Pushing Location Based Media Art to the Limit

Technical Challenges and Solutions

yvan vander sanden & Zimcke Van de Staey
Mute, Leuven, Belgium
contact: zimcke@mutecode.com — yvan@mutecode.com

Abstract

The current use of location based sound art for mobile devices is quite straightforward. Honourable exceptions not withstanding, projects often go no further than linking predefined zones to one or more sounds. While this approach is certainly interesting because of its low entry barrier – even the most casual listener will instantly understand the correlation between position and sound – it will not aid us in sustaining interest in this art form for very long.
In our presentation we will discuss a few alternative approaches to sound / location mapping with more surprising and varying results. This can be attained by indirect interpretation of the GPS signal itself – as we have done in our *LineWalk I* application – and also by combining GPS with the multitude of readily available sensors.
Of course there are some hurdles to be expected: implementing sensor based interaction is no small task for the digital artist. And to add to our challenges, the mobile market is still highly fragmented. It's not enough to write your software just for Android or iOS. This is why we propose to piggyback on the game industry: game developers use the same means for a different outcome. While more general software development continues to struggle with delivering Android, iOS and Windows phone versions of their software, cross platform mobile development in particular is already quite mature within the gaming industry. And as far as sensors and interface programming go, there are several game engines available that make it possible with ease. However, there is one thing missing: imaginative audio support. It would be a shame to go the extra mile when it comes to being creative with interaction, while at the same time being limited by a sound interface that will do little more than play sounds at certain locations.
In this matter, the options are limited. LibPD for one can be a useful tool, but it feels a bit like programming your interface directly in openGL. A more high-level sound engine preferably one that combines a 3D positioning system, DSP processing, software synths and composition tools would be welcome. We will finish our presentation with a modest proposal in this direction.

Contents

Introduction **2**
1 Sound to Location Mapping **2**
1.1 Relative Mapping 2
1.2 Data Communication 3
1.3 Not All is New 4

2 Technical Challenges 5
2.1 Cross Platform Frameworks 6
HTML and Javascript Based Frameworks • Precompiling Frameworks • Native Mode Frameworks
2.2 Game Design Frameworks 8
Esenthel Game Engine
3 Audio 9
3.1 Imagination 10
3.2 a Wishlist 10
3.3 the YSE Sound Engine 12
4 Conclusion 13

Introduction

Over the past several years, we have seen many new location based sound art for mobile devices. This led to an extensive research on the artistic possibilities of the ubiquitous sensors available on these devices.

Notwithstanding those efforts of integrating the sensors in mobile music applications and the numerous writings on this subject, the current sound to location mapping approaches are quite straightforward. The concept used by many applications often solely consists of linking predefined zones to one or more sounds. We have made two such applications for Musica[1]: *Curvices*[2] (2013) and *Klankroute Maasland*[3] (2014). And while this is a fair choice for projects in the touristic domain, there are numerous other designs possible.

1. Sound to Location Mapping

Linking sounds to zones has the advantage that, from the perspective of the audience, it is very easy to understand what is going on. Current GPS technology on mobile devices is also accurate enough to implement this with reasonable detail.

On the other hand, the 'direct' approach does not give the user much of a challenge. There is no room for discovery or surprises, both of which are an important part of artistic experiences.

1.1 Relative Mapping

More innovative ways to map sounds can be realised by interpreting the data in a relative, indirect manner. The GPS sensor can be used to track the movement of the listener instead of focusing on the exact location. As an experiment, we did develop an interactive application that is based on this concept: *LineWalk I*[4] (2013), as seen in figure 3. The app is a seven- to ten-minute composition, during which the listener can pace back and forth. He is supposed to "walk on a line", such as when he or she is waiting for the bus. The macro structure of the piece is entirely precomposed, but new sounds will appear on different locations on the screen. The listener can decide whether or not to follow these sounds.

To increase the interactive possibilities in *Linewalk I*, we implemented a small software synthesizer. This greatly improved the flexibility of sound generation on a micro level. Because the idea was to experiment with different types of user interaction, we implemented the following concepts throughout the three parts of the composition:

- pitches of continuous sounds can shift by walking further from the starting point;

[1] http://www.musica.be/
[2] http://www.musica.be/en/curvices-rozalie-hirs-music-poetry-voice-cox-grusenmeyer-design-animation
[3] http://www.klankatlas.eu/routes/klankroutemaasland/
[4] https://play.google.com/store/apps/details?id=com.mute.linewalkI

