Orbits - a gesture-controlled video game for psycho-acoustic tests.

Sascha Bienert

23rd February 2010

Eidesstattliche Erklärung

(Affirmation)

Hiermit versichere ich, die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Sascha Bienert
Berlin, den 01. März 2010

Acknowledgements

Abstract

Contents

Chapter 1

Interacting with Sound

1.1 Perception

The human being is endued with five senses[1] (Vision, Hearing, Taste, Smell, Touch) as already described by Aristotle [1]. Amongst all of them vision and hearing are the most important for human computer interfaces; by means of them most of the communication between human and computer takes place [2].

Sight is our main sense, 80% of all information that we perceive is sensed by our eyes, but it is limited to the viewing cone. Eventually hearing gives us the ability to "draw" the complete picture of our local environment. With our ears we can sense information about objects that are hidden or outside our field of vision. Furthermore we are able to perceive things that are not visible, e. g. small insects, material properties or such abstract things as music.

If one takes a textbook to read about the ear's topology, functionality and capabilities, its ability to deviate distance and azimuth of an auditory source, and the perception of loudness, intensity and frequency are always elaborately discussed. In recent years, a very interesting question has been subject to various studies: How do we perceive ecological information such

[1] Modern physiology attributes more four senses to the human (such as equilibrioception and thermoception).

as properties of the source that produced a sound? Sound is part of our environment, almost every event of our natural surroundings, every activity, involves sounds — sounds that include information about the underlying physical incidents. However, besides the "natural" environment, we humans spend more and more time in "artificial" environments, whose sounds have to be generated artificially too.

In many cases perception is shaped by multi-modal influences. The size of an object for example can be seen *and* felt. Taste and Smell also show a close relationship; everybody remembers how strange things "taste" when we suffer a cold and cannot smell. But what about the weight of things or the textures of surfaces? It clearly can be felt by lifting or touching them. It is an interesting topic to explore if there are other ways to perceive those qualities. As things in computer games in most cases cannot be touched and some properties clearly cannot be imparted by means of the visual sense (by displaying them on the screen), it is worthwile to dedicate our intention to the possibilities that lie in auditory perception.

Two related aspects have been subject to a number of scientific studies over the past years: firstly, to which extent are humans able to recognise properties of the sound source through auditory reception? And secondly, can human performance at control tasks in human computer interfaces be augmented by adding sound? Especially continuous sound is a sparsely investigated topic in the designing of human computer interfaces.

1.1.1 Perception of ecological properties

There are a number of papers about perception of sound-producing processes of our environment[2]. Lederman (1979 [4]), Wildes & Richards (1988 [5]), Gaver (1988 [6]), Lutfi & Oh (1997 [7]) and Klatzy, Pai & Krotkov (2000 [8]) addressed the perception of material, Lakatos et al. (1997 [9]), Carello et al. (1998 [10]) and Kunkler-Peck & Turvey (2000 [11]) the shape and size of objects, Houben, Kohlrauch & Hermes (2001 [12]) the perception of velocity and Warren & Verbrugge (1984 [13]) the interaction between objects.

[2]An overview can be found in [3].

Vanderveer was the first to describe the *ecological approach* to auditory perception (1979 [14]), an approach characterised by a focus on events of our environment. William W. Gaver proposed the concept of two modes of listening, which he called *musical listening* (the traditional approach: pitch, loudness, duration, etc.) and *everyday listening* (1988 [6], 1993 [15] [16]). New attributes and qualities of perception are of most interest, those that tell something about the source of a sound-producing process or the process itself (which object? what has happened to it?).

Besides objects and physical processes, there has also been research on sounds caused by human beings. For instance, Repp (1987 [17]) investigated the perception of the sounds of clapping hands and Li, Logan & Pastore (1991 [18]) presented subjects the sounds of walking persons.

Those papers show that sounds play a roll when we explore our environment. However, most of them base upon sets of prerecorded sounds whereas continuous (realtime) sound feedback has rarely been studied. It is a major motivation of this thesis to establish a basis for studies on the perception of realtime-generated sounds.

The ecological approach and everyday listening

J. J. Gibson (1950, 1966, 1979) was the major developer and proponent of the *ecological approach* to perception [19]. Gibson's approach competes with the so-called *"cue theory"*, another theoretical approach. Both theories originate from considerations about the (visual) perception of depth. Goldstein [19] explains:

> The cue theory focuses on identifying information in the retinal image that is correlated with depth in the world [...] and proposes a number of types of cues that signal depth in a scene. [...] In contrast, the ecological approach focuses not on the information in the retinal image, but on the information that exists "out there" in the environment.

Gibson pointed out that our environment contains constant information that does not change when an observer changes position or moves through the environment. He called that constant information *invariant information*. Goldstein about Gibson's approach:

> Gibson's emphasis on invariant information reflects his commitment to studying perception that occurs in the natural environment (hence the term *ecological approach to perception*). Gibson pointed out that, as people go about perceiving their environment, they are usually moving, and therefore, they need to use invariant information that doesn't change every time they observe their environment from a different point of view.
>
> In addition to his assertion that the study of perception should be concerned with how moving observers use invariant information to perceive, Gibson made another assertion that we need to consider in order to fully appreciate his approach. Gibson argued strongly against two important components of the constructivist-based approach [...]: the retinal image and processing. [...]
>
> However, many (researchers) feel that information in addition to Gibson's invariants are involved in perception, and that perceptual processing is far too important a part of perceptual process to ignore. Many researchers also feel that, for Gibson's approach to be truly meaningful, it is necessary to go beyond identifying information that is *available* for perception and to determine whether that information is actually *used* for perception.

In [13] Warren & Verbrugge summarise: "(The) general perspective on auditory perception may be called *ecological acoustics*, by analogy to the ecological optics advocated by Gibson (1961, 1966, 1979) as an approach to vision. The ecological approach combines a physical analysis of the source event, the identification of higher order acoustic properties specific to that event, and empirical tests of the listener's ability to detect such information, in an attempt to avoid the introduction of ad hoc processing principles to account for perception (Shaw, Turvey, & Mace, 1981)." In [15] Gaver describes the continuum from event to experience (see fig. 1.1) and finds that the emerging sound "provides information about an *interaction* of *materials* at a *location* in an *environment*".

Gaver established the concept of *everyday listening* (1988 [6], 1993 [15] [16]). He defines everday listening "as the perception of events from the sounds they make" and distinguishes it from *musical listening*, which he describes as "the perception of structures created by patterning attributes of sound itself" ([6], p. 3). In [15] he illustrates that difference:

> Imagine that you are walking along a road at night when you hear a sound. On the one hand, you might pay attention to its pitch and loudness

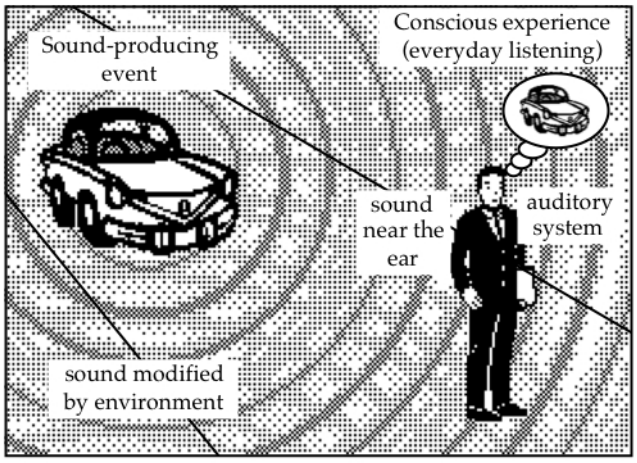

Figure 1.1: "The continuum from world to experience. A source event causes sound waves: some radiate directly to an observation point, others are modified by the environment before reflecting to the listener. Invariant patterning of the acoustic array near the ear is picked up by the auditory system, providing information for the experience of everyday listening" (Gaver [15])

and the ways they change with time. You might attend to the sound's timbre, whether it is rough or smooth, bright or dull. You might even notice that it masks other sounds, rendering them inaudible. These are all examples of *musical listening*, in which the perceptual dimensions and attributes of concern have to do with the sound itself, and are those used in the creation of music. These are the sorts of perceptual phenomena of concern to most traditional psychologists interested in sound and hearing.

On the other hand, as you stand there in the road, it is likely that you will not listen to the sound itself at all. Instead, you are likely to notice that the sound is made by an automobile with a large and powerful engine. Your attention is likely to be drawn to the fact that it is approaching quickly from behind. And you might even attend to the environment, hearing that the road you are on is actually a narrow alley, with echoing walls on each side.

This is an example of *everyday listening*, the experience of listening to events rather than sounds. Most of our experience of hearing the day-to-day world is one of everyday listening: we are concerned with listening to the things going on around us, with hearing which are important to avoid and which might offer possibilities for action. The perceptual dimensions and attributes of concern correspond to those of the sound-producing event and its environment, not to those of the sound itself. This sort of experience seems qualitatively different from musical listening, and is not well understood by

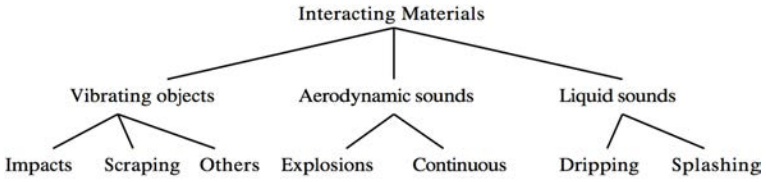

Figure 1.2: "A hierarchical description of simple sonic events" (Gaver [15])

traditional approaches to audition.

Gaver was seeking to "develop an account of ecologically relevant perceptual entities: the dimensions and features of events that we actually obtain through listening"[3] and "describe the acoustic properties of sounds that convey information about the things we hear"[4]. Therefore he developed a descriptive framework of the perceptual attributes and dimensions that characterise the auditory perception of events and focussed on audible attributes of source events. Fig. 1.2 shows a map of the space of everyday sounds. "Sound-producing events are distinguished first by broad classes of materials and then by the interactions that can cause them to sound. Most generally, sounds indicate that something has happened, that an event has occurred, that there has been an interaction of materials. All sounds, then, convey this information."

Material, shape, size, velocity and interaction

Lederman (1979 [4]) investigated the role of tactile and auditory information in the perception of surface texture. Subjects were asked to judge the roughness of aluminium plates numerically. Three conditions were presented: only tactile, only auditory, tactile-plus-auditory. Auditory judgements were similar, but not identical to corresponding haptic touch judgments. Roughness estimates between the tactile and tactile-plus-auditory condition did not differ, i.e. subjects tended to use the tactile cues if both sources of information were available. However, subjects were able to judge roughness using only

[3] "What in the world do we hear?" [15]
[4] "How do we hear it?" [16]

the sounds produced by touching the surfaces[5].

Guided by a model of vibrating solids[6] Wildes & Richards (1988 [5]) were searching for an acoustical parameter that characterizes material. The coefficient of internal friction $\tan\phi$ is an instrinsic parameter of solid material that describes its dynamical behaviour[7]. Wildes & Richards derived two related measures of internal friction in the acoustical domain: decay time t_e and bandwidth Q^{-1}. However, they did not study which role t_e and Q^{-1} play at the perception of material.

William W. Gavers (1988 [6]) dissertation is titled "Everyday Listening and Auditory Icons" (q.v. page 11), and contains two empirical investigations about the identification of 17 different environmental sounds ([6], chapter 3, p. 38 ff.) and categorisation of material of struck metal and wooden bars of different lengths ([6], chapter 4, p. 46 ff.). In the first experiment subjects were asked to describe presented sounds as detailed as possible. Among these sounds were impacts sounds, crumpling sounds, paper sounds, liquid sounds and complex sounds. Impacts and liquid sounds were always identified. Crumpling can sounds were often confused with multiple impacts, whereas crumpling paper was rarely confused with impacts. Subjects seemed to be able to extract information from impact sounds "about the material of sound-producing objects, as well as their hollowness and sometimes their sizes" ([6], p. 40). The complex sounds also showed interesting results: "all subjects accurately identified the razor sound as made by a machine, (but) they were fairly uncertain about the exact nature of that machine" ([6], p. 42). Walking sounds, writing with chalk and opening/closing doors were always correctly identified, whereas only one subject correctly recognised an opened and closed file drawer. Gaver states that "[...] accuracy of perception is not an all or none matter, but depends on the required level of generality. Sometimes subjects' responses were accurate only in their abstract similarities with the actual events, but other times they were correct about quite detailed aspects of the events." ([6], p. 39)

[5] In the auditory condition the subjects did not touch the surfaces themselves, the experimenter moved his fingertips along onto the aluminium plates.

[6] standard anelastic linear solids

[7] $\tan\phi$ is the measure of anelasticity.

Lutfi & Oh (1997 [7]) studied the discrimination of material in the sound generated by an idealized struck bar, rigidly clamped at one end. "Frequency, intensity, and decay modulus are, for a fixed geometry and fixed driving force, uniquely determined by the material composition of the bar." Listeners were asked to discriminate changes in material composition. In one half of the experiment they were presented two stimuli, one of them being an iron sound, and the other either silver, steel or copper. In the second half they were presented glass sounds with crystal, quartz and aluminium being the alternatives. Upon each presentation a random pertubation in each of the acoustic parameters was introduced. "For each material and procedure a set of 100 waveforms was synthesised where the values of mass density and elasticity for each waveform were selected independently and at random from normal distributions." Correlations with the listener's response were calculated and "reveal(ed) that listeners fail to make optimal use of the information in the acoustic waveform by tending to give undue weight, for a given material change, to changes in component frequency".

Klatzky, Pai & Krotkov (2000 [8]) investigated the relation between material perception and variables that govern synthesis of contact sounds (fundamental frequency and frequency-dependent rate of decay). In the first two experiments subjects judged the similarity of synthesised sounds with respect to material on a continuous scale. The sounds were synthesised according to the theory of clamped struck bars struck at an intermediate point. Stimuli had the same values of frequency and decay, but in the second experiment they were equalized by overall energy. As results did not differ significantly in the two experiments they showed that decay rate, rather than total energy or sound duration, was the critical factor in determining similarity. In experiment 3 subjects had to judge the difference in the perceived length of the objects. It was shown that similarity judgments in the first two studies were specific to instructions to judge material. In experiment 4 subjects had to categorize the material of the objects using four response alternatives: rubber, wood, glass and steel. The results demonstrated that judgments of material and length difference both depend significantly on frequency and decay, even though decay played a substantially larger role in material sim-

ilarity judgments than in length.

Lakatos, McAdams & Caussé (1997 [9]) investigated listeners' ability to discriminate geometric shape. "In cross-modal matching tasks, subjects listened to recordings of pairs of metal bars (Experiment1) or wooden bars (Experiment2) struck in sequence and then selected a visual depiction of the bar cross sections that correctly represented their relative widths and heights from two opposing pairs presented on a computer screen." All the stimuli were equalized in loudness. For the analysis a 75% performance criterion across all trials was applied. Subjects that did not discriminate the geometric differences with at least 75% correct answers in a two-alternative forced choice (2AFC) task were excluded from further analyses. For the metal bars 8.3% of the subjects were subsequently excluded, and 16.6% for the wooden bars. Results were then analysed using a multidimensional scaling program (MDS). A two-dimensional solution for the data related to the steel bars was found to be the most appropiate. "The two dimensions appeared to be related to the W/H ratio of the bars and spectral centroidLog Likelihood (LogL) and Values of Information Criterion". For the data related to the wooden bars a stable one-dimensional MDS solution was found. Again, the coordinates along this dimension correlated with the W/H ration of the bars. In summary, the authors state that "the results suggest that listeners can encode the auditory properties of sound sources by extracting certain invariant physical characteristics of their gross geometric properties from their acoustic behavior".

Carello, Anderson and Kunkler-Peck (1998 [10]) investigated the ability to perceive the precise sizes of objects on the basis of sound. Subjects judged the perceived length of dropping wooden dowels of different lengths onto a hard surface. It was shown that the ordinal and metrical success (relationship between acoustical and perceptual levels) was related to length, i.e. the actual length is a predictor for length estimates. That followed an analysis using three acoustical features: duration, amplitude and frequency to examine their account for perceptual performance and it was revealed that "none of the simple regressions of perceived length onto these acoustic variables was as successful as actual length in accounting for performance in the two

experiments individually or combined". A kinematic analysis of the falling rods "suggests the potential relevance of an object's inertia tensor in constraining perception of that object's length, analogous to the case that has been made for perceiving length by effortful touch".

Kunkler-Peck & Turvey (2000 [11]) investigated shape recognition from impact sounds. In Experiment 1, the heights and widths of steel plates of identical mass (and thus identical area) were varied[8]. The steel plates were struck by a steel pendulum[9]. "A report apparatus [...] consisted of four independently movable wooden dowels (two horizontal and two vertical), whose positions could be adjusted along a single wire track [...]". Subjects were asked to indicate either height or width of a presented plate[10] by moving two of the report dowels (either the horizontal or the vertical ones). Subjects uniformly underestimated the actual dimensions, but ordered properly. Furthermore, results suggest that the "participants' responses were scaled by a definite impression of [...] height and width". Simple regressions of perceived dimensions or modal frequencies as predictors were computed and showed correlations of at least 0.95. Thus "Experiment 1 revealed that listeners could disciminate the heights and widths of vibrating rectangualr objects". In the second experiment participants were asked to scale the dimensions of steel, Plexiglas and wooden plates. Actual dimensions were not precisely reproduced — as in experiment 1 they were underestimated — but ordered correctly and subjects' perceived dimensions showed definite scaling again. Material was found to "modulate the perceptual measures of plate geometry". In Experiment 3 and 4 Kunkler-Peck and Turvey investigated shape recognition directly. In Experiment 3 subjects were asked to judge whether a vibrating plate is circular, triangular or rectangular. The plates were made of steel[11]. "Participants were not given practice trials—they were never provided feedback throughout the experiment." For Experiment 3 the authors found that "participants accurately identified the corrent shape at a level well above chance". In Experiment 4 plates of different shape (circu-

[8] A square, a medium rectangle and a long rectangle were used.

