AIRS
An Architecture for Interactive Real-time Systems

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Michael L. Neilsen

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Professor Peter J. Willemsen

August 2010

© Michael L. Neilsen 2010

Acknowledgements

I am grateful for all of the support and encouragement I have received from my adviser, Pete Willemsen, and my good friend, R. Isle.

Abstract

Interactive real-time systems lend themselves to a variety of applications including entertainment, education, 3D modeling, art and design, security and surveillance, and medical training. These types of systems may be implemented as interactive simulations, virtual and augmented reality systems, computer and video games, and artistic installations.

In this paper, an architecture called AIRS (Architecture for Interactive Real-time Systems) is presented. AIRS is a client-server architecture. Its aims are to provide a flexible framework for light-weight clients to collaborate and communicate with a server through message passing. The server's role is to maintain state, access local or remote data, and facilitate media output, such as playing audio and displaying video. Examples of systems that could use this architecture are presented.

In the spirit of working toward a concrete implementation of AIRS, a prototype called Duck was developed. Duck is an interactive real-time system that allows users to create and position geometric objects in a 3D scene, associate audio samples with them, and interact with them indirectly through a physics simulation.

Three test sessions were conducted. Each user was prompted with audio that a scene produces and was instructed to create a scene that matches the audio. Three such scenes are included, progressing in difficulty. On average, users spent 1.5 minutes and 1.3 attempts to solve the first scene and 3 minutes and 2 attempts to solve the second scene. No users solved the third scene.

Contents

List of Tables v

List of Figures vi

1 Introduction 1

2 Background and Architecture 3

2.1 Architecture 6

2.1.1 Configuration Examples 9

2.2 Clients 11

2.3 Audio Creation System 14

3 Development 17

3.1 Real-Time Systems 17

3.1.1 An Example 18

3.2 A Functional Duck 20

3.2.1 Current Functionality 20

3.2.2 Testing 21

3.3 Duck's Main Loop 21

3.4 User Input 23

3.5 GUI 25
3.5.1 CEGUI 25
3.5.2 Initialization 26
3.5.3 Callbacks 26
3.6 Spatialized Audio 27
3.7 Physics 29
3.7.1 Bullet Physics 30
3.7.2 Collisions 32
3.8 Event System 33
3.9 Rendering 34
3.10 frameEnded 35
3.11 Selections 35
3.12 Recording and Testing 37

4 Results **40**
4.1 Introduction and "Free Play" 40
4.2 Solving Scenes 40
4.3 Questionnaire 42

5 Future Work **47**

Bibliography **50**

List of Tables

4.1 Number of attempts made to solve a scene during each test session. The above table shows the number of attempts made to solve each scene. If the user was unable to solve a scene within the time alloted, its column is marked with a '-'. 41

4.2 Average number of attempts for each completed scene. Averages are computed from users that were able to solve the scene. '-' denotes that no users solved the scene. 42

4.3 Approximate amount of time to solve a scene during each test session. The above table shows approximately how much time was needed to solve each scene. Time is displayed in minutes and seconds. If the user was unable to solve a scene within the time alloted, its column is marked with a '-'. 45

4.4 Average elapsed time for each completed scene. Averages are computed from users that were able to solve the scene. '-' denotes that no users solved the scene. 45

4.5 Questionnaire ratings. The above table shows the ratings given for the questionnaire for each test session. 46

4.6 Questionnaire rating averages. The above table shows the average rating for each question in the questionnaire. 46

List of Figures

2.1 An illustration of AIRS's client-server architecture. Clients send messages to the server for processing. The server handles incoming messages by performing the requested operations and consults data banks when necessary. Output is sent to attached output devices. 7

2.2 Spatial Audio Demo. In this screenshot, the red circle represents a looping audio sample. The arrow represents the listener. Audio is being played ahead and slightly to the left of the listener. 11

2.3 A scene implementing a drum. The mallet falls and strikes the slab, which causes an audio sample to play. The slab is offset and returns to rest. The audio sample stops playing. 16

3.1 Duck's main loop. Each frame has three stages: frameStarted, rendering, and frameEnded. In frameStarted, input is processed, the listener's position is updated, and the physics simulation is stepped. In the rendering stage, Ogre renders one frame. In frameEnded, the GUI's status is updated. 23

3.2 Duck's event system. Every geometric object in a scene has zero or more triggers associated with it; every trigger has zero or more events associated with it. When a trigger fires, all of its associated events are executed. 33
3.3 A typical SolutionTable. The "Offset (ms)" column indicates the number of milliseconds after the recording began that the PlayAudioEvent was executed. The "PAE" column indicates which PlayAudioEvent was stored in that TableEntry. Each row represents a TableEntry. Note that while PAEs are denoted by numeric values here, a TableEntry stores an actual copy of a PAE. 37
3.4 Relationship between SolutionTable and TableEntry. Methods and members not discussed in this section have been omitted from the diagram. 38
3.5 A SolutionTable before and after normalization. Normalization only changes the offsets of the table's entries. The number of entries and the associated PlayAudioEvents are not affected or altered. 39

4.1 Scene I. The highlighted object is the only object being affected by gravity. The other objects are static–that is, they have zero mass and remain stationary. 42
4.2 Scene II. The highlighted object is the only object being affected by gravity. The other objects are static–that is, they have zero mass and remain stationary. 43
4.3 Scene III. The highlighted object is the only object being affected by gravity. The other objects are static–that is, they have zero mass and remain stationary. 44

1 Introduction

First, take any data as input. Server log files, data packets exchanged over a network, audio from a shortwave radio, the path of a rain drop sliding down a window, footage of a medical procedure...

Next, pick your processing and output. How do you want to see, hear, or feel this data? Do you want to distribute, disassemble, analyze, or simulate this data?

Now introduce a client. How will you interact with your processing and output? Keyboard and mouse? A Head mounted display and tracking equipment? A touch screen and stylus? A mobile phone? A modified synthesizer?

Finally, pick your colleagues, collaborators, and adversaries. Do you want to share your output with others? Do you want to share your input? If so, are other clients directly connected to the processing and output, or are they distributed remotely? Are they in another country? How do you want to share your experience?

Architecture for Interactive Real-time Systems (AIRS) serves as an architecture for facilitating the construction of interactive real-time systems. It defines the roles above and establishes the interaction between them. Applications built on top of AIRS should supply data as input, implement some form of processing and produce some sort of output, and define client interaction and collaboration.

AIRS provides a guideline for how an interactive real-time can be structured. Using an existing architecture helps developers by guiding design decisions and implementation details. As AIRS was conceived from designs for an interactive audio

creation system[1], it caters to multimedia rich interactive environments.

A prototype application was developed to explore AIRS. The initial goal of writing this prototype was to realize the interactive audio creation system in its entirety. To do this, many abstractions were necessary to provide for the system's modularity. As design and implementation carried on, patterns in overall design emerged; these were extracted out, and the principal components of AIRS surfaced. The goals of writing the prototype expanded into extracting out components that were both generalizable and functional. These would be serve as the foundation for an implementation of AIRS.

In this paper, AIRS is described as an architecture. Examples of applications that could run on top of the architecture are given. An audio and visual production system is described in terms of AIRS components. The development of a prototype of this implementation is discussed from the stand-point of an interactive real-time system. Finally, the results of three test sessions are reported.

[1] described in section 2.2 as "Interactive Audio Creation" client type and in more detail in section 2.3

2 Background and Architecture

This chapter describes and classifies AIRS as an architecture for interactive real-time systems, describes motivation for AIRS's existence, presents a suitable architecture for it, and provides several examples of implementations of applications running on top of the architecture.

AIRS is a interactive real-time system for audio and video creation. As a client-server system, the server provides computations for physics and audio synthesis and filtering, data access, and hardware and media output. The client is a specific application that serves the users' goals for audio and video creation. Examples of implementations of clients are servers are presented in later sections.

Interactive real-time systems have characteristics that lend themselves to a variety of useful applications. Some examples are:

- Video conferencing using augmented reality to better capture large gestures, spatial distances, and eye contact [11].
- Mobile security and surveillance systems that can be tele-operated through eye tracking [9].
- Entertainment through interactive games and story-telling [12, 10].
- Simulating surgical procedures, such as cardiac intervention, to reduce risks and costs of training [13, 8].

- A 3D interactive learning environment for teaching deaf children math concepts and sign language [1].
- A software system that tracks eye movements and eye pauses to allow children with severe motor impairments to draw pictures [6].
- Using haptic feedback to enhance 3D modeling and texturing [5].

Using interactive real-time systems, users can receive near-immediate feedback to their actions. In response to this feedback, users can formulate their next actions accordingly. This exchange lends well to creative systems because it welcomes a variety of reactions from the user, which may need to be formulated and planed in advance, or on the spot. In that sense, the user is exercising creativity through making decisions and performing actions within the system or environment actively and dynamically.

These systems share characteristics in common with computer games and real-time simulations. The actions that the user takes with such systems produce immediate results that affect the surrounding environment. The environment may continue to change over time, based on the users' actions or the actions and behavior of various entities in the system. An example from a recent computer game is given in section 3.1.1.

This is different from the behavior of many desktop applications. These may function in somewhat of a fixed sequence of states. For example, a web browser may appear idle until the user inputs a web address and submits the request. After the request is placed, the web browser fetches the content and then displays it. Afterwards, the browser returns to its apparent idle state, waiting for more input. On the other hand, an interactive real-time system may continuously apply operations and update internal state during periods of user inactivity, and after receiving input,

perform a variety of actions, depending on a combination of internal state and user input. Interestingly, current web browsers and web sites are exhibiting some of these characteristics; for example, an RSS news aggregator might update itself in between user input and recommend news stories based on the types of stories the user reads frequently.