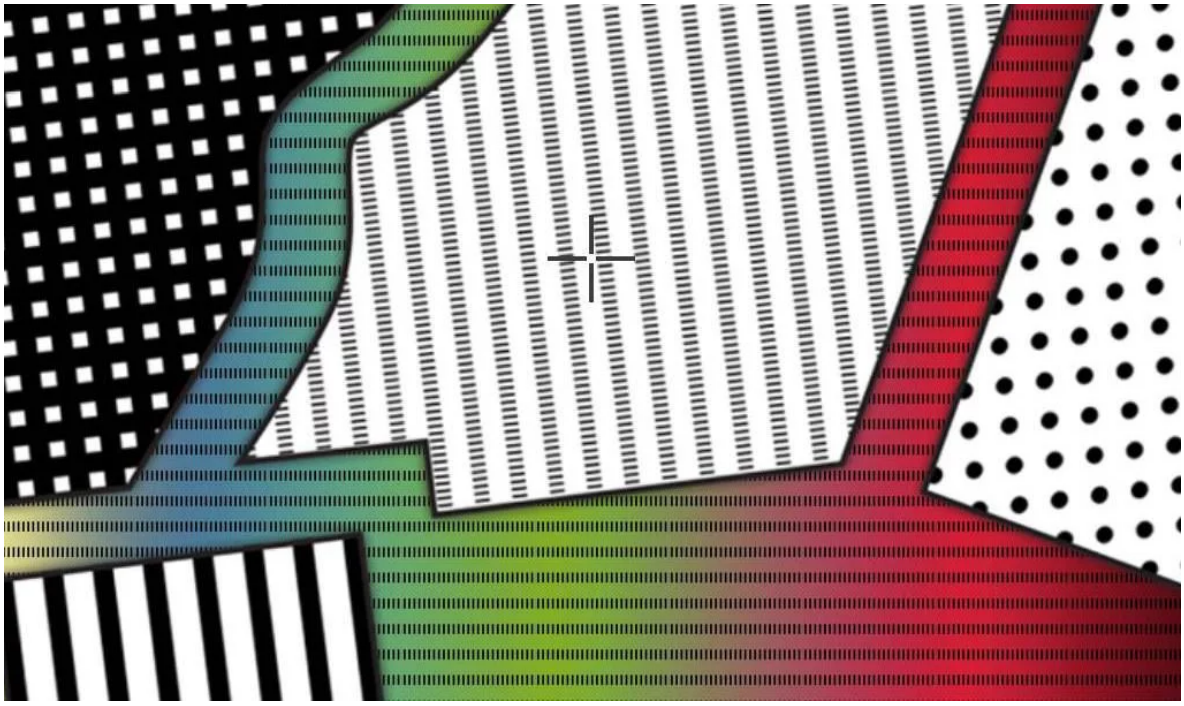

Figure 1: Screenshot from Curvices

- areas in which a specific sound is triggered can grow when the listener stays within that area;
- sounds can be changed or moved to another position when the listener has them activated for a certain period in time;
- sound triggers can be numbered, thereby forcing the listener to engage them in a particular order;
- different layers of the composition can be activated by standing on a specific location;
- sounds can be placed in the stereo field according to their visual appearance on the screen.

By combining all these concepts together, we were able to create a complex set of possible interactions for the listener. But while this enlarges the reusability of the app, there is the danger that the interaction will be unclear to the listener. We tried to counter this in two ways:

1. Progress will never halt completely. The user cannot get 'stuck' when he does not understand the interaction. Instead the music will continue and the interaction will *(hopefully)* become clearer over time. Whenever the listener comes to understand a specific method of interaction, he will be encouraged to start the music again and this time have more impact on the way the composition evolves.
2. All interactions are visualized in a straightforward manner because most people are far better in interpreting images than sounds. By ensuring a strong link between the audible and the visual content, the listener is hinted at the possible result of his decisions.

1.2 Data Communication

Other ways of extending location based media art can be realised by using the wireless

