computer and Information Science at the Norwegian University of Science and Technology. It builds upon previously completed work focusing on King Sverre's Zion fortress, the ruins of which can be seen in Trondheim. A virtual model of the fortress has already been created, along with software that allows a user to explore it.

A previous project undertaken by the author added a virtual guide character to the Sverresborg virtual museum. The guide character took on the appearance of a medieval warrior, and presented each feature of interest from the site using vocal narrations. The guide could both lead the user on an automatically constructed tour of the site, and follow the user whenever they decided to take the initiative.

Although these high-level behaviours worked well, the guide character appeared stiff and unnatural. It spoke with a human voice, but lacked the body language and non-verbal communication that is expected when humans speak to one another. In film and similar media, this kind of low-level detail can be added manually by animators, but in an interactive system a more flexible and automatic solution that can adapt to external events is required.

Recently there has been great interest in **multimodal behaviour generation frameworks**[1]. These frameworks formalise a processing pipeline that turns functional intentions into realistic behaviours, and those behaviours into renderable animation. There is hope that elements of these frameworks can be standardised, to allow interoperability and bring the benefits of the research to a wider range of applications.

This masters thesis will propose that multimodal behaviour generation frameworks are an appropriate way to increase the believability of characters in virtual heritage applications. It will evaluate several frameworks and implement a working prototype of the Sverresborg virtual museum, with a virtual guide character driven by a multimodal behaviour generation framework.

2 Related Work

This section surveys related work on multimodal behaviour generation frameworks in general, and multimodal agents in virtual heritage applications specifically.

2.1 Multimodal Behaviour Generation Frameworks

A number of different multimodal behaviour generation frameworks have been designed by independent groups. This section examines several frameworks and the expression languages they define.

2.1.1 SAIBA with FML and BML

Researchers around the world working on virtual conversational agents have often found that they need to implement a lot of functionality outside of their specific area of focus in order to have a complete and testable system [1]. This constant reinventing the wheel is time consuming and undesirable. It became obvious that a method of working together and re-using complimentary components would be beneficial. With these goals in mind, an international group of experts met at the University of Reykjavik in 2005 and developed SAIBA, a multimodal behaviour generation framework.

The SAIBA framework defines a three step process: **intent planning**, **behaviour planning** and **behaviour realisation** [1]:

- In **intent planning**, the agent's goals are decided. An example goal would be to tell someone that the local football team won. This intent is then specified using a standardised intent specification language called FML, the Functional Markup Language. The FML data is passed to the next step.
- In **behaviour planning**, the functional intent is decoded from FML and an appropriate way to carry out the intent is formulated. The exact formulation may vary depending on context and custom, but will in the common case involve verbal and non-verbal behaviour. This behaviour is then specified using a standardised behaviour specification language called BML, the Behavioural Markup Language. The BML data is passed on to the final step.
- In **behaviour realisation**, the requested behaviour is decoded from BML and actuated on the embodiment of the agent. In the common case this involves animating a 3D model, however other forms of rendering are also possible.

The most interesting aspect of the SAIBA framework is that it defines two standard specification languages, FML and BML, as the interfaces between its three components. In theory any components compatible with these languages can be connected together to form a working system[1]. A standardised interface also makes SAIBA components attractive choices for integration into other research and production projects.

Work on the SAIBA framework is ongoing. So far neither FML nor BML have reached a level of stability that can support widespread adoption, however there are already a number of projects making use of one or the other [2].

As the Behavioural Markup Language (BML) will be a key component of this project, it is explored in more depth in Section 4.1.

2.1.2 ACE MAX with MURML

MAX is a Multimodal Assembly eXpert, a virtual agent that demonstrates how to perform complex construction tasks[3]. He is an example use of the Articulated Communicator Engine (ACE), a multimodal behaviour generator and realiser. ACE was developed at the University of Bielefeld, and is centred around a behavioural specification language called MURML.

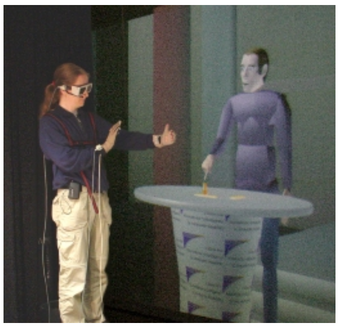


Figure 1: A user interacting with the University of Bielefeld's MAX

Like BML, MURML is an XML-based language that serves as the interface between the behavioural planner and the realiser[4]. Unlike BML, which takes on a fairly free form, MURML is built around a unit called an "utterance". An utterance is a string of spoken words paired with the gestures that accompany them. Each MURML utterance definition therefore includes a passage of text, annotated with time markers at key points, and a behaviour specification that is synchronised to the markers.

Behaviours are specified either by their communicative intent, such as to point at an object, or in more detail via their spatiotemporal features, such as the location and orientation of different body parts[4]. This blurs the distinction between behaviour planning and realisation somewhat. In addition, a sophisticated constraint system allows sub-movements to be synchronised.

Although sophisticated and feature-rich, MURML has not been embraced as a standard to the extent that BML has.

2.1.3 MagiCster with DPML and APML

MagiCster is a European project that aims to develop believable conversational interface agents[5]. Like SAIBA, the MagiCster architecture divides the problem into sub-components: the mind and the body. The mind is responsible for deciding the agent's intentions which it then communicates to the body to be realised either on a virtual model or in some other way.

One of MagiCster's sub-projects is the development of DPML and APML, both XML-based languages:

- **DPML** is the Discourse Plan Markup Language. It specifies the communicative intent of the agent, and is the output of the "mind" component [6].
- **APML** is the Affective Presentation Markup Language. It specifies the communicative function, or the "meaning", to be realised [6].

A component called the **plan enricher** corresponds roughly to the **behaviour planning** step of the SAIBA architecture. The plan enricher sits between the mind and the body, and translates DPML into APML.

While MagiCster's DPML and APML seem superficially similar to SAIBA's FML and BML, there are subtle differences. MagiCster's APML specifies communicative function, such as "greet Bob", while SAIBA's BML specifies the specific behavioural implementation of function, such as "nod head toward Bob" and "smile". Thus MagiCster leaves more interpretation work to the body or realiser component.

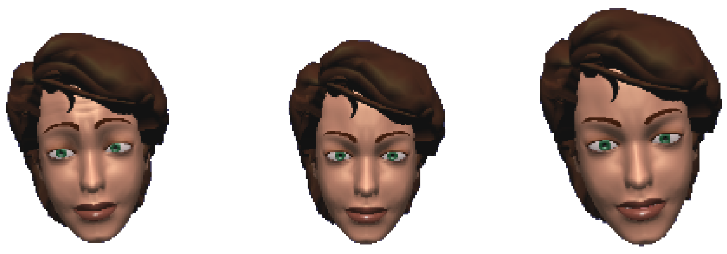


Figure 2: A MagiCster virtual agent displays different emotions in response to APML

2.1.4 SmartBody and BML Realiser

SmartBody is an open-source character animation system developed to serve the "behaviour realiser" role in the SAIBA framework. It was developed at the Information Sciences Institute and the Institute for Creative Technologies at the University of Southern California[2]. Although it does not yet implement the full range of the Behaviour Markup Language, it has become one of the more popular implementations of the standard.

Related work was also performed by two students at the University of Reykjavik, who completed a project adapting SmartBody for use in standalone applications, outside of the ICT/USC environment that it previously relied upon [7]. Their work has made it significantly easier to use SmartBody in new projects.

As SmartBody will form an integral part of this project, further discussion of its strengths, weaknesses and architecture is deferred to Section 4.2.

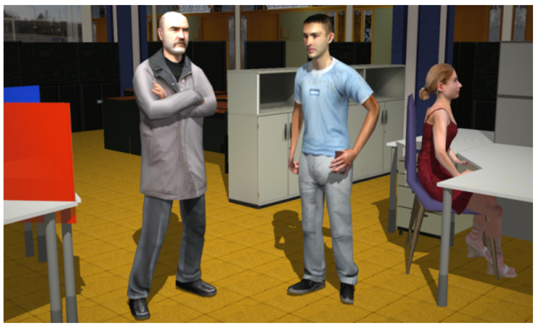


Figure 3: The University of Reykjavik's system (SmartBody and Panda3D)

2.2 Multimodal Guides in Virtual Heritage

Guides with multimodal performance capabilities have been used previously in virtual heritage applications, although generally without employing a full behaviour generation framework.

2.2.1 An Augmented Reality Virtual Guide for Sverresborg Museum

The previous project completed by the author at the Norwegian University of Science and Technology in 2008 involved creating a virtual guide character in a virtual Sverresborg museum. The guide character took on the appearance of a medieval warrior, and presented each feature of interest from the site using vocal narrations. The guide could both lead the user on an automatically constructed tour of the site, and follow the user whenever they decided to take the initiative[8]. This work was in turn based upon earlier work undertaken by Jorge Ordóñes Serrano[9], who in 2004 developed a simple fixed-path virtual tour guide for the same museum.

The author's 2008 system used a simple animation technique in which at any time one pre-recorded, keyframe-based skeletal animation could be playing, either in looped mode or as a once-off performance[8]. Transitions between animations were also possible.

The use of pre-recorded animations placed a significant limitation on the type and range of behaviour the guide character could exhibit. Although theoretically any type of behaviour could be implemented, it would always require authoring a new keyframe-based skeletal animation, which is no easy task and is well beyond the skill-set of the target audience. As a result, only a small set of animations were authored and the guide character appeared stiff and unnatural. It spoke with a human voice, but lacked the body language and non-verbal communication that is expected when humans speak to one another.

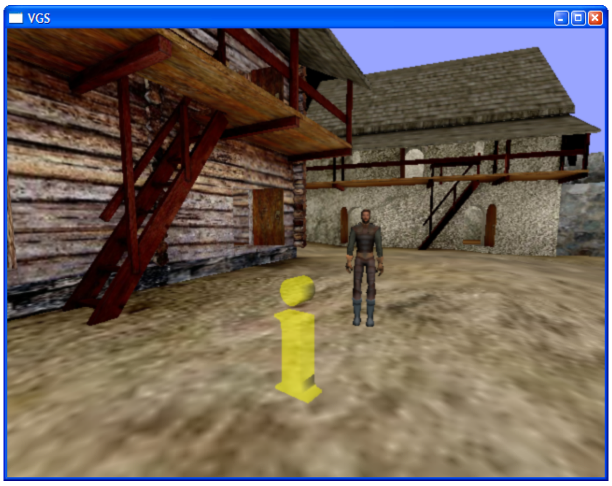


Figure 4: The Virtual Sverresborg Museum

2.2.2 A Multimodal Guide for Virtual 3D Models of Cultural Heritage Artefacts

The University of Palermo has developed a virtual multimodal guide application that takes the user on a tour of a historic room inside Palazzo Steri [10]. Built on a platform of XHTML+Voice and VRML/X3D, all interaction occurs within a web browser, making it very easy for distant visitors to enjoy the exhibits.

The system is notable because the vocal communication with the virtual guide character is bidirectional – the guide not only narrates to the user, but the user can speak commands and ask simple questions[10].

Although the exhibit itself is 3D, the guide character appears to be a pre-rendered 2D image.

3 Goals and Requirements

This section sets out the goals of the project and the specific functional requirements that will be addressed.

3.1 Goals

The primary goal of the project is to replace the primitive animation system that currently drives the Virtual Guide application with a more modern and flexible system based on a multimodal behaviour generation framework. The SAIBA framework[1], and more specifically the Behavioural Markup Language (BML) has been selected as the target technology, as it appears to have the best chances of standardisation. It also has the best freely-available software components at the current time, namely SmartBody and related works [2].

The secondary goal of the project is to use the new structured animation system to implement a layer of non-verbal behaviours for the virtual guide character, and any ancillary characters within the virtual museum. These behaviours will consist of both unconscious behaviours, generated automatically by the system according to the social context, and conscious behaviours that are explicitly requested. A simple scripting mechanism will be created to allow behavioural scripts to be authored and executed as part of the museum experience.

Finally, the utility of both BML as a language and SmartBody as a component will be evaluated within the virtual heritage setting.