Many desktop software packages provide audio and video creation facilities; however, many of these are either not interactive real-time systems or they require a background knowledge of music theory, sound synthesis, and computer programming. AIRS aims to fill this gap and provide an interactive real-time system that does not pre-suppose a background of music theory, synthesis, or programming knowledge.

One software package in particular provided inspiration and formed a basis for formulating AIRS's aims. Pure Data is software for building real-time audio applications using a visual programming language. Pure Data can accept raw (general) data as input, and users can design their programs to interpret that data in various ways to produce audio, filtering and effects, and analysis. General purpose software can be written in Pure Data as well, but the programs are still subject to running through a Pure Data process. There are many extensions available to Pure Data that add functionality such as access to network streams and graphics hardware via OpenGL. Techniques such as "Evolutionary Computation" have been implemented with Pure Data to produce non-linear sound synthesis [4].

With a proper understanding of how to work with Pure Data, the extensibility and flexibility of Pure Data provides a wide range of possibility in terms of creative expression. However, new users without prior prior programming experience may be dissuaded by the effort involved to learn Pure Data as a visual programming language. Furthermore, new users without at least some background in audio synthesis theory[1]

[1] For interested readers, [7] provides a nice introduction to synthesis theory.

may find it challenging to manipulate raw data to produce output with Pure Data's constructs. Other challenges new users may face are audio latency issues and Pure Data application design.

2.1 Architecture

In this section, AIRS's architecture is detailed. Motivations and examples of configurations are provided. A configuration describes the roles and resources of the components of the architecture.

Three commercial game engines currently used for popular games are Unreal Engine, id Tech, and Source Engine. Game engines such as these typically offer functionality for graphics, audio, memory management, events, multimedia, physics, and artificial intelligence, scripting, and networking [8]. These components interface with external data (such as network data, game maps, and models) and hardware (such as network interfaces, graphics cards, input from keyboards, mice, and gamepads). AIRS categorizes the game engine's functionality, external data, and hardware into architectural components. This generalization is advantageous because it does not enforce a specific line of functionality for any of the components of the architecture.

Figure 2.1 illustrates AIRS's architecture. The core of AIRS is the server. The server receives and processes commands from clients. Output (such as video, audio, or raw data) is sent to output devices. Attached to the server are data banks that contain data for additional resources or processing.

The architecture is provided as a client-server system. The purpose of this is to decouple client implementations and server configurations. Servers can be configured to output specific types of media (for example, only audio, only video, or both) and run over a network. Clients can be implemented on a variety of devices. The require-

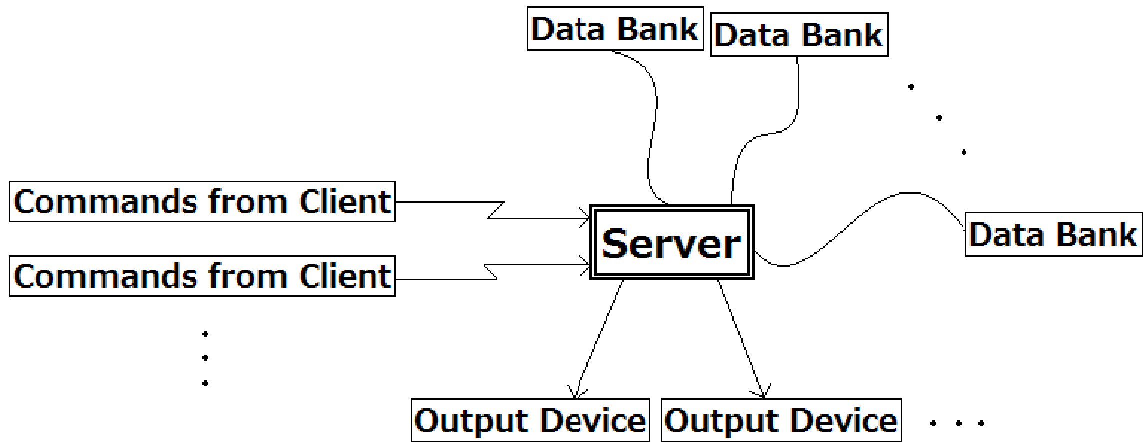

Figure 2.1: An illustration of AIRS's client-server architecture. Clients send messages to the server for processing. The server handles incoming messages by performing the requested operations and consults data banks when necessary. Output is sent to attached output devices.

ments for clients are intentionally slim–minimally, a client should be able to send a message to a server, where it will then be processed. Small requirements are desirable for implementing AIRS clients on hardware such as phones or hand-held gaming consoles, which may not have adequate support for producing output themselves or handling computations such as physics calculations and audio synthesis and filtering. Clients and servers can also be implemented together on one system, assuming the adequate resources are present, and communicate over a loopback connection.

Heavy computations may be localized in the server if advantageous. For example, computations such as physics calculations, data analysis and cryptography could be performed on the server and then sent directly to an output device. This is advantageous because it reduces requirements on a client's resources and minimized communication time. Clients may contribute to or participate with computations if desired. This may be appropriate for a configuration where the client are server are centralized on one system.

Data banks are resources that are inspired by Pure Data's approach to viewing raw data as a source of input for processing, effects, and output. Data banks can store or provide an interface for retrieving data that can be useful for an implementation's functionality. For example, data banks may hold multimedia data (such as audio or video), random numbers, coordinate and geometry information, data from a network stream, MIDI signals, and so on. Data banks are intentionally placed close to the server so that the server can be implemented to access the data as quickly as necessary or possible. Placing data banks near the server also keeps them out of direct interaction with clients, which reinforces the decoupling between the server and clients.

Output devices are attached to the server. Examples of output devices are not limited to video displays and speakers. Output can be sent to other devices such as network interfaces or serial ports. The output destination is defined by the role or desired goal of the system. Multiple output devices can be used. For example, a system may output audio to several speakers, video to a projector, and stream playback to other clients over a network device.

To communicate to a server, a client must send a message. A message encapsulates an instruction for the server to perform an action or update data in a data bank. Messages are intended to be minimal so that they can be delivered rapidly so that delivery latency is not problematic. Some examples of messages may be "play the audio sample 'drum.wav'", "change the position of the listener in the scene", "mute the left channel of playback", or "disconnect all other clients". Messages should contain sufficient data for the server to act upon the request, but clients should be encouraged not to attach large amounts of data to commonly transmitted messages. When possible, large data sets should be accessible from a data bank instead.

The types of messages sent from clients and supported by servers are dependent on

the aims of the system, its supported output devices, and the types of computations the server is designed to perform. For example, it would not make sense for a system running without a video display to respond to a message placing a request to "blank the video display".

The architecture is designed to support multiple clients. This allows several clients to interact and contribute to the system's functionality by sending messages independently of one another. The server can maintain state information regarding clients and their messages. Multi-client support enables the server to host collaborative installations. For example, clients from around the world can interact over the Internet to produce audio output and visualize data.

2.1.1 Configuration Examples

This section describes examples of specific configurations. A configuration is a description of the roles and resources of the components of AIRS's architecture. Each configuration describes the roles of the server, client(s), output devices, data banks, and the system's intended functionality. The intended functionality of the system will describe how the client will function, what devices clients may use, and the types of messages supported by the server. In that sense, a configuration not only describes the roles and resources of the architecture, but also can be used to define or guide the development and presentation of an client. Clients are discussed in more detail in section 2.2.

Name: Minimal

Server: Running on a laptop.

Client: Laptop keyboard.

Output Devices: Laptop speakers.

Data Banks: Single Data Bank holding a single audio sample.

Functionality: The user presses the space bar on his or her keyboard. This sends a command to the server to play the audio sample. The user hears the audio sample play through his or her laptop speakers.

Name: Spatial Audio Demo

Server: Running locally or remotely on personal computer. Accessed over a network connection.

Client: Running locally or remotely on personal computer. Sends messages to Sound Server over network connection. Using W, A, S, and D on their keyboard, the user can move the position and orientation of a "listener". Using the arrow keys on their keyboard, the user can move the position and orientation of a "sound source".

Output Devices: Personal computer speakers for audio. Attached monitor or LCD for video.

Data Banks: Two Data Banks. One Data Bank holds a single audio sample. The other Data Bank stores the position and orientation of a "listener" and "audio source".