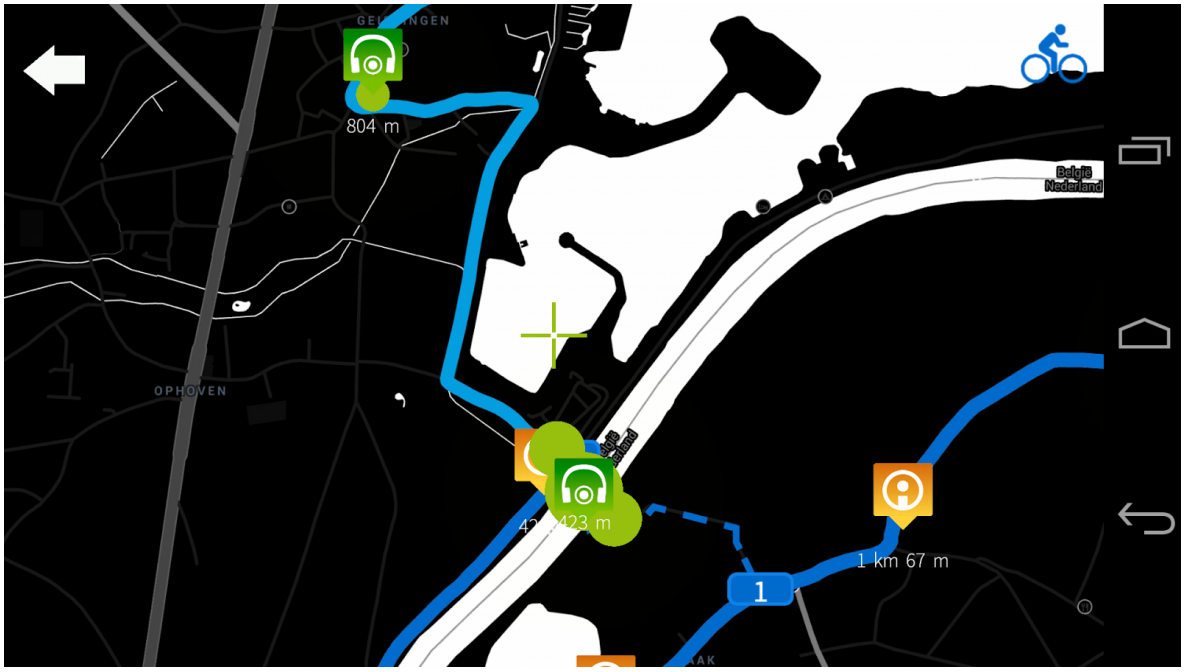

Figure 2: Screenshot from klankroute maasland

data network on mobile devices. Networked applications can exchange information with a server-side component and adapt their behaviour accordingly. This gives artists the ability to define flexible sound works instead of pre-composed content. Above all this is an interesting way to create interactivity for the listeners. An idea that we would like to work on is an app with movable sound objects, where a listener can virtually "pick up" a sound from its original location and release it somewhere else. When locations are stored online instead of on the listener's device, this approach enables more lasting interactions with the environment. And because these interactions are visible to all other listeners, they have more meaning. It does not have to end with sound objects either: the same approach could be used for text, images or even audio filters.

The approach above can be described as a closed circuit. A server on the network interacts with all client applications to exchange information. Networked applications can also make use of more general data that is available on the internet. The obvious example would be the current weather conditions. Other ideas could be the location of nearby landmarks, altitude differences, population density, vegetation, or basically every kind of location based information that is available online. For the most part, application programming interfaces for this kind of information already exist.

In the near feature, we think that working with so called 'big data' will also be achievable. Possibly an application could also use the data about the user itself, which is already gathered by companies like Google and Facebook[5].

1.3 Not All is New

Creating a meaningful musical experience with this data might be a challenge. But when we look for ways to interpret all this data, we're not really entering uncharted territory. An inventory of the possibilities is be-

[5] After all, advertising companies do it, the NSA does it, our governments do it, so why shouldn't we?

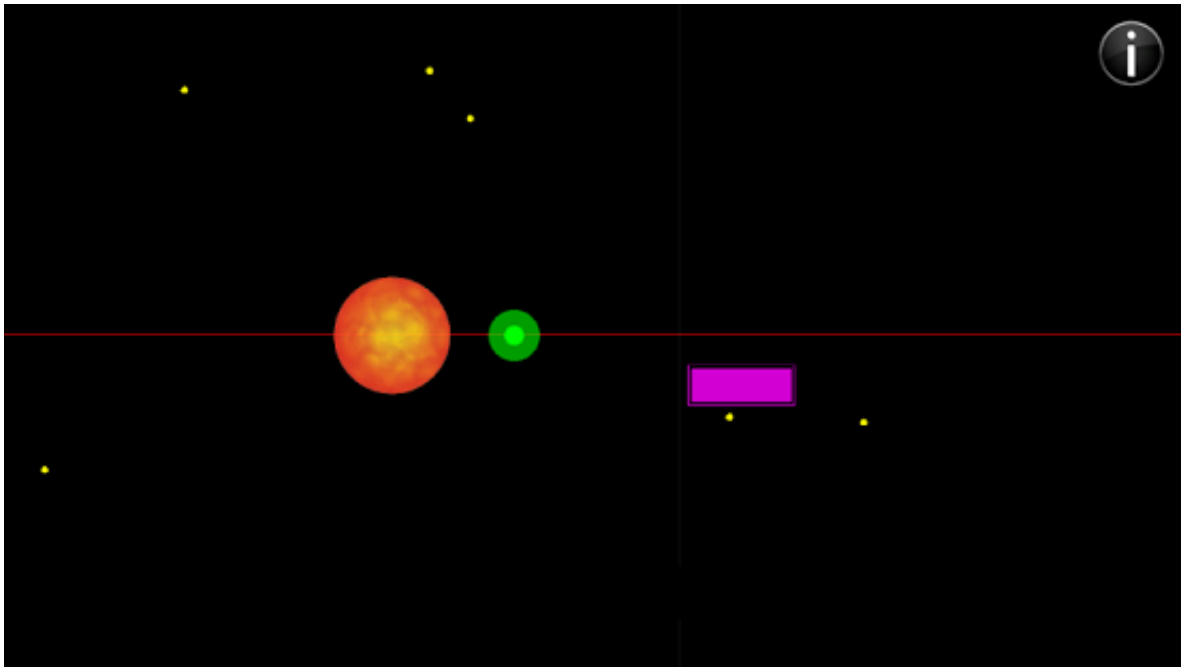

Figure 3: Screenshot from Linewalk I. The green dot symbolizes the position of the listener.

yond the scope of this paper, but they are actually not that different from the more established practise where composers map data from (movement) sensors to musical expression. As an example we'd like to refer to the work done by Kristof Lauwers and Godfried-Willem Raes at Logos foundation[6].

