A System for Using 3d Animations in Live Performances

University of Oulu
Department of Information Processing Sciences
Toni Alatalo
A System for Using 3d Animations
in Live Performances

Abstract

This master's thesis addresses the problem of using computer based systems for visualisations in live performances. A prototype system for animation control revealed points of views in the areas in controlling multiple entities in many ways at the same time, and in mapping controls to actions.

In this case, the constructed system was aimed at controlling 3d character animations and implemented in Python using the API provided by an interactive 3d graphics suite called Blender. The first real use of the solution was in a music/dance event during the Oulu music video festival in August 2003, and this study is limited to that early phase of development with discussions about future challenges and possibilities.

The system basically worked, proving the central model — featuring untypical so-called event objects, which are used to implement all possible actions in a scene — viable. However, in the evaluation major restructuring is proposed. Also, the prototyping brought up severe limitations with the selected engine, which has later resulted to both fixes in that engine and use of other engines. As a conceptual result, the use of high-level variables such as the amount of *difference* in a scene is identified as a powerful tool, explored with the real-time controlled variance in the animation of so-called clones in this system, and with possibly fruitful theoretical underpinnings found in gestalt theory.

Contents

1 Introduction — computer based systems for live performances . . . 5
2 The Research Method Used — Constructive Research / Design Science . . . 7
3 Related Work — Systems for Live Performances and Character Animation . . . 10
3.1 Software Applications for Visual Live Performances . . . 10
3.2 Character animation . . . 12
4 Requirements for the System . . . 13
4.1 Locus of Control . . . 14
4.2 Requirements for 3d character animation . . . 15
4.3 Requirements set by the operating environment . . . 15
4.4 Technical requirements . . . 17
4.5 Features wish-list . . . 17
4.6 Selected primary requirements . . . 17
5 The Design and Implementation . . . 17
5.1 Selecting the 3d engine . . . 18
5.2 Blender and its game engine . . . 19
5.3 General architecture . . . 20
5.4 The Class Library . . . 21
5.4.1 High-level Events map Controls to Actions . . . 21
5.4.2 Movers in Scenes . . . 24
5.4.3 The Animation System — working around GameBlender peculiarities . . . 28
5.4.4 Clones . . . 29
5.4.5 Direct manipulation of bones . . . 31
5.4.6 Controllers . . . 31
5.4.7 Time . . . 31
5.4.8 Camera control . . . 32
5.4.9 Missing elements . . . 34
5.5 A setup file . . . 35
6 Evaluation . . . 36
6.1 Model / system (in itself & compared to others) . . . 36
6.1.1 Comparison of the event system with others . . . 37
6.1.2 Cognitive, user centered and authoring issues . . . 38
6.2 Usage procedures . . . 41
6.2.1 Having the Data in Blender . . . 42

6.2.2 Configuring Blender objects for Mover 42
6.2.3 Making the setup 43
6.3 Use of the prototype 44
6.3.1 VJing In Time Tunnel X 44
6.3.2 Peer Review: Perceptions of Fellow VJ Application Developers 45
6.3.3 A remark from a Puppet-maker 45
6.4 Use for non-performance purposes in the Wireless Airguitar project 46
6.5 Component re-use 46
6.6 Review of the research questions 47
7 Future Development 48
7.1 Refactoring the core 48
7.2 Different possible directions with various technologies 49
7.2.1 Authoring for non-programmers via Blender-integration ('BlenderVJ') . . . 49
7.2.2 Good programmability of high-level features with Soya ('Play') 50
7.2.3 Distributed system using VOS and/or Twisted ('DanceWorld') 51
7.3 Strategies for the interplay of the different areas 52
8 Conclusions 53

1 Introduction — computer based systems for live performances

“It will work out somehow, it’s theatre!”

—(quote from Shakespear in love, as explained by Joel Ryan of STEIM.)

Live events — real time action, where what happens in the moment is what counts — can be seen as quite remote and foreign to the areas where use of complex computer systems usually takes place, such as office work. Instead of preparing documents for future uses, the live use must produce meaningful symbols (or symbol manipulations) instantly. One major area where computer based systems are used in real-time are controlling systems, such as aviation traffic conrol which is essentially coordination, or machines in industrial production. Here the focus is, instead, on presentation systems, which are used to control a show for an audience.

Surely, the problem of presentations is also addressed by the common production suites, which provide tools for preparing and showing what are commonly referred as digital “slides”. These may include rich media but the act of performing, presenting them, is typically reduced to proceeding along a strict sequence, i.e. clicking a button to get the next slide, or in a more fine-grained way, to the next part of the current slide. Even in an interesting recent tool for interactive presentations, where the ’slides’ are programmed and can hence include complex behaviour as e.g. dynamic diagrams, the presenting is about simple proceeding in a sequence (Zongker & Salesin 2003)[1]. A reasonable motivation for this is that when giving presentations it is most important to be able to focus on the speaking, that the pre-made sequence with the key points often supports.

In the use cases for the systems in this study, the focus is solely on the visual output of the system. The user focuses entirely on producing interesting visualisations, not having to do anything else, and often being even out of the view or at least focus of the audience. The aim is at more complex models, providing several degrees of freedom for even improvised action, enabling new meanings to emerge “as it happens”.

Instead of trying to make a program perform, the goal is enabling rich interaction for the user — subscribing to the metaphor of instrument making (Puckette 1991). In this view, live performing provides an interesting area of computer use that is at the same time very free, but also very demanding. That is, on the one hand, for the systems for live performing there are few rules that can not be broken: almost anything can be tried out. They are not security critical in the sense that e.g. banking or health-care systems are, so experimental architectures and unconventional ways of use can be introduced without unacceptable risks. But, on the other hand, they are mission critical in the sense that they must be highly relevant. They must be useful and dependable in the very intense usage setting, a live show, and enable contributing somehing very rich and interesting. Live performances are valuable *per se*, but maybe this combination of endless possibilities and high demands can give birth to something that has applicability in other settings too.

Apart from office work and presentations, there are other areas of research that take us nearer to the focus of this study. The system described here is basically about show, the usage could fairly be described as entertainment (although the underlying motivations might differ, and that whole concept is contradictory). There — in the traditions and innovations of theatre, dance and

[1]This was also the case in a separate example, where a game programming library was used to author a graphically intensive presentation, namely the Psyco (a Python JIT-compiler) presentation in the ACCU 2004 conference, using the Pygame library. For further information see http://psyco.sourceforge.net/

music — lives the notion of live performing. In these performing arts, multitudes of techniques have been taken into use throughout times: costumes to dress according to a role, instruments to produce different sounds and lights and decorations to create the atmosphere etc. Instead of a complete review and conceptual analysis of performing (or any) arts, however, this study presents the construction of a system addressing such usage. The idea is to use the making of the first prototype to identify interesting problem areas where more comprehensive literature research could be used to find solutions in future work.

In an overview model of interactive arts (Dannenberg & Bates 1995), the systems are seen to compose of four main components: 1. a human artist, 2. the 'instrument' (there: an artistically competent agent that realizes the artist's intentions), 3. interaction, including input from human players and output from the system and 4. an optional audience. A differentiating dimension is identified depending on whether the focus is on the a) *process*, where there may be no audience at all, but e.g. amateurs interacting with the system, or on b) *product*, where there's likely to be a highly skilled player using the system for an audience, that can enjoy the performance without knowing the process (Dannenberg & Bates 1995). In this work, the system development is made with the latter area in focus, but as noted by the authors of that model, many works include aspects of both ends of the continuum, and the implementation described here may be used for the former purpose as well, i.e. for playing without an audience.

Before proceeding within the topic of performing art systems, two related areas of research are introduced (they may also be seen seen as one from two angles), to clarifiy the similarities and differences of those areas to the one in focus here.

Firstly, the entertainment industry has during the past near decades embraced computer-based systems, as what may be called digital media , in breath-taking efforts that manifest strongest probably in production and distribution of movies. The applications there vary from non-linear editing of traditionally filmed material to complete computer-based animation, 3d special effects in cinemaphotographed scenes being a common combination of both. However, this has had only a little or no effect to the *event* of seeing a movie in the sense that it still comes down to watching serieses of images and listening to the accompanying sounds. Despite this, the basic techniques in computer graphics that this development has partially driven, are becoming applicaple in live performances, too. And this brings us to the other part of this field, which in a way surprisingly is (or may be seen as) *technologically* almost identical to what is needed for this study.

Computer games today apply several technologies developed earlier for production use elsewhere, such as a) film production and b) scientific visualisation, but with the same strong and challenging requirement as in live performing: perceived real-time. In fast paced action this is taken to extreme: in so-called first person shooters gamers need to go far beyond the 25fps of TV up to 70fps or more to survive, and arcade games may reach 200fps. For the purposes of this study, the European TV standard of 25fps is assumed adequate (it is also the maximum when using analog video, as was the case in the first use the prototype described here). Game technology achieves this by two means that make it distinct from the aforementioned non-real time areas; a) the visual quality is lower than in films b) scientific accuracy is neglected.

Finally, the relations to people open the differences of games versus performance software that are the key to the design goals that will be described in the following chapters. To put it brief and simple, games are usually challening in the way that it's relatively obvious what to do, but it's difficult to actually pull through the planned actions even if they are dramaturgically simple (like jumping to a platform or keeping a car on the road), so that success is rewarding *to the player*. In the terms of the aforementioned overview of interactive arts, the focus is on the process (Dannenberg & Bates 1995). For performing, even complex things should be easy to do and well under control, so that experiences or sensations could be provided for *other people*, i.e. the focus is on the product (Dannenberg & Bates 1995). The metaphor used is instrument making (Puckette 1991) — like a musician, the author wishes to have a powerful tool that may be difficult to learn at first, but must have enough flexibility for the performer to be able to create new meaningful compositions with it. So even though game technology is very close the needs in this study, the subject is actually about diverging from the goals of gaming. Specifics of this will be discussed along with the design, with the notion of *locus of control* as an attempt to clarify some ideas about making game-like scenes for shows, that are quite different from actual games.

Within this framework, the focus in work is creating a system for using 3d character animations in live performances. The motivation basically is that the author had rewarding experiences from using video content with interestingly moving people in VJ gigs, and he had enjoyed controlling them within the limitations of 1-dimensional time control (known as video scrubbing or scrathing) that video has. Existing computer game technologies feature real-time controls of character animations with numerous degrees of freedom, but no such system existed for performance use. So this system was necessary to be able to try it — experiment and learn from the experiences, to find out more about what it all is about. The system is called KyperMover and it is released as open source software on the Kyperjokki VJ engine Wiki.

The rest of this master thesis is organized as follows: first, the research method used is described and the research questions are presented. Then a look is taken on general requirements and related work, i.e. other systems for live performances, including some earlier work by the author. The pinned down requirements and the process of constructing the new system is described in chapter three. Chapter four presents the design and implementation of the system, focusing on the issues of control that are identified as a central problem area. Then the system is evaluated, based on its functioning and the experience from use of the prototype in an actual event. Chapter 7 discusses future development, from refactoring the core of the system to different areas where new functionality would be interesting. In the end, overall conclusions are gathered.

2 The Research Method Used — Constructive Research / Design Science

Within information systems research, two general branches or paradigms can be separated based on the nature of the activities and purposes of work done using different methods: a) natural or behavioral science, which is to *develop* and *validate* **theories** and b) design science, where innovative **artifacts** are *built* and *evaluated* (Hevner, March, J & Ram 2005). Design science research is similar to technical constructive research in the classification of Jarvinen (Jarvinen 2003). Further, within the design science paradigm, Hevner et al. emphasize that "researchers must identify important problems and, informed by prior natural, behavioral and design science knowledge, create innovative IT artifacts that address them." In this work, quite unorthodoxically and certainly controversially, the point of entrance and initial focus was on creating. The idea to apply such reverse order spun from the unawareness of the possible theoretical issues relevant to the matter. So the idea was to quickly explore the field by making a prototype, to know what questions and literatures should be studied to be able to make future versions of the system good.

Before continuing with the general requirements and guidelines given in the design science research methodology literature, a closer look is taken at the research problems and attempted solutions in this study. For one, questions rise about the *importance* or *relevance* of the problem area, namely visual live performance systems, and this particular system within that area. Revisiting the characterization of the research area that was already overviewed in the introduction, that question is answered in two different ways in the following. Firstly (**1.**) the author views that live performances are valuable *per se.* Although they are not considered in information systems research nor much within software engineering literature, and are quite marginal in the practical software development (even though an increasing number of tools does exist, as will be shown in the next chapter on related work), visual live performing is certainly an established activity in the society in e.g. theatre and dance — and nowadays as videojockeying (VJing) in parties too, which is the focus of this study. In fact, *novelty* in itself is considered as a criteria when judging the relevance of design science research, i.e. when information technology is applied in areas "not previously believed to be amendable to IT support" (Markus et. al. 2002) in (Hevner et al. 2005). Given that, what is the motivation for creating this new system, then? As mentioned in the introduction, and as will be explored in more detail in chapter 4 to define requirements for the system, the author wanted this system for experimenting with more degrees of freedom in using character animations in visualisations. Based on succesful usage of such video material, and the fact that suitable underlying technologies had already been developed for games, there was the opportunity to get to experiment with a new way of doing a show, and hence learn from it. Secondly (**2.**), as elaborated in the introduction, the author suspects that the *simultaneously free but demanding* nature of live performing is a potential place for experimenting with new solutions that may have more general applicability.

Another question rised by those initial requirements is the degree of being "informed by prior natural, behavioral and design science knowledge" (Hevner et al. 2005). One problem in this study is that it has not been very clear what the **relevant existing knowledge** would be. As stated in the introduction, the purpose here is to take a first step — build a first prototype and use it — in order to find out which kinds of questions there actually are, to be better able to identify which literatures would be fruitful in the search for underlying theories and models to use in the future development. But this is of course no excuse for omitting the use existing knowledge, and in the following chapter on related work an overview and analysis of existing similar systems and theories about them is given. As Hevner et al. further state, in the iterative problem solving process the initial constructions should be based on known theory. In software development, a form where that knowledge is embodied are the so-called patterns (Gamma, Helm, Johnson & Vlissides 1993), and that literature is used as points of reference for the design and implementation presented here. But the wider literature research on higher level issues, branching out to humanities and art, and perhaps also deeper into software engineering and computer science, is left for future work and dealt with only in the discussion chapter of this study.

Besides those initial requirements, **research rigor** is emphasized as one of the design science guidelines (Hevner et al. 2005). This refers to the application of rigorous methods in both the construction and evaluation of the design artifact. However, the authors note that an overemphasis on rigor may result in loss of *relevance*, so the approach should be balanced. In this study, both the construction and evaluation of the system are quite informal. As for construction, the nature of the programming effort was fit for rapid prototyping, taking as much as possible out of the limited resources of one programmer during the period less than one month. The method was inspired by extreme programming or the agile methods movement, emphasizing early and frequent releases (daily), focusing on simple incremental advances in functionality (instead of working on a large overall design to begin with), and refactoring the system relentlessly while adding new functionality to come up with an elegant design (Beck 2000). But even the application of this methodology was not rigorous, as e.g. no unit tests were made, despite the requirements and recommendations in the literature that suggest even test-driven development. This is due to the author's unfamiliarity with test driven development from before at the time, and also the probable difficulties with it in the selected technical environment. Concerning rigorous evaluation, the focus of this study is on the build process, making a first prototype, and finding potential areas for enhancement in it. So this stage is too early for large evalution studies, which would also be too laborous to be performed in this same work. However, in the chapter on evaluation, initial analysis is made from many different angles: the system in itself, compared with other systems, as commented by peers and observations from actual use of the prototype for the intended and other purposes. This should suffice in giving a whole estimation of the qualities of the constructed artifact to a sufficient extent.

Finally, there is the notion of **communicating** the results to different audiences. When discussing information systems research, Hevner et al. identify two audiences: technology-oriented and management-oriented (Hevner et al. 2005). The requirements for communicating about a design science study for those audiences are, according to them, that technology-oriented audiences need a detailed enough description to be able to implement the artifact, and management-oriented audiences need information for deciding whether their organization should invest in it. The system in this study is not, however, an organizational information system, and there are differences for other reasons too.

On one hand, regarding the technically-oriented audience, the artifact is published as open source software, so to be used as such it does not require re-implementation, and also for differing implementations the source code of this implementation can be used as a source of information. Also some of the components may be re-used in other systems, as will be shown later. The need for the technical audience is viewed more as explaining the design decisions and the motivations behind them, so that their potential usefulness and degrees of elegance can be judged for both regarding this and similar systems and applicability in other areas.

On the other hand, as for non-technical audiences, the author has difficulties in imagining any managers or management-oriented people involved (if not a director in some performance group?). Instead, potential users of the system are considered. In fact, in a proposed structure of an information systems design theory thesis or article, Gregor suggests (Gregor 2004) that the chapter on the description of the system instantiation is similar to documentation.

So two purposes of this thesis are to both document the design and implementation of the system for developers, and describe it for potential users that do not need to be know about all the underlying technical details. As for other purposes and audiences, the author can only hope that the work presents this research area in a way that also people not interested in neither developing nor using these systems can get an overview, and that reseachers in other areas may find some of the results interesting.

The main research question is: How to make a system for using 3d character animations in live performances?