Functionality: The "listener" represents the user relative to a "sound source" in 3D space. As the user moves and manipulates the listener and the sound source, he or she hears the sample looping from the relative position and direction of the sound source. The user tracks the position and orientation of the listener and sound source graphically. For example, the if the user is positioned to the left of the sound source, the user will hear the sample looping in their right speaker. If the sound source is behind the listener and the listener and sound source are facing in opposite directions, the user will hear the sample looping "behind" them and somewhat muted (due to the opposite facing directions). Figure 2.2 illustrates one implementation of

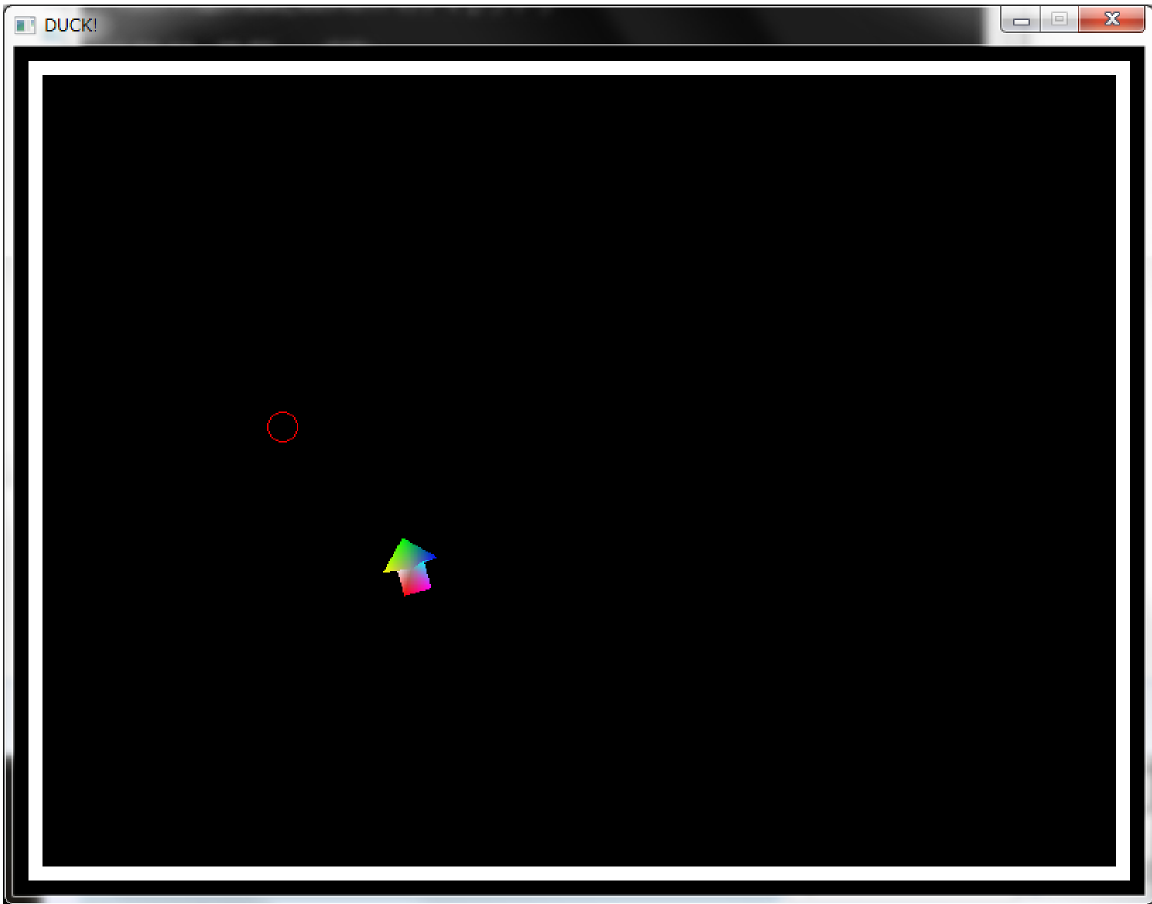

Figure 2.2: Spatial Audio Demo. In this screenshot, the red circle represents a looping audio sample. The arrow represents the listener. Audio is being played ahead and slightly to the left of the listener.

this configuration.

The next section describes the role of a client in AIRS's architecture.

2.2 Clients

Minimally, the client's purpose is to interact with the server via messages. Messages are exchanged via a network connection. This requirement is intentionally min-

imal so that clients can be implemented on a variety of devices. For example, clients can be implemented on hand-held gaming systems, mobile phones, head mounted displays, and desktop computers. Clients can produce their own output and perform their own calculations as well, so long as their hardware supports it. Some examples of clients are presented here.

Client Type: Data Visualization

Description: A server's data banks are a database of scientific data. This data could be tabulated and accessed through traditional methods such as database lookups, but would convey more information if visualized in real-time and supported dynamic interacted. To achieve this, a client could be implemented to run in a head mounted display. The user's motions could be followed by motion tracking devices. The user can visualize the data and manipulate it with hand gestures and navigate through the data by walking around.

Client Type: Artistic Installation

Description: The server's output devices are video projectors and surround sound speakers. The projectors are projecting onto the walls of of a room, and the speakers are positioned around the corners of the room's ceiling. Upon entering the room, users can take a spherical device that resembles the size and weight of a tennis ball. In this example, the spherical device is the client. Participants can throw these devices to one another and bounce them around the room. The balls' positions and orientations are tracked, and as they are passed around, the projectors display particle effects over a dark background that blend and weave with one another relative to the paths of the balls crossing. The speakers play ambient noise that becomes more and less gritty relative to the balls' velocity. When a collision occurs, the speakers rumble a

low frequency sound and the projections quickly lighten and the darken.

Client Type: Interactive Audio Creation

Description: Clients interact with a 3D scene via a desktop computer (display, mouse, and keyboard) or head mounted display and tracking devices.

The user can navigate this space (a scene) and create primitive objects such as cones, spheres, cylinders, cubes, and ropes. Primitive objects can be combined into composite objects. Global physics settings such as gravity can be configured. Objects can maintain internal state or access the state of the scene.

Per-object physics settings are present as well–that is, two sphere objects may have different mass and restitution settings that would affect how they react to collisions with one another. The user can load audio samples from disk and associate events in their scene with audio events. For example, when two spheres collide with one another, an audio event can be triggered to occur, and a sample can be played. This event can in turn trigger another event, which may affect another object. For example, after two spheres collide and audio is played, cubes can spawn and fall from above onto the spheres, causing more audio events to be triggered.

Special objects can be created to filter or synthesize audio. For example, the surface of a cube could be configured to filter audio so that any audio events triggered upon a collision with the cube will have a muted, or flat sound. Another example would be an object that can accept "audio input" and send out "audio output". Audio that comes through the input channel is filtered and sent out the output channel. Examples of this could be voice changers (filter a man's voice to sound like a woman's voice and vice versa), amplifiers, and echo effects.

Synthesis objects can also be created. They can be triggered to synthesize an audio effect where synthesis parameters can come from internal state or the state of

the scene. For example, a synthesis cube may “collect” collisions that occur with it and build up a profile of how the synthesized sound will be constructed. When the synthesis cube is triggered to synthesize audio, it can vary based on how many collisions occurred with it or an average velocity of collisions.

Interactive audio creation was the focus of AIRS's development. The remaining discussion in this paper will be in the context of the latter described system.

2.3 Audio Creation System

This section describes the computations and libraries involved in the audio creation system and an event system that glues together the objects in a scene. An example of a scene is presented.

This audio creation system's core computations are for physics simulations and audio synthesis and filtering. A physics simulation serves to increase the quality of an interactive real-time system by providing an infrastructure for automatically updating the state of an environment at regular time intervals. Audio synthesis and filtering can produce audio and alter how it sounds.

Bullet Physics is a suitable library for AIRS's physics computations. Bullet Physics is an open source physics library written in C++. It provides rigid and soft body dynamics and collision detection. Rigid body dynamics pertain the mechanics of bodies that do not deform. Examples of this are non-deformable solids such as cones, spheres, cubes, and cylinders. Soft body dynamics pertain to the mechanics of deformable bodies that still retain some shape. Examples of this are ropes, fabrics, and compressible or deformable solids.

For audio synthesis and filtering, CLAM (C++ Library for Audio and Music)

appears to be a suitable library for providing filtering and analysis capabilities to AIRS. CLAM can be used as a C++ library or an application framework. As a library, CLAM provides cross-platform support for analysis and synthesis as well as audio output device access. If used in conjunction with CLAM's NetworkEditor application, users can create analysis, synthesis, and filter patches to be applied to raw data [2]. Much of CLAM's internal messaging is represented by XML, which can be transferred across a physical network. As an application framework, CLAM could supplement AIRS's extensibility by allowing users without programming knowledge to create analysis, synthesis, and filtering mechanisms.

For example, a user may want to construct a reverb effect to include in his or her scenes. This should be possible to build and export using CLAM's NetworkEditor. Stand-alone applications built with CLAM could also interface with AIRS's networking components to provide another avenue for interacting with AIRS's functionality. AIRS could also benefit from using CLAM's analysis functionality to detect patterns or properties in audio to generate programmed responses.

The user will be concerned with how objects interact within their scene. An event system provides a means to manage these interactions. An event is an action that occurs within the scene, such as playing audio or changing the properties of an object. Events are executed by triggers. Examples of triggers are collisions and objects crossing set thresholds. An event can also be used as a trigger, allowing chaining events.