2. Technical Challenges

Developing mobile applications often results in developing as many applications as you support platforms. Not only do these platforms use different programming languages, they also don't always offer the same possibilities, at least not at programming level.

In Tonal Tools[7], we needed to visualize the notes played on a keyboard while playing back a midi file, as seen in figure 4. While the result looks more or lest the same on both iOS and Android, we needed a very different implementation. On Android, we could use a simple AudioPlayer (check this name) object to play back a midi file. But once it is playing, there is no way to keep track on note on/off events. We had to write our own midi file parser, extract all note on/off information and synchronise that with the AudioPlayer object.

IOS on the other hand provides us with a callback function which is called every time a note on / note off event happens. This makes the visualisation of that event very easy to implement. But setting up a system to actually play the midi file is not straightforward at all. There is no object like the Android AudioPlayer to just play the file. IOS does not even have default midi sounds: these have to be provided by the software.

This small example shows us just how different both systems really are. It's not enough to just translate code to another language: often it takes completely different code to create an application for both platforms.

But even within one platform things are

[6] http://logosfoundation.org/scores_gwr/Namuda_Links/namuda_studies.html
[7] An application for iOS and Android that we've created for Musica

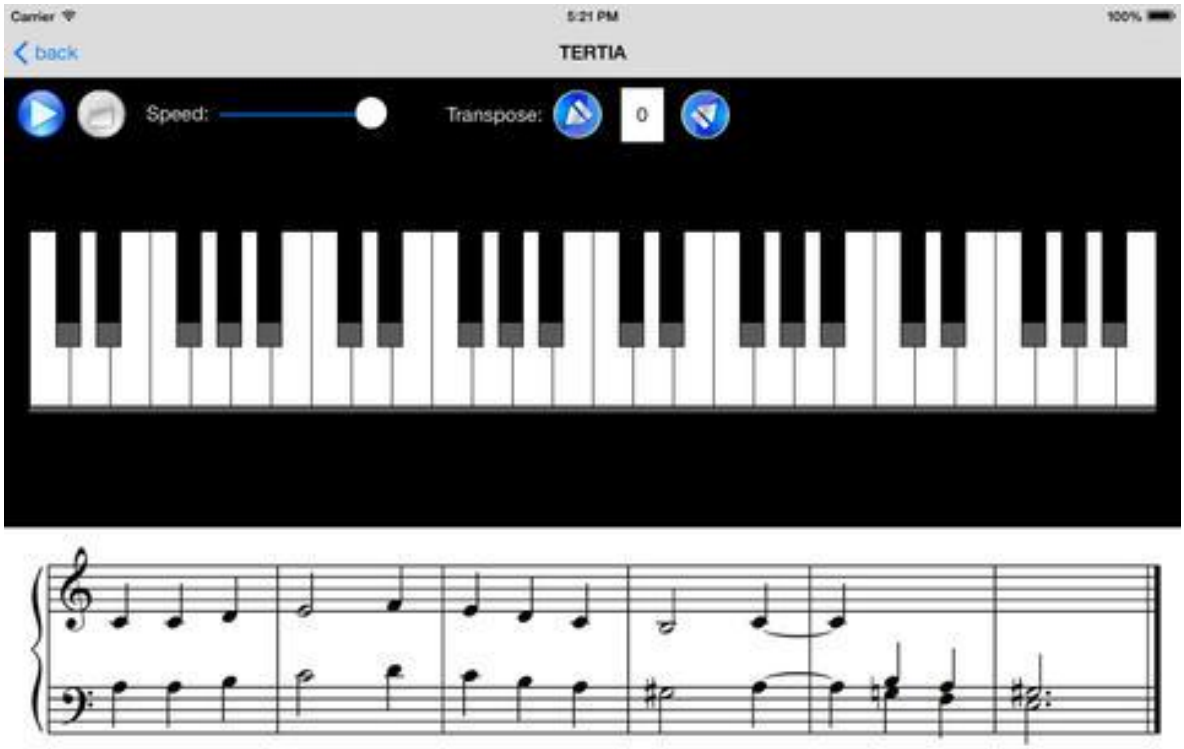

Figure 4: Tonal Tools includes a keyboard display which is synchronized to a MIDI file player.

not always the same. One version might require a piece of code to be different from another version. When Apple introduced retina displays, developers were required to adjust their interfaces to the new screensize. Android als made a lot of changes to their geolocation system in Kitkat. Most of them are backwards compatible, but not all. At the very least, your code will not be optimal if you don't review those changes.

