Geometry Middleware for Audio Engines in Games with Dynamic Worlds

Stefan Irndorfer

M A S T E R A R B E I T

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2014

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 19, 2014

Stefan Irndorfer

Contents

Kurzfassung

Diese Arbeit beschreibt ein System (Dynamic Geometry Middleware), das eine Audio-Engine in die Lage versetzt, mit einer dynamisch, also zur Laufzeit generierten und veränderten/zerstörten Spielwelt-Geometrie, umzugehen. Die Audio-Engine soll trotz dieses Setups umgebungsabhängige Effekte wie Hall und Echos sowie das korrekte "'Panning"' und Filtern von Sounds bieten können.

Dabei wird davon ausgegangen, dass eine Audio-Engine bereits vorhanden ist, die diese Features grundsätzlich mitbringt. Es wird beschrieben, welche Schnittstellen zur Audio-Engine bzw. der Spiel-Logik vonnöten sind und wie die Daten-Strukturen bezüglich der Spielwelt-Geometrie und der dazu verwendeten "'Baumaterialien"' beschaffen sein müssen, um mit den beschriebenen Algorithmen zu funtionieren.

Weiter wird darauf eingegangen welche sinnvollen Abstriche im Vergleich zur realen akustischen Physik gemacht werden können um bei akzeptablen Laufzeitgeschwindigkeiten maximalen Realismus zu erzielen.

Abstract

An algorithmic solution (Dynamic Geometry Middleware) is presented, which enables a game audio engine to handle a dynamically created game world geometry and changes to this geometry during runtime. The audio-engine should be enabled to provide environmental effects such as reverberation and delay-effects, attenuation over distance and sound-occlusion besides a surrounding sound panning in such a dynamic game geometry. Furthermore the requirements to this audio-engine (which is encapsulated from the game-logic-system) and necessary deductions to the real worlds acoustics are discussed.

Chapter 1

Introduction

Within the last 15 years video game industry increased the amount of attention on game sound. Reasons for that may be the enhanced computing power their clients have at disposal, the wide spread of superior (surround) sound systems which are in use to play video games, the knowledge about the benefits of a "good game sound" related to immersion and realism. Therefore a wide range of software-tools and dedicated specialists are around to satisfy the markets and the gamers expectations.

What means "good sound" in a technical manner besides the usage of well produced sound samples and unique musical compositions? It is the meaningful utilisation of the correct panning, the usage of distance-effects (filtering, attenuation) and the consideration of occlusion and obstruction (the absence of a direct path from source to listener). Furthermore "environmental effects" (reverb and delay) must be parameterised properly according to the game worlds geometry and its material composition (e.g. a storage building, a small living-room, a cave etc.).

A typical approach in game audio programming is to define such sound areas within a well known game world and assign a proper reverberation effect on each sound source which is located in this area. But within the last years so called open world games became increasingly popular which means, that the game world geometry may be created dynamically, during runtime and cannot be foreseen by the audio programmer.

1.1 Concept of this thesis

This theses describes an audio system which copes with the challenges of a dynamically created game world geometry and furthermore changes to this system like destroyed walls. It describes the features of a state-of-the-art game audio system and their relation to the game worlds geometry. A major part will cover techniques which assure that the present geometry model is kept up to date in spite of changes to this geometry during runtime.

The presence of a basic sound-engine is assumed, and the requirements to this systems data structures and its interface are described. This basic sound-engine uses OpenAL + EFX (Effects Extension). When it comes to the description of common features of audio engines, OpenAL + EFX will be used as reference.

The major part of this work will cover an algorithmic solution for the sound system to keep an approximation of the current game world by organising its geometry using a tree-structure. The occurring examples will work on a 2D game world, but the described techniques could be expanded for a 3D solution as well.

The algorithmic examples refer to Java implementations and explanations about structural matters are done by an object orientated point of view.

1.2 Motivation

Due to the increasing popularity of open world games and other games, that build their game world dynamically during runtime (e.g. "Minecraft[1]") there is a strong demand for a system, that can find the right acoustical settings for this game world geometry without human intervention.

Another field of application would be games, which provide user level editors. It is not usual to put the strain of defining audio parameters to the user, so a system to automatise this procedure has to be present.

The related programming work is done in Java utilising OpenAL Soft[2] and its effects extension EFX. Each of these technologies are working cross-platform. OpenAL Soft and EFX are LGPL-licensed[3] which means a working solution could be provided to the public.

[1] https://minecraft.net/

[2] http://kcat.strangesoft.net/openal.html

[3] See [12]) for more information about LGPL.

Chapter 2

State of the Art

2.1 Current Situation on the Market

Gameplay bases on three components: Graphics, behaviour (physics and AI) and sound. During the 90s e.g. there were huge improvements in the first two components developed while sound was left kind of behind. The CPU was strongly limited and the game-sound had to get along with left overs. *In contrast, sound generation and propagation has not received as much attention due to the extremely high computational cost of simulating realistic sounds [9].*

When searching for up to date literature concerning the generation of environmental and acoustic data in game programming, one will quickly figure out, that most of the relevant papers/books were written until the mid 2000s. Accordingly to various online discussions at this time the computational power of normal PCs has reached a level that made it unnecessary to buy dedicated audio hardware and the marked started to shrink.

3D-Sound technologies such as audio wave tracing are used nowadays in games where audio has been payed enough attention. Games with very sophisticated soundscapes (e.g. Battlefield 4, Crackdown) usually use their own home brewed systems. There is no sound engine around, that can be used with Java directly and provides the possibility to define materials with their acoustic parameters and wraps the functionality of OpenAL into a tidy interface at the same time.

2.1.1 Audio APIs

This section lists the most important currently available audio APIs and gives an overview of their features.

OpenAL (Open Audio Library)

OpenAL is an interface to the audio hardware. In order to produce high-quality audio output, specifically multichannel output of 3D arrangements of sound sources around a listener, the interface consists of numerous functions [11]. OpenAL is designed to be cross-platform and is meant to be an audio equivalent to OpenGL. The corresponding code-base to this thesis is done by using OpenALs Java binding via the LWJGL[1], which allows to deploy software for Windows, Linux and Mac OS.

OpenAL Soft is an LGPL-licenced [12] software implementation of the OpenAL 3D audio API. It is a free alternative to the proprietary OpenAL. OpenAL Soft provides the capabilities for playing audio in a virtual 3D environment. Distance attenuation, doppler shift, and directional sound emitters are among the features handled by the API. More advanced effects, including air absorption, occlusion, and environmental reverb, are available through the EFX extension. It also facilitates streaming audio, multi-channel buffers, and audio capture [13].

The frequency filtering capabilities of OpenAL Soft are rather coarse grained. There is a low-pass filter and a high-pass filter only.

XAudio2

XAudio2 is a low-level audio API by Microsoft. It provides a signal processing and mixing foundation for games that is similar to its predecessors, DirectSound and XAudio [14].

Moreover it provides the possibility to place sounds into a 3D environment as well as a set of filtering operations (high/low/band-pass filters), environmental effects (e.g. reverb) and occlusion/obstruction filtering. See section 2.2 for more information about these features.

Web Audio API

The “Web Audio API” is specified as a high-level JavaScript API for processing and synthesising audio in web applications.

> The primary paradigm is of an audio routing graph, where a number of AudioNode objects are connected together to define the overall audio rendering. The actual processing will primarily take place in the underlying implementation (typically optimised Assembly/C/C++ code), but direct JavaScript processing and synthesis is also supported [16].

It supports filters (high pass, low pass, band pass, high shelf, low shelf, peaking and notch), sound synthesis (OscillatorNode). Additionally a set of

[1] Light weight Java game library (www.lwjgl.org).

effect nodes are available (Convolver Node, which is basically a reverb effect, DelayNode, DynamicCompressorNode, GainNode etc.).

2.1.2 Audio Middleware

The term "Audio Middleware" is a generalisation of software, which links the game logic (or the games objects) to the audio system.

> Middleware is software that connects game developers with the hardware ... they use in development. Just as Pro Tools lets you generate sound from a computer, middleware lets users links sounds to game objects etc. [2].

Audio Middleware can be seen as an abstraction of the underlying Audio APIs functionality.

The currently most popular and powerful audio middleware systems are Audiokinetics **Wwise**[2] and **Fmod**[3]. Both ship along with a graphical interface similar to a DAW (Digital Audio Workstation). They link game events and game entities with audio events and encapsulate audio designers work from coding. Both are capable to deploy cross-platform output.

An automated handling of a dynamic changing to the game world is missing at both systems. A dedicated software layer is necessary with both systems.

2.2 Features

This section is a summary of features which are expected from a state of the art game audio system. Since OpenAL Soft and its effects extension EFX is used in the corresponding code base some explanations are done referring to OpenALs specification and its restrictions concerning OpenAL Soft.

A goal of this section is to point out how important the presence of a geometry model is, when it comes to the automation of these features parameterisation.

2.2.1 3D Localisation

This term basically means that the audio output of a virtual source is properly panned accordingly to the appearance of this source in the games scene. The so achieved spacial hearing experience depends on the hardware and the speakers setup. More sophisticated technologies take the HRTF (Head Related Transfer Function) into account which simulates the configuration of humans physiology like phase differences caused by the location of both ears.

[2] https://www.audiokinetic.com/products/wwise/
[3] http://www.fmod.org/

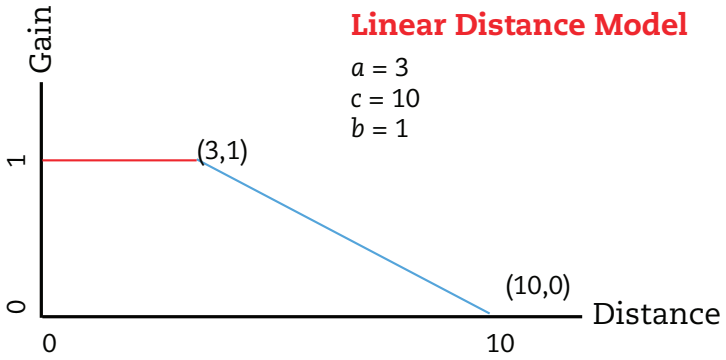


Figure 2.1: Sounds from sources within the radius of the reference distance a will be clamped to $g = 1.0$ while sounds from beyond the maximum distance c would not be heard at all. In between the gain value is interpolated linearly (rolloff factor b).

2.2.2 Distance Effects

Air Absorption

Air Absorption models the filtering of high frequencies over increasing distances between source and listener. This factor may be altered to simulate sound propagation in different media (e.g. under water). OpenAL-EFX already provides a function to set an air absorption factor.

Attenuation Over Distance

OpenAL comes along with various "distance models". The most reasonable and common in top down 2D-games is the "clamped linear distance model". It can be calculated like shown in the equations 2.1, 2.2 and 2.3, where d is the current distance, g is the gain, a is the reference distance, b is the rolloff factor and c is the maximum distance (See [6] for more details.). The equations

$$d_1 = max(d, a), \tag{2.1}$$

$$d_2 = min(d_1, c), \tag{2.2}$$

$$g = (1 - b \cdot \frac{d_2 - a}{c - a}) \tag{2.3}$$

determine the gain of a source in relation to its distance to the listener. Its behaviour can be seen in figure 2.1.

2.2.3 Environmental Effects

Although real-world sound propagation and environmental effects are quite complicated and have many contributing factors, by far, the two greatest

determining factors in overall sound characteristics are *environmental geometry* and *material composition* [1].

Environmental effects mean the natural phenomenons of *reverberations* and *echoes*. These additions to the original sound gives the listener orientation and information about the environment he is currently in (e.g. a cave, a stadium, a small living room etc.). This is caused by sound waves which bounce off of various structural components before they can reach the listeners ear.

Other then in real worlds physics—where sound propagates in concentric and continuous waves—in digital applications sound has to be discretised. In other words what is considered a circularly spreading wave in real world acoustics can be modelled as set of rays circularly arranged from the same origins.

A first intention, when it comes to the work on a sound engine, may be the achievement of a maximal level of realism. In non real-time applications which neglect computational complexity, environmental data are gained by using an intensive utilisation of ray cast operations. They are used for example in 3D walk-around software which may be applied in architectural previsualisation for example. In the paper *A beam tracing method for interactive architectural acoustics* a system which uses beam tracing data structures and algorithms to compute propagational paths from a static source to a moving listener is introduced. This approach needs at least a preprocession to work in real time [5]. Other than in many cases of computer game development computational cost is not an important issue since these calculations can be done offline.

Considering that the major chunk of the available computational power in a typical game-cycle is dedicated to graphics and logic, the audio procession has to get along with "the rest". That means, that expensive ray cast operations in order to compute a maximal number of *early reflections* and *late reflections* (see section 2.2.3) must be reduced to a minimum while a good parameterised utilisation of *delay* and *reverb effects* could lead to a satisfying result as well.

Sound waves in the digital domain are considered a set of rays. This section describes the relevant occurrences of sound reception in a game world in order to achieve a realistic impression.

Direct Signal

The direct signal is perceived, when a sound reaches the listeners ears without any interferences. The signal will be affected by *distance effects* only (see section 2.2.2).

In real wold physics this sound would be received delayed accordingly to the distance, the sound waves have to travel. In my experience in many computer games audio this fact is ignored. In some cases a delayed sound

would be considered rather a bug than realism.

Early reflections

Early reflections (or *first-order-reflections*) correspond to what audio producers know as *echoe-* or *delay-effects*. The used terminology leans against the parameters of common reverb and delay effects, like *Apples Logic Pro´s* reverb effect *PlatinumVerb*.

As shown in figure 2.2 an early reflections sound path is bounced off of an geometrical obstacle once before it hits the listener. The oncoming signals strength and its frequency composition is determined by the material, the geometric obstacle is made of and the total length of this path (compared to the direct signal). So the absorption behaviour of the material causes filtering operations which are added to the *distance effects*.

Pre-Delay

The pre-delay is the time between the reception of the *direct signal* and an *early reflection* (see figure 2.3). Its value is determined by the speed the sound propagates in the current medium. A default assumption may be dry air at 20°C which gives a speed of sound of 343 metres per second. At this point may be mentioned, that a good architecture should provide a global unit of distances.