The following figure is an example of how the event system can be used to implement a drum with a sphere, cone, and box. First, velocity is applied to the drum stick. This can be an event in and of itself, or it can be a consequence of the physics engine stepping. Once the drum stick hits the drum, a collision is detected and, as a trigger, fires a "play audio" event. This plays an audio sample, such as the sound

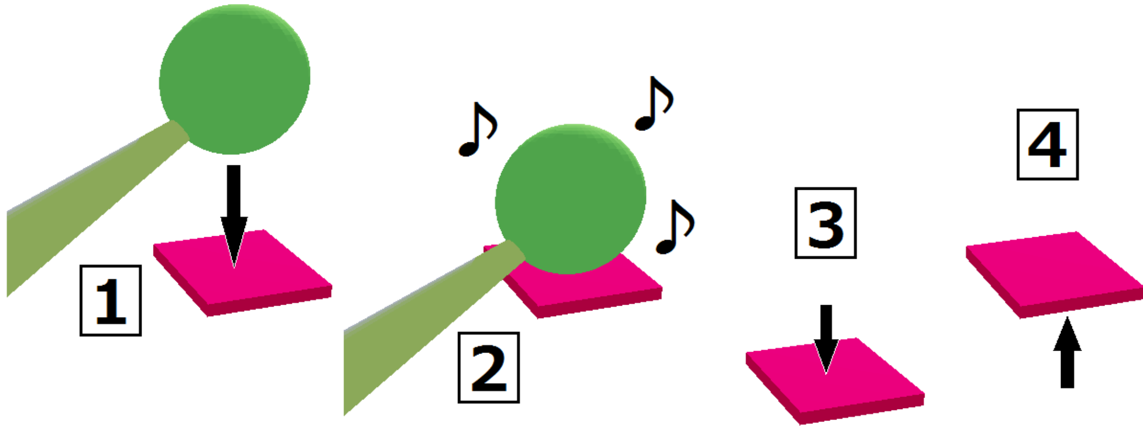

Figure 2.3: A scene implementing a drum. The mallet falls and strikes the slab, which causes an audio sample to play. The slab is offset and returns to rest. The audio sample stops playing.

of a snare drum or a gong reverberating. The collision also fires an event to transfer the drum stick's vertical velocity into the block, which gives the block a negative vertical velocity. Next, the block is displaced downward, on behalf of its new velocity, until a "low elevation" threshold is reached. This threshold is a trigger that fires an event that reverses the block's velocity upwards. Finally, the block rises until a "high elevation" threshold is reached. This trigger fires and two events occur. First, the block's velocity is set back to zero, so it is at rest again. Second, if the audio sample is still playing, it fades out and stops. The user may strike the box again to repeat the effect.

3 Development

This chapter describes the development of a prototype application called Duck. Duck is an interactive real-time system that allows users to build a 3D scene that creates audio. Users can engage Duck in a "puzzle mode" that prompts the user with a sequence of audio, and in turn, their task is to build a scene that produces similar audio.

3.1 Real-Time Systems

Real-time systems characteristically exhibit looping behavior, beginning after initialization, and ending when instructed to quit. For instance, the main loop of a computer game might resemble the following:

```
while (running)
{
  processUserInput();
  updateArtificialIntelligence();
  handleCollisions();
  moveEnemies();
  playSoundEffects();
  renderSingleFrame();
}
```

While the above loop is fairly straight-forward, each of the loop's procedures may vary widely in complexity. In fact, each procedure may exhibit its own looping

behavior. For example, if the user is pressing multiple keys simultaneously, the *receiveUserInput* function above may need to loop over some set to determine which keys are being pressed.

Another point to notice in the above loop is the relation between rendering and iterations. Each time the above loop is iterated, a frame is rendered to the screen. Relative to other procedures in the main loop, rendering may consume a large amount of time. Because of the strong relation between rendering one frame and iterating over the loop, other procedures may be adversely affected by delays between being called. This timing information is crucial to manage to create a responsive real-time system, and illustrates one of the many challenges involved with building such a system.

3.1.1 An Example

Many computer games are examples of real-time systems. Left 4 Dead is an example of a relatively recent computer game that exemplifies one such interactive real-time system. Left 4 Dead is a first-person shooter that supports multiplayer gameplay over the Internet.

In this game, players cooperate with one another to navigate through a zombie infested environment to reach safe points. As players work their way through each map, their behavior shapes the obstacles that lie ahead of them. This is accomplished by the game's artificial intelligence mechanism, "The Director".

The Director monitors various activity in the game, such as the players' movement, how quickly they move through the map, how often they fire guns, and how often they miss shots. Based on this data, The Director may place difficult enemies in the path of players that are doing well, and ease the difficulty on players that appear to be struggling by placing helpful items such as health kits and weapons along their

path [3]. In the sequel, Left 4 Dead 2, there are several maps that support dynamic weather effects–if the players are doing too well, The Director will start a heavy storm with rain and lightning that reduces players' range of vision and hearing to balance the difficulty.

In both games, the sounds and music that the players hear reflect the state of the environment. When players are near stronger enemies, the game's music is supplemented with an ambient melody that reflects the nature of the enemy. For example, if the enemy is light-weight and fast, the supplemented melody is short and sharp and is played by a violin or the high notes of a piano; if the enemy is fat and slow, the melody has a sauntering pace and is played with a cello or the low notes of a piano [3].

The players' behavior in the game influences The Director's decisions, which in turn affect the environment, which in turn affect the players' behavior. This type of cyclic exchange is common in interactive real-time systems; as the user interacts with the system, one subsystem operates accordingly, which may affect other subsystems. While Duck is no where near as complex as Left 4 Dead, they both share challenges that are common to designing and implementing interactive real-time systems.

One challenge of designing a real-time system is determining how each component of the system coordinates with one another. An overview of Duck's functionality as seen from the user is presented, followed by a description of the subsystems present in Duck, their roles, and how they are coordinated with one another in Duck's main loop.

3.2 A Functional Duck

3.2.1 Current Functionality

Duck, in its current state, allows the user to create geometric objects in a scene. There are four geometric object types: a box, cylinder, sphere, and cone. Once an object is created, it can be positioned anywhere in the scene.

Each object has properties associated with it. The object can be given a name, audio sample, and values mass and restitution. The object's name may become useful in the future for such features as creating and exporting compound objects. The audio sample will play whenever the object comes in contact with another object. Mass represents the object's physical mass, and restitution affects how "bouncy" the object is.

The physics simulation can be toggled on and off. When the physics simulation is active, objects are affected by gravity. There are two exemptions from this. First, if an object is configured to have a mass of zero, it will be a "static" object that holds its position in space. Second, objects can be "grabbed" by the user. When an object is grabbed, the user has total control over where it is placed, regardless of the state of the physics simulation.

During collisions between objects, the audio samples associated with each object in the collision are played. If the sample has one audio channel, it is played back spatially. .wav files in Duck's "samples" directory are loaded when Duck initializes, and from then on can be selected in the object configuration dialog.

3.2.2 Testing

For the purpose of testing, Duck currently includes a puzzle mechanism. There are three pre-loaded scenes. Each pre-loaded scene consists of several objects configured in such a way that when the physics simulation is activated, the collisions that occur produce a distinct sequence of audio effects. Currently, Duck comes with four audio samples: "bass.wav", "cymbal.wav", "ride.wav", and "snare.wav". The user's task is to construct a scene that can produce a chain of audio effects that closely matches the pre-loaded scene's output.

When one of these scenes is loaded, the screen blanks and the physics simulation is activated. The subsequent audio that's created from the scene is recorded for later playback. The pre-loaded scene can be manually cleared, un-blanking the screen and leaving a blank scene. The recording is preserved.

As the screen was blanked when the demo scene was loaded and recorded, the user's only guide is the recorded audio, which can be played back as a hint. When the user has constructed a scene that he or she feels reproduces a similar chain of audio, he or she can test against the solution by recording their attempt and comparing it to the recording. This counts as an attempt. Multiple attempts can be made. If the recordings match within a set threshold, the user is shown a message that they have solved the puzzle and how many attempts they made in doing so.

The next section discusses Duck's subsystems and how they're related to one another.

3.3 Duck's Main Loop

Duck uses "Object-Oriented Graphics Rendering Engine" (Ogre) for rendering. Ogre is a multi-platform, open source graphics rendering engine written in C++.

It supports DirectX and OpenGL. Similar to the example main loop given in the introduction to this chapter, Duck renders one frame per main loop iteration. Duck's main loop is basically the following:

```
while (running)
{
  ogre->renderOneFrame();
}
```

Calling `ogre->renderOneFrame();` begins a chain of procedures such as gathering user input, playing audio, and moving shapes and objects on the screen. This section describes the abstractions provided by Ogre to specify when these procedures occur.

A key component of Ogre's rendering loop is an interface called *FrameListener*. *FrameListener*s are registered with Ogre and subsequently called at the beginning and end of Ogre's internal render loop. Two important methods of the *FrameListener* interface are

```
bool frameStarted(const FrameEvent &evt);
bool frameEnded(const FrameEvent &evt);
```

At the beginning of each render loop, each registered *FrameListener*'s *frameStarted* method is called. Afterwards, Ogre renders the frame. Finally, at the end of each render loop, each registered *FrameListener*'s *frameEnded* method is called.

A *FrameEvent* encapsulates how much time has passed since the last render loop event or frame. This timing information is useful for ensuring that the physics simulation is stepped by an appropriate time value. This will be discussed in more detail later in this chapter.

Duck uses a single *FrameListener*. Its *frameStarted* method performs the following:

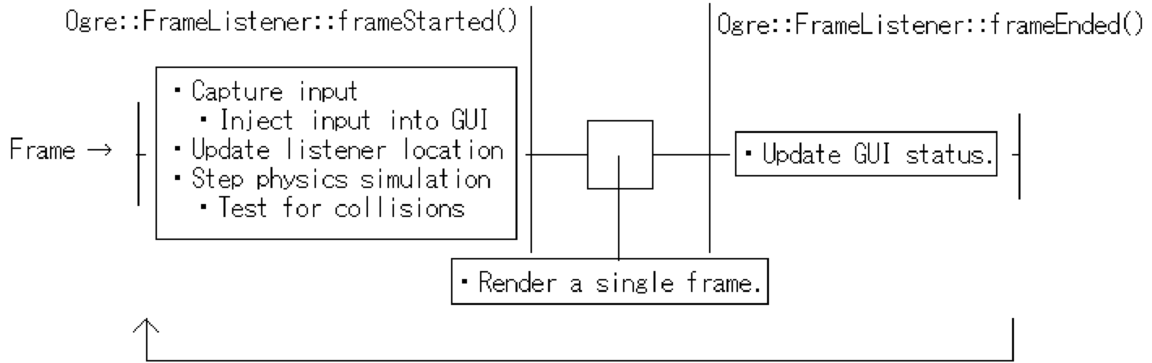

Figure 3.1: Duck's main loop. Each frame has three stages: frameStarted, rendering, and frameEnded. In frameStarted, input is processed, the listener's position is updated, and the physics simulation is stepped. In the rendering stage, Ogre renders one frame. In frameEnded, the GUI's status is updated.