The conditions being, that the time and resources are limited to some weeks of work by one person (more on the schedule in chapter 5).

Subquestion 1: How to make it so that many things, e.g. several characters, could be controlled by a single user (performer) in many ways at the same time?

Subquestion 2: How to support moving to the music?

These are obviously related to the requirements for the system, which are discussed in more detail in chapter 4.

3 Related Work — Systems for Live Performances and Character Animation

Related work is covered here concerning two areas: live performing and character animation.

3.1 Software Applications for Visual Live Performances

Computer-based real-time systems have been used in music performances since the 1970s, encompassing sound synthesis, algorhitmic compositions, automatic analysis etc., in both non-interactive and interactive pieces (Lippe 2002). On one end, the systems target automatic analysis of music played by a performer in real-time, so that the computer system can try to accompany it (Goto & Muraoka 1997). Also the work presented here targets accompanying music, but with visuals. So the solutions for automatic music analysis are very applicable in this area of research. The focus here, however, is not is not to make a fully automated accompanying system for a performer, but more to make an instrument — just a visual one — for human performers to use. On the music side this computer-based instrument-making idea was advocated in (Puckette 1991). A centre in Holland called Steim, the studio for electro-instrumental music, which is dedicated solely for performing arts and has a long history of electronic instrument development, emphasizes the use of the body and the idea of *touch*, referring to how the "knowledge of the fingers or lips" is crucial in playing musical instruments (http://www.steim.org/). In this work, the control devices are not addressed directly, so that advise is only kept in mind.

Dance is of course very closely related to this work, namely visual performing with character animations, because that is what dance is in the sense that the main means of expression is body movements (albait in dance they are real, versus the virtual ones in this work). One way to describe the work here is that it is a virtual dance system. Also, especially in contemporary dance, the use of video has become popular in the recent years. Specifically relevant to this work are dance pieces where computer-based character animations have been used as an element. One such piece is Biped by Merce Cunningham, which is exceptionally documented in the computer science literature, giving us a possibility to examine it here (Abouaf 1999a). The motions for the virtual dancers in Biped were derived from motion capture data, which was filtered and mapped onto a skeleton model. Later that was used to generate animations with a look of simplified hand-drawn chalk drawings. So regarding the the act of performing (and the requirements for the system for it), pieces like Biped are still just about playback of pre-made video, and hence do not contribute to the problem in focus here.. However, it should be noted that during the making of the choreography, Cunningham reportedly uses and has been involved in the development of a character animation tool (Life Forms), and describes one of it uses as "a memory device", the main interest being "as always, in discovery" (Abouaf 1999b). With this in mind a possible application area for improvisational systems featuring character animation, like the one here aims at being, is exploring interesting choreographies, and 'writing them down' for future use.

The most relevant other systems here are of course the ones with the same goal, i.e. tools for visual live performing, or video jockeying (VJing) as it is called. The bulk of VJ software is, quite naturally, video players, with capabilities for real-time switching, mixing and time-control. Tools like this include the popular ArKaos VJ, eXisTenZ, FreeJ and also KyperjokkiWorkPSX made by the author of this work using the Pixelshox Engine and Studio. Some of these video tools are plain 2d programs, but a trend has been to add 3d capabilities and to base new tools in hardware accelerated 3d via OpenGL (see e.g. gephex?) or Direct3D (e.g. Pilgrim) APIs. To give some overview to these tools before going to the details of the system described in this work, a closer look to some selected ones follows.

ArKaos VJ is popular commercial VJ tool, that can be used to play back video clips, show still images and mix and apply effects to them. Clips/images and effects are triggered with the computer keyboard or via midi controllers. Arkaos includes an event recorder, that saves information about keypresses during a run to a logfile that can be used to re-render the sequence afterwards (log editability to correct mistakes?). ArKaos has no 'scene construction', i.e. each content item is an image or video and they can't really be composited, although simple positioning is supported. Also, there is no abstraction for higher level events/actions — just the direct mapping of keys to images and effects — but of course external programs (like a midi sequencer) could be used to construct more complex events/actions and drive ArKaos accordingly. The new version (2.3) added time control which was not supported before, which could also be driven from a midi sequencer for being able to have different modes etc. Like most VJ programs, Arkaos features simple and straightforward use of media material: images and videos can be easily added to the 'bank' and they can be assigned to controls by dragging and dropping them to the keyboard GUI widget.

PixelShox is an interestingly different approach to building a VJ app., as it aims to differ from the predominant 'ArKaos model' by offering tools to construct sets in much more open ways, in order to enable making visuals that go with the music. Concretely it's an OpenGL engine with plugins for creating objects, colors, textures (also from video clips and live camera!), modifying and positiong them etc. Included is an integrated development environment (IDE) for 'wiring' the plugins together (e.g. live video input texture generator output to texture input of an environment map, or mapping the mouse pointer coordinates to a math plugin that sets the parameters of a Bezier curve based also on audio input analysis). Among the plugins is a JavaScript plugin, that enables free processing of numerical inputs and outputs. Previously, the author of this study developed a system using PixelShox, featuring video and object switchers and using the outputs of the switchers in different compositions (in pixelshox: “effects”), in a tool called (working name) KyperjokkiWorkPSX. There is also a time code component with real-time (mouse) controls to it, that was utilized by the video playback routines (both written using the JavaScript plugin). KyperjokkiWorkPSX has been used by the author of this work regularly for performances since early 2003. A main lesson learned for this work is the well-proven time code functionality, which is reimplemented and improved on here for the new tool. A major limitation with PixelShox was the restricted programmability, which forced the author to construct repetitive node graphs for certain basic features that would have been simpler to program in a few lines of code.

One interesting system, called Upstage, addresses *on-line* live performing (http://www.upstage.org.nz/). It was actually published after the work done for the system described here, but is also made with quite different goals in mind (has simple 2d graphics and audio and video chat features). Anyhow, there are relevant intersections with the systems which will be discussed along with other ideas for future development.

Apart from the aforementioned systems for performing art, as mentioned in the introduction, presentation applications are relevant to this work in the respect that they also are about showing visuals to an audience — one major difference being in the use situation, where it is best for the presenter to focus on speaking to the audience, and not controlling the visuals (not to mention improvisingly creating them). However, looking into the authoring of animated presentations gives interesting insight to the problem area. A system called Slithy is an animation tool for creating and giving presentations (Zongker & Salesin 2003). It was developed for creating meaningful animations, instead of the effect-animations that are typically added to essentially static Powerpoint slides. It is a script-based programming system, utilizing the Python programming language in an imperative style, providing a straightforward API for creating *parameterized diagrams*, *animation objects* and *interactive objects* (Zongker & Salesin 2003). The programming interface was selected, instead of a GUI, to emphasize power over ease of use, to encompass the large variety of animations that the authors envisioned users wanting to create. A similar decision was made in the work here, except that the library in this work is more about defining controls to pre-made animations, than about the actual making of the animations. Besides showing parametrisized diagrams, Slithy also features zooming and panning large canvases to have nice non-distruptive transitions from one part of a presentation to another (it uses OpenGL which supports this nicely). Regarding the style of content, the authors interestingly recognized — to their surprize — that the principles of classic animation do not necessarily apply to presentations. Instead, they propose principles such as: *make all movement meaningful*, *reinforce structure with transitions* and *do one thing at a time* (Zongker & Salesin 2003). Obviously, the style of animation in a show is much nearer to the classical style familiar from films, music videos etc., than the informational style of presentations, but it is still worth noting that there may be differences and we should be open to see them.

In conclusion, certain design goals are identified from existing systems: 1. supporting straightforward use of large amounts of material 2. a powerful time control mechanism for both scrubbing and direct play 3. a freely programmable interface instead of a GUI to avoid any restrictions

3.2 Character animation

Character animations are widely used in a range of areas, from medical diagnosis to film and games industry, and robotics. The data for the actual movements can be derived from motion capture devices, biomechanics literature, or manually made animation s, and sophisticated systems have been developed to compose them even automatically based on existing controllers and physics simulation (Petros Faloutsos 2001). The systems for composing the animations are outside the scope of this study, but the focus is on controlling them, with live performing in mind. Interestingly, controlling character animation during simulation has been seen a key feature for also the more common application of these technologies, i.e. making animations (Boulic, Huang, Magnenat-Thalmann & Thaimann 1993).

One relevant related area of research are virtual human figures, for which computer system development has been been done already for more than 30 years (Badler, Palmer & Bindiganavale 1999). There higher level controls for animations have been developed, one technique being the so-called parametrisized action representation (PAR), which are conceptual representations of actions including information about preconditions, objects inolved etc. (Badler et al. 1999). The PAR system has been extended to support automatic modifying of the executed animations too by variables describing the personality and emotions of the animated character (Badler, Allbeck, Zhao & Byun 2002). As will be shown along the design, the system presented here shares central features with the PAR system, regarding how actions are defined and used, but with fundamental differences too. Regarding the user interface, differing from the mainstream animation tools, the

PAR developers have explored natural language processing for altering the behaviour of the articial agents (Bindiganavale, Schuler, Allbeck, Badler, Joshi & Palmer 2000). For performing that is an interesting possiblity, as verbal commands can be given quickly and contain lot of information, but is not addressed in this work.

A three-layer design model for constructing non-verbally communicative avatars is proposed in (Kujanpää & Manninen n.d.). The model is developed for increasing the expressiveness of avatars in networked computer games, to enhance communication between players, so the differences with performing for an audience must be kept in mind when evaluating the applicability of it here. On the first level, there are the elements of non-verbal communication used, referring to certain actions such as waving a hand or smiling. Then there is variance, parameters for the movement (e.g. speed, or sharp/softness). Finally, in so-called personalization the individual characteristics for a model are constructed, by e.g. selecting the possible variations it has for certain movements. In the system presented here, there is no achitectural support for variance, as it is limited to the playback of the pre-made animations. However, that model can still be used as a guideline for the animators. And, as will be shown, the technique for constructing the performance setups, i.e. what models and with which animations there are to be used, supports the final personalization phase as there the set of animations is individually configured for each model. And returning to the issue of variations, the ability to affect the style of the movements during a performance would be a very powerful feature, and should definitely be kept in mind in future development.

4 Requirements for the System

A primary motivation for this work was experimenting. The author had good experiences about using character animation centered content in VJing, in the form of controlling the playback of video and animation clips (for example video cuts of humouristic kung-fu scenes from Hong Kong movies for tight lively breakbeat music, and drawn animated smooth capoeira sequences for flowing techno). Also, the experience from the real-time controllable 3d compositions that PixelShox had enabled was encouraging. So there was curiosity about using freely real-time controllable 3d character animations. Furthermore, the start-up company, where the author is involved, had some real-time controllable character animation centered product ideas. So a main requirement for the system was to simply enable doing that experimentation.

In this chapter, particular requirements are identified (or constructed, in that view of software requirements elicitation) in the following ways: First, general ideas, or a design philosophy, about what kind of things should be targeted, and especially how they differ from common structures in computer games, are presented. Then, specific requirements for the wanted content type, 3d characters and animations, are identified. Also, a view is taken at the physical operating environment and the usage situation where the system should be usable. Further, technical non-fuctional requirements are outlined. Finally, the results of initial brainstorming sessions for features are presented as a wishlist, and in conclusion certain primary requirements are selected as acceptance criteria.

Here the requirements are presented as they were seen to begin with, i.e. what the initial estimations or 'best guesses' about them were before the actual design and implementation began, in early August 2003.

4.1 Locus of Control

An aspect of the design goals was conceptualized with the help of something called 'the locus of control'[2]. This is used to refer to the things that the user of a system is controlling. The attempt is to clarify the thinking about the separation of conventional computer games and systems for performing for the audience, which may use similar underlying techniques but for different purpose. To illustrate this thinking, two examples that the author had in mind in summer 2003 before starting the work on the implementation, are described in the following. The author is unaware of existing characterizations of this idea in the literature.

Car follows road. Car driving games are one of the most typical computer games, falling in the category of simulations (even though many fun ones may be totally unrealistic). There a typical scenario is one where the car is on the road that continues on, often disappearing out of sight as it curves to either side or perhaps turns downhill. The controls for the player resemble what there is for a real driver in a normal car, i.e. he/she can e.g. accelerate or deaccelerate and turn the car left or right. The goal, and hence the challenge in the gameplay, is to stay on the road, with extra challenge usually added by constraints such as the time or other elements such as traffic etc. Now, thinking of a show, it is probably not very interesting for the audience to watch someone driving a car, — nor are the controls of a car very powerful means of expression, or an instrument, for the performer. But by shifting the 'locus of control' something more interesting might perhaps be made out of this scenario. For example the driving of the car would be automated, but the performer would control other elements in the scene. For a dance event it might be interesting to have a preset where the controls, e.g. a joystick, would determine where the road will go next, i.e. if it turns left or right, or uphill or downhill, so that the performer might adjust it to the music (along with the colours of the sky etc.). This system would be totally uninteresting as a game, as there is basically no challenge whatsoever, but as an instrument for performing it would provide quite a rich scene where many things, such as the automated driving, can be happening but the user could still have strong overall control of it, being able to manipulate the look and feel of the whole scene.

Character bounces a ball. Another example of a gamelike scene, where this technique or design philosophy of changing the controls could be applied, is one where a e.g. a human or human-like character is bouncing a ball. In a game, the challenge would again typically be similar than when actually bouncing a ball, i.e. moving the character so that it will keep the ball up in the air, possibly trying to get extra points by using different parts of the body or different movements for it. Even the basic challenge of keeping the ball in the air might be quite difficult, demanding exact timing etc. But what would be interesting to control when using such a scene as an element in a show? One possibility is the ball: the performer could e.g. move the ball around with the mouse, and the character would be automatically hitting it in interesting ways when the ball would be brought to certain distances from it. There again there would be no challenge in keeping the ball in the air, as the character would be automated and no exact positioning nor timing would be required by the user. Instead, the performer would be able to concentrate wholly on the show, on how to get interesting movements and visual compositions out of the situation, and would have the tempo and rhythm under control. Additionally, there could be controls for selecting the style of movements that the character is using, perhaps using the metaphor of mood (e.g. relaxed or aggressive motion), or indeed the shape, colour or texture of the character itself (and why not the ball too).

[2]This 'locus of control' is not the same concept as it is in psychology, after Rotter, but is used here due to the lack of a better / non-overlapping term.

These two example hopefully demonstrate the kinds of functionalities that were sought for in the development, and clarify how the systems used for performing can be at the same time very similar to and different from games. The analysis of underlying theories of controlling etc. that could be used to both formalize and richen these ideas are at this point left for future research. However, these scenarios were **not** set as requirements for this system, but the selected focus was on character animations — dancing to the music — instead.

4.2 Requirements for 3d character animation

Basically the requirement regarding 3d character animations for the system was that it needs to be able to show them, and that the user can control them flexibly in real-time. This means that the engine needs to have a skeletal animation system and adequate means for controlling the playback of them via a programmable interface. This is a standard feature in 3d game engines, and for the minimal functionality there are no special requirements. It was considered highly desirable, though, to be able to mix or blend several movements. Also the possibility to freely move the bones, apart from playback of pre-made animations, was considered interesting.

Concerning the material, the possibility to import material (i.e. 3d shapes and animations) from popular modelling and animation applications, including the one used by the modeller & animator in the project (Blender), was naturally required. Additionally, the possibility to use industry-standard motion capture data, i.e. ready animations of different movements, using the Biovision hierarhical format (BVH) was seen desirable.

4.3 Requirements set by the operating environment

As the usage situation in live performances differs greatly from common computer usage in homes and offices, it is necessary to identify the requirements set by it for the system. Basically these were already approached in the introduction, but here we return to the issue to identify them in a more specific and complete manner.

For one, the events where the performances typically take place involve temporary arrangements — sometimes the whole physical set-up, including e.g. electric wires to power up the devices, is built in a matter of hours. On such occasions, computing resources that may be common in workplaces or at home, such as local file servers or external network connectivity, can not be taken for granted. The simplest thing is to rely on nothing but a standalone application running on one computer, with connectors ready to plug it to the typically analogue audio and video systems. Also, on the stage there may be limited room, when the control devices should be pretty compact.

However, as a single gig may last even up to 6-8 hours, i.e. the duration of the whole party, there must also be enough variety in the material and the ways it is used if a goal is to keep it interesting and different throughout the performance (of course this depends greatly on the nature of the event, the role of the visualisation there, the style of the performance etc.). Sometimes this requires different software, that may again require different operating systems and/or hardware. Also in some cases, where the computer-based visualisation has a major role in the overall show, several devices are required to have redundancy in case of failures. So sometimes several computers are needed, in which case they can be used for single visual output usin a separate video mixer device.