3.2 Limitations

Although the SAIBA framework defines three components and two languages (intent planning, behaviour planning and behaviour realisation, connected by FML and BML)[1], this project will only attempt to incorporate the behaviour realisation component and the behavioural markup language. The earlier components (intent planning, functional markup language) have not yet reached a level of maturity and stability that suggest them for this kind of project.

This means that the Virtual Guide System will be using BML and the SmartBody realiser as its output stage, but will still need to generate behavioural descriptions to be realised. This will be accomplished via a combination of simple algorithms, adapted from the previous project, and scripted sequences, which will be a new feature.

3.3 Requirements

This section attempts to set out a formal list of requirements that the new system shall fulfil:

- Whereas the previous system supported a single animated human character in the museum world, the new system shall support any number of characters. One character will be the virtual guide, while the remaining characters can be used to represent contemporary people going about their lives.
- All control of the animated human characters shall be performed via the Behavioural Markup Language, by directing small fragments of XML+BML to each character.
- Subconscious non-verbal behaviour will be generated automatically for the guide character based on the current context, and passed to the character via BML for realisation.
- The system shall support creating scripts consisting of timed delivery of BML fragments. These scripts can be attached to features of interest within the museum, and played back on cue. Scripts can use BML, augmented as necessary, to create and delete characters, move and animate characters, and have characters speak pre-recorded content.
- To the extent possible, the previous functionality of the system shall be preserved. This includes the features of the guide character, the physics simulation, and the support for multiple virtual reality and augmented reality configurations.

3.4 Plan

With the background and requirements now established, Section 4 will explore the Behavioural Markup Language and the SmartBody component in more detail, determining their strengths and weaknesses and identifying how they can be incorporated into the project. Then, Section 5 will present a software architecture that builds upon the previous project, but includes the new components and requirements. Section 6 will describe the implementation of the proposed architecture, identifying the challenges and solutions that arose during the implementation phase. Finally, Section 7 will evaluate the new software and comment upon the suitability of multimodal behaviour generation frameworks, and BML/SmartBody in particular, for virtual heritage applications.

4 Technologies

This section examines the technologies that have been selected for use in the project: the Behavioural Markup Language (BML) and the leading behaviour realiser supporting that language, SmartBody.

4.1 BML

The Behavioural Markup Language is the interface between the behaviour planning and behaviour realisation components of the SAIBA framework [1]. It is an extensible XML-based language designed for maximum interoperability between heterogeneous components.

Based on the survey of behaviour description languages in Section 2, BML appears to have the best chance for widespread adoption and standardisation. As such it has been selected as the language to be used in this project.

4.1.1 History

Despite its growing popularity, BML remains in a state of flux. The original version of the language, hence-forth known as the pre-1.0 version, was published in 2006[11]. Since then, its primary contributors have been working together on an official version 1.0, which is not yet complete. The two versions share the same goals and are structurally very similar, but version 1.0 aims to unify and simplify the the language.

Despite the potential advantages of BML 1.0, this project must instead adopt the pre-1.0 version, as the only available BML realiser has not yet been updated to support version 1.0 – see Section 4.2.1. As a consequence of this, the remainder of this section describes the pre-1.0 version.

4.1.2 Structure

The basic unit of expression in BML is the **behaviour block**. A behaviour block is a fragment of well-formed XML, consisting of an XML header and a top-level `<bml>` element. The `<bml>` element in turn contains one or more **behaviour elements**. Each behaviour element describes a behaviour to be executed by a part of the human body.

An example behaviour block follows. It consists of four behaviour elements:

- An instruction to look at someone named John
- Some text to be spoken
- A nod of the head, to be executed at the start of the speech, and

- A beat of the hand, to be executed upon the spoken word “text”

```
<?xml version="1.0"?>
<bml>
  <gaze id="b0" target="john"/>
  <speech id="sp" type="application/ssml+xml">
    <text>Speak this <mark name="t"/>text</text>
  </speech>
  <head id="b1" type="NOD" stroke="sp:start"/>
  <gesture id="b2" type="BEAT" stroke="sp:t"/>
</bml>
```

4.1.3 Behaviour Types

The pre-1.0 version of BML defines ten behaviour elements[12]:

- The `<torso>` element directs the character to assume a new posture in its torso region (spine and shoulders). The posture is loaded from a library of pre-created postures based on a posture name.

- The `<legs>` element directs the character to assume a new posture in its legs. The posture is loaded from a library of pre-created postures based on a posture name.

- The `<gesture>` element directs the character to perform a coordinated movement of one or more limbs. Five different gesture types are defined: point, beat, conduit, generic and lexicalised. In addition to specifying the gesture type, the element also specifies which hand is to be used, where the gesture is to occur in space, how fast it should be executed, and any special modifications to the orientation of the hands and the flow of the movement.

- The `<head>` element directs the character to change the orientation of its head with respect to the current rest orientation. Different types of movement, such as nod, shake and tilt can be executed.

- The `<gaze>` element directs the character to look in a specific direction or at a named target within the world. The gaze can incorporate synchronised movement of the eyes, head, neck, torso and legs, or it can be restricted to a subset of these.

- The `<face>` element directs the character to form a facial expression. The expression can be described in terms of movement of the eyebrows, mouth and eyelids, or parts thereof, or it can be described using the standard Facial Action Coding System (FACS).

- The `<lips>` element directs the character to move its lips, forming a named visme. Lip movement is combined with the current facial expression.

- The `<speech>` element directs the character to speak a passage of dialogue. Speech can be specified as text that should be rendered using a text-to-speech engine, or as a reference to a recorded audio file.

- The `<body>` element is a mixed element that can both direct the character to assume a new full-body posture, as well as direct it to move to a new location, typically by walking or running. Postures are specified by name, and loaded from a library of pre-created postures. If movement is requested, the target location is specified as a named location in world space. The final arrival position and orientation can also be modified, such that the character can be directed to stop in front of and facing the target object.

- The `<wait>` element is a complex element that instructs the behaviour realiser to delay execution either until a condition becomes true, or until a specified delay has elapsed. It is designed to allow pauses in the performance.

4.1.4 Synchronisation

If more than one behaviour is specified within a single behaviour block, the default interpretation is to execute all of the specified behaviours concurrently. This mechanism provides a very simple type of synchronisation, by starting a set of behaviours at the same moment. A much more feature rich synchronisation system is also implemented, allowing more complex compound behaviours to be expressed.

Every behaviour element has by definition two key moments, or “sync points”: its start and its end. BML defines five additional sync points that occur between start and end: ready, stroke-start, stroke, stroke-end and relax [12]. In order:

- `start` is the earliest moment in the behaviour.

- `ready` occurs when the character has performed any preparatory movements that put it in the correct pose to begin the behaviour.

- `stroke-start` occurs once the character stops holding the ready pose and begins the main action of the behaviour.

- `stroke` occurs at the moment of maximum effort or maximum movement, typically in the middle of the behaviour.

- `stroke-end` occurs when the main action is complete. The character will hold this pose until it begins to relax.
- `relax` occurs when the character stops holding its stroke-end pose and begins to relax toward its rest pose.
- `end` is the last moment in the behaviour.

Naturally not all behaviours have all seven sync points, so in some cases sync points may be collapsed onto one another – for example a behaviour that does not require any preparatory action may combine `start` and `ready`.

The BML synchronisation system is based around the concept of synchronising a sync point from one behaviour with a sync point from another behaviour. For example, one can define a beat of the hand followed by a nod of the head by synchronising the `stroke` of the nod with the `end` of the beat:

```
<?xml version="1.0"?>
<bml>
  <gesture id="b" type="BEAT"/>
  <head id="n" type="NOD" stroke="b:end"/>
</bml>
```

Every behaviour element is required to have a unique id name in its `id` attribute, allowing it to be referenced by other behaviour elements. Dependent behaviours can then include an attribute for each of their sync points that they wish to synchronise with other behaviours, specifying the id name of the other behaviour and which of its sync points should be used.

It is also possible to include an offset in a synchronisation request. A sync point from one behaviour can be synchronised to a sync point from another behaviour, plus or minus some fixed time offset[12].

In many circumstances hard synchronisation is unnecessary. It is often sufficient to say that one event must occur before or after another. This is somewhat verbose to achieve in the pre-1.0 version of BML, where one must use the `<constraint>` element. BML 1.0 plans to introduce a more concise syntax that allows the use of predicates in normal synchronisation attributes, for example `stroke="after(b:end)"`[2].

4.1.5 Extensibility

BML aims to be able to describe a comprehensive range of behaviours in a way general enough to be useful in a wide range of applications. This is referred to as the core language. It also recognises that specific applications will have additional requirements, both in terms of custom behaviours and more detailed description of the existing behaviours. To support these scenarios, BML has an extension mechanism.

The first part of the extension mechanism is support for levels of description within behaviour elements[2]. Any behaviour element can have one or more child elements named `<description>`, each with a `level` and `type` attribute. The behaviour realiser is free to choose the highest-level description element that it understands, or to fall back to the core BML element. The contents of description elements is not defined by the BML specification, allowing other behaviour and animation languages to be embedded.

The second part of the extension mechanism is the use of XML namespaces to allow new XML elements and attributes to be added without the risk of disturbing the core BML namespace now or in the future[2]. New behaviour elements can be added with an appropriate prefix, while additional information can be supplied to existing behaviour elements via prefixed attributes.

4.2 SmartBody

SmartBody fills the role of the behaviour realiser in the SAIBA framework. It accepts BML as input, and generates hierarchical skeletal animation data as output. It is an open-source project that originated at and is maintained by the Institute for Creative Technologies at the University of Southern California[13].

The project was released to the public in August of 2008, and since then has seen continuous gradual improvement.

4.2.1 Limitations

Although SmartBody is the premiere BML behaviour realiser, it is still in a relatively early stage of development, and has limited functionality in many important areas.

4.2.1.1 BML Compliance

The current version of SmartBody does not implement the new draft 1.0 version of BML, opting instead to implement the pre-1.0 version [7]. Therefore, as noted in Section 4.1.1, this project must also adopt pre-1.0 BML.

Even within pre-1.0 BML, SmartBody currently only supports a subset of the BML behaviour elements defined in the standard. The following elements are supported[14]:

- The `<gaze>` element is well supported. SmartBody characters can be directed to gaze at other each other, or at fixed world points. When characters gaze at each other, they can focus on different parts of the body, to achieve eye contact or other effects. An offset can be applied to offset the gaze in a fixed direction. The gaze is realised by turning the eyeballs, neck and and upper spine in a natural way. SmartBody extends standard BML in a number of ways, allowing the behaviour markup to specify a roll of the head, custom joint limits and smoothing for all of the joints that are involved in the gaze.

- The `<head>` element is also quite well supported, with both the `nod` and `shake` behaviours implemented.

- SmartBody supports the `<face>` element, allowing facial expressions to be described using the Facial Action Coding System (FACS).

- The `<body>` element is only partially supported. It can be used to change the character's current resting posture by loading a new posture from the library of pre-created poses. The new posture is used until replaced. SmartBody does not currently support character movement, which will be discussed in more detail in Section 4.2.1.2.

- Both forms of the `<speech>` element are supported, however if text-to-speech is to be used, SmartBody simply forwards the text to an external TTS engine, which is then responsible for rendering the speech. Pre-recorded audio is fully supported – a speech element can specify an audio file which will be played back by the SmartBody process. This will be discussed in more detail in Section 4.2.1.3.

SmartBody defines a pair of additional behaviour elements that extend standard BML. This is done using XML namespacing, as per the standard.

- The `<sbm:animation>` element directs the character to execute a pre-created keyframe-based animation, selected by name from a library of such animations. This extension element is very important, as it provides a mechanism to fill in some of SmartBody's missing support for other BML elements. For example, the `<gesture>` element is critically missing, so it is not possible to direct a character to perform any kind of arm gesture by specifying gesture type, magnitude, handedness, etc. Luckily the same effect can be approximated by instructing SmartBody to playback a pre-created animation of the desired gesture. This is not as flexible, but is the only available method at the current time.

- The `<sbm:event>` element is a partial back-port of the `<event>` element found in BML 1.0. It directs SmartBody to emit an event back to the BML source at a given moment.