Late reflections

Late reflections are higher order reflections and in general perceived as a diffuse "tail of sound" (see figures 2.2 and 2.3). Therefore a *reverb* effect comes into play. Its parameterisation depends on the attributes of the room, the listener is currently in. The rooms' shape, its material composition and the listeners position within this environment should be considered for proper settings.

Initial Delay

This parameter sets the amount of time between the direct signal and the beginning of the diffuse sound tail. It is normally greater than the *early reflections* pre-delay [15].

Decay Time

Also known as the *Reverb Time* sets the length of the reverb tail. Big rooms cause longer decay times than smaller ones do. Of course, the material composition of this room has to be taken into account as well.

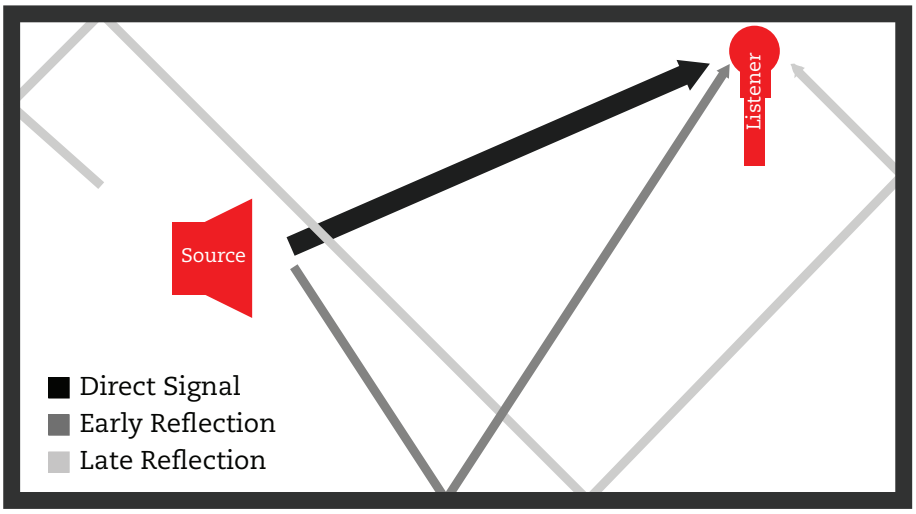


Figure 2.2: Representative paths of a direct signal, an early reflection and a higher order reflection.

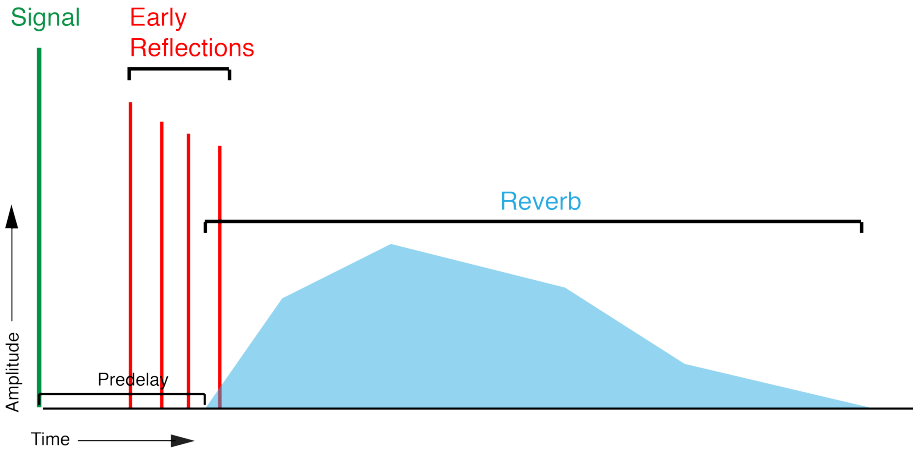


Figure 2.3: Occurrence of a signal with its environmental effects over time (x-axis).

2.2.4 Occluded/Obstructed Sounds

Occlusion or obstruction are given by the absence of a *direct path* from source to listener where occlusion can be described as a total obstruction. OpenAL "Effects Extensions Guide" [6] suggest, that only a sound which is in the same acoustic room, than the listener is can possibly obstructed. If a source is located in a different environment its occlusion related filtering would mask any obstruction-filtering anyway. This approach bares optimisation potential though it is not a strictly true assumption in the real world (See

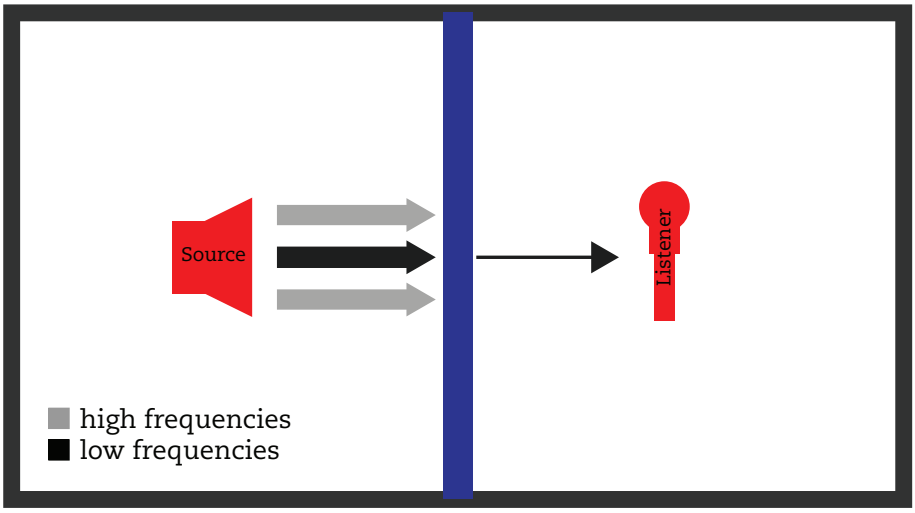


Figure 2.4: The occluding geometries materials attributes determine the attenuation of the signals high-frequencies and low-frequencies.

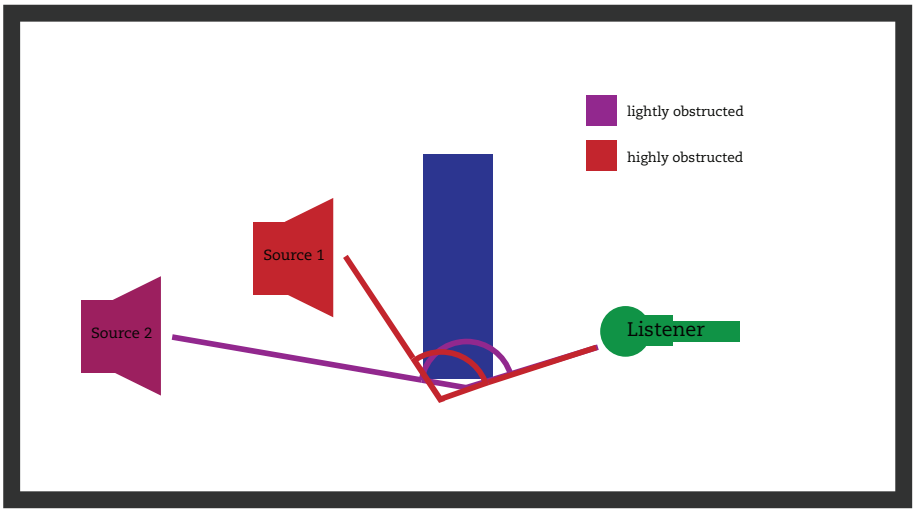


Figure 2.5: The obstructive effect is determined by the included angle in the path between source and listener.

figures 2.4 and 2.5 for examples.).

Storing potential obstacles bounding boxes will increase efficiency greatly, but at the expense of some accuracy [6]. The presence of an obstacle can be determined by checking whether the direct line between source and listener intersects a bounding box.

2.2.5 Material Composition

The level of sophistication in materials behaviour is determined by the filtering capabilities of the underlying audio API. In case of OpenAL Soft filtering operations can be processed either on low frequencies (LF-Reference = 250 Hz) or high frequencies (HF-Reference = 5000 kHz). So a materials attributes consist of its reflectivity (LF and HF) and its absorption (LF and HF). A linear absorbing behaviour of materials like concrete can be recommended as an efficient solution. Though this approach differs from real world acoustics it satisfies the demands of games audio.

2.2.6 Apertures

To work with multiple acoustic environments a concept of apertures is required. In this context apertures are areas of wall segments which divide two acoustic environments and are permeable for audio signals. A very obvious example would be a clear-air opening like a doorway or an open window between two rooms. But furthermore closed doors and walls which are accordingly to their thickness and material composition not total occluding for a sound signal are considered to be apertures. So apertures are needed to pan sounds from adjacent rooms in a right way considering the game worlds geometry and its material composition.

Figure 2.6 illustrates the placement of apertures in a typical 2D game world scenario. As one can see in the image, apertures are basically lines, while the placement of a source needs a point coordinate only. In practical use the source will be placed on the apertures point nearest to the listener.

Regarding a dynamical change to the geometry a recalculation of the affected rooms apertures has to be performed. Apertures must be considered dynamic and are heavily affected by the users interaction (destroyed walls, doors etc.). For calculations concerning the placement and updating of apertures see 4.6.

2.2.7 Channel Strip Model

The priorly described features will result in a multichannel output (e.g. stereo/surround) which basically utilises panning, filtering operations and environmental effects (reverb and delay). Their setup in the context of a channel strip model can be seen in figure 2.7. Since the environmental effects result in additional signals to the original sources signal, a duplication of the direct signal is sent to “Auxiliary Send Channels”.

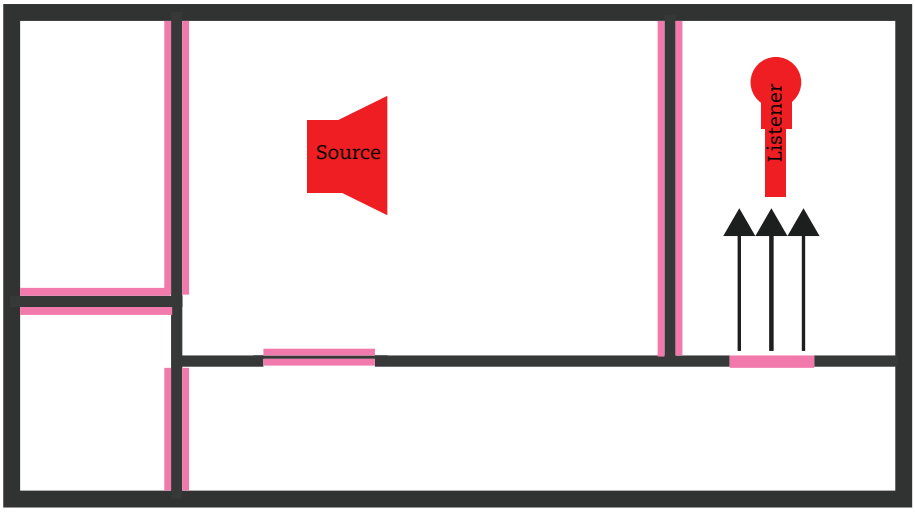


Figure 2.6: The pink structures mark the apertures—all areas on the walls, which are permeable for a sound signal. Noises from adjacent environments have to be perceived from these apertures.

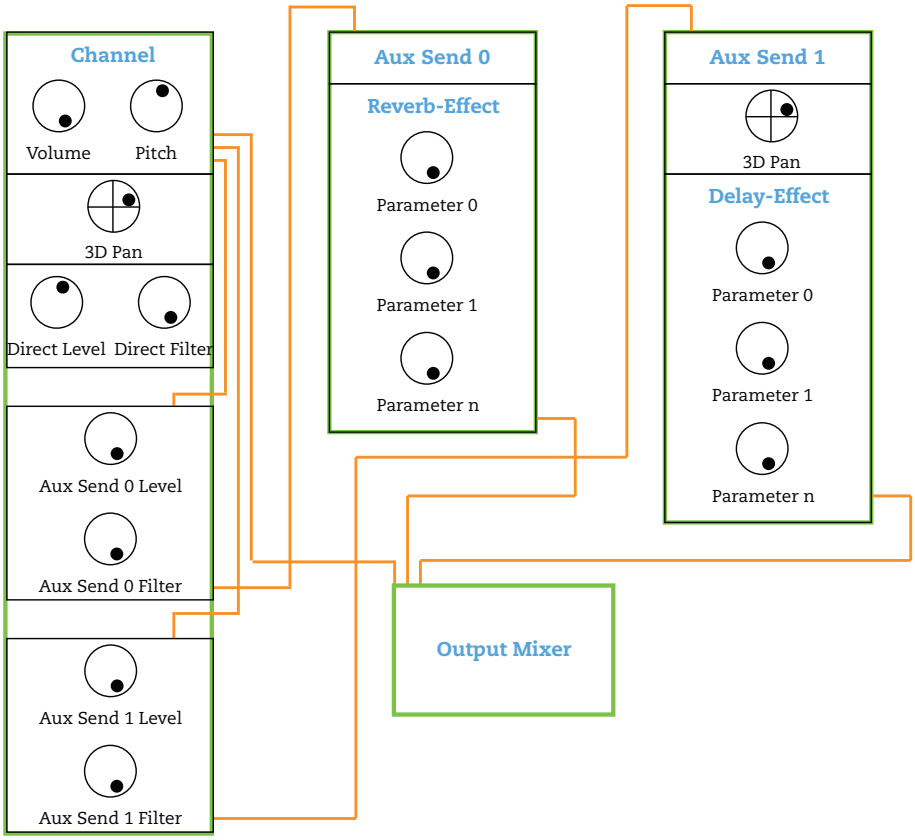


Figure 2.7: A channel strip model of the audio output as known by hardware mixers or DAW-Software (Digital Audio Workstations) is presented in the image. The direct signal passes through the "Channel" where its panning and distance effects (filter) takes place. The outgoing signal is split and sent to "Aux Send 0" and "Aux Send 1" where the environmental effects are interconnected.

Chapter 3

The Dynamic Geometry Middleware

The main task of the Dynamic Geometry Middleware (DGM) is to keep the game worlds geometry model up to date so that the sound engine maintains its capability to provide the features as described in chapter 2 in a dynamically changing and/or a dynamically generated game world.

3.1 Integration Into a Games Architecture

The layer diagram in figure 3.1 illustrates the integration of the DGM in a typical game architecture.