[9] The pendulum was always released from the same point at each trial.

[10] The plate was struck (and thus audible), but not visible to the subject.

[11] The plates were of the same material, mass and surface area.

lar, triangular, rectangular) and material (steel, wood, Plexiglas) were used and participants were asked to judge shape and material. The material was almost perfectly identified, and shape was correctly identified at a level well above chance. In the analysis it was revealed "that there was a tendency for participants to assiciate a partucilar material with a particular shape (wood with circle, steel with triangle, and Plexiglas with rectangle)".

Warren & Verbrugge (1984 [13]) investigated the identification of breaking and bouncing events. Comprising the ecological approach (q.v. page 11) they distinguish between the *structural invariant* of an event ("the information that specifies the kind of object and its properties under change" [13]) and the *transformational invariant* ("the information that specifies the style of change itself" [13]). The authors explored "the acoustic consequences of dropping a glass object and its subsequent bouncing or breaking" to "identify the transformational invariants specific to the two styles of change and sufficient to convey them to a listener." Structural invariants (material, size, shape) were not discussed. Experimental stimuli were recorded by dropping different glass objects from different heights. Thus for each object a breaking and a bouncing sound was recorded. The authors verified the subject's ability to correctly identify these two kinds of events with the recorded stimuli. Thereupon two experiments with synthetic stimuli were conducted. The bouncing sound was synthesised by quasi-periodic pulse trains generated by recordings of glass tokens, and the breaking sound was synthesised by superimposing the same pulse trains using different damping coefficients. The subjects were able to identify the events very accurately. The transformational invariants for bouncing were found to be a single damped quasi-periodic sequence of pulses. In contrast, for breaking these invariants are a multiple damped, quasi-periodic sequence of pulses.

1.1.2 Continuous sound feedback: related work

In this section I want to present some papers that make use of continuous sound feedback, which beyond the musical context is a rather uncharted field of research.

Rath & Rocchesso investigated the influence of continuous auditory feedback on the perception of the state of a manipulated object (2005 [20]). They developed "a sound model for rolling interaction that enables the continuous control and immediate acoustic expression of the involved ecological parameters" and that "runs in real time in its complete control-feedback behavior". The sound model is based on physical considerations (but involves a degree of simplification and abstraction) and the sound is computed in real time. Details about the sound model can be found in [21] and will be discussed later in this work (q.v. page 45). The authors "embodied the model into a simple control metaphor of balancing a ball on a tiltable track". Users have access through a physical representation of the balancing track, a 1-meter-long wooden stick. This tangible audio–visual interface has been called *"Ballancer"* and can be seen at figure 1.3. An identification test showed that the "overall association of the synthetic sound with rolling was high". The test consisted of four scenarios. In the first scenario subjects were asked to free-associate short sound examples "of a small ball rolling on a plain, smooth, hard surface until coming to a rest" synthesised by the model (virtual ball and synthesised sound and display). In the second scenario subjects were blindfolded and had access to the balancing track, they heard sound generated (in real-time) by the same sound model and were asked the same question (blindfolded and synthesised sound). They were instructed to carefully move their arm up and down and listen to the sonic reaction of the device. Finally they were asked to identify what they heard. In the third and fourth scenario the sound was not synthesised by the rolling sound model, but a mechanical device was used (a glass marble rolling on a track, please compare figure 1.3). Subjects were blindfolded in both tests. In the third scenario subjects listened to the sound of the small marble and were asked what they heard. In the fourth scenario these (blindfolded) subjects were given access to the track, followed by the same question. The authors subsume their findings in the following way:

> Summarizing the results of the questions about the sounds and the tangible-audible device, we can state that the subjects intuitively understood the modeled metaphor. The combination of modeling everyday sounds

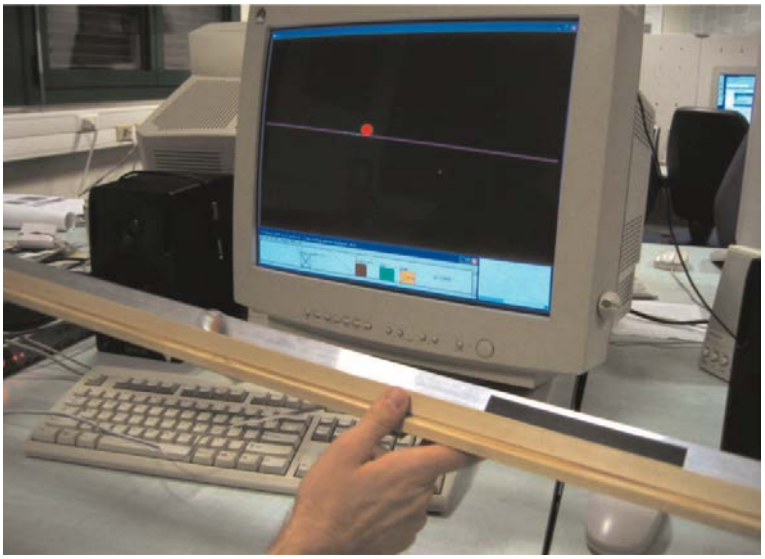

Figure 1.3: *Ballancer* with a glass marble rolling on its upper face's aluminum track (Rath & Rocchesso [20]).

> and using a familiar control metaphor exhibits the advantage that virtually no explanation and learning are necessary. With our approach, users can immediately understand and react to transported information without being instructed, in contrast to systems that use abstract sounds and controls. The identification of the scenario is even clearer for the tangible-audible interface than for the actual mechanical device that provides a physical realization of the metaphor. This demonstrates the effectiveness of the cartoonification[12] approach to sound modeling: Although subjects perceive the device as fictitious, nevertheless it can quite reliably elicit an intended mental association, even more clearly than the real thing.

In a performance test Rath & Rocchesso evaluated the question of "whether users, besides identifying and appropriately using the sound model and control interface, actually perceive the dynamical ecological information contained in the sound and make use of this information". Subjects were asked to perform a specific task, "consisting of moving—by balancing and tilting the track—the virtual ball from a resting position at the left end of the track into a 15-cm-long mark slightly to the right of the center and stopping it".

[12]The term "cartoonification" is described in the article. A deeper discussion can be found at Gaver [6]. Briefly, it comprehends (intended) simplification and exaggeration to reinforce an illusion of substance.

This test was performed under various conditions of sensory feedback and the movement data was recorded. During the test "feedback about the position and velocity of the virtual ball [...] was given acoustically through sound from the rolling model and/or visually on the computer screen, as a schematic representation of the ball on the track". The subjects were asked to perform the task as fast as they could. "The graphical display [...] was realized in four different sizes: 1/3, 1/6, 1/12, all of the 19-inch computer screen". Finally, the subjects were asked to solve the task without visual display and auditory feedback only. Rath & Rocchesso found that "the average time needed to perform the task improved significantly with the auditory feedback from the model" for all display sizes. An analysis of the recorded user movements revealed a "different, more efficient behaviour of acceleration and stopping the virtual ball before reaching the target area". Furthermore, all subjects were able to solve the task with purely auditory feedback, without display.

In a more recent study (2008) Rath & Schleicher investigated the influence of continuous auditory feedback on various measures of performance and quality of interaction in a control task (2008 [22]). They used the experimental tangible audio–visual interface *Ballancer* like described above. In this study subjects were supplied with "optimal" visual feedback conditions: the graphical display spanned the whole range of the control stick ($1m$) and the virtual ball on the screen had two differently coloured halves (in the previous experiement it was monochrome) so that the turning movement itself may serve as an additional visual cue of ball movement (please compare figure 1.4). In the previous research (as described above), the position of the virtual ball was reflected through amplitude stereo panning and the target area was marked by a different acoustic behaviour. By contrast, this study "employs conditions of sonic feedback that exclude any possibility of positional information in the sound". More precisely, the authors specify: "The momentary velocity of the virtual ball is used as the only input parameter. The sonic feedback does not depend on the controlled virtual ball's location in- or outside the target area and is entirely monophonic, i.e. does not contain any attributes of localisation". In a user test three sound conditions were used: a *rolling sound*, an *abstract sound* and *no sound*. The

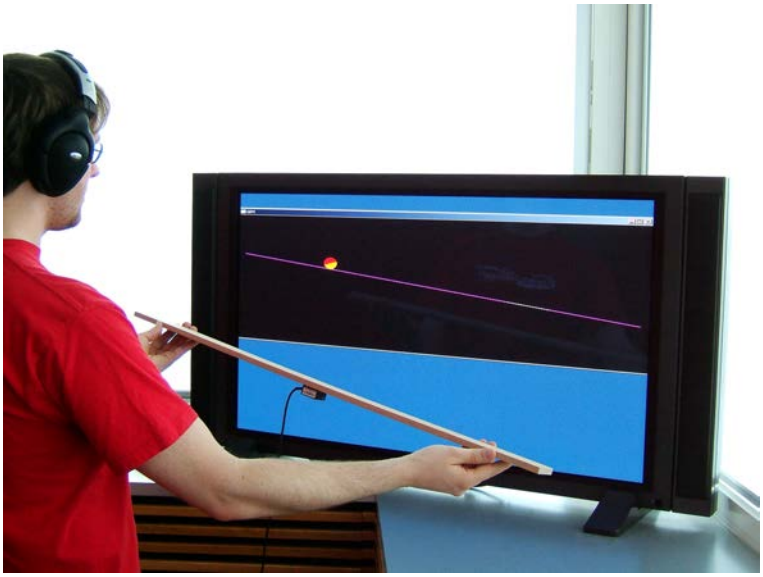

Figure 1.4: The *Ballancer* in the configuration with a wide-screen display spanning the whole size of the 1–m physical control stickball (Rath & Schleicher [22]).

rolling sound model is the one described above — "a simplified physical model of the interaction of the rolling object and the plane to roll on". In the second sound model, called *abstract sound*, any idea of realism is intentionally ignored. Still, the purpose is to express the velocity of a controlled movement. With two simplifications compared to the rolling sound model have been made: firstly, interaction between ball and surface is disregarded (the centre of the virtual ball moves like an ideal needle of a record player that follows the profile of the groove), and secondly, the surface profile has been replaced by a the shape of a lowpass-filtered sawtooth signal (primarily being a bandpass-filtered noise at the rolling model). Subjects were asked to move the virtual ball into a graphically marked target area as fast as possible — under the described sound conditions. The different conditions were presented in different sets of 20 runs each. Across all subjects the order of sets was counterbalanced with intent to "cancel out" training effects in the comparison of different conditions of feedback averaged over all subjects. The whole series of all conditions was repeated once for each subject. The first half of the test is regarded as the "untrained" state and the second as the "trained". After a statistical analysis of their data the authors found that

"on average over all subjects the target reaching task is concluded faster with additional auditory feedback than without" (for the very good visual feedback conditions of the wide–screen display used in this study). Apparently, "the strength of the performance improvement through sound [...] interacts with the actual type of auditory feedback ("rolling" or "abstract") and the factor of evolving time in the course of the experiment". The task performance in the untrained series was significantly faster with *rolling sound* than with *no sound* and the performance with *abstract sound* was also faster than with *no sound* (but not as much). The trained state in the second half of the experiment the *abstract sound* led to the best average (being significantly shorter than without sound). The average performance with *rolling sound* was still better than with *no sound*. In further analysis two characteristic indices of quality of control movements in the recorded movement trajectories have been derived and analysed: the number of "ball oscillations" and "inclinations swaps". Both were seen to be smaller with sound in average.

Avanzini et al. explored the role of auditory cues in a cross-modal presentation with different sensory modalities (visual and tactile) [23]. In a control task subjects were asked to rate the resistance of an object to motion based on visual and auditory cues[13]. In one experiment also haptic feedback was presented. The results are partially conflicting and not very strong in general.

In 2001 Müller-Tomfelde and Münch presented a work about modeling and sonifying pen strokes on surfaces [24] to bring back the lost sound qualities of otherwise silent electronic white boards or pen tablets. The sound feedback provides information about the kind of surface, the pen and the way of writing. The authors note the redundant character of these information in the real world but also emphasise the potential use of such feedback in virtual environments to make them feel more natural and coherent.

Müller-Tomfelde and Steiner also developed an interactive electronic whiteboard with audio feedback called DynaWall® [25]. In contrast to audio feedback at a standard computer environment the DynaWall is a team work place, audio feedback of actions of a single person is presented to the whole group.

[13] A sound of friction was synthesised based on a physical description of the frictional interaction between two facing objects.

It consists of computer projectors, touch-sensitive displays and loudspeakers behind them. The audio enhanced interaction is achieved by three types of feedback: gesture recognition feedback in form of gesture melodies (rythm patterns), a haptic subsonic feedback (subsonic sine waves make the display vibrate[14]) and a continuous sound feedback for moving objects constisting of coloured noise[15]. They also experimented with different sound models to imitate different properties of the surface like "chalk on slate", "boardmarker on a flipchart" or "a book on a table". At the DynaWall people can "throw" objects to other users who hear them coming before they can see them due to their limited field of vision. The sonification of actions at the whiteboard give the working group an overlook of what everybody is busy with.

1.2 Technical aspects

1.2.1 Sampling: a conventional method for discrete and continuous feedback

The playback of sound recordings is called sampling. It is currently the most deployed method to set something to sound — both singular events and continuous processes.

Sounds in video games also almost solely consist of prerecorded, reworked (if required) and finally played back sounds. Since the repeated playback of recorded sounds for a series of successional discrete events (like the bouncing of a ball) can quickly seem monotonous or unnatural, several techniques to vary the sound are common. One of them is to employ a collection of recordings, which for instance includes different volume levels of the same event. Otherwise one can also playback *one* recording with different volume levels or playback rate.

For continuous playback a so-called *loop* is used. At this, a sample is treated in such a manner that it can be played back seamlessly without

[14]The sine wave is tuned with some harmonics to make the feedback also audible.

[15]A seamless sample of coloured noise is looped, a low-pass filter corresponds to the users actions and is controlled in real-time.

breaks. To diversify the sound one can likewise crossfade other samples during the loop, which then again run seamlessly in a loop, or one changes playback speed and volume. For the case of a rolling sound in particular it is important to avoid unwanted periodicities.

Although sampling is a way to sonify the behaviour of moving objects, this procedure is nevertheless always restricted to the prerecorded material.

1.2.2 Synthesising Sound

An alternative for sampling is synthesising. There are various ways of synthesising, in general they can be grouped into "signal-based" and "source-model based" approaches, respectivly hybrids of those.

Signal-based means that the sound pressure signal, that is responsible for the auditory percept, is at focus. Here various methods (additive–, substractive synthesis, frequency modulation, ...) are used to synthesise a signal that fulfills known psycho-physical criteria necessary to produce a certain perception. If one knows which properties of the sound pressure signal make us perceive some specific event, one can synthesise such a signal and expect the same perceptual result.

The second approach is to model the sound source. Describing the sound source by mathematical means is an alternative way to describe the auditory event, although the relationship between the mathematical description of the source and the evoked preception might not be clear. Again, there are different approaches to finally generate a sound based on a mathematical description of the sound source behaviour. Obviously, one way is to describe the physical processes as detailed as possible and expect that the quality of perception comes with the quality of the simulation. Another approach is to include known psycho-acoustic findings in the development of a sound source model (to only simulate behaviour relevant for perception).

Synthesising a sound pressure signal based on physical considerations about the behaviour of the underlying physical processes is termed "physical modelling" or "physically informed synthesis".

1.2.3 The approach in this thesis: Physically informed synthesis by means of a modal object

The rolling sound model in this thesis is based on physical considerations (q.v. page 45). Differential equations are the fundamental instrument if one wants to describe the temporal and spatial progression of a physical system and they are in terms of auditive perception the adequate approach. The description of the system can be made pursuant to a number of requirements of accuracy, richness of detail or simplicity. Besides the derivation of differential equations the development of digital numerical algorithms that apply them is the most essential step. Technical considerations like computing time (for real-time computing) and accuracy play a decisive role. In this case real-time computation (besides low latency) is the dominating demand[16]. "Modal synthesis" is the technique chosen in this thesis.

For the description of a physical system by means of modal parameters it is essential that the state of the system can be described by a time-variant state vector $z(\vec{t})$ (in the "state space" Z). The temporal behaviour of the physical system is then described in the form

$$z'(t) = Az(t) + fext(t) \tag{1.1}$$

with A being the linear operator that maps $A : Z \longrightarrow Z$ and $fext(t)$ external impacts like forces, pressure, etc. It is important to stress the linearity of A, it is a mandatory precondition for the formulation of a modal description of the temporal behaviour of the system. The basic principle to solve the last equation is to describe A by means of its eigenvectors $\vec{v}$. If $z(\vec{t})$ lies inside of an eigenspace of A to the eigenvalue λ and $fext(t) = 0$ ("homogeneous" case) the equation simplifies to

$$z'(t) = \lambda z(t) \tag{1.2}$$

which can be solved by means of an exponential function: $zt = e^{\lambda t} z(0)$. In the case of a pair of conjugate-complex eigenvalues $\lambda_{1,2} = r \pm \omega i$ this describes

[16] The same applies for video rendering and reading the accelerometer data.

a damped sinusoidal wave of the form

$$x(t) = e^{rt} \sin(\omega t) x(0). \tag{1.3}$$

In general it can be shown (with reservations) that every free oscillation of the system described above consists of exponentially decaying oscillations[26].