1. Capture mouse and keyboard input.

2. Update listener's position for spatialized audio.

3. Update physics simulation.

Its *frameEnded* method updates the status of Duck's GUI.

The following sections will describe each of Duck's subsystems roughly in the order that they are encountered during each of the main loop's iterations. The first action performed by the *frameStarted* method is processing user input.

3.4 User Input

Duck's keyboard and mouse support is handled through Open Source Input System (OIS). Input events can be handled in two ways: by polling OIS devices directly (unbuffered input) and by setting up event listeners (buffered input). An advantageous location to handle input in a system using Ogre is in the *frameStarted* or *frameEnded* method of a *FrameListener*.

OIS can be polled through instances of *OIS::Keyboard* and *OIS::Mouse*. For example, given

```
OIS::Mouse* m_pMouse;
OIS::Keyboard* m_pKeyboard;
```

OIS can be polled by the following:

```
// Checks if the left mouse button is down.
if (m_pMouse->buttonDown(OIS::MB_Left)) { // ...

// Checks if "C" is down.
if (m_pKeyboard->isKeyDown(OIS::KC_C)) { // ...
```

This is referred to as unbuffered input. This can be placed directly inside of a *frameStarted* or *frameEnded* method. Unbuffered input, while relatively simple and straightforward to utilize, does not track key state between frames. Additional bookkeeping is necessary to determine whether a key is being pressed or held down throughout multiple frames.

An alternative to this is to use event listeners. Event listeners are callbacks that are registered with OIS and then subsequently called when specific types of events occur. For example, methods for handling events such as key presses and key releases can be defined. This is also known as buffered input. Buffered input maintains input state between frames.

Buffered input can be used in conjunction with polling the OIS devices directly. For example, on a key release event, OIS can be polled to find out if another key is down; this may be useful for implementing key combinations such as CTRL+C or CTRL+V.

Some polling behavior is still necessary when using buffered input. Device inputs need to be "captured" by OIS each frame. For example, Duck's *FrameListener*'s *frameStarted* event does this as follows:

```
ogreFramework->m_pKeyboard->capture();
ogreFramework->m_pMouse->capture();
```

OIS can then determine whether to call registered event listeners based on the current state of input.

When one of Duck's user input listeners is called, it notifies the graphical user interface (GUI) system of the user's action. The next section discusses the GUI system in more detail.

3.5 GUI

Duck's user input listeners are set up to forward events to the GUI system for processing. When user input is captured in *frameStarted*, the OIS listeners "inject" the input into the GUI system. This is important for displaying the mouse cursor in the correct location and ensuring that GUI elements such as dialogs, windows, buttons, combo boxes, and text boxes behave appropriately when interacted with. This behavior can be specified through registering callbacks with the GUI elements.

The rest of this section discusses the GUI system and this process in more detail.

3.5.1 CEGUI

Duck uses Crazy Eddie's GUI system (CEGUI) for managing its graphical user interface. CEGUI is a multi-platform, open source C++ library for creating and displaying user interfaces in graphical applications using OpenGL or DirectX. CEGUI

makes use of XML files for configuring layouts and GUI properties, such as window and GUI element locations. GUI elements are referenced in code by the names specified for each element in the XML file for that GUI.

3.5.2 Initialization

CEGUI integrates with Ogre by providing a library and class called *OgreRenderer*. By initializing CEGUI with *OgreRenderer*, GUI elements are drawn using Ogre's rendering mechanisms. This is a welcome abstraction for at least two reasons. First, it allows the GUI system to function on top of Ogre's cross-platform rendering support. Second, the details of how and when GUI elements are rendered is hidden from the user. Once initialized properly, the GUI is displayed when frames are rendered. CEGUI includes support for other renderers as well.

When CEGUI is being initialized in Duck, a visual theme, mouse cursor image, and default font are set. As mentioned above, CEGUI uses XML files for configuring layouts; layouts can be loaded by specifying the name of the corresponding XML file. While Duck currently only uses one GUI layout, multiple layouts can be used in an application. The theme, cursor, and fonts settings can be shared over active layouts to provide a uniform look and feel.

3.5.3 Callbacks

CEGUI is notified of user interaction such as mouse movement, mouse clicks, and key presses through the various "inject" methods supplied by *CEGUI::System*. Some examples of these methods are

```
bool CEGUI::System::injectMouseMove(float delta_x, float delta_y);
bool CEGUI::System::injectMouseButtonDown(MouseButton button);
```

```
bool CEGUI::System::injectKeyDown(uint key_code);
```

When user input is captured in *frameStarted*, Duck's OIS listeners inject the input into CEGUI using these methods.

Callbacks are registered with GUI elements. This is similar to how OIS handles user input; user input that is injected into CEGUI is used to determine which callbacks to call based on which GUI element the user has interacted with. Duck currently uses one callback for handling mouse clicks on push buttons such as those on the main menu and "Edit Object Properties" dialog. This callback is called whenever a push button registered for this callback is clicked. The name of the GUI element is extracted at the beginning of the callback and an appropriate action is taken based on that name.

For example, if the user clicks on the "Create Sphere" push button on Duck's main menu, the push button's name "Construct/Sphere" is passed into the callback, and the following is performed:

```
// ...
if (senderID == "Construct/Sphere")
{
  SceneManager::getInstance()->createSphere(DEFAULT_MASS, DEFAULT_DIM);
}
// ...
```

3.6 Spatialized Audio

After handling user input, Duck updates the listener's position in the scene. This amounts to checking the current position and orientation of the camera and updating audio system with this data. The following is from the *frameStarted* method:

```
Ogre::Vector3 position = ogreFramework->m_pCamera->getPosition();
Ogre::Vector3 direction = ogreFramework->m_pCamera->getDirection();

soundManager->setListenerPosition(position.x, position.y, position.z);
soundManager->setListenerDirection(direction.x, direction.y, direction.z);
```

Duck uses Simple and Fast Multimedia Library (SFML) for handling audio device access and playing audio files. SFML provides spatialized audio through Open Audio Library (OpenAL). To spatialize audio with SFML, two steps must be made. First, a listener needs to be positioned someplace in a scene. Second, a sound must be associated with a position in the scene. Based on these two positions, the user can hear the sound play from a relative direction from his or her position in the scene.

To interface with OpenAL through SFML, SFML maintains a singleton that represents the listener in a 3D environment. The above *setListenerPosition* and *setListenerFunctions* simply call SFML's wrapper functions to indirectly access this part of OpenAL's spatialized audio functionality:

```
void SoundManager::setListenerPosition(float x, float y, float z)
{
  sf::Listener::SetPosition(x, y, z);
}

void SoundManager::setListenerDirection(float x, float y, float z)
{
  sf::Listener::SetTarget(x, y, z);
}
```

Another component of successfully spatializing audio through SFML is to set each instance of SFML's *sf::Sound* class to an appropriate position before playing. This is done through a call to

```
void sf::Sound::SetPosition (float x, float y, float z)
```

While SFML maintains only one listener, each *sf::Sound* instance can be given a unique position. This allows this listener to hear multiple sounds from all directions around him or her. While the listener's position is updated when *frameStarted* is called, sound positions are determined and set inside Duck's event system, which will be discussed in a future section.

3.7 Physics

After handling user input and updating the listener's location, Duck's *frameStarted* method updates the physics simulation. It does this by calling

```
physics->update(evt.timeSinceLastFrame);
```

where *evt* is a reference to an *Ogre::FrameEvent*. As mentioned previously, a *FrameEvent* maintains information regarding how much time has elapsed between events or frames. This is used here to instruct the physics simulation how far to advance the entities in the system through the simulation.

Duck's *Physics::update* method is as follows:

```
void Physics::update(btScalar timeStep)
{
  if (isActive)
    dynamicsWorld->stepSimulation(timeStep, MAX_SUBSTEPS);
}
```

Therefore, for each call to *frameStarted*, if the physics simulation is active, it updates (steps forward) from the previous frame's state. The following section describes the Duck's underlying physics engine.

3.7.1 Bullet Physics

Duck uses Bullet Physics for a physics engine. Bullet is an open source physics library written in C++. It provides rigid and soft body dynamics and collision detection.

In Bullet, a *btCollisionWorld* defines an interface and container for managing instances of an object capable of colliding, called a *btCollisionObject*. Duck uses an implementation of *btCollisionWorld* called *btDiscreteDynamicsWorld* for its rigid body simulations. Rigid bodies are represented by an implementation of a *btCollisionObject* called *btRigidBody*.

When a geometric object such as a sphere or cone is created in Duck, its representative *btRigidBody* is added to the *btDiscreteDynamicsWorld*:

```
// rigidBodyCI is a "Construction Info" object that encapsulates
// initial rigid body parameters such as mass, motion state,
// collision shape, and inertia.
rigidBody = new btRigidBody(rigidBodyCI);

// This adds the rigid body to the btDiscreteDynamicsWorld.
PhysicsManager::getInstance()->dynamicsWorld->addRigidBody(rigidBody);
```

When *dynamicsWorld*'s *stepSimulation* method is called, all registered *btRigidBody* instances are transformed based on the physics simulation. For example, a new position and velocity may be set on a *btRigidBody* based on its previous position and velocity and any accelerations registered with that *btDiscreteDynamicsWorld*.