Supporting several platforms and keeping track of changes in all of them takes a lot of time. Time we'd prefer to spend otherwise, I am sure. Which is why Mute opted to use a cross-platform development framework. We prefer that someone else keeps track of all those differences and changes, so that we can focus on the actual application development.

2.1 Cross Platform Frameworks

Given the difficulties and extra workload it takes to develop applications for multiple (mobile) platforms, it is no surprise that cross-platform development frameworks are getting a lot of attention nowadays. To give complete listing of available frameworks is beyond the scope of this article. Also, our list would surely be outdated in a few months. Still, a brief overview of the most important options might be helpful.

2.1.1 HTML and Javascript Based Frameworks

With the advent of HTML5 and other recent technologies such as jQuery, modern websites can behave more and more like real applications. This is why some cross-platform frameworks do not really create an application. All the functionality is created inside a collection of webpages is packaged so that it can be viewed locally on a mobile platform.

While this can make development easier, it also has some important drawbacks. Generally speaking, the available functionality is quite limited compared to native applica-

tions. The application does not automatically have access to all the features of the platform.

Equally important in the context of this paper is that this approach creates really slow applications. This can be a problem if you want your audio to do more than simply start and stop playing. *(A slow application does not mean that your CPU is playing golf half of the time, it means that it needs a lot more time to do calculations. Therefore, slow applications drain your batteries faster.)*

2.1.2 Precompiling Frameworks

Another approach is to develop in a generalized environment which automatically generates the correct code for several platforms. Once you compile that code into a program, you have an application for the supported platform. This has the advantage the result is native code, which performs a lot better than JavaScript. The user interface elements will often translate to an iOS element on iOS and an Android element on Android.

While this seems like a good idea there is, again, one drawback. If your interface elements look like the native ones, users will also expect the application to work like a native application. But there are a lot more differences between platforms than just the shape of the buttons. With iPhone tabs go on the bottom, but Android tabs are on top. And Windows 8 scrolls everything horizontally instead of using tabs[8]. The implication is that you will end up designing a separate interface for every platform, which is exactly the thing you're trying to avoid.

Another drawback is that the not so expensive options in this category often take their time when the target platform changes its requirements. This may leave you with no option to publish your app in the store. More expensive frameworks do not have this problem but are, well, expensive.

2.1.3 Native Mode Frameworks

Here's when it gets confusing: mobile development uses the word 'native' in two different contexts. When we discussed the idea 'native development' before, we meant development with the tools for that particular platform, like x-code for iOS or eclipse for Android. But both platforms also have a 'native mode'. This is the low level software layer which operates below the high level objective C *(iOS)* or Java *(Android)* interface. As an example we will briefly show the difference with figure 5: a diagram of Android's internal design.

Android applications (in blue) are written in Java, but they rely on the application framework and libraries written in C++ code. Native mode applications only have a placeholder in Java while the real core of the program sits on the same level as the library (in green). While the native mode doesn't give you easy access to the application framework provided by the SDK, it does have one advantage: you can use the same programming language on both Android and iOS. This means a library can be written that handles the internals differently while still giving the programmer a single interface, without any performance loss. And because the developer can use c++ to create the app, it is also easy to include other c++ libraries.

A simple function to get the current longitude on your location might look like this:

```
double getLongitude() {
  #if defined ANDROID
    return AndroidsLongitudeFunction();
  #else
    return ApplesLongitudeFunction();
  #endif
}
```

The application developer only has to know that the function 'getLongitude()' exists and that it will return the current longitude. Whatever happens internally is up to

[8] http://www.netwhisperer.com/2013/03/27/mobile-app-development-native-vs-cross-platform/