1.3 Video games with rolling objects — an overview

1.3.1 Sound feedback

There are quite a few video games on the market that involve the steering or in some way interacting of/with a ball or ball-like object. They all include sound to illustrate actions on the screen. Nevertheless, most modern games restrict sound feedback to bouncing sounds, that is to say one can hear a sound when the ball bounces against boards or the underground. Only a few games (like "Marble Blast Gold" (2003), "Labyrinth" (2007), "Kororinpa" (2007)) give a feedback about the state of motion in terms of a rolling sound. In this respect "Labyrinth" (and similiar games) are an exception, because here the rolling sound plays a decisive roll amongst all in-game sounds and contributes decisively to the game experience. Mostly though, rolling sounds (or bouncing) are masked by other sound effects or spoken word.

Sampling seems to be the established way to produce all those sounds in video games.

1.3.2 Control & interaction

Except for few exceptions ("Marble Blast Gold") the player does not directly steer the ball, but tilts the game scenery and the ball reacts according to a gravitational force component parallel to the surface (downhill-slope force). In recent years classical controllers for video games, like gamepad, keyboard our mouse, are to some extent enhanced or replaced with/by acceleromet-

ers. Depending on the task to fulfill in a video game an accelerometer can establish a perfectly different experience for the player compared to a steering concept that relies on a keyboard. Interaction tasks that are geared to real world paradigms, like turning, moving or shaking an object, seem to be downright suited for controlling/steering via accelerometer. The situation of a rolling object that is balanced by the player, the issue of this thesis, is one of those cases. Many modern variants of "rolling-ball-games" make use of accelerometer techniques: "Super Monkey Ball Banana Blitz" & "Kororinpa" for Nintendo Wii, "Labyrinth" & "Super Monkey Ball" for the iPhone.

Chapter 2

The Game

2.1 Game concept

As indicated in the previous chapter, there are many aspects in human psycho-acoustic perception that could be subject to further research in future. Motivated by the *Ballancer* experiments (q.v. page 19) and inspired by recent trends in the field of interactive video games, the idea to create an experimental interface with game-like character for psycho-acoustical tests came into existence. Thus, the main motivation of this work is to establish a platform that facilitates those (psychomechanical) tests, in particular concerning sound feedback and gestural control, in a playful manner. On this background, continuous steering of a moving object is more at the centre of interest in this work than cognitive capabilities like recollection, finding a path, learning or orientation (as it is the case with a labyrinth game) — continuous gestural control interaction and sound feedback is the main focus.

2.1.1 Appearance

Accelerometers in modern entertainment devices give the opportunity to determine the inclination of the device (in this case a *MacBook*, which will be discussed later, q.v. page 36). In this manner it can be used as an input device for steering tasks in human-machine interaction. One apparent scenario is to route an object on a plane or on a track. Indeed, there are

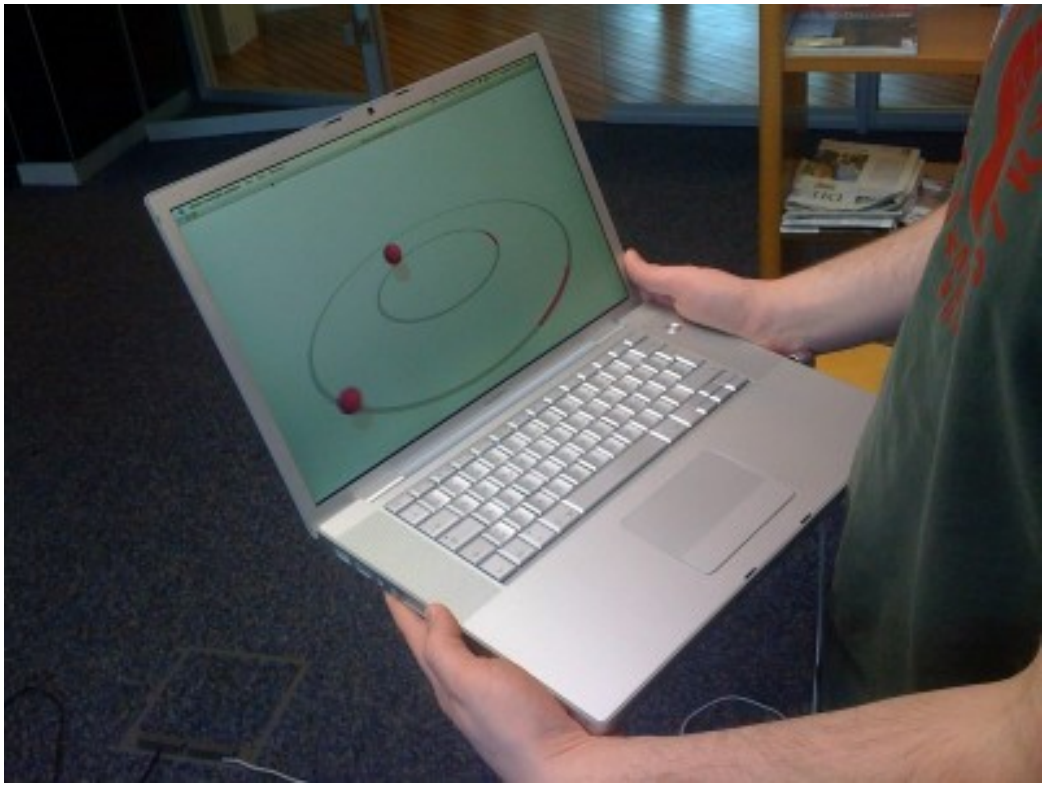

Figure 2.1: The *Orbits* game being played on a *MacBook Pro*.

quite a few games where the user has to balance an object (mostly a ball) in a maze-like virtual world[1]. However, prioritising gestural control over cognitive skills, we chose the scenario of objects tied to circular tracks moving under the effects of gravity and friction. Tilting the tangible device causes a period of oscillation around the equilibrium, until, due to friction, the objects eventually come to rest. Each object can be steered to any place on the circle by performing adequate balancing movements. The player can also make the object rotate in a circular manner on the track. As denoted in the thesis' title the game is called *"Orbits"*. Figure 2.1 shows the game running on a *MacBook Pro*.

The first realisation of the scenario consists of two separate tracks with two independently moving objects. Both circular tracks contain target areas,

[1]For example common marble games like "Labyrinth" (iPhone), "Super Monkey Ball" (iPhone, Wii), "Kororinpa" (Wii).

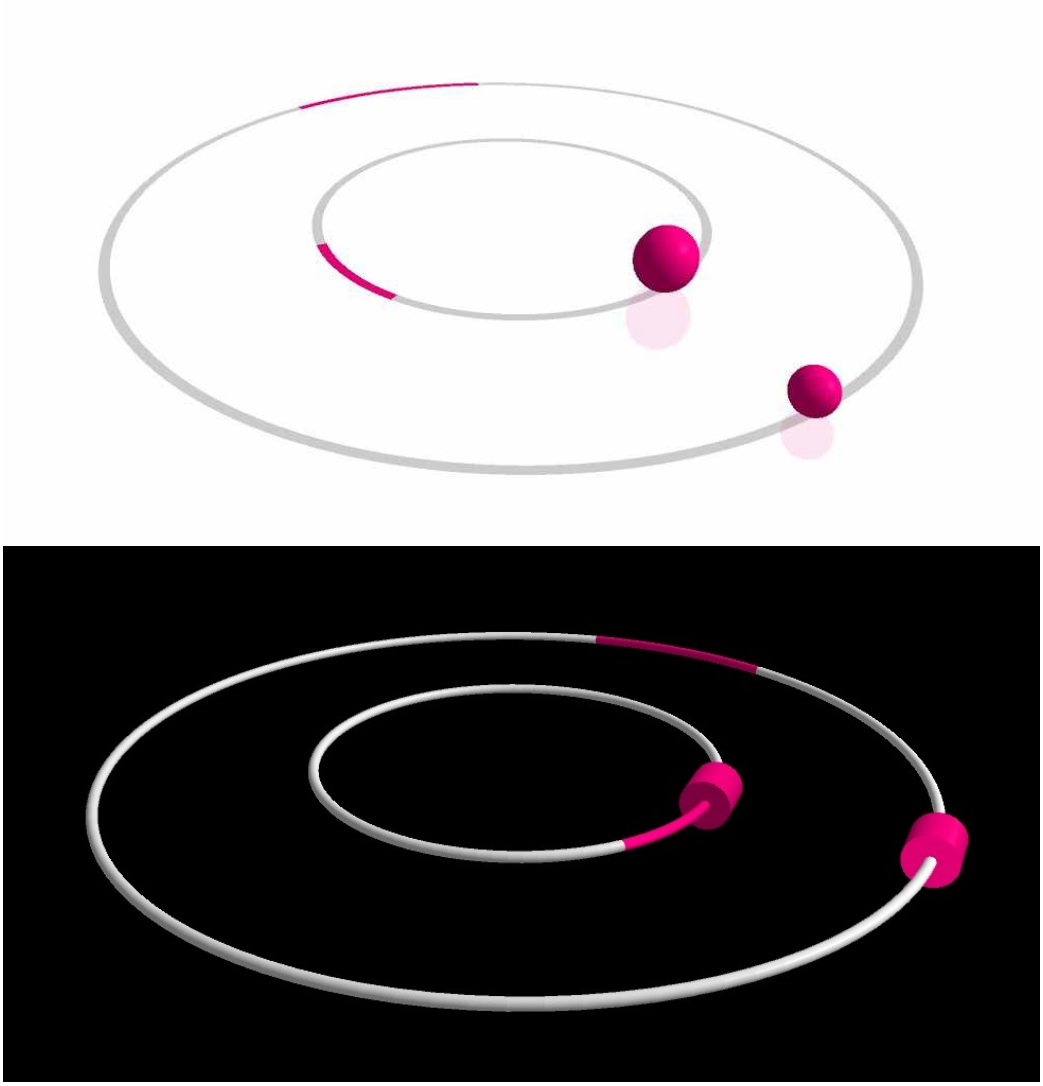

Figure 2.2: Screenshots of two visual elements of the *Orbits* game.

but the challenge does not consist in finding them since they are clearly visible and finding a path to reach these areas is trivial; please compare figure 2.2.

Using two tracks not only supplies possibilities to create different tasks to accomplish for the user (as will be described below), but also enables to present two different types of continuous auditory feedback at the same time. The audio feedback consists of a number of different sounds, ranging from rather realistic to abstract ones. Explorations of the influence of diverse types of acoustic feedback on human performance in control tasks and user

experience is one central interest for which this application has been designed. In the first prototype five different types of sound feedback have been realised:

1. a sound of a rolling ball,
2. a sound of a rolling ball on a surface with a pattern of two different degrees of roughness,
3. a sound of an object sliding on a surface,
4. a *band limited impulse train (BLIT)*,
5. a sound of single clicks with fixed spatial distance,
6. a sinusodial sound with inverted[2] relation between velocity of the ball and pitch.

For the visual appearance of the moving objects in the game three simple examples have been chosen as graphical representation. A ball, rolling on a circle, is our standard object. The second object is a cylinder which is sliding on a circular rail. These two examples can be seen in figure 2.2). The third object is a cone, sliding on a rail.

The object that is mainly used is the ball (primarily because of the focus on rolling sound). Some other sounds (e.g. the sliding sound) may seem unsuitable for the sonification of a ball's movement, a supposition that can be explored by means of the application. In this regard the sliding cylinders can give an impression of the interdependency of auditive and visual perception (for example by altering the objects but not the sound model or vice versa). Besides their function to visually maintain (or contradict) auditory feedback the cylinders are also useful to underline a stronger degree of friction.

2.1.2 The player's task

As game logic and complexity was not the focus, we kept the game as simple as possible in this regards, disclaiming usual methods of making

[2] "Inverted" in a sense that the manner of the feedback contradicts our experience about how the pitch changes when velocity increases: low velocity — high pitch and vice versa.

games challenging: time limits, an increasing number of tasks and goals to achieve... The principle task always remains the same (under varying conditions): the player has to steer both objects (e.g. two balls) into their target areas by tilting the device. The task is fulfilled once both balls are in their target areas at the same time. When the task is fulfilled the target areas change their position and the game continues.

There are different strategies to reach the goal. Besides carefully balancing both balls, one obvious strategy is to let them roll more or less randomly in circles until they by chance are in their target areas at the same time. By decreasing the size of the target areas gradually (e.g. every time a task is fulfilled) the second strategy becomes less effective and attentive careful balancing is necessary.

On the other hand letting the balls circulate at the track demands a movement pattern which is interesting as well. Periodically tilting the device in a rotatory manner and thus sustaining the velocity of the ball is a steering movement that clearly differs from a balancing task. It is here important to mention that objects react in different ways to inclinations due to different friction parameters whereby, with some skill, it is possible to let only one ball circulate and keep the other one fixed (more or less) at the same place. In contrast to balancing the ball, letting it rotate is a dynamical, periodical process. The player may therefore rely more on the dynamical information that he perceives aurally than on what can be seen at the screen. In this respect the "resonance-like" behaviour of the balls movement at periodic circular acceleration plays a decisive role. Of course the player can visually perceive in which way the ball reacts to a rotary tilting movement, but such kind of continuous, dynamical information is one strength of the auditive perceptual channel (compare e.g. [22]).

Therefore, we finally thought about a way to force the player to perform balancing and "rotation control" simultaneously and designed the following scenario as part of the game: the size of the target area of one of two moving objects (on parallel circular tracks) is determined by the velocity of the second object. While the position of the target is constant its size increases with the velocity of the object moving on the parallel track. Under these conditions,

in order to fulfill the task the ball that affects its neighbour's target size *must* move because a velocity of zero corresponds to a target size of zero. Letting said object rotate is now not the only way to reach the task but clearly (depending of course on the exact parameters of the coupling) the most obvious and the player is more or less forced to apply the following strategy:

- The faster the circulating ball moves the bigger becomes the target area of the other ball.
- If the ball with the varying target size reached it's goal the player has to keep it there.
- He would keep it there just for a moment, because the circulating ball frequently passes through it's goal.

In this scenario a rotary inclination of the device is a typical gesture the player performs to play the game — a circumstance that certainly supports the idea of porting the application to a smaller, truly handheld device.

Finally, we implemented another game mode, that we expect to force the player to waive random strategies. In this scenario, the objects (or maybe just one of them) can only move in one direction. In a setup where both objects can only move in opposing directions one has to balance carefully and smart.

2.1.3 Possibilities to create tasks for tests

Altogether, so far a number of instruments have been realised, that may be combined: a set of sound models, two visually different objects, friction parameters to make them also "feel" different and targets that can vary in position and size (also velocity-dependent target-size). Of course the point is not to arbitrarily combine those building blocks which are all related to each other. A higher setting of the friction parameter will rather suit a sliding sound and the cylinder visualisation, just as the ball visualisation naturally fits the rolling sound. Under the aspect of creating a preferably convincing

game one would try to set a combination that is as realistic as possible whereas for psychoacoustical tests it is also interesting to choose unintuitive (or even contradicting) combinations.

With some programming knowledge and after studying relevant sections in the source code (which will be discussed in the following section) it is possible to record desired data while a subject is playing the game. Such data could include: the inclination of the device, position and velocity of the objects, task times, etc. In the current state *Orbits* records and saves task times and the conditions under which they were measured: sound model, visual representation, friction ...

2.2 Technical realisation and sound feedback

The game has been designed as a standalone application that should run on all laptops of the *MacBook/MacBook Pro* series. These devices have been chosen because of the built-in accelerometer (q.v. page 40), their computing power (which is sufficient for real-time computed audio feedback) and finally because of the presence of common and well-documented programming interfaces on this platform (and their availabilty at the *Deutsche Telekom Laboratories* where this thesis has been written).

2.2.1 Architecture

The operation system (OS) on *Apple's* computers is called *Mac OS X* and it is based on the *UNIX* operating system. The native application programming interface (API) for dealing with sound in *Apple's Mac OS X* operating system is *"Core Audio"* [27]. The audio processing of the application *Orbits* is implemented as a "callback"[3] architecture with *Core Audio*[4], which means that *Core Audio* instructs our program to render audio output and write it into a buffer that can be referenced by an assigned memory address. It is important to understand that we are *not(!)* telling *Core Audio* to play back

[3](q.v. page 37 and 59)

[4]Here *Core Audio's "Audio Unit Framework"* is used, a plug-in architecture that can be used for effects and virtual instruments.

precasted audio samples — rather, audio output is continuously computed and (frame-wise) written into an audio buffer.

Graphical rendering is done with *"OpenGL"*, a cross-platform API for computer graphics. For scheduling of keystroke input and graphical output the cross-platform free library *"GLUT"* is used. *GLUT* also supports callback driven event processing, which means that we also have a callback function for video processing (besides the function for audio).

Audio

Core Audio is designed to handle all audio needs in Mac OS X [27]. It can be used to generate, record, mix, edit, process, and play audio (or MIDI data). *Core Audio* covers a number of features that make it a strong API, the most essential for this thesis are low latency through the signal chain, and interfaces for audio synthesis and processing called *"audio units"*. Audio units are plug-ins for handling or generating audio signals. *Apple Inc.* about audio units [27]:

> Audio units describe their capabilities and configuration information using properties. Properties are key-value pairs that describe non-time varying characteristics, such as the number of channels in an audio unit, the audio data stream format it supports (and) the sampling rate it accepts [...]. Each audio unit type has several required properties, as defined by Apple, but (one is) free to define additional properties [...]. Audio units also contain various parameters, the types of which depend on the capabilities of the audio unit. Parameters typically represent settings that are adjustable in real time, often by the end user.
>
> A signal chain composed of audio units typically ends in an output unit. An output unit often interfaces with hardware [...], but this is not a requirement. Output units differ from other audio units in that they are the only units that can start and stop data flow independently. Standard audio units rely on a "pull" mechanism to obtain data. Each audio unit registers a callback with its successor in the audio chain. When an output unit starts the data flow (triggered by the host application), its render function calls back to the previous unit in the chain to ask for data, which in turn calls its predecessor, and so on.