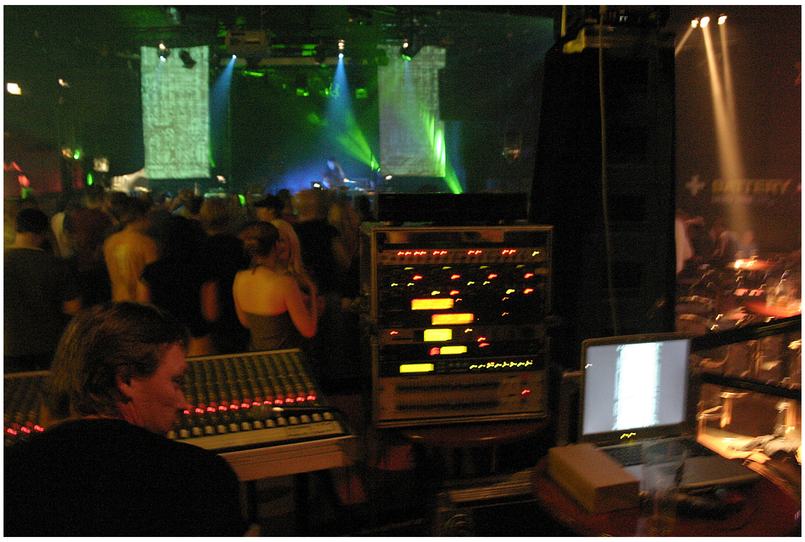

Figure 1. A minimal VJ set, where a single laptop handles the visuals. (Photo (c) Heli Hintikka / http://www.pixheli.com/)

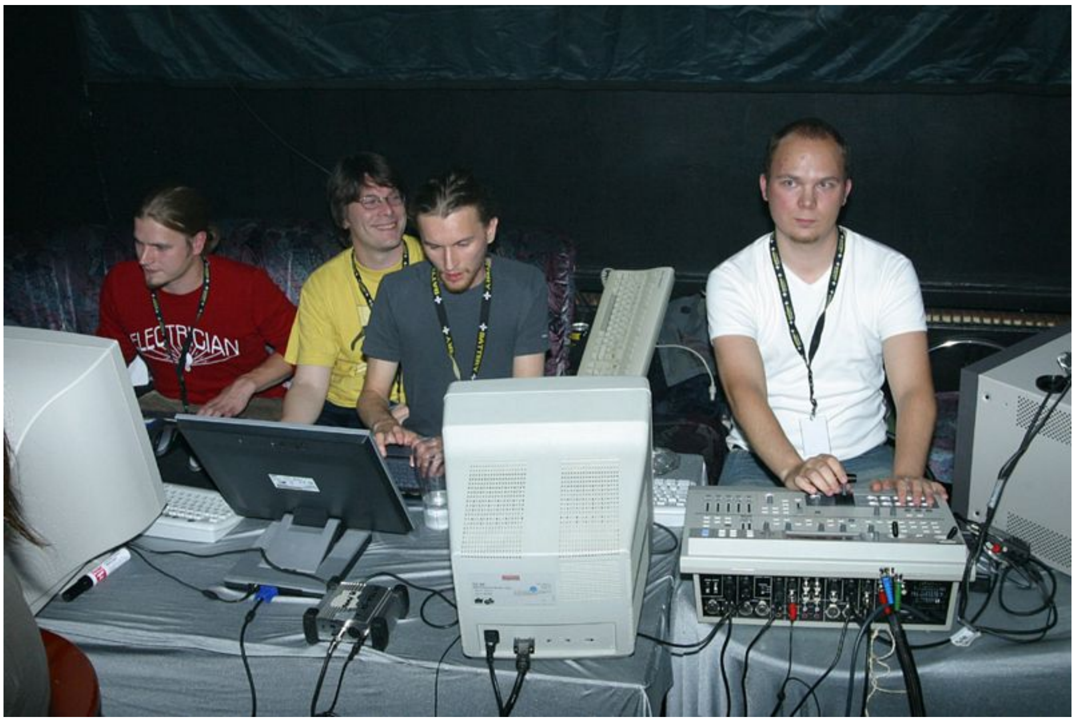

Figure 2. A more complex VJ set, where multiple computers are as video sources to a single video mixer. The system presented in this work is in actual use in the computer in the middle. (Photo (c) Heli Hintikka / http://www.pixheli.com/)

For the particular target of the development described here, the plan was to have other computers with other software for other kinds of output (such as video playback), and for this system it sufficed to deal with the character animations only. Using a black background for the empty parts in the resulting image makes it trivial to mix it with other video sources.

4.4 Technical requirements

The system is not 'hard-real-time', meaning that lagging results are not completely worthless, using the definition from (Dannenberg 1989). Furthermore, accuracy down to milliseconds is not required, unlike with music. Instead, a 'best effort' approach suffices, i.e. very low-latency is desired but occasional lag is not catastrophic.

Stability demands are high, and quick error recovery desirable.

The system should be able to use large banks of material, and feature easy switching from showing some models & animations to others.

4.5 Features wish-list

To come up with a set of functional requirements, a sort of a wishlist of features to have was drafted in some intense brainstorming sessions. These were initially drawn on paper and put up on the studio wall, where the software was later to be implemented. Later, they were partly collected to a collaborative web space as a Wiki node at http://studio.kyperjokki.fi/engine/KyperMoverWishList . This list was simply derived from the practical VJing experience of the developers, and used loosely to map the problem space — not as an actual requirements specification, and is presented here as a sketch just to give some idea of the range of issues that are relevant to this kind of system from different perspectives. The terms are not explained here, nor is the understanding of the list necessary for reading this work.

4.6 Selected primary requirements

As concrete goals for the development, and minimal criteria for the acceptance of the system, the basic functional requirements were selected as follows: the system must enable having 3d objects with animations so that a) the objects can be switched, b) the animation to use can be selected and c) the animation playback can be controlled live. Other functionality, such as the ones listed in the wishlist out of the early brainstorming sessions, are optional. The rationale for the selection of these core features was basically identifying what is the absolute minimum to be able to perform with the system meaninfully: the perfomer must have a wealth of material at hand to fit the mood during the event (mainly the music), hence the requirement for switching objects (a) and selecting animations (b) . Also the phase and speed of the animations must be in relation to the music and dancers somehow interestingly and meaningfully, e.g. move to the beat or be opposite to it (especially slow visual for a fast period in music can be an effective contrast), and hence the playback must be controllable (c).

5 The Design and Implementation

The software development method used was inspired by the agile methods movement (Beck 2000), that has drawn quite some attention during the recent years. Practically it meant that there was always a functional prototype at the end of the day, and that the whole system was refactored several times even during the short development period of few weeks that we are looking at here. New functionality and refactoring (that was often needed for the new functionality) were always introduced in a new branch, so that there was always a way to go back if the attempt failed.

Therefore, practical requirements to meet the high-level goal were refined and discovered continuously while designing and implementing the software system.

The schedule set thight requirements on the process. To guarantee having a working system by the deadline, the first version was targeted to be in shape in two weeks — about in the halfway of the whole development time. One of the reasons for this was preparing for the worst case scenario, so that if the first attempt would fail totally there would be still time to start again and reach the minimum goals. Two kinds of potential risks were seen: 1. failure of the engine-to-be-selected to meet the demands 2. failure to construct the own system, that uses to engine. So it was estimated that in two weeks either the system could be moved to use another engine, or made differently using the same engine that had been used to begin with.

Table 1: Schedule of the development time

Due date	**Phase of Development**
7 / 21 / 2003	pre-alpha experiments
8 / 1 / 2003	selection of the 3d engine
8 / 15 / 2003	functional alpha1 prototype
8 / 27 / 2003	full-featured alpha2
8 / 30 / 2003	reliable beta in actual use

5.1 Selecting the 3d engine

A key part of the whole undertaking was evaluating different 3d engines and finally selecting one of them (and then sticking to it, when there were difficulties because of it). For future development, this remains a daunting on-going task. This table represents the picture that the author got of the selected candidate open source 3d game engines, in autumn 2003. An on-line version of the table is available at http://studio.kyperjokki.fi/engine/ComparisonTable , with possible updates, but no guarantees (there are many engines and comparing is hard work).

engine / feature	Soya	GameBlender	Crystal- space
Character animation	Cal3d integrated, seemingly nice controls. Mixing untested but probably works as usual with Cal3d.	The own armature system, no mixing of multiple movements affecting same bones.	Cal3d
Content creation	Animated characters can be imported from Blender.	Is integrated into Blender, so setups work straight.	projects have their own ways. later CEL and crystal2blend

Python bindings	The engine is written totally for and partly in Python, nice API.	GameLogic API, Python scripts use logic bricks, can be cumbersome.	Generated with SWIG, which gave a C++:ish API at that time.
Platform availability	GPL source, uses crossplatform Python libs, is developed on Linux.	GPL source, binaries for mac, win, linux, irix, bsd, ..	GPL source, used on mac, win linux etc.

5.2 Blender and its game engine

Blender was selected as the engine. It is actually a 3d modelling and animation and rendering tool, which very exceptionally includes a real-time engine for interactive 3d. Blender can import data in some and via plugins/scripts several popular formats used in other programs, and there are tools for importing character animation / movement data from Biovision hierarchical (BVH) files, that is commonly used for motion capture data. Elemental to Blender is the embedded Python interpreter and the APIs for programming it, both on the modeling/animating/rendering side and in interactive 3d / gamelogic. Python is extremely well suitable for both rapid prototyping and programming product quality object-oriented software, so the support for the language was a major factor in the decision.

Blender is originally developed by Ton Roosendal and later by others too "for an animation studio in an animation studio", but later released as a product for other users too. The tool was distributed free of cost to anyone interested, and became quite popular. There was a business model of selling advanced features required for commercial publishing, but the company doing it, Not a Number (or NaN) closed down during the financial turmoils of early 2000s. However a fundraise was organized for the users and managed to collect more than the required 100,000e to open source the program. The Blender Foundation was founded, including animation studios and the Society for Old and New Media (in Amsterdam, http://waag.org) as Gold sponsors and, interestingly for some, Nokia as a silver sponsor. The application is now free software (both as in speech and as in beer), as the source code is licensed under the GNU General Public License (GPL). It is supported on several platforms (currently at least IRIX, Linux, Mac OS X and Windows, and also a version for PocketPC). In the open source development, there is an active group of programmers (the author being a hang-around member at the time of this work) that has already put many, even major, new features and bugfixes in place. The foundation employs the lead programmer Ton Roosendal, who is to Blender perhaps something like what Linus was/is to Linux, full-time.

So using Blender for modeling and animation seems to have a solid future, and it was succesfully used by the project group to make the content needed in this project (the author did not do any modeling nor animation himself).

The Blender game engine (Ketsji), however, is another story. To put it short, it was seen sufficient (though not perfect) for the needs of this project. It was also a somewhat safe choice, as the author had some earlier experience from progamming with it and it's quite an old an well-known engine with an active group of very supportive users.

So, finally, a design decision was to use Blender for live performing, and as it happened, it was used for almost everything in the project: modeling, animating and even texturing for simple objects. But, as will be shown in the following, this lead to some awkward situations in programming as there are some (even severe) limitations with the game engine and it's API. After the project, the author has actually looked into fixing/developing the API to straighten some of the workarounds that were needed. It should be noted, however, that programming with the GameBlender API is very high level, with no need to (or even way to) go into details of graphics rendering etc, mostly about logic (the module is actually called GameLogic).

5.3 General architecture

Certain general principles for designing real-time applications for live performances are put forth in (Dannenberg 1989). While the focus there is on music, the system presented here is aligned with several of those guidelines, namely: non-preemptiveness, event-driven multiprocessing and the separation of graphics from audio processing.

The overall structure is layered, loosely according to the common *layers* pattern (Buschmann, Meunier, Rohnert, Sommerlad & Stal 1996) (p.29). This layering is completely conventional, as it is the way 3d engines are typically used. Simply, there is the underlying 3d-engine that is responsible for all low level functionality, such as controlling the graphics hardware via (in this case) the OpenGL library, managing the object shapes, textures, animations etc. The system described here, codenamed KyperMover, uses the selected engine GameBlender for all activity. However, the layering is not strict, as a pure form of the layers-pattern would require. That is, there are no implementation independent interfaces defined that the modules would use. Instead, the API provided by the engine is used directly. This is done to avoid premature unnecessary work. If some such interface definitions are feasible at all, their design and implementation is left for future work, and the issue is hence revisited in the chapter focusing on that.

Also, on a closer look, the matter is more complex. There is in a way an additional third layer in-between the underlying engine and the logic that drives the performance. This refers to the components that are implemented in the Blender project file, to in a way glue together the engine and the logic parts. Due to the somewhat limited ways of programming that GameBlender supports, some quite messy hacks had to be made into some of those components, to e.g. initalize the objects in ways that support their flexible use later etc. On the elegant side, the 'controls' module implements the mapping of controls to action-driving events which will be described later in this chapter. As the solutions on this layer are very much engine specific, should be hidden from the users, and contain close to none generally interesting code, their examination is omitted from this work.

The logic, consisting of two Python modules, initially called 'kyperjokki' for the basic library and 'ksetup' for the configuration file, are in external files. However, also both of them directly access the GameLogic module via which GameBlender provides its API. So the layers are indeed not completely isolated, but there are central interdependencies. The focus of the descriptions here are on this top-level layer.

Table 3: The layered architecture

Layer	**Implementing modules**
Logic	kyperjokki, ksetup

Table 3: The layered architecture

Layer	**Implementing modules**
Controls	(controls, initialisations)
3d-engine	GameBlender (ketsji)

From another perspective, the overall architecture can also been seen as a data flow, resembling the *pipes and filters* architectural pattern, even though the implementation is nothing like it (Buschmann et al. 1996) (p. 53). The flow here is that there are multiple original sources for multiple kinds of data, such as the 3d shapes, textures (i.e. images) used in them, skeletons used for character animations and the animation data. Also, the users of the system are expected to author controlling logic to be used in the performances, and pack and configure all these parts into so-called setups. Blender, true to its name, can be used to do all this. The first implementation does not support any other ways, i.e. it can only work with a properly prepared Blender project file.

5.4 The Class Library

The main result of the work is the class library written in Python that implements the system, together with the necessary configurations in the Blender project file. Besides providing the basic features the author already wrote, the idea is that the API and the programming model would be usable for other users, and the original author in the future, to implement new controlling logic to their visual scenes. The class library is presented as a high level UML diagram in Figure 4., and relevant parts of it are reviewed thoroughly in this chapter.

5.4.1 High-level Events map Controls to Actions

After initial drafting and discussions, it was decided that a model for what were called 'events' would be implemented first. The target was a high-level even abstraction to bind controls (input devices, such as mouses, keyboards, joystics, midi-devices etc.) to low-level actions (what the engine is told to do), so that the actual events would be independent of the controls used at the time. A reason for this was the want of an event recorder (similarly to ArKaos, as described in chapter 2), but so that the log could be well editable (or even written from scratch to start with) and not be affected by changes in control mappings.

There are existing event-driven systems for both live performances (Dannenberg & Rubine 1995) and controlling animations (Boulic et al. 1993), as well as for making interactive systems / games in Python (http://www.pygame.org/) and networking (http://twistedmatrix.com/). For the purpose of controlling the Blender game engine, however, the existing ones could not be directly used. So an own implementation was written and tied to how the game engine is run from the Blender side.

Here the notion of an event is somewhat different from the common concept in programming. Often 'event' refers to a message, such as MOUSEBUTTONDOWN in SDL, which are processed by an event handler, e.g. the main loop in a game, which further initiates the appropriate actions that that event should cause. The event itself may be just the signal, or optionally include some additional information, such as which mouse button was pressed. In the following, the system described *includes the actions* that the event causes *in the event object itself.* Furthermore, the architecture is made so that nothing else can initiate any actions — it is all driven by what is in these events. This is contrary to a common design in games, where the main loop e.g. calls the update-methods of all game characters, evaluates the events triggered by the input devices used as controls and consequently e.g. changes the states of the game characters, does collision checking etc. In this system, the main loop only handles the control inputs and activates and deactivates events in the current scene accordingly, updates the values of controller objects that are used to adjust certain actions, and updates the scene by executing the actions within active event objects. A more detailed look into this design and implementation follows, and is later reviewed in the evaluation section.

Yet a few words about the event-class from a conceptual angle. As described above, the events here differ from the common ones in that they include the code to drive the actions they are about, hence they are not simple signals (and they don't include any information about external controlling devices, as events in game and GUI programming often do). What are they, then? The concept the author had in mind when developing the system was that these events are about *what happens in the virtual scene*. So any given moment, the events that are active in a scene define how that scene will change for the following moment. The wording might well be misleading, and actually at a point it was seriously considered that these events would be renamed as *actions*. A reason for not doing this to begin with, and why the change has not been made afterwards either, is that the term is already heavily used in the system, for in Blender the character animations are put together as actions, as will be seen later. A better term (hopefully not 'happening') would be welcome, if this basic idea proves feasible.[3]

A motivation for this approach was to facilitate controlling many things at the same time, by for example changing a more general aspect of the whole scene that affects several characters, instead of controlling a single particular character like usually in games (as was discussed with the requirements, via the design philosophy of chancing the 'locus of control').