The listing below mentions the Dynamic Geometry Middlewares tasks:

- Passing play-, pause-, stopping requests from the game logic towards the sound engine.
- Keep game worlds geometry model consisting of wall/obstacle-segments, rooms/regions and apertures.
- Update the geometry model in case of a changing. This process has to be invoked by the game logic utilising the DGMs interface.
- Automatically parameterisation of environmental effects regarding to a rooms geometrical and material set up and the listeners and sources positions. The parameterisation of environmental effects settings is not covered by this work. The presence of an up to date geometrical model can be seen as a fundamental prerequisite for such a task though.
- Providing an interface which enables the game logics programmers to announce creational and destructive events. And furthermore define materials and specify their acoustic behaviours in terms of reflectivity and absorption.

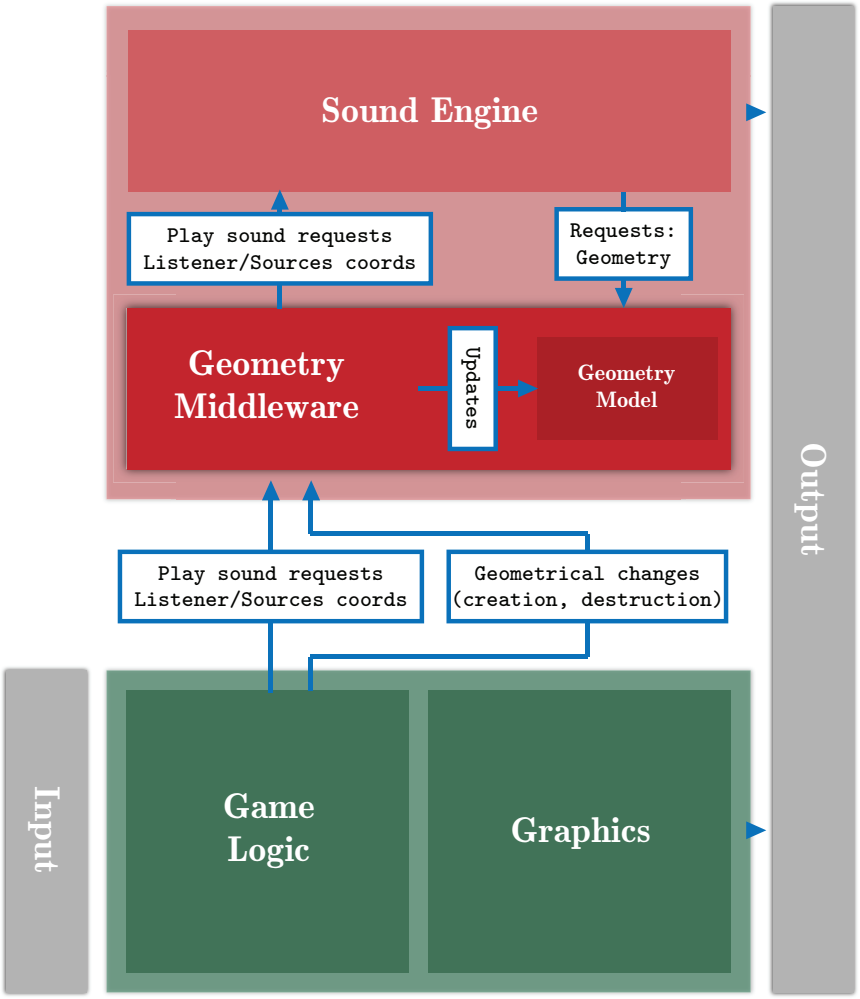


Figure 3.1: This Layer Diagram shows the role of the DGM in the context of a game engine architecture. The red areas mark the domain of the games audio procession while the game engines core is shown in the green areas. The output section on the right represents audio and graphics output. "Play sound request" stands for play, stop and pause requests. A closer explanation of the geometry model itself and its updating algorithms take place in section 4.

3.2 Representation of Materials

Each wall in the game world has to consist of a material. In this data-object are reflectivity and occlusion values (e.g. for high and low frequencies) recorded. The algorithms described in section 4 will work under the assumption that all rooms in this game world are closed structures. This required a material "Air", which is used in clear-air openings (see section

2.2.6) like open doors or windows and has no reflective or absorptive impacts. Additionally a discussion how to decide whether an opening between two rooms is a door or a short corridor can be omitted this way.

The "Material" class provides the method to compute the remaining gain after a signal has travelled through one distance unit of this material. Even though this is a total denial of real worlds physics it is absolutely adequate for game audio to make this fall off linear over distance.

3.3 Representation of the Geometry

In order to perform efficient collision detection geometrical objects are represented as rectangles with its length and width, its position coordinates and its material. A frequently performed task for a game sound system in general is to check for intersections of propagational paths with bounding boxes. To keep the performance of the system in acceptable dimensions an axis aligned geometry is recommended. Though rotated geometry elements could be handled as well this thesis covers the procession of axis aligned geometry only.

3.4 Representation of Environments/Regions

In the following sections rooms are also called regions. In game audio a room is considered to be an acoustic environment with its own reverberant parameters. These parameters can be derived from the rooms material composition (reflectivity), its size and shape.

Concerning the reflective quality of a room, its adjoining walls material composition have to be known. For the reverbs parameterisation the average of theses walls material attributes is utilised. *By controlling reverb parameters such as room size, reverb level, reflections level, air absorption, reverberation decay time etc, a sound designer can accurately simulate many different types of environment [8].*

The "Region" object keeps connectivity data which store occlusion values and aperture coordinates to adjacent rooms. In order to provide the features of occlusion in combination with apertures (see 2.2) all acoustic environments are organised in a graph where each room is represented by a node and the occlusion values are represented by the edges.

Figures 3.2(a) and (b) show a game world which is resolved in this fashion. It remains a simple tree search task to determine what parts of a signal and from which direction could a listener in room "B" perceive a sound that occurs in environment "D".

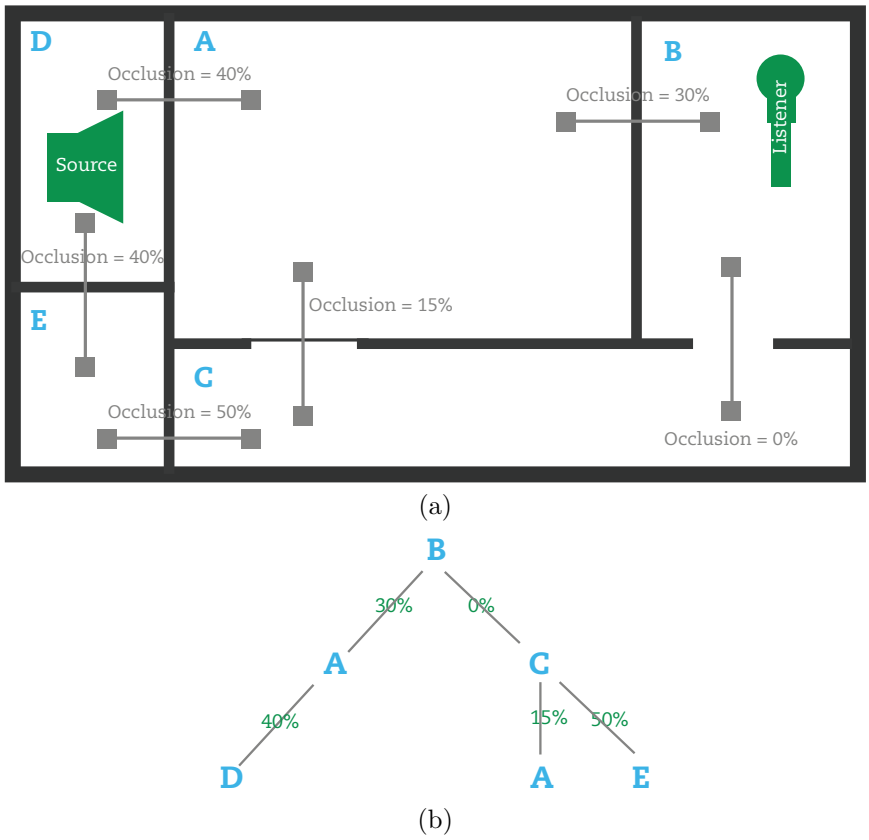


Figure 3.2: Figure (a) shows a typical 2D-Game scenario. We have serval rooms which are connected by apertures with different occlusion values. The representation of this game world as a tree-like structure will make it simple to determine what the listener in B can hear from the adjacent rooms. Note, that the tree in (b) contains the node A twice. The algorithm will decide to render the path with less occlusion and ignore other occurrences of this node.

3.4.1 The PAS (Potential Audible Set)

Since the attenuation over distance as described in 2.2.2 the acoustic scope of each room has a limitation. The PAS of a room consists of its neighbouring environments within the radius of the `max distance` variable. The rooms´ center could be taken as the PAS's center as a very simple solution.

As an additional optimisation each room node stores a reference to the neighbouring rooms within the PAS only, since sounds from outside the PAS can be ignored at all.

Chapter 4

Implementations

4.1 Quad-Tree

In order to keep the sound system encapsulated from the games logic it has to preserve its own data to describe rooms and separations of these rooms. Therefore a change of the worlds geometry (creation, destruction etc.) must entail a measuring of these changes and further more a merge with the so far known data. These data will be needed to localise sound sources and the sound listener within the game world. In other words the data structure has to be capable to describe polygonal shapes and be proper for efficient collision detection.

K-d trees are k-dimensional trees. They are used to organise k-dimensional data. K-d trees in the 2D-domain are called **quad-trees** while **oct-trees** are used for 3-dimensional data. In this section a 2-dimensional game world example is discussed. In order to get the described algorithms working on 3-dimensional data the algorithms must be extended properly.

Quad-trees can be seen as axis aligned hierarchical partitionings of 2-dimensional data.

> A quad-tree is a tree whose nodes are either leaves or have four children [7].

The root node includes the whole game world and each parent node has 4 children. Until the fulfilment of a stop criteria each parent node is divided into 4 child nodes in recursive fashion. Typical criteria for stopping the recursive creation of sub nodes include the tree reaching a maximum depth or the child nodes getting smaller than an arbitrary set minimum size.

4.1.1 The QuadTreeNode Object

In the following algorithmic descriptions `QuadTreeNode` objects are assumed to be composed like

$$n \leftarrow < f, s, m, r, t, w, x, y > .$$

The following lists the data a `QuadTreeNode` has to record:

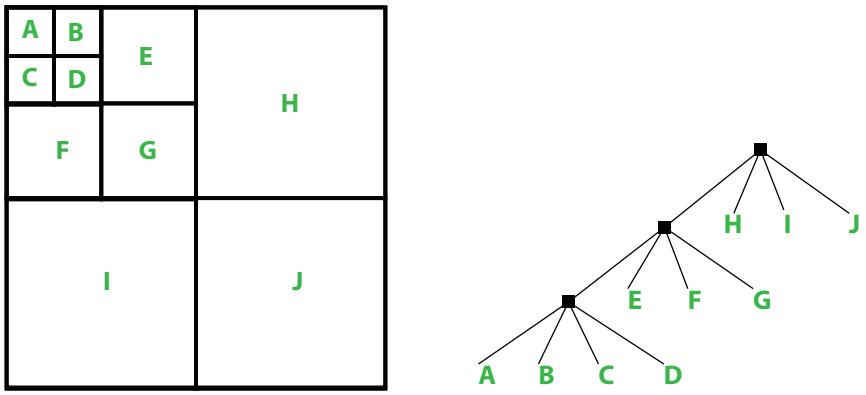
- f: `QuadTreeNode father`: The denotation of the parent element as "father" is adopted from [10].
- s: `QuadTreeNode[] sons`: This denotation is also transferred from [10]. The presence of parent and child elements is a basic requirement to work as a tree structure. Furthermore if a request for a QuadTreeNodes child element returns `null`, this element is a leaf.
- m: `Material material` determines if a leaf belongs to a wall or a room. The absence of an assigned material member indicates that a leaf belongs to a room.
- r: `Region label`: If the request for the label of a QuadTreeNode returns `null`, the element is currently not assigned to a region.
- t: `short sonType`: Indexes if this element is a northwestern, northeastern, southwestern or southeastern quadrant of its `father` (see 4.6).
- w, x, y: (double width, double x, double y).

4.1.2 Creation of a Quad-Tree

In the following the child nodes are denoted as quadrants or sons. Figure 4.1 shows a simple example of a quad-tree and its corresponding graph (see [4]).

Algorithm 4.1 shows the recursive computation of a quad-tree where leafs store weather they belong to the inner of a room or to a wall segment. The `QuadTreeNode` objects are initialised with its material variable `material = null`, which means it is considered to be a room leaf by default. During the creation process a material is assigned if the specific node is identified as a wall leaf. This distinction will be needed when it comes to an interpretation of the quad-trees data (see section 4.3). The accuracy of the result and the runtime complexity are defined by ϵ and m and the general decision, which collision detection technique should be used (see section 4.2). Where m is the minimum size of leafs which means it is the stopping criteria for this algorithm. The value of m is arbitrary set and dictates the balance between resolution of the quad-tree and runtime performance.

As shown in figures 4.2 this data structure can be used to approximate any polygonal shape. As mentioned above the quad trees components are axis-aligned. In this specific example the wall segments are axis-aligned as well which is of course not compulsory. Inside the method `yieldNodeType`



(a) (b)

Figure 4.1: A simple example of a quad-tree. Note, that the maximal number of searching iterations per tree level is 4 which leads to very efficient behaviour when it comes e.g. to the determination of the listeners position.

a collision detection algorithm is called. See section 4.2 for different box intersection and collision detection methods and section 4.7 for their impact on the performance.