4.2.1.2 Locomotion Limitations

As mentioned briefly in the previous section, SmartBody does not currently support any kind of locomotion (walking, running, etc). Support for locomotion is scheduled for implementation in the medium term on the project's roadmap.

This shortcoming is a critical problem for this project, which seeks to use SmartBody to animate a virtual guide character that accompanies a user through a virtual or augmented reality museum. The guide character must be able to walk around the museum, leading the user to exhibits of interest, and following them when they decide to deviate from the planned route, ready to provide assistance [8].

As walking is not an optional feature, a locomotion system will need to be developed outside of SmartBody. The proposed system is described in Section 5.2.2.

4.2.1.3 Speech Limitations

SmartBody implements both text-to-speech and pre-recorded dialogue. For this project, pre-recorded dialog is the most attractive option for two reasons:

- It allows much greater freedom in the style of voice, with the possibility of using professional voice actors to record high quality material that fits with the subject and time period of the museum.

- It does not require an external text-to-speech engine, which would otherwise need to be licensed.

The pre-recorded audio system does not support a positional/directional 3D audio mixer, as was implemented in the previous virtual guide project[8]. This is not thought to be a significant problem, as in this application it is more important that the speech be audible to the user, rather than to have realistic 3D audio effects.

SmartBody also requires that audio files be uncompressed. This may prove to be a problem in a real deployment of the application, where there could be a significant quantity of recorded audio. It is also a step backward from the support for compressed audio in the previous project. Again however this is not thought to be a significant problem for this prototype project.

Nonetheless, support for compressed audio and positional/directional 3D audio mixing would be very desirable in production quality code.

Possible speech solutions are discussed in Section 5.2.3.

4.2.2 Architecture

Figure 5 shows an overview of SmartBody's architecture, illustrating its input, processing pipeline and output.

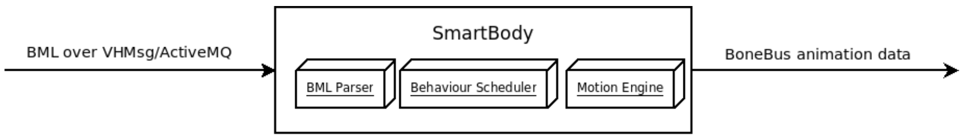


Figure 5: SmartBody architecture

4.2.2.1 VHuman

SmartBody is designed to be a component in the distributed VHuman architecture developed at the University of Southern California. The VHuman architecture is a component architecture where different components, such as a behaviour realiser, a voice realiser, input awareness systems, etc, cooperate toward a common goal by exchanging messages. These components may run on the same computer, or on physically separate hardware. The common theme is that they always run as separate processes, receiving inputs and transmitting outputs. This approach allows components to be developed independently.

The core of the VHuman architecture is a message passing system called VHMsg, which is in turn based on Apache's ActiveMQ product. Once an ActiveMQ server has been established, multiple clients connect to it and exchange messages in real-time. In the case of SmartBody, BML would be received from the behaviour planner or other source via VHMsg and ActiveMQ. SmartBody supports a number of other command messages over VHMsg, in addition to BML payloads.

Although the VHuman architecture is a core part of SmartBody's design, it is possible to use it without establishing a VHuman "network". Commands, including BML payloads can be delivered directly to the SmartBody process via the keyboard.

4.2.2.2 BoneBus

Normally the output of a VHuman component would be sent via VHMsg to another VHuman component for consumption. In the case of SmartBody, the output is animation data, in the form of regular updates for the translations and orientations of a set of bones in one or more character skeletons. The updates need to be very regular to achieve smooth animation. SmartBody's authors concluded that these regular updates would be too much to carry over VHMsg, and decided instead to implement a custom, light-weight protocol to carry the output animation data from SmartBody to the 3D renderer component. This protocol is called BoneBus.

BoneBus is implemented over a raw TCP stream, and carries one batch of bone translations plus one batch of bone orientations per character per frame. A client library is provided with SmartBody to ease integration of BoneBus into renderers.

4.2.2.3 Internal Renderer

SmartBody also features an internal renderer, able to visualise all of the presently active characters. Use of the renderer is optional, and it would normally be disabled in a configuration where an external renderer is connected via BoneBus. It is however a useful debugging tool.

5 Architecture

This section develops the software architecture for the new virtual guide application, incorporating BML and SmartBody as the core character animation system. It begins with a quick review of the architecture of the previous project, before analysing how this architecture must be modified and extended to accommodate the new requirements. In the process it identifies and proposes solutions to various integration challenges.

5.1 Previous Work

The previous virtual guide application designed and implemented by the author [8] was based upon a simple but effective architecture that split functionality into logical modules (see figure 6):

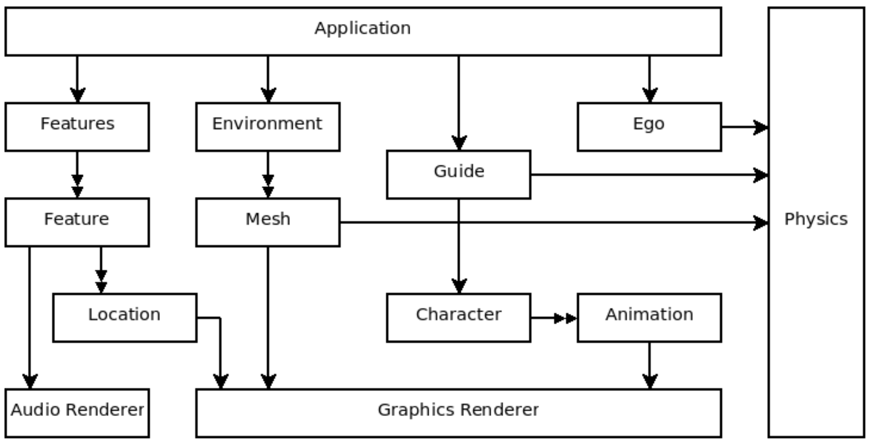


Figure 6: Architecture of the previous Virtual Guide project

- The **Application** module was responsible for coordinating the startup, shutdown and function of the software.
- The **Features** module was responsible for loading and maintaining the database of features of interest, that is, parts of the museum that the user may want to visit and learn more about.
- The **Environment** module was responsible for loading and maintaining the virtual representation of the museum environment, in the form of one or more 3D meshes corresponding to real and/or virtual geometry.
- The **Guide** module was responsible for the intelligence and behaviour of the virtual guide character. It monitored the user's own behaviour and adjusted that of the guide to try to be as helpful and facilitating as possible.

- The **Ego** module represented the user's presence within the virtual world, and could be configured to support either virtual reality or augmented reality.

- The **Physics** module interfaced with NVIDIA PhysX[15] to provide a limited physical simulation of the virtual environment, so that both the virtual guide and the user's ego were constrained by appropriate physical laws.

- The **Character** module performed simple character animation for the virtual guide, based upon a 3D mesh and skeleton with a library of pre-authored skeletal animations.

- The **Graphics Renderer** module was responsible for rendering the virtual world. It was able to work together with OpenGL and VRJuggler[16] to support varied VR and AR configurations.

- The **Audio Renderer** module was responsible for decoding and rendering the guide character's pre-recorded speech.

5.2 Considerations

The following sections address specific considerations drawn from the requirements set out in Section 3.3, as well as solutions to integration problems raised in Section 4.2.1.

5.2.1 SmartBody Process Integration

As described in Section 4.2.2.1, SmartBody has been designed to be a component within the VHuman architecture, operating as a standalone process and accepting its input over VHMsg/ActiveMQ, and sending its output over a proprietary protocol named BoneBus. As a result of this design, using it outside of a VHuman setup requires some extra work.

Some thought has been given to this problem in the SmartBody codebase, which is divided into a library and a wrapper executable. The library contains all of the core functionality, such as BML interpretation and animation, while the wrapper executable links to the library and provides something runnable. The theory is that other applications wishing to integrate SmartBody technology can simply link to the core library, and communicate with it directly by passing in BML and pulling out skeleton data.

Unfortunately this code separation has not been strictly enforced, and no clean interface between the core library and the executable has been defined. As a result, the two are intimately interconnected in a way that makes it practically impossible to incorporate the library in another application. This situation is scheduled to be rectified in the medium term, according to the project roadmap [17].

In the immediate term, another solution is required. The Panda BML Realiser project at the University of Reykjavik[7] developed a simple solution that keeps SmartBody running out-of-process, but eliminates the need for a VHMsg/ActiveMQ connection by piggybacking BML input onto the BoneBus output protocol, as shown in figure 7:

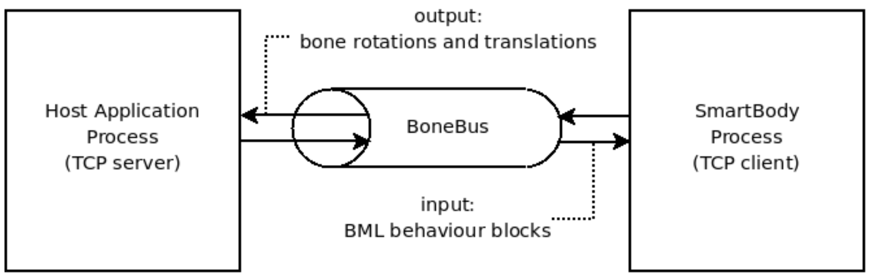


Figure 7: BoneBus with a BML back channel

In this configuration, the host application process takes on the role of the renderer, and becomes the BoneBus server. The SmartBody process is the BoneBus client, establishing a TCP connection to the server. This connection carries bone rotation and translation data from SmartBody to the renderer as per normal. In addition, the host application is able to send BML behaviour blocks to SmartBody over the same socket.

Despite the complex inter-process-communication (IPC) requirements, keeping SmartBody out-of-process actually simplifies the implementation by insulating the host application from SmartBody internals. This allows SmartBody to be rebuilt or upgraded without necessarily affecting the host application.

5.2.2 Locomotion

As discussed in Section 4.2.1.2, SmartBody does not presently support any form of locomotion. Characters can assume different animated poses, but they can not be directed to walk or run through the world. Locomotion is an essential part of the virtual guide system, and was fully implemented in the previous iteration of the project. The requirements set forth in Section 3.3 stipulate that core functionality like locomotion must be preserved through the integration of SmartBody and BML. A solution is therefore required.

The SmartBody team have scheduled the implementation of locomotion for the medium term on their development roadmap[17]. The possibility of going ahead and implementing locomotion in SmartBody now was considered, but discarded, as it appears that large-scale changes would be required in the SmartBody library. These would likely take longer than the time available for this project.

A more practical solution is to implement locomotion support outside of SmartBody. The application code can intercept BML behaviour elements that describe locomotion requests, and instead of forwarding them to the external SmartBody process (which would not understand them), it can instead perform its own actions. These actions would include moving the character's origin through world space, with assistance from the PhysX physics module to avoid collisions. Figure 8 illustrates the proposed architecture:

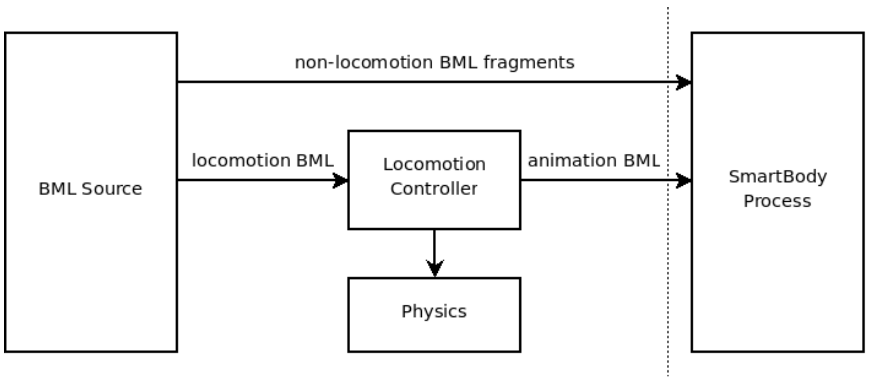


Figure 8: Locomotion filter

This type of "filter" implementation is in many ways less flexible than a full implementation of `<body>` locomotion requests within SmartBody. For example, it does not allow other SmartBody behaviours to synchronise to the locomotion behaviour. However it should be sufficient for the purposes of the project, allowing the virtual guide character to move around the world as it did before.