These events or actions also resemble the command objects of the command pattern (Gamma et al. 1993), to the extent that refactoring them to be commands should be seriously considered. But then extra care must be taken to see whether these constructs, currently and originally built as virtual events, are and/or should be the kind of commands that pattern describes. For example, the command classes are often implemented by having a single *do* method (and may have an additional *undo*), but not supporting the long-spanning and potentially multi-phased actions that the event class here is designed for. Also, the command pattern involves defining supplier components, which are not used here (although the resources that the event-driven actions manipulate could be seen as such). Furthermore, as Eckel (Eckel n.d.b) notes in his book Thinking in Patterns in Python, that the command pattern, as used in e.g. C++ and Java, is sometimes redundant in Python, where all functions already are also objects. In fact, the reverse can be true too and perhaps essential for the refactoring of this mechanism: that is, in Python any object can be made callable (like functions and methods are), simple by defining a special *call* method (spefically __call__, compare with the *act* method in the following paragraph). When writing this first implementation in autumn 2003, the author was actually unaware of this. For an example of the utilization of this feature, see e.g. the Freevo2 Action & Event system, where events are typical simple messages which trigger certain actions on selected items, the (menu) items having several possible actions defined as normal methods, but also being callable so that the special *call* method executes the default action (Meyer 2005). This issue is revisited in the later chapter on future development.

[3]A performance system from STEIM calls sequence of events *licks*, and some imagination could be used in the renaming of this construct too. http://www.steim.org/

Here, the design is presented as-is, to base the evaluation and planning of future work.

The events have three special methods: *start*, *act* and *stop*. The start method is called when some control or another event triggers the event. The default behaviour and some interesting specializations demonstrating their potential usages are described in the following:

```
class Event:
    def __init__(self, scene, type)

    def act(self, mover=None):

    def start(self):
        self.scene.act_events.append(self)

    def stop(self):
        self.scene.act_events.remove(self)
```

Minimally, the start method must add the event itself to the group (list) of active events in the scene instance it is bound to. The scene class is used to encapsulate everything, and to drive all action. Optionally, the start method may be extended to do anything. For example, an event that makes the camera track a target defines the target in its start method, and the event type for running animations reserves the animation channels (a Blender specific concept) for those animations.

The *act* method is called every cycle, i.e. in every iteration of the mainloop, as long the event is active. The default behaviour is to execute all code that is in the list of actions of the event, to allow the creation of simple events without making custom methods. For more complex events, the base class is typically subclassed and a special *act* method defined.

Stop is the counterpart of *start*, i.e. by default it just removes the event from the group of active events in the particular scene. An interesting specialization of it was programmed for a camera moving event, however, as there it does not deactivate immediately but initiates a stopping sequence during which the movement of the camera slows down gradually. Initially that was implemented by having the *stop* method change the state of the event to 'stopping', and not removing it from the active events, which causes the scene update mechanism to continue calling its *act*, which was extended to treat that mode specially i.e. do the gradual slowing-down and finally, after a counter has run to the threshold, stop the motion and call the standard stopping methods of events, deactivating it in the end. This results in softer camera motion, but the generalization of this technique or some other for softened and other kinds of stylished motions is left for future research.

```
class GlobalEvent(Event):
    def __init__(self, scene):
        Event.__init__(self, scene, "global")

class MoverEvent(Event):
    def __init__(self, scene):
        Event.__init__(self, scene, "mover")
```

Furthermore, all events are typed in two mutually exclusive kinds: so-called a) 'global' and b) 'mover' events. This decision was made to allow for a) controlling general actions, such as changes in the scene, with the 'global events' and b) controlling several character animations at the same time with the 'mover events'. The difference is that 'global events' are called once per cycle without any special context, but the 'mover events' were initially called once per cycle for every active 'mover', i.e. controlled animated 3d object. For example, a) activating or deactivating 'movers' is a global event that is called once per cycle in the scene, and b) moving all active movers forward is a 'mover event', i.e. the scene update called that event.act with the particular 'mover' as a parameter, once for each mover per cycle. This design/implementation was changed, however, so that currently the 'mover events' *act* methods are also called only once per cycle, but they all internally repeat the same actions for every active mover. Furthermore, special events were made to control the movers by calling their methods. The further feasibility of this design is yet to be analyzed, but in the early prototyping it did enable both the exact control of single things and (rudimentary) control of multiple things simultaneously. As there is principally no limit to the complexity of how and which actions an event causes, the 'mover events' could be composed to result in more delicate movements, e.g. with respect to the spatial relationships of the different 'movers'.

5.4.2 Movers in Scenes

The Scene class was made to encapsulate basically everything needed. It can be seen to consist of, or to aggregate, the different components that are needed for the whole system to work, along the lines of the whole-part design pattern (Buschmann et al. 1996) (p. 225). However, in contrast to the pattern, the Scene does *not* prevent direct access to these constituent parts. Instead, some of the parts are designed to provide information to, at least other parts, but also to outside (e.g. events that manipulate the things in the scenes, and the control code that sets values of controllers). So some of the parts and some of their properties are considered belonging to the public interface this Scene provides. This violates the so-called law of Demeter, or the principle: 'don't talk to strangers', i.e. that one should never address objects via another object (Lieberherr, Holland & Riel 1988). In other words, structural knowledge about the referred objects should be minimized (this is also called structure-shy design). However, utilizing structural knowledge of the Scene class, and even the classes it encapsulates, in the methods that use it was considered a convenient technique at least in this rapid prototyping phase. Details about this will be shown in the descriptions that follow. At least in some cases the data accessed via the structure is like global variables (e.g. the current timecode), to which access is allowed in the Demeter rules, but which are not technically global variables in this system (i.e. as shown below, timecode is a property of the scene-bound instanciation of the TimeCode class).

All potential events are registered in a scene (at their creation i.e. initialisation), partly to allow communicating about them over the network with remote controllers. This registry is technically a Python list, called **pot_events**, and conceptually is 'what *can* happen' in the scene. Similarily, there are **act_events** for the currently active events, i.e. for 'what *is* happening'. These resemble the two registries in the Parametrized Action Represesentation (PAR) system, namely the uninstanciated (UPAR) and the instanciated (IPAR) actions, but there are central differences: UPARs are generic, containing only the default parameters for an action, and IPARs contain specific information about the agent, objects and other properties involved (Badler et al. 2002). However, in this system, it is the event class definitions that correspond the UPARs, and the so-called potential events (in pot_events) are already instanciated, effectively tied into the current scene. Furthermore, the active events (in act_events) are not bound to a single agent/actor, like the IPARs are: as was discussed above, they may involve no actors at all (GlobalEvent) or several at the same time (MoverEvent), even so that the actors/movers involved change during the activity of the event. These differences reflect the design goals of these two different systems: in the virtual human research, from and for which the PAR system has been made, the goal is to simulate human behaviour with autonomous artificial agents, possibly involved in complex tasks that for example a training simulation may require. Here the goal is to allow a real human to manipulate a scene with characters, to achive visually compelling compositions.

In the light of the command pattern mentioned as a potential way to see the event class here, this part of the scene class can be consequently seen as a command processor (Buschmann et al. 1996) (p. 277). However, there are differences: in that description of a command processor, it would be responsible for auxiliary features such as logging. Yet in this work, as noted when describing the Event class, the plan is to implement logging as a part of the events/commands themselves. Also, a part of this command processor -like activity is implemented outside this Scene class, in the controllers module included in the Blender project file. The division of labour is that that controllers module handles user input and manipulates the information in act_events, that the Scene then later uses to actually run the action.

As a note regarding the technical implementation, there is no reason in the current implementation why these groups of events should be ordered (as lists are). Therefore, they might be refactored e.g. to be either dictionaries or sets. With dictionaries they could be accessed by name when needed (e.g. when doing remote controlling over the net, which was ugly in the first experiment where the meaningless list indices were used to identify the event in the network call). Sets will be a new built-in type (or class, as they are united) in Python 2.4, which was not available at the time of the writing of this system (Python 2.0 had to be used because it is the one embedded in Blender 2.25, which was the stable release with the real-time/game engine at the time). Sets provide, besides basic methods for adding and removing from the group, methods familiar from set theory in mathematics, such as intersections and unions, checking for supersets and subsets, which might prove useful for these groups in the future. Following the extreme programming principle or practice called 'you aren't gonna need it' or YAGNI, which states: "Always implement things when you actually need them, never when you just foresee that you need them.", neither of these refactorings have been made yet (it is not even known what the concrete needs will be, and therefore it can't be known which solution would be the most sensible) (http://xp.c2.com/YouArentGonnaNeedIt.html)

Also, it should be noted that the ordering in act_events *may* sometimes actually matter, e.g. the behaviour of certain events depends on some state that other events also effect. A simple example follows: First, let there be an instance of an event type called EverythingKeptStraight, that keeps all visual objects in the scene somehow aligned, referred in act_events — i.e. such a thing 'is happening'. Then, there are other events that move some of those same objects somehow. Now, the result will depend on the ordering of those references to the event instances in the act_events of that scene, because they are executed by iterating that list one at a time, once in every cycle, as will be shown below. So if the EverythingKeptStraight is after all the other movement events in that list, all the objects are actually aligned (they may still move, as the aligning is done after the other movements and may hence differ in time). Or if the other movement events are executed *after* EverythingKeptStraight, the objects do not appear aligned, because they were moved within the same cycle after the aligning adjustment had been made. But this is incidental, and not designed. So the question rises: what defines the order of the events in the list? Interestingly, it should in the current implementation be the order in which the events have been activated, e.g. in which order the keys that activate these behaviours have been pressed. So if the user first activates EverythingKeptStraight, and watches it for a while, and then activates some movements, that might for instance result in strange (and perhaps interesting!) jitter. Then, if the user wishes to override movements with EverythingKeptStraight, he/she should (first de- and then) reactivate it, while keeping the other movement events active, so that it ends up being later in the list. A GUI could of course show this list and provide means for manipulation. So if these data structures for the groups of events are refactored, means for defining the order of execution should considered.

Similarily, there are lists for the groups of all movers registered in the scene (pot_movers) and the currently active ones (act_movers), to which the actions driven by the events should apply.

Also, the scene has references to controller abstractions and some supporting classes that will be examined later. It also keeps several state variables, such as different modes, a reference to the active camera etc. These are part of the public interface, and hence ready to be used by the action-driving events and potentially other code too.

After the work presented here had been done, a Scene class was added to the Python API of Blender itself by spring 2004 (the modelling/animation/rendering side, not the game engine). That addition will probably help in streamlining the process of instanciating these real-time Scenes, which will be looked at later when presenting the setup file and again in the chapter on evaluation. Also, the news came in in August 2004 that a Scene module is included in the official release of the game engine too. So in a future refactoring this self-made Scene class should be redesigned and implemented to interoperate with that somehow. There, following the composition style that is already practiced here, instead of inheritance, should be considered (Gamma et al. 1993) (p163). That is, this Scene could be made a MoverScene, or VJScene or whatever that subclasses the base Scene (if that is even possible, it depends how the newly added module in gameblender is made). However, it is probably better to have a separate construct, that just uses what the engine provides (similarly to how Mover objects have an association called '.bobject', referring to the blender object that they wrap).

```
class Scene:
    #all potential events in this scene (setup / preset?)
    self.pot_events = []

    #events that are currently activated (by some control)
    self.act_events = []

    #movers in this setup (from blender side to get their GameObjects?)
    self.pot_movers=[] #all potential
    self.act_movers=[] #all to which actions should apply now

    # mouse axes go to these now
    self.horizontal = Slide()
    self.vertical = Slide()
```

The engine, i.e. GameBlender in this first implementation, calls the update method of the scene which does some maintaining and centrally calls further the events to act, which results in the actual actions back on the engine side.

```
def update(self):
    self.time.update(self.controllerobject.timer)
```

```
        for event in self.act_events:
            event.act()
```

Currently the only utility method in the scene class is the one used to add new movers to a scene, which uses the *addobject* actuator of the Blender game engine, which further has been customized to register the newly added objects as needed.

```
    def addMover(self, number):
        self.addobject.setObject(self.movernames[number-1])
```

The title class of the whole project, Mover, represents basically any object that is to be shown and controlled, i.e. moved and animated. Each mover instance is bound to a scene, may be activated and deactivated there and typically has a set of animations. There are methods to turn them in different directions, and to move (currently backwards and forwards), and to run the animations. As described with the events, there are now also special mover events that call these methods with different parameters, and the methods are responsible for iterating all actions for all active mover instances to allow for simultaneous control of several kinds of characters.

```
     class Mover:
     def clone(self, direction):
     def declone(self, direction):

     #selections and removals
     def select(self):
     def deselect(self):
     def remove(self):

     #actual actions/methods/functions .. things that can do!
     def turn(self, direction):
         self.turnings = {"left": "self.turn_y = self.turn_y + self.step",
                          "right":"self.turn_y = self.turn_y - self.step",
                          "up": "self.turn_x = self.turn_x + self.step",
                          "down": "self.turn_x = self.turn_x - self.step",
                         }

    def move(self, direction):
    def animate(self, motionnumber, todo):
```

There is a known liability in the command pattern, that the event framework here resembles, namely "potential for an excessive number of (..) classes" (Buschmann et al. 1996) (p. 289). Of the suggested techniques for solving the problem there, equivalents of grouping, unifying and having pre-programmed macro-command objects are already a part of the event framework here. Additionally, to facilitate adding several simple events that manipulate the movers in straightforward ways, like turning them around or moving backwards and forwards, some general mover events were made. MoveMover and TurnMover take directions (like "forward" or "left") as parameters, and can hence be instanciated for the different needs without extra classes. CommandMover is used to instanciate events that call given methods of movers. Examples of their use are shown later in the setup file.

```
class CommandMover(MoverEvent):
class MoveMover(MoverEvent):
class TurnMover(MoveMover):
```

5.4.3 The Animation System — working around GameBlender peculiarities

Combining the references to the actual animations on the Blender side, instanciated as CharacterAnimation objects within movers, and the special mover event for controlling animations (AnimateMover), this animation system is probably the most complex construct in the project. Changes to it had to be made quite briefly before taking into use, and rationalising / refactoring might well be in place.

One factor causing additional need for complexity was the need/want to re-use the so-called *action actuators*, that must be used to run the animations in GameBlender, for multiple different animations. As this is probably the most advanced (although still not anything fancy) hack around the limitations of GameBlender, and perhaps also a teaching lesson or at least an explaining factor of the implementation, it is probably worthwile to discuss the solution here in a bit more detail.

The idea is that for each (3d/visual) object, there may be any number - preferably a large number, to allow for flexibility, improvisation and richness of expression when performing — of animations i.e. *actions* in Blender terminology, which manipulate the object. In the Blender game engine, there is a system of sensors, controllers and actuators — called Logic Bricks (?) — that can be used to make interactive applications even eithout programming, Python controllers being a special case. There, an *action actuator* are often bound to a certain action, so that calling a particular actuator activates that action. The programming API, however, does provide means for changing the target action of an action actuator. Also, there is a method for directly controlling the animation via programming, but unfortunately it does not work (is either not implemented or broken). An alternative way is to set the action actuator to be so-called property driven, meaning that the actuator sets the animation frame based on the value of the given attribute of the object in Blender, which can be set via the Python API. However, the engine is made so that every actuator can be used only once at a time, per cycle. Therefore, having multiple concurrent animations for an object, of an basically unlimited set of possible animations, is not trivial.

The solution here is that for every mover-to-become, i.e. for every object in Blender to be controlled via this system, there must be as many action actuators as concurrent animations/actions are needed. For the initialisation to work, the number of those — called *animation channels* hereafter — has to be stored in a property of that object (though it might be possible to get their number by analysing the list of all actuators for that object, that is something to study). When a mover is 'invoked', i.e. added by an addObject actuator to the visible layer, that preparation method gets the references to the given number of animation actuators. Then, when an animation is activated (by the AnimateMover event for all active movers), the CharacterAnimation classes 'begin' methods (analogous to Event.start, but named differently as an animation is not an event) which are instanciated in the movers in a particular order, reserve the first available animation channel and start running it via that actuator (after it has been told the wanted action name). When an animation is stopped, its channel is freed.

```
class AnimateMover(MoverEvent):
    def start(self):
    def act(self):
    def stop(self):

class CharacterAnimation:
    self.position = self.length * self.mover.scene.time.code

    clonephasing = self.mover.scene.vertical.value
cloneposition = self.position + (clonephasing * clonedistance)
```

5.4.4 Clones

A special feature of the mover system, and the mover events discussed above, including the animation playback component, are the clones. They are about having multiple copies of the same character visible in the same scene at the same time. During the implementation of this first prototype, the author hesitated whether to have them all as separate mover instances, or somehow included in a single one. After experimenting a little with having separate ones, the solution where they are included in a single mover instance was pursued. This meant that all events that deal with movers must handle the clones as well, which can certainly be awkard, but opens up interesting possiblities too, as demonstrated by the character animation system which was presented in the previous part, and is now further examined here regarding the clones.

So clones are copies of the same model, with the same animations etc. In this first prototype, they are structured so that there or so-called *horizontal* and *vertical* clones, which appear to the side or above/beneath the original object accordingly. Furthermore it is made so that when adding them, they appear one by one on opposite sides, that is: when adding the first horizontal clone, it appears on the right side of the original object, then the next one appears on the left, the next again on the right etc.

When a character animation is active for a mover, the position (frame) of the animation is set according to the time code in the system (more about the time system below). Normally, this means that animation is at the beginning when the relative timecode is 0, and it is linearly played back so that the animation reaches end (last frame) when the time is near 1. However, for the animation playback of clones — i.e. additional copies of a model in the scene (mover), the behaviour was modified. An extra variable, called *clonephasing*, was introduced, an initially controlled with the 'vertical' slide (mouse y coordinate in the first setup). This variable is used so that the current position in the animation (frame number) is increased by the (tulo) of the clonephasing variable, and the so-called *clone distance*, which is simply the order number of the clone. So the effect is that when *clonephasing* is set to 0 by the user (i.e. the mouse y coordinate is 0), all clones animate in exactly the same phase as the master mover. Then, when the variable is risen to e.g. 0.1 by sliding the mouse upwards, the clones that are next to the original one are a little ahead, and the possible additional ones are more so etc. This is illustrated in the figure.