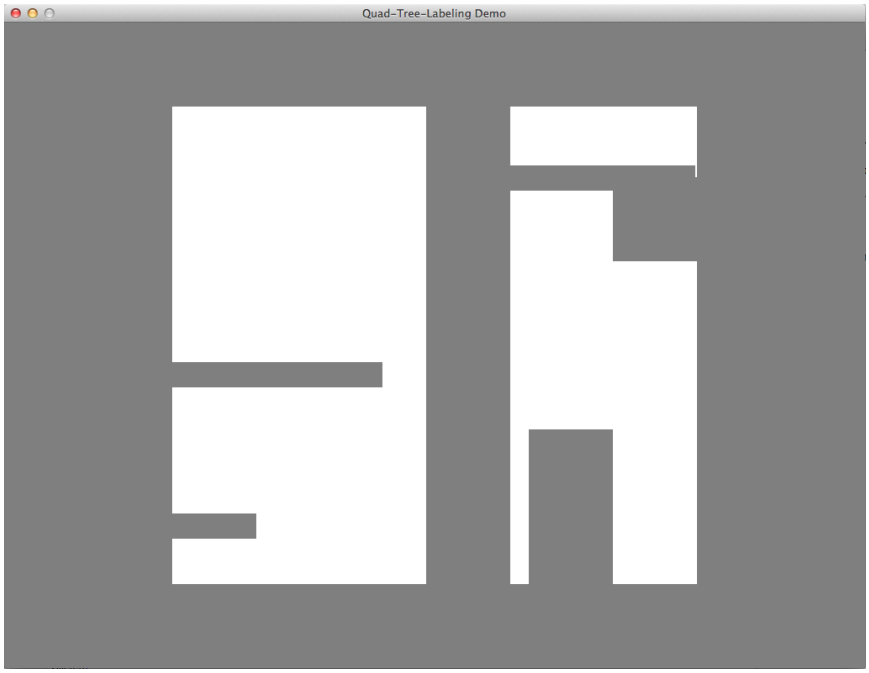
4.2 Collision Detection

A very important issue in terms of performance and accuracy is the collision detection method, which is used when it comes to a quad-trees generation. The algorithm 4.1 calls a generically denoted `collitionDetectionMethod`. This section gives an overview of different techniques and their qualities. See section 4.7 for a direct comparison of the discussed techniques performance and memory usage properties.

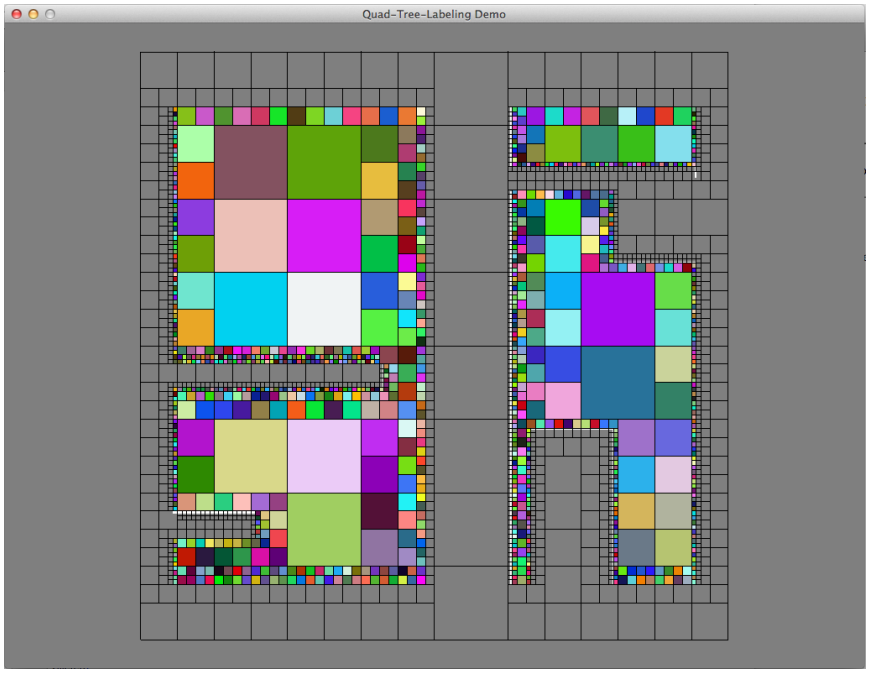
4.2.1 Point-Box Intersection

A very basic routine in 2D programming is to answer the question whether a point $P(x, y)$ intersects with a box (in the so far given examples the wall sections are represented as axis-aligned boxes) $R(u, v, w, h)$, where u, v mark the upper left corner point and w, h are width and height of the rectangle. Algorithm 4.2 shows the basic routine. Its integration in the quad-tree creation procedure is demonstrated in algorithm 4.3.

The major advantage of this approach is free adjustable (via ϵ as shown in algorithm 4.1) level of accuracy and the minimal number of quad-tree leafs it creates. A quad-tree creation which utilised this algorithm processes a collision detection for every position x, y within the quad-trees root node which is of course the most costly strategy one can imagine.



(a)



(b)

Figure 4.2: Figure (a) shows an example of a 2D world. The grey structures mark the walls. The screenshot has an original resolution of 1024×768 where $s = 700, m = 4$ and $\epsilon = 10$. Figure (b) shows the leafs of the resulting quad-tree (filled with random colour). The inaccuracy of this approximation can be seen at the white remaining areas.

Algorithm 4.1: Recursive computation of a quad-tree. A QuadTreeNode object n, which is the root node of the computed tree is passed in. s is the start size of the the quad-tree. m is the minimal size a leaf can have. ϵ is the threshold wether a leaf marks a wall or a room.

```
1:  function computeQuadTree(n)
2:      i, m ← yieldNodeType(n, m, ε)
3:      if i = 0 then
4:          c_S ← n.s/2                          ▷ c_S = size of the child nodes.
5:
6:          childNW ←< n, NW, c_S, x, y >
7:          n.s.[NW] ← childNW
8:          computeQuadTree(childNW)
9:
10: ... divide into NE, SE, SW in same fashion
11:
12:     else
13:         if i = 2 then
14:             n.m ← m
15:         end if
16:     end if
17: end function
18:
19:
20: function yieldNodeType(n, m, ε)
21:     d ← s/2 ≥ m                   ▷ Denotes if a leaf is further divisible.
22:     return < collisionDetectionMethod(n, m, ε, d) >
23: end function
```

4.2.2 AABB-AABB Intersection

A way more effective strategy is to make profit from the fact that quad-tree nodes are boxes as well, that can be easily tested for intersection with the worlds geometry segments. As the name suggests a constraint for this technique is of course that the game world is represented in axis aligned rectangles. See algorithm 4.4 for the collision test itself and 4.5 for its integration in the quad-tree creation process.

In this approach the priorly used threshold value ϵ has no affection and the results accuracy is determined by the minimum size m only. Another method (`isTotallyIncluded`(n, w)) is introduced, which checks if a quad-tree node n is totally included by a wall segment w. Otherwise it has to be further divided—if divisible. If it cannot be divided anymore, it is declared as a wall leaf.

This strategy is very fast since it operates on boxes with a minimum

Algorithm 4.2: A basic point-box intersection test. A point $P_{x,y}$ and a rectangle $R =< x, y, w, h >$ are passed in. The method returns 1 if P is located on R. Else 0 is returned. Note that the lines 2—12 could be expressed as "OR" constraints as well but the given solution performs better since unnecessary check-ups may be omitted.

```
1:  function pointBoxIntersection(P, R)
2:      if P.x < R.x then
3:          return 0
4:      end if
5:      if P.x > R.x + R.w then
6:          return 0
7:      end if
8:      if P.y > R.y then
9:          return 0
10:     end if
11:     if P.y < R.y − R.h then
12:         return 0
13:     end if
14:     return 1
15: end function
```

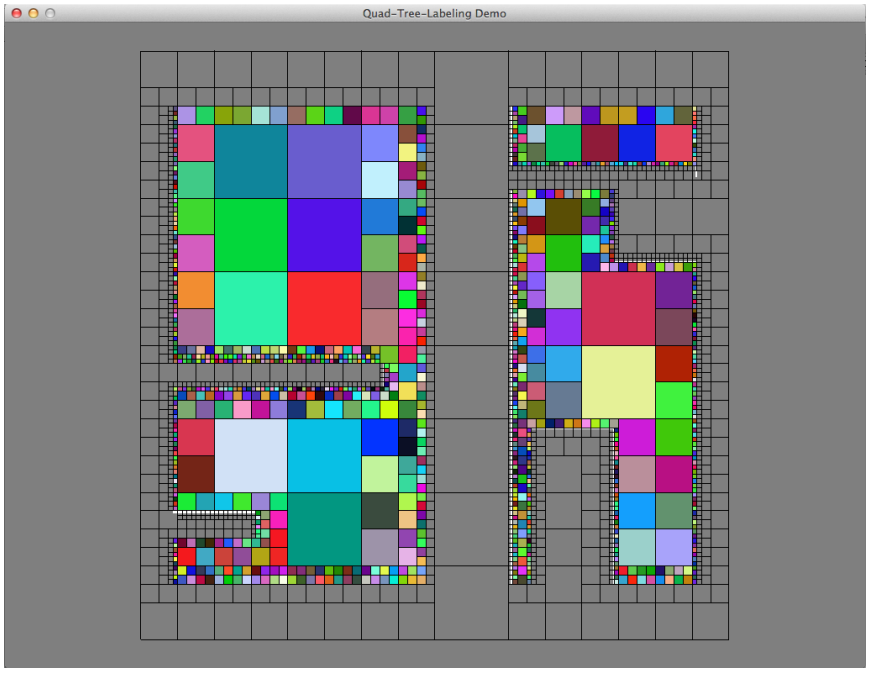
size of m but its accuracy is limited at the same coarse level. Furthermore, it creates a needlessly high number of quad-tree leafs which raise memory usage and may have negative affection to the performance of the system in large game worlds.

See figure 4.3 for a comparison of the so far presented collision detection techniques results and section 4.7 for a discussion of performance and memory usage.

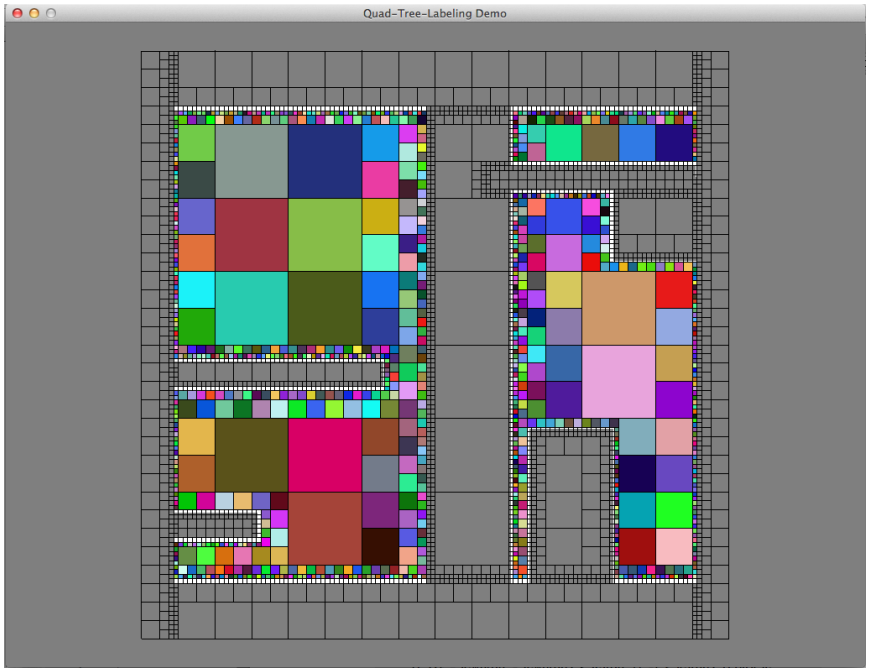
4.2.3 Hybrid 1

This technique combines the priorly described techniques in order to get the efficient behaviour of the AABB-AABB algorithm and the accuracy characteristics of the point-box intersection test. Therefore the functionality of the point-box intersection approach is extracted to the method `pointBoxCollision` and in the so far known `yieldNodeType` integrated as shown in algorithm 4.6.

Figure 4.4 shows a result of the "Hybrid 1" collision detection algorithm. It reveals an improvement in terms of accuracy compared to the pure AABB-AABB intersection algorithm. The slightly worse result in computational time can be seen in section 4.7.



(a)



(b)

Figure 4.3: Comparison of AABB-AABB intersection method (b) to point-box intersection detection (a). The AABB-AABB approach produces unnecessarily more quad-tree leaf instances, is less accurate but significantly faster than the point-box intersection technique (Hybrid 2 version—see 4.2).

Algorithm 4.3: The call of `collisionWithWalls` performs a point-box-intersection test of the point $P_{x,y}$ with every known wall segment. The method returns 0 if n must be divided, 1 if n is a room leaf and 2 if n is a wall leaf. In this case the material m is returned as well. Otherwise (0 or 1) `null` is returned instead of a material object.

```
1:  function YIELDNODETYPE(n, m, ε)
2:      s ← n.s/2
3:      d ← s/2 ≥ m                    ▷ Denotes if a leaf is further divisible.
4:      p_C ← 0                          ▷ Positive collision counter variable
5:      n_C ← 0                          ▷ Negative collision counter variable
6:      m ← null                              ▷ material initialised by null.
7:      for all positions x, y of n do
8:          if collisionWithWalls(x, y) then
9:              p_C ← p_C + 1
10:             if m = null then
11:                 m ← collisionWithWalls(x, y).m
12:             end if
13:         else
14:             n_C ← n_C + 1
15:             if n_C > ε∧!d then
16:                 return 1, null
17:             end if
18:             if p_C > ε∧!d then
19:                 return 2, m
20:             end if
21:             if n_C > ε ∧ p_C > ε ∧ d then
22:                 return 0, null
23:             end if
24:         end if
25:     end for
26: end function
```

4.2.4 Hybrid 2

The output result of "Hybrid 2" is exactly the same as a quad-tree processed by using the point-box-intersection technique while revealing a way better computational efficiency. The algorithm can be seen in 4.7.

4.3 Labeling a Quad-Tree

Like described in section 4.1 we gained data which allow very efficient tree search operations but furthermore there is still the need to connect the leafs of our search tree semantically and determine their affiliation with a room.