My application contains such an output unit. It is the only audio unit in use and its callback function renders the audio output directly without calling on another audio unit's callback function. A callback function is a function that is passed to another function as an argument, and will be called by this one under certain conditions. This means that in this case a function (declared and defined in the code — the *callback function*) is passed to the output unit. That function is capable of rendering audio data into an audio buffer of specified length. Whenever the output *audio unit* decides that an audio buffer needs to be rendered, it calls the callback function. The most important thing to underline is that the callback function is the place where single audio frames are computed in real-time, no prerecorded samples are used. The way audio data is computed is discussed at page 43 and a section dedicated to the source code of audio callback function can be found at page 59.

Video

An application that comprises 2D or 3D graphics, running on different hardware and/or platforms, relies on cross-platform APIs for computer graphics. *OpenGL* is such a cross-language, cross-platform specification. Today it is the industry standard for high performance graphics[5] [28]. Clearly, the *Orbits* application includes 3D computer graphics. The underlying code is written according to the *OpenGL* specification.

The *"OpenGL Programming Guide"* [29] is a prevalent reference on the topic of *OpenGL* programming and has been my guide. I will not describe every aspect of *OpenGL* here, since this is not a programming guide, but I want to give a short overview about some basic principles.

OpenGL has been designed as a so-called "state machine", which means that functions are called without passing all necessary parameters. Instead, the *OpenGL state machine* uses the same values until the according states are changed. The reason for this design is that almost every change of the drawing mode involves costly reorganisation of the graphics pipeline, which

[5]However, Direct3D is a prominent alternative on *Windows* operating systems.

is therefore avoided if possible. For example, thousands of vertices can be rendered with the same colour, which is set only once in the beginning. Sometimes a state in the state machine never changes (like the source of light in the *Orbits* application).

Modeling, viewing and projection transformations are represented by matrix multiplications in *OpenGL* [29]. The modeling transformation is used to position and orient models (e.g. rotate, translate, scale). The viewing transformation is analogous to positioning and aiming a camera. And eventually, specifying the projection transformation is like choosing a lens for a camera: this transformation determines what the field of view or viewing volume is (and therefore what objects are inside it). The projection matrix also determines *how* objects are projected onto the screen (perspective projection or orthographic). Since matrix multiplications are (in general) not commutative it is a crucial task to structure the code properly and keep track of the transformations (note: *OpenGL* is a "state machine").

OpenGL is designed to be independent of any window system or operating system (it contains no commands for opening windows or reading user input). However, it is impossible to write a graphics program without at least opening a window. The *OpenGL Utility Toolkit (GLUT)* is a (support) library of utilities for *OpenGL* programs [30]. It includes functions to perform window definition and control or monitoring of keyboard and mouse input. *Orbits* uses *GLUT's* window control capabilities (for fullscreen video rendering and a menu window) and instructs it to schedule (q.v. page 42) the video rendering using a callback function. Furthermore, *GLUT* functions handle events from the keyboard (for example to close the application).

One particular design characteristic of *GLUT* is the function `glutMainLoop()`, it is *GLUT's* event handling loop. This function never returns, which is an obstacle if one wants to create its own event loop; and besides that, *GLUT* cannot exit the event loop — circumstances that may not be satisfactory for full-featured *OpenGL* applications. Both issues are not critical for *Orbits*, which will be discussed in the scheduling section (q.v. page 42).

Accelerometer data

An accelerometer is an "instrument that measures the rate at which the velocity of an object is changing (i.e., its acceleration). Acceleration cannot be measured directly. An accelerometer, therefore, measures the force exerted by restraints that are placed on a reference mass to hold its position fixed in an accelerating body" [31]. Accelerometers are increasingly being incorporated into personal electronic devices. They are used to align the screen depending on the direction the device is held[6], they support motion[7] or steering[8] input to video games or they serve as pedometers[9]. Some laptops feature an accelerometer to detect drops[10], which is with respect to this thesis of particular importance.

Since 2005 *Apple* uses accelerometers to protect the built-in hard drives in their portable systems. This includes "all *Intel*-based *Apple* portables such as the *MacBook*, *MacBook Pro*, *MacBook Air*, *PowerBook G4* computers starting with *PowerBook G4* (12-inch 1.5GHz), *PowerBook G4* (15-inch 1.67/1.5GHz), *PowerBook G4* (17-inch 1.67GHz), and *iBook G4* computers starting with *iBook G4* (Mid 2005)" [32]. *Apple* calls this feature *"Sudden Motion Sensor"*:

> The Sudden Motion Sensor is designed to detect unusually strong vibrations, sudden changes in position or accelerated movement. If the computer is dropped, the Sudden Motion Sensor instantly parks the hard drive heads to help reduce the risk of damage to the hard drive on impact. When the Sudden Motion Sensor senses that the Apple portable's position is once again stable, it unlocks the hard drive heads, and you are up and running within seconds.

Specifications about the accelerometer sensors used are hardly available. Officially *Apple* does not support access to these built-in sensors to the pub-

[6] For example: smartphones, digital audio players, personal digital assistants, ...

[7] *Nintendo Wii* et al.

[8] *Sony PlayStation 3* et al.

[9] Notable manufacturers: *Omron*, *Sportline*, *Garmin Foot*. Personal electronic devices incorporating pedometer functionality: *Apple iPod Nano*, *Nike&iPod*, *Nokia 5500 Sports Phone*, *Sony Ericsson W710 walkman phone*, *Nintendo DS*, ...

[10] Notable technologies: *Active Protection System* by *Lenovo*, *Sudden Motion System* by *Apple* [32]

Figure 2.3: The *Orbits'* calibration routine is strongly recommended before using the application.

lic. Nevertheless, the internet discloses several examples of free software that uses *Apple's* accelerometer[11]. Some technical information about the sensors can be found here: [33]. There are also two open-source libraries available: *SMSLib* [34] and *UniMotion* [35]. However, we decided to use none of them. Basically because we wanted to have access to the raw sensor data and implement our own calibration algorithm.

Currently (2010) three different sensor types have been reported in *Apple* portable computers, known as `SMCMotionSensor`, `IOI2CMotionSensor` and `PMUMotionSensor`. They have different data types, the first 16-bit integer and the latter two 8-bit integer, which means they are variably sensitive. This is one obstacle that requires a calibration to use the application on different hardware. The second difficulty is the orientation of the sensors. The x-, y- and z-axis of those accelerometers can be mirrored depending on the model of the notebook. The *Orbits* application has a built-in calibration routine that is capable of graduating those differences. The calibration routine measures sensor values in three specified positions of the device (please compare fig. 2.3) and then transforms all subsequent sets of sensor data (by means of a matrix multiplication/base change).

It is strongly recommended to enter the calibration mode *once* before the application is used. Calibration data is stored and will be loaded automatically next time the application is lauched. Nevertheless, the *Orbits* application possesses orientation data of some (few) *MacBook*/*MacBook Pro* models, which means that if the application is launched on one of those models for the first time, the axis orientation is automatically adjusted and the behaviour

[11] For example *SeisMac* from *Suitable Systems*, *LiquidMac* by Oriol Ferrer Mesia

of the game “feels” correct (though further calibration is suggested).

2.2.2 Scheduling

The application comprehends physical calculations (movement of objects), real-time computed audio feedback and visual representation of a scene. Scheduling of these tasks have to be carefully considered. This section gives an overview about the basic structure of the program and how the single processes are related to each other.

As described above, when the application is started it registers itself as a client for audio (*Core Audio*) and video processing (*GLUT*) and detects the motion sensor. All physical simulation is computed with audio rate.

Regarding the physics there are basically two tasks to perform: calculate the movement of the object along the track and synthesise the sound. The rolling sound model is predicated on physical considerations of the interaction between ball and surface (q.v. page 45). Besides the macroscopic movement of the ball that can be seen, there is a microscopic behaviour that can only be heard, both must be computed. One approach (actually the most common, in combination with sample-based audio) would be to use a physics engine for the calculation of the ball's macroscopic movement with a defined temporal resolution and synthesise audio feedback on the basis of the result. In this case the quality of the feedback would strongly depend on the update rate of the used physics engine. The opposite approach would be to unify macroscopic and microscopic movement in one model to obtain both as the output of the same algorithm, which would make movement and sound much more coherent but on the other hand is also much more difficult to achieve. In the *Orbits* simulation a method is chosen which is a compromise of these two solutions. We make use of two separate models but both are always calculated in parallel within the same cycle of the audio processing, which means that we update the position of the ball with audio rate and instantaneously calculate the next sample frame of audio feedback in dependence of the position and velocity parallel to the track.

Structure and scheduling are as follows: all physical objects and al-

gorithms of the current active simulation are contained in a data structure that is called "scene". In fact there is a number of different scenes with different objects, sounds, etc., but only one scene is active at a time (the others are stored in memory, ready to be activated). Only the currently active scene is updated, which means that the physical behaviour of the objects and the sound output is computed (with audio rate). These updates are performed within the audio callback function. The video callback function induces the rendering of the scene by accessing relevant data computed in the audio loop.

The buffer size of our audio callback routine is fixed at 1024 samples and the audio feedback is calculated with $44100Hz$ with the result, that the audio callback function is called with about $43Hz$. Each time the audio callback function is executed the accelerometer data is read *once*, *1024* updates of the scene are performed and the buffer is filled with samples. There is no direct control about the exact moment when the audio callback function is called, one only knows that it happens in average 43 times per second. By relying on the callback cycle of the audio driver a certain jitter is unavoidable which I however accept since an average update rate of $43Hz$ is quite high for accelerometer data used for balancing control.

As already mentioned the video rendering is also executed by means of a callback function, for which I chose a fixed rate of $50Hz$. Since values relevant for graphical output (object position...) are computed in the audio callback cycle the same remarks apply here as for updates of accelerometer data.

2.2.3 Sound feedback

Sound feedback in the *Orbits* application is not just fancy supplement but one of the central interests. The application is designed to facilitate new ways to explore the effects of the sound on the player. Does sound enhance his/her performance? Does it affect his attitude, his perception of the game? As described (q.v. page 43), recent experiments with the *Ballancer* interface showed that the perception of velocity plays a key role in this context, it can improve a player's performance in a control task. In addition it is interesting

to reconsider the influence of different degrees of abstraction of sound feedback for which reason a number of different sound models are employed in *Orbits*.

As already mentioned this game is based on steering objects on plains affected by gravity. Hence the scenario of a rolling ball is the starting point of our considerations about *sound models* for the game, about their qualities and their implementation. The "rolling sound model" considers the profile and the vibration of the surface as well as the vertical interaction of the surface and the rolling ball (all sound models are described on the subsequent pages). It is our informal supposition that the detailed nature of this vertical interaction plays a strong role in the auditory perception of rolling: it appears that the mixture of short periods of microscopic bouncing and longer periods of contact between the rolling object and the surface is an important factor for sounds to be classified by human listeners as "rolling". The first step to simplify the rolling sound algorithm is therefore to disregard the vertical interaction. The "sliding sound model" only considers the vibration of the surface and the sweeping of the object across the surface's profile.

One very specific scenario of sliding, as a next step, is given by the needle of a record player. Clearly, if the needle of the record player drives faster across the record we perceive the increasing velocity. Of course the profile of the track on that record is the dominating factor, vibrations of the record are hardly audible. On the basis of this metaphor we included a more abstract sound model incorporating a periodic (highly artificial) surface profile that produces a bandlimited signal when sampled ("BLIT").

Finally we totally conflict with the starting point: we eliminate all ecological influences and even contradict to our experience of how the pitch of a sound of continuous interaction behaves. The "inverted sound" is a tone with a sinusoidal waveshape and frequency inversely proportional to the velocity of the object. The higher the velocity of the moving object (ball, cylinder) the lower the frequency of the sine and vice versa.

Rolling sound

The rolling sound model affords a high degree of realism, it is the natural acoustic complement of what can be seen on the screen when the balls are rolling at their tracks. The rolling sound model embraces the vibrations of the surface, the surface's profile and the vertical contact interaction of the vibrating surface and the ball. Details of the algorithm are described in dedicated articles by M. Rath ([21] [26]), in the following I try to give an idea of the main points.

In the *Orbits*-scenario two objects are interacting with each other: the ball (resp. a cylinder) and the surface. There are different approaches to describe the behaviour of vibrating and interacting objects; we chose the modal approach (compare [26]) for a model of the inner resonance behaviour of the vibrating surface. We did that on the assumption that the surface's vibration plays the dominant part in the sound that we hear when an object is rolling or sliding across it while the vibration of the solid rolling object itself (e.g. a marble) can hardly be heard directly. The modal description of the surface (in the following "modal object") is the source where our (rolling) sound output comes from: the velocity of the surface is written into the audio buffer.

If a force is applied to the modal object is starts vibrating and dies away after some time due to inner friction. As the ball is bouncing it applies a force to the surface. A rolling ball or a sliding object is continuously impacting to the surface because natural surfaces are not perfectly smooth. There are different approaches to model that contact. Regarding the profile as a (force) input signal to the modal object is one way, discussed in the next subsection. A more accurate approach is to include physical and geometrical considerations: the rolling sound model.

The basic principle of the rolling sound has already been introduced in [21] and implemented as a patch for *Pure Data*[12]. Recently an improved contact model has been presented in [26]. In this thesis the improved algorithm

[12] A graphical programming language for the creation of interactive computer music and multimedia works.

of [21] is implemented based upon the contact model of [26] written in the programming language *C++*[13]. In the following I give a short summary about the ideas and methods.

Despite some interesting psychoacoustic studies (e.g. [12]) the question of how to exactly describe the acoustic features responsible for a sound event to be perceived as "rolling" is still not perfectly answered. This observation supports a physics-based approach in the synthesis of rolling sounds. We look at "rolling" as a process of sustained bouncing, eventually microscopic bouncing. The second aspect is that we allow the ball to penetrate the surface, thus a contact is not a change of state at a discrete moment in time. Contacts are continuing processes that have to be modelled. To sum up, the interaction can be both continuous and sporadic: there are moments of contact, in particular continuous contact, and moments when surface and ball are not in contact.

The modelling of the contact is one key building block of the rolling sound model. [26] introduces a method of energy-stable modelling of continuous or repeated contact in dynamical situations, a method that is also applied here. The basic idea is to use a modal description for the period of contact too. If there is no contact the state of the surface is described by a modal object, the ball's state is stored separately. In case of contact, surface and ball are regarded as one object, which is described by means of modal parameters as well. The model of rolling includes frequent "switching" between both configurations.

Besides physical considerations (improving the contact model) there are also geometric aspects to consider. Although it is a constrain that the contact only takes place at one contact point, the ball, of course, has a geometric attribute that should not be disregarded: its radius. The profile of the surface is simplified to a white noise distribution of profile points (with constant horizontal distance, but scattered vertical position). One can imagine that the ball cannot touch every point since it is too big to fill each gap. Therefore the profile needs to be filtered. Instead of bandpass filtering (as in the next

[13]Object-oriented programming, portability and the speed of the compiled code were the main reasons to chose C++.

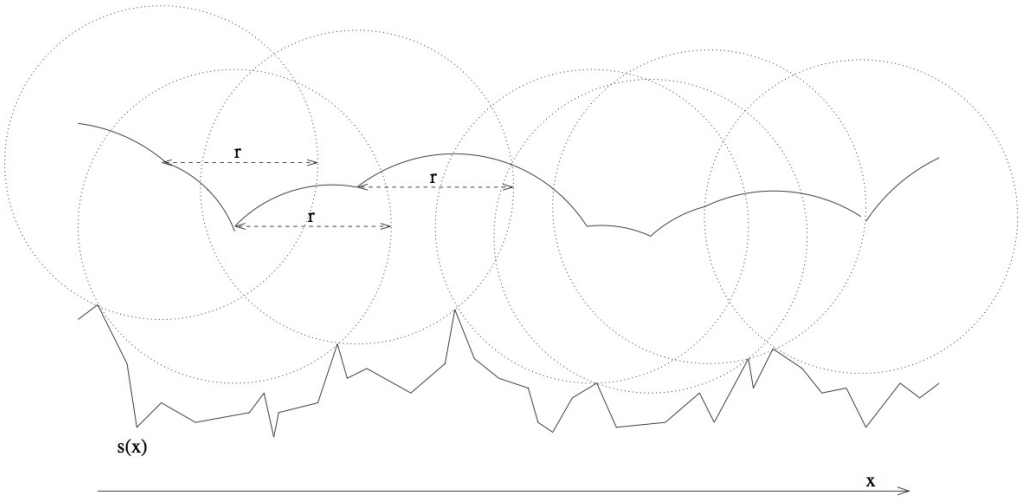

Figure 2.4: The hypothetical trajectory of a rolling object as it would perfectly follow a surface profile $s(x)$ (Rath [21])

section) we have to filter the "profile signal" based on geometrical considerations. The exact procedure is explained in [21] and for now called "rolling filter". A sketch from that paper can be seen in figure 2.4. To sum up, the result is a profile signal consisting of arcs of circles which is the path that a ball would follow if it was perfectly succeeding the surface's profile.

Since the rolling sound model contains the modelling of contacts (and resulting forces) based on physical considerations the profile is regarded as a position-dependent (and thus timevariant) input parameter to the contact model. The contact model described above involves a continuous computation of the distance between ball and surface. At this point the profile of the surface is incorporated as a vertical distance offset. Besides several "white noise variants" as profiles for surfaces the rolling filter technique also enables more complex profiles like saw tooth surfaces[14] that enable interesting possibilities to imitate natural surfaces.