This mechanism would obviously need tuning from a mathematical standpoint, but already proved interesting to play with and useful in performance. And from a theoretical standpoint, it gave rise to an interesting high-level variable that may be useful in VJ tools: the clonephasing, discussed above, is in a way the amount of *difference* there is in the scene. This simple but powerful notion was derived from the studies in improvizational systems, and theoretical work about dance improvisation and cognitive sciences, including gestalt theory (Solano 2004). So the otherwise perhaps unelegant solution to handle clones as special cases by the so-called mover events, i.e. the components that drive all actions involving the animated characters, including the character

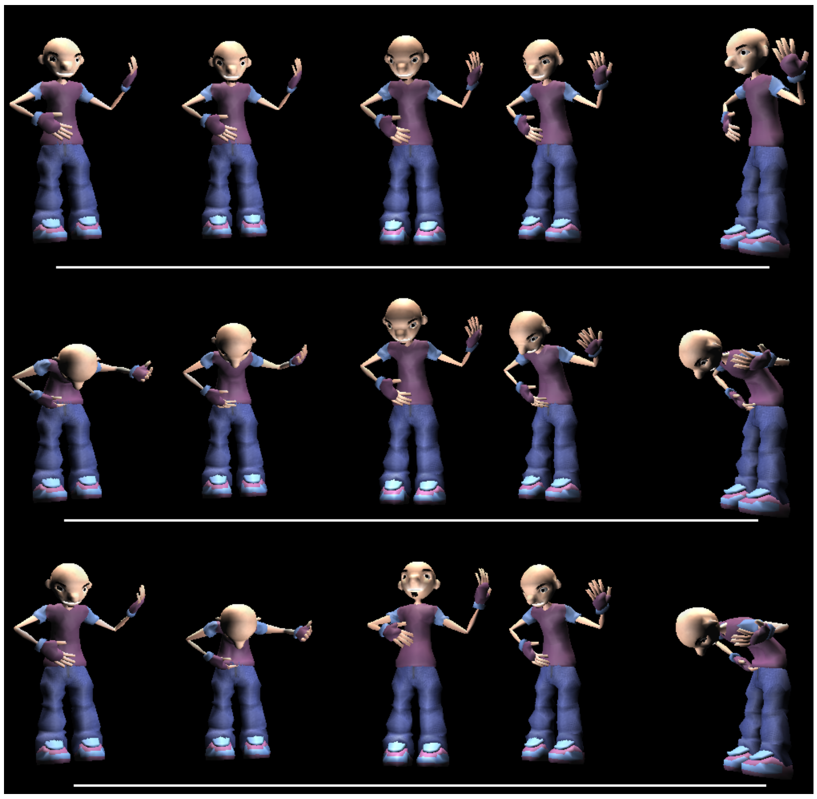

Figure 3. The same composition with three different values of global difference, resulting in the animations of the clones being in same phase when the difference in minimal (topmost image), with slightly increasing difference when the variable has a small value (middle image), and in radically different when set at maximum (bottom image).

animation component, lead to this implementation where the phases of the same animation in different clones can be modified, and to the the overall notion that *difference* could be an interesting high-level user-controllable variable, which different functions in a performance system could hook to various effects. This notion of difference can perhaps be seen as the counterpart of the central concept of *similarity* in gestalt theory, where other factors include *proximity*, *closure* and *simplicity* (Wertheimer 1923). The potential in applying the theory is addressed in the evaluation of the results in this work.

5.4.5 Direct manipulation of bones

Besides the playback of pre-made animations, which control the bones in the skeleton according to the data derived from either artist-made or motion captured animation, the direct manipulation of bones in real-time was experimented. A bit surprisingly, the otherwise limited Blender game engine provided a straightforward means for that, via the setChannel method of the action actuator, which receives a 4x4 matrix defining the new parameters for an individual bone. In this first test, a new mode called JuggleMode was added as an event, in implemented to that when active it sets the position of a certain bone freely based on the current mouse position. In the default setup, that mode is active when the space bar is pressed, and the bone to move is defined by giving its name when configuring objects for the system. This simple mechanism is not really useful, but proved that the system works, leaving better ways to use it open for consideration.

5.4.6 Controllers

Apart from the event-system, scene, movers and animation runners, some initial work was made for what could be classified as controllers in the sense that they output values on which e.g. the animations may depend (perhaps the term 'modifiers' could describe these too).

For one, there is a simple abstraction for controllers, of which the only implementation so far is an as simple model for slides, which only have a value from 0-1. In the prototype, there are two such slides capturing the values from mouse coordinates, called simply *horizontal* and *vertical.*

```
class Controller:
class Slide(Controller):
```

In future development, the way controllers are bound to scenes should be revamped. One way would be to make a dictionary Scene.controllers, where the different controllers would be accessible by names, and could then be better configured by the user in the setups. A good opportunity for this refactoring is when adding support for MIDI-controllers.

5.4.7 Time

As moving to the music was a central goal in the development, targetting the use in dance events, the RhytmMaster was made. The first implementation assumes an even beat, which was known to mostly be the case in the first event. There are methods for inputting beats and 'tahti's, based on which the tempo (beats per minute, bpm) is counted and can be given. The animations are bound to this information by default.

When considering more complicated differently structured, and arbitrary, rhythms, a question rises whether to extend this system or just rely on external output from e.g. a MIDI sequencer. The deciding factors are not very clear yet, but there probably are potentially interesting things that can be better done with this internal RhythmMaster. For example, changes in rhythm could be triggered by certain conditions in the scene (although things like that could of course be communicated back to an external sequencer too). Also, it should be noted that Blender already includes powerful time-curve features in the interpolation (IPO) module, that might be usable for shaping rhythms. At this point, however, all that is left for future work.

A special TimeCode class was made for driving the animations (and could be used for e.g. video, if not music, too). This time code is designed for looping items, and has the value from 0-1 (beginning to end). A TimeCode instance is always a part of a Scene, as scene.time , so the current timecode is accessible as scene.time.code to anything that has a reference to the scene, and used this way by the animation components.

In the normal play-mode it adjusts the phase according to the durations given by the RhytmMaster, but there is also a 'scrub' (as it's commonly called in video, similar to 'scratch') mode where the timecode i.e. the phase of animation playback is mapped from a controller, currently the horizontal slide i.e. the mouse x-axis. This is similar to what the author had implemented earlier with PixelShox (in JavaScript), as mentioned in section 2 / related work. The current TimeCode class is suitable for playing back and controlling short looping contents, such as single character animations and video clips, but in order to support long spanning narratives in the future some other kind of time constructs will probably be needed. Also, if more complicated rhythms are to be supported by the RhythmMaster component, this TimeCode class must be re-designed and -implemented as well.

```
class TimeCode:
    def update(self, timer):
        if self.mode == "play":
            self.step = self.totalcyclelength / self.scene.rhythm.getLength()
            self.code = self.code + self.step
            if self.code > 1:
                self.code = self.code - 1

        elif self.mode == "scrub":
            self.code = self.scene.horizontal.value
```

This way, the basic goal of being able to 'dance to the music' with the characters, was reached.

An overall view to this resulting class library is shown as a UML class diagram in Figure x.

5.4.8 Camera control

Controlling the camera is a basic requirement for almost any 3d system, and a rudimentary solution was made for it here also. First, support for zooming was added, so that the user can modify the virtual lense of the camera during the performance. In the Blender game engine this is done by modifying the camera lense via a suitable IPO. Additionally, a feature was added to have camera tracking, i.e. the camera turn towards the target (e.g. if it has moved). Also, late in the development there was an attempt to add more camera controls, i.e. to have commands for positioning the camera relatively from the target, e.g. to have it in the front or behind the selected mover. However, all this was severely limited by the engine: it was found that due to the way the

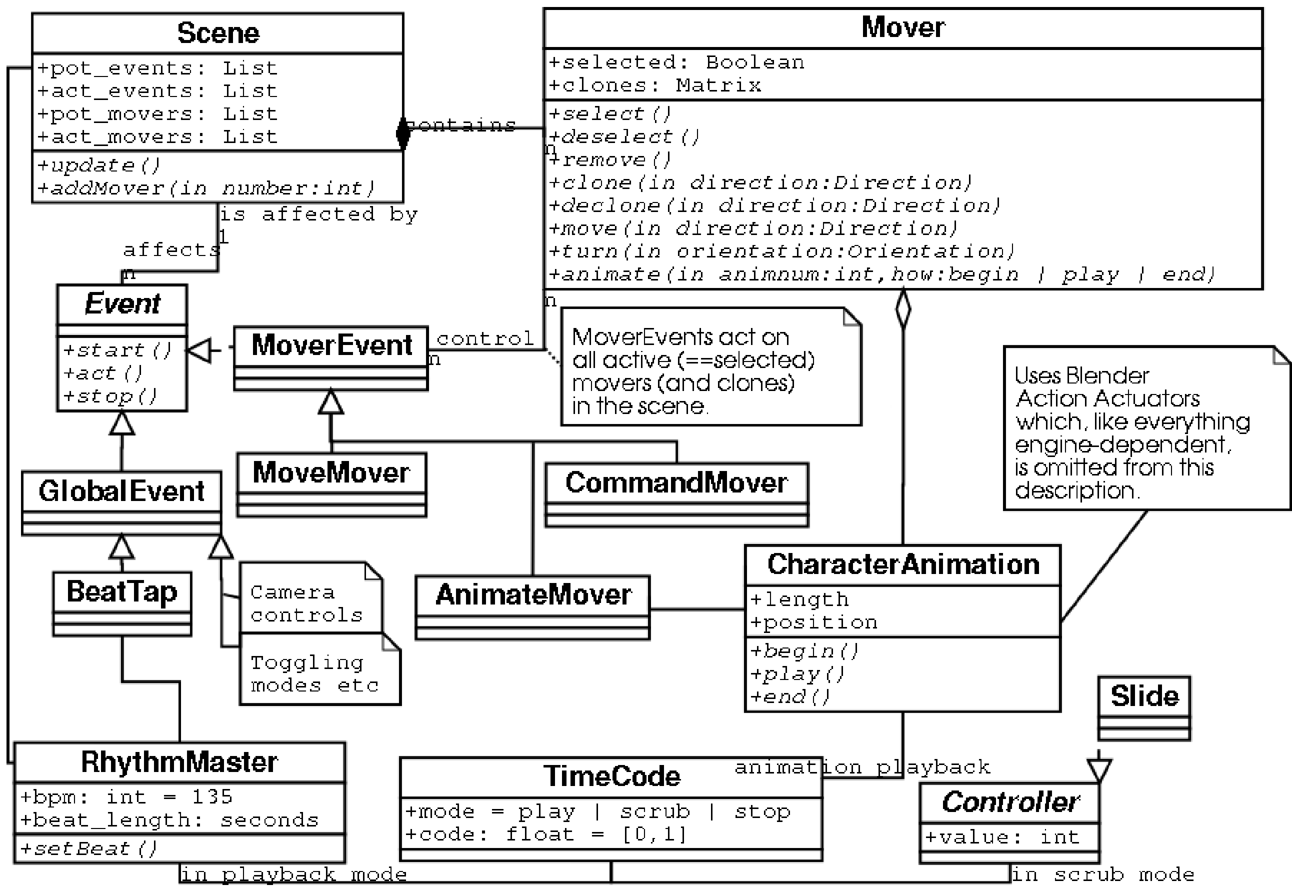

Figure 4. The original design and implementation as a UML class diagram

Blender game engine actuators worked, there was no sensible way of giving a selected object as a target parameter for the camera actions, e.g. tracking. So the camera controls where left very poor in the first implementation, which limited the use mostly for situations where the camera was stationary, and the movers positioned near the initial focus area. Later the author reported this to the new developers of the engine, and in 2004 the Blender game engine was changed to overcome that limitation. So now with little work these originally unfinished controls could be made functional, enchanching the usability of the tool greatly.

5.4.9 Missing elements

Several essential areas of functionality are missing in this first implementation. These include the utilization of the so-called non-deforming animation and other uses of the so-called IPOs in Blender, light control, special effects like fog, and collision detection and other physics simulation.

The term non-deforming animations refer to the type of animation where the shape of the objects remain unaltered (unlike in character animation), but the objects move in the scene. There are two typical ways of doing it that Blender supports: keyframe animation / IPOs and paths. With keyframes the positions of the object in certain timeframes are defined, so that one position is interpolated to the other when playing back the animation. Paths are invisible curve objects in the scene, which movements of the visible objects can be made to follow (e.g. so that an object goes from one end of a curve to the other in 100 frames, along the curve). So with these techniques pre-made movements can be defined, but the use of those was not addressed in the implementation, and is left here for future research. As an additional note, after the first use of this prototype the author became interested in group movements, formations. Interestingly the non-realtime animation side of Blender provides a quite peculiar tool for that, called dupliverts. Using them, objects are made to appear in the vertices (points) of another object. For example, using a hexagon object for dupliverts, some other object like a model of a human can be put to appear in each six points of the hexagon. Then when the hexagon is moved, rotated, deformed etc. the other objects move accordingly. This provides simple means for an animator to deal with group formations. The author has not investigated whether the mechanism works on the game engine side, but surely a similar one can be implemented with many engines.

In this system, there is no support for controlling lights during performance. Here of course the virtual lights in the computer-generated 3d scene are referred, not the actual lights in the place of performance, which are outside the scope of this system. In the first implementation, the use is limited to having the lights in the predefined positions (set in the Blender GUI). There is nothing extra difficult in it technically, i.e. lights are just objects that can be moved around normally, but there was no time to make controls for them. Surely, there are many interesting things to do, starting from basic effects, and not forgetting some classic techniques like having a spot-light follow a central figure in the scene etc. Also mere chances in the colour of light can be a powerful effect.

Configuring effects like fog or mist, or setting an background image for the scene, are handled via the world object in Blender. That has not been utilized in this system, again not due design nor technical reasons, but only lack of time.

Another lacking area is the utilization of physics, like having collision detection and simulated gravity. Combined with character animation, they can be really impressive tools, as demonstrated by the popularity of so-called ragdoll systems.

5.5 A setup file

Besides the actual content (models, animations, lights etc.) on the Blender side (made as mover.blend) and the class library providing the basic functionality (written as kyperjokki.py, will be refactored differently to a module), the third main component in this project was the setup file (ksetup.py in the 1st prototype). Unlike the class library which is meant for programmers only, the setup file is intended to be something that non-programming users could edit. Despite that, it is a genuine Python file and may include arbitary complexity. A practice in the development was to do experiments and special cases in the setup file, to be integrated in the library later.

There are several mandatory parts that a setup file for this system must include.

```
scene.movernames = ("Guy", "Snake", "Kenguru", ...)
```

Firstly, the setup must declare (as an attribute called 'movernames' for the scene) the names of the objects in Blender that are to be used as movers in this setup. The order in which the names of the movers are given is significant, as it determines their numbers which are used to active/deactive and select/deselect them.

To be animated, these objects must be so-called armatures, which is the Blender term for skeletons, and which have the actions with the animation data. However, a plain mesh (a 3d shape, with no armature) may be used as a mover too — it will just have no animations bound to it. Besides the armature system used here and in general for character animations, in Blender there is also the so-called IPO system (interpolation) for both non-deforming object animations and mesh-deforming vertex animations, plus it can be also used to affect many other things in scenes. In this first prototype, IPOs were used only to control the zoom of of the camera lense as a special case. As a future development, an elegant way to support IPOs in this system might be to enable defining a 'mover' with IPO animations similarily as the armature ones currently , just using the IPO block names and lengths instead of the action information. Fascinating challenges lie also in incorporating support for animated textures (by enabling the manipulation of UV-coordinates, which is well supported by the engine), which intriguinqly would bring us closer to having videotextures.

```
guy = kyperjokki.Mover("Guy", [("ABC", 12),
                                ("Guitar", 44),
                                 ...
                                ],
                          "chest",
                          scene)
```

Then, the movers must be instanciated. Here the order does not matter. The first argument is, again, the name of the object (mesh or armature) in Blender. As the second argument, a list of animation name and length (in frames) pairs is given. This determines the order of the animations, used for control bindings — the first animation/action name given will be the first animation for that object, activated for every active mover when the animation event is activated with the first animation as an attribute. The third argument is a single string, defining the name of the bone to be used when using the mode where an individual bone (on the armature) is used to manipulate the shape of the object. The final argument is the scene in which the mover is to be created.

Additionally, the set-up file may include any kind of event definitions etc., but that is optional, if only functionality implemented in the library is used. Currently, however, there are yet many crucial functions implemented in the set-up file that have not been moved yet.

The final mandatory element in the set up file are the keybindings. It is implemented as a Python dictionary (which again is a hashtable), which basically maps so-called keys to values. Here the keycodes (as evaluated by Blender) are used as the dictionary keys, the value consisting of the associated event instance and a constant declaring the keybinding type. Currently three types of bindings are implemented: WHENPRESSED, which causes the event be active as long as the key is pressed down but not released, TOGGLE which toggles the event/action on and off at every press, and ONCLICK which triggers the event only once when the key is pressed (and again when pressed again, of course). This design was pretty ad-hoc and grew with the implementation, but has served quite well.