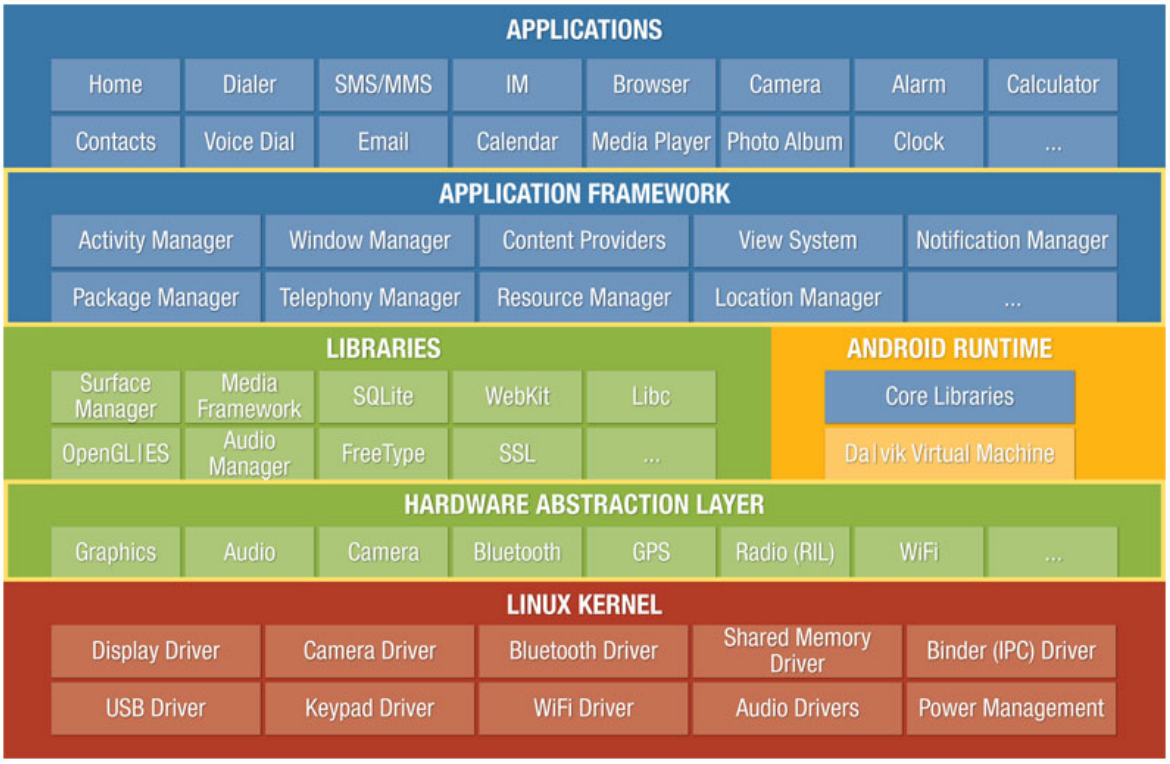

Figure 5: The Android architecture[9]

the maintainer of the library.

These native applications have no access to the standard user interface objects in Android or iOS, but we think this is actually an advantage. A problem only arises when cross platform frameworks try to mimic the standard interfaces, in which they never succeed entirely. Doing so creates an expectation from the user: an application that looks like a standard application should behave like one in all circumstances. But in contrast, if the user interface is entirely different users will not expect it to behave like a standard application.

Both performance and flexibility are important for the creation of mobile media art. This is why we think that the advantages of using a cross-platform native mode framework greatly outnumber the main disadvantage, namely that access to the application framework is more difficult. (Note that it is not entirely impossible!)

2.2 Game Design Frameworks

Native mode frameworks are especially popular with game designers. This should come as no surprise: only native mode applications are capable of accessing the graphics processing unit (GPU) directly. Therefore they are the only way to create demanding visual software, which is what games often are. Game engines have been supporting cross-platform mobile development for a few years now. And while direct GPU access might not be overly important to all media art, there are several other advantages to game engines from which we might benefit:

- Game development depends a lot of importing resources, like images and sounds. They mostly have a very efficient pipeline for this which really saves time during development.
- Both Android and iOS support OpenSL ES in native mode, which is a mobile so-

lution for playing and mixing sounds that is not available through the Application framework.

- Because of the steep competition, game designers really want to use the latest technologies. While more general frameworks might take a while to implement a new features, this is often done within a matter of weeks by game engines.

- Game engines are a highly competitive market. Only a few games really generate a lot of income, and the engine developers know this. While a cross-platform framework for business applications often cost a lot of money, game engines are relatively affordable.

As sound artists we are naturally most occupied with the audio side of what we create. But we should not forget that our product is also a visual one. Whether we like it or not, the is this GUI, the graphical user interface.

Game designers, always creating a unique 'feel' for every game, have understood this very well. When using the standard Android or iOS interface, it is easy to create an interface that looks OK. But when you want to create something unique, be it a game or an art product, OK is not enough. An application should engage a user not only with game mechanics or sound, but also through its interface. Neglecting the visual side of a product is a missed opportunity, we think.

2.2.1 Esenthel Game Engine

At Mute, we created quite a few mobile applications with the Esenthel game engine[10]. There are other game engines which are better known, but we have quite a few reasons why we keep using this one:

- The Application Programming Interface is very logical and well documented. Once you're familiar with the basics, development in Esenthel is really fast because of this.

- The built-in code editor has some very interesting auto-complete functions and shows relevant documentation while you type. It even creates your header files for you, takes care of pointers, includes and external declarations. Again, this greatly increases our development speed.

- Everything you do is automatically synchronized over the network. This means you can work on a windows PC for hours and after that open up your Macbook and test your project on iPad in a matter of minutes.

- Assets *(sounds as well as images, video and 3D models)* can be imported with drag and drop. They will automatically be converted to a usable format. Once imported, they can even be dragged onto your code wherever you want to use them.

- Because code is written in C++, almost every other C++ library can be added to your project. If you have a source license you can even add the functionality you need directly to the Esenthel engine.

- Support is really good. Almost every time we had a question, a solution was given by the engine developer within a day. Requests for (small) new features are often handled within a few weeks.

- The price is the lowest we could find. A complete source license costs only 13.75 Euro a month. The binary license is only 6.88 Euro a month.