[14]combined with above-mentioned white noise "micro structure"...

Sliding sound

The first degree of abstraction is to neglect the vertical interaction between the object and the surface. Instead of a precomputed profile we use bandpass filtered white noise. The centre frequency of the filter is proportional to the velocity of the ball, the bandwidth is kept constant. The vibration of the surface is emulated by means of modal parameters as described in the previous section.

In this model the vertical interaction between object and surface is not considered, as well as the geometry of the object (e.g. the ball's shape). We only incorporate the surface's profile and simplify the contact between object and surface in various ways: 1. object and surface are *always* in contact, 2. the object cannot penetrate the surface and 3. there is only one point of contact. 1. and 2. imply that the object is perfectly strobing the outline of the profile, hence we utilise the profile of the surface as (force) input signal to the modal object (in a way similar to some previously used algorithms, compare e.g. [36]). We exert a bandpass filtered white noise as signal. The centre frequency of the filter is proportional to the (horizontal) velocity of the ball, the bandwidth is kept constant; the sound model can be regarded as a modal resonator. In accordance with our informal supposition on the characteristics of contact sound the audible result resembles the sound of a sliding object.

BLIT

The bandlimited impulse train is the second degree of abstraction. The strobing of the profile is the idea behind this sound model. It disregards vibrations of the surface as well as vertical interaction. A periodic profile is the basis for the perception of a pitch. Like the needle of a record player we strobe the profile, the perceived pitch depends on the speed of the needle (resp. ball, cylinder). Important criteria for the profile are that it has to be bandlimited but not sinusoidal and preferably easy to compute.

In 1975 Moorer[15] introduced an economic method to synthesise band-

[15]Moorer mentions David Lewin of Harvard University (and maybe others) having made

limited complex audio spectra by means of Discrete Summation Formulae (DSF) [37]. This synthesis technique is based on the possibility of expressing certain sums of trigonometric functions in a closed forms that are much more economic to compute than the sum itself. DSF allows the synthesis of bandlimited periodic signals. By varying parameters of the underlying formulae a wide class of bandlimited waveforms, including a bandlimited impulse train (BLIT), can be generated (as shown by Stilson and Smith [38], Välimäki [39] and others).

For our specific purpose a restricted number of parameters in the BLIT approach are sufficient and we make use the following formula which is a simplified (by omitting phase parameters) version of [37], p. 3:

$$\sum_{k=0}^{N} a^k \sin(k\beta)) = \frac{a\sin\beta - a^{N+1}(\sin((N+1)\beta) - a\sin(N\beta))}{1+a^2-2a\cos(\beta)}.$$

Here a is the "amplitude ratio" of the harmonics and influences the balance of higher and lower frequencies while N is the number of components, thus determining the bandwidth of the signal.

Inverted sound

The inverted sound provides the highest degree of abstraction. Still the pitch is the way to sonify the velocity of the object, but here pitch and velocity are *inversely* proportional, a certainly unnatural dependence. The second factor that makes the sound artificial is the sinusoidal waveshape. The reciprocal proportionality will result in a very high pitch for a large part of playing situations. If the pitch is close to the upper boundary of the hearing range it is adequate to synthesise a sine instead of a bandlimited signal, because both can hardly be distinguished.

---

the same discovery independently at the same time since the underlying mathematical formulae are quite old.

Chapter 3

The Source Code

I do not want to introduce every line of code in this section, but some parts are worth explaining and especially helpful to fully understand the issues discussed in chapter 2 (q.v. page 36). With the help of the source code, I will depict the structure of the application from a different point of view. Only the basic architecture will be discussed and no algorithms.

The source code has been written in *C++*. If you have basic programming knowledge, you will probably not experience difficulties to understand the code introduced below. In my opinion, the syntax of C++ is understandable, even when you are not an expert. Therefore, I will refrain from translating parts of the code into pseudo-code. In addition, to not confuse the reader, I will present only excepts[1] with (for comprehension) relevant parts[2].

Due to the complexity of the project, it is, of course, split into separate files. They are compiled and linked by means of a *makefile*[3].

A complete documentation of the source code generated with *Doxygen*[4] and the source code itself is included in the attachment of this thesis.

[1] Excerpts are marked with *[...]*, which is not valid C++ code.

[2] For full, detailed comprehension I recommend to read the real source code. Due to the excerpts, some things in the code snippets presented here do not make sense from a programmer's point of view.

[3] "In software development, *make* is a utility for automatically building executable programs and libraries from source code. Files called *makefiles* specify how to derive the target program from each of its dependencies." (Wikipedia, [40])

[4] Doxygen is a source code documentation generator tool [41].

3.1 The main files: `orbits.hpp`, `orbits.cpp`

The main files of the application are `orbits.hpp`, including the declarations, and `orbits.cpp`, including the definitions of all callback functions (which are described in the following sections), a function to specify/create scenes and the `main()` function (plus some other functions that will not be discussed here). I will start my introduction to the source code with a brief discussion of these two files. `orbits.hpp`:

```
1  [...]
2
3  struct t_params4audio_callback
4  {
5    t_audio_param* p_audio_param;
6    c_scene** pp_scene;
7    t_game_state* p_game_state;
8    t_motion_values* p_motion_values;
9    c_motion_sensor_mac* p_ms_mac;
10 };
11
12 struct t_global_data
13 {
14   t_game_state game_state;
15   c_scene* p_scene;
16   c_scene* a_scenes;
17   c_scene_element** aa_elements;
18
19   c_scene_element el_in_0;
20   c_scene_element el_in_1;
21   c_scene_element el_in_2;
22   c_scene_element el_in_3;
23   c_scene_element el_in_4;
24   c_scene_element el_in_5;
25   c_scene_element el_out_0;
26   c_scene_element el_out_1;
27   c_scene_element el_out_2;
28   c_scene_element el_out_3;
29   c_scene_element el_out_4;
30   c_scene_element el_out_5;
31
32   t_audio_param audio_param;
33   t_video_param video_param;
34
35   t_motion_values motion_values;
36   c_motion_sensor_mac ms_mac;
37   c_scene_calibration_mode scene_calibration_mode;
38   int calibration_flag;
39
40   int main_window_ID;
41   int gui_flag;
42   c_orbits_gui* p_main_gui;
43   c_glut_screen_terminal glut_terminal;
44   c_glut_screen_terminal glut_highscore_table;
45   c_glut_screen_terminal glut_info;
46   c_2D_status_bar status_bar;
47   c_2D_number _2D_current_time;
48   c_2D_number _2D_average_time;
49
50   c_modal_object_simple notification_sound;
51
52   int record_flag;
```

```
53   int init_scenes_from_record_flag;
54   c_savegame savegame;
55   string filename;
56 };
57
58 void read_ms(c_motion_sensor_mac* p_ms_mac, t_motion_values* p_motion_values);
59
60 void set_record_state(int record_flag);
61 int save_config();
62 int load_config();
63 int open_savegame(char* filename);
64 int close_savegame(char* filename);
65 void close_application();
66
67 void audio_callback_function(t_params4audio_callback* p_params4audio_callback,
68                              int frame_start, int frame_end,
69                              void* left_channel, void* right_channel);
70
71 void callback4coreaudio (void* p_params4audio_callback_val, UInt32 inFrames,
72                          void* left_channel, void* right_channel);
73
74 [...]
75
76 void draw_orbits(void);
77 void redisplay_orbits(int value);
78 void reshape_orbits(int width, int height);
79 void keyboard_orbits(unsigned char key, int x, int y);
80
81 [...]
82
83 void calibration_callback(int value);
84 void motion_sensor_callback(int value);
85
86 void initGL(void);
87 void toggle_gui(void);
88 void toggle_fullscreen(void);
89 void init_scenes(double timestep);
90 void open_scene(int i);
91
92 void build_highscore_table();
93
94 int main(const int argc, char** argv);
```

Two structs are declared in `orbits.hpp`: `t_global_data` (line 12) and `t_params4audio_callback` (line 3).

`t_global_data` is a container for all global variables. It contains a `t_game_state` struct (line 14, q.v. page 74), references to *scenes* (line 15 and 16, q.v. page 67) and *scene elements* (line 17 - 30, q.v. page 68), a `t_motion_values` struct (line 35) and a `c_motion_sensor_mac` object (line 36), several components of the graphical user interface (GUI) and a `c_savegame` (line 54).

The `t_params4audio_callback` struct contains relevant parameters for the audio callback function (q.v. page 59 and page 56). Besides other parameters (number of frames, audio buffer memory address), the audio callback function has one parameter that points to a specified memory address. This

pointer is dedicated to supply the function with all necessary data. Objects and variables declared elsewhere in the code are not known to the callback function. Since the callback function has to update the state of all objects, it needs references — that is the purpose of `t_params4audio_callback` (one very ugly alternative is to declared those objects as global variables). t_params4audio_callback contains a references to an `t_audio_param` struct which contains parameters to render the audio output (line 5, q.v. page 56), a reference to the current scene (line 6), the `t_game_state` struct (line 7), the `t_motion_values` (line 8) and to the `c_motion_sensor_mac` object (line 9).

Among the function declarations, the audio callback function (line 67) and the graphics rendering callbacks (line 76 and 77) can be found. There are also some other important functions (`initGL()`, `init_scenes()`, `read_ms()`, ...) declared in `orbits.hpp` and defined in `orbits.cpp`. They will be discussed in the subsequent sections.

Thus, the only one function, that will be introduced here, is `main()`. The `main()` function is the schedule of the initialisation process.

```
1  int main(const int argc, char** argv)
2  {
3    srand(time(NULL));
4    p_global = new t_global_data;
5
6  [...]
7
8    if (!p_global->init_scenes_from_record_flag) p_global->filename = "local.sav";
9    if (load_config())
10     {
11       VERBOSE_PRINT(">_loading_preferences..._success\n");
12     }
13
14 [...]
15
16   /* init GL stuff */
17   initGL();
18   p_global->gui_flag = 0;
19   p_global->video_param.fullscreen_flag = 0;
20   toggle_fullscreen();
21
22 [...]
23
24   /* init audio */
25   int return_code;
26   init_scenes(1./SAMPLE_RATE);
27   t_params4audio_callback params4audio_callback;
28   params4audio_callback.p_audio_param = &p_global->audio_param;
29   params4audio_callback.pp_scene = &p_global->p_scene;
30   params4audio_callback.p_game_state = &p_global->game_state;
31   params4audio_callback.p_motion_values = &p_global->motion_values;
32   params4audio_callback.p_ms_mac = &p_global->ms_mac;
33
```

```
34    t_params4coreaudio params4coreaudio = set_params4coreaudio(SAMPLE_RATE, AUDIO_BUFFER_SIZE);
35    return_code = create_audio_unit(&params4coreaudio, &params4audio_callback);
36
37    if (return_code) exit(1);
38    VERBOSE_PRINT("> Press [m] for main menu, [o] for options or [q] for quit.\n");
39
40    Glow::MainLoop();
41 }
```

At first a `t_global_data` struct is created and `p_global` is the reference to it (line 4). This struct contains data and (still uninitialised) objects as described above. A file name of local saved games is specified in line 8 and a locally saved config file is loaded in line 9.

The next important step follows in line 17 (and subsequent lines). `initGL()` is a function that sets up the graphical processing by GLUT, which is discussed at page 62 and `toggle_fullscreen()` switches to full-screen mode. However, at this point of the program no window for *OpenGL* rendering is open yet, because we cannot start rendering graphics before the scenes are initialised.

The internal processing of the scenes (including the *sound models* and physics of the objects, q.v. page 67) ultimately depends on the specifications of the audio rendering environment, more precisely the *sample rate*. In line 26 `init_scenes()` is called with that sample rate as a parameter[5]. In lines 27 - 32, the `t_params4audio_callback` struct (q.v. page 56) is initialised, and in line 34 a `t_params4coreaudio` struct (q.v. page 55). This may seem confusing, but theses sctructs and their purpose is subject of the next section. The final step to initialise the audio processing is to create an *audio unit* (q.v. page 37) which is done by calling the `create_audio_unit()` function in line 38 (q.v. page 55).

Now the scenes have been initialised, the audio processing has started (and thus accelerometer data processing, simulation of the physics, game logic, etc.) and the application is ready to render graphics and handle user input (as keystrokes). Like mentioned previously, graphics rendering and the event handling is done by GLUT, hence the last function call in `main()` is a call is related to GLUT. `Glow::MainLoop();` in line 40 starts the event pro-

[5]Strictly speaking, the paramter is the reciprocal of the sample rate, in this code called *timestep*.

cessing loop. This indirectly starts the `glutMainLoop()`, which is explained in more detail at page 62.

3.2 Audio processing

Orbits initialises an audio unit to process its audio output. The initialisation process of the audio unit is defined in `core_audio_functions.cpp` and the callback function(s) are defined in `main.cpp`. Before I start to describe the initialisation and processing routine step by step I would like to introduce three structs that have been mentioned before: `t_params4coreaudio`, `t_params4audio_callback` and `t_audio_param`.

3.2.1 The `t_params4coreaudio` struct

`t_params4coreaudio` is declared in `core_audio_functions.cpp`:

```
1  struct t_params4coreaudio
2  {
3    Float64        sample_rate;
4    UInt32         audio_buffer_size;
5    SInt32         num_channels;
6    SInt32         which_format;
7    UInt32         format_ID;
8
9    UInt32 format_flags;
10   UInt32 bytes_in_a_packet;
11   UInt32 bits_per_channel;
12   UInt32 bytes_per_frame;
13
14   UInt32 frames_per_packet;
15
16   AudioUnit output_unit;
17 };
```

`t_params4coreaudio` contains all parameters of the audio unit (sample rate, audio buffer size, number of channels, ...) and an `AudioUnit` object. It is not necessary to go to much into detail at this point, the one important thing to understand is that this struct is a container data type to specify an audio unit and it contains an audio unit itself, which is initialised with these parameters in `create_audio_unit()` (q.v. page 57). The `t_params4coreaudio` struct is only created and used (once) to initialise the output audio unit.

3.2.2 The t_params4audio_callback struct

The second important audio struct is called t_params4audio_callback and, as showed before, declared in orbits.hpp:

```
1 struct t_params4audio_callback
2 {
3   t_audio_param* p_audio_param;
4   c_scene** pp_scene;
5   t_game_state* p_game_state;
6   t_motion_values* p_motion_values;
7   c_motion_sensor_mac* p_ms_mac;
8 };
```

As mentioned at page 52 the audio callback function expects one reference to a memory address as a parameter. This memory address, a so-called *pointer*, is eligible for election by the programmer and it is the one clean way to reference to objects (created elsewhere in the code) from inside the audio callback function. Since all physical processes are updated in the audio callback function it obviously needs references to the objects involved.

This is the purpose of t_params4audio_callback. A reference to a t_params4audio_callback struct is the pointer that is passed to the callback function. t_params4audio_callback contains references to a scene, to the game state (q.v. page 74), the motion values, the sensor and a reference to a t_audio_param struct.

3.2.3 The t_audio_param struct

The t_audio_param struct is declared in param_4_audio_rendering.hpp and it contains parameters that specify how to (audio-) render a scene and is passed as an argument every time a scene has to be rendered:

```
1 struct t_audio_param
2 {
3   double volume;
4   int mute_flag;
5
6   int stereo_flag;
7   double stereo_horizon;
8
9 [...]
10 };
```

It includes the volume of the application, a flag that specifies whether the application is muted and two values regarding stereo. The stereo_flag indicates if stereo output has to be computed and stereo_horizon specifies

the “stereo range”. Roughly spoken, if an object is at the total right edge of a scene, the `stereo_horizon` value specifies the volume of the left ear speaker.

3.2.4 The audio initialisation process

Now that the audio structs are introduced we can have a deeper look into the audio chain. The first step (after setting up the structs in `main.cpp`) is the audio initialisation process defined in `core_audio_functions.cpp`:

```
1  [...]
2
3  OSStatus core_audio_process_function(
4     void* p_params4audio_callback, AudioUnitRenderActionFlags* ioActionFlags, \
5     const AudioTimeStamp* inTimeStamp,\
6     UInt32 inBusNumber, UInt32 inNumberFrames,\
7     AudioBufferList* ioData)
8  {
9    callback4coreaudio(p_params4audio_callback, inNumberFrames,
10                      ioData->mBuffers[0].mData, ioData->mBuffers[1].mData);
11   return noErr;
12 }
13
14 int  create_audio_unit (t_params4coreaudio* p_params4coreaudio,
15                         void* p_params4audio_callback)
16 {
17 [...]
18
19   err = OpenAComponent(comp, &p_params4coreaudio->output_unit);
20
21 [...]
22
23   AURenderCallbackStruct input;
24   input.inputProc = core_audio_process_function;
25   input.inputProcRefCon = p_params4audio_callback;
26
27   err = AudioUnitSetProperty (p_params4coreaudio->output_unit,
28                               kAudioUnitProperty_SetRenderCallback,
29                               kAudioUnitScope_Input,
30                               0,
31                               &input,
32                               sizeof(input));
33
34 [...]
35
36   AudioStreamBasicDescription streamFormat;
37   streamFormat.mSampleRate = p_params4coreaudio->sample_rate;
38   streamFormat.mFormatID = p_params4coreaudio->format_ID;
39   streamFormat.mFormatFlags = p_params4coreaudio->format_flags;
40   streamFormat.mBytesPerPacket = p_params4coreaudio->bytes_in_a_packet;
41   streamFormat.mFramesPerPacket = p_params4coreaudio->frames_per_packet;
42   streamFormat.mBytesPerFrame = p_params4coreaudio->bytes_per_frame;
43   streamFormat.mChannelsPerFrame = p_params4coreaudio->num_channels;
44   streamFormat.mBitsPerChannel = p_params4coreaudio->bits_per_channel;
45
46 [...]
47
48   err = AudioUnitSetProperty (p_params4coreaudio->output_unit,
49                               kAudioUnitProperty_StreamFormat,
50                               kAudioUnitScope_Input,
51                               0,
52                               &streamFormat,
```

```
53                                  sizeof(AudioStreamBasicDescription));
54
55 [...]
56
57   err = AudioUnitInitialize(p_params4coreaudio->output_unit);
58
59 [...]
60
61   err = AudioOutputUnitStart (p_params4coreaudio->output_unit);
62
63 [...]
64
65   AudioDeviceID CurrentDevice;
66   size = sizeof(Float64);
67   err = AudioUnitGetProperty (p_params4coreaudio->output_unit,
68                               kAudioOutputUnitProperty_CurrentDevice,
69                               0,
70                               0,
71                               &CurrentDevice,
72                               &size);
73
74 [...]
75
76   err = AudioDeviceSetProperty (CurrentDevice,
77                                 NULL,
78                                 0,
79                                 false,
80                                 kAudioDevicePropertyBufferFrameSize,
81                                 sizeof(p_params4coreaudio->audio_buffer_size),
82                                 &p_params4coreaudio->audio_buffer_size);
83 [...]
84 }
```

This code snippet is quite long (though shortened), but easy to understand. `create_audio_unit()` function in line 14 takes two parameters: a reference to a `t_params4coreaudio` struct and a reference to an address in memory named "`p_params4audio_callback`". As described above the first parameter describes the properties of the audio unit that will be created. The second argument is a pointer of the `void` type. It is the programmer's responsibility to cast this address to whatever data type it refers to. As described above, in this application it is a pointer to an `t_params4audio_callback` struct.