The remaining problem is that if the BML locomotion requests are filtered out entirely before reaching SmartBody, the virtual characters will appear to float along without any type of walk animation. This is a regression from the previous project. A potential solution is for the "locomotion filter" to generate an additional behaviour request that directs SmartBody to play a looping walk cycle animation, using the `<sbm:animation>` extension element. The walk cycle animation can be started when the character begins locomotion, and terminated when it reaches its target location.

The final point worthy of note is that implementing locomotion within the application rather than within SmartBody has the advantage of allowing tighter integration with PhysX. If locomotion were to be implemented in SmartBody, SmartBody would need sufficient information and capabilities to create a walk path through the world that did not intersect any obstacles. This could be achieved either by sending a full description of the world geometry, or by pre-calculating the exact path the character should take, with obstacles taken into account, and then passing that path to SmartBody piece by piece as a series of locomotion requests. Handling locomotion before the behaviour reaches SmartBody is thus a significant simplification.

Three BML extensions will be introduced to support locomotion (using the extension mechanism described in Section 4.1.5):

- The `<vgs:walk>` behaviour element will direct the character to walk to a given target location, optionally stopping a certain distance away. Once this behaviour request has been issued, the character will remain in the "walking" state until it reaches its destination, or until the walk request is cancelled via a blank `<vgs:walk>` element. If the target location moves while the character is still in the "walking" state, it will adjust its course accordingly.
- The `<vgs:walking>` element is a special "status request" element, allowing a BML client to query whether the character is currently in the walking state or not. The locomotion manager will reply to this query with a `<vgs:walking>` element if the character is in the walking state.
- The `<vgs:position>` element is another special "status request" element, which queries the current world position of a character. The locomotion manager will reply with a `<vgs:position>` element containing the character's current x, y and z coordinates as attributes.

An example of the `<vgs:walk>` behaviour element follows. It instructs the target character to walk towards the "ego" location, stopping when around 2.0 meters away:

```
<?xml version="1.0"?>
<bml>
  <vgs:walk target="ego" proximity="2.0"/>
</bml>
```

In addition to the extension elements, the locomotion controller will need to intercept the standard BML `<gaze>` request. If the character is directed to gaze toward a target that is behind them, the locomotion controller will be responsible for turning the whole body around, allowing the gaze to be carried out naturally.

5.2.3 Speech

As discussed in Section 4.2.1.3, SmartBody supports pre-recorded voice invoked via the `<speech>` element. There are some limitations, such as the lack of 3D audio mixing and the requirement that all recorded sound files be in uncompressed PCM format, which is a minor regression from the previous iteration of the project which supported OGG/Vorbis [18] compression.

One option to address this regression is to intercept `<speech>` behaviour elements in the same way as walk elements are intercepted to support locomotion. Speech could then be handled within the host application instead of by the SmartBody process. This would make the full capabilities of the existing system available, including real-time decompression of audio files and 3D audio mixing based on the relative position of the speaker and the user/listener.

The downside of this approach is that it does not allow other behaviour elements to synchronise to the speech, because SmartBody would not be aware of it. Unlike locomotion, where synchronisation is unlikely to be required, it is almost certain that other behaviours such as hand and facial gestures will need to be synchronised with key phrases in a character's speech.

A potential solution to this problem would be to eliminate the pre-SmartBody filter, and pass the speech request through as per normal, but then modify SmartBody to reflect the speech request back to the host application via new BoneBus commands. The host application would then be able to take care of the audio rendering, while SmartBody would still know about the speech and be able to synchronise other behaviours to it.

Although the reflection solution should work, it would require compatibility-breaking changes to SmartBody and the BoneBus protocol, and will therefore not be attempted as part of this project. The filtering solution is also unsuitable, as the unavailability of synchronisation would severely limit the utility of BML in the virtual guide application. Instead, speech will be handled directly by SmartBody, despite the requirement for uncompressed audio files.

5.2.4 Multiple BML-Driven Characters

The requirements set out in Section 3.3 call for the new system to support multiple characters controlled entirely via BML. This is a departure from the previous iteration of the system that supported only a single guide character. The new system must retain the guide character, now animated via BML, and add a mechanism for managing ancillary characters that can be used as "props" in the virtual museum.

Given these requirements and the process integration technique outlined in the previous sections, an architecture for creating and controlling characters must be created where the state for each character is split between the main application process and the SmartBody process.

In addition to accepting BML behaviour blocks as input, SmartBody also accepts non-BML commands that direct it to create new characters and destroy existing characters. For the sake of architectural simplicity and to follow the project requirements to the letter, the application process will need to define new BML extension elements for creating and destroying characters, and intercept these before they reach the SmartBody process, in much the same way as it will intercept locomotion requests. This will give it the opportunity to create its own data structures before SmartBody creates its character state. Once the host-side character state has been created, the non-BML character creation command can be emitted to SmartBody. SmartBody will create the character and send the appropriate notifications back over BoneBus. The host application will then need to match and link the new BoneBus character to its existing state. A similar process will apply when destroying a character.

Figure 9 illustrates the character control architecture:

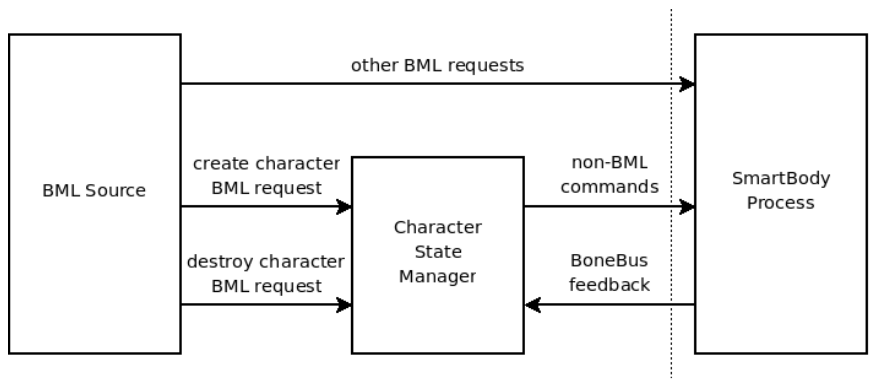


Figure 9: Character create and destroy architecture

Two new BML extension elements will need to be defined (using the extension mechanism described in Section 4.1.5) to support character creation and destruction: `<vgs:create>` and `<vgs:destroy>`:

- Unlike all other BML behaviour elements, `<vgs:create>` is not sent to a particular character. Instead it is sent to the system, directing it to create a new animated character with the given name and model. The character state manager will intercept this element, create the necessary local state, and then send the appropriate non-BML command to SmartBody. SmartBody will create its own representation of the character, which will be reflected in the BoneBus data it sends back to the application's character state manager.

- `<vgs:destroy>` on the other hand is sent to a specific character. The character state manager will intercept it, destroy the local character state, and then forward the appropriate non-BML command to SmartBody to destroy its representation of the character.

An example of the `<vgs:create>` element follows. It instructs the character state manager to create a new character using the “brad” model, positioned at the location named “cottage”:

```
<?xml version="1.0"?>
<vgs:create>
  <character model="brad"/>
  <position target="cottage"/>
</vgs:create>
```

5.2.5 Scriptable Behaviour

The requirements call for the ability to create scripts that are executed when the guide is presenting a particular feature of interest from the museum, and that are able to create new ancillary characters, as well as direct them to move, behave, speak and finally be destroyed. An example use case for this functionality could be if the guide were to present a blacksmith’s workstation to the visitor. While the guide speaks, a script could be executed that creates an authentic looking blacksmith character, hard at work using his tools. As the guide describes how the blacksmith works, he could mirror the description with appropriate animations. In theory he could even have a short vocal interchange with the guide and/or the visitor.

To meet this requirement, it will be possible to associate a script file with each feature of interest. Thanks to the design decision that all character control be expressed in BML, the script files will only need to contain a sequence of BML behaviour blocks, each annotated with the target character name and a delivery time: the time at which the behaviour block should be delivered to the behaviour realiser and executed.

These script files are very similar to SmartBody Sequence (SEQ) files, which are already supported within the SmartBody codebase. There are two key differences: Firstly, SmartBody SEQ files are line-based, and thus not proper XML documents. This limits the ability to expand the syntax of the files without breaking compatibility. The script files will be conformant XML documents, wrapping the behaviour elements in a parent element that specifies the target character name and delivery time. An example block is shown below:

```
<vgs:block actor="guide" time="5.0">
  <gaze id="b0" target="john"/>
  <head id="b1" type="NOD" stroke="b0:end"/>
</vgs:block>
```

The second difference is that the behaviour blocks will not be sent directly to SmartBody, but rather through the internal BML filters. This will allow them to create and destroy characters using the `<vgs:create>` and `<vgs:destroy>` elements, as well as make use of features such as locomotion.

Unfortunately SmartBody does not yet support synchronising behaviours between characters (or between behaviour blocks). This means that it will be difficult to create synchronised verbal exchanges between characters, or to synchronise the behaviour of one character (such as the blacksmith) to the speech of another (such as the guide). This is an unfortunate limitation, but it can be partially worked around using the timed delivery of behaviour blocks. This is not as flexible or intuitive as proper cross-character synchronisation, but it will have to do.

5.2.6 Subconscious Non-Verbal Behaviour

The requirements set out in Section 3.3 call for the system to automatically generate subconscious non-verbal behaviour for the characters it manages. These subconscious behaviours will fill an important gap between the behaviours explicitly requested by the existing guide-control algorithm and behaviours explicitly requested by script sequences (as described in the previous section). Automatic behaviour generation will free the site designer from having to script every last detail of the virtual characters.

Each character managed by the system will have a subconscious behaviour generator, which takes as input the character's position and orientation within the world, and the positions and orientations of both surrounding characters and the user. As an output it will generate BML behaviour blocks representing subconscious behaviour for the character in question.

The most important aspect of subconscious behaviour generation is control over a character's eyes. People typically do not stare directly at a fixed location for any extended period of time. Staring at a person in this way is likely to make them very uncomfortable, especially if there is no periodic blinking. Instead, real people blink their eyes regularly, and move their gaze from target to target over time. These behaviours can be implemented programatically.

When enabled, the automatic gaze control system will enumerate the objects of interest that are within a character's field of view, and switch their gaze from object to object at random intervals. The frequency that the character will gaze at a given object will depend on the distance to the object, whether the object is a person (real or virtual), and whether there is any speech active. Characters will prefer to gaze at other people rather than inanimate objects, especially while speaking. When gazing at the user, a character will gaze directly at the camera location, simulating eye contact. When gazing at each other, characters will target the others' eyeball (which is the standard implementation in SmartBody).

Aside from the user and the animated characters, the gaze control system will also include all named objects within the world, such as the features of interest at the museum.

The automatic gaze control system will be implemented as a BML filter, much as the locomotion and character creation/destruction extensions. A new attribute called "wander" will be added to the standard BML `<gaze>` element. If "wander" is not specified, the `<gaze>` request will be passed through to SmartBody unmodified, and the character will gaze continually at the specified target. If "wander" is included, it will specify the size of the field of view cone that shall be used to identify objects that the character can gaze at. Each random gaze switch will be implemented by generating a `<gaze>` request to SmartBody.

Note that other more pronounced behaviours, such as hand gestures and larger head gestures, are not generated automatically. These must be explicitly requested by the guide controller or scripted sequences, where they can be properly synchronised with parts of the dialogue.