3. Audio

A game engine is a good option for cross-platform mobile development. It even enables us to use the most flexible audio solution there is on mobile platforms: OpenSL ES. Unfortunately, this most flexible solution is not very flexible at all. Here are some limitations:

[10] http://www.esenthel.com

- A hard limit of 32 simultaneous sounds. This is not much once you start to use sounds as a software sampler. Even a monophonic instrument needs at least two or three sounds to sound natural.

- Sound playback speed is limited from 0.5 to 2.0. This means you cannot create a software sampler from one sound and have it cover more than 2 octaves.

- Reverb is not available. We certainly don't want to add a reverb to everything, but it can be useful.

- No MIDI support. Although OpenSL ES also defines MIDI support, the Android implementation lacks MIDI.

On top of that, the OpenSL ES interface is unwieldy at least: it doesn't come with lots of examples and only has a primitive C API. The latter makes your code particularly prone to errors. Because of this we would recommend the same strategy as we used for targeting multiple platforms: let someone else worry about it. Or at least we don't want to worry about it while we're working on an application.

3.1 Imagination

It is our view that art is the realisation of our imagination. And while there will always be limits to what is possible, we feel that the current state of sound programming interfaces really limits what we can realize. Hardware development has taken an enormous leap over that last years: we can do more on a recent phone than was possible on a computer fifteen years ago.

In general, software development kept up with that. The game engines mentioned earlier in this article all provide an easy to use and flexible interface to create software that uses graphics in new and fascinating ways. Unfortunately, sound doesn't seem to be very important to most companies. Understandable too, because most sound artists won't request much more than playing prerecorded audio tracks.

To improve the current situation, sound artists should think of new ways to work with sound. But as long as the technology doesn't allow for real creativity, this won't happen. A much needed improvement of audio programming possibilities should go hand in hand with sound artists working with those new possibilities.

3.2 a Wishlist

So what exactly would be the ideal sound library? What features should it contain? As a programmer creating audio art, we think that were in a good position to imagine what wed like to work with.

Firstly, there are some general programming requirements:

- It should be fast. There is no use for a sound engine which takes up half of the available CPU power. Preferably all DSP calculations are divided over all available processors. This speed requirement is important, because it rules out all programming languages with automatic garbage collections.

- It should work on all major platforms, without application code modifications. At the very least on Windows, Linux and Android. Mac and iOS would be nice also, for as long as they still have a market share.

- It should be understandable without looking at the documentation all the time. If you look at a function's name and it's not clear what it's supposed to do, the function name is badly chosen. On the other hand, documentation must be extensive and there should be plenty of tutorials.

- Frequently used concepts should be ready for use. I do not want to be able to create a looping sound object that can be played at variable speed, I want to be there.

The same goes for mixing channels, reverb and most established DSP functions. You would think this goes without saying, but everyone who has used Pure Data knows that even for starting and pausing a simple sound you have to put in an extra ramp object to avoid audio clicks. Since we all need to do that, I think it would be better to implement that sort of thing right away, at the engine level.

- It should be easy to use right, and difficult to use wrong.

While the last one is an established programming mantra, Were still surprised how complicated most existing sound engines are. As an example, here is the recommended way to initialize the FMOD sound engine:

```
UInt version;
_errorCheck(_system->getVersion(&version));
if (version < FMOD_VERSION) Exit("FMOD: outdated fmod dll.");

// get the number of available audio drivers
_errorCheck(_system->getNumDrivers(&_numDrivers));
if (_numDrivers == 0) {
  // continue without sound
  _errorCheck(_system->setOutput(FMOD_OUTPUTTYPE_NOSOUND));
} else {
  // revert to default driver if the ID is not right
  if (driverID > _numDrivers) driverID = 0;

  // Get driver info
  _errorCheck(_system->getDriverCaps(driverID, &_caps, 0, &_speakermode));

  // Set speakermode
  _errorCheck(_system->setSpeakerMode(_speakermode));
  if (_caps & FMOD_CAPS_HARDWARE_EMULATED) {
    _errorCheck(_system->setDSPBufferSize(1024, 10));
  }

  // something wrong with sigmatel audio drivers
  _errorCheck(_system->getDriverInfo(driverID, _driverName, 256, 0));
  if(strstr(_driverName, "SigmaTel")) {
    _errorCheck(_system->setSoftwareFormat(48000, FMOD_SOUND_FORMAT_PCMFLOAT, 0, 0,
        FMOD_DSP_RESAMPLER_LINEAR));
  }
}

// try initializing the sound system
FMOD_RESULT r = _system->init(1000, FMOD_INIT_NORMAL, 0);

// if something went wrong, revert to play stereo mode
if (r == FMOD_ERR_OUTPUT_CREATEBUFFER) {
  _errorCheck(_system->setSpeakerMode(FMOD_SPEAKERMODE_STEREO));
  _errorCheck(_system->init(1000, FMOD_INIT_NORMAL, 0));
}

_system->set3DSettings(_dopplerscale, _distancefactor, _rolloffscale);
```

This should be enough to scare away everyone but the most proficient programmers. And even those will have to program with a manual on a second screen. We think it should be something like this:

```
SOUND::System.init();
```

If a programmer is not satisfied with the default options, he should be able to change those. Reasonable defaults go a long way though. If, like in the example above, something is known to be wrong with sigmatel audio drivers, this should be handled by the engine, not by the application programmer.