In line 19 `create_audio_unit()` opens an audio unit — the one that is stored in `t_params4coreaudio`. Two important settings are made in line 24 and 25: the callback function is specified (`core_audio_process_function()`) and the pointer that is passed to that callback function every time that it is called (`p_params4audio_callback`). Both settings are applied in line 27.

The audio processing parameters stored in `t_params4audio_callback` are passed on to the audio unit in lines 36 - 48. The audio unit is initialised with these settings in line 57, and, finally, started in line 61.

The size of the audio buffer is not a property of the audio unit, but a property of the audio rendering device. With respect to the *Orbits* application it is not interesting to determine which device would that be[6]. For example, there could be several audio processing units (sound cards) available. Instead, we assume that there is at least one and we ask the system for a reference to the current device in line 67. Now that we have a reference to the audio rendering device (which will be the built-in sound card of the *MacBook* in the standard case), we tell that device to render audio with a specified buffer size in line 76 (a value which is stored in the `t_params4audio_callback` struct).

3.2.5 The audio callback function(s)

This topic may be a little bit confusing. Infact, there are several functions involved in the audio callback process. Nevertheless, it is important to understand that there is only *one* audio unit, only *one* audio process and only *one* audio callback function call (which is then passed to subsequent functions.)

Technically, there is only one callback function: the `core_audio_process_function()`, defined above in line 3. This is the function that the audio unit is calling on. As one can easily see in line 9, the only thing this function does is calling another function: `callback4coreaudio()`, defined in `orbits.cpp`:

```
1 void callback4coreaudio (void* p_params4audio_callback_val, UInt32 inFrames,
2                          void* left_channel, void* right_channel)
3 {
4   audio_callback_function(p_params4audio_callback_val, 0, inFrames,
5                           left_channel, right_channel);
6 }
```

This function is very short too. Its name indicates that it is the callback function for *Core Audio*. The reason is that *Orbits* supports different sound processing environments. One other environment, though not relevant for this thesis, is the *JACK Audio Connection KIT (JACK)* [42]. *JACK* is a cross-plattform audio processing environment (which enables *Orbits* to be compiled for other operating systems[7]). Therefore, another call-

[6] More precisely we do not care *how many* and *which* devices are out there.

[7] Currently, *JACK* is availabe for *Linux*, *Mac OS X*, *Windows* and *Solaris/OpenSolaris*. However, one should keep in mind that the current version of *Orbits* relies on the built-in

back function is implemented in `orbits.cpp`: `callback4jack()` (not shown here). Both[8], `callback4coreaudio()` respectively `callback4jack()`, call on `audio_callback_function()` — *the* audio callback function:

```
1  void audio_callback_function(t_params4audio_callback* p_params4audio_callback,
2                               int frame_start, int frame_end,
3                               void* left_channel, void* right_channel)
4  {
5    read_ms(&p_global->ms_mac, &p_global->motion_values);
6    (*p_params4audio_callback->pp_scene)->set_pitch_yaw_roll(
7                                         p_params4audio_callback->p_motion_values->pitch,
8                                         p_params4audio_callback->p_motion_values->yaw,
9                                         p_params4audio_callback->p_motion_values->roll);
10   static double left_float;
11   static double right_float;
12   for (int frame = frame_start; frame < frame_end; frame++)
13     {
14       (*p_params4audio_callback->pp_scene)->update();
15
16       if (p_params4audio_callback->p_audio_param->stereo_flag)
17         {
18           (*p_params4audio_callback->pp_scene)->render_audio(
19                                         p_params4audio_callback->p_audio_param,
20                                         &left_float, &right_float);
21         }
22       else
23         {
24           left_float = (*p_params4audio_callback->pp_scene)->render_audio(
25                                         p_params4audio_callback->p_audio_param);
26           right_float = left_float;
27         }
28 [...]
29
30       p_global->notification_sound.update();
31       static double noti_float;
32       noti_float = p_global->notification_sound.get_vel()*1000;
33
34       if (!p_params4audio_callback->p_audio_param->mute_flag)
35         {
36           left_float += noti_float;
37           right_float += noti_float;
38         }
39       else
40         {
41           left_float = 0.;
42           right_float = 0.;
43         }
44
45      if (p_params4audio_callback->p_game_state->update())
46         {
47           p_global->notification_sound.apply_dirac_represented_continuous_force(1.);
48 [...]
49         }
50 [...]
51
52       static_cast<Float32*>(left_channel)[frame]  = left_float;
53       static_cast<Float32*>(right_channel)[frame] = right_float;
54     }
55 }
```

accelerometer. Infact, a binary for *LINUX* can be compiled, but without accelerometer the whole concept does not make sense (though mouse steering has been tested under *LINUX*).

[8]Note: of course *Orbits* would be compiled either with *Core Audio* or *JACK* support!

The `audio_callback_function()` can be regarded as the actual audio callback function, since the other functions introduced above directly route here. The first parameter is a reference to the `t_params4audio_callback` struct presented above. The other parameters specify the size of the audio buffer and references to memory addresses for the audio output[9].

In the audio callback function one has to differentiate between commands that are called once (buffer-wise) and operations that are executed at every audio sample (with audio rate). As one can see, there is only one operation that is executed once: `read_ms()` (line 5). `read_ms()` reads data from the motion sensor, which is forwarded to the current scene in line 6.

The next block of code is a `for`-loop from line 12 to 54. These are the operations that are carried out with audio rate. The loop begins with an update of the scene in line 14. This computes the macroscopic physical behaviour of the moving objects (balls, cylinders) with respect to the current accelerometer data. The behaviour of the current sound models (encapsulated in the scene and its objects) is updated by the the function `render_audio()` (q.v. page 67) in line 18, respectively line 24 (this depends on the stereo setting).

In line 30 a notification sound is updated. Whenever the player successfully fulfills a task, the notification sounds. The notification itself is based on the sound rendering objects used for the rolling sound, sliding sound, etc. and therefore it needs to be updated with audio rate as well.

The game state is updated in line 45. This is explained at page 74. Basically, it is checked if the balls are in their targets, and if so, what is the subsequent target position or scene.

If one tracks what happens with the two variables `left_float` and `right_float` in lines 20 (resp. 24 and 26) and 36 & 37, one can see that they contain the audio output of one audio frame — the left and the right audio channel. These two values are written into the audio buffers at the end of every run of the loop — until the buffers are filled.

[9]The rendering can be done in mono or stereo, but in either case two channels are used.

3.3 Video processing

As described at page 39, the *OpenGL* graphics rendering and keystroke inputs are handled by *GLUT*. Before one starts rendering polygons, lines or textures, one has to setup a window for the rendering, register some callbacks and start the event handling process.

If one compares the *Orbits* code with a standard tutorial about graphics rendering with *GLUT*, one will notice two important differences. The thing can be found in line 5 of the code below. The documentation on *GLUT* [43] states that a the function `glutInit()` "initialize(s) the *GLUT* library and negotiate(s) a session with the window system.". However, this function cannot be found in the *Orbits* code. The reason is that *Orbits* uses another library — *Glow* — to handle a graphical menu (with button and sliders, etc.). *Glow* needs control about the window initialisation and the event processing, therefore `Glow::Init()` is called in line 5, which then initialises windows.

The event processing is the second difference between *Orbits* and other *Glut*-based applications. When all necessary objects are initialised and all audio and video setup has been made, it is the regular moment to start the *glutMainLoop()*. As *Glow* needs control about the event loop, the `Glow::MainLoop()` (q.v. page 51) is called instead.

Most of the setup for *GLUT* is done in the `initGL()` function in `orbits.cpp`:

```
1  void initGL(void)
2  {
3    int n_glutargs = 0;
4
5    Glow::Init(n_glutargs, NULL);
6    glutInitDisplayMode(GLUT_DEPTH|GLUT_DOUBLE|GLUT_RGBA|GLUT_MULTISAMPLE);
7    glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
8    glutInitWindowPosition(WINDOW_POS_X, WINDOW_POS_Y);
9    p_global->main_window_ID = glutCreateWindow(NULL);
10
11   glutDisplayFunc(draw_orbits);
12   glutReshapeFunc(reshape_orbits);
13   glutTimerFunc(100, redisplay_orbits, 0);
14   glutKeyboardFunc(keyboard_orbits);
15
16 [...]
17
18   glViewport(0,0,p_global->video_param.window_width,p_global->video_param.window_height);
19
20   glMatrixMode(GL_PROJECTION);          // Select The Projection Matrix
21   glLoadIdentity();                     // Reset The Projection Matrix
22
23   gluPerspective(45.0f,(GLfloat)p_global->video_param.window_width/
24                  (GLfloat)p_global->video_param.window_height,0.1f,100.0f);
25
```

```
26    glMatrixMode(GL_MODELVIEW);            // Select The Modelview Matrix
27    glLoadIdentity();                      // Reset The Modelview Matrix
28
29    /* Set up Light */
30    glEnable(GL_LIGHT0);
31    GLfloat light_source[4] = { 2.0, -1.0, 2.0, 1.0 };
32    GLfloat light_ambient[4] = { 0.5, 0.5, 0.5, 1.0 };
33    p_global->video_param.light0_x = light_source[0];
34    p_global->video_param.light0_y = light_source[1];
35    p_global->video_param.light0_z = light_source[2];
36
37    glLightfv(GL_LIGHT0, GL_POSITION, &light_source[0]);
38    glLightfv(GL_LIGHT0, GL_AMBIENT, &light_ambient[0]);
39
40    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION,  1.0f);
41    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.2f);
42    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.08f);
43 }
```

As pointed out, the function starts the window initialisation via `Glow::Init()` in line 5 (and sets some window properties in lines 6 - 8). Afterwards, the function registers several callback functions. In line 11 it registers `draw_orbits()` as `glutDisplayFunc()`, which means that `draw_ballacer()` will be called whenever `GLUT` decides that the graphics output needs to be rendered. Furthermore, a `glutReshapeFunc()`, a "redisplay"-function (that is called by a time: `glutTimerFunc()`) and a `glutKeyboardFunc()` is specifed. Their purpose is explained in the next section.

As mentioned previously, one needs to specify *how* graphics content is rendered into the graphics window (q.v. page 38). This is done in line 18 and 23 by setting a `glViewport()` and a `gluPerspective()`.

It is important to consider that `initGL()` is only called once when the application starts. All setup here is only done once. That also applies to the last setup in `initGL()`: the source of light (lines 30 - 42). In connection with the remarks about *OpenGL* as a "state machine" it has been noted briefly that the light setup is one of the states that remains the same through the entire life-time of the application.

3.3.1 Video callback functions

All video callback functions are declared in `orbits.hpp` and defined in `orbits.cpp`. At first I will introduce the most obvious one — the rendering callback function `draw_orbits()`:

```
1  void draw_orbits(void)
2  {
3  [...]
4    p_global->p_scene->render_video(&p_global->video_param);
5  [...]
6
7    p_global->_2D_current_time.render(&p_global->video_param);
8    p_global->_2D_average_time.render(&p_global->video_param);
9
10   p_global->glut_terminal.render(&p_global->video_param);
11   if (p_global->video_param.display_highscores_flag)
12     {
13       build_highscore_table();
14       p_global->glut_highscore_table.render(&p_global->video_param);
15     }
16   if (p_global->video_param.display_info_flag)
17     {
18       p_global->glut_info.render(&p_global->video_param);
19     }
20   p_global->status_bar.render(&p_global->video_param);
21
22   glutSwapBuffers();
23 }
```

This function calls the `render_video()` function of the scene in line 4 and passes as a reference to a `t_video_param` struct as an argument to that function (q.v. page 66). In lines 7 - 20 the function updates several elements of the graphical user interface (number, text massages, ...) and finallt it calls `glutSwapBuffers()`. *Orbits* uses two buffers to render graphical content — while one video buffer is filled with content, the other one is displayed, and vice versa. `glutSwapBuffers()` performs the necessary buffer swap.

Another callback function is `reshape_orbits()`, is it called whenever the size of the current graphics window changes:

```
1  void reshape_orbits(int width, int height)
2  {
3  [...]
4
5    glViewport(0,0,p_global->video_param.window_width,p_global->video_param.window_height);
6
7    glMatrixMode(GL_PROJECTION);
8    glLoadIdentity();
9
10   gluPerspective(45.0f,(GLfloat)p_global->video_param.window_width/
11                        (GLfloat)p_global->video_param.window_height,0.1f,100.0f);
12
13   glMatrixMode(GL_MODELVIEW);
14   glLoadIdentity();
15 }
```

Besides adjusting the positions of some GUI elements (not shown in the code above) `reshape_orbits()` primarily resets the `glViewport()` (line 5) and the `gluPerspective()` (line 10), depending on the new window size in pixels (`width`, `height`).

In the beginning of this section, a function was mentioned that was called by a `glutTimerFunc()`. `glutTimerFunc()` is a callback that is called only once, when the specified time has elapsed. Since a `glutTimerFunc()` can call itself when it is finished, it can be used a callback that is called on a regular basis. `redisplay_orbits()` is doing exactly that:

```
1 void redisplay_orbits(int value)
2 {
3   glutSetWindow(p_global->main_window_ID);
4   glutPostRedisplay();
5   glutTimerFunc(1000/VIDEO_FRAMES_PER_SECOND, redisplay_orbits, 0);
6 }
```

`redisplay_orbits()` calls the function `glutPostRedisplay()` (line 4) and then calls itself with a specified time in *ms* (line 5). `glutPostRedisplay()` performs the actual *displaying* on the screen. `draw_orbits()` has filled the video rendering buffer with data and performed a buffer swap, but that buffer is rendered on the screen only now — with a fixed video rate (of $50Hz$, q.v. page 42).

The fourth *GLUT* callback function `keyboard_orbits` is rather long (and in general not very interesting). I will only present a schematic draft of that function here:

```
1 void keyboard_orbits(unsigned char key, int x, int y)
2 {
3   switch (key)
4     {
5     case 43:
6       /* Key "+" */
7       p_global->audio_param.volume *= 1.25;
8       break;
9
10    case 45:
11      /* Key "-" */
12      p_global->audio_param.volume *= 0.8;
13      break;
14
15    case 113:
16      /* Key "q" */
17      close_application();
18      break;
19    };
20 }
```

GLUT calls that function whenever a key has been pressed. The function performs tasks depending on the key stroke. In the example above it raises/loweres the audio volume if +/- is pressed and closes the application at *q*.

3.3.2 The `t_video_param` struct

Analogue to audio rendering, parameters that specify *how* a scene is rendered are stored in a struct: the `t_video_param` struct (q.v. page 56 and page 63).

`t_video_param` is defined in `param_4_video_rendering.hpp`:

```
1  struct t_video_param
2  {
3    int window_width;
4    int window_height;
5
6    int fullscreen_flag;
7    int default_window_width;
8    int default_window_height;
9
10   int display_highscores_flag;
11   int display_info_flag;
12
13   int fixed_view_flag;
14   double angle_x;
15   double angle_y;
16   double angle_z;
17   double zoom;
18
19   double* p_pitch;
20   double* p_yaw;
21   double* p_roll;
22
23   double light0_x;
24   double light0_y;
25   double light0_z;
26   double reflection_transparency;
27
28   double scene_infinity;
29   int circle_vertices;
30   int ball_vertices;
31
32 [...]
33 };
```

As one can see `t_video_param` contains information about the (default) size of the window (line 3 - 8), flags that indicate if highscores of information has to be displayed (line 10, 11), the perspective and the distance of the viewer (lines 13 - 17), the current inclination of the scene (lines 19 - 21) and the position of the light (lines 23 - 25). And finally it also contains parameters that specify how detailed things are rendered in lines 28 - 30 (e.g. how many vertices has a circle).

In the `draw_orbits()` callback function the `t_video_param` struct is passed to the current scene as an argument every time that scene is rendered (q.v. page 63). This is also analogue to the audio rendering (q.v. page 56).