One peculiarity about this mapping of controls and action-controlling events is the instanciation of events. As shown in the example extract below, it may be done either elsewhere in the set up file, or within the dictionary only.

```
keybindings = {#keyID, event, controlbinding type,
    ZKEY: (zoomOut, WHENPRESSED),
    AKEY: (zoomIn, WHENPRESSED),
    LEFTARROWKEY: (kyperjokki.TurnMover(scene, "left"), WHENPRESSED),
    RIGHTARROWKEY: (kyperjokki.TurnMover(scene, "right"), WHENPRESSED),
    TWOKEY: (SelectMover(scene, 2), ONCLICK),
    QKEY: (kyperjokki.AnimateMover(scene, 1), WHENPRESSED),
    BKEY: (countBeat, ONCLICK),
    PAD2: (setCameraFront, WHENPRESSED),
    }
```

To summarize, the setup file provides powerful ways to both include material, that is characters and animations for them, for a set, and an easily modifyable keybinding configuration. Also the users are free to implement any kind of new events here and add controls to them, given that they have the necessary programming skills with Blender Python. This system does not match the ease of use of other VJ tools with GUIs, but should still be usable for non-programmers too as they can modify the text file easily. Also adding a GUI for the configuration should be completely straightforward.

6 Evaluation

Here the constructed system is analyzed in itself, compared with other systems, as commented by peers and with observations from actual use of the prototype for the intended and other purposes.

6.1 Model / system (in itself & compared to others)

A basic evaluation criteria for a design science artifact / software construct is whether it works or not. As usual in software development, this is to be evaluated against the requirements, or acceptance criteria, set for it. Recalling the basic requirements selected in the end of chapter 4, and the system described in chapter 5, it can be seen that the implementation meets the basic goals. How well it does it, compared to a) other design possibilities within the selected set of technologies and b) other similar systems made with other technologies, is further evaluated here.

The evaluation here is, however, very limited. No formal analysis of the system itself nor controlled studies of the usage have been made. The analysis of the system is limited to the authors own observations, with only little observations from competent peers. Also the experiences from use are so far limited to the author's own, with only some preliminary observations by others.

Also, the more advanced evaluation criteria are unclear. True to the nature of prototyping and iterative development, a motivation for building the system was indeed to better identify interesting questions about how such systems can and should be made. Yet the unclarity is not even limited to the design and implementation choices of the system, but also the different potential purposes and ways of usage are open to discussion — the author holds the view that, perhaps a bit strangely, by making a system and using it, it can be learned what kind of systems and for what purposes would actually be interesting to have. So besides evaluating the system against the original criteria that were known when the development started, in this chapter attempts are made to also bring out the lessons learned while making the system about what requirements should be set for the different further developments and/or other systems addressing similar goals.

6.1.1 Comparison of the event system with others

As mentioned in the chapter overviewing related work, there are several kinds of existing event systems, for both similar and different purposes than the one presented in this work. Here, this newly introduced one is evaluated in comparison with closely related or otherwise relevant others.

Let us begin with a brief recall of the model introduced here. These so-called events were designed to map controls to actions, and implemented so that they may include much of the logic of the actions themselves (otherwise the logic is in e.g. the methods of the mover class, used to control the animated visual objects). Conceptually, they refer to basically anything that 'happens' *within the system*, meaning 'the virtual world' or scene, technically what is executed in the 3d real-time / game engine. So for examble, the 'happening' that the camera zoom changes, or that all clones of selected 'movers' arrange themselves in a circle around their originators, are events in this system. Unfortunately, this may cause confusion, as there are conflicting existing definitions and ways of use, as will be examined in the following.

Often so-called events are used to refer to controller inputs that the computer-based system receives from the outside, e.g. keypresses and mouse pointer movement. An example of this in the event system of Pygame, which is based on the cross-platform Simple Directmedia Library (SDL) (http://www.pygame.org/docs/ref/pygame_event.html). This way, all action may be controlled by any other means, and the event system is limited to handling control device input only. As mentioned when describing the design and implementation, a decision was almost made to rename the event class presented in this work to 'action', but as this implementation is so tightly bound with Blender where that word is already reserved for something specific and well suitable, (i.e. mesh deformation driven by the movements of an armature), the change was withheld.

Returning to examining the Pygame/SDL event system, there is the possibility of defining so-called user events, besides the default control device events. Each event type is identified with a numeric code, actually defined in the underlying library (SDL) that manages the operating system dependent interaction with the hardware, so that the user events must be assigned their own type with another number. Additionally, the newly created event types can be assigned members or attributes. So basically it may be possible to define Pygame user events that are similar to the events in this work, i.e. which would include the consequent actions in themselves. These special kind of user events could be perhaps called (and subclassed?) as 'action events' to facilitate their special treatment. This may be tried out if a new version of the system presented here is implemented using e.g. the Soya3d engine, which also uses the SDL event loop (and can use the Pygame event loop too), and/or if Pygame itself is used to perform with 2d graphics and video. Furthermore, regarding this relationship of the Event class here with the other event systems, and the possibility of conceiving the one here as ActionEvent, in the Virtual Object System (VOS) by interreality.org there actually is a class called ActionEvent used with the so-called Actors, that have been made for animation control purposes. The author has actually discussed the development of the animation systems for VOS, as they have not yet been implemented, so further evaluation of these event systems is all-in-all an interesting area for future work.

6.1.2 Cognitive, user centered and authoring issues

Apart from the technicalities of the event system, the key question when thinking its design is what it means for the users.

The authoring of more advanced and complex, or just larger, functionality was not addressed in the original requirements nor even in the wishlist, which does however mention 'set construction' as an area where functionality is needed. Examples of such authoring include making of more complex 'acts' involving several movers/characters that move and animate somehow relative to each other or other things, or making some sort of composite actions or sequences which e.g. activate certain smaller actions or events in certain order / at certain intervals. During the development of this basic system, the author got more interested in models for such compositions, and was even sketching UMLs of high-level classes all the way up to Manuscript, familiar from the scriptwriting technique used in theatre plays and films. But currently, there is no such support for an author in this system — just the barebones and hopefully flexible event system, that hopefully can be used to create interesting setups (or scenes or acts), or else must be redesigned accordingly. (A way how e.g. the concept of a manuscript, implemented as a class definitition, that further composes of a sequence of scenes, - (vrt. banks in videobank)

Another challenging way to look at the issue of authoring, both for the design of the system and the user, is the concept of Choreography in dance. That might lead to, besides the long-spanning highlevel abstractions that would probably appear in Manuscript after related concepts in screen- and playwriting, also to detailed descriptions of body movements, that are missing from the current system as well. Examples of this kind of detailed authoring would be defining deviations in the exact positions of a body part in a movement. Currently, the authoring of the character animations is mostly outside the scope of this system, as it is made for controlling the playback of the pre-made animations. However, real-time control of individual bones of the armatures, i.e. the skeletal models used for the character animations, has already been experimented and is preliminary supported in the current implementation (the so-called juggle mode, enabled when holding done the space key, maps the mouse movement to an individual bone in the active movers). Continuing in the spirit of experimenting with changes in the 'locus of control', an author might want to define events that affect the style in which the movements are made. So, for example, there might be ready made basic movements for moving limbs and other body parts (so-called poses in Blender), and composite actions (as they are called in Blender) such as walking, kneeling down, jumping etc., ready-made as currently. But additionally, there would be e.g. event types called SmoothMovements or NervousMovements, that would change an aspect of all movements that occur while being active, making them smoother or appear nervous in these cases. However, making such events is extremely complex as it probably requires calculating delicate 4x4 matrix transforms for the individual bones that would modify the original movements to match the desired style. Perhaps some basic methods, hiding the required mathematics, for doing this kinds of things could be added to the library to facilitate authoring of such real-time modifications of movements. Or, avoiding the complexities in real-time modification, support using a pre-made variety of different versions of the same movements.

So in this view, the class library of this system is to provide the basic building blocks that should be useful for further creating the building blocks for making such more advanced constructs as the ones discussed above. In a way, the events define a vocabulary and syntax that the authors can use. In fact, the design pattern called interpreter has been described as a way to create a (scripting) language for the user (Eckel n.d.a). If the current design fails to do that well — as it may, even though it did fulfill the basic requirements for the first prototype — it should be redesigned (and the program and library refactored) to better suite those future purposes. So the questions follow: how? And where to look for answers?

Interestingly, a key is the problematic notion of event itself. Concerning narratives, they can be defined as *sequences of events*, involving causality. On the surface, this seems to solve the problem: as all actions in this system are defined by events, and narratives are sequences of events, just make sequences of these events in order to make narratives. This, however, remains mostly untested with this system and is surely a much more problematic area when going to the details, which is left for future work.

On the other hand, instead of narration, dance and choreography open a different perspective to the potential ways of authoring with this kind of systems. There the means of composition and the relationship to the notion of event is also left for future research.

So what is left to say about this event system? Let us take the evaluation to something more concrete by inroducing a couple of examples of authoring with it. These were not thought of by the author when developing the system, but came up in discussions with a collegue afterwards, and therefore are more potential in pointing out weaknesses in the design:

A choreography for two, with collision detection with external actors: In this example, there are (at least) three actors, of which two participate in a certainly shaped motion (and are implemented as Movers).

A way to author/implement this motion, or 'choreography', with the current system would be to create a new event, e.g. ChoreographyForTwo, by perhaps subclassing MoverEvent (even though this would not be exactly like the current MoverEvents as we will see). ChoreographyForTwo would then e.g. take as arguments the Mover-instances that are supposed to participate in the motion, let's call them moverA and moverB. This would be done in the start method, e.g. so that the two first selected movers would be assigned, and the initiation canceled if no two movers were selected. The start-method of the event would then e.g. check the positions that the participants have in the beginning, if the upcoming motion driven by this event, i.e. defined in this choreography, would be relative to the initial positions. Then, as long as the event is active, its act-method would be called from the mainloop and it would move the two movers accordingly, either by using the existing move- and turn-methods, or by adding new methods for setting arbitary positions (which should be straightforward as Blender's setPosition should work in the game engine too).

So far, this example demonstrates the intended use of the library nicely: existing framework could be used to add this new kind of functionality. Even better, the architecture seems to suite it well, as the mover-instances do not need to be any special kinds not know anything about this new feature, and the movement caused by the choreography-event would be coherently driven by a single function, that could easily be paramatrisized e.g. so that the paths of the movements and the speeds could be controlled by the performer with e.g. sliders. But, to be honest, this far this example was presented by the author in the discussion, and only the remaining part brought up by the collegue shows some problems. As a sidenote, also support for the existing functionality for making such animations in Blender, such as object IPO curves and using regular 3d curves as animation paths, should be seriously considered for authoring this system.

The question posed by the collegue was, what to do when a third object (MoverC) is introduced that moves accross the scene, which is again pretty straightforward by making e.g. an event that does just that, say SceneCrossing(MoverC) (or TakeAcrossScene(MoverC) if they are thought more of as actions). But what if that third object can collide with the two participating in the choreography, resulting in e.g. the stopping of the participant and the new third object? This is not a trivial problem for several reasons. For one, the system does not utilize collision detection at all yet. This is not difficult *per se*, though, as the Blender game engine (and other potential ones that could be used) include collision detection, and even if they would not, it would be relatively easy to do for a simple case like this (just checking if some of the movers overlap). But the difficult question is how to handle the detected collisions, and how to affect the active events that are manipulating the movers that collide (and how to even know which events are actually manipulating which movers and how).

Continuing in the mind-set of making new kinds of events for everything that is needed, a CollisionDetection event (or again, DetectCollisions if thinking as actions) is conceivable (in the case of Blender, it should be made to receive signals from the collision sensors in the engine). What could it do about the detected collisions, then? Still in the event world, it might somehow be made to signal the other events about the collisions, so that they would act accordingly (in this case ChoreographyForTwo and SceneCrossing should stop moving the movers that collided). But at least to the authors mind, this seems awfully clumsy and overly complex, and might in practice prove almost impossible. Another, perhaps more sensible solution would be to add some state to movers or use the existing state (such as if they are active or not), make collisions change that state (e.g. deactivate the movers that collided, perhaps via a collision method), and the movement methods respect that state so that inactive movers would not move even if there were active events moving them. This of course depends on the overall requirements for collision detection with regard to the different events (and the games / physics simulation like realistic handling of collisions, that the underlying engines can usually already do, is of course quite another story - this is about trying to do something different with that information).

So by having collisions detected and handled by changing the states of the movers, this case where the colliding would result in halts could be implemented quite reasonably. But what if there is an additional change in the wanted behaviour (or should we say: a new event in the narrative, caused by the previous ones?), and the collision would now stop MoverC as before, but make the participating mover in the ChoreographyForTwo, let's say MoverA, change it's behaviour so it continues to move but restricts its movement to the area where it has not collided. As the motion of MoverA is still driven by ChoreographyForTwo, so the approach that was abandoned in the previous paragraph, where the collisions would be signalled to the relevant active events, would now be the straightforward way to achieve this functionality within this framework. Another way might be to further develop the internal states that the movers have, and the ways they are taken account by the methods used to move them, i.e. to enable having restrictions in the areas where they move and somehow adjust the motions accordingly. But that would easily lead to awkward situations, e.g. when the Choreography event thinks it is still moving the previously collided mover somewhere, when it is actually elsewhere and moving differently. Yet another approach, and one that should be given serious thought, is to think the whole design of the system over. Obviously this example event/choreography/narrative is not necessarily a representative one, but it may well exhibit some of the interesting challenged that lie in the authoring of computer based animation. Certainly one thing to do for future development would be the search the literature and construct meaningful test cases.

In conclusion, the author is leaning towards the decision that this complex notion of an event should be discarded. Instead, the better known and more straightforward notion of command, deriving from the pattern with the same name (Gamma et al. 1993), should be used. A strong argument for this is that the term event is commonly used for a different, but confusingly similar purpose, widely. A command is straightforward: after all, these are what the user tells the system to do. Also a very strong argument for this change arises from a simple analysis of the so-called events that the author made for the first experiment: many of them are named like commands, not like events! Considering names like TurnMover, AnimateMover, countBeat etc. this seems clear. Somewhat surprisingly, the semantics of the Command pattern and the event system here are not so different in the end, even though they were initially observed as coming from different angles. Surely, these events-changed-to-commands would not be quite ordinary: as noted when descriving the design, they have special starting and stopping procedures and may have very long-spanning durations (even being always active), and include complex structures like a state-machine in their execution. Yet, already because of the confusing naming overlap with the Event class, some derivationg of Command would be at least less bad.

Related to this, the separation of the current event-management to a different component, perhaps according to the CommandProcessor pattern (Buschmann et al. 1996) (p. 277), should be considered. This should be done especially if it facilitates developing new features to the scene class to support more interesting movements and/or having a visual representation for it (an environment, or at least a background-image). Of course controlling the event-execution directly from the scene does not prevent adding features to it, but by separating it to a different component making that single class too complex could be better avoided.

Finally, as noted already along the presentation of the current design / implementation, the special powers of the Python object model should be utilized when implementing the event/command/action system in that language. That is, on the one hand, that there's a remark that the command pattern (as used in C+ or Java) may be redundant in Python, where all functions already are objects too (Eckel n.d.b). And on the other hand, vice versa, that all objects can be made *callable* (like functions and methods are), as demonstrated by the Freevo2 event-driven menu system, where the menu items have possible actions and are callable, calling them resulting to executing the default behaviour (Meyer 2005). How to utilize that here is left for future work.

6.2 Usage procedures

A way to evaluate the construct is to examine the procedures required for using it. Firstly, using this version, only the Blender game engine is supported, so all data (models, animations, scene setup) to be used in a performance must be in Blender. Secondly, it must be configured in Blender in a certain manner. Finally, the setup has to be made to define what material is in use and how. In the following, each of these steps is reviewed.

6.2.1 Having the Data in Blender

As described when presenting the design and implementation, any Blender object (i.e. a 3d object) can be 'made a Mover', i.e. specified as a target of control for this system. The data can be either created in Blender from scratch, using the modelling and animation tools it provides, or either partially or fully imported from other tools, or motion capture data etc. Based on the usage of this system, Blender appears to cover the required procedures for preparing a set of data for a performance well. As the future of its real-time engine is uncertain, and it has shortcomings in both features and programmability, a move to another engine might be needed. In that case, it would seem now desirable to support having Blender as a preparation tool anyhow (*composing*), even if the actual performance tool (*instrument*) would be on another basis.

Yet a remark on preparing content (for performances): creating 3d animations generally takes a lot of work, and it may be hard to achieve expressive results. Very little in this work is actually related to 3d animations themsevels, but more on general controlling logic of anything, and especially (looping) time-based content. So for continuing work there are logical steps to take in two directions: (1.) try the system with other kinds of content, e.g. 2d objects / characters or video (the latter is possible with the animated textures even in the Blender engine, and has been examined with other engines, see http://studio.kyperjokki.fi/engine/VideoTextures) (2.) develop features that support better use of 3d animations (the initial attempts to use BVH motion capture data failed, but new support for it has been added to Blender since and it might already work).