With these programming requirements in mind, we can move on to the audio features. In general, there are several categories of sound manipulation that can contribute to the way we'd like to work with audio:

- 3D sound positioning. We would like to position and move sounds in a virtual world. This can be interesting for location based sound art, but it also opens up possibilities for many other concepts we're working on. In general, this is what sound engines for games use.
- Low level DSP functionality. We'd like to be able to manipulate every audio stream at sample level. Pure Data is a good example of software to do this kind of thing.
- Software synths and MIDI support. While sound positioning and manipulation is very interesting, we still believe in motives, melodies and chords. (Although not necessarily in a conventional way.) The idea of a playing instrument which you can give instructions still has merit. Sequencers and dedicated software synths aside, this functionality is non existent in most sound software.
- Macro structure (or composition) functions. Most sound libraries are occupied with the 'here and now'. We think it could be very interesting to include functions that define how sound evolves on a larger scale, directly into the sound library. This kind of functionality can be found in programs like GMT and the Composers Desktop Project.

Needless to say we have not found a sound library that can do all of this. And while it is not impossible to work with several libraries together in your software, doing so is quite complicated. You will need to compile every library for all targeted platforms, compile their dependencies and write a lot of functions just to pass data from one library to another.

Nonetheless, when new and innovative audio art is our goal, we will gain a lot from an easy to use library which has all this to offer.

3.3 the YSE Sound Engine

Last year we started development on a new sound engine: YSE. With the list above in mind, we think that even a partially completed engine can help us create audio in ways we are not able to do today.

Since we want a fast library, but with an easy to use syntax, we didn't write it in C. While C does the fastest code in most circumstances, it isn't the easiest language for everyday use. As a compromise we used C++11. This allows us to use C code for the most crucial parts of the engine, while still providing a solid and easy to use interface.

Because we are also working on the Attr-X project (a virtual online world about sound manipulation) we started with 3D sound placement and sound synthesis. At the end of the year we had a first stable version available which is already implemented in Attr-X.

This first version of our library only supports Windows, Mac and Linux. But now that we get more and more involved in mobile projects we made a lot of changes to have it compile on Android and iOS. This stage is also nearly done. We have not published a stable release just yet, but that is a matter of a few months at most.

Right now, the following features are implemented:

- 3D positioning of all sound objects and effects.
- Nearly all functionality provided by commercial sounds engines for gaming. (Doppler effect, reverb, rolloff scaling, sound virtualization, etc.)
- Flexible speaker setup. Contrary to gaming solutions we are not only interested on playing audio on a home surround system. Audio installation art can require any possible speaker setup. With YSE you can adjust all speaker positions. All 3D sounds will be outputted to the correct speakers, according to their position.
- Basic DSP functions, comparable to the vanilla version of Pure Data. DSP objects can be used as a sound source in a 3D environment, as well as a sound filter attached to a sound, a channel or a location in 3D.
- Full resource management. You don't have to worry about acquiring and releasing memory. Also, sound files that are no longer used will be erased from memory by the engine.
- Audio calculations are spread out over multiple cores. The library is completely lock free for the best performance.

Some stress tests were done on a Windows 8 pc with a quad core i7 processor. With this set-up the engine can render about 2500 sounds simultaneously[11]. With virtualization turned on, we can have 80.000 sounds in memory, moving around in a 3D space and have the engine play the 100 most important ones. On a Nexus 5 mobile phone with android 4.4, we were still able to play about 700 sounds at the same time.

While our implementation is far from complete we do think some of the hardest work is done now. We will continue to work on more advanced functionality, adding software synths and macro functions, but at the same time, we expect to release our first software based on the engine this year.

YSE is open source, released with the Eclipse license. This means it can be used by other open source projects, as well as in commercial products. We hope that by sharing the source, contributions and improvements will follow over time.

4. Conclusion

Location based media art is emerging for a few years now. We think that projects often use only a few of the available techniques to create an engaging user experience. If we want to keep our audience's attention, we will have to keep innovating. But integrating new technologies, complicated interactions and engaging interfaces requires a lot more programming. On top of that, the audience is divided over several platforms, the most important ones being Android and iOS right now. This too requires extra work from the programmer.

Because of these reasons we think it is best to use an integrated, cross-platform toolkit for development. And although none of them is specifically intended for art projects, game engines offer much we can use.

If a game engine can be used in combination with a sound engine that opens up new ways of working with audio, artists will be able to create even more new and innovating mobile art projects. And because such an engine does not exist, we've set a modest first step in developing one, hoping to inspire others to work with us.

[11] Of course we never intend to actually play that much sounds at the same time. This merely shows that there is plenty of headroom to do other calculations in our software.