5.3 Proposed Architecture

The proposed architecture is illustrated in Figure 10:

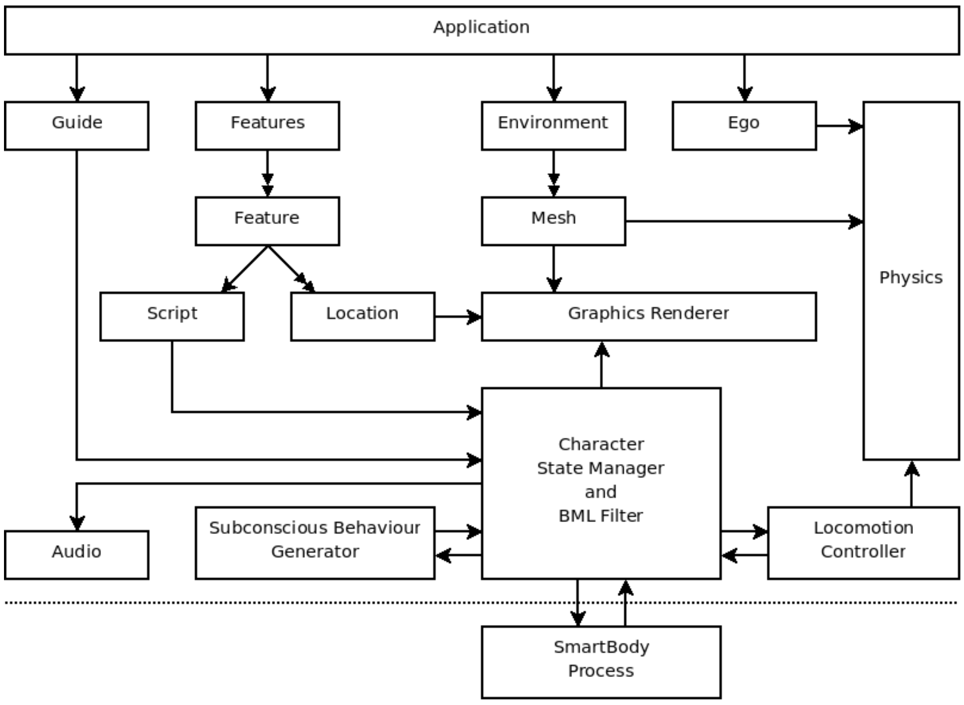


Figure 10: The proposed architecture

5.3.1 Changes

The new architecture has much in common with the previous architecture, illustrated in Figure 6. Specifically:

- The **Application** module remains responsible for coordinating the system through initialisation, runtime and termination. It has new subsystems to manage, but will otherwise operate in the same way.
- The **Environment**, **Ego**, **Physics** and **Audio Renderer** modules are unchanged.
- The **Features** module remains responsible for maintaining a database of the features of interest at the virtual museum. Whereas previously each feature of interest could be linked to a single recorded voice file, in the new architecture each feature can also be linked to a BML script file (see Section 5.2.5).
- The **Guide** module remains responsible for managing the virtual guide character that assists the user on their visit. Although its input and logic functionality have not changed, the output of the guide module will now be BML behaviour blocks instead of raw animation data.
- The **Graphics Renderer** module is largely unchanged. It still renders a 3D view of the environment meshes, the features of interest and the guide character. It must now also support rendering any number of independent characters in addition to the guide.
- The **Character** module has been replaced by a set of new modules comprising the behaviour subsystem: the **Character State Manager and BML Filter**, the **Subconscious Behaviour Generator**, the **Locomotion Controller** and of course the **SmartBody Process** and associated inter-process communication systems.

6 Implementation

This section describes the implementation of the proposed architecture, including the specific challenges that arose and how they were solved. Situations where the implementation diverged from the proposed architecture are also noted.

6.1 SmartBody Integration

The first significant implementation step was the integration of SmartBody as an external process responsible for managing the state of all animated characters.

6.1.1 Terminology

Throughout this thesis, the term "character" is used to refer to an instance of a virtual human. The form that character takes is referred to as the "model", such that several "characters" might share a common "model", and thus look the same. This terminology is consistent with that used within SmartBody and its associated literature.

The accompanying codebase however uses a slightly different nomenclature for its file names and class names, due to its heritage in the previous iteration of the virtual guide project. In the codebase, instances are known as "actors", and models are known as "character models".

This difference should be kept in mind when navigating the codebase.

6.1.2 BoneBus Protocol

As described in Section 5.2.1, the virtual guide system software and SmartBody run in separate processes. They communicate using the BoneBus protocol, which is in turn carried over TCP and UDP sockets. The original BoneBus protocol as developed by the SmartBody team supported one-way communication, transporting character creation and deletion notices and skeletal data from SmartBody to the renderer. The Panda BML Realiser team at the University of Reykjavik[7] extended the BoneBus protocol to support a back channel, allowing BML commands to be sent from the renderer to SmartBody.

Both the forward and back channels of the BoneBus protocol send packets of plain-text information over a TCP socket. For the original SmartBody-to-renderer packets, the packet delimiter is the semicolon character (;). For the renderer-to-SmartBody packets, the delimiter is three carriage return characters (\r\r\r in C notation).

During the implementation of the virtual guide system's BoneBus protocol transceiver module, it was discovered that the existing BoneBus library as used inside SmartBody suffered from a socket programming bug that could cause intermittent data loss and/or packet truncation corruption. This failure mode arose because the implementation treated the TCP socket as a packet-oriented connection when it is in fact a stream-oriented connection. Although the implementation was prepared for multiple packets to be received during one `recv()` call (thus the use of packet delimiters), it was not prepared for instances where only part of a packet was received during a call. If only the first part of a packet was received during one call, that truncated packet would be interpreted without all of its data, possibly causing incorrect results. The next `recv()` call would then retrieve the next part of the packet, which would fail in the interpreter because it did not start with a valid command.

It is likely that this failure mode did not occur very often during testing, as when both ends of a TCP socket connection reside on the same host, TCP `send()` and `recv()` calls often appear to operate in a packet-oriented manner. This behaviour is however not guaranteed by the TCP protocol at all, and in fact partial receives did occur often enough during the development of the virtual guide system that the problem in SmartBody had to be addressed.

A simple patch was applied to the SmartBody BoneBus library to enable the buffering of partially received packets until the full packet arrives.

6.1.3 Position Synchronisation

As the external SmartBody process is now responsible for realising all of the character animation in the virtual guide system, it needs to be supplied with sufficient information about the state of the virtual world to perform its calculations. The obvious parts of this information are the BML behaviour request elements that direct different characters to execute various behaviours, and the character management instructions that create and destroy characters as needed.

However this information alone is not enough. Some behaviour requests, such as the standard BML `<gaze>` element, direct a character to look at a particular named location within the world. To be able to execute this request, SmartBody needs to have up-to-date information about the location and orientation of all objects of interest. This includes both other animated characters, and inanimate objects that the characters might want to interact with.

Providing location and orientation information for the animated characters is trivial, as SmartBody is already aware of the existence of all characters, and naturally maintains data structures with their current state. There are a set of non-BML commands that can be used by the application to feed character base positions and orientations to SmartBody. The virtual guide system sends these commands whenever characters move or rotate.

Inanimate objects present more of a problem, as they exist only within the host application. SmartBody offers a solution in the form of "pawns". Pawns are proxy objects that can be created within the SmartBody process, to represent real objects in the host application process. Each pawn has a name, a position and an orientation, which can be updated using a new set of non-BML commands.

The virtual guide system creates SmartBody pawns for all feature points in the virtual museum. The pawns are named after the feature to which they belong, followed by the point type (activation, location or guide) and the point number. For example "wall_a1" would be the name of the first activation point for the feature named "wall". These names are arbitrary, but important, as they can be used within BML scripts to create complex animations where characters walk to, gaze at or gesture toward known locations within the world (see Section 5.2.5 for a discussion of the scripted sequence architecture).

In addition to the feature points, the software also registers a pawn to represent the user's current camera position. This allows animated characters to look the user in the eye.

6.1.4 Speech Issues

Section 5.2.3 described several different architectures that could be implemented to support speech, with deferring levels of involvement for SmartBody and the host application. The preferred solution was to simply forward all `<speech>` elements directly to SmartBody and allow it to render the pre-recorded speech, as this provided maximum flexibility in terms of synchronising facial and other animations with the voice recording.

Unfortunately this proposed architecture could not be implemented, as it did not meet a key implementation requirement: that the host application be able to determine when the speech track has completed. This requirement is essential because the virtual guide's control logic must be able to determine when various speeches are complete, so that it can then move on to the next state.

Normally when SmartBody is operating in a VHuman/VHMsg environment (see Section 4.2.2.1), every BML request that is processed is answered with a completion message that acknowledges the successful (or unsuccessful) execution of the behaviour. These acknowledgement messages could be used to determine when speech behaviour requests have completed. Unfortunately in the SmartBody process integration architecture that has been implemented for this project (see Section 5.2.1), VHuman/VHMsg is not used. Instead, BML requests are carried over a BoneBus connection, which does not support acknowledgements.

One potential solution to this problem would be to modify the BoneBus protocol and library to support the delivery of acknowledgement messages. This improvement could also have benefits in other areas. Some work toward this goal was undertaken, however it was aborted after other problems with the recorded audio speech system were discovered.

Upon closer examination, the internal audio system currently implemented within SmartBody appeared to be a placeholder implementation, without proper error checking or memory management. Uncompressed audio files were loaded and played, but never properly tracked or freed. The virtual guide system needs to support lengthy passages of dialog, which in uncompressed form would take significant amounts of memory. Failing to properly track and free these memory allocations would cause a significant memory leak and render the software unstable over any extended period of time (such as when in use at a museum).

As a result of this analysis, it was decided to implement recorded audio playback within the virtual guide process, and leave SmartBody out of the loop. This is a non-optimal architecture, as it leaves SmartBody with no way to synchronise other behaviours with the speech. However it avoids the memory leak problem, does not require changes to SmartBody or BoneBus, and in addition retains the support for compressed audio files and 3D audio mixing that were implemented in the previous iteration of the project.

The recorded audio system was implemented using the same BML filter architecture used for the locomotion controller, and proposed as one of the alternatives for speech in Section 5.2.3.

6.2 Character Rendering

In the previous iteration of the project, character rendering consisted of a relatively simple human model, deformed via playback of pre-authored keyframe-based skeletal animation sequences, with some support for blending between different animations[8]. The new implementation involves a much more detailed human model, and requires the renderer to accept raw bone translations and rotations from SmartBody rather than managing playback of its own animation sequences.

6.2.1 Character Model

One of the first steps to implementing a more detailed character animation system is to adopt a more detailed character model. SmartBody defines a standard human skeleton hierarchy (illustrated in Figure 11), with a standard set of bone names, but it does not require any particular mesh model to wrap round that skeleton.

Nonetheless the skeleton is relatively complex, with over 100 bones. Creating a character mesh that is detailed enough to support this skeleton is a job that would take considerable time, even for an experienced 3D artist. Instead this project will use a character model that has already been used with SmartBody, and is in fact shipped with the SmartBody distribution.

The character model known as “brad” is included as part of a SmartBody demo application, demonstrating an experimental connection between SmartBody and the open-source 3D engine OGRE[19]. Brad appears fairly generic, and is thus a reasonable candidate for the virtual guide in this application.

Brad appears only in OGRE binary format, so some conversion was necessary to extract the relevant mesh and skeleton data. The OGRE XMLConverter was first used to convert the binary format to an XML-based format, and then a custom converter was written in C++ (and included in the codebase) to convert the OGRE XML format to the XML format supported by the virtual guide system’s existing renderer.

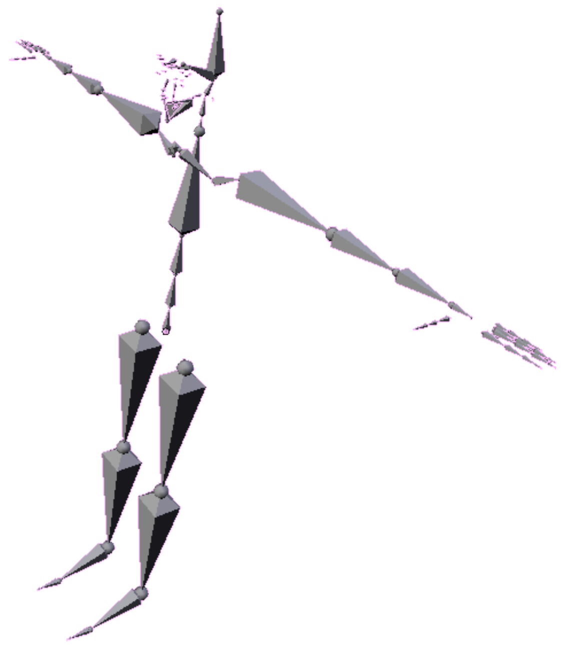


Figure 11: The standard SmartBody skeleton

6.2.2 Matrix Palette Skinning

As noted in the preceding section, the original character model employed in the first version of the virtual guide project was relatively simple, with only 16 bones. The new character model on the other hand uses the standard SmartBody skeleton, which has over 100 bones. This increase in skeletal complexity has certain implications for the rendering process.