This difficulty in having (large amounts) of quality content brings us to an interesting cultural phenomena, that has so far been neglected in this technology-centered work, and was not explicitly realized when developing the system. VJ culture, much like DJing before it, is often about so-called sampling, i.e. using short pieces of existing work made by others to compose the own whole that (in this case) is born live in the performance. In VJing, these 'samples' are simply short video clips, often taken from well-known TV shows or films, or even commercials or news broadcasts, which are then put to a different context. Now, sampling audio or video is easy. But re-using 3d characters and animations in this way is not common at all, may be technically difficult and probably questionable also legalwise. Perhaps we will see interesting developments in this area in near-future, and who knows what is happening in some underground scenes somewhere, as in some subcultures popular well-known figures, like game characters, have been embodied as 3d objects with animations for long. Also, one VJ gig in the Koneisto 2004 festival featured using photos of Finnish politicians faces as textures of the heads of simple block 3d characters, combining the ease of sampling of 2d graphics (there photos) with the use of 3d models. More elaborate discussion on this is outside the scope of this work, but the author is looking forward to learning more about it. For studies covering the cultural side too, see http://vj-book.com/ .

6.2.2 Configuring Blender objects for Mover

Besides the actual creation and/or importing of the needed data in Blender, certain configuration must be made for the controlling system to work. These steps are described here briefly, also to allow their critical examination and identifying possibilities for improvement.

For one, each Blender object (mesh or armature) to be made a mover, must initialize itself in a particular way. For that, a so called property *sensor* is used to run the initialising script (initmover.py). This occurs when that object is added to the active layer, i.e. showed. The initialisation registers the object and calls the invoke method which prepares the animation control system etc. So called cloning, i.e. adding several instances of the same object (which are all actually copies of the original one, that remains hidden on another layer), is also detected by this initialisation script and handled as special cases to e.g. position the clones correctly. So the user must make sure that calling this initialisation script is made correctly, for every kind of object used. This would be unnecessary on more freely programmable engines, where the execution of needed procedures could surely be made the default action on all added objects without the user needing to configure anything.

Similarly, the *actuators* that must be used in the Blender game engine to execute almost any action, must be added to every object. The author is unaware of any possibilities to add actuators by programming, without extending the Python API of Blender. Currently, minimally three are required, and they all must be named specifically for the controlling script to be able to find the references to them. These are a so-called motion actuator named “move”, a so-called IPO actuator named “ipo” for different manipulations and an edit object actuator with the action of deleting the object, named “endobject”.

Additionally, for armatures which have actions for animations, as many so-called action actuators must be made as are to be used concurrently. Also, this number of action actuators must be declared in an integer property (attribute) of the object named as “maxacts” (for movers without actions, this must be set to zero). The action actuators must be given names in the form for “animatesn”, where n is the order number of the actuator, up to the amount defined in the “maxacts” property.

Obviously, all this would be unnecessary in a different system, that enables the program to do all necessary initialization for itself. However, within the limits of the Blender game engine and the strict deadline for the first prototype, the solution is not without merits. As described when presenting the design and implementation, the control system achieves the goal of supporting any number of animations and using several of them at the same time both on a single object and for several of them. Compared to a system where the ’composer’, or the person / user making the setup, would e.g. have to create and dedicate animation actuators for every action separately, this is much more elegant especially when taking into account that control mappings (e.g. keybindings) would then probably have to be defined separately too etc.

In conclusion, it seems justified to say that the system that was designed and implemented achieves the level of usability and flexibility/power required. Users should be well able to use it without needing to know about programming, and not having to specify any such logic. The unfortunately necessary steps described above are straightforward and similar for every object, so the required configuration should be easily put in place by using existing ones as examples. If this implementation, using the Blender game engine, is to be developed further, there are possibilities for enchancements by e.g. extending the API to support creating actuators. Also, as noted when describing the design and implementation of the Scene class in this work, a Scene class has been recently added to Blender itself, and could probably be used to automate many of these initially manual steps.

6.2.3 Making the setup

As described in the chapter about design and implementation, the setup file with the definitions of the movers and keybindings is meant to be editable by the users. There one must write the names of the objects to define their order for keybindings, and again defining their motions with the framecount and the name of the bone to be directly manipulated. The keybindings are defined by mapping control codes of the keys to Python dictionary (hashtable) entries, which include either references to instances of the special event class or instanciate them in place. Such arcane activities are of course not at all what is expected from a modern user interface. But the goal here is different: not something for anyone to use, but a specilized tool for those who know what they want, and need extreme flexibility of it. Therefore the setup file is a genuine source file, Turing complete one might say, and the system sets basically no limits to what can be programmed there. Security has not been a concern, as the first prototype was a standalone application on a single computer, with no use of networking, and the planned networking is with trusted peers. The aim is that engine specific implementations and general mechanics could be isolated in the library, leaving the focus with the setup file to artistic concerns, without sacrificing any power of expression.

On the other hand, as Blender itself has taken steps to allow making interactive 3d, e.g. games and art, by not-programming, with the use of so-called logic bricks, this system could also try to take more advantage of that (instead of only trying to find ways around the limitations set by that system). For example, as the game logic editing in Blender already has a nice interface for defining keybindings, that could be perhaps used somehow. Elaborating on that example, a goal (yet unused, but the implementation supports it) with the way of defining keybindings in this system was that they should be easily redefineable anywhere, so that e.g. a special mode could be defined for even a single movement of a particular kind of object. Not compromising this, but achieving better integration with the existing GUI in Blender, would be an interesting future challenge. And as mentioned in the chapter on design and implementation, the controllers (especially if/when MIDI support is added) should be refactored to something like the keybindings mapping dictionary.

6.3 Use of the prototype

6.3.1 VJing In Time Tunnel X

Basically the first use of the prototype, the event which set the deadline for this phase of development, was a success in the sense that the system worked: it was usable and did not crash or otherwise misbehave, and the content made for it contributed to the overall performance. Of course, there were and are several issues that must be critically observed.

Firstly, the system was (and is) yet too cumbersome for a satisfying solo performance. This was acceptable and even planned for this event, where we performed as a group and actually each unit had the same limitation, but is a serious limitation anyhow and should be set as a target to overcome in future development.

A contributing factor to this is that there is no mechanism for dealing with the scene, or backgrounds, just the actual objects ('movers') on a black screen. This is suitable for a group performance, where the output of this system is mixed with others, and one of the best moments in the event (for the author) actually was being able to control the own character in a video (pre-captured from a video game). Anyhow, dealing with scenes, e.g. having props in the environment where the characters are, and/or using background images (or videos, of the engine supports) are clear areas to work on.

Another way to facilitate solo performing, and more fluent and dynamic use in general, can be drawn as a lesson learned from the earlier work with PixelShox (mentioned in the chapter on related work). There the real-time actions are typically turning on/off, or switching in between, so-called effects (besides controlling them). Those 'effects' in PixelShox are typically whole scenes that are visually interesting right away, so they can be simply activated in a performance, and then be worked with via the controls to suit the situation. In the first use of the system studied here that was not possible, but a basic configuration had to be first put in place by the performer 'off-line', not in public, before it was in shape to show — then it could be worked on further in the live performance. The basic system should allow for such presets, i.e. having a functions (made as an event encapsulating the required actions, mapped to some control) that e.g. first empty the scene and then add certain objects to certain positions, set their orientation and other state the correct way etc. This is, however, yet untested and of course also an area with unlimited possibilities and amount of potential work (in e.g. developing the logics of the behaviours).

So a main criticism, due to these limitations, of the first use of the prototype was the unability of the performer to react. If the output of this system was shown in public, no dramatic changes to its scene could be made, as it would have required manually working with intermediate situations that would have been not suitable for a show.

Secondly, the content was limited in both quantity (amount of objects) and the quality (the expressioness of the animations). This is not totally unrelated to this study of the technical system, as it currently supports only 3d objects in Blender, animated using the action system which uses the armatures. Making such content, especially in the large amounts sometimes needed for gigs lasting even six hours, is not easy and not much such content is readily available. The possibilities of using 2d graphics and animation and video footage, utilising motion-capture data, and the controversial potential in the so-called sampling of 3d objects and animations, as mentioned in the evaluation of having the data in Blender, are some ways to deal with this difficulty in the future.

6.3.2 Peer Review: Perceptions of Fellow VJ Application Developers

As the program is open source, a way to evaluate it are critical comments from interested parties. A window of opportunity for this opened on March 16th 2004, when a person using the pseydonym 'pildanovak' posted a message about 'Blender as a VJ-ing tool' on the Blender development forums (http://www.blender.org/modules.php?op=modload&name=phpBB2&file=viewtopic&t=3202). He had made a system himself, and asked if there are others with similar ideas and interest for developing it further. Some other people reacted with enthusiasm, and told about their work in different areas (midi control modules and VJing setups). The author of this work participated in the discussion, mentioning the development of the software construct described here. 'Pildanovak', who can be regarded as a competent judge having developed a system for similar purposes with the same tools himself, said: "i've tried the kyperjokki files, they work fine, and the code is nice written. the operations done with the objects are nice, but i'm not sure if this kind of operations is universal enough. Thanks for sharing." Later Pildanovak released a new BlenderVJ tool where any Blender scene with animations can be made MIDI-controllable, and the author of this work got the idea of fitting Mover to be a special type of scene in that system.

6.3.3 A remark from a Puppet-maker

On an occasion during the winter of 2003-2004, the author had a chance to demonstrate the system briefly to a full-time professional puppet-maker. A main reaction was to the cloning functionality, as crafting several (especially very similar) physical puppets by hand for group scenes is not so rewarding. On an afterthought this is not surprising, as it truly is an area of functionality where computer-based systems excel, and also in movie-making computer generated scenes have been used largely to create mass scenes. Furthermore, there a lot of work has been put into creating variety, in the form of random deviations in structure or movements to create an illusion of a social crowd of independent actors. This is surely something to keep in mind when identifying areas for future development.

6.4 Use for non-performance purposes in the Wireless Airguitar project

In parallel to this internal development project, targeted initially at live performances using large screens and powerful computers, this work was related to another project developing applications and content for mobile phones. That project was called The Wireless Airguitar, the Airguitar World Championships organized by and during the Oulu Music Video Festival.

There a part of the project was to make animated characters play airguitar so that they would be visually suitable for the small displays of mobile phones. Therefore a low-polygon model with clear characteristics was made, and animated with guitar-playing movements. The same model was actually used in the performance too, but with different animations (as in that event there was no guitar music), taking advantage of the cloning function to utilise the larger display area (as the more powerful desktop computer easily allows animating several such characters simultaneously). This contributed to the ideas about combining both public displays and mobile usage in the same system, elaborated in the coming subsection about sketching the First and the Last.

Later in autumn, small animations with the airguitar theme were made for mobile phones, using that model and the system presented here. This performance system actually did not support making those animations — the author just set clones of the model suitably and animated them with the performance controls (taking advantage of the possibility to shift the animation phases of the clones), and captured the frames as images to be packaged as an animation. The built-in sequence editor of Blender could have been used instead, and probably using this tool was practical for composing and recording animation only because the author was familiar and comfortable with the tool. However, as research done elsewhere (Boulic et al. 1993) suggests that adjusting animations in real-time is beneficial in making animations too, and this system has been designed with the possibility of recording, editing and playing back the events in mind, there is potential in such usage. A game-to-IPO utility exists for Blender, that can be used to bring animations from the realtime engine back to the animating side.

6.5 Component re-use

As the limits of the Blender game engine were known, and the applicability of different engines for different purposes and in different environments recognized, a goal in programming was to have reusable components that could be used with other engines too. For the core functionality, this is possible by having high-level logic that does not depend on the interface to the engine that actually executes the action. Same goes for the setup configuration. These are implemented as external Python modules, that the Blender project uses, so they could be used by other engines too. Mostly only Blender specific sensor readings and object initialisations are within the Blender project file, unaccessible for other applications. On the content side (models and animations), reuse in other engines is enabled by converters (import&export plugins), which exist for several engines already and can be written for basically any.

However, akin to the agile programming principles a) do the simplest thing that could possibly work and b) 'you aint gonna need it', no (or very little) extra work was done for having the program actually work with other engines, as using it with the one specific engine was the definite primary goal in the stage of the development looked at here. The idea is that when other engines will actually be needed, the required modifications can and will be done. And if that anticipated need never concretisizes, no work has been wasted in preparation and most importantly: the program has not been made more complex in vain (possibly to the extent of not having it work at all).

Yet, several components could straightforwardly be made engine independent to begin with. These include the controls-to-actions mapping Event classes, the Control abstraction and the RhythmMaster and TimeCode used for animation control (could be used for e.g. video too). Therefore, they could be used in other projects with other engines straight away, so that enhancements in e.g. time code control would apply when using in Blender, too.

The state-encapsulating Scene class, the Mover class and especially CharacterAnimation include both Blender independent and specific attributes and methods, the latter being about using GameObject properties and the GameLogic actuator mechanisms. For reusing these classes across different engines, the common and specific parts should be separated.

6.6 Review of the research questions

To conclude the evaluation, it is now time to return to the research questions presented in chapter two, to see if and how they can be answered.

All in all, to address the main question, this work presents one solution to the problem of making a system for controlling 3d character animations in live performances. With respect to the conditions set by the time and people resources, the use of a ready made, stable or at least relatively well-known 3d engine with support for high-level, or agile, programming made the development possible within a few weeks by one person. Also the lightweight software process, inspired by the extreme and/or agile programming movement, enabled rapid progress, while guaranteeing the delivery of a working product (as one was 'released' at least by the end of the day, every working day on the project). Furthermore, the use of a real-time engine that is included in the content authoring suite used, namely Blender, made it straightforward to create and add new models and animations to the system, and to modify them afterwards.

Regarding the subquestions, firstly, the system supports controlling many things at the same time via the unconventional notion of 'events' (for the lack of a better word so far), referring to the things that can happen (potential events) or are currently happening (active events) in the virtual scene. The users are expected to author the setups by programming such events, that can affect any aspect of the scene and many things simultaneously. The special so-called mover events are used for controlling the movable visual objects, which in this case are typically equipped with skeletal character animations. By default, the mover events affect all active movers in the scene. Thereby, they can be used to control several things at the time. Additionally, so-called phasing was implemented to the movement of the so-called clones, i.e. replicated movers, so that the user can control with a slide (currently mouse y axis) whether the clones animate synchronously or differently. Interestingly, as reported with the description of the clone phasing component, that mechanism can be seen as utilizing the notion of *similarity* in gestalt theory, which proposes also other high-level concepts for perception of forms and structures, such as *proximity* and *closure* (Wertheimer 1923). Based on that, similar dynamic structures that emerge as perceived closeness

/ nearness, or beloning / not-belonging, etc. in the scene could be built, and controlled by the perfomer, potentially achieving great expressiveness. Gestalt theory has become a modern classic, and been applied in several areas, including graphical design of software user interfaces and also machine vision, so there is a healthy amount of literature to draw from when applying it in software (Moore & Fitz 1993).

Secondly, moving to the music was implemented by having two special supporting classes to be associated with the scene, namely TimeCode and RhythmMaster. This time model was targeted for short looping content, such as individual movements in character animation, the model being simply that the range of the timecode is from 0 to 1, from beginning to end. The RhythmMaster can be given the tempo of the music in beats-per-minute (bpm), which is a common format in dance music, and also be fed with beat messages, which are triggered by the user in the first implementation by tapping a key, but could as well be sent e.g. from an automated audio analyzer. Based on that information, the RhythmMaster adjusts the timecode accordingly, which the animations then use for determining their playback. Also, controller devices (via the controller/slide abstraction) can be used to determine the timecode directly, overriding the use of the RhythmMaster (in the current implementation the tabulator key toggles this mode, and maps the mouse x axis to the timecode when enabled). These ways enabled 'moving to the music' in practice adequately in the dance event where the first implementation was used. Also, as there is nothing character animation specific in the timing, this model can probably be well reused with other kinds of content, perhaps when having video loops is implemented in the system.

Additionally, outside the core requirements but coming from the wishlist, logging the sessions (as inspired by Arkaos) is supported by having all action being driven by the aforementioned events. To store and replay the use of this system, it should suffice to save the information about the starting and stopping of these events, and all the values of the controllers. This has not been implemented yet, but as all the events must either subclass or instanciate Event, it can be done afterwards by adding the required code to that superclass, and to the component responsible for dealing with the control values.

7 Future Development

Practical work to be done is presented in two ways in the following. First, the short-term plan to refactor the core is outlined. Then, different directions for long-term future development are identified.

7.1 Refactoring the core

The evaluation of the so-called event system concluded in a recommendation to refactor it. To begin with, it should be conceptualized at least a little differently. As for implementation, the required minimum change is simply renaming the abstract base class (currently Event) and the implementing subclasses (such as SelectMover, TurnMover, ScrubMode and BeatTap). As noted in the evaluation, feasible possiblity is to try simplyfying the idea behind these so-called events and resort to the more straightforward, well-known and hopefully easily understandable concept of *command*, and to some extent the design pattern that goes by the same name (Gamma et al. 1993).