Algorithm 4.4: A straight forward intersection detecting algorithm for axis-aligned boxes only. It has a boolean returning type and two rectangles $R1, R2$ as input values. Each rectangle consists of an upper left corner point (x, y), a width (w) and height (h) value.

```
1: function AABBAABBCOLLISION(R1, R2)
2:     if R1.x + R1.w < R2.x ∨ R1.x > R2.x + R2.w then
3:         return 0
4:     end if
5:     if R1.y > R2.y − R2.h ∨ R1.y − R1.h > R2.y then
6:         return 0
7:     end if
8:     return 1
9: end function
```

Algorithm 4.5: Integration of the "AABB-AABB" intersection test in the quad-tree creation process. A new object type `Wall` $w \leftarrow < x, y, w, h, m >$ appears here, which is an axis-aligned rectangle with already described properties and an additional `Material` m assigned. The game worlds geometry consists of `Walls`.

```
1: function YIELDNODETYPE(n, m)
2:     s ← n.s/2
3:     d ← (s/2 ≥ m)                    ▷ Denotes if a leaf is further divisible
4:     w ← checkCollisionWithAllWalls(n)
5:     if w ≠ null then
6:         if isTotallyIncluded(n, w) then
7:             return 2, w.m
8:         else
9:             if d then
10:                return 0, null
11:            else
12:                return 2, w.m
13:            end if
14:        end if
15:    else
16:        return 1, null
17:    end if
18: end function
```

As one looks at the input-data there is only one binary question to ask for each x, y in the game world $I_{(x,y)}$. Whether a coordinate (x, y) belongs to a wall or a room. This setup reminds strongly of the problem of **labeling**, which typically comes from the field of digital imaging.

Algorithm 4.6: This algorithm differs from the AABB-AABB approach by utilising the point-box-collision technique if a node cant be divided anymore and it is neither a pure room leaf nor a pure wall leaf.

```
1:  function YIELDNODETYPE(n, m, ε)
2:      w ← checkCollisionWithAllWalls(n)
3:      if w ≠ null then
4:          if isTotallyIncluded(n, w) then
5:              return 2, w.m
6:          else
7:              if d then
8:                  return 0, null
9:              else
10:                 return pointBoxCollision(n, m, ε)
11:             end if
12:         end if
13:     else
14:         return 1, null
15:     end if
16: end function
```

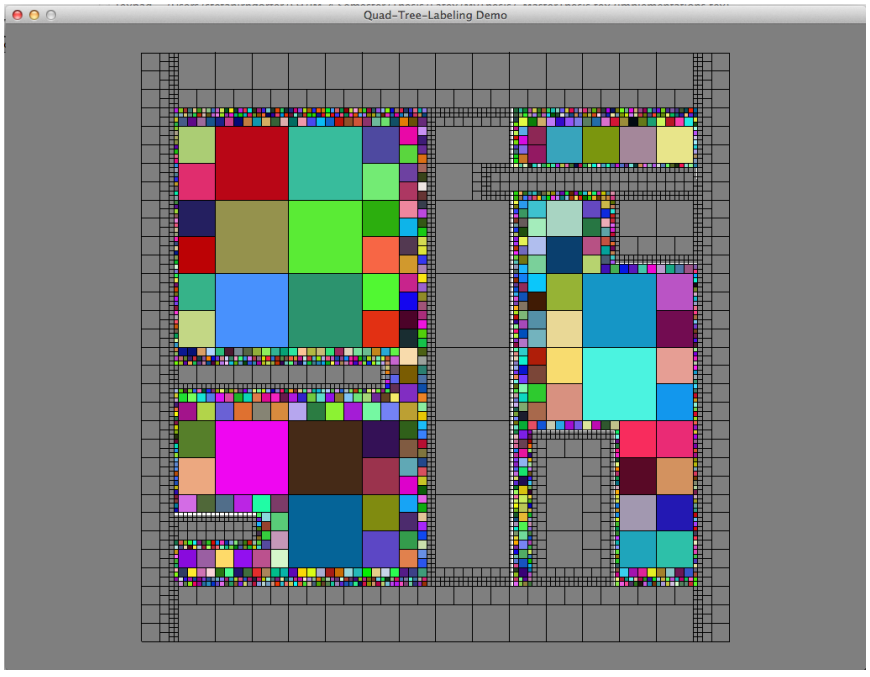
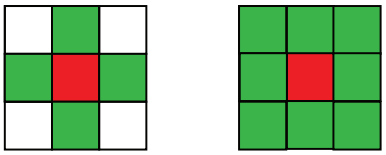


Figure 4.4: Result of a quad-tree creation utilising the "Hybrid 1" collision detection technique. Its accuracy is regarding to the threshold parameter ϵ, while still unnecessarily many nodes are produced.

Algorithm 4.7: The presented algorithm performs the "Hybrid 2" collision detection and differs to "Hybrid 1" by resolving **all** nodes which are not purely wall or room nodes utilising the point-box-intersection technique.

```
1: function YIELDNODETYPE(n, m, ε)
2:     w ← checkCollisionWithAllWalls(n)
3:     if w ≠ null then
4:         if isTotallyIncluded(n, w) then return 2, w.m
5:         elsereturn pointBoxCollision(n, m, ε)
6:         end if
7:     else
8:         return 1, null
9:     end if
10: end function
```



(a) (b)

Figure 4.5: A key concept for common algorithms in binary image labeling is the connectivity of single pixels. Figure (a) shows a 4-neighbourhood while a 8-neighbourhood can be seen in (b).

4.3.1 Introduction: Labeling

> In binary images, a pixel can take on exactly one of two values. These values are often thought of as representing the "foreground" or the "background" in the image [3].

The primary task of region labeling in binary images is to interpret the number and type of objects in such images. A connected binary region is a group of touching foreground pixels in the simplest case.

There are various techniques known to solve this problem on binary images such as region labeling with flood filling (either iterative, recursive or combined solutions[1]).

Independent of which technique is used, the type of neighbouring must be settled before to determine of two pixels are "connected" to each other. Figure 4.5 shows how a 4- or a 8-connected definition of "neighbouring" looks like. The different definitions could lead to different results (see [3]).

[1] See [3] for more detailed information about the various techniques on region labeling in binary images.

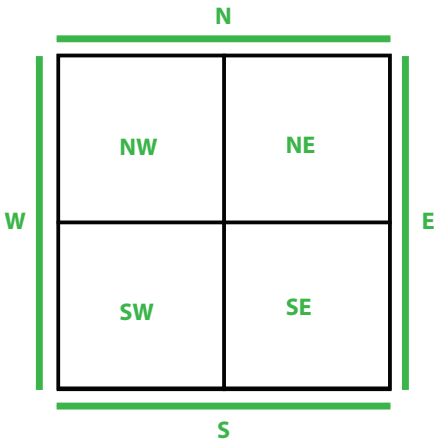


Figure 4.6: Note, that the quadrants are denoted as "Northwest", "Northeast", "Southwest" and "Southeast", while the boundaries are called like geographic directions (N, E, S, W).

4.3.2 Algorithmic Solution for (Binary) Quad-Trees

As seen in section 4.3.1 a very fundamental idea of labeling algorithms in general are neighbourhoods. But in case of a quad-tree this concept is not given in a clear way. This section refers to an algorithmic solution by *Hanan Samet* published in 1981 (see [10]).

Fundamentals

A fundamental concept for this approach is the relation between a quadrant and the boundaries of its parent node as shown in 4.6.

The "reflect" method returns the `sonType` value r, which is located at a given `sonType` i reflected over a direction d. Consider figure 4.6 at the following examples to understand `sonType reflect(direction, sonType)`: `reflect(W, NW)` returns `NE`, `reflect(S, NW)` returns `SW`.

The "adj" method is very simple and returns `true` or `false` if a given `sonType` value r is adjacent to a border d. For example: `adj(W, NW)` gives `true` while `adj(S, NW)` returns `false`. See figure 4.7 for the relationship of the so far described classes.

4.3.3 Key Functions of the Labeling Class

The visitation of the quad-tree starts from its northwestern most element and propagates towards the east and the south. It traverses the tree in postorder by recursive calls. The main call of the labeling procedure is shown

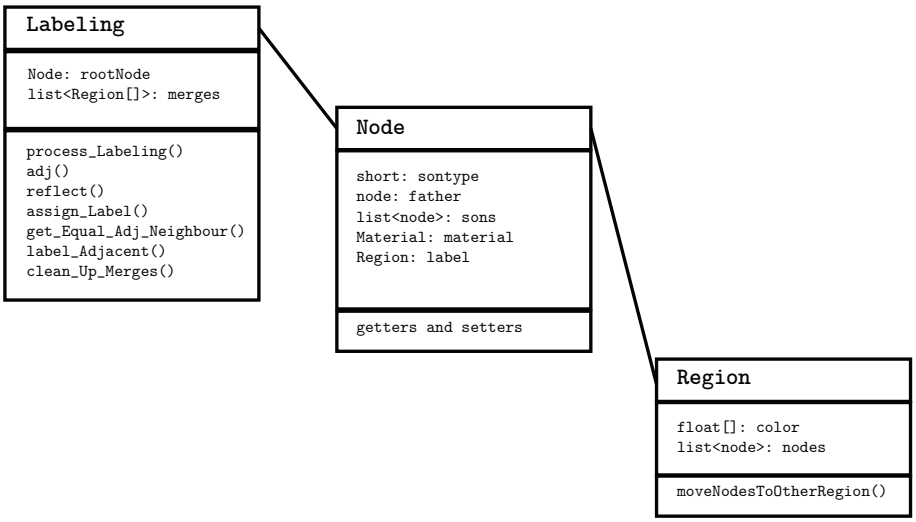


Figure 4.7: The class diagram visualises the relations of the so far described classes. A closer look at the "Labeling" class's methods is taken in section 4.3.3.

in algorithm 4.8. As already mentioned a major problem is the retrieval of neighbouring elements which is solved by the function `getEqualAdjacentNeighbour(QuadTreeNode` p`, short` d`)`. It returns the neighbour of node p in direction d, which has a greater or equal size. If the node p is on the edge of the game world, `null` is returned [10]. See algorithm 4.9.

4.3.4 Merging Regions

At line 7 in algorithm 4.10 a method `assignLabel(`p`,` q`)` is called. The input parameters q and p are both QuadTreeNodes which are identified as neighbouring room elements. In other words, these elements belong to the same region and have to end up with the same label. If p already is labeled, q will be assigned the same label and vice versa. If none of these leafs is labeled yet, a new region object is created and assigned to both as label. Under certain circumstances q and p are already assigned with different labels (see 4.12). In this case these regions are added to `ArrayList<Region[]> merges`. Iterating over `merges` and merging its entries is the final step of this labeling technique.

4.4 The General Quad-Tree Traversal Technique

The combination of the algorithms 4.9 and 4.10 is a fundamental technique to "operate" on a quad-tree. The retrieval of a regions border elements and computation of apertures as described in 4.6 use this approach as well.

Algorithm 4.8: Main call of the labeling procedure as described in [10]. A QuadTreeNode p is passed in.

```
1: function LABEL(p)
2:     if p.s[0] ≠ null then                    ▷ label in order NW, NE, SW, SE
3:         label(p.s[NW])
4:         label(p.s[NE])
5:         label(p.s[SW])
6:         label(p.s[SE])
7:     else
8:         if p.m = null then
9:             q ← getEqualAdjNeighbour(p, E)
10:            if q ≠ null then
11:                labelAdjacent(q, NW, SW, p)
12:            end if
13:            q ← getEqualAdjNeighbour(p, S)
14:            if q ≠ null then
15:                labelAdjacent(q, NW, NE, p)
16:            end if
17:        end if
18:    end if
19: end function
```

Algorithm 4.9: Algorithmic description of the retrieval of an equally or greater sized neighbouring node of node p in direction d.

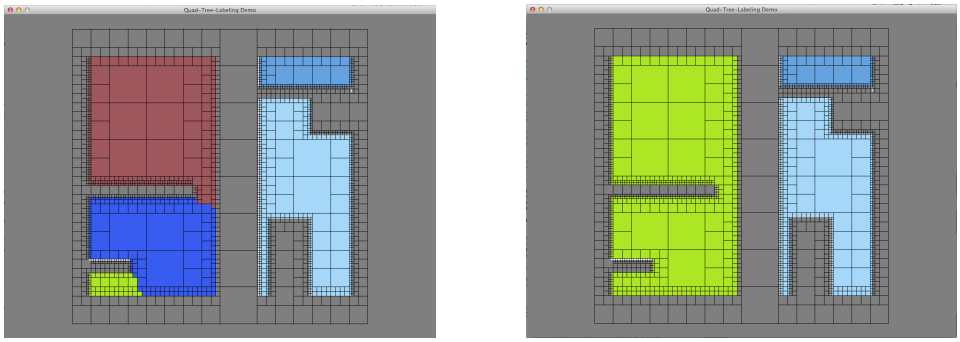
```
1: function GETEQUALADJNEIGHBOUR(p, d)
2:     QuadTreeNode q ← null                    ▷ init node q with null
3:     if p.f ≠ null ∧ adj(d, p.t) then
4:         q ← getEqualAdjNeighbour(p.f, d)
5:     else
6:         q = p.f
7:     end if
8:     if q ≠ null ∧ q.s[0] ≠ null then
9:         r ← reflect(d, p.t)
10:        q = q.s[r]
11:    end if
12:    return q
13: end function
```

4.4.1 Generalisation of this Technique

In the following sections this technique will be denoted as **general quadtree traversal** technique. To generalise the function `labelAdjacent()` has

Algorithm 4.10: Finds all descendants of node r adjacent to node p, in quadrants q_1 and q_2.

```
1: function LABELADJACENT(r, q1, q2, p)
2:     if r.s[0] ≠ null then
3:         labelAdjacent(r.s[q1], q1, q2, p)
4:         labelAdjacent(r.s[q2], q1, q2, p)
5:     else
6:         if p.m = null then
7:             assignLabel(p, r)
8:         end if
9:     end if
10: end function
```



(a) (b)

Figure 4.8: Since the algorithm propagates from the northwestern most element towards the east and the south, in certain constellations a wall segment may "cover" underlying leafs. See the yellow and dark blue areas in example (a). After processing `cleanUp_Merges()` the affected regions are unified as shown in (b).

to be changed to `operateOnAdjacent()` which is described in algorithm 4.11.

Let us discuss an example for a deeper understanding. A `QuadTreeNode` P is given and an operation on the element(s) in the east should take place. Algorithm 4.12 shows how to invoke this action.