Matrix Palette Skinning is an animation technique in which a static 3D mesh is attached to a skeleton by linking each vertex in the mesh with one or more bones in the skeleton[20]. As bones are moved according to animation parameters, the vertices they are linked to are also moved, deforming the mesh to follow the skeleton. This technique is popular for its speed and flexibility.

The first iteration of the project used matrix palette skinning to deform the character mesh. The algorithm was implemented as a vertex shader program, allowing it to run on the graphics hardware where available. Performing matrix palette skinning in hardware provides a significant performance boost, particularly with highly detailed meshes, as the CPU is freed of the task of transforming thousands of vertices and uploading the resulting data to the GPU for rendering. Instead of uploading new data for every vertex, it need only upload new data for every bone.

With the more complex character model, featuring both more vertices and more bones, hardware shader-based matrix palette skinning would in theory provide an even more significant performance boost than before. Unfortunately the large number of bones in the skeleton comprises too much constant data to be uploaded to a vertex shader program on common hardware. Each bone requires one 4x3 matrix, meaning that in practice each bone requires 3 constant vector (or "uniform") registers. The NVIDIA GeForce 7600 GPU used for testing supports only 256 registers, for a maximum of 85 bone matrices.

There are many potential solutions to this problem:

- The simplest solution is to require the use of newer graphics hardware which supports a greater number of uniform registers. The latest GPU designs support up to 1024 registers, which would allow skeletons of up to 341 bones. Although free to implement, this solution would restrict the application to all but the latest graphics hardware.

- Bone matrices could be communicated to the vertex shader via a different channel. Instead of using uniform registers, the matrix elements could be encoded into a floating point texture. Vertex-texture-fetch could then be used to access this information and reconstruct the matrices within the vertex shader. This method has the advantage of supporting almost unlimited numbers of bones (due to the massive texture sizes supported on modern hardware), however requires more work in the vertex shader. Furthermore the ability to access texture resources from a vertex shader is also a relatively modern feature, and the number of GPUs that support vertex-texture-fetch but do not support a large number of uniform registers is low.

- The mesh could be rendered in several passes, with each pass containing vertices that share a subset of the bones in the skeleton. Only the relevant bones would need to be uploaded for each pass, staying within the hardware limits. This method would require significant organisation on the software side to partition the mesh and coordinate multi-pass rendering, and would of course require multiple draw-calls for each character.

- A final solution is to simply implement matrix palette skinning in software. This places an additional calculation burden on the CPU to perform the vertex transformations, and a bandwidth burden transferring the bulky vertex data to the GPU for rendering.

In an ideal world, several of the above solutions would be employed. For hardware with sufficient resource limitations, skinning would occur entirely in the vertex shader. If that were impossible, a fallback to multi-pass rendering or software skinning would be employed.

For this application, however, the old vertex shader implementation was simply replaced by a new software implementation. The justification for this is that the application is likely to be GPU-bound, so shifting work from the GPU to the CPU should not cause a significant performance regression.

6.2.3 Walk Animation

The locomotion architecture proposed in Section 5.2.2 calls for a walk-cycle animation that can be played back by SmartBody while the host application takes care of the character movement and physics constraints.

The movement side of the architecture has been successfully implemented, by adapting the PhysX-assisted locomotion implementation from the previous iteration of the project to respond to the new `<vgs:walk>` BML extension element. The character calculates the heading from its current position to the walk target, turns its body at an appropriate speed, and moves forward until the target is reached. While the walk process is underway, a `<body posture="Walk"/>` BML request is generated and sent to SmartBody, directing it to play a looped walk animation.

Unfortunately the SmartBody distribution does not include a walk-cycle animation that can be used for this purpose.

To resolve this, an original walk-cycle animation has been authored using the open-source 3D modelling software Blender[21]. The Blender output was then converted to SmartBody's SKM motion file format using a custom file format converter developed in C++ (and included in the codebase).

6.2.4 Facial Animation

Unfortunately with the exception of the eyeballs, the current implementation does not include any facial animation. This results in a somewhat blank stare from virtual characters when they are speaking to the user, which is quite unrealistic. It is also a slight regression from the previous iteration of the virtual guide system, which at least "flapped" the jaw of the character during speech.

The reasons for this limitation were not identified until the implementation phase of the project. Although SmartBody has extensive support for facial expression animation, which can be synchronised with speech in various ways, it currently outputs this animation as vismes rather than as bone positions and orientations. The character renderer must therefore be able to further interpret these vismes into facial animation, for example through the use of a set of morph targets included with the character model. The character renderer implemented for this project does not support morph targets, and the character model that has been used does not include any morph targets. Together this makes facial animation impossible.

The SmartBody developers plan to transition from visme-based facial animation to bone-based face control. When this transition is complete, the existing character skeleton (which includes facial bones) and the existing matrix palette skinning approach should be able to support facial animation. According to the SmartBody roadmap, this work is scheduled for the short term [17].

6.3 Software

The software developed in the previous iteration of the project has been subject to major additions and refactoring to support the new architecture.

6.3.1 Refactoring

The previous iteration of the virtual guide system, as described in Section 5.1 and more fully in [8] consisted of an "application" program and an entirely separate "editor" program. This approach made sense, as the application program that would be used by the museum visitor did not need all of the extra functionality of the editor, which would be used more occasionally by those responsible for setting up the virtual museum.

The disadvantage of this discrete approach was that there was a lot of duplicated code in the two programs. When improvements were to be made, similar but not quite identical code would often need to be committed to both programs, creating an unnecessary maintenance headache.

It was desirable that this situation be remedied before beginning work on the new virtual guide system. The old codebase was therefore refactored into three parts: the "common" part, shared by both programs, the application-specific part and the editor-specific part.

This refactoring provided a unique opportunity to address another shortcoming of the original implementation: The original editor software was only able to edit the virtual world, so if the museum manager wanted to preview his or her new creation as the visitor would experience it, they would need to launch the application program separately, wait for it to load, then navigate from the start to the area of interest, etc. While possible, this was hardly efficient.

The new codebase has been refactored such that everything required to run the virtual museum, including the BML/SmartBody-based animation, is contained within the common classes. This allows both the application program and the editor program to run the museum as the visitor would experience it, with the same graphics, animation, sound, control logic, etc. The editor now has a “run mode” that can be invoked with a mouse click.

6.3.2 Application

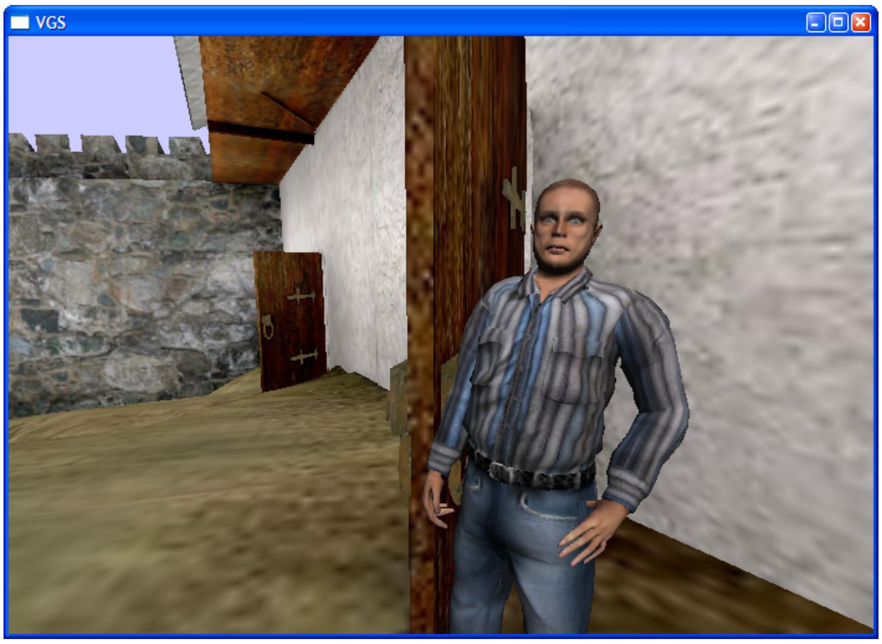


Figure 12: The new SmartBody-animated virtual guide in action

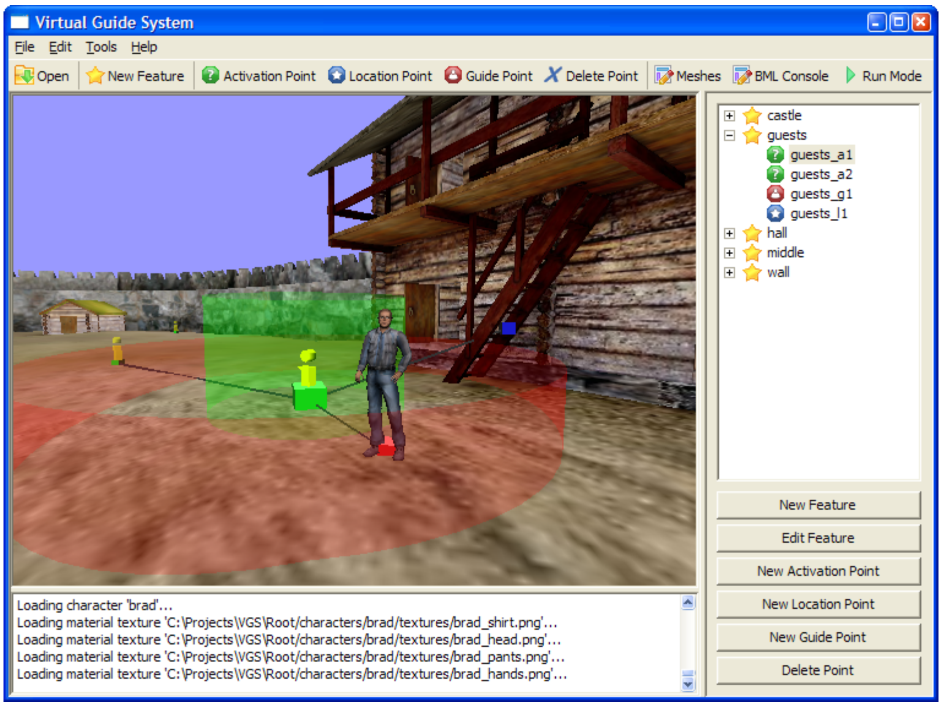
Although the codebase has been refactored to move all of the functionality required to “run” the virtual museum into the common classes, the application program still contains a significant amount of unique code that is not part of the editor.

This is required because the application still supports multiple input and output configurations, which is a requirement inherited from the previous iteration of the virtual guide system. The application module can be built on top of the Simple DirectMedia Layer (SDL)[22], to provide a simple full-screen output with mouse and keyboard input, or it can be built on top of VRJuggler[16], to support a variety of virtual reality input modes such as head tracking, and a variety of output modes such as stereo rendering.

In the default SDL-based build, the application uses a simple input model where movement of the mouse allows the user to look around, while the left and right mouse buttons allow them to move forward and backward, respectively.

6.3.3 Editor

The editor has been significantly improved since the previous iteration of the virtual guide system. As before, it is built upon the wxWidgets[23] GUI framework, however it now also includes all of the functionality required to not only edit the virtual museum, but also to run it in-place with animation and audio.



```
Loading character 'brad'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_shirt.png'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_head.png'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_pants.png'...
Loading material texture 'C:\Projects\VGS\Root/characters/brad/textures/brad_hands.png'...
```

Figure 13: The new virtual museum editor

Perhaps the most visible change is that it now provides a full 3D view of the virtual museum, rather than the top-down 2D view from the previous iteration of the system. This change was made necessary by the improvement in the quality of animation – now that the user can see where characters are looking, it has become more important to position objects of interest accurately in the virtual world. The height of an object is critical if a character is to look at it – if the target height is incorrect, the character's eyes will be noticeably misdirected.