When this transformation occurs, methods of the *btRigidBody*'s registered *btMotionState* are called. A *btMotionState* is an interface that Bullet provides to establish a relationship between physics simulation steps and rendering. Using this interface, Bullet can be set up to update a node in Ogre's scene graph every time a rigid body

is altered. Duck implements *btMotionState*'s *setWorldTransform* method to achieve this:

```
void OgreMotionState::setWorldTransform(const btTransform& newTransform)
{
  transform = newTransform;

  if (NULL == node)
  {
    return;
  }

  btQuaternion rot = transform.getRotation();
  btVector3 pos = transform.getOrigin();

  node->setOrientation(rot.w(), rot.x(), rot.y(), rot.z());
  node->setPosition(pos.x(), pos.y(), pos.z());
}
```

In the above code, node is a pointer to the node in Ogre's scene graph that represents this rigid body. Each geometric object in Duck is associated with a unique node in Ogre's scene graph in addition to its own *btRigidBody* during initialization. When the physics simulation steps, each affected *btRigidBody* calls *setWorldTransform*, which in turn updates the node in the scene graph. The upshot is that, when it comes to rendering a frame, Ogre's scene graph contains the updated positions of each affected geometric object.

Another important aspect of a *btRigidBody* is that it is also a *btCollisionShape*. A *btCollisionShape* is what Bullet uses for tracking and handling collisions between *btCollisionObject*s, and through inheritance, *btRigidBody*s as well. Bullet comes

with several *btCollisionShapes* defined, such a *btSphereShape*, *btCylinderShape*, *bt-ConeShape*, and *btBoxShape*. Duck uses these for its geometric spheres, cylinders, cones, and boxes, respectively.

Whenever a *btRigidBody* changes direction or orientation, its corresponding *bt-CollisionShape* is updated. While the *setWorldTransform* method above updates Ogre's scene graph when these changes occur, user initiated changes to geometric objects must propagate to the corresponding *btRigidBody* and *btCollisionShape* elsewhere. For example, if a user wants to manually reposition a sphere, the sphere's scene graph node must be updated for it to render in a new location. Additionally, its underlying *btCollisionShape* needs to be repositioned as well. If it isn't, then collisions could still occur where the shape was previously rendered. Changes to an scene node's orientation and size also need to propagate back to the object's *btCollisionShape*.

3.7.2 Collisions

Collision handling in Duck is handled through a callback registered with Bullet. This callback is internally called during simulation steps; each time this is called, each pair of *btCollisionShape*s in the *btDynamicsWorld* is iterated over through abstractions provided by Bullet. If a pair of these objects is eligible for a collision, then the contact points between the objects are inspected. If a new contact point between objects is discovered, then the objects are cast into Duck's geometric object representation and and all *CollisionTrigger*s associated with the objects are fired. The next section discusses Duck's event system, triggers, and events in more detail.

Figure 3.2: Duck's event system. Every geometric object in a scene has zero or more triggers associated with it; every trigger has zero or more events associated with it. When a trigger fires, all of its associated events are executed.

3.8 Event System

Duck uses an event system to coordinate and manage the results of actions and events in a scene. The two main components of the event system are triggers and events. Triggers are capable of firing an event; an event is a specific action that impacts the scene or produces audio or video when triggered. Geometric objects have triggers associated with them. Eventually, specific sections of an object's surface will hold their own set of triggers, which will allow more fine-grain interaction with an object.

Duck currently implements two triggers–a *NullTrigger* and a *CollisionTrigger*. The *NullTrigger* has no functionality and is useful as a placeholder for empty registers. The *CollisionTrigger* is fired on collisions detected by the physics system. An example of a trigger to be implemented in the future is a *ThresholdTrigger*. This will conditionally fire if an object has been displaced out of a specified threshold.

Additionally, Duck currently implements two events–a *NullEvent* and a *PlayAudioEvent*. Similar to the *NullTrigger*, the *NullEvent* has no functionality and is useful as a placeholder for empty triggers. The *PlayAudioEvent* plays an audio sample from disk when triggered. An example of an event to be implemented in the future is a *ChangePhysicsEvent*. This will change the physics properties of an object or scene when fired. For example, an object's mass, velocity, or restitution could be changed

by a *ChangePhysicsEvent*. Additionally, the scene's gravity could be changed when this event is fired.

The following lists the relevant public methods of the *Trigger* and *Event* interfaces.

```
Trigger:
void bang();
void bang(float x, float y, float z);
bool addEvent(int index, EventPtr event);
bool removeEvent(unsigned int index);

Event:
void execute();
```

New events should be inherited from the *Event* class. *Event*'s pure virtual method *run()* needs to be implemented. *Event*'s *execute()* method creates and starts a new *boost::thread* and binds the *run()* method to it, so when events are executed, they execute in their own thread of control.

3.9 Rendering

When Ogre renders a frame, it consults its internal scene graph to determine what to render. The nodes in the scene graph are updated indirectly before each frame when the *frameStarted* method updates the physics simulation. As mentioned previously, stepping the physics simulation updates Ogre's scene graph whenever an object in the scene has been affected by the physics simulation. Changes to position and orientation are examples of these changes that are reflected by the object's corresponding node in Ogre's scene graph.

A scene graph is an approach to structuring entities in a renderable scene. Ogre maintains a root scene graph node. Forming hierarchies of nodes is advantageous

because it allows groups of children nodes that share the same parent node to share similar characteristics and structures. For example, a simple automobile can be represented in a scene graph by a node that represents the automobile's body with children that represent the wheels. When the automobile node moves, the wheels move along with it. Scene graphs also provide opportunities for optimization. Specific branches of a scene graph can be rendered while other branches are left untouched.

3.10 frameEnded

The *frameEnded* callback updates the status the GUI by calling

```
gui->update();
```

This updates the status of any active and visible menus. If an object in the scene has been selected, then the selected object's name, mass, velocity, and restitution are updated with current values. Since this update occurs at the end of every frame, if the user selects and object and starts the physics simulation, the objects velocity is updated and displayed to the user in real-time.

Additionally, update will update the notification text of the main menu with the Duck's current state. For example, if the physics simulation is inactive, then the status text will read "Physics is disabled." If the user has activated the physics simulation and is recording the activity in the scene, the status text will read "(RECORDING) Physics is active."

3.11 Selections

A somewhat complicated procedure to handle is when the user selects an object in the scene. From the user's perspective, the procedure is rather simple. They move

the mouse cursor so that it is positioned over a visible portion of an object and click their mouse. Afterwards, the object visually appears as selected, and menus and dialogs are updated to reflect the newly selected object's properties.

However, selections are somewhat complicated to handle internally, as they permeate through most of Duck's subsystems. The following discussion details how a single selection relates to OIS, CEGUI, Bullet, and Ogre.

As each frame is processed, OIS checks its buffered input to see if the user has moved his or her mouse. If so, the new mouse position is fed into CEGUI so that when the GUI elements are rendered, the mouse cursor is in an appropriate position.

After the user has positioned their mouse cursor over a visible portion of the object that they'd like to select and clicks the mouse, the *mousePressed* listener checks to see if the user has clicked in the scene or on a GUI element, such as a menu or dialog. If the user clicked on a GUI element, then this wasn't a valid selection attempt; CEGUI will process the mouse click on its registered call-backs.

If the selection was a valid attempt, then the current mouse position is converted from CEGUI coordinates into normalized screen coordinates. A ray is created by calling a utility method that is provided by Ogre's *Camera* class; this method accepts normalized screen coordinates and returns a ray with origin at the camera's position that extends through the viewport at the specified location and into the scene's world space coordinates. The ray's terminal point is extended into the scene by a distance of *MAX_CAST_DIST*, which represents how far back an object can be from the camera and still be eligible for selection.

This modified ray is then passed to Bullet's ray casting mechanism. Bullet performs a ray test to find the nearest rigid body, if one exists, along the ray. If one is found, the object represented by that body is set to selected, and its node in Ogre's scene graph is set to show its bounding box so the user has a visual cue that the

Offset (ms)	*PAE*
123	1
456	2
789	1
112233	3

Figure 3.3: A typical SolutionTable. The "Offset (ms)" column indicates the number of milliseconds after the recording began that the PlayAudioEvent was executed. The "PAE" column indicates which PlayAudioEvent was stored in that TableEntry. Each row represents a TableEntry. Note that while PAEs are denoted by numeric values here, a TableEntry stores an actual copy of a PAE.

object has been selected.

3.12 Recording and Testing

While somewhat separate from Duck's main loop, the recording testing mechanisms are worthy of some discussion. Two classes make up this mechanism: *SolutionTable* and *TableEntry*. Recording and testing are composed of two broad operations. When recording, a *SolutionTable* is constructed. When testing, two *SolutionTable*s are compared for equality.

A *SolutionTable* is a container for *TableEntry*s. Each *TableEntry* is composed of the number of milliseconds after when the recording began and a copy of a *PlayAudioEvent*. Figure 3.3 shows a typical *SolutionTable*. Figure 3.4 shows the relationship between *SolutionTable*s and *TableEntry*s.