3.4 Important classes & structs

The *Orbits* application has been created to provide a platform to setup a custom test, based on the implemented sound models and graphical components. Of course it it possible to implement yet another sound model or scene element and use it in *Orbits*, but it would go too much into detail to describe how to do that. In this regard, I recommend reading the project's documentation. There you will find information about the `c_sound_model` class, which is used as an interface to implement custom sound algorithms.

In this section I want to give an overview about some essential modules of the project and explain how they work together. If one wants to create its own testing setup, one has to write a function similar to `init_scenes()` in `orbits.cpp`. This section should be useful to understand what happens in `init_scenes()`.

3.4.1 The `c_scene` class

The first class I want to introduce is *c_scene* (in the following called *scene*). A scene comprises everything one can see at the screen or hear from the speakers. Everything that is needed to create a scene in *Orbits* is strictly unitised. Creating a custom scene, does *not* mean that a new `c_scene` class has to be written or deviated through inheritance! Instead, single modules are created and assigned to the scene, which will be shown at the following pages. An excerpt of `c_scene.hpp` (the class definition):

```
1  class c_scene
2  {
3  protected:
4
5    int num_elements;
6    c_scene_element** a_scene_elements;
7
8  [...]
9
10 public:
11
12   c_scene();
13
14   void init(int num_elements_val, c_scene_element** a_scene_elements_val,
15             int* a_velocity_goals_val, double underground_color_val[3]);
16
17   ~c_scene();
18
19   int underground_on;
```

```
20    double underground_color[3];
21
22    virtual void render_audio(t_audio_param* p_audio_param, double* p_left, double* p_right);
23    virtual void render_video(t_video_param* p_video_param);
24
25    void update();
26    void set_pitch_yaw_roll(double pitch_val, double yaw_val, double roll_val);
27
28    void set_element(int n_element, c_scene_element* a_scene_element_new);
29    void set_goal_pos_width_deg(int n_element, double goal_pos_deg_val,
30                                double goal_width_deg_val);
31
32    int all_inside();
33
34  [...]
35  };
```

In terms of member variables, a scene consists of a number of `c_scene_elements` (line 5 & 6, q.v. page 57) and as you can see in line 14 it is initialised with references to those elements. The scene also contains the underground (or background) colour in line 20, since single elements do not have a background.

The scene has two render functions: `render_audio()` and `render_video()` (lines 22 & 23). As mentioned above, these functions have references to a `t_audio_param`, respectively a `t_video_param` struct, which specify how a single audio or video frame should be rendered.

`set_pitch_yaw_roll()` (line 26) applies motion sensor data to the scene and `update()` (line 25) updates the macroscopic behaviour of all objects (scene elements).

Goal positions and widths can set for single scene elements with `set_goal_pos_width_deg()` (line 29) and `all_inside()` (line 29) checks if all objects (in all scene elements) are inside the correspondig target areas.

As indicated (*[...]*, there are a lot more functions, but these are the most important.

3.4.2 The `c_scene_element` class

Scene elements are the building blocks a scene is made of. Analogue to the scene, there is only one `c_scene_element` class, which can be fully customised with various attributes. `c_scene_element.hpp`:

```
1  class c_scene_element
2  {
3
```

```
4  private:
5
6    c_sound_model* p_sound_model;
7    c_mass_on_curve* p_mass_on_curve;
8    c_graphics_module* p_graphics_module;
9
10   double const_grav_acc;
11   double pitch;
12   double yaw;
13   double roll;
14
15 [...]
16
17 public:
18
19   c_scene_element();
20
21   void init(t_sonic_behaviour sonic_behaviour_val,
22             t_movement_behaviour movement_behaviour_val,
23             t_graphical_representation graphical_representation_val,
24             double const_grav_acc_val);
25
26   ~c_scene_element();
27
28   void render_audio(t_audio_param* p_audio_param, double* p_left, double* p_right);
29   void render_video(t_video_param* p_video_param);
30
31   void update();
32   void set_pitch_yaw_roll(double pitch_val, double yaw_val, double roll_val);
33
34   void set_goal_pos_width_deg(double goal_pos_deg_val, double goal_width_deg_val);
35
36   int is_in_goal();
37
38 [...]
39 };
```

A scene element contains three important objects: a `c_sound_model`, a `c_mass_on_curve` and a `c_graphics_module` (lines 6 – 8). The sound models have been described in the previous chapter (q.v. page 43). Likewise, the graphics modules have been mentioned (ball, cylinder). The `c_mass_on_curve` class describes the physical behaviour (with regard to gravity) of a point-mass that moves along a curve. For now, it is not necessary to go too much into detail with these objects. If one wants to develop a different sound model or graphical object than the ones included in this work, then detailed knowledge about these classes is important, because on has to write a class that inherits from those.

For understanding how to create a custom scene with the currently existing instruments, three structs are much more interesting: `t_sonic_behaviour`, `t_movement_behaviour` and `t_graphical_representation` (compare line 21). As their names suggest, these structs contain information about the sonic and movement behaviour of the scene element, as well as the graphical represent-

ation. They are used to create a scene element with the required properties and will be explained in more detail below.

Analogue to the scene, the scene element has functions to update its state, to render audio and video output and to setup a goal.

The `t_sonic_behaviour` struct

The sonic behaviour struct contains various parameters to describe sound models as can be seen in the code below (lines 3 – 30):

```
1  struct t_sonic_behaviour
2  {
3    enum_sound_model sound_model_type;
4
5    /* rolling sound parameters */
6    double ball_radius;
7    int num_points; double length; double aspect_ratio;
8    int num_modes; double* p_freqs;
9    double* p_t_es; double* p_weights;
10   double mass; double mass_pos; double mass_vel;
11   double switch_point; double* a_k_contact; double* a_c_contact;
12
13   /* rolling sound saw profile parameters */
14   double saw_width; double saw_height;
15
16   /* rolling sound pattern profile parameters */
17   double pattern1_width; double pattern2_width; double pattern1_aspect_ratio;
18   double pattern2_aspect_ratio;
19
20   /* sliding sound parameters */
21   double band_width;
22
23   /* BLIT sound parameters */
24   double a; int N; double volume_val;
25
26   /* click sound parameters */
27   double step_width;
28
29   /* general parameters (used by some or all sound models) */
30   double timestep; double volume; double pitch_factor;
31
32   void init_rolling_sound(double ball_radius_val,
33                           int num_points_val, double length_val, double aspect_ratio_val,
34                           int num_modes_val, double* p_freqs_val,
35                           double* p_t_es_val, double* p_weights_val,
36                           double mass_val, double mass_pos_val, double mass_vel_val,
37                           double switch_point_val, double* a_k_contact_val,
38                           double* a_c_contact_val,
39                           double timestep_val, double volume_val);
40
41   void init_rolling_sound_saw(double ball_radius_val,
42                               int num_points_val, double length_val, double saw_width_val,
43                               double saw_height_val, double aspect_ratio_val,
44                               int num_modes_val, double* p_freqs_val,
45                               double* p_t_es_val, double* p_weights_val,
46                               double mass_val, double mass_pos_val, double mass_vel_val,
47                               double switch_point_val, double* a_k_contact_val,
48                               double* a_c_contact_val,
49                               double timestep_val, double volume_val);
50
```

```
51   void init_rolling_sound_pattern(double ball_radius_val,
52                                   int num_points_val, double length_val,
53                                   double pattern1_width_val, double pattern2_width_val,
54                                   double pattern1_aspect_ratio_val,
55                                   double pattern2_aspect_ratio_val,
56                                   int num_modes_val, double* p_freqs_val,
57                                   double* p_t_es_val, double* p_weights_val,
58                                   double mass_val, double mass_pos_val, double mass_vel_val,
59                                   double switch_point_val, double* a_k_contact_val,
60                                   double* a_c_contact_val,
61                                   double timestep_val, double volume_val);
62
63   void init_sliding_sound(double pitch_factor_val, double band_width_val,
64                           int num_modes_val, double* p_freqs_val, double* p_t_es_val,
65                           double* p_weights_val,
66                           double timestep_val, double volume_val);
67
68   void init_BLIT(double a_val, int N_val, double pitch_factor_val, double volume_val);
69
70   void init_inverted_sound(double pitch_factor_val, double volume_val);
71
72   void init_click_sound(double length_val, double step_width_val,
73                         int num_modes_val, double* p_freqs_val, double* p_t_es_val,
74                         double* p_weights_val,
75                         double timestep_val, double volume_val);
76 };
```

One has to create a `t_sonic_behaviour` struct in order to initialise a scene element, but it is not intended to specify all necessary field in the struct manually (it is quite probable that something is missed and the application crashes). On the contrary, there are designated functions to arrange the struct. Every sound model is represented by one *init*-function with a set of obligatory parameters (which prevents from missing one).

The `t_movement_behaviour` struct

The idea behind these structs is more or less the same: providing a convenient way to specify a custom scene element without the fussiness to implement a separate scene element class for every possible setup. Hence, `t_movement_behaviour` resembles `t_sonic_behaviour` in structure and use. It contains parameters that describe a point mass and a movement curve (*Orbits* only contains circular curves, but could be extended with other shapes). Again, the init functions are the most important part. Currently, there are four functions to set up circular movement in both or just one direction. Optionally, a geometry factor can be specified, that influences the moment of inertia.

```
1 struct t_movement_behaviour
2 {
```

```
3   enum_movement_model movement_model_type;
4
5   double circle_radius;
6
7   /* general parameters (used by some or all movement models) */
8   double length;
9   double mass;
10  double geometry_fact;
11  double c_fric;
12  int direction;
13  double timestep;
14
15  void init_movement_on_circle(double circle_radius_val, double mass_val,
16                               double c_fric_val, double timestep_val);
17
18  void init_movement_on_circle(double circle_radius_val, double mass_val,
19                               double geometry_fact, double c_fric_val,
20                               double timestep_val);
21
22  void init_movement_on_circle_one_way(double circle_radius_val, double mass_val,
23                                       double c_fric_val, int direction_val,
24                                       double timestep_val);
25
26  void init_movement_on_circle_one_way(double circle_radius_val, double mass_val,
27                                       double geometry_fact, double c_fric_val,
28                                       int direction_val, double timestep_val);
29
30 };
```

The `t_graphical_representation` struct

The graphical representation comprises a ball and a cylinder. Various parameters regarding the colours, glossiness and size of the objects can be specified:

```
1  struct t_graphical_representation
2  {
3    enum_graphics_module graphics_module_type;
4
5    /* track parameters*/
6    double circle_radius;
7    double line_p1_x; double line_p1_y;
8    double line_p2_x; double line_p2_y;
9    double line_width;
10   double line_color[3];
11
12   /* goal position parameters */
13   double goal_pos_deg; double goal_width_deg;
14   double goal_border_close; double goal_border_far;
15   double goal_color[3];
16   double alternate_goal_color[3];
17
18   /* moving objects parameters */
19   double ball_radius;
20   double cyl_radius1; double cyl_radius2; double cyl_length;
21   double object_color[3];
22   GLfloat object_ambient[4];
23   GLfloat object_diffuse[4];
24   GLfloat object_specular[4];
25   GLfloat object_shininess;
26
27   void init_ball_on_circle(double line_color_val[3], double goal_color_val[3],
```

```
28                              double alternate_goal_color_val[3], double object_color_val[3],
29                              GLfloat object_ambient_val[4], GLfloat object_diffuse_val[4],
30                              GLfloat object_specular_val[4], GLfloat object_shininess_val,
31                              double circle_radius_val, double line_width_val,
32                              double goal_pos_deg_val, double goal_width_deg_val,
33                              double ball_radius_val);
34
35    void init_cyl_on_ring(double line_color_val[3], double goal_color_val[3],
36                          double alternate_goal_color_val[3], double object_color_val[3],
37                          GLfloat object_ambient_val[4], GLfloat object_diffuse_val[4],
38                          GLfloat object_specular_val[4], GLfloat object_shininess_val,
39                          double circle_radius_val, double line_width_val,
40                          double goal_pos_deg_val, double goal_width_deg_val,
41                          double cyl_radius1_val, double cyl_radius2_val,
42                          double cyl_length_val);
43
44  [...]
45  };
```

3.4.3 Creating a custom game

Finally, I will explain briefly how to create a custom test setup, including the required number of scenes and scene elements with sound models, graphical representations and movement behaviour. This setup is done in the `init_scenes()` function in the main files of the application (`orbits.hpp`, `orbits.cpp`). Before the source code of that function will be discussed, one important part of the program needs to be introduced: the game logic.

Game logic comprises setting as the number of scene, the chronological order of scenes, the target sizes, positions, number of targets, etc. — in short, the schedule of the game (respectively user test). This schedule is administered in a struct, the `t_game_state` struct.

Single "game levels" are called *steps*, they specify game settings (like target sizes and positions) for one scene. In this regard, it is important to stress the difference between scenes and steps. A scene comprises the visual and auditive representation, whereas a step defines the tasks to fulfill in this scene. For instance, a scene can occur several times in game schedule, each time with different degree of difficulty, number or size of targets, etc. Steps are represented by a `t_step` struct.

The `t_game_state` struct

This struct checks if the task is fulfilled, it changes target positions and proceeds to the next step when appropriate. Consequentially, it also measures the task times (which will be saved in a file).

The game state struct holds references to the current active scene (line 3), the complete array of all scenes (line 4) and a savegame (line 5):

```
1  struct t_game_state
2  {
3    c_scene** pp_active_scene;
4    c_scene** pa_scenes;
5    c_savegame* p_savegame;
6
7    int num_steps;
8    int n_current_step;
9    t_step* a_steps;
10
11   double time_game_start;
12   double time_current_step_start;
13   double time_current_target_start;
14
15   double time_game;
16   double time_current_step;
17   double time_current_target;
18
19   list<double> l_times;
20
21   double time_last_step;
22   double time_last_target;
23
24   void init(c_scene** pp_active_scene_val, c_scene** pa_scenes_val,
25             int num_steps_val, t_step* a_steps_val,
26             c_savegame* p_savegame_val);
27
28   int update();
29
30 [...]
31 };
```

Besides several variables to measure task times (lines 11 - 22), the struct also contains an array of `t_steps` in line 9 (please compare: `init_function()` in line 24).

The `update()` function in line 28 updates the game state and is called from within the audio callback function (q.v. page 59).

The `t_step` struct

A step can be regarded as a set of settings that specify the conditions of the task to fulfill by the player/subject. As you can see below, it contains a reference to the scene, that it used in the step (line 3), the number of scene

elements (in *Orbits* currently always two elements are used) and parameters that specify the targets (lines 6 - 9):

```
1  struct t_step
2  {
3    c_scene* p_scene;
4    int num_elements;
5
6    int num_targets;
7    int n_current_target;
8    enum_target_width_mode* a_target_width_mode;
9    double* a_target_width;
10
11   void init(c_scene* p_scene_val, int num_targets_val,
12             enum_target_width_mode* p_target_width_mode_val,
13             double* p_target_width_val);
14 };
```

The number of targets in the step (`num_targets`, the target widths (`a_target_width` array) and a target mode can be specified. The target mode can be chosen amongst these values: `FIXED_TARGET_SIZE`, `INCREASING_TARGET_SIZE`, `DECREASING_TARGET_SIZE` and `RANDOM_TARGET_SIZE` (please compare `enum_target_width_mode.hpp` whicg is part of the source code of this thesis).

The `init_scenes()` function

Finally, this is the most essentail step: initialising all the structs and objects and configure a custom game/user test. The `init_scenes()` function in `orbits.cpp` does all that.

At the following pages I will explain how to create a game, including the following conditions: *one* set of rolling sound parameters, *two* ball sizes, *two* circle sizes (an inner circle and an outer circle) and *one* set of parameters that specify movement behaviour.

Under these conditions, the following structs need to be initialised:

1. *four* sound model structs (*1 rolling sound* x *2 ball sizes* x *2 circle sizes*)
2. *four* graphical representation structs (*2 ball sizes* x *2 circle sizes*)
3. *two* movement behaviour structs (*1 movement behaviour* x *2 circle sizes*)

These structs are used to create *two* scenes, which both consist of *two* scene elements (a small and a big circle). The first scene is built using the

smaller balls, the second scene uses bigger balls. Finally a game schedule is created (`t_game_state`) with *four* steps. Every scene appears *two* times (which sums to four steps), but the target size mode differs.

I chose this simple setup to demonstrate the use of all structs and objects introduced above, and I hope it is a useful guide to create own games/tests.