The switch to 3D was eased by the refactoring described in Section 6.3.1, which gave the editor the same basic 3D engine as the application. There were still several considerations specific to the editor that needed to be addressed:

The first consideration was the 3D representation of features of interest and their associated points. As before, each feature of interest at the museum can have one or more activation points, where the user can stand to activate the feature's narration, plus one or more guide points, where the virtual guide can stand while speaking, plus one or more location points, which the virtual guide can look at and point to. In 2D, these points were represented by small coloured circles. This has been extended to 3D, where the points are represented by small coloured cubes. The currently selected cube spins slowly on its Y axis.

Another consideration was how to select points using the mouse. In 2D this was achieved by simply finding the closest point to where the mouse was clicked in 2D Cartesian space. For the new 3D view, a ray-cube intersection test was implemented. Although this test is sufficient to determine the closest cube that the user's selection "ray" hits, it does not account for other geometry, such as environment meshes, that may be in the way. To solve this problem, the value of the depth buffer for the pixel that the user clicks is read back, and compared to the depth of the closest intersecting cube. If the depth buffer value is less, some other static geometry must be occluding and there is no cube hit.

Once an activation, location or guide point has been selected, there needs to be a way to move it around within the world using a "click-and-drag" metaphor. Again with 2D this was very simple, as the selected point could simply be moved to the mouse cursor position. In 3D the point has three degrees of freedom, while the user's mouse pointer has only two. This problem has been solved in many different ways in a variety of 3D software applications, and in this case the best solution was to read the depth buffer at the mouse location and project a ray to that depth. With this technique, the user can position a point on top of any visible surface in the virtual world with ease. The disadvantage is that it is impossible to position points floating in empty space. However for the activation, location and guide points this is never necessary.

Activation points have some additional properties beyond simple position. They also have an inner and an outer radius. The user must come within the inner radius of the point to activate the feature narration, and must exit the outer radius to abort the narration. In 3D these radii are rendered as open semi-transparent cylinders, with green for the inner radius and red for the outer. This choice of colour and shape provides a fairly intuitive view of the positive inner region of space and the negative outer region. To edit the radii, the user can hold the Ctrl key and click on the activation point. Dragging the mouse then scales either the inner or outer radius, depending on the state of the shift key. This could probably be more intuitive.

Placing new points in the world used to be a trivial operation. After selecting an "Add Point" command from the menu, the new point was placed in the centre of the current 2D view. Although the same could be implemented in 3D, it is much less likely that a projected ray from the centre of the view would be where the user wants the new point. Instead, the 3D view has the concept of the current "cursor location". This is a familiar interactive device from text editors, where the current cursor location is usually visible as a blinking vertical line, and can be moved by clicking anywhere in the document. The same metaphor is used for the 3D cursor – it appears as a blinking yellow dot, and can be repositioned by clicking anywhere in the world. New points are added at the current 3D cursor position.

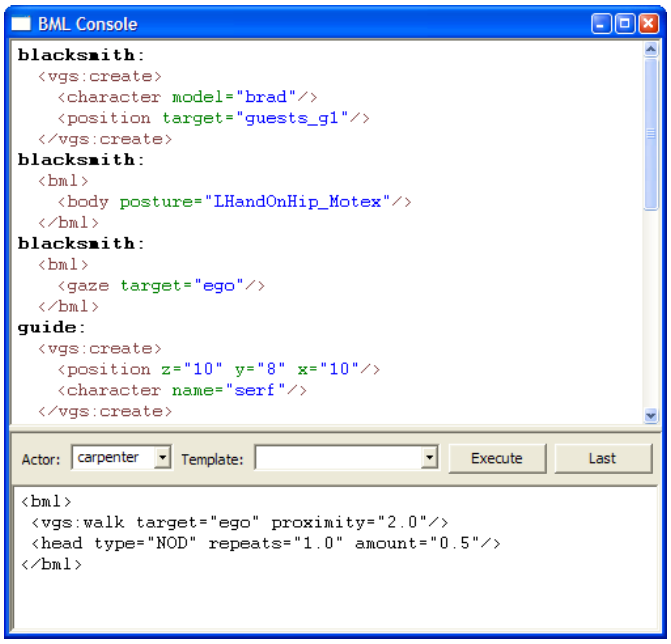
Camera movement in the 3D view is also a consideration. In 2D, the user could pan around by clicking and dragging with the right mouse button. In 3D there is both camera orientation and position to consider. The right mouse button is still used for camera movement, but now clicking and dragging causes the camera's orientation to be changed, allowing the user to look around. If the left mouse button is held in addition to the right, the camera's position is changed, allowing the user to move. Although the prospect of holding down two mouse buttons seems daunting at first, this turns out to be a very intuitive and convenient method of navigation. Camera movement is not subject to physical constraints, so the camera can pass through solid geometry.

Although the transition from 2D to 3D provides a richer, "what you see is what you get" view of the virtual world, it does make it slightly harder to get a simple visual overview of what is in the scene. To assist with this, a simple scene graph tree has been added to the right-hand side of the editor window. This tree displays the name of every feature of interest in the museum, and beneath each feature the name of each associated activation, location and guide point. When a new point is selected in the 3D view, the selection is mirrored in the tree (and vice-versa). This makes it easy to pick points by name without hunting around in 3D. An added benefit is that the tree displays the name of every point, which can be useful when developing BML scripts (see Sections 6.3.4 and 6.3.5).

6.3.4 BML Console

The core improvement in this iteration of the virtual guide project is the switch from simple keyframe-based animation to animation expressed in BML and realised by SmartBody, and the ability to have multiple virtual characters operating concurrently. The BML Console is a new addition to the editor program that makes working with BML easy.

The BML Console can be opened from the menu or toolbar once a virtual museum project has been opened and a SmartBody process launched. It has two components: the output panel and the input panel.



```
blacksmith:
  <vgs:create>
    <character model="brad"/>
    <position target="guests_g1"/>
  </vgs:create>
blacksmith:
  <bml>
    <body posture="LHandOnHip_Motex"/>
  </bml>
blacksmith:
  <bml>
    <gaze target="ego"/>
  </bml>
guide:
  <vgs:create>
    <position z="10" y="8" x="10"/>
    <character name="serf"/>
  </vgs:create>
```

```
<bml>
 <vgs:walk target="ego" proximity="2.0"/>
 <head type="NOD" repeats="1.0" amount="0.5"/>
</bml>
```

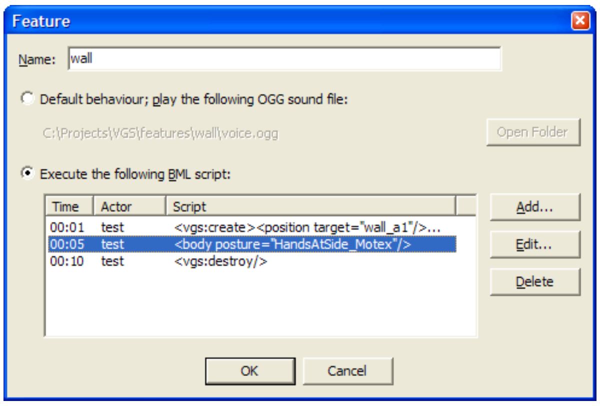
Figure 14: The new BML Console window

The output panel appears at the top of the BML Console window, and shows a live log of every BML request that is generated, either manually by the user or automatically by the virtual guide's logic systems in "run mode". Each BML request is displayed in syntax-coloured XML, preceded by a heading that shows the name of the character to which the request is being directed. The BML log is useful for identifying which behaviours are being requested and when, so that appropriate changes and improvements to the animation can be made.

The input panel appears at the bottom of the BML Console window. This component allows the user to type in or paste a BML request and send it to a character for immediate execution. A drop-down list of active character names is available for quick access, or a new character name can be typed in. In addition to entering BML requests manually, there is a selection of templates for commonly used behaviour request elements, including both standard BML elements and extension elements defined by the virtual guide system.

6.3.5 Script Editor

In the previous iteration of the virtual guide project, the only possible action when the user activated a feature of interest was to have the virtual guide look at them and speak a pre-recorded voice track, which would typically contain a narrative describing the feature. The new implementation retains



Feature

Name: wall

[ ] Default behaviour; play the following OGG sound file:

C:\Projects\VGS\features\wall\voice.ogg

[✓] Execute the following BML script:

Time	Actor	Script
00:01	test	<vgs:create><position target="wall_a1"/>...
00:05	test	<body posture="HandsAtSide_Motex"/>
00:10	test	<vgs:destroy/>

Figure 15: The new BML script editor

this default behaviour, but also allows scripted sequences which have complete control over the guide character's behaviour and vocalisations, and can even create and destroy new fully animated characters. The scripting architecture was described in detail in Section 5.2.5.

Script files are human-readable XML files containing blocks of BML, and can thus be edited manually with any text editor or XML editor. To ease this process, the editor program includes a built-in script editor that can be accessed by selecting a feature and choosing "Edit Feature" from the menu.

The built-in script editor displays a list of BML blocks, showing for each block the execution start time, the target character name, and the first part of the behaviour request. Blocks can be added, edited and deleted.

6.3.6 Known Issues

While the software works well, and confirms the viability of the proposed architecture, it is not perfect. In particular there are a number of known issues that should be noted:

- **Floating characters** – The animated characters often appear to "float" over the ground. If a character is standing still, its feet should be locked to fixed locations on the ground, allowing the rest of the body to sway gently around that position. Instead, it appears that the centre of the character (around the hips) is locked to a fixed location in space, while the rest of the body – including the feet – sway around that point. This even allows the feet to slide around on the ground, which is unrealistic. It is likely that this phenomenon arises due to the hierarchy of the skeleton, where the "root bone", and thus the character's origin, is at the base of the spine. The animations that are applied to the skeleton should of course compensate for this by moving and rotating that root bone as appropriate to keep the feet locked in place, but for some reason this does not happen. The Panda BML Realiser project noticed a similar effect in their work, and speculated that it may be a bug within SmartBody[24].

- **No facial animation** – Perhaps the most disconcerting issue is the lack of facial animation when characters speak. The problems behind this issue have been discussed at length in Section 6.1.4.

- **Orphan SmartBody processes** – Both the application program and the editor program are able to launch a SmartBody process automatically, and accept the incoming BoneBus connection from that newly launched process. Unfortunately they do not terminate the process they launched upon exit, so the user must close the SmartBody window manually. In a production environment this would need to be addressed.

- **Erratic animation** – Following from the previous known issue, if more than one SmartBody process is running, very erratic animation may be observed. This occurs because part of the BoneBus protocol uses UDP on a fixed port to communicate. UDP is a connectionless protocol, and it is possible for more than one SmartBody process to be sending animation data to the virtual guide software concurrently. These multiple sets of animation data will not be in sync, and will result in jittery erratic movements.

- **Unable to run more than one instance** – Although it is unlikely that anyone would want to run more than once instance of the application, or more than once instance of the editor concurrently, it is not inconceivable that someone may want to run one editor and one application concurrently. This is currently not possible, because of the fixed TCP and UDP ports used for BoneBus communication. It would be possible to remove this limitation, and eliminate the previous known issue by implementing dynamically allocated port numbers that are communicated to the SmartBody process when launching.

7 Evaluation

This section evaluates the effectiveness of the new virtual guide system architecture and the prototype that has been developed in this masters thesis, as well as the viability of SmartBody and the Behavioural Markup Language as tools for character animation in virtual heritage applications.

7.1 Behaviour Markup Language

The growing interest in realistic virtual human agents has seen the development of a significant number of different behaviour expression languages and behaviour realisation frameworks, as evidenced by Section 2.1. If this trend of inventing new languages continues, however, interoperability between independently developed animation implementations will remain difficult if not impossible.

A standard behaviour expression language is highly desirable, and the Behavioural Markup Language appears to be well positioned to fill this role. The current version of the language specification (as used in this project) is workable, and the upcoming version 1.0 takes further steps in the right direction, removing internal inconsistencies and improving expressivity.