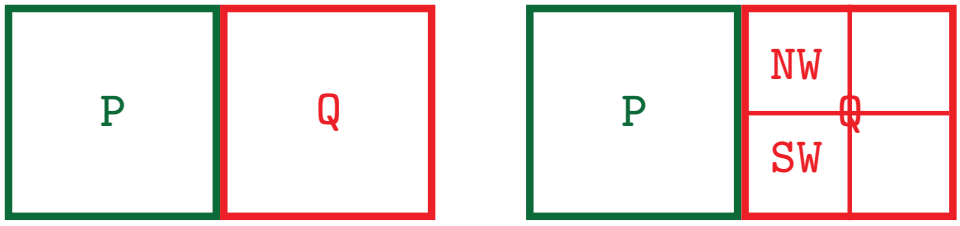
The neighbourhood in the **east** of a node P should be explored. The call of `getEqualAdjacentNeighbour(`P`,` E`)` will return Q. If $Q \neq null$, it will be passed to `operateOnAdjacent(`Q`,` NW`,` SW`,` $[x]$`)`. In figure 4.9 (a) Q is a leaf, while in (b) a node has to be handled. In this case `operateOnAdjacent` will do recursive calls to operate on both: The northwestern and the southwestern quadrants of Q.

Algorithm 4.11: Generalisation of algorithm 4.10. Again a node r and the sontypes (NW, NE etc.) q_1, q_2 are passed in. Additional parameters which might be needed for the specific procedure are denoted as $[x]$.

```
1: function operateOnAdjacent(r, q1, q2, [x])
2:     if r.s[0] ≠ null then
3:         labelAdjacent(r.s[q1], q1, q2, [x])
4:         labelAdjacent(r.s[q2], q1, q2, [x])
5:     else
6:         ...                          ▷ Operations on the desired element.
7:     end if
8: end function
```

Algorithm 4.12: The `sonType` values q_1 and q_2 are the reflected quadrants to the desired direction. Example: Eastern direction northwestern and southwestern quadrants.

```
1: ...
2: QuadTreeNode Q ← getEqualAdjacentNeighbour(P, E)
3: if Q ≠ null then
4:     operateOnAdjacent(Q, NW, SW, [x])
5: end if
6: ...                          ▷ ... can be performed towards N, E, S and W
```



(a) (b)

Figure 4.9: The method `operateOnAdjacent` has to distinguish weather Q is a leaf (a) or a node (b). In case of (b) a recursive call will carry out the operation (which may use the parameter $[x]$) on both (NW and SW) quadrants.

4.5 Handling Changes to the Geometry

As one thinks about the graph of the quad-tree example in figure 4.1, a changing on the described geometry—whether a destruction or a creation—can be simply handled by the replacement of the affected subtree. In other words the major part of the magic has already happened in the sections 4.1.2 and 4.3.

4.5.1 Merging Changes into Existing Data

To process a changing, that affects several wall segments we assume that a list of added or removed geometry elements is passed to our system. As a next step the outer bounds of the geometrical changing b is computed. This simple task is described in algorithm 4.13.

Algorithm 4.13: This function simply computes and returns the boundary coordinates ($b =< i_{x,y}, j_{x,y} >$) of a list of rectangles ($w =< e_1, e_2, ..., e_n >$). Note, that we assume an upright positive y-axis.

```
1:  function computeOuterBounds(w)
2:      i.x ← MAX                          ▷ init returning values with MIN/MAX.
3:      i.y ← −MAX
4:      j.x ← −MAX
5:      j.y ← MAX
6:      for all e of w do
7:          if e.x < i.x then
8:              i.x ← e.x
9:          end if
10:         if e.x + e.w > j.x then
11:             j.x ← e.x + e.w
12:         end if
13:         if e.y > i.y then
14:             i.y ← e.y
15:         end if
16:         if e.y − e.h < j.y then
17:             j.y ← e.y − e.h
18:         end if
19:     end for
20:     b ←< i, j >
21:     return b
22: end function
```

With this data the so far known quad-tree can be traversed, starting at the root node r, in order to find the `QuadTreeNode` n, which totally embraces the outer bounds b. Algorithm 4.14 demonstrates how this can be achieved by utilising the method
`boolean totallyIncluded(QuadTreeNode` n`, double[]`b),
which simply returns `true` if n totally includes b, else `false` is returned. The algorithm works under the assumption that b is at least totally included by the root node r.

Now, that the root node of the subtree which has to be recalculated is known, it simply can be passed to the `computeQuadTree` algorithm (4.1).

The logical next step is to pass this subtrees root node to the labeling

Algorithm 4.14: This method returns the `QuadTreeNode` n which totally embraces a rectangle described by the coordinate set $b =< i_{x,y}, j_{x,y} >$.

```
1:  function findAllEmbracingNode(b, r)
2:      n ← r                                          ▷ Copy input node r
3:      c ← r.s
4:      if totallyIncluded(c[NW], b) then
5:          n ← findAllEmbracingNode(b, c[NW])
6:      else
7:          if totallyIncluded(c[NE], b) then
8:              n ← findAllEmbracingNode(b, c[NE])
9:          else
10:             if totallyIncluded(c[SW], b) then
11:                 n ← findAllEmbracingNode(b, c[SW])
12:             else
13:                 if totallyIncluded(c[SE], b) then
14:                     n ← findAllEmbracingNode(b, c[SE])
15:                 end if
16:             end if
17:         end if
18:     end if
19:     return n
20: end function
```

algorithm (4.8). Figure 4.10 shows the intermediate result prior the call of `cleanUp_Merges()`.

As one can see in the image, this would lead to an unsatisfying result since the northern border of n adjoins an already known region which has to be merged with the currently calculated region in n. To solve this problem all the bordering elements of n have to be visited in order to check, if already labeled, non-wall elements are beyond their borders. Algorithm 4.15 shows how this can be accomplished, assuming, that n is already known and by using a variant of the general quad-tree traversal technique (see section 4.4). The final result can be seen in figure 4.11.

In the discussed example two additional wall segments appeared with no influence to the number of regions in the game world. See figures 4.12 (a) and (b) for another example where a wall segment is removed, so that two priorly known regions merge to one.

4.6 Calculating Apertures

As described in section 2.2.6 apertures are lines, located on the border of a region. The placement of these lines depends on the occluding attributes of

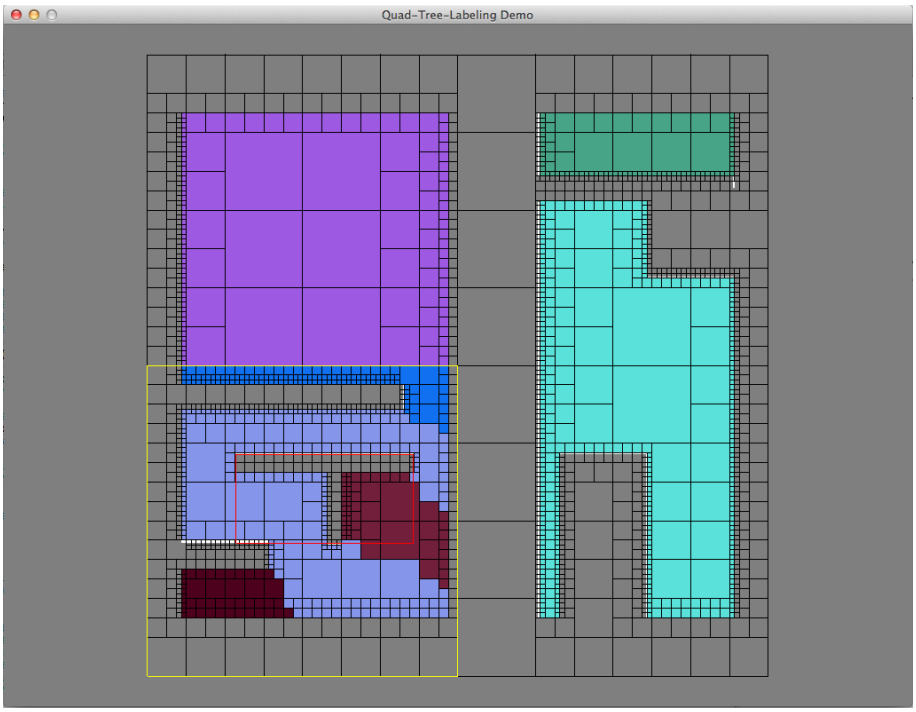


Figure 4.10: The red rectangle marks the outer boundaries of the changed geometry, while the yellow box denotes the node n which is the nearest ancestor of the affected leafs that totally includes the red rectangle.

the wall in this section. To be clearer, a sound that occurs in an adjacent room will appear from the aperture which has less occlusion to the room, the listener is currently in.

Sources are placed on point coordinates and not on lines. So the source has to be placed on the point on the aperture which is nearest to the listener.

4.6.1 Finding a Quad-Tree Regions Border Elements

A first step to yield the apertures of a region to its adjacent ones is to find the bordering elements. The knowledge of a regions border is not just necessary to calculate apertures. The material composition of a regions "walls", which is needed to parameterise the environmental effects (see section 2.2.3), can only be derived from the border elements of this room. To set up the environmental effects, the "shape" of a room should also be considered beside the area. Therefore other geometrical properties of this region may be calculated[2]. The very interesting question how e.g. the eccentricity of a quad-tree region could be calculated and how the result should influence the

[2]See [3] for more information about different geometrical properties (e.g. compactness and roundness) of binary image regions.

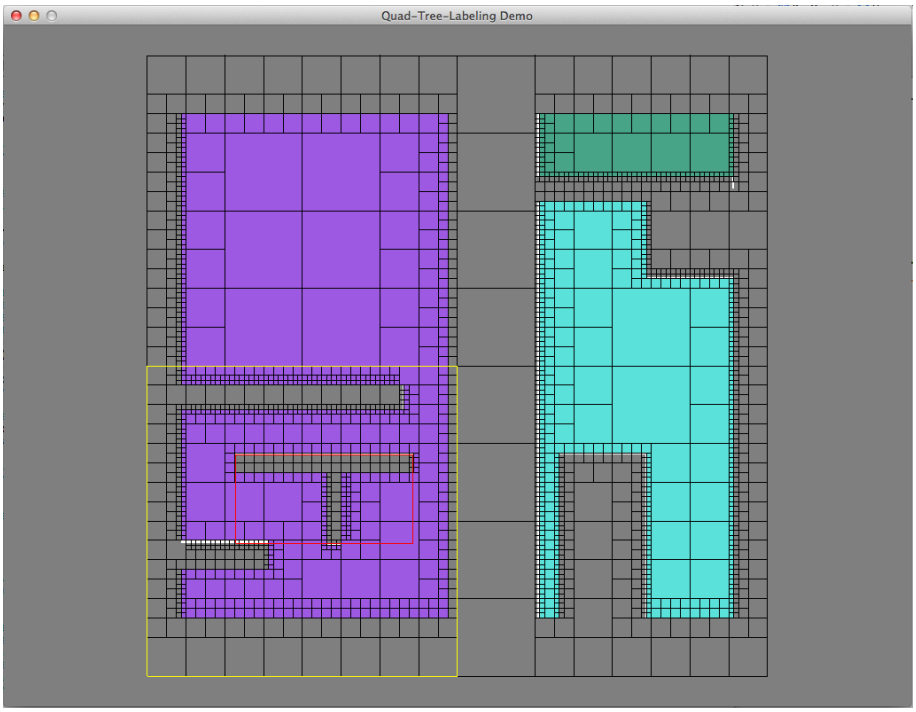
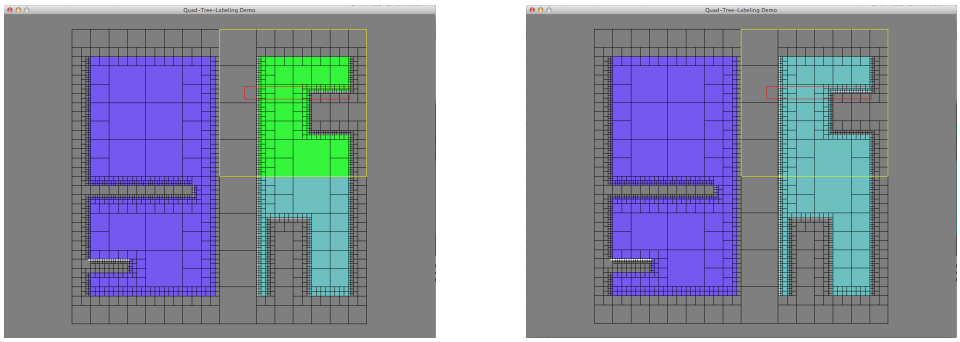


Figure 4.11: Solved merging by using the algorithm like described in 4.5.1.



(a) (b)

Figure 4.12: A wall segment is removed, so that priorly known regions have to merge. Image (a) shows the status before the call of `cleanUp_Merges()`. The final result can be seen in (b).

parameterisation of a reverb effect would exceed the scope of this work.

The border elements of a quad-tree region can be found by using the general quad-tree traversal technique (see 4.4) for each leaf of a region in every direction. This variant of `operateOnAdjacent` has to set the entry

Algorithm 4.15: The input value `QuadTreeNode` n embraces all affected elements of the quad-tree. This algorithm describes how to visit the borders (N, W, S, E) of n in order to check the seam to the so far known quad-tree to perform a sufficient merging later on.

```
1: function RELABEL(n)
2:     removeAllLabels(n)
3:     label(n)
4:     visitBorder(N, n, NW, NE)
5:     visitBorder(W, n, NW, SW)
6:     visitBorder(S, n, SW, SE)
7:     visitBorder(E, n, NE, SE)
8: end function
9: function VISITBORDER(p, q1, q2, d)
10:    if p.s[0] ≠ null then
11:        visitBorder(p.s[q1], q1, q2, d)
12:        visitBorder(p.s[q2], q1, q2, d)
13:    else
14:        q ← getEqualAdjNeighbour(p, d)
15:        if q ≠ null then
16:            assignLabel(p, q)
17:        end if
18:    end if
19: end function
```

`isBorderElement` ← `true` and the direction of the adjacent wall element in each `QuadTreeNode` which has a neighbouring wall element. Additionally a list of these border leafs `List<QuadTreeNode> borderElements` must be collected and stored in the respective region object.

4.6.2 Extracting Apertures

Object-Type: ConnectivityData

Each leaf, which is identified as a bordering element gets a connectivity data object assigned.

$$c \leftarrow < r, d, g_l, g_h >$$

Connectivity data consist of:

- r: `adjacentRegion`—Initialised to `null`.
- d: `direction`—The direction towards the adjacent wall element.
- g_l: `lowFrequencyGain`—Initialised to 1.0.
- g_h:`highFrequencyGain`—Initialised to 1.0.