The following code samples are excerpts from the *Orbits'* `init_scenes()` function[10]:

```
1  void init_scenes(double timestep)
2  {
3    /**********************
4     * define colours
5     **********************/
6
7  [...]
8
9    if (p_global->init_scenes_from_record_flag)
10     {
11       /**********************
12        * open savegame
13        **********************/
14
15 [...]
16
17     }
18   else
19     {
20       double inner_circle_radius = 0.1;
21       double outer_circle_radius = 0.2;
22       double ball_radius = 0.015;
23       double ball_mass = 0.02;
24
25       double inner_size_fact = 1.;
26       double outer_size_fact = 1.;
27
28       double const_grav_acc = -9.81;
```

Here, at first some colours are defined and it is checked whether we load scenes from a recorded savegame. In case we do not, the first parameters are specified — the sizes of circles and balls (the sizes of balls will be changed in subsequent steps). Sound parameters for the rolling sounds are specified in the following lines:

```
1        /**********************
2         * sound parameter
3         **********************/
4
5        const int num_modes = 2;
6        double freq0s[num_modes] = { 300., 5000. };
7        double t_es[num_modes] = { .0015, .003 };
8        double weights[num_modes] = { 15., 8. };
9
10 [...]
```

[10]The source code attached to this thesis might look slightly different in some aspects, as this is only meant to be an example.

```
11
12      double switch_point = .00001; //INFINITY;
13      double a_k_contact[2] = {1., 2.};
14      double a_c_contact[2] = {1., 1.};
15
16 [...]
17      /* rolling filter profile */
18      double dpp = 0.0005; // distance between profile points
19      double aspect_ratio = 0.08;
20
21      double inner_length = 2*M_PI*inner_circle_radius;
22      int inner_num_points = (int)(inner_length/dpp)-1;
23      double outer_length = 2*M_PI*outer_circle_radius;
24      int outer_num_points = (int)(outer_length/dpp)-1;
25
26 [...]
```

All rolling sound models that are created below are based on these parameters. The same applies to movement and graphics parameters:

```
1       /**********************
2        * movement parameter
3        **********************/
4       double geometry_fact_solid_ball = 5./7.;
5       double rolling_c_fric = -0.015;
6
7       /**********************
8        * graphics parameter
9        *
10       **********************/
11      double line_width = 0.005;
12      double goal_pos_deg = 0;
13      double goal_width_deg = 90;
14
15 [...]
```

Now that all properties are set up, it is time to initialise the structs (`t_sonic_behaviour`, `t_graphical_representation`, `t_movement_behaviour`). At first, the inner circle structs are initialised (note the `inner_size_factor` in line 6 & 24):

```
1       /**********************
2        * init structs
3        *
4        **********************/
5
6       inner_size_fact = 1.;
7       t_sonic_behaviour inner_rolling_size1;
8       inner_rolling_size1.init_rolling_sound(ball_radius * inner_size_fact,
9                                              inner_num_points, inner_length, aspect_ratio,
10                                             num_modes, &freq0s[0], &t_es[0], &weights[0],
11                                             ball_mass * pow(inner_size_fact, 3.),
12                                             inner_size_fact * ball_radius + .005, 0,
13                                             switch_point, &a_k_contact[0],
14                                             &a_c_contact[0], timestep, 1.);
15 [...]
16
17      t_graphical_representation inner_graph_size1;
18      inner_graph_size1.init_ball_on_circle(grey08, telekom, magenta, telekom,
19                                            telekom_ambient, telekom_diffuse,
20                                            telekom_specular, shininess,
21                                            inner_circle_radius, line_width, goal_pos_deg,
```

```
22                                              goal_width_deg, ball_radius * inner_size_fact);
23
24      inner_size_fact = 2.0;
25      t_sonic_behaviour inner_rolling_size2;
26      inner_rolling_size2.init_rolling_sound(ball_radius * inner_size_fact,
27                                             inner_num_points, inner_length, aspect_ratio,
28                                             num_modes, &freq0s[0], &t_es[0], &weights[0],
29                                             ball_mass * pow(inner_size_fact, 3.),
30                                             inner_size_fact * ball_radius + .005, 0,
31                                             switch_point, &a_k_contact[0],
32                                             &a_c_contact[0], timestep, 1.);
33 [...]
34      t_graphical_representation inner_graph_size2;
35      inner_graph_size2.init_ball_on_circle(grey08, telekom, magenta, telekom,
36                                            telekom_ambient, telekom_diffuse,
37                                            telekom_specular, shininess,
38                                            inner_circle_radius, line_width, goal_pos_deg,
39                                            goal_width_deg, ball_radius * inner_size_fact);
```

The first two sound models and graphical representations are initialised. The same procedure follows now for the outer circle radius, again two sound models and graphical representations are created (one for each ball size — line 1 & 20):

```
1       outer_size_fact = 1.0;
2       t_sonic_behaviour outer_rolling_size1;
3       outer_rolling_size1.init_rolling_sound(ball_radius * outer_size_fact,
4                                              outer_num_points, outer_length,
5                                              aspect_ratio, num_modes, &freq0s[0],
6                                              &t_es[0], &weights[0],
7                                              ball_mass * pow(outer_size_fact, 3.),
8                                              outer_size_fact * ball_radius + .005, 0,
9                                              switch_point, &a_k_contact[0],
10                                             &a_c_contact[0], timestep, 1.);
11 [...]
12      t_graphical_representation outer_graph_size1;
13      outer_graph_size1.init_ball_on_circle(grey08, telekom, magenta, telekom,
14                                            telekom_ambient, telekom_diffuse,
15                                            telekom_specular, shininess,
16                                            outer_circle_radius, line_width,
17                                            goal_pos_deg, goal_width_deg,
18                                            ball_radius * outer_size_fact);
19
20      outer_size_fact = 2.0;
21      t_sonic_behaviour outer_rolling_size2;
22      outer_rolling_size2.init_rolling_sound(ball_radius * outer_size_fact,
23                                             outer_num_points, outer_length,
24                                             aspect_ratio, num_modes, &freq0s[0],
25                                             &t_es[0], &weights[0],
26                                             ball_mass * pow(outer_size_fact, 3.),
27                                             outer_size_fact * ball_radius + .005, 0,
28                                             switch_point, &a_k_contact[0],
29                                             &a_c_contact[0], timestep, 1.);
30 [...]
31      t_graphical_representation outer_graph_size2;
32      outer_graph_size2.init_ball_on_circle(grey08, telekom, magenta, telekom,
33                                            telekom_ambient, telekom_diffuse,
34                                            telekom_specular, shininess,
35                                            outer_circle_radius, line_width,
36                                            goal_pos_deg, goal_width_deg,
37                                            ball_radius * outer_size_fact);
38
39 [...]
```

Two structs are still missing. Like mentioned above, two `t_movement_behaviour` structs are needed, one for each circle size:

```
1       t_movement_behaviour inner_rolling_mov;
2       inner_rolling_mov.init_movement_on_circle(inner_circle_radius,
3                                                 ball_mass * pow(inner_size_fact, 3.),
4                                                 geometry_fact_solid_ball,
5                                                 rolling_c_fric, timestep);
6
7       t_movement_behaviour outer_rolling_mov;
8       outer_rolling_mov.init_movement_on_circle(outer_circle_radius,
9                                                 ball_mass * pow(outer_size_fact, 3.),
10                                                geometry_fact_solid_ball,
11                                                rolling_c_fric, timestep);
12
13 [...]
```

When all structs are created, initialising scene element is an easy step. Four scene elements are created with the structs initialised above:

```
1       /**********************
2        * init scene elements
3        **********************/
4
5       p_global->el_in_0.init(inner_rolling_size1, inner_rolling_mov,
6                              inner_graph_size1, const_grav_acc);
7       p_global->el_out_0.init(outer_rolling_size1, outer_rolling_mov,
8                               outer_graph_size1, const_grav_acc);
9
10      p_global->el_in_1.init(inner_rolling_size2, inner_rolling_mov,
11                             inner_graph_size2, const_grav_acc);
12      p_global->el_out_1.init(outer_rolling_size2, outer_rolling_mov,
13                              outer_graph_size2, const_grav_acc);
14
15 [...]
```

The next step is to create the two scene. All scenes are stored in one array in *Orbits* (line 5, below), in this case, of course, the array has the length 2. To create a scene one has to pass a list of scene elements to the scene's `init()` function. This list is created in line 7 of the code below:

```
1       /**********************
2        * init scenes
3        **********************/
4
5       p_global->a_scenes = new c_scene[2];
6       c_scene_element* scene_elements_list[2];
7       int a_velocity_goals[2]  = {-1, -1};
8
9       scene_elements_list[1] = &p_global->el_in_0;
10      scene_elements_list[0] = &p_global->el_out_0;
11      p_global->a_scenes[0].init(2, scene_elements_list, &a_velocity_goals[0], white);
12
13      scene_elements_list[1] = &p_global->el_in_1;
14      scene_elements_list[0] = &p_global->el_out_1;
15      p_global->a_scenes[1].init(2, scene_elements_list, &a_velocity_goals[0], white);
16
17 [...]
```

The two scenes are initialised in lines 11 and 15 of the code above.

Now these scenes can be used to create a game schedule. Four steps are created In the following code, the first two steps have a fixed target size. The third and fourth step are set up with decreasing target size. As one can see in lines 15, 22, 29 and 36 the two scenes are used two times. The last step to initialise the game is to call the game state's `init()` function (line 35):

```
1       /**********************
2        * init game
3        **********************/
4
5       int num_steps = 4;
6       t_step* a_steps = new t_step[num_steps];
7
8       enum_target_width_mode a_target_width_mode[2];
9       double a_target_width[2];
10
11      a_target_width_mode[0] = FIXED_TARGET_SIZE;
12      a_target_width_mode[1] = FIXED_TARGET_SIZE;
13      a_target_width[0] = 30;
14      a_target_width[1] = 30;
15      a_steps[0].init(&p_global->a_scenes[0], 4, &a_target_width_mode[0],
16                      &a_target_width[0]);
17
18      a_target_width_mode[0] = FIXED_TARGET_SIZE;
19      a_target_width_mode[1] = FIXED_TARGET_SIZE;
20      a_target_width[0] = 30;
21      a_target_width[1] = 30;
22      a_steps[1].init(&p_global->a_scenes[1], 4, &a_target_width_mode[0],
23                      &a_target_width[0]);
24
25      a_target_width_mode[0] = DECREASING_TARGET_SIZE;
26      a_target_width_mode[1] = DECREASING_TARGET_SIZE;
27      a_target_width[0] = 30;
28      a_target_width[1] = 30;
29      a_steps[2].init(&p_global->a_scenes[0], 4, &a_target_width_mode[0],
30                      &a_target_width[0]);
31
32      a_target_width_mode[0] = DECREASING_TARGET_SIZE;
33      a_target_width_mode[1] = DECREASING_TARGET_SIZE;
34      a_target_width[0] = 30;
35      a_target_width[1] = 30;
36      a_steps[3].init(&p_global->a_scenes[1], 4, &a_target_width_mode[0],
37                      &a_target_width[0]);
38
39      p_global->game_state.init(&p_global->p_scene, &p_global->a_scenes, num_steps,
40                                a_steps, &p_global->savegame);
41
42 [...]
43    }
44 [...]
45 };
```

Chapter 4

Proposals and Improvements

Chapter 5

Résumé

Bibliography

[1] David Kaufmann. *Gedenkbuch zur Erinnerung an David Kaufmann.* hrsg. von M. Brann und F. Rosenthal, Breslau, 1900. This is a posthumous bibliography of David Kaufmanns work. This thesis refers to Kaufmanns work "Die Sinne. Beiträge zur Geschichte der Physiologie und Psychologie im Mittelalter. Aus Hebräischen und Arabischen Quellen", Budapest, 1884.

[2] Markus Dahm. *Grundlagen der Mensch-Computer-Interaktion.* Pearson Studium, Muenchen, 2006.

[3] Bruno L. Giordano. Everyday listening: an annotated bibliography. In Davide Rocchesso and Federico Fontana, editors, *The Sounding Object*, pages 1–14. Mondo Estremo, Firenze, Italy, 2003.

[4] S. J. Lederman. Auditory texture perception. *Perception*, 8:93–103, 1979.

[5] R. P. Wildes and W. A. Richards. Recovering material properties from sound. *Natural Computation*, pages 356–363, 1988.

[6] William W. Gaver. *Everyday listening and auditory icons.* PhD thesis, University of California, San Diego, 1988. Chair-Norman,, Donald A.

[7] R. A. Lutfi and E. L. Oh. Auditory discrimination of material changes in a struck-clamped bar. *J. Acoust. Soc. Am.*, 102(6):3647–3656, December 1997.

[8] Roberta L. Klatzky, Dinesh K. Pai, and Eric P. Krotkov. Perception of material from contact sounds. *Presence: Teleoper. Virtual Environ.*, 9(4):399–410, 2000.

[9] S. Lakatos, S. McAdams, and R. Caussé. The representation of auditory source characteristics: simple geometric form. *Perception & Psychophysics*, 59(8):1180–1190, 1997.

[10] C. Carello, K. L. Anderson, and A. J. Kunkler-Peck. Perception of object length by sound. *Psychological Science*, 9(3):211–214, May 1998.

[11] A. J. Kunkler-Peck and M. T. Turvey. Hearing shape. *J. of Experimental Psychology: Human Perception and Performance*, 26(1):279–294, 2000.

[12] M. Houben, A. Kohlrausch, and D. Hermes. Auditory cues determining the perception of the size and speed of rolling balls. In J. Hiipakka, N. Zacharov, and T. Takala, editors, *Proceedings of the 7th International Conference on Auditory Display (ICAD2001)*, pages 105–110, Espoo, Finland, 2001. Laboratory of Acoustics and Audio Signal Processing and the Telecommunications Software and Multimedia Laboratory, Helsinki University of Technology.

[13] William H. Warren and Robert R. Verbrugge. Auditory perception of breaking and bouncing events: a case study in ecological acoustics. *Journal of Experimental Psychology: Human Perception and Performance*, 10(5):704–712, 1984.

[14] N. J. Vanderveer. *Ecological Acoustics: Human perception of environmental sounds*. PhD thesis, Georgia Institute of Technology, 1979. Dissertation Abstracts International, 40, 4543B. (University Microfilms No. 80-04-002).

[15] W. W. Gaver. What in the world do we hear? an ecological approach to auditory event perception. *Ecological Psychology*, 5(1):1–29, 1993.

[16] W. W. Gaver. How Do We Hear in the World? Explorations in Ecological Acoustics. *Ecological Psychology*, 5(4):285–313, Apr. 1993.

[17] Bruno H. Repp. The sound of two hands clapping: An exploratory study. *The Journal of the Acoustical Society of America*, 81(4):1100–1109, 1987.

[18] X. Li, R. J. Logan, and R. E. Pastore. Perception of acoustic source characteristics: Walking sounds. *The Journal of the Acoustical Society of America*, 90(6):3036–3049, December 1991.

[19] E. Bruce Goldstein. *Sensation and Perception*. Brooks/Cole Publishing Company, Pacific Groove, CA, USA, 4 edition, 1996.

[20] Matthias Rath and Davide Rocchesso. Informative sonic feedback for continuous human–machine interaction — controlling a sound model of a rolling ball. *IEEE Multimedia Special on Interactive Sonification*, 12(2):60–69, April 2005.

[21] Matthias Rath. An expressive real-time sound model of rolling. In *Proceedings of the 6th "International Conference on Digital Audio Effects" (DAFx-03)*, London, United Kingdom, September 2003.

[22] Matthias Rath and Robert Schleicher. On the relevance of auditory feedback for quality of control in a balancing task. *Acta Acustica united with Acustica*, 94(1):12–20, January 2008.

[23] Federico Avanzini, Davide Rocchesso, and Stefania Serafin. Friction sounds for sensory substitution. In S. Barrass and P. Vickers, editors, *Proceedings of the 10th International Conference on Auditory Display (ICAD2004)*, Sydney, Australia, 2004. International Community for Auditory Display (ICAD), International Community for Auditory Display (ICAD).

[24] Christian Müller-Tomfelde and Tobias Münch. Modeling and sonifying pen strokes on surfaces. In *In Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, pages 6–8, 2001.

[25] Christian Müller-Tomfelde and S. Steiner. Audio-enhanced collaboration at an interactive electronic whiteboard. In J. Hiipakka, N. Zacharov, and T. Takala, editors, *Proceedings of the 7th International Conference on Auditory Display (ICAD2001)*, pages 267–271, Espoo, Finland, 2001. Laboratory of Acoustics and Audio Signal Processing and the Telecommunications Software and Multimedia Laboratory, Helsinki University of Technology, Laboratory of Acoustics and Audio Signal Processing and the Telecommunications Software and Multimedia Laboratory, Helsinki University of Technology.

[26] Matthias Rath. Energy-stable modelling of contacting modal objects with piece-wise linear interaction force. In *Proceedings of the 11th "International Conference on Digital Audio Effects" (DAFx-08)*, Espoo, Finland, September 2008.

[27] Apple Inc., 2010. http://developer.apple.com.

[28] Khronos Group, 2010. http://www.opengl.org.

[29] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[30] Khronos Group. Glut - the opengl utility toolkit, 2010. http://www.opengl.org/resources/libraries/glut/.

[31] Encyclopædia Britannica Online, 2010. http://www.britannica.com.

[32] Apple Inc. Apple portables: About the sudden motion sensor, 2008. http://support.apple.com/kb/HT1935.

[33] Amit Singh. Mac os x internals: A systems approach, 2005. http://osxbook.com/software/sms/.

[34] Suitable Systems. Smslib, 2010. http://www.suitable.com/tools/smslib.html.

[35] Unimotion. http://unimotion.sourceforge.net/.

[36] K. van del Doel, P. G. Kry, and D. K. Pai. Foleyautomatic: Physically-based sound effects for interactive simulation and animation. In *Proc. ACM Siggraph 2001*, Los Angeles, Aug. 2001.

[37] James A. Moorer. The synthesis of complex audio spectra by means of discrete summation formulae. *Journal of the Audio Engineering Society*, 24 (Dec.):717–727, 1975. (Also available as CCRMA Report No. STAN-M-5).

[38] Tim Stilson and Julius Smith. Alias-free digital synthesis of classic analog waveforms. In *Proceedings of the 1996 International Computer Music Conference - Hong Kong*, San Francisco, CA, USA, 1996. ICMA.

[39] Vesa Välimäki. Discrete-time synthesis of the sawtooth waveform with reduced aliasing. *IEEE Signal Processing Letters*, 12(3 (Mar.)):214–217, 2005.

[40] Inc. Wikipedia Foundation. make (software), 2010. http://en.wikipedia.org/wiki/Make_(software).

[41] Dimitri van Heesch. Doxygen, 2009. http://www.doxygen.org/.

[42] Jack audio connection kit, 2009. http://jackaudio.org/developers.

[43] Inc. Mark J. Kilgard, Silicon Graphics. The opengl utility toolkit (glut) programming interface, 1996.

List of Figures