Virtual heritage applications, like a vast number of other virtual reality systems, can benefit greatly from standardised frameworks and expression languages. A standard behaviour expression language like BML, combined with an off-the-shelf behaviour realiser component like SmartBody, can free these applications from having to re-implement character animation all over again. Instead they can express the desired behaviour in a standard way, and leave the implementation details to the the realiser. In this way, applications such as the virtual guide system can enjoy the latest advances in behaviour realisation and character animation research without having to follow and re-implement them.

In the future, applications requiring a virtual human agent may be able to abstract themselves even further away from the details of animation, by expressing high-level functional intentions instead of low-level behaviours. The Functional Markup Language is an effort in this direction [1].

Overall the Behavioural Markup Language appears to have a bright future, and it will be exciting to see how it evolves as its adoption grows.

7.2 SmartBody

SmartBody is perhaps the best known and most functional BML realiser currently available for use. Although it is already of considerable size and complexity, the project is still in its early days, as the developers experiment and learn how best to architect a flexible and powerful behaviour realiser. The codebase is certainly not at production quality at this point, nor does it claim to be. It is strictly a research project that will doubtless undergo ongoing improvements and even major restructuring efforts as it moves forward.

In its current state, SmartBody is rather difficult to use for anything other than experimentation. A lot of glue code is needed to work around missing features and to connect its inputs and outputs with the application program. This situation is likely to improve rapidly over time, as more applications seek to incorporate it as their behaviour realiser.

Perhaps the most urgent missing feature right now is the lack of support for any kind of locomotion. Both the current pre-1.0 version and the upcoming 1.0 version of the BML specification define behaviour elements that describe locomotion, but these are not yet supported. Without built-in locomotion, characters must either be confined to a spot, or locomotion must be implemented external to SmartBody, as has been successfully achieved in this virtual guide project.

Despite its current limitations, SmartBody is already a very powerful solution for character animation. Its parametric approach permits much more flexibility than keyframe-based systems. This increase in animation quality and flexibility is readily visible when comparing the new virtual guide system with the old non-SmartBody implementation. The most noticeable change is that characters can now make true eye contact with the user and with each other, which is a tremendously important social cue.

This project has demonstrated that, with some workarounds, it is possible to use SmartBody in its current form in virtual heritage applications. As more and more BML behaviour types are supported, the possibilities for rich character animation will continue to improve. At that point it may no longer make sense to attempt to re-implement character animation, given that a functional off-the-shelf component like SmartBody is readily available.

7.3 Virtual Guide Architecture

The starting point for this project was the first iteration of the virtual guide architecture, developed by the same author in a previous report[8]. The original architecture already provided a flexible system for creating and running interactive virtual museums with a walking, talking guide character able to assist users on their visit.

With the integration of BML and SmartBody, the quality of character animation has been dramatically increased. The virtual guide appears more detailed and realistic, and is able to look the user in the eye and make expressive gestures as he or she speaks. In addition, the system has become much more flexible, allowing the museum designer to create scripted sequences involving multiple animated characters interacting with each other, the environment and the user.

These changes required a fundamental redesign of the way characters were managed in the virtual guide system architecture. Character state is now shared between the host application and an external SmartBody process, with all of the inter-process communication that entails. The old method of controlling characters with a limited set of API calls has been replaced by a flexible and extensible BML processing pipeline, where all changes originate as fragments of XML. This not only allows new character animation capabilities to be introduced seamlessly, but also makes the playback and recording of behaviour scripts a trivial feature. Finally, by decoupling the core application from the behaviour realiser, the virtual guide system will now immediately benefit from future improvements in SmartBody without itself needing to be upgraded.

In addition to the integration of BML and SmartBody, other minor work has been performed on the virtual guide system codebase. The codebase has been refactored into a set of common classes, and two applications that depend on those classes. These changes allow both the runtime application program and the editor program to present the same 3D view of the virtual museum, offering a "what you see is what you get" editing metaphor. This simplifies the process of editing, and allows the museum designer to preview their work without switching programs.

Overall, the new virtual guide system architecture with BML and SmartBody not only offers higher quality animation, but is also significantly more flexible than its predecessor.

7.4 Future Work

The new virtual guide system works well enough for many simple deployment scenarios, however it is far from perfect. The following areas have been identified for future work:

- The current speech implementation leaves a lot to be desired. Due to the problems outlined in Section 6.1.4, speech requests are intercepted and handled internally by the application, leaving SmartBody unable to schedule synchronised facial animation. This, in combination with the next item, leave characters' faces blank and expressionless when they speak. If the application were modified to use the VHMsg bus (see Section 4.2.2.1) instead of BoneBus (see Section 4.2.2.2), SmartBody's own speech implementation could be utilised. It may however actually be simpler to make the necessary changes to SmartBody to deliver "speech complete" messages over BoneBus, avoiding the need to switch protocols.

- The current 3D renderer does not support vismes for facial animation, while SmartBody currently requires them for its facial animation implementation. As with the previous item, this leaves characters' faces blank and expressionless. In the short term, visme support could be added to the 3D renderer, perhaps using morph targets. In the longer term, SmartBody plans to transition to the use of bones for facial animation, which will solve this problem automatically (see Section 6.2.4).

- The integration with BML and SmartBody allows a much greater range of character animation than was previously available. The virtual guide's control logic has been upgraded to control the guide using BML fragments instead of the old API calls, and it has been expanded to take advantage of some of the new animation capabilities. There is now a great opportunity to further expand the control logic to take full advantage of SmartBody's current capabilities. The guide character could for example assume multiple different postures depending on how long it has been waiting for the user, or new hand gestures could be added to help the guide point at parts of the environment.

- The virtual guide architecture has been designed to support both virtual reality configurations and augmented reality configurations. In virtual reality configurations, the user's entire view is rendered, and their position and orientation in the virtual world are simulated using PhysX. In augmented reality configurations, rendered elements are super-imposed on top of a live camera feed, and the user's position and orientation are collected from external sensors such as a GPS receiver and an electronic compass. The current implementation of the architecture has only preliminary support for augmented reality configurations. Full support could be added by coding interfaces to head-mounted cameras, position sensors and orientation sensors.

- Beyond these major items, there are several other known issues with the current implementation identified in Section 6.3.6 that could be corrected.

8 Conclusion

This masters thesis began by proposing that multimodal behaviour generation frameworks are an appropriate way to increase the believability of characters in virtual heritage applications.

After evaluating several options, a combination of the Behavioural Markup Language (BML), and the open-source BML realiser SmartBody, were selected for integration into a prototype virtual heritage application. BML is the behaviour specification language of the SAIBA framework, which originated at the University of Reykjavik. SmartBody originated at the Institute for Creative Technologies at the University of Southern California.

A previous virtual heritage application developed by the author at the Norwegian University of Science and Technology was used as a starting point. The architecture of this original application already provided a flexible system for creating and running interactive virtual museums with a walking, talking guide character able to assist users on their visit. Its character animation capabilities were, however, limited.

A set of goals and requirements for the new and improved virtual museum guide project were then identified. With these targets in mind, a new software architecture was designed, based upon the architecture of the previous application, but extended and modified in order to integrate BML and SmartBody. This new architecture was then implemented and evaluated.

The integration of BML and SmartBody has resulted in a dramatic improvement in the quality of character animation in the application. The virtual guide now appears more detailed and realistic than it did before, specifically because it is now able to adjust its posture in response to its environment, making eye contact with the visitor and performing various hand gestures. In addition, the museum designer is now able to use a greatly improved editor program to create scripted behaviour sequences involving multiple animated characters interacting with each other, the environment and the visitor.

The revised virtual museum guide architecture developed and implemented in this thesis is more powerful and flexible compared to the previous version of the application. These improvements have been achieved through the successful integration of selected parts of the SAIBA multimodal behaviour generation framework. This confirms that multimodal behaviour generation frameworks are an appropriate and viable way to increase the believability of characters in virtual heritage applications.

References

[1] Hannes Högni Vilhjálmsson. *Representing Communicative Function and Behavior in Multimodal Communication.* Springer-Verlag, Berlin, Heidelberg, 2009.

[2] H. Vilhjálmson, N. Cantelmo, J. Cassell, N.E. Chafai, M. Kipp, S. Kopp, M. Mancini, S.C. Marsella, A.N. Marshall, C. Pelachaud, Z.M. Ruttkay, K. Thorisson, H. van Welbergen, and R.J. van der Werf. The behavior markup language: Recent developments and challenges. In C. Pelachaud, J-C. Martin, E. André, G. Collet, K. Karpouzis, and D. Pelé, editors, *Proceedings of the 7th International Conference on Intelligent Virtual Agents*, volume 4722 of *Lecture Notes in Computer Science*, pages 99–120, Berlin, 2007. Springer. ISBN=978-3-540-74996-7, ISSN=0302-9743.

[3] Dr. Stefan Kopp. Max and ace, April 2009. `http://www.techfak.uni-bielefeld.de/ skopp/max.html`.

[4] A. Kranstedt, S. Kopp, and I. Wachsmuth. The behavior markup language: Recent developments and challenges. In *Proceedings of the AAMAS Workshop on "Embodied conversational agents Let's specify and evaluate them"*, 2002.

[5] Edinburgh Language Technology Group. Magicster project pages, April 2009. `http://www.ltg.ed.ac.uk/magicster/`.

[6] Berardina De Carolis, Catherine Pelachaud, Isabella Poggi, and Mark Steedman. Apml, a mark-up language for believable behavior generation. In *Proceedings of AAMAS 2002 Workshop on "Embodied Conversational Agents: Let's Specify and Compare Them*, 2002.

[7] Bjarni Pór Árnason and Ægir Porsteinsson. Bml realizer: An open source bml animation toolkit. Master's thesis, Reykjavik University, 2008.

[8] Michael Stokes. An augmented reality virtual guide for sverresborg museum. Fordypning project at NTNU, 2008.

[9] Jorge Ordóñes Serrano. Virtual guide for a virtual heritage environment. Master's thesis, Norwegian University of Science and Technology, 2004.

[10] Eleonora Trumello, Antonella Santangelo, Antonio Gentile, and Salvatore Gaglio. A multimodal guide for virtual 3d models of cultural heritage artifacts. In *CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 735–740, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Stefan Kopp, Brigitte Krenn, Stacy Marsella, Andrew N. Marshall, Catherine Pelachaud, Hannes Pirker, Kristinn R. Thórisson, and Hannes Vilhjálmsson. Towards a common framework for multimodal generation: The behavior markup language. In *Proceedings of the 6th International Conference in Intelligent Virtual Agents, 2006*, pages 205–217, 2006.

[12] MindMakers.org. Behavior markup language (bml), April 2009. `http://wiki.mindmakers.org/projects:BML:main`.

[13] Marcus Thiebaux, Stacy Marsella, Andrew N. Marshall, and Marcelo Kallmann. Smartbody: behavior realization for embodied conversational agents. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 151–158, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[14] University of Southern California. Smartbody bml, May 2009. `http://smartbody.wiki.sourceforge.net/SmartBody+BML`.

[15] NVIDIA Corporation. Physx, October 2008. `http://developer.nvidia.com/object/physx.html`.

[16] Infiniscape Corporation. Vrjuggler, October 2008. `http://www.vrjuggler.org/`.

[17] Institute for Creative Technologies at University of Southern California. Smartbody roadmap, May 2009. `http://smartbody.wiki.sourceforge.net/Roadmap`.

[18] Xiph Foundation. Ogg vorbis, October 2008. `http://www.vorbis.com/`.

[19] Torus Knot Software Ltd. Ogre 3d, October 2008. `http://www.ogre3d.org/`.

[20] Jason Weber. Run-time skin deformation. In *Proceedings of Game Developers Conference, 2000*. Intel Architecture Labs, 2000.

[21] Blender Foundation. Blender, October 2008. `http://www.blender.org/`.

[22] SDL Community. Simple directmedia layer, October 2008. `http://www.libsdl.org/`.

[23] wxWidgets Community. wxwidgets, October 2008. `http://www.wxwidgets.org/`.

[24] Center for Analysis and Design of Intelligent Agents Reykjavik University. Bml realizer - pandabmlr - known issues, May 2009. `http://cadia.ru.is/wiki/public:bmlr:pandabmlr`.