As mentioned above, recording corresponds to building a *SolutionTable*. When Duck is put into a recording state it initializes a *SolutionTable*. When a collision occurs, each colliding object's *PlayAudioEvent*s are copied into a *TableEntry*. This *TableEntry* will also contain the number of milliseconds that have passed since recording began–or in other words, the number of milliseconds since the table was created

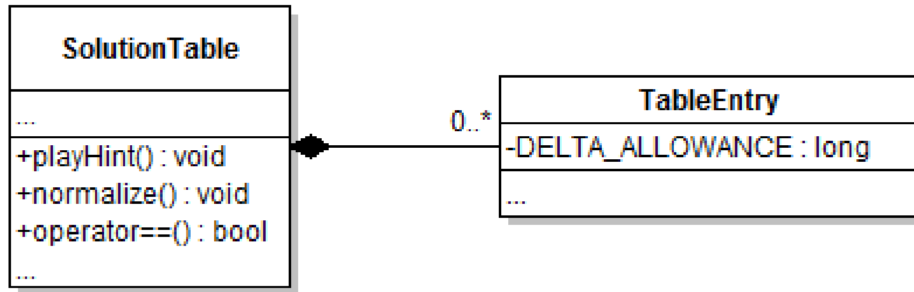

Figure 3.4: Relationship between SolutionTable and TableEntry. Methods and members not discussed in this section have been omitted from the diagram.

or initialized. Finally, this *TableEntry* is inserted into the *SolutionTable*.

Testing corresponds to comparing two tables for equality. Two *SolutionTable*s are equal if each *SolutionTable*'s *TableEntry*s are equal. Two conditions are required to be met for two *TableEntry*s to be equal. First, they must have matching *PlayAudioEvent*s. For example, a *PlayAudioEvent* for "bass.wav" would match another with "bass.wav", but not one with "snare.wav". Second, each *TableEntry* must occur within a timing threshold from one another.

TableEntry defines a constant called *DELTA_ALLOWANCE* that specifies the number of milliseconds that corresponding *PlayAudioEvent*s can differ in timing between two *TableEntry*s and still be equal. For example, given two *TableEntry*s, TE1 and TE2, if TE1 occurs at 1200 milliseconds and TE2 occurs at 1300 milliseconds, then a *DELTA_ALLOWANCE* of 100 would permit the two to be equal, while a *DELTA_ALLOWANCE* of 50 would not.

For hint-playback, a *SolutionTable*'s *TableEntry*s are traversed, and each stored *PlayAudioEvent* is executed at intervals that reflect their stored timings.

When building a table, timings for the first entry can vary significantly. Long delays between starting to record and entering the first entry are problematic for hint-playback and comparing tables. For hint-playback, a delay before the first entry

Un-Normalized Table	
Offset (ms)	*PAE*
123	1
456	2
789	1
112233	3

Normalized Table	
Offset (ms)	*PAE*
0	1
333	2
666	1
112110	3

Figure 3.5: A SolutionTable before and after normalization. Normalization only changes the offsets of the table's entries. The number of entries and the associated PlayAudioEvents are not affected or altered.

is entered would cause silence before the hint's audio plays through. This does not make for a very helpful hint. For comparing tables, the initial timings become more and more difficult to time correctly. In other words, if the user is trying to solve a scene, in addition to making sure that their objects trigger audio at the right timing intervals, he or she would also need to carefully match the silence in the hint-playback with a delay before they start recording.

To address these issues, a normalization mechanism was implemented. *SolutionTable::normalize()* shifts each of its *TableEntry* timings to be relative to the first entry shifted to a 0 ms offset. Figure 3.5 illustrates this.

When comparing two tables or playing back a hint, the tables in question are normalized first. Thus hint-playback begins immediately and comparing two tables is much less error-prone. A nice side-effect of the latter is that the *DELTA_ALLOWANCE* can be decreased in confidence. If *DELTA_ALLOWANCE* is too large, then solving a scene becomes too easy.

4 Results

Each test session consisted of three stages:

1. Introduction and "free play".
2. Solving scenes.
3. Answering brief questionnaire.

4.1 Introduction and "Free Play"

Each user was given a brief introduction the controls necessary to move around the scene and manipulate objects. An explanation of how objects can be configured to play audio when colliding with other objects was given, as well as instruction on how to start and stop the physics simulation.

After the brief introduction, each user was given five minutes of "free play", in which they had an opportunity to become accustomed to the controls and the necessary actions to produce audio in a scene.

4.2 Solving Scenes

After the introduction, the user's task was to "solve" three pre-loaded scenes. As mentioned in the previous chapter, loading one of the pre-loaded scenes consists of the screen blanking, the scene loading in the background, the physics simulation

activating, and the resulting audio being recorded for hint-playback. The scene was then manually cleared and the screen un-blanked.

The user was given five minutes to solve the scene. Solving entails designing a new scene from scratch, recording the sounds that it makes, and comparing that sound against the pre-loaded scene's recorded output. For testing, a *DELTA_ALLOWANCE* of 1500 milliseconds was used; that is, each of the user's audio events were permitted to be within 1500 milliseconds of the timing of the pre-loaded scene's corresponding audio events. If the user could solve the scene, then the number of attempts made and the approximate amount of time taken to solve it were recorded, and the user progressed to the next scene. If the user could not solve the scene in the alloted time, then the next scene was loaded.

Figures 4.1, 4.2, and 4.3 show the three pre-loaded scenes as they were programmed.

Table 4.1 shows the number of attempts made for each session, and 4.2 shows the average number of attempts for successful completions. Table 4.3 shows the approximate amount of time taken for each test session, and table 4.4 shows the average elapsed time for each successful completions.

	Number of Attempts		
Test Session	**Scene I**	**Scene II**	**Scene III**
1	2	4	-
2	1	1	-
3	1	-	-

Table 4.1: Number of attempts made to solve a scene during each test session. The above table shows the number of attempts made to solve each scene. If the user was unable to solve a scene within the time alloted, its column is marked with a '-'.

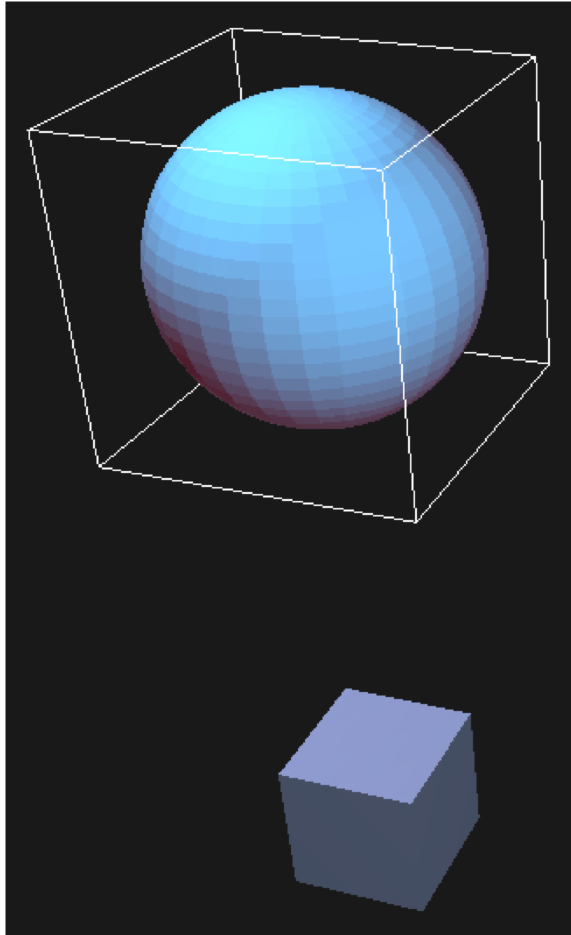

Figure 4.1: Scene I. The highlighted object is the only object being affected by gravity. The other objects are static–that is, they have zero mass and remain stationary.

4.3 Questionnaire

The following questionnaire was given after each test:

```
Please rate the following questions on a scale from 1 to 5, where
```

Scene I	Scene II	Scene III
1.3	2	-

Table 4.2: Average number of attempts for each completed scene. Averages are computed from users that were able to solve the scene. '-' denotes that no users solved the scene.

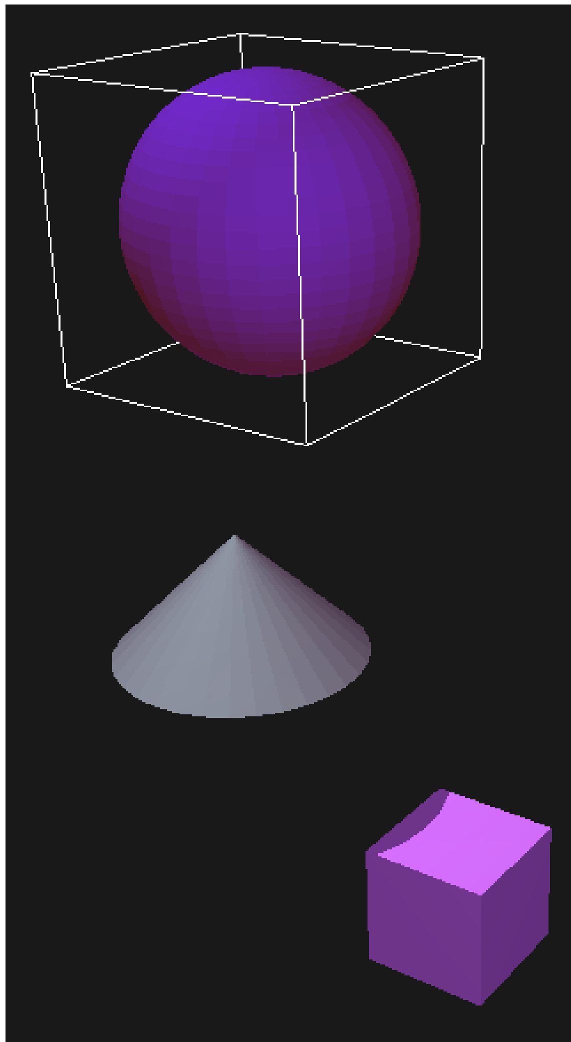

Figure 4.2: Scene II. The highlighted object is the only object being affected by gravity. The other objects are static–that is, they have zero mass and remain stationary.

```
5 - Completely agree

4 - Somewhat agree

3 - Neither agree nor disagree

2 - Somewhat disagree

1 - Completely disagree
```

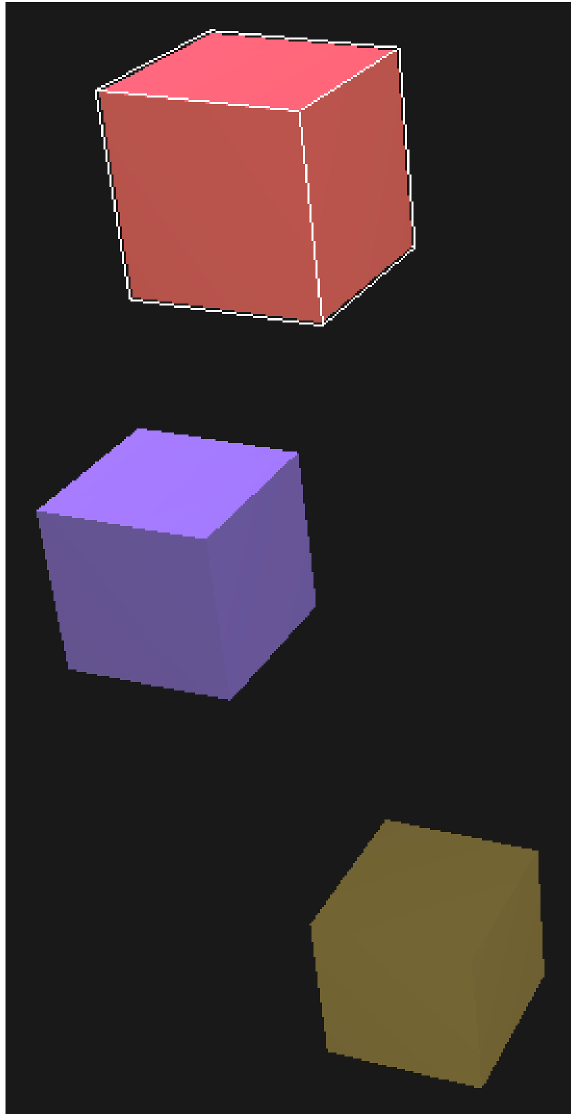

Figure 4.3: Scene III. The highlighted object is the only object being affected by gravity. The other objects are static–that is, they have zero mass and remain stationary.

```
1) The scene was easy to navigate.

2) It was easy to place and position objects.

3) This was enjoyable to use.

4) This software seems useful or interesting to use in other projects, applications,

   or domains.
```

Table 4.5 shows the results of the questionnaire for each session.

Table 4.6 shows the average rating for each question.

	Time Elapsed (Min:Sec)		
Test Session	**Scene I**	**Scene II**	**Scene III**
1	2:15	3:45	-
2	1:05	2:15	-
3	1:15	-	-

Table 4.3: Approximate amount of time to solve a scene during each test session. The above table shows approximately how much time was needed to solve each scene. Time is displayed in minutes and seconds. If the user was unable to solve a scene within the time alloted, its column is marked with a '-'.

Scene I	Scene II	Scene III
1:30	2:00	-

Table 4.4: Average elapsed time for each completed scene. Averages are computed from users that were able to solve the scene. '-' denotes that no users solved the scene.

Test Session	Ratings			
	Question 1	**Question 2**	**Question 3**	**Question 4**
1	4	4	5	5
2	4	2	4	5
3	4	4	5	4

Table 4.5: Questionnaire ratings. The above table shows the ratings given for the questionnaire for each test session.

Question 1	Question 2	Question 3	Question 4
4.0	3.3	4.7	4.7

Table 4.6: Questionnaire rating averages. The above table shows the average rating for each question in the questionnaire.

5 Future Work

There are two categories of future work: architecture and implementation.

First, the architecture described in chapter 2 still needs to be implemented. Initial attempts at developing the architecture focused on piecing the overall design together and providing networking support. This work was suspended in favor of working on a concrete implementation before implementing the architecture as a general framework.

Second, there are several possible improvements that can be made to the existing implementation.

- User interaction improvements: The current mouse and keyboard input methods seemed somewhat cumbersome during test sessions. Users suggested improvements such as auto-selecting an object after construction and auto-repeating object movement when a placement key is held.

- Support for other input/output methods: One such example would be adding support for tracking using a head mounted display. Scenes could be interacted with using hand-gestures. Physical objects could be tracked for more tactile interaction.

- Composite objects: Provide support for combining several objects into a compound object. One example of this would be the mallet shown in figure 2.3.

- Scaling: Provide support for scaling an object's size.

- Filtering support: Add support for special filtering objects. When an event is executed, its output or effect could be routed through a filtering object to alter its effect. For example, a *PlayAudioEvent* could be routed through a *ReverbFilter* that would play the audio sample with a reverb echo. Another example would be a filter to modify the physics settings, such as a filter that causes objects traveling through it to act as if submerged in water.

- Saving and exporting scenes, instruments, and filters: Provide a mechanism for storing creations and collaborating with other users.

- More triggers and events: Develop a *ThresholdTrigger* and *PhysicsEvent*. Consider developing a markup language to rapidly create and distribute more triggers and events.

- Registers ("Trigger banks"): Triggers could be associated with individual sides or vertices of objects instead of the entire object. For example, a cube could have a register for each side and each vertex, providing a total 14 possible points to attach triggers.

- Integrate with the architecture: Migrate generalizable components of the implementation into architectural slots. Methodically introduce a dependency between the architecture and implementation until the described client-server relationship is exposed.

- GUI improvements: The current GUI is somewhat cumbersome to use, customize, and maintain. While suitable for a prototype, it appears unprofessional. For a desktop application, this implementation might benefit from using a more traditional GUI, such as that offered by QT.

- Libraries: The libraries used by the current implementation could be scaled down. It may be advantageous to use OpenGL for graphics (removing the Ogre dependency), QT for windowing, GUI, and threading (removing dependencies on CEGUI and boost), and OpenAL for audio (removing the SFML dependency). However, as the implementation scales, some of the libraries currently in use may become more beneficial to use.

Bibliography

[1] Nicoletta Adamo-Villani, Edward Carpenter, and Laura Arns. An immersive virtual environment for learning sign language mathematics. In *ACM SIGGRAPH 2006 Educators program*, page 20, Boston, Massachusetts, 2006. ACM.

[2] Xavier Amatriain, Pau Arum, and Miguel Ramrez. CLAM, yet another library for audio and music processing? In *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 46–47, Seattle, Washington, 2002. ACM.

[3] Charlie Brown, Gautam Babbar, Kelly Thornton, and Mike Booth. Left 4 dead 1 in-game developer commentary. This information is available from either the ”Developer Commentary” game mode or the L4D SDK resource packages., 2008.

[4] Jose Fornari. An essynth implementation in PD. In *ACM SIGGRAPH 2006 Research posters*, page 106, Boston, Massachusetts, 2006. ACM.

[5] Mark Foskey, Miguel A. Otaduy, and Ming C. Lin. ArtNova: touch-enabled 3D model design. In *ACM SIGGRAPH 2005 Courses*, page 188, Los Angeles, California, 2005. ACM.

[6] Anthony Hornof, Anna Cavender, and Rob Hoselton. EyeDraw: a system for drawing pictures with the eyes. In *CHI '04 extended abstracts on Human factors in computing systems*, pages 1251–1254, Vienna, Austria, 2004. ACM.

[7] Jari Kleimola. Design and implementation of a software sound synthesizer. Master's thesis, Helsinki University of Technology, 2005.

[8] Stefan Marks, John Windsor, and Burkhard Wnsche. Evaluation of game engines for simulated surgical training. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 273–280, Perth, Australia, 2007. ACM.

[9] Renaud Ott, Mario Gutirrez, Daniel Thalmann, and Frdric Vexo. Advanced virtual reality technologies for surveillance and security applications. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 163–170, Hong Kong, China, 2006. ACM.

[10] Michal Ponder, Bruno Herbelin, Tom Molet, Sebastien Schertenlieb, Branislav Ulicny, George Papagiannakis, Nadia Magnenat-Thalmann, and Daniel Thalmann. Immersive VR decision training: telling interactive stories featuring advanced virtual human simulation technologies. In *Proceedings of the workshop on Virtual environments 2003*, pages 97–106, Zurich, Switzerland, 2003. ACM.

[11] Simon Prince, Adrian David Cheok, Farzam Farbiz, Todd Williamson, Nik Johnson, Mark Billinghurst, and Hirokazu Kato. 3-D live: real time interaction for mixed reality. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 364–371, New Orleans, Louisiana, USA, 2002. ACM.

[12] Petri Purho. Crayon physics deluxe. http://crayonphysics.com/.

[13] Rongdong Yu, Xiuzi Ye, Sanyuan Zhang, Yin Zhang, Yiyu Cai, Jianmin Zheng, Wenyu Chen, Patricia Chiang, and Mon Hnin Tun. An architecture of a VR simulation system for cardiac intervention. In *Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, pages 1–2, Singapore, 2008. ACM.