In doing this, however, the richness of the concept of event, and related concepts, should be embraced. For this to result in meaningful structures, the relevant literatures — largely in humanities, the author supposes — should be consulted. Also, closer to the practical programming, lessons learned from this first implementation should be analyzed. Centrally, the current implementation demonstrates events, or commands-to-become, with very different targets and timespans (the examples mentioned above were selected to demonstrate this). It is now implemented by specifying different types of bindings from the controls to the events/commands, namely the types WHENPRESSED, TOGGLE and ONCLICK presented in the part on the setup file (which are then actually handled in the controllers module within the Blender project file, which is not described in this work). This means that the events/commands themselves don't know anything about their duration (although some of them surely assume some of it in their behaviour, i.e. some of the mode-events don't actually launch any actions, and on the other extreme, long-spanning actions may include a state-machine which would be useless if would be activated once only (which the ONCLICK binding does)). On the other hand, this enables reusing the same events in different kinds of bindings trivially (e.g. both ONCLICK and WHENPRESSED bindings may well be useful for some manipulations).

There is also a way to introduce the notion of commands, and yet keep the event-construct (although a better name for it would still be welcome). In this model, the commands would probably be very simple, and one thing they could do would be activating and de-activating events. In this case, if e.g. how modes are currently implemented is not seen as useful to do with the event system, there could be some other mechanism for that that the command system uses (perhaps just commands to set the modes).

Anyhow, some change is required. Therefore, the author wishes to delay the actual release of the program untill this refactoring is done. It does not necessarily have to be a lot of work: as noted in the evaluation, many of these so-called events are like commands already. Furthermore, the base classes and basic functionality around them total in probably only tens of lines code, in any case probably less than a hundred, it's all pretty simple and no great changes would hopefully be required. The author estimates that this could be done in a single work day, perhaps another day for testing (or vice versa, if this project can finally switch to test-driven development!), and re-thinking and refactoring. So this is identified as an early next step.

7.2 Different possible directions with various technologies

There are several possible ways for future development with respect to distinct usage scenarios. At least in the author's view, each of them entails, or in a way bundles, technology selection. However, this being about software, the selections are not mutually exclusive. In fact, their integration is well possible, even foreseeable. Yet, especially within the limits of a single developer, choices regarding focus and priorities must be made. In the following, three different paths are described shortly. After that, some notes about their interplay are made, with respect to the discussion on component re-use in the evaluation.

7.2.1 Authoring for non-programmers via Blender-integration ('BlenderVJ')

One way to target future development based on the work here would be continuing to use the Blender game engine and adding more support to the features it has. Let this approach be be codenamed 'BlenderVJ', as the nickname Pildanovak called his system that was discussed in the chapter evaluation with peer review. Along this route, it would probably be wise to integrate with the other parts of Blender too, also to support relatively straightforward authoring by non-programmers via the graphical user interface.

Pildanovak's BlenderVJ originally (in 2003) used the Blender game logic system heavily, including message actuators etc., unlike in the system presented here, where pure Python has been used for communicating between objects. With a new version, published in late 2004, BlenderVJ started using the newly added library support in the Blender Python API, which enables linking other projects (.blend files) to one project. This way, the project file that includes the presentation engine mechanics (in case of BlenderVJ, mostly the MIDI input support), does not need to include and content, which can be in several separate project files. Moreover, the gamelogic message actuators work between when scenes in different projects, so that input control values (e.g. from MIDI) can be received from the engine in the project files with the content. This way, any Blender scene, which is e.g. animated with IPOs, can be made interactively controllable for VJ use, with controls to switch with scene is active, with no programming required (the Blender logic brick editing suffices). After this change, also the work presented here, KyperMover, might be work as such a scene under Blender VJ. The author gave a presentation of both these tools, thinking about the future developments, in the Blender conference 2004 — for more info about that, see http://www.blender3d.org/cms/Toni_Alatalo.442.0.html .

7.2.2 Good programmability of high-level features with Soya ('Play')

Another way would be to re-implement the system using the Soya 3d engine. An immediate benefit of this would be getting to use the Cal3d character animation library, which is more advanced than the character animation playback with armatures and actions in the Blender game engine, supporting e.g. blending several movements affecting the same bones at the same time. Furthermore, in the new examples in the tutorials of the recent (spring 2004) 0.7alpha release of Soya, there is an even more advanced example of character animation with Cal3d, where an item is dynamically added to the hand of a character (as demonstrated in the character-animation-2 tutorial by the soya project).

However, the author has found the application programming interface (API) even more promising. Soya targets at being a very high-level engine, and provides really elegant methods for creating and controlling 3d scenes and objects. The author of this work believes that this would be a great help in the development of advanced features for e.g. positioning the different movers interestingly and making them interact with respect to each other, the environment and to different user input controls. Moreover, there seem to be no limits to what the engine can do. For example, using the Blender game engine, there is no way to (at least currently, to the author's knowledge) to add new *kinds* of objects during runtime — only pre-modelled objects can be added to scenes (and even than can be done only via so-called addObject actuators which can be awkward programmingwise). In Soya, there is a complete easy-to-use modelling API as a part of the library. This facilitates the implementation of scenes such as the car-follows-road example described in the chapter on requirements. Another interesting possibility is utilizing interactive procedural modelling, such as L-Systems or other plant growing algorithms — the author of this work actually published a small toy application featuring this in spring 2005, see http://studio.kyperjokki.fi/engine/SoyaPlant .

Also, Soya integrates well with Blender, as it is the modelling and animation tool that the developers of the engine themselves use (for their own games). Not only are there mature tools for exporting Blender models and character animations to be used in Soya, there is an editor for Soya worlds that connects directly to Blender. Besides, the Soya API seems to be similar to the way Blender objects are structured, and is probably inspired by it. This all means that concerning this project, little effort would have been wasted when moving to Soya, as all the ready-made models and animations could be reused, and new features for e.g. dealing with textures could probably be developed well with respect to how they work on the authoring side.

Besides the possibilities of advanced real-time character animation and dynamic modelling mentioned above, the author of this work considers Soya as good candidate platform for a system with higher-level controlling logic than the system presented here. This refers to the discussion on authoring in the evaluation chapter, with the mentioning of the concept of manuscript from screen-/playwriting. Therefore this option is codenamed 'play', referring to both gaming and theatre.

Surely there are other well programmable 3d engines too, such as Ogre3d, of which the version 1.0 was released in early 2005. It is powerful both in terms of visuality, utilizing the features of current graphics cards, and in perfomance, being also a well structured programming library. It is written in C++, but the Python bindings have recently (in early 2005) matured too, so that whole projects can be implemented in Python too (leaving C++ necessary for cpu intensive parts only). Considering the needs of this system, it has an own character animations system (i.e. does not use e.g. cal3d), and there are export tools for e.g. Blender for getting the models and animations into Ogre (and the author of this work did this succesfully in spring 2005, albeit little glitches in the tools). Concerning the other needs of this 'play' track, the author has not yet examined the modelling API, but suspects that it is an elegant one. One planned way to learn more about it is to port Soya experiments, such as the aforementioned plant toy, to Ogre.

7.2.3 Distributed system using VOS and/or Twisted ('DanceWorld')

A third option for next step in the development would be to target networking first, before any or much work on new features. Simple tests with remote controlling were already made with the current engine in October 2003, when the author successfully triggered events/actions on and off over an IP network. That model, where is still a single engine instance that is just controlled from many computers, may be fit for VJing/ / live performance purposes in a single physical space where everyone can see the same displays. Also several other implementations, where a gameblender scene is controlled over the network, exist. One implementation, that uses CORBA event service for synchronization, has been used for controlling robots (http://people.mech.kuleuven.ac.be/~pissaris/misc/blender/blendercorba.html).

However, having a true networked or even a distributed system would open more possibilities, for e.g. connecting several events that are occurring simultaneously in remote places, via a virtual world. The author has some experience in performing in such events, when participating in so-called net.audio co-streaming sessions organized by Xchange (http://xchange.re-lab.net/). Also, the author has made some experiments in this direction as a part of the yearly Airguitar Worldchampionships event (as reported in http://www.nettime.org/Lists-Archives/nettime-l-9908/msg00132.html).

A technology for distributed computing, targeting mainly collaborative 3d worlds, is the Virtual Object System (VOS, http://interreality.org/). For the purposes here, it suffices to say that it enables having 3d scenes shared as in networked computer games, but also so that parts of the scenes (e.g. certain avatars i.e. characters) may reside on different computers and be not controlled by a central server (i.e. it enables peer-to-peer communications too). There is a VOS plugin for the open source 3d engine Crystal Space, which enables moving around in the worlds.

Both the reference VOS implementations (e.g. the vosworld server) and all of the Crystal Space 3d engine and the VOS plugin for it are all written in C++, with performance in mind and utilizing threading. This means that they are relatively high-performing pieces of software. However, conserning the needs of the project described in this work, writing C++ would be an unnecessarily low level task, as the author would have to start worrying about the details of memory management and be encumbered by an inferior (less flexible) object model and the lack of built-in data structures such as lists, dictionaries and sets of Python. Now in early 2005 early Python bindings have been introduced for VOS, but the author has not studied them in more detail.

The contents for VOS worlds are also created with Blender, and Cal3d is also the default character animation library in Crystal Space, so the remarks about the interplay with Blender that were risen when discussing Soya hold at least partially here too.

Besides VOS, there are numerous other technologies for distributed computing. One that the author has evaluated and experimented with is called Twisted, which is an event-based (and can be ran in a single process without threads) networking framework written in Python, and suitable for e.g. network games (also various chat protocols and basically all Internet service protocols have been implemented with it). The distributed object system for Twisted is called Spread, and it was been implemented at least in Java in ELisp too (http://twistedmatrix.com/products/twisted). So, instead of VOS, Twisted could be used to network/distribute the KyperMover or some continuation of it. Another option might be implementing the virtual object protocol (VOP) of VOS with Twisted, to enable making VOS applications in Python without having Python bindings for the C++ implementations of it. Also, regarding on-line live performances, the recently released system called Upstage is also an interesting candidate for integrating the system described in this work. The Upstage server is written in Python using Twisted, but the client is (a 2d engine) written in the ActionScript of Macromedia Flash, so it runs without installing anything in many web-browsers which is a major benefit when considering participatory web events (such as the play-togethers of the airguitar world championships event). Yet, regarding networked gaming, the Soya project released a Twisted-using library, called Tofu, in early 2005. It is still early in development, but works at least for some applications.

7.3 Strategies for the interplay of the different areas

A resulting challenge from the exploration of the different areas , where future development could be made, is their interplay. Obviously, having nice authoring for non-programmers, enabling good programmability for the underlying library and performance logic, and networking the system to be a collaborative virtual environment are not mutually exclusive goals. But making the pieces so that they fit together is not a trivial challenge — especially if simultaneous support for different solutions (e.g. different 3d engines, or even 2d and 3d engines at the same time).

As mentioned in the section on evaluation of code reusability, there are different well-known strategies for achieving platform independent for interoperable pieces of software, such as definining interfaces that different classes can implement and the model-view-controller pattern. Evaluating their usefulness, which probably demands experimentation, for the future development of this tool and VJing applications in general, is left for future work.

8 Conclusions

A tool that addresses the research questions was developed and used by the author, and is published for others to see and use too. The development opened a plethora of new questions, with directions for looking solutions in different literatures.

Initially, the question was simply how to make a system for controlling 3d character animations in live performances / for VJing (as none such tools were seen to exist). Then, a first issue was the actual controlling mechanism. It was seen that some abstraction that maps controls to actions would contribute for a good design. This was implemented as so-called events. Later in evaluation this changed to a recommendation for considering the command pattern, with the lessons learned from the usage of the structure made in this system in mind.

No use studies were made, but experiences from usage suggest that the prototype filled the initial limited need successfully, proving the design and implementation basically viable. Central weaknesses were identified, including the system being too limited for solo perfoming and too cumbersome for the performer to react. To solve those issues necessary areas of improvement were seen to include dealing with scenes with backgrounds and having preset configurations which are immediately visually compelling. There are also issues in having adequate content, both in quality and quantity. Then again, there are other use purposes for this kind of tools, described in more detail in the chapter on evaluation, where the aforementioned demands are not as great, such as trying out ideas for other projects and use of the computer-generated visuals to complement shows with other elements (like physical puppets).

As a conceptual result, the principle of *similarity* from Gestalt theory was found to frame the so-called clonephasing implementation fruitfully, so that new functionality could be designed based on the other concepts there, such as *proximity*, *closure* and *simplicity*. Obviously many other literatures, supposedly the ones on dance and theatre, would have much to give too.

Future challenges include a variety of issues in several possible directions, ranging from the refactoring the core of the system itself to the integration of it to other systems, such as Blender and possibly other authoring tools, other real-time engines, controller device interfaces such as MIDI, and shared computing platforms for networked collaboration. All these related areas of technology develop continuously, and a central challenge for future work in this area are the strategies for making them all work together.

References

Abouaf, J. 1999a. Biped: A Dance with Virtual and Company Dancers, Part 1.. IEEE MultiMedia. Vol. 6 no. 3., 4–7.

Abouaf, J. 1999b. Biped: A Dance with Virtual and Company Dancers, Part 2.. IEEE MultiMedia. Vol. 6 no. 4., 5–7.

Badler, N., Allbeck, J., Zhao, L. & Byun, M. 2002. Representing and Parameterizing Agent Behaviors. In Proc. Computer Animation, IEEE Computer Society. Geneva, Switzerland.

Badler, N. I., Palmer, M. S. & Bindiganavale, R. 1999. Animation control for real-time virtual humans. Commun. ACM. Vol. 42 no. 8., 64–73.

Beck, K. 2000. Emergent Control in Extreme Programming. Cutter IT Journal. Vol. 13 no. 11., 22–24.

Bindiganavale, R., Schuler, W., Allbeck, J. M., Badler, N. I., Joshi, A. K. & Palmer, M. 2000. Dynamically altering agent behaviors using natural language instructions. In International Conference on Autonomous Agents. 293–300.

Boulic, R., Huang, Z., Magnenat-Thalmann, N. & Thaimann, D. 1993. A Unified Framework for the Motion Manipulation of Articulated Figures with the TRACK System. In Second Conf. on CAD and Graphics. Beijing.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. 1996. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley Sons.

Dannenberg, R. 1989. Real Time Control For Interactive Computer Music and Animation. In Proceedings of The Arts and Technology II: A Symposium, Connecticut College. 85–94.

Dannenberg, R. & Bates, J. 1995. A model for interactive art. In Proceedings of the Fifth Biennial Symposium for Arts Technology. 103–111.

Dannenberg & Rubine 1995. Toward Modular, Portable, Real-Time Software. In Proceedings of the International Computer Music Conference. 65–72.

Eckel, B. n.d.a. Thinking in Patterns with Java. Work in progress at http://mindview.net/Books/TIPatterns/.

Eckel, B. n.d.b. Thinking in Python. Work in progress at http://www.mindview.net/Books/TIPython.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1993. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Goto, M. & Muraoka, Y. 1997. Issues in evaluating beat tracking systems. In Working Notes of the IJCAI-97 Workshop on Issues in AI and Music.

Gregor, S. 2004. Evaluating contributions to information systems design theories. A presentation in the INFWEST.IT seminar on constructive research.

Hevner, A., March, S., J, J. P. & Ram, S. 2005. Design Science Research in Information Systems. Management Information Systems Quarterly (MISQ). Vol. 28 no. 1., 75–105.

Jarvinen, P. 2003. On research methods. University of Tampere.

Kujanpää, T. & Manninen, T. n.d.. Supporting Visual Elements of Non-Verbal Communication in Computer Game Avatars.. In M. Copier & J. Raessens, eds, Proceedings of Level Up - Digital Games Research Conference. 220–233. Universiteit Utrecht.

Lieberherr, K., Holland, I. & Riel, A. 1988. Object-oriented programming: an objective sense of style. In OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications. 323–334. ACM Press. New York, NY, USA.

Lippe, C. 2002. Real-Time Interaction Among Composers, Performers, and Computer Systems. In Information Processing Society of Japan SIG Notes. Vol. 123. Available: http://www.music.buffalo.edu/faculty/lippe/pdfs/Japan-2002.pdf 1–6.

Meyer, D. 2005. Action class for items: freevo/ui/src/menu/action.py. Available from svn.freevo.org.

Moore, P. & Fitz, C. 1993. Gestalt Theory and Instructional Design. Journal of Technical Writing and Communication. Vol. 23 no. 2., 137–157.

Petros Faloutsos Michiel van de Panne, D. T. 2001. Composable Controllers for Physics-Based Character Animation. In Proc. SIGGRAPH.

Puckette, M. 1991. Something Digital. Computer Music Journal. Vol. 15 no. 4., 65–69.

Solano, M. B. 2004. Towards an aesthetics of cognitive systems: a post-humanist perspective for cognitive studies of improvisational dance with dynamic real-time multimedia environments. Ohio State University.

Wertheimer, M. 1923. Laws of organization in perceptual forms. First published as Untersuchungen zur Lehre von der Gestalt II, in Psycologische Forschung. Vol. 4 , 301–350. Translation published in Ellis, W. (1938). A source book of Gestalt psychology (pp. 71-88).

Zongker, D. E. & Salesin, D. H. 2003. On Creating Animated Presentations. In D. Breen & M. Lin, eds, Eurographics/SIGGRAPH Symposium on Computer Animation. New York, NY, USA.