In order to find these apertures the following requirements have to be fulfilled after the successive search for a regions border elements like described above:

- Every region object keeps a list of its border leafs (see section 4.6.1)
- Every border leaf keeps a record about the direction (N, E, S, W) to the adjacent wall element.
- Every wall leaf of our quad-tree keeps a record about the "material" (see section 3.2) it is made of.
- Every region (room) has closed boundaries. Openings like windows and doors are modelled by using a "clear air material".

Since apertures consist of a subset of the bordering elements, the algorithm has to visit each border element n of a region R and perform an exploration towards its `borderDirection` d. Again the general quad-tree traversal technique (see section 4.4) is used. It proceeds towards d and decreases `lowFrequencyGain` l and `highFrequencyGain` h accordingly to the current wall elements size and the material it is made of.

The stopping criteria for this procedure are:

- A room node is reached.
- The gain values l and h have become ≤ 0.0.

As soon as the exploration has reached a room leaf, a check for this leafs regional belonging has to take place, to make sure, that the found region is not R again.

The method `exploreBorder` is another variant of `operateOnAdjacent` (see algorithm 4.11). It explores a wall, starting from a bordering leaf n towards direction d as shown in algorithm 4.16. The QuadTreeNode q is the adjacent wall element which was priorly found by the `getEqualAdjacentNeighbour` method, while c is the current `ConnectivityData` object.

A shortcoming of this approach is that diagonally located elements are neglected. Therefore an enhancement of the quad-tree traversal could be considered.

4.7 Performance and Memory Usage

4.7.1 Performance Tests

Since the collision detection calculations take the major part of the computing time, this section has a strong focus on the comparison of the different techniques described in section 4.2. The described performance measurements were done without any graphical representation so that different rendering times could not blur the outcomes. The tests were done on the already well known game world geometry setup as used in the sections before.

Algorithm 4.16: The algorithm performs a walls occlusion measurement starting from a QuadTreeNode n. QuadTreeNode q is the adjacent wall and c is the current connectivity data.

```
1: function exploreBorder(q, q1, q2, n, c)
2:     if b.s[NW] ≠ null then
3:         exploreBorder(q.s[q1], q1, q2, c)
4:         c2 ← copy(c)
5:         exploreBorder(q.s[q2], q2, q1, c2)
6:     else
7:         if q.m = null then                ▷ A non wall element is reached
8:             if q ≠ n then
9:                 c ← null
10:                return                     ▷ We are in the same region!
11:            else
12:                c.r ← q.r
13:                n.c ← c
14:            end if
15:        else
16:            s ← q.w
17:            m ← q.m
18:            l ← c.g_l
19:            h ← c.g_h
20:            ...      ▷ Compute l and h reduction accordingly to s and m.
21:            if l ≤ 0.0 ∧ h ≤ 0.0 then
22:                c ← null return            ▷ Total occlusion
23:            else
24:                c.g_h ← h
25:                c.g_l ← l
26:            end if
27:            d ← q.c.d
28:            r ← getEqualAdjacentNeighbour(q, d)
29:            if r ≠ null then
30:                exploreBorder(r, q1, q2, n, c)
31:            else
32:                return
33:            end if
34:        end if
35:    end if
36: end function
```

Table 4.1: s = the scale factor. We assume a root node size of 700 units ($s = 1$). If for example $s = 0.5$ root node size is 350. The whole geometry is scaled by s as well. $sMin$ = the minimum size of a leaf, ϵ = the accuracy threshold value, t = the duration (of collision detection only) in milliseconds and M = the amount of created leafs.

Technique	s	$sMin$	ϵ	t	M
Point-Box	0.5	10	4	221.427	841
AABB-AABB	0.5	10	-	18.949	1312
Hybrid 1	0.5	10	4	40.754	1312
Hybrid 2	0.5	10	4	327.946	841
Point-Box	1.0	10	4	1632.224	1891
AABB-AABB	1.0	10	-	32.712	2833
Hybrid 1	1.0	10	4	103.44	2833
Hybrid 2	1.0	10	4	727.914	1891
Point-Box	1.5	10	4	6740.274	3166
AABB-AABB	1.5	10	-	66.768	5956
Hybrid 1	1.5	10	4	241.53	5956
Hybrid 2	1.5	10	4	2282.84	3166

The table 4.1 shows the results of performance tests with different collision detection techniques as described in section 4.2 on different scales of the example game worlds. As one can see in column M the "AABB-AABB" approach and the "Hybrid 1" variant generate the same number of quad-tree leafs while the "Point-Box" intersection technique and the "Hybrid 2" variant generate an identical quad-tree as well.

Figure 4.13 compares these groups graphical and shows their characteristics in differently scaled set ups.

The computing time measurements of the quad-tree creation can be seen in 4.14, while the major chunk of computational complexity is used for collision detections as shown in 4.15.

4.7.2 Memory Usage

Figure 4.16 shows the class diagram of the so far described implementations. In table 4.2 the memory usage of these classes on a 64 bit machine is listed. Each instance has a 16-bit header and its total size is rounded up to the next multiple of 8[3]. Table 4.3 exemplifies how these memory usage calculations are done on the "Material" class `Concrete`.

[3]http://btoddb-java-sizing.blogspot.co.at/

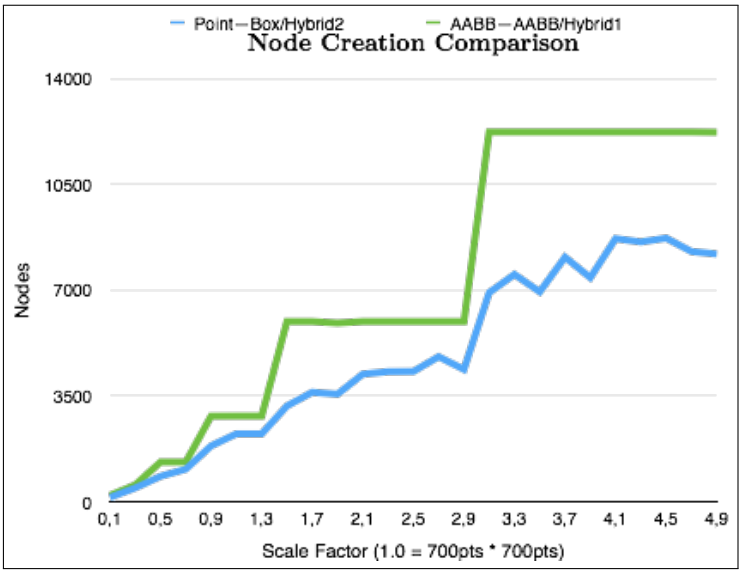


Figure 4.13: The growth of the number of generated nodes has no linear behaviour since different scaled set ups can be divided more or less efficient into quad-tree nodes.

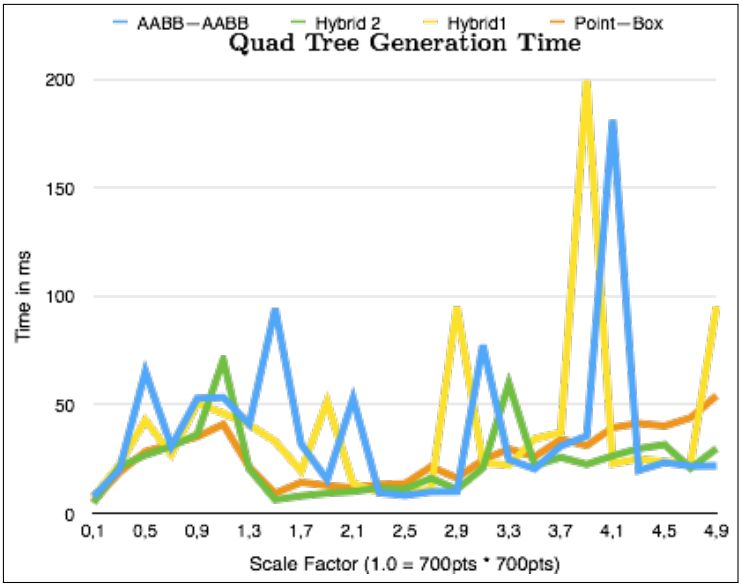
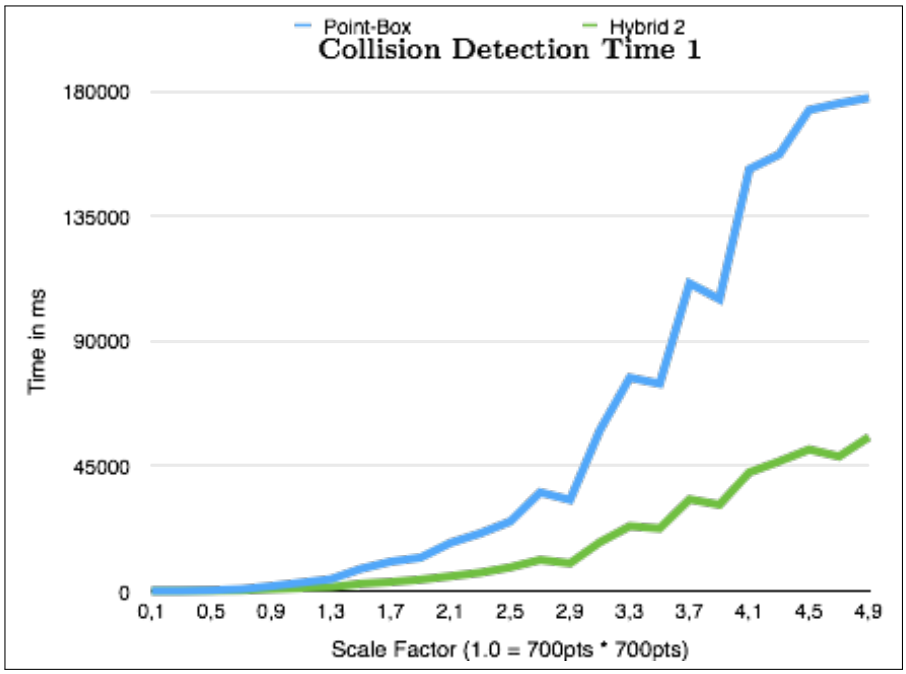
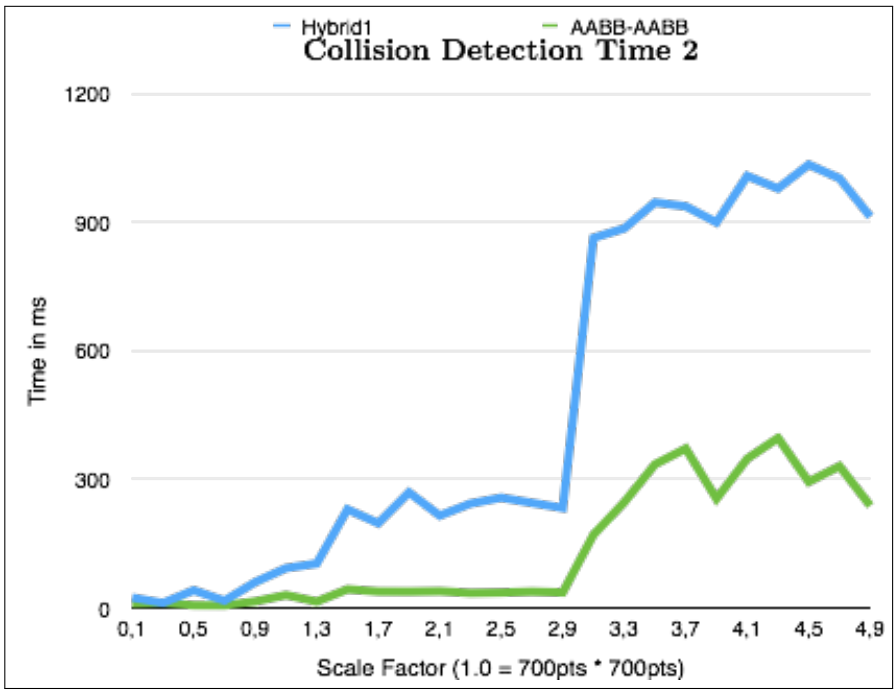


Figure 4.14: This graph shows the computational times of the quad-tree generation itself which means the collision detection times are subtracted. There are similarities between “Point-Box” and “Hybrid 2” as well as between “AABB-AABB” and “Hybrid 1” noticeable again. This is caused by the same number of produced nodes the respective group has to process.



(a)



(b)

Figure 4.15: The graphics (a) and (b) compare the performance of the described collision detection techniques on different scales of the example game world.

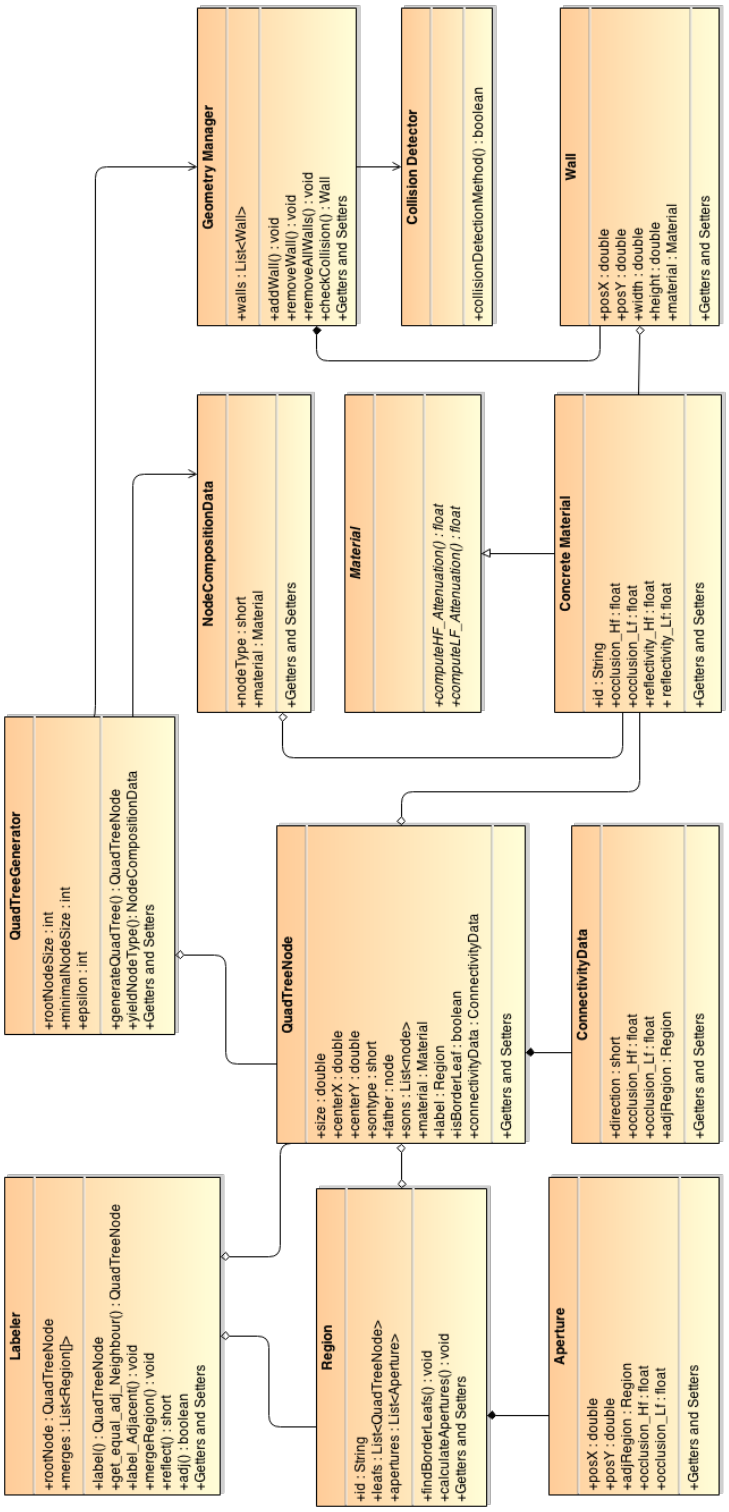


Figure 4.16: The class diagram related to the techniques described in section 4. Any memory usage calculations in this section refer to this listings.

Table 4.2: Memory usage in bytes per instance of each class as seen in class diagram 4.16. The byte values are minimum sized. Arrays, which are held by some classes are assumed to be empty. Their minimum memory size (32 bites) is taken into account.

Class	**bytes / instance**
QuadTreeNode (Leaf)	104
QuadTreeNode (Node)	144
ConnectivityData	40
Region	200
Aperture	48
Material	152
Wall	56

Table 4.3: Memory usage calculation of the class "Concrete". Note, that the total sum 152 is divisible by 8. Otherwise an additional padding must be taken into account.

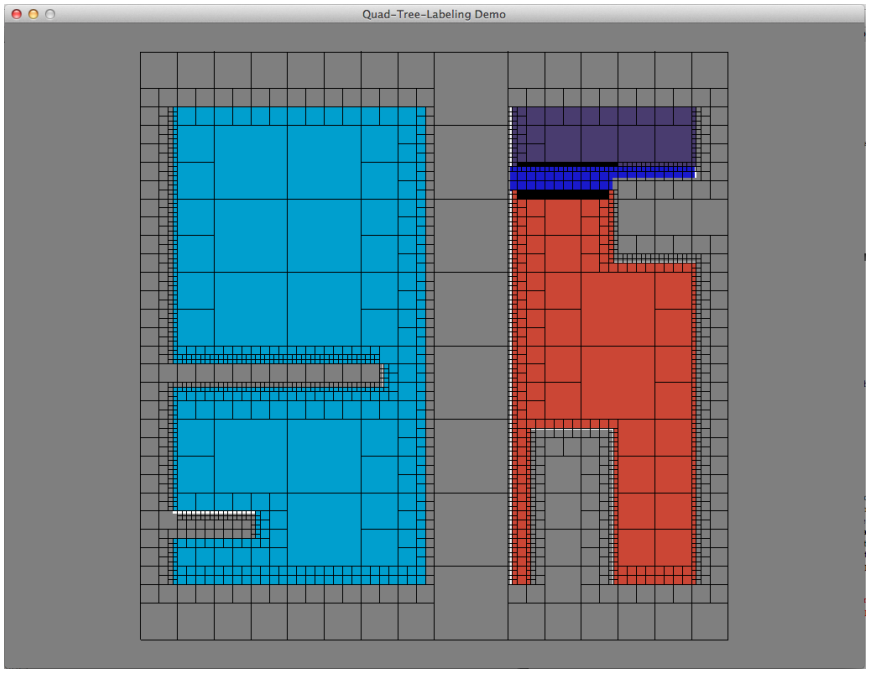
Concrete	
Class header	16 bytes
String id	
String overhead	64 bytes
`"Material14102574195622250000"` $\rightarrow 27 * 2$ bytes	52 bytes
Padding (to the next multiple of 8)	4 bytes
float occlusion_Hf	4 bytes
float occlusion_Lf	4 bytes
float reflectivity_Hf	4 bytes
float reflectivity_Lf	4 bytes
Total	152 bytes

Example

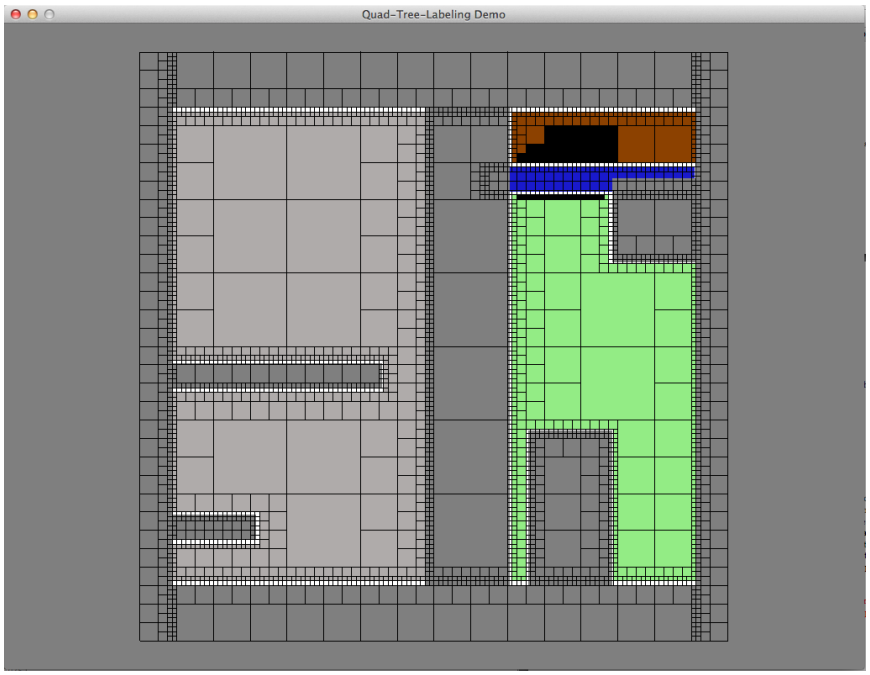
Since the listings in table 4.2 are calculated with empty arrays in the various classes actual memory usage has to be demonstrated by examples.

Let us compare the memory usages of the same game world geometry solved by different collision detection algorithms. As described in the sections 4.2 and 4.7.1 the resulting quad-trees differ depending on the used collision detection technique.

Figure 4.17 shows the solved example game world by utilising the Hybrid 2 (a) approach, respectively the AABB-AABB collision detection technique (b). The different results in terms of memory cost are listed in table 4.4.



(a)



(b)

Figure 4.17: The figures above show the test game world differently solved. For (a) "Hybrid 2" collision detection algorithm was used while "AABB-AABB" was utilised for (b). The walls consist of 17 "Concrete" segments and 1 "Clear Air" wall (the horizontal dark blue segment in the upper right). Leafs which have acoustic connectivity to adjacent Regions are drawn black.

Table 4.4: The table below sums up the memory usage of all objects as created on the game world examples shown in figures 4.17 (a) and (b). The column c denotes the count of the particular object, while B stands for byte. L abbreviates "Leafs" and N replaces "Nodes".

Hybrid 2			**AABB-AABB**		
c	Object	B	c	Object	B
1891	L	196664	2833	L	294632
630	N	65520	944	N	98176
3	Region:		3	Region:	
	R1: 85 L, 1 Arp.	872		R1: 89 L, 1 Arp.	904
	R2: 371 L, 1 Arp.	3160		R2: 336 L, 1 Arp.	2888
	R3: 495 L	3952		R3: 502 L	4208
32	ConnectivityData	1280	23	ConnectivityData	920
2	Aperture	96	2	Aperture	96
2	Material	304	2	Material	304
18	Wall	1008	18	Wall	1008
Total		272856	Total		403136

4.8 Optimisation

In the sections above the computations of regions, the regions borders, the connectivity to other regions and the location of this connections (apertures) have been described in separated steps. All of these procedures use their specified variants of the **general quad-tree traversal** technique. An obvious optimisation approach is to combine these steps to minimise the number iterations. For didactic purposes and the possibility to do performance tests on each of these steps they are described separately in this thesis.

As one can see in section 4.7.1 the computational complexity of operations on quad-trees in general depends strongly on the size of this tree while the trees size increases in steps on linearly increased scales of the game world. This means, that a potential optimisation lies in the game worlds design. In order to optimise a game worlds design in respect to the "Dynamic Geometry Middleware" further investigations would be necessary to derive the needed set of rules.

Another improvement in terms of run time complexity concerns the collision detection. In current implementations every node is collision tested against every known wall segment during the quad-trees creation. But a collision test actually is necessary against those wall segments which have a positive collision test agains the father of this node only. Therefore an additional list of colliding wall segments for every node must be recorded.

This would increase the memory usage. A pay off may be expected on game worlds of larger scales.

Chapter 5

Conclusion

In general the presented system turned out to be a very flexible tool when it comes to approximate arbitrary polygonal shapes (rooms) where initially the shapes boundaries (wall segments) are known only. In particular the practical use in context of a games sound system still is to be proven. The restriction on an axis aligned game world geometry (which is just a matter of the utilised collision detection technique) and the fact that the system neglects diagonally located neighbouring leafs limits the range of games the software could be implemented in.

A still missing link to the sound engine is the automated parameterisation of environmental effects based on the "Dynamic Geometry Middleware's" results. The area and material composition can easily extracted from the presented system which basically is sufficient for setting up a proper reverb effect. But to achieve more sophisticated results and correctly parameterise e.g. a predelay and decay times, further calculation concerning the room's geometrical properties (e.g. the compactness of a region) have to take place.

Since these operations are well explored in the field of "digital image procession" (in relation to binary image labeling) a transfer of these techniques from into the domain of quad-tree regions would be very interesting task and a potential field for further scientific works.

References

Literature

[1] James Boer. *Game Audio Programming*. 1st ed. Charles River Media, Inc, 2003 (cit. on p. 7).

[2] Alexander Brandon. "Audio Middleware – The Essential Link From Studio to Game Design". In: *audioNEXT* (Mar. 2007), pp. 68–70 (cit. on p. 5).

[3] Wilhelm Burger and Mark Burge. *Digital Image Processing – An Algorithmic Introduction Using Java*. 1st ed. Heidelberg: Springer-Verlag, 2008 (cit. on pp. 28, 36).

[4] Christer Ericson. *Real Time Collision Detection*. Elsevier, San Francisco, 2005 (cit. on p. 19).

[5] Thomas Funkhouser. "A beam tracing method for interactive architectural acoustics". In: vol. 2. Journal of the Acoustical Society of America, 2004, pp. 739–756 (cit. on p. 7).

[6] Garin Hiebert. *OpenAL Programmer's Guide – OpenAL Versions 1.0 and 1.1*. Creative Technology Limited. July 2006 (cit. on pp. 6, 9, 10).

[7] Gregory M. Hunter and Kenneth Steiglitz. "Operations on Images Using Quad Trees". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2 (Apr. 1979), pp. 145–153 (cit. on p. 18).

[8] Daniel Peacock. *OpenAL – Effects Extention Guide*. Creative Technology Limited. July 2006 (cit. on p. 16).

[9] Nikunj Raghuvanshi et al. "Real-Time Sound Synthesis and Propagation for Games". In: *Communication of the ACM—Creating a science of games* 50.7 (July 2007), pp. 66–73 (cit. on p. 3).

[10] Hanan SAMET. "Connected Component Labeling Using Quadtrees". In: *Jounal of the Association for Computing Machinery* 28.3 (July 1981), pp. 487–501 (cit. on pp. 19, 29–31).

[11] Loki Software. *OpenAL 1.1 Specification and Reference*. June 2005 (cit. on p. 4).

Online sources

[12] gno.org. *GNU Lesser General Public License*. 2007. URL: http://www.gnu.org/licenses/lgpl.html (cit. on pp. 2, 4).

[13] http://kcat.strangesoft.net/. *OpenAL Soft*. 2013. URL: http://kcat.strangesoft.net/openal.html (cit. on p. 4).

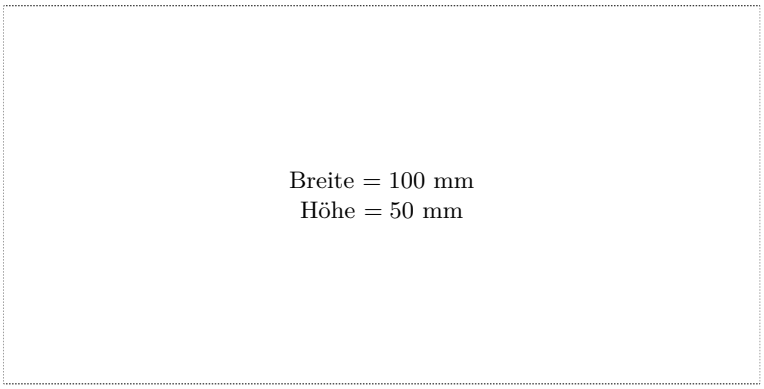
[14] Microsoft. *XAudio2 Introduction*. 2014. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/ee415813(v=vs.85).aspx (cit. on p. 4).

[15] Point Blank Online. *Reverb Parameters Explained in Logic*. Nov. 2011. URL: http://plus.pointblanklondon.com/reverb-parameters-explained-in-logic-pre-delay-attack-decay-diffusion-density-spread-room-shape/ (cit. on p. 8).

[16] W3C. *Web Audio API – W3C Working Draft 27 June 2014*. 2014. URL: http://webaudio.github.io/web-audio-api/ (cit. on p. 4).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —