Le language audio pour mobiles MAUDL et son moteur de rendu audio interactif IXE

Yohan Lasorsa, Jacques Lemordant

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Le language audio pour mobiles MAUDL et son moteur de rendu audio interactif IXE

Yohan Lasorsa, Jacques Lemordant

N° 7847

Décembre 2011

Représentation et traitement des données et des connaissances

ISSN 0249-6399

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche GRENOBLE - RHÔNE-ALPES

Mobile Audio Language (MAUDL) and its Interactive eXtensible Engine (IXE)

Yohan Lasorsa[1], Jacques Lemordant[2]

Représentation et traitement des données et des connaissances
Equipe-Projet WAM

Rapport de recherche n° 7847 – Décembre 2011 - 40 pages

Abstract: Building a navigation system only based on audio guidance is a very complex task to carry out. You need to provide the user enough information to guide him without flooding him under an heavy load of sounds. Multiple kinds of guidance clues must be provided without overloading the auditive space. You also have to sort informations to give the user the most pertinent one at a given time. Finally, the system should be able to guide the user precisely in all kinds of environments. Based on these objectives, the Mobile Audio Language (MAUDL) has been defined in this research work, after a review of the limitations and problems existing with the current formats within a navigation context. Customization of the audio rendering is one aspect showcased by the usage of the different features available in MAUDL. In addition, a new sound manager named Interactive eXtensible Engine (IXE) has been developed to provide a software support to the language. It integrates all the current features of MAUDL and has been specifically designed for mobile platforms. This research report details the various problems encountered while developping such a system and the technical decisions that led to the conception of this library.

Keywords: interactive audio, guidance, 3D spatialization, augmented reality audio, XML format

[1] INRIA – yohan.lasorsa@inria.fr
[2] UJF-INRIA-LIG – jacques.lemordant@inria.fr

Centre de echerche INRIA Grenoble - Rhône-Alpes
655, Avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 – Télécopie : +33 4 76 61 52 52

Le language audio pour mobiles MAUDL et son moteur de rendu audio interactif IXE

Résumé: La construction d'un système de navigation basé uniquement sur l'audio est une tâche complexe à mettre en œuvre : il faut pouvoir indiquer un nombre suffisant d'informations à l'utilisateur sans le noyer dans une foultitude de sons, être capable de fournir plusieurs types d'indices de guidage sans surcharger l'espace auditif, savoir trier l'information pour fournir l'information la plus pertinente à un instant donné et enfin pouvoir diriger l'usager de manière précise dans divers types d'environnements. C'est dans cette optique que le format Mobile Audio Language (MAUDL) a été défini au cours de ces travaux de recherche, après avoir abordé les limitations et problèmes posés par ce type de contexte appliqué aux formats existants. La personnalisation du rendu audio de guidage est notamment mise en avant par l'utilisation des différentes fonctionnalités du format. De plus, afin de fournir un support logiciel au format, un gestionnaire audio nommé Interactive eXtensible Engine capable d'intégrer les fonctionnalités actuelles et futures du langage a été développé pour les plateformes mobiles. Ce rapport détaille notamment les diverses limitations rencontrées et les choix techniques effectués pour la conception d'une telle librairie.

Mots clés: audio interactif, guidage, spatialisation 3D, réalité augmentée audio, format XML

1 Introduction

Dans le contexte d'un projet destiné à développer des outils basés sur la haute technologie pour améliorer l'autonomie des personnes qui ont un handicap, la navigation est une composante majeure. Utiliser des composantes audio pour réaliser le guidage est le meilleur moyen de viser un public et des usages très larges tout en restant capable de fournir des informations précises et adaptées, à fortiori lorsque les personnes sont malvoyantes.

Représentant le cœur de ce projet, notre travail de recherche sur l'audio a pour but de définir un langage de description de guides audio pour la navigation dans des bâtiments et de réaliser un navigateur embarqué pour la navigation dans ces bâtiments. L'objectif est de permettre la création d'un web de navigation indoor pour personnes en situation de handicap visuel, permettant ainsi de télécharger le guide audio d'un bâtiment avant de s'y rendre ou à l'entrée du dit bâtiment. On essayera d'être la plupart du temps indépendant d'une infrastructure préexistante dans le bâtiment de façon à accroitre l'autonomie de la personne et l'usage du système. L'accent sera mis sur la réalisation d'un guidage audio-vocal de haute qualité et du type réalité augmentée. Le guidage sera interactif, permettant d'émettre des requêtes sur l'environnement proche ou les espaces constitutifs plus lointains.

Afin d'aboutir à ce résultat, l'objectif de ces travaux est de construire un gestionnaire de sons avec spatialisation 3D, mixage et rendu environnemental recevant des événements en provenance du gestionnaire de positions et d'interrogation de l'environnement. Ce gestionnaire lira des documents dans lesquels se trouveront les spécifications audio associées aux événements reçus. Ces documents seront personnalisables par l'utilisateur et ainsi adaptables aux besoins de chacun.

Ce document présente en particulier les résultats des développements effectués dans le cadre de ces travaux. Il présente les différents aspects du format audio adapté à la navigation et au guidage qui a été défini, ainsi que son utilisation à travers un gestionnaire de sons qui fournira le rendu audio final. L'utilisation des différentes fonctionnalités du format ayant pour but la personnalisation du rendu audio de guidage est notamment mise en avant, au travers de différents exemples d'utilisation du langage de description audio.

Des informations additionnelles ainsi qu'une partie des logiciels décrits dans ce document sont accessibles librement sur le web :

- Audio interactif, sur le site de l'équipe WAM :
  http://wam.inrialpes.fr/iaudio
- GForge des outils destinés à l'audio interactif :
  https://gforge.inria.fr/projects/iaudio

Le reste de ce document est organisé en deux sous-sections, traitant chacune respectivement du format défini pour le guidage audio et de son implémentation au travers d'une libraire disponible notamment pour la plate-forme mobile iOS d'Apple[1].

2 Format pour le guidage audio

2.1 Motivations

La construction d'un système de navigation basé uniquement sur l'audio est une tâche complexe à mettre en œuvre : il faut pouvoir indiquer un nombre suffisant d'informations à l'utilisateur sans le noyer dans une foultitude de sons, être capable de fournir plusieurs types

d'indices de guidage sans surcharger l'espace auditif, savoir trier l'information pour fournir l'information la plus pertinente à un instant donné et enfin pouvoir diriger l'usager de manière précise dans divers types d'environnements.

Un tel système comporte divers challenges à relever, et notamment le fait que pour être optimal, un tel système de guidage audio se doit d'être adaptable facilement aux besoins particuliers de la personne qui l'utilise. Dans cette optique d'adaptation, un format d'échange devient alors indispensable pour faire le lien entre le gestionnaire de cartographie et de position de l'application de guidage et le gestionnaire de sons.

Si l'on analyse le problème de navigation audio d'un point de vue technique, il a toutes les caractéristiques d'une application de type audio interactif. D'un point de vue général, ce système a pour but de construire une bande son dynamique capable de réagir à de multiples événements au cours du temps, ce qui correspond bien à la définition d'audio interactif. Dans un premiers temps, nous allons donc nous attarder à analyser les formats déjà existants dans ce domaine et voir quels sont leurs points forts et leurs points faibles dans une optique de guidage. Nous allons ensuite définir les objectifs qu'un format audio destiné au guidage doit pouvoir atteindre, pour enfin décrire le format que nous allons utiliser dans le cadre de ce projet.

2.2 Audio interactif et navigation

2.2.1 Formats existants et limitations

Dans le domaine de l'audio interactif, de nombreux travaux ont été effectués depuis des années, mais sans jamais vraiment aboutir à un format d'échange universel, reconnu et surtout implémenté et utilisé. En 1999, le groupe de réflexion *Interactive Audio Special Interest Group* (IASIG) a publié les bases attendues d'un système audio interactif à travers les recommandations *Interactive 3D Audio Rendering Guideline Level 2.0* (I3DL2)[2]. Ces spécifications précisent d'une manière générale ce qui est conseillé d'implémenter dans un tel système et quelles sont les méthodes de spatialisations globalement reconnues pour le rendu audio en 3D. Plus tard en 2005, la branche *Synchronized Multimedia Activity* du World Wide Web Consortium (W3C) a développé le format XML nommé *Synchronized Multimedia Integration Language* version 2.0 (SMIL 2.0)[3] pour la coordination de présentations multimédia ou l'audio, la vidéo, le texte et les images sont combinés en temps réel. Ce format a ainsi défini un standard en ce qui concerne la synchronisation d'éléments multimédia en langage XML. Enfin, plus de huit années après la publication des recommandations ID3L2, le groupe IASIG a annoncé l'achèvement d'un nouveau format de fichier pour l'audio interactif pour compléter les ID3L2. Ce nouveau format, basé sur le standard ouvert de format de fichier XMF, est appelé *Interactive XMF* (iXMF)[4]. Le but du groupe IASIG au travers du développement de ce format est de mettre le contrôle de la partie artistique aux mains des artistes, en évitant aux programmeurs à devoir prendre des décisions artistiques, d'éliminer le besoin de portage sur de nouvelles plates-formes et enfin de réduire les temps, coûts et stress de production.

Le format iXMF présente de nombreux atouts sur ses principes, mais souffre de plusieurs lacunes problématiques sur sa forme : étant un format de fichier binaire, il est n'est pas prévu pour être extensible et peut difficilement s'intégrer à d'autres documents. De plus, le format étant très complexe dans sa structure avec des possibilités de scripting audio avancées, il n'existe à ce jour aucune implémentation de référence, à fortiori sur plate-forme mobile. La seule implémentation connue à ce jour serait intégrée au SDK de la console PS3 de Sony selon une annonce officielle, mais aucune indication sur le degré d'implémentation du format n'est disponible, toutes les informations sur ce SDK étant protégées sous accord de non-divulgation.

Depuis 2009, un format XML pour l'audio interactif sur mobile est développé au sein de l'équipe WAM et se nomme *Advanced/Audio Mobile Markup Language* (A2ML)[5]. Il se base sur les principes de iXMF pour les structures audio et SMIL pour la synchronisation des divers éléments. L'un des avantages de l'utilisation d'un format XML pour définir de l'audio interactif est de pouvoir utiliser les nombreux outils existants pour transformer, éditer et adapter les documents XML. Un autre avantage est que le développement d'éditeurs graphiques pour les compositeurs audio peut être réalisé très facilement, par l'utilisation du format XML comme un mécanisme de sérialisation. Une implémentation partielle d'une API utilisant ce format a été développée pour la plate-forme iOS, basée sur l'API audio bas niveau FMOD de Firelight Technologies[6]. Le format A2ML ainsi que sa librairie associée ont été utilisés dans un premier temps dans un but de prototypage pour l'application de navigation. Cependant, ce format ayant été conçu initialement pour les applications de type jeux vidéos et paysage sonores, différents problèmes furent rapidement soulevés conduisant à la nécessité d'aménager le format pour les besoins spécifiques d'une application de guidage.

2.2.2 Objectifs d'un format adapté au guidage

Dans une optique d'un système de guidage audio, on retrouve en premier lieu tout ce qui constitue les bases de l'audio interactif : structuration audio, mixage des sons et synchronisation des éléments audio sur des événements de l'application. Cependant, des besoins supplémentaires et particuliers à la construction d'un système de guidage sont également nécessaires : il faut que le langage fournisse un moyen d'organiser les sons à priori et dynamiquement durant la navigation pour que l'utilisateur puisse obtenir l'indication la plus pertinente à chaque instant, et adaptée à son handicap. De plus, la capacité d'intégration d'un flux d'informations audio d'une personne étant limitée, à fortiori quand elle reçoit de l'information en provenance de deux sources différentes (l'environnement réel et les indications de guidage audio), il indispensable de limiter la quantité d'information audio transmises simultanément et dans le temps afin de ne pas surcharger l'utilisateur.

En sus de fonctionnalités d'audio interactif, le format doit donc permettre une gestion de priorité des différentes indications, avec des mécanismes d'ordonnancement évolués permettant le filtrage suivant des paramètres multiples. On distingue ainsi pour le guidage trois paramètres indispensables pour le filtrage : la priorité d'une indication, ainsi que les distances temporelle et géographique par rapport à l'instant de déclenchement de cette indication. En effet, de par le jeu des priorités, lorsqu'une indication audio sera déclenchée, rien ne garantie que celle-ci sera jouée immédiatement, une ou plusieurs indications plus importantes pouvant passer avant. De ce fait, quelle sera la garantie que lorsque cette indication sera finalement donnée à l'utilisateur, celle-ci sera encore pertinente ? C'est ici que l'on distingue plusieurs cas :

- Les indications pertinentes géographiquement uniquement (un trou dans la chaussée par exemple) qui doivent être éliminée dès que l'on s'éloigne de leur zone de pertinence.
- Les indications pertinentes dans le temps uniquement (comme le nom de la prochaine gare par exemple, si l'on se trouve dans un train) qui n'ont plus de sens si après qu'un certain temps se soit écoulé.
- Les indications pertinentes géographiquement et temporellement, qui regroupent les deux contraintes précédentes, comme la présence de travaux dans une bouche de métro par exemple.
- Les indications « toujours pertinentes » (dans le contexte de la navigation), tel que la météo du lieu de destination par exemple (on distingue bien ici la pertinence de l'indication en elle-même et non de son contenu).

Ces différents cas et leurs paramètres de filtrage correspondants doivent donc pouvoir être configurés dans le format de document décrivant les informations de guidage audio.

2.3 Le format MAUDL

2.3.1 Description

Partant de ces différents objectifs, un format adapté à la conception d'un système de guidage a été défini et nommé *Mobile Audio Language* (MAUDL). Ce format a été conçu avec le but d'être le plus léger, simple et direct possible tout aux répondant aux besoins exprimés. En effet, celui-ci a pour finalité d'être utilisé sur plate-forme mobile et d'être facilement adaptable et configurable par les utilisateurs pour convenir à tout type de handicap et préférences personnelles.

Ce format s'appuie sur les travaux réalisés sur le langage A2ML, notamment pour ce qui est des structures audio et de la synchronisation, toujours basée sur un système similaire à SMIL, mais uniquement orienté événements. Des nouvelles fonctionnalités propres à la problématique de guidage ont ensuite été ajoutées sur ces bases pour obtenir la forme actuel du langage MAUDL (voir annexe 4.1 pour le schéma Relax-NG du format).

L'architecture d'un document MAUDL s'organise autour de 3 axes principaux (figure 1) :

- La définition des sources sonores (*sounds*), qui permet d'effectuer des variations aléatoires ou définies d'une source sonore à l'aide d'une hiérarchie simple (*sounds* → *subsounds*).
- La définition de files d'attente de sons (*queues*) à priorité ou non, qui permettent d'organiser la lecture de sources sonores ou autres files d'attente de manière à définir un enchaînement de celles-ci suivant plusieurs paramètres. Elles permettent notamment de jouer en priorité certaines sources sonores (ce qui est indispensable pour le guidage), et définir des paramètres de validité des sources, par rapport à une durée ou une distance parcourue depuis le moment où la source a été ajoutée à la file, par exemple.
- La création d'une hiérarchie de mixage libre, où plusieurs niveaux de mixeurs peuvent être définis afin de créer des groupes de mixages pour les sources sonores et files d'attente de sons.

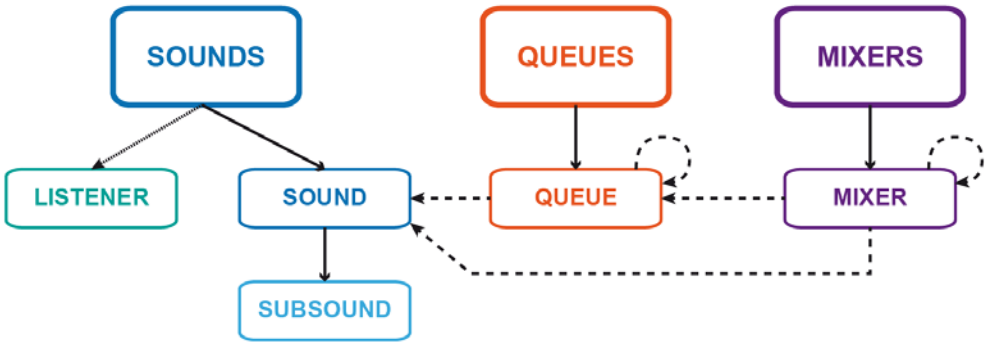

Figure 1: Architecture du langage MAUDL

Concernant la structuration audio, celle-ci reste minimaliste, dans un but de simplicité et flexibilité : on peut définir 2 types d'éléments, les sons (*sound*) qui peuvent éventuellement contenir plusieurs sous-sons (*subsound*). Un élément « son » est une méta-structure séquentielle liée ainsi à un ou plusieurs fichiers audio, avec des paramètres de mixage qui lui sont propres (volume, pan), de rendu (2D/3D), de synchronisation et éventuellement de position (pour le rendu 3D). Un son est toujours composé (implicitement ou explicitement) d'au moins un sous-son. Cette structuration permet d'introduire entre autres choses des variations d'une même icône audio : par exemple, pour sonoriser une porte, on peut choisir d'utiliser 3 enregistrements de bruits de porte différents, dont un seul sera choisi aléatoirement lors du déclenchement de la lecture du son (figure 2).

Figure 2: Exemple de structuration audio pour une icône audio de porte

Au niveau de la synchronisation, on distingue deux structures de média contrôlables : les sons et les files d'attentes de son. Concernant le contrôle de lecture de ces éléments, on peut les synchroniser sur un ou plusieurs événements les contrôles suivants :

- Démarrage de la lecture du son / de la file d'attente (attribut *start*).
- Mise en pause de la lecture du son / de la file d'attente (attribut *pause*).
- Arrêt de la lecture du son / de la file d'attente (attribut *stop*).
- Réinitialisation du pointeur de lecture du son / du son courant d'une file d'attente au début du fichier audio (attribut *reset*).
- Ajout du son / de la file d'attente à une nouvelle file d'attente (attribut *enqueue*).

On retrouve donc l'ensemble des contrôles de lecture de média de base. Pour ce qui est du dernier cas, nous allons maintenant voir en détail comment fonctionnent les files d'attente de son.

Une file d'attente de son, telle que son nom l'indique, est une structure dans laquelle un ou plusieurs sons ordonnés attendent d'être joués. Un seul son est joué à la fois, et dès qu'un son a fini de jouer il est supprimé de la file d'attente. Dans son mode le plus basique, la file d'attente peut être vue comme un système d'enchaînement de sons. Elle peut être configurée pour démarrer

automatiquement ou non la lecture dès qu'un son est ajouté (attribut *autoStart*), et une taille maximum peut être définie sous la forme du nombre maximum de sons qu'elle peut contenir (les sons de la file d'attente situés au delà de cette limite étant supprimés de la file).

De plus, la file d'attente peut également servir de système de tri et de filtrage des sons qui lui sont ajoutés : au sein d'une file d'attente, les sons peut être ordonnés selon leur ordre d'ajout ou leur priorité (attribut *sort*). Les sons peuvent être ensuite filtrés dynamiquement suivant des paramètres de temporalité et de proximité définis lors de leur ajout à la file d'attente. La valeur de l'attribut *enqueue* est ainsi construite suivant cette forme, les deux derniers paramètres étant facultatifs (la valeur par défaut pour ceux-ci étant *-1*):

```
<événement>:<nom de la file d'attente>[:<temps de validité ou -1>:[<distance de validité ou -1>]]
```

Le tri des sons et le filtrage de ceux-ci sera ensuite effectué à chaque fois qu'un son est ajouté à la file d'attente ou que le son courant a fini de jouer. Concernant le filtrage par temps de validité (en secondes), le temps de référence est compté à partir du moment où le son est ajouté à la file. Ensuite, le calcul pour déterminer si le son doit être supprimé de la file se base sur l'horloge de celle-ci qui peut être configurée de deux manières différentes (attribut *timebase*) :

- En mode temps réel (*realtime*), le temps s'écoule de manière continue sans aucune interruption.
- En mode temps de lecture (*playtime*), le temps ne s'écoule que lorsque la file d'attente est en train de jouer un son. Si elle est arrêtée ou en pause, le temps ne s'écoule pas.

Pour ce qui est de la distance de validité (exprimée en unités), la position de référence est celle de l'auditeur (*listener*) au moment où le son est ajouté à la file d'attente. De la même manière, le calcul de la distance pour déterminer ou non le filtrage sera basé sur la position actuelle de l'auditeur. Il revient à la charge de l'application de mettre à jour régulièrement cette position pour que le filtrage fonctionne comme attendu.

Enfin, nous arrivons à l'étape de mixage des différents sons. Dans le format MAUDL, il est possible de créer une hiérarchie libre de mixeurs, sans limites sur le nombre de niveaux. Ceci est très utile pour pouvoir réaliser des sous-groupes de mixage et ainsi agir facilement sur les différents niveaux. Par ailleurs, il est intéressant de noter que les sons se comportent comme des mixeurs de sous-sons, et que les files d'attentes comme des mixeurs de sons. De même, il existe toujours un mixeur global contrôlant l'ensemble des niveaux des éléments audio, même si il n'est pas déclaré explicitement dans le document. Voici par exemple l'organisation d'une hiérarchie de mixage possible, correspondant à l'exemple 5 de la section suivante :

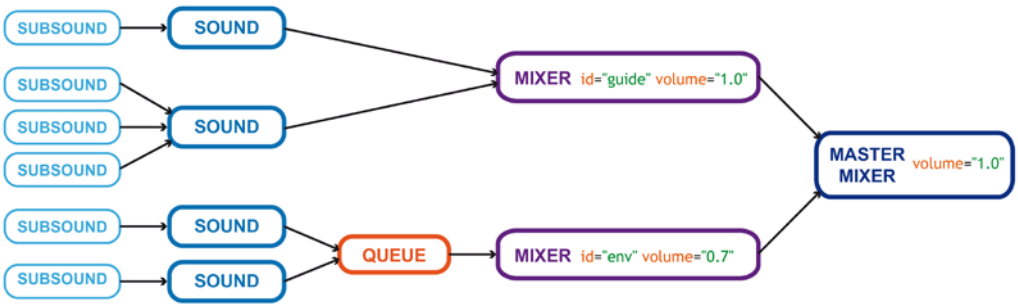

Figure 3: Exemple de hiérarchie de mixage

2.3.2 Utilisation et exemples

Afin d'illustrer plus en détail l'utilisation des différentes fonctionnalités du format, nous allons étudier différents exemples de cas d'utilisation du langage. A titre de référence, un exemple de document illustrant l'ensemble des éléments et attributs du langage est également disponible mais ne sera pas traité ici (voir annexe 4.2).

- *Exemple 1 : structuration audio avec sélection aléatoire*

Dans cet exemple nous allons voir comment créer une structure audio permettant d'introduire des variations aléatoires du son à chaque déclenchement. Pour illustrer cet exemple, nous allons prendre le cas de le sonorisation de deux événements simple, un bruit de pas et la signalisation de l'arrivée à la destination : à chaque pas que fait l'utilisateur, nous voulons qu'il reçoive un retour auditif, avec une variation du son joué à chaque fois pour éviter la monotonie. Une indication sonore simple sera également jouée pour signifier à l'utilisateur son arrivée à destination. Voici un exemple de document correspondant à ce cas :

```
<maudl>
  <sounds>
    <sound id="arrivée" src="arrivée.wav" play="navigation.arrivée"/>
    <sound id="pas" pick="omitMostRecent" play="navigation.pas">
      <subsound src="pas1.wav"/>
      <subsound src="pas2.wav"/>
      <subsound src="pas3.wav"/>
    </sound>
  </sounds>
</maudl>
```

On distingue ici deux types de son :

- Un son simple (arrivée), constitué d'un seul fichier audio. Le son sera joué dès que l'événement *navigation.arrivée* sera envoyé au gestionnaire de son.
- Un son complexe (pas), constitué de 3 fichiers audio contenant différentes variations d'un bruit de pas. Le son sera joué dès que l'événement *navigation.pas* sera envoyé au gestionnaire de son.

On note un attribut particulier sur le son « pas » : l'attribut *pick* spécifie la méthode de sélection du sous-son qui sera joué à chaque fois que la lecture du son est déclenchée. Il y a cinq valeurs possibles pour cet attribut :

- *ordered* : les sous-sons sont joués les uns après les autres, dans l'ordre dans lequel ils sont définis (valeur par défaut).
- *reversed* : les sous-sons sont joués les uns après les autres, dans l'ordre inverse dans lequel ils sont définis.
- *random* : un sous-son est choisi aléatoirement à chaque fois que le son est déclenché.
- *omitMostRecent* : un sous-son est choisi aléatoirement à chaque fois que le son est déclenché, à l'exclusion du dernier sous-son joué.
- *fixed* : le sous-son sélectionné ne change pas au déclenchement du son. La sélection d'un sous-son différent passe par l'utilisation de l'attribut *setNext* spécifiable dans un

sous-son : celui-ci est sélectionné lorsque l'événement spécifié dans cet attribut est déclenché.

Dans notre cas d'utilisation, nous avons utilisé la valeur *omitMostRecent*, puisque nous voulons un bruit de pas différent à chaque pas.

- *Exemple 2 : Spatialisation des sons*

Dans cet exemple, nous allons voir comment utiliser la spatialisation en 3D d'un son. Pour illustrer cet exemple, nous allons prendre le cas de la sonorisation d'une scène de concert de piano virtuelle au sein d'un théâtre. Cette scène sera constitué de deux éléments sonores : la musique du piano qui joue sur la scène, et la simulation de la réverbération de la salle de théâtre. Voici le document correspondant à ce cas :

```
<maudl>
  <sounds listenerPos="0.0 0.0 5.0" listenerLookAt="0.0 0.0 1.0" listenerRearAttenuation="0.5"
          listenerFrontAngle="180" rolloff="log" scale="1.0">
    <sound id="piano" src="piano.wav" loopCount="-1" render3D="advanced" pos="2.0 0.0 10.0"
          min="1" max="100" play="concert.start" stop="concert.stop"/>
    <sound id="reverb" src="reverb.wav" loopCount="-1" render3D="advanced" panFactor="0.0"
          pos="0.0 0.0 -10.0" min="1"  max="100" volume="0.2" play="concert.start"
          stop="concert.stop"/>
  </sounds>
</maudl>
```

Tout d'abord, une configuration de la scène audio est effectuée de manière globale au travers des divers attributs de l'élément *sounds*. Ces attributs définissent principalement les caractéristiques de l'auditeur (*listener*) et du modèle utilisé pour la spatialisation 3D (pour plus d'information sur les modèles existants et leurs paramètres, voir la section 2.4.2) :

- L'attribut *listenerPos* permet de définir la position initiale de l'auditeur dans l'espace virtuel. Dans notre cas l'auditeur est positionné initialement face à la scène, assis au second rang.
- L'attribut *listenerLookAt* permet de définir la direction initiale dans lequel l'auditeur regarde dans l'espace virtuel. Dans notre cas l'auditeur regarde initialement en direction de la scène.
- L'attribut *listenerRearAttenuation* spécifie la quantité maximum d'atténuation arrière utilisée pour créer un effet de focus audio dans la direction regardée (voir figure 11). Dans notre cas, l'atténuation maximale est fixée à la moitié du volume normal, pour focaliser l'attention de l'auditeur sur ce qui se trouve devant lui.
- L'attribut *listenerFrontAngle* spécifie la taille de l'angle définissant ce qui se trouve devant l'auditeur, et donc à partir duquel l'atténuation arrière entre en jeu (voir figure 11). Dans notre cas nous avons conservé la valeur par défaut de 180°.
- L'attribut *rolloff* spécifie le modèle d'atténuation de volume des sources audio spatialisées par rapport à leur distance à l'auditeur qui sera utilisé. Cet attribut possède deux valeurs possibles : *log* pour un modèle d'atténuation logarithmique qui correspond au modèle physique réaliste, et *linear* pour un modèle d'atténuation linéaire. Dans notre cas, nous avons choisi le modèle réaliste.

- L'attribut *scale* spécifie le facteur d'échelle pour l'atténuation du volume des sources audio spatialisées. Un facteur 2 signifiera par exemple que l'atténuation sera 2 fois plus rapide qu'en temps normal.

Il est intéressant de noter que le positionnement d'objets dans l'espace virtuel n'utilise pas d'unité de mesure spécifique. Il revient à l'utilisateur du format de choisir sa propre unité de distance (cm, m, pieds...) en fonction de ses besoins.

Maintenant que l'espace sonore et l'auditeur sont définit, il convient de positionner notre source sonore (un piano en train de jouer) dans l'espace. Etudions maintenant en détail l'effet de certains attributs de cette source sonore :

- L'attribut *loopCount* permet de spécifier le nombre de fois que le son va jouer. Par défaut, le son joue une seule fois et s'arrête une fois le fichier audio terminé. Ici nous avons défini la valeur à *-1* qui indique que le son va jouer en boucle indéfiniment.
- L'attribut *render3D* spécifie si le son doit être spatialisé ou non. Dans notre cas, le piano le sera, le mode *advanced* indiquant que l'atténuation arrière est prise en compte.
- L'attribut *pos* permet de définir la position initiale de la source sonore dans l'espace virtuel. Dans notre cas le piano est situé sur la scène devant l'auditeur, légèrement sur la droite.
- Les attributs *min* et *max* contrôlent l'atténuation du volume du son en fonction de la distance avec l'auditeur (voir figure 4).

Concernant la réverbération du piano, elle sera simulée de manière simple : nous avons enregistré un fichier audio avec un effet de réverbération pré-calculé sur le même son qui est joué par le piano. Les deux sons (piano et reverb) seront démarré et arrêtés en même temps, en étant synchronisés sur les même événements *concert.start* et *concert.stop*. Le volume est initialement réglé à 20% du son de piano, correspondant à une reverb audible classique. Avec d'ajuster dynamiquement la balance entre le son direct (piano) et la reverb, nous allons simplement positionner la reverb dans l'espace virtuel (activation du rendu 3D) et utiliser l'atténuation du volume pour jouer sur la balance. Cependant le son reverb étant sensé provenir de tous les cotés à la fois, nous choisissions d'ignorer l'effet du rendu 3D sur la panoramique du son en mettant l'attribut *panFactor* à 0. Cet attribut permet de régler la balance entre rendu 3D et rendu 2D classique sur la panoramique du son. Enfin, le son de reverb sera positionné au fond de la salle de théâtre, derrière l'auditeur : ainsi lorsque l'auditeur se rapproche du piano, la quantité de reverb diminue et le son direct augmente, et inversement lorsqu'il s'éloigne.

- *Exemple 3 : Synchronisation de sons avec interactions*

Dans cet exemple, nous allons voir comment tirer parti du système de synchronisation pour réaliser des interactions entre les divers éléments audio. Pour illustrer cet exemple, nous allons prendre le cas d'une visite guidée virtuelle, durant laquelle divers événements ont lieu pour maintenir l'attention de l'utilisateur. La voix du guide parle en continu, mais peut s'interrompre brutalement pour vous dire de faire attention à un obstacle immédiat, comme une voiture ou un skateboard qui s'avance vers vous. Voici le document correspondant à ce cas :

```
<maudl>
  <sounds>
    <sound id="voix_visite" src="visite.wav" play="guide.parle" pause="attention.started"/>
    <sound id="voix_attention" src="attention.wav"
```

```
        play="danger_voiture.started danger_skate.started"/>
    <sound id="danger_voiture" src="voiture.wav" play="événement.voiture"/>
    <sound id="danger_skate" src="skateboard.wav" play="événement.skate"/>
  </sounds>
</maudl>
```

Prêtons attention aux divers événements externe qui contrôlent cette scène audio :

- L'événement *guide.parle* tout d'abord déclenche la voix du guide pour la visite.
- Les événements *événement.voiture* et *événement.skate* déclenchent respectivement les sons de voiture et de skateboard.

Ces trois événements peuvent être envoyés par l'application au gestionnaire de sons. Mais il faut également noter que les sons en eux-mêmes génèrent des événements internes qui peuvent être utilisés lors de la description d'une scène audio. Ces événements internes sont de la forme :

```
<id de l'élément ciblé>.<type d'événement>
```

Voici les différents types d'événements internes existants :

- *started* : la lecture du son ou de la file d'attente vient de démarrer.
- *ended* : la lecture du son ou de la file d'attente vient de terminer.
- *stopped* : la lecture du son ou de la file d'attente a été stoppée.
- *paused* : la lecture du son ou de la file d'attente a été mise en pause.
- *resumed* : la lecture du son ou de la file d'attente a été reprise après une pause.
- *reset* : la position de lecture du son ou du son courant de la file d'attente a été réinitialisée au début du fichier audio.
- *queued* : le son ou la file d'attente a été ajouté à une file d'attente.

Voyons maintenant comment ces événements ont été mis à profit dans notre cas. Lorsque les sons *danger_voiture* et *danger_skate* vont commencer à jouer, le son *voix_attention* va également démarrer (utilisation des événements internes *danger_voiture.started* et *danger_skate.started*). De plus, lorsque le son *voix_attention* va démarrer, la voix de la visite sera automatiquement mise en pause (utilisation de l'événement interne *voix_attention.started*).

L'utilisation d'événements internes peut ainsi permettre des enchaînements complexes d'interactions entre sons. Dans ce cas volontairement simple, nous aurions bien sûr pu nous contenter d'utiliser uniquement des événements externes. Mais dans des cas de scènes audio beaucoup plus complexes, si par exemple les son de voiture et de skate pouvaient être déclenchés par de multiples événements externes différents, l'intérêt de l'utilisation d'événements internes devient de suite plus évident et facilite grandement l'écriture de document représentant de telles scènes audio.

- *Exemple 4 : Utilisation d'une file d'attente*

Dans cet exemple, nous allons voir comment utiliser une file d'attente pour réaliser les opérations d'enchaînement, d'organisation et de filtrage de sons. Pour illustrer cet exemple, nous prenons le cas d'un système de navigation ayant différentes informations à fournir, de nature et de priorités variées. Voici le document correspondant :

```
<maudl>
   <sounds classOrder="danger obstacle info">
      <sound id="trou" src="trou.wav" enqueue="nav.trou:file_nav:-1:5" class="danger"/>
      <sound id="porte" src="porte.wav" enqueue="nav.porte:file_nav:-1:5" class="obstacle"/>
      <sound id="accueil" src="accueil.wav" enqueue="nav.accueil:file_nav:10" class="info"/>
      <sound id="toilettes" src="toilettes.wav" enqueue="nav.toilettes:file_nav:10" class="info"/>
   </sounds>
   <queues>
      <queue id="file_nav" autoPlay="true" timeBase="realTime" maxElements="-1" sort="class"/>
   </queues>
</maudl>
```

On peut voir que 4 sons son définis ici, organisés en 3 classes différentes : danger, obstacle et information. Ces classes sont organisées par ordre décroissant de priorité grâce à l'attribut *classOrder*. Une file d'attente *file_nav* a été déclarée dans le but de recevoir ces différents sons. Celle-ci est configurée pour jouer automatiquement dès qu'elle contient un son (attribut *autoPlay*) et trier les sons par ordre de priorité (attribut *sort*). Elle utilisera une horloge continue (attribut *timeBase*) et peut contenir un nombre illimité d'éléments (valeur *-1* pour l'attribut *maxElements*), puisque nous allons choisir de les filtrer suivant différents critères.

Lors du déclenchement des 4 sons que nous avons définis via les événements *nav.**, ceux-ci seront ajoutés à la file d'attente *file_nav* avec divers paramètres de validité. Pour rappel, voici la structure de la valeur d'un attribut *enqueue* :

```
<événement>:<nom de la file d'attente>[:<temps de validité ou -1>:[<distance de validité ou -1>]]
```

On voit dans notre exemple que les sons *trou* et *porte* ont tous deux un temps de validité illimité, mais une distance de validité fixée à 5 unités (dans notre cas, considérons que ce sont des mètres). A l'opposé, les sons *accueil* et *toilettes* ont une durée de validité de 10 secondes, mais une distance de validité infinie

Lorsque ces sons seront déclenchés durant la navigation, il seront ainsi automatiquement ordonnés selon leur classe de priorité et retirés de la file d'attente si l'un de leurs paramètres de validité a expiré.

- *Exemple 5 : Création d'une hiérarchie de mixage*

Dans cet exemple, nous allons nous intéresser au mixage des différents éléments audio de notre document. Pour illustrer ce cas, nous allons reprendre l'exemple du système de navigation fournissant différentes informations de manière auditive. Voici le document que nous allons étudier:

```
<maudl>
   <sounds classOrder="danger info">
      <sound id="arrivée" src="arrivée.wav" play="nav.arrivée"/>
      <sound id="pas" pick="omitMostRecent" play="nav.pas">
         <subsound src="pas1.wav"/>
         <subsound src="pas2.wav"/>
         <subsound src="pas3.wav"/>
```

```
      </sound>
      <sound id="trou" src="trou.wav" enqueue="nav.trou:file_nav:-1:5" class="danger"/>
      <sound id="toilettes" src="toilettes.wav" enqueue="nav.toilettes:file_nav:10" class="info"/>
   </sounds>
   <queues>
      <queue id="file_nav" autoPlay="true" timeBase="realTime" maxElements="-1" sort="class"/>
   </queues>
   <mixers volume="1.0">
      <mixer id="guide" volume="1.0">arrivée, pas</mixer>
      <mixer id="env" volume="0.7">file_nav</mixer>
   </mixers>
</maudl>
```

Ce document est composé de 4 sons : 2 sont liés à la sonorisation du guidage de la personne (*arrivée* et *pas*) et 2 sont liés au rendu de l'environnement (*trou* et *toilettes*). Ces derniers seront joués par l'intermédiaire de la file d'attente *file_nav*, de la même manière que dans l'exemple précédent.

Ce qui nous intéresse plus particulièrement dans cet exemple ce sont les élément *mixer* qui ont été ajoutés, pour créer une hiérarchie de mixage. Le volume du mixeur global est ici défini explicitement à 100% (attribut *volume* de l'élément *mixers*), pour illustrer la présence de ce mixeur qui sera dans tous les cas en bout de chaîne. Nous avons ensuite créé 2 sous-groupes de mixage *guide* et *env* qui contiennent respectivement les sons liés aux guidage et à l'environnement. Le mixeur des sons décrivant l'environnement a été configuré avec un volume plus faible que celui destiné au guidage, en raison de son importance moindre. La hiérarchie de mixage finale de cette configuration est représentée par la figure 3 de la section précédente.

Il est donc très facile d'organiser une hiérarchie de mixage complexe au travers de groupes de mixage séparés pour contrôler directement le volume de plusieurs catégories de sons de manière indépendante.

2.4 Pointeur audio 3D

2.4.1 Objectifs

La nécessité principale d'une application de navigation est d'indiquer l'itinéraire à suivre à la personne guidée. Pour se faire, il est donc indispensable de pouvoir lui communiquer la direction à suivre à un instant donné, et de pouvoir lui faire ajuster sa propre direction dynamiquement en cas d'erreur. Pour répondre à ce besoin, nous faisons donc l'usage d'un pointeur audio que nous spatialisons en 3D et positionnions dans la direction que l'utilisateur doit prendre.

L'utilisation d'un tel pointeur a déjà été expérimentée à de nombreuses reprises dans différentes configurations de guidage, notamment dans le cadre de visites virtuelles[7] ou encore pour localiser des objets en intérieur[8].

La qualité de la spatialisation 3D et surtout son effet perçu par l'utilisateur joue ici un grand rôle puisqu'elle sera le critère déterminant sur la précision du guidage. A noter cependant que la précision de l'indication de direction sera dépendante de la précision de la localisation de la personne à guider effectuée par le module correspondant. En effet, si la position de l'utilisateur est estimée avec une trop grande erreur, la direction indiquée par le pointeur audio peut ainsi se retrouvée faussée. Cet aspect du guidage audio est donc intimement lié à la précision du module de localisation.

2.4.2 Techniques de spatialisation 3D

Il existe différentes techniques pour simuler le positionnement 3D d'un son dans l'espace, qui s'appuient sur les différents paramètres physiques et physiologiques qui permettent à notre oreille d'entendre notre environnement en 3D.

Ces paramètres sont de différentes natures :

- L'intensité du son, qui donne des indices sur la distance d'une source sonore, ainsi que sur sa direction de par la différence d'intensité entre chaque oreille (IID, *Interaural Intensity Difference*).
- Le délai entre le son perçu par chaque oreille (ITD, *Interaural Time Difference*), qui donne un indice sur la direction d'une source sonore.
- Les fréquences perçues par chaque oreille, dont la fonction de filtrage se représente par une HRTF (*Head Related Tranfer Function*), liée à la forme du corps et de l'oreille et donc différente pour chaque personne. Celle-ci donne des indices sur la direction ainsi que le positionnement en hauteur d'une source sonore.
- Les premières réflexions (*early reflections*) qui correspondent aux rebonds immédiats du son sur les murs et les objets avoisinants la source sonore, donnent un indice sur la taille de la salle dans laquelle l'auditeur se situe ainsi que sur la position d'objets proches.
- Les réflexions tardives *(late reflection* ou *reverb wash*) correspondent à la conséquence de la réverbération d'une source sonore dans une salle, et donne un indice sur la distance de la source sonore et la taille de la salle.

On peut résumer ces paramètres et leurs effets psychoacoustiques dans le tableau suivant :

Tableau 1: Effets psychoacoustiques des divers paramètres audio de spatialisation

		Effets psychoacoustiques perçus					
		Gauche / Droite	Devant / Derrière	Haut / Bas	Distance	Taille de la salle	Position d'objet proches
Paramètres	Intensité (IID)	●	(●)		●		
	Délai (ITD)	●	(●)				
	HRTF	●	●	●			
	Premières Réflexions					●	●
	Réflexions Tardives				●	●	
(●) : fournit un indice, mais paramètre non discriminant si utilisé seul							

Dans le cadre de la réalisation d'un pointeur audio de guidage spatialisé, notre objectif est de pouvoir indiquer une direction à suivre à l'utilisateur. Nous allons donc nous intéresser principalement au rendu de cette direction, qui correspond à l'azimuth dans le plan utilisateur. Le but recherché ici se résumer donc à permettre à l'utilisateur de discerner efficacement la position gauche/droite et devant/derrière d'une source sonore, ainsi que sa distance par rapport à sa destination.

La technique la plus répandue et utilisée en spatialisation audio est le calcul de différence d'intensité entre chaque oreille : il revient à simplement positionner la source dans l'espace stéréo (canaux gauche/droite) en fonction orientation de l'auditeur et de l'angle à la source, puis calculer l'atténuation du volume en fonction de la distance à celle-ci. Cette technique étant simple à mettre en œuvre et peu coûteuse en calculs, elle est très adaptée à une usage sur plateformes mobiles, et est notamment utilisée dans l'API audio OpenAL[9] disponible entre autres sur l'iPhone.

Il existe différents modèles de calcul pour l'atténuation du volume d'une source audio en fonction de son éloignement par rapport à l'auditeur. Nous avons choisi d'utiliser le modèle IDCM (*Inverse Distance Clamped Rolloff Model*), étant celui qui se rapproche du modèle physique réel et recommandé par les *Interactive Audio Rendering Guidelines*[2].

Si l'on définit par *d* la distance, le gain de la source audio *Gain(d)* est calculé par dérivation (4.1-4.3) et est utilisé pour ajuster le signal audio et aboutir à une atténuation naturelle du volume (figure 4). Le facteur d'atténuation *R* est utilisé pour augmenter ou diminuer l'effet d'atténuation, suivant les besoins.

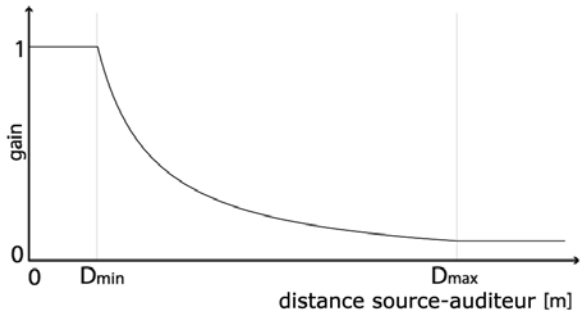

Figure 4: Gain d'une source calculé sur le modèle d'atténuation inverse de la distance

$$d = \max(d, D_{min}) \qquad (4.1)$$

$$d = \min(d, D_{max}) \qquad (4.2)$$

$$Gain(d) = \frac{D_{min}}{(D_{min} + R \times (d - D_{min}))} \qquad (4.3)$$

Concernant le positionnement de la source dans l'espace stéréo, on effectue une simple balance entre les canaux gauche et droite en fonction de l'angle de la source par rapport à l'axe horizontal formé par l'alignement des oreilles de l'auditeur. Cette méthode présente néanmoins un problème : elle ne permet pas de distinguer si une source audio se situe devant ou derrière

l'auditeur. En effet, deux source audio en provenance d'un angle identique par rapport à l'axe de l'auditeur forment ce que l'on appelle le cône de confusion (figure 5). L'auditeur est ainsi incapable de déterminer si les sources formant ce cône se situent devant ou derrière lui. Ce problème est principalement du à la simplification du modèle de tête utilisé (une sphère), et peut se résoudre par une approche tenant compte des paramètres anatomique de la tête de l'utilisateur.

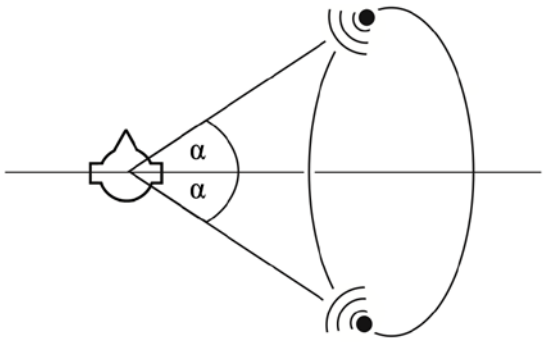

Figure 5: Deux sources avec un angle axe-oreille identique partagent le même cône de confusion

Pour effectuer une spatialisation tenant compte des paramètres anatomiques de la personne on utilise alors une HRTF pour réaliser la fonction de filtrage correspondant à l'auditeur. Le choix de la HRTF n'est cependant pas anodin : celle-ci étant liée à la morphologie d'une personne, cela implique que pour obtenir des résultats optimaux, cette personne devra se faire mesurer son HRTF personnelle à l'aide d'équipement spéciaux dans une chambre anéchoïde, telle que celle présente à l'IRCAM[10]. Il existe des solutions simples pour pallier à cette contrainte, comme par exemple d'utiliser des HRTF généralisée qui fonctionnent pour une tête « moyenne » (avec le désavantage d'être moins précises et de ne pas marcher chez certaines personnes) ou encore de piocher dans des bases de données de HRTF pré-existantes telles que *Listen* de l'IRCAM[11] par exemple, celle qui se rapproche le plus de la personne ciblée.

Cette contrainte mise à part, il existe un autre problème d'ordre technique cette fois concernant l'utilisation de HRTF : l'application de ce filtre nécessite des ressources de calcul importantes, ce qui est problématique dans le cas de notre application sur mobile, la puissance de calcul étant limitée. Une solution existe cependant pour pallier à ce problème : l'utilisation de fichiers audio dans lesquels on aura pré-calculé différentes positions dans l'espace et appliqué la HRTF correspondante. C'est cette solution que nous avons mis en œuvre pour la création du pointeur audio de guidage.

Le principe est de diviser l'espace circulaire autour de l'auditeur en un nombre défini de points, et de pré-calculer la spatialisation 3D avec HRTF du pointeur audio pour chacun de ces points (figure 6).

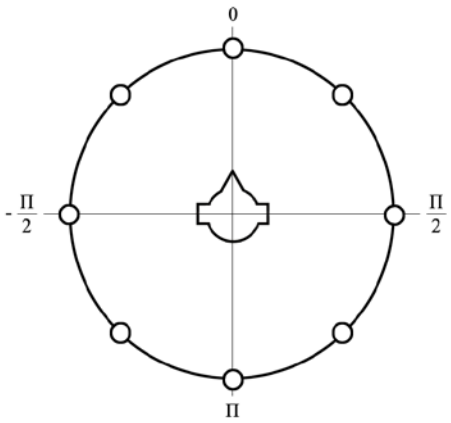

Figure 6: Pré-calcul du pointeur de guidage autour de l'auditeur sur 8 positions

Dans une optique de guidage, un minimum de 8 points sont nécessaires (4 points cardinaux + 4 diagonales). La granularité peut être augmentée pour plus de précision dans le guidage, au prix d'un coût en ressource mémoire plus important.

L'étape finale est ensuite d'effectuer le guidage à proprement parler, en utilisant les multiples échantillons audio. Deux méthodes sont alors envisageables, et nous avons implémenté chacune d'elles pour les expérimenter :

- En considérant que la durée du son du pointeur de guidage est relativement courte (1-2s), on peut se contenter de jouer les échantillons de guidage les uns à la suite des autre, en choisissant à chaque fois l'échantillon se rapprochant le plus de la direction à prendre à l'instant *t* où le son commence.
- En démarrant l'ensemble les échantillons en même temps, on joue sur le mixage de ses sources pour obtenir par interpolation la direction de guidage souhaitée, de manière continue. A chaque échantillon est affecté un poids P_{yi} correspondant à son volume de mixage (figure 7) et variant suivant l'angle d'orientation de la direction β relatif à la tête de l'auditeur (figure 8).

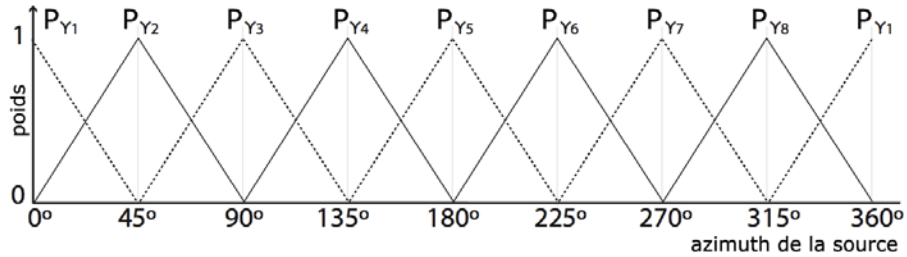

Figure 7: Poids des différents échantillons

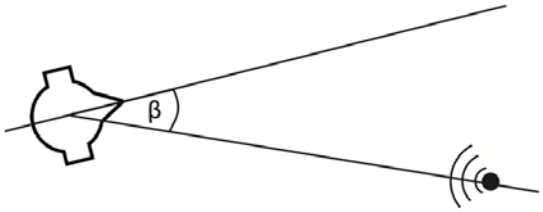

Figure 8: Angle d'orientation *β* de la source relatif à la tête de l'auditeur

Pour ce qui est du premier cas d'expérimentation, voici son écriture dans le langage MAUDL, à titre d'exemple (seuls les paramètres de synchronisation changent pour l'adapter au second cas) :

```
<maudl>
  <sounds>
    <sound id="pointer" loopCount="-1" pick="manual" render3D="no" play="nav.pointer.start"
        stop="nav.pointer.stop">
      <subsound src="pointer_p0.wav" setNext="pointer.p0"/>
      <subsound src="pointer_p1.wav" setNext="pointer.p1"/>
      <subsound src="pointer_p2.wav" setNext="pointer.p2"/>
      <subsound src="pointer_p3.wav" setNext="pointer.p3"/>
      <subsound src="pointer_p4.wav" setNext="pointer.p4"/>
      <subsound src="pointer_p5.wav" setNext="pointer.p5"/>
      <subsound src="pointer_p6.wav" setNext="pointer.p6"/>
      <subsound src="pointer_p7.wav" setNext="pointer.p7"/>
    </sound>
  </sounds>
</maudl>
```

Ici nous utilisons un seul bloc son nommé *pointer* dans lequel nous ajoutons les 8 échantillons correspondant aux 8 positions pré-calculées du pointeur sous forme de sous-sons. Il est configuré pour jouer en boucle (attribut *loopCount* à -1) avec une sélection manuelle des échantillons (attribut *pick*) et avec le rendu 3D désactivé, celui-ci étant pré-calculé. Au cours du déplacement, le gestionnaire de position sélectionnera l'échantillon à jouer correspondant le mieux à la direction à prendre en envoyant au gestionnaire audio un événement de la forme *pointer.p**.

2.4.3 Mise en oeuvre

Afin de donner à l'utilisateur le maximum d'informations au travers du pointer audio et pour minimiser ses chances d'erreurs tout en augmentant sa précision, nous nous sommes basé la technique précédente pour construire un pointeur de guidage disposant de 16 positions pré-calculées, avec cependant un degré d'information supplémentaire : le son spatialisé utilisé par le pointeur varie en fonction de la direction indiquée et permet à l'utilisateur de mieux corriger son orientation (figure 9). Nous avons ainsi découpé en 5 zones les différentes directions possibles indiquées par le pointeur et ajouté une aide à la direction par le biais d'un changement dans le type de son du pointeur. Ceci permet à l'utilisateur de discriminer plus rapidement et efficacement la direction à prendre.

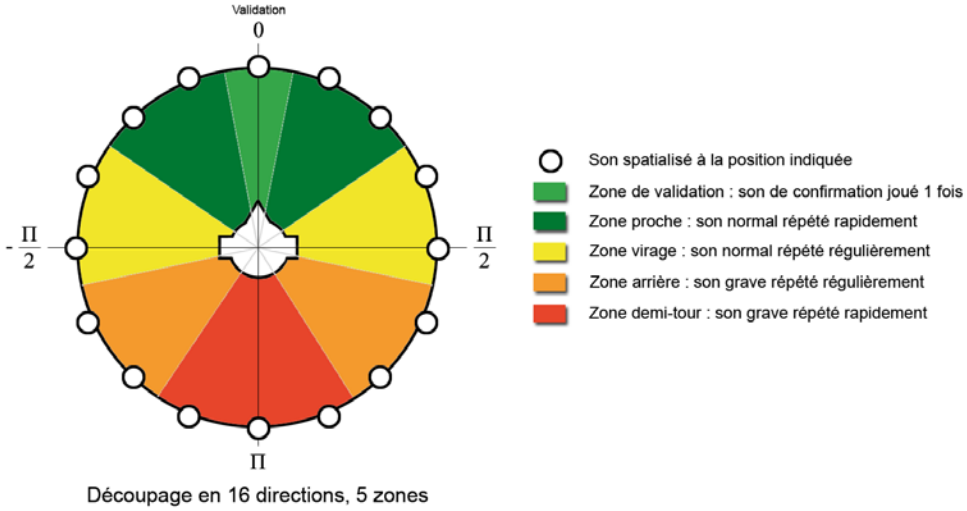

Figure 9: Pointeur audio avec variation de zone

A noter que de par les retours utilisateurs que nous avons eu et pour mieux convenir à tous les types de handicap (notamment pour les personnes ayant des difficultés à entendre les sons spatialisés), un second type de pointeur audio basé sur le même concept mais utilisant cette fois des indications vocales à la place d'un sons spatialisé a été défini. Il est activable à la demande de l'utilisateur par le biais des feuilles de styles audio personnalisables au sein de l'application de navigation.

2.4.4 Utilisation d'un head tracker pour améliorer la spatialisation

Au sein du système de navigation actuel, la direction indiquée par le pointeur audio de guidage considère que la tête de l'auditeur est orientée de la même manière que son corps. Si cette supposition fonctionne dans la plupart des cas, il est beaucoup plus naturel pour l'utilisateur de tourner sa tête sans bouger son corps dans une optique de recherche de direction.

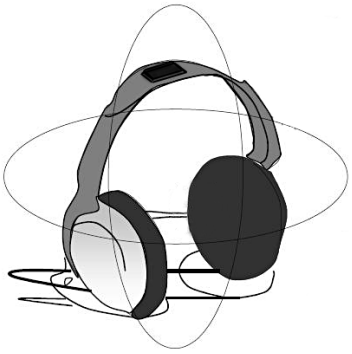

Figure 10: Exemple de dispositif de head tracking utilisant un gyroscope

Afin de compléter notre système de spatialisation, un dispositif de *head tracking* (figure 10) a ainsi été développé en partenariat avec la société Resonance. Le module matériel se compose

d'un accessoire à connecter à l'iPhone constitué de différents capteurs dont notamment un gyroscope et un magnétomètre. Le module, une fois positionné sur la tête de l'utilisateur, renvoie les valeurs valeurs de ces capteurs qui seront utilisées pour corriger l'orientation de la tête de l'auditeur dans la spatialisation : on stabilise ainsi l'espace sonore virtuel et la précision du pointeur de guidage s'en trouve grandement renforcée.

Cependant, les travaux effectués sur ce système en sont restés au stade expérimental, certains problèmes d'ordre technique n'ayant pu être résolus dans le temps imparti, notamment la déviation du gyroscope du head tracker dans le temps causant une rotation de l'espace sonore et faussant le rendu 3D à terme.

2.5 Rendu environnemental

Parallèlement à la tâche de guidage, l'utilisateur a également besoin de pouvoir connaître différentes informations sur l'environnement qui l'entoure au cours de son déplacement. Pour se faire, nous avons besoin de sonoriser les différents points d'intérêts (POI) signalés sur la carte durant un déplacement. Ces POIs seront ensuite mis en correspondance avec des icônes audio par l'intermédiaire d'une feuille de style audio décrite dans le langage MAUDL précédemment défini.

Il également nécessaire dans la plupart des cas de donner un positionnement dans l'espace à ces POI, et pour cela nous allons donc reprendre certaines techniques de spatialisation que nous avons vues dans la partie précédente. Dans un but de simplicité, pour économiser les ressources matérielles de la plate-forme mobile et ne nécessitant une localisation très précise de ces POI, nous avons choisi de nous contenter d'une spatialisation audio basée sur l'IID améliorée.

Le défaut principal de cette technique étant lié à la confusion devant/derrière des sources sonores, nous allons modifier la courbe de réponse en volume pour y intégrer une atténuation arrière, créant ainsi un effet de focus sur les sources audio positionnés devant l'auditeur (figure 11).

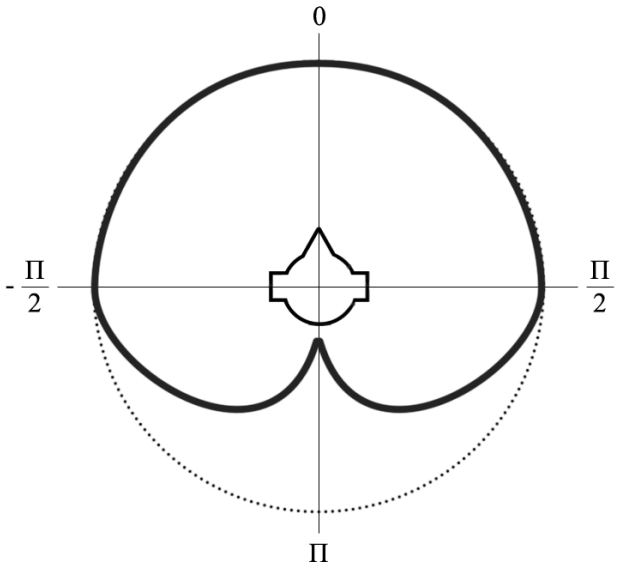

Figure 11: Courbe d'atténuation suivant l'orientation de la source

L'attention de l'utilisateur est ainsi orientée instinctivement vers les POIs sonorisés se trouvant devant lui tout en discriminant par effet de masquage ceux situés derrière.

Concernant l'association des icônes audio aux POIs proches, celle-ci se fera par l'intermédiaire d'événements nommés qui seront déclenchés par le gestionnaire de localisation. La feuille de style audio est ensuite définie de la manière suivant pour sonoriser cet événement :

```
<maudl>
  <sounds classOrder="danger obstacle info">
    <sound id="hole" src="icon/hole.wav" render3D="advanced"
      enqueue="nav.poi.hole:audio_icons:-1:5" class="danger"/>
    <sound id="door" src="icon/door.wav" render3D="advanced"
      enqueue="nav.poi.door:audio_icons:-1:5" class="obstacle"/>
    <sound id="lift" src="icon/lift.wav" render3D="advanced" enqueue="nav.poi.lift:audio_icons:10"
        class="info"/>
    <sound id="toilets" src="icon/toilets.wav" render3D="advanced"
      enqueue="nav.poi.toilets:audio_icons:10" class="info"/>
  </sounds>
  <queues>
    <queue id="audio_icons" autoPlay="true" timeBase="realTime" maxElements="-1" sort="class"/>
  </queues>
</maudl>
```

Dans cet exemple nous avons défini 3 classes de priorité pour les icônes audio, ordonnés par ordre décroissant d'importance :

- *danger*, qui correspond à l'indication d'un danger immédiat.
- *obstacle*, qui correspond à l'annonce d'un obstacle proche.
- *info*, qui correspond à une indication uniquement informative.

Ces icônes audio sont automatiquement ajoutées à la file d'attente audio_icons dès lors que le gestionnaire de position signale le passage de l'utilisateur à proximité d'un POI correspondant par l'intermédiaire d'un événement de la forme *nav.poi.**. Grâce à la spécification de priorité de ces classes (attribut *classOrder*) et de par la configuration du tri des sons par classe de priorité spécifié dans la file d'attente (attribut *sort*), les objets audio seront ordonnés automatiquement par ordre d'importance dans la file.

Nous avons également utilisé ici les attributs de validité d'un son dans la file d'attente pour configurer le comportement de ceux-ci : pour les POIs de type danger ou obstacle, tant que l'utilisateur se trouve à proximité il a besoin d'en être informé, c'est pour cela que la durée de validité est infinie (-1) et la distance de validité est fixée à 5m. Concernant les POIs de type information, la distance importe moins par contre la priorité étant de délivrer les icônes audio prioritaires en premier, si le l'information tarde trop à venir elle sera évincée (durée de validité fixée à 10s) pour ne pas encombrer la file d'attente audio. Etant d'importance moindre, si ces icônes audio ne sont pas jouées cela n'influera pas sur le guidage ou la sécurité de l'usager. De plus, le gestionnaire de position peut de toutes façons être interrogé à la demande si l'usager souhaite absolument obtenir cette information et qu'elle a été évincer par le déclenchement automatique.

Concernant la file audio, elle est configurée ici de manière standard, pour jouer automatiquement les sons dès leur ajout à celle-ci, fonctionnant en temps réel pour ce qui est du filtrage suivant la durée de validité des sons ajoutés et pouvant contenir un nombre illimité de sons (-1),

ceux étant configurés pour être de toutes façons éliminés au bout d'une certaine durée ou distance parcourue.

3 Gestionnaire de sons

3.1 Présentation et organisation

Afin de tirer profit du langage de description audio présenté dans les paragraphes précédents, il est nécessaire de disposer d'un gestionnaire de sons capable de charger des documents MAUDL et d'opérer le rendu audio en temps réel en interaction avec l'application de guidage. C'est la tâche la plus complexe à réaliser, d'autant plus que le gestionnaire de sons doit prendre en charge le rendu sonore 2D mais aussi 3D, et idéalement pouvoir fonctionner sur de multiples plateformes mobiles.

Des travaux antérieurs ont aboutis à un prototype de gestionnaire audio écrit autour du langage A2ML et reposant sur la librairie FMOD (http://www.fmod.org), qui outre les problèmes de licence, posait également souci au niveau de la compatibilité avec les différents appareils mobiles puisque la seule plateforme mobile actuellement supportée est iOS.

Afin de mieux correspondre aux besoins du projet, un tout nouveau moteur de rendu audio nommé *Interactive eXtensible Engine* (IXE) a ainsi été développé, pour tirer parti de l'architecture du langage MAUDL. Ce moteur de rendu est implémenté au sein de la librairie *libmaud*. Bien qu'actuellement le prototype de cette librairie fonctionne sur iOS, elle a été conçue pour être facilement portable sur d'autres systèmes, et ne dépend pas d'autres librairies audio tierces.

Les point majeurs de cette API de rendu sont actuellement :

- Le déclenchement de sources sonores par événements avec une faible latence
- Les files de sons ainsi que la gestion automatique de priorité pour les sources sonores
- Le rendu audio avec positionnement 3D selon divers modèles
- La synthèse vocale multilingue, incluant le français

3.2 La librairie libmaud

3.2.1 Fonctionnement de l'API

Pour commencer, le fonctionnement de cette API s'articule autour de la classe *ADSoundManager*. Notez que pour des raisons techniques incluant notamment le fait que le langage Objective-C ne possède pas de notion d'espace de nom, toutes les classes de la librairie *libmaud* ont été préfixées par « AD ». La classe *ADSoundManager* est de type *singleton* et une seule instance partagée de cette classe existe à la fois dans un programme. Elle s'occupe de gérer les tâches suivantes :

- Initialiser le matériel audio et gérer les éventuelles interruptions
- Charger et décharger dynamiquement des documents MAUDL
- Créer la hiérarchie d'objets audio nécessaires pour un document MAUDL
- Gérer la transmission d'événements entre l'application et les objets audio
- Maintenir un index des objets audio créés et de leurs identifiants associés

- Configurer les paramètres généraux du rendu audio : positionnement de l'auditeur, modèle de rendu 3D utilisé, définition des classes de priorités
- Synthétiser de la voix pour les objets audio via le moteur de *Text-To-Speech* intégré

En dehors de cette classe, l'API s'appuie sur des objets audio dont les concepts sont directement calqués sur le langage MAUDL. On peut le voir sur cette représentation simplifiée du diagramme de classe de l'API :

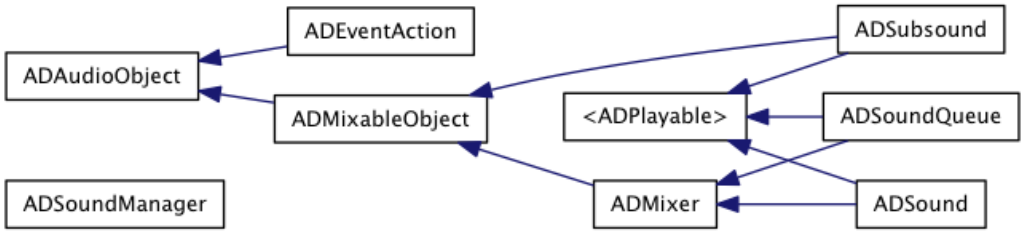

Figure 12: Diagramme de classe simplifié de la librairie libmaud

Ainsi, les classes *ADSound*, *ADSubsound*, *ADMixer* et *ADSoundQueue* sont les représentations objets des éléments Sound, Subsound, Mixer et Queue du langage MAUDL (voir figure 1).

Ces objets audio sont normalement initialisés et configurés automatiquement par le sound manager lors du chargement de documents MAUDL, cependant il reste toujours possible de créer et manipuler ces objets directement. Cette approche est cependant déconseillée sauf pour certains cas très spécifiques, l'API ayant été conçue pour fonctionner à partir de documents MAUDL, dont les objets sont par la suite manipulés à l'aide d'événements.

3.2.2 Utilisation et exemples

La majorité des applications peuvent ainsi se contenter d'utiliser uniquement la classe *ADSoundManager* pour l'ensemble de leurs besoins audio, la définition du contenu audio se faisant intégralement par le biais des documents MAUDL. Voici un scénario général d'utilisation de la librairie, faisant uniquement usage de cette classe :

```
//** Initialisation de l'application ****************************************

// Récupération de l'instance partagée du sound manager.
// L'instance est automatiquement créée au besoin et l'audio initialisé en
// même temps si nécessaire.
ADSoundManager *soundManager = [ADSoundManager sharedSoundManager];

// Chargement d'un document MAUDL et construction des objets audio
[soundManager loadXMLFromFile:@"monfichier.xml"];

[...]

//** Au cours de l'application ***********************************************

// Déclenchement d'un événement
[soundManager triggerEventName:@"monson.jouer"];

[...]

//** Fermeture de l'application **********************************************

// Destruction de tous les objets audio et libération de la mémoire
```

```
[ADSoundManager releaseSoundManager];
```

Ce scénario couvre un usage classique de la librairie : chargement d'un document audio puis déclenchement des sons à travers des événement. Il est également possible d'ajouter des informations supplémentaires lors du déclenchement d'un événement, dans le but de modifier certains paramètres des sons ou de changer la manière dont ils seront joués. Pour cela, il faut utiliser la classe *ADProperties*, qui permet de transmettre un certain nombre de propriétés définies à l'objet audio destinataire de l'événement. Voici un exemple de ce cas d'utilisation :

```
//** Au cours de l'application ***********************************************

// Récupération de l'instance partagée du sound manager
// On considère ici qu'un document MAUDL a été chargé préalablement
ADSoundManager *soundManager = [ADSoundManager sharedSoundManager];

// Création d'un objet properties
ADProperties *properties = [[[ADProperties alloc] init] autorelease];

// Définition d'une nouvelle position 3D pour le son à jouer
point3f_t position = { 4.0, 0.0, -2.0 };

// Création d'un objet conteneur pour la position
NSData *positionData = [NSData dataWithBytes:&position length:sizeof(position)];

// Ajout de de la propriété "position" avec la nouvelle valeur désirée
[properties addPropertyWithName:AD_PROPERTY_POSITION value:positionData];

// Déclenchement de l'événement avec les propriétés définies
[soundManager triggerEventName:@"monson.jouer" object:properties];
```

Dans cet exemple un objet *ADProperties* est créé auquel l'on va rajouter la propriété *position* affectée d'une valeur. Cet objet sera ensuit transmis conjointement à l'événement, le destinataire se chargeant par la suite de traiter ses données, en l'occurrence changer sa position dans l'espace. A noter cependant que si la position du son qui recevra cet événement sera changée, pour que cela cause un changement sur son rendu celui-ci devra être configuré dans le document MAUDL pour avoir un rendu 3D (voir section 2.3.2, *exemple 2*).

Enfin, pour des scénarios d'application plus complexes qui nécessitent de modifier en profondeur les attributs des objets audio au cours de l'exécution, il est possible de manipuler ceux-ci directement, comme nous allons le voir dans cet exemple :

```
//** Au cours de l'application ***********************************************

// Récupération de l'instance partagée du sound manager
// On considère ici qu'un document MAUDL a été chargé préalablement
ADSoundManager *soundManager = [ADSoundManager sharedSoundManager];


// Récupération de l'objet son avec l'ID spécifié
ADSound *monson = [soundManager objectForID:@"monson"];

// Modification des attributs de l'objet son
monson.numLoops = 2;
monson.subsoundPickMode = SubsoundPickOmitMostRecent;
```

Ici l'objet son correspondant à l'identifiant « monson » est récupéré, et ses attributs sont directement modifiés par le programme. Attention toutefois, car cet usage des objets audio est déconseillé autant que possible, car si le document MAUDL chargé ne correspond plus à ce à

quoi le programme s'attend (par exemple si dans cet exemple l'identifiant de l'objet son est modifié), le résultat pourrait être différent de celui attendu.

3.2.3 Fonctionnalités avancées

En complément de l'ensemble des fonctionnalités définies dans le langage MAUDL, la librairie libmaud supporte quelques fonctionnalités supplémentaires permettant des interactions plus avancées entre l'application et la modalité audio.

Tout d'abord, l'API autorise le chargement et le déchargement de documents audio à la volée, sans interruption (dans la mesure du possible) du flux de lecture audio. Il est ainsi possible de charger dynamiquement des documents audio venant construire ou compléter une scène audio :

```
//** Initialisation de l'application ***************************************

// Récupération de l'instance partagée du sound manager
ADSoundManager *soundManager = [ADSoundManager sharedSoundManager];

// Chargement d'un document MAUDL servant de base pour la scène audio
[soundManager loadXMLFromFile:@"base_audio.xml"];

[...]

//** Au cours de l'application ********************************************

// Chargement dynamique d'un nouveau document venant compléter la scène audio
NSString *documentID = [soundManager loadXMLFromFile:@"complement_audio.xml"];

[...]

// Déchargement du document à la volée et destruction des objets audio associés
[soundManager unloadXMLWithID:documentID];

[...]

//** Fermeture de l'application *******************************************

// Destruction de tous les objets audio et libération de la mémoire
[ADSoundManager releaseSoundManager];
```

Cette technique permet de composer une scène audio à partir de fragments de documents MAUDL pouvant provenir de sources différentes. Mais elle possède une autre propriété très importante puisqu'elle permet également de construire des scènes complexes avec des références d'objets audio inter-documents. En effet, il est possible par exemple de synchroniser des objets audio en provenance d'un document A sur les événements produits par les objets audio en provenance d'un document B. Voire même, ajouter ces objets lors de leur lecture à une file d'attente définie dans un document C, elle même faisant partie d'un groupe de mixage défini dans un document D. Ceci n'est en quelques mots qu'un rapide aperçu des vastes possibilités offertes par cette souplesse de fonctionnement.

Une autre fonctionnalité avancée offerte par l'API est de permettre de récupérer les événements internes en provenance du gestionnaire audio pour synchroniser en retour certaines parties de l'applications (comme l'affichage par exemple) sur le flux audio :

```
//** Méthode annexe synchronisée sur l'audio ******************************

- (void)audioCallback:(NSNotification *)notification {
    // traitement de l'événement
    ADAudioObject *audioObject = (ADAudioObject *)[notification object];
    NSLog(@"fin de la lecture de l'objet: %@", [audioObject objectID]);
```

```
}

//** Au cours de l'application ********************************************

// Récupération de l'instance partagée du sound manager
// On considère ici qu'un document MAUDL a été chargé préalablement
ADSoundManager *soundManager = [ADSoundManager sharedSoundManager];

// Spécifie une méthode à appeler lorsque l'objet son avec l'ID "monson"
// a terminé de jouer
[soundManager.eventNotificationCenter addObserver:self
                                         selector:@selector(audioCallback)
                                             name:@"monson.ended"
                                           object:nil];
```

Dans cet exemple, une méthode *audioCallback* est déclarée. L'objectif est que celle-ci soit appelée lorsque l'objet audio avec l'identifiant « monson » se termine. Nous faisons alors appel au centre de notification du sound manager (objet *NSNotification* standard du framework de base en objective-C) pour spécifier d'exécuter cette méthode dès que l'événement « monson.ended » est reçu. Cet événement est en fait généré automatiquement par le sound manager, et sert notamment à synchroniser les divers objets audio d'un document MAUDL entre eux. Ici nous récupérons cet événement normalement interne au sound manager pour effectuer des traitements dans notre application qui seront synchronisé sur le flux audio, ce qui offre de nombreuses possibilités.

Les événements de ce type sont spécifiés dans le langage MAUDL, et concernent uniquement les objets dérivants de l'interface *ADPlayable* (voir figure 12). Voici une liste exhaustive de ces événements :

- <id>.started : l'objet audio a démarré ou redémarré.
- <id>.ended : l'objet audio s'est terminé.
- <id>.stopped : l'objet audio a été stoppé.
- <id>.paused : l'objet audio a été mis en pause.
- <id>.resumed : l'objet audio a repris après une pause.
- <id>.reset : l'objet audio a été réinitialisé au début.
- <id>.queued : l'objet audio a été ajouté à une file d'attente.
- <id>.ignored : l'objet audio a ignoré une commandé de démarrage.

A ces événements « standards » viennent s'ajouter 3 événements spécifiques à la librairie libmaud, définis par les constantes suivantes :

- AD_EVENT_3D_SETTINGS_UPDATED : déclenché lorsque les paramètres 3D globaux du sound manager ont été mis à jour.
- AD_EVENT_BEGIN_AUDIO_INTERRUPTION : déclenché au début d'une interruption audio du système d'exploitation.
- AD_EVENT_END_AUDIO_INTERRUPTION : déclenché à la fin d'une interruption audio du système d'exploitation.

3.2.4 Dépendances et limitations actuelles

Développée dans le but de permettre au prototype de notre application de navigation de fonctionner, la librairie libmaud souffre néanmoins de quelques limitations à ce jour. Conçue en Objective-C, elle n'est de ce fait compilable directement que sur les plate-formes Apple (iOS et OS X). Elle s'appuie également sur le framework audio *AVFoundation* de iOS pour le rendu audio bas niveau. Ceci a de plus pour effet d'empêcher le bouclage au sample près de certains types de fichiers audio (compressés notamment), limitation due à la manière dont le framework *AVFoundation* gère la lecture de ces fichiers. Ces choix ont été faits consciemment afin d'avoir rapidement un prototype de la librairie fonctionnel et performant, car utilisant les API natives du système.

Cependant, la librairie a malgré tout été développée afin d'être portable facilement sur d'autres systèmes, de par son architecture très peu dépendante des API du système hôte : la seule classe où des API natives sont utilisées est la classe *ADSubsound*. Outre la problématique du langage, il suffirait juste de modifier cette classe pour changer l'API de rendu audio bas niveau de libmaud, pour lui permettre d'utiliser OpenSL[12] sur le système Android[13] par exemple. Des travaux sont actuellement en cours pour effectuer un portage de la librairie en C++ sur diverses plateformes.

3.3 Interrogation de l'environnement

3.3.1 Problématique et solution proposée

Dans le cadre de la conception d'une application d'aide au déplacement pour personnes malvoyantes, outre la problématique du guidage de la personne se pose un autre challenge à résoudre : comment fournir à la demande de la personne des informations pertinentes sur son environnement. Divers travaux ont déjà été effectués dans ce domaine, notamment pour essayer de fournir une représentation audio d'un espace visuel[14] à l'aide de capteurs de distances. Nous sommes cependant partis sur une approche différente, puisque l'objectif du système d'interrogation souhaité était de pouvoir fournir des informations thématiques à différents niveaux de distance (proche, moyenne distance et lointain).

Le système d'interrogation de l'environnement que nous avons choisi de développer se base sur les principes d'applications de réalité augmentée telle que Layar[15], qui permettent en visant une direction donnée avec la caméra d'un appareil mobile d'afficher les POIs avoisinants. Nous avons gardé ce principe de visée pour l'interrogation, cependant le système étant destiné aux personnes malvoyantes, les POIs situés dans la direction visée seront signalés de manière auditive et non plus visuelle (figure 13).

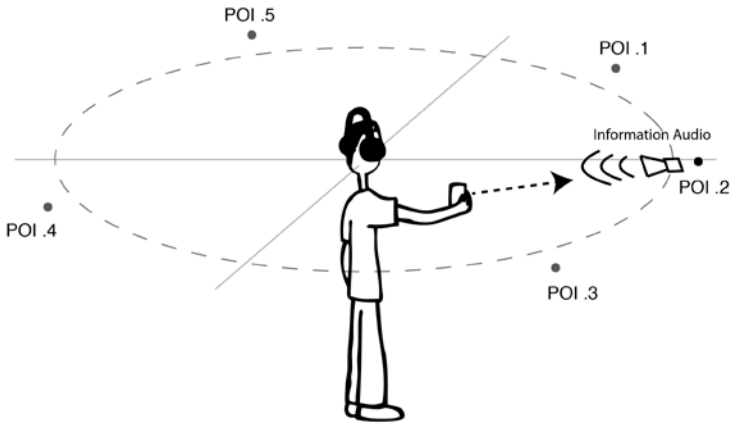

Figure 13: Système d'interrogation audio de l'environnement

Ce système présente plusieurs avantages dans le cadre de notre application :

- Le geste de pointage est instinctif, et permet de balayer l'horizon rapidement pour trouver l'information souhaitée.
- Les POIs existent déjà dans le système de cartographie utilisé pour la navigation, au format *OpenStreetMap* (OSM)[16].
- Le système de rendu audio est déjà intégré, avec de la synthèse vocale possible (libmaud).

Néanmoins, une amélioration de ce système reste possible, car il est nécessaire de pointer aléatoirement autour de soi pour découvrir les POIs environnants. C'est pour cela qu'en addition du système d'interrogation, nous avons conçu un système simple de découverte des POIs en utilisant les capacités de rendu audio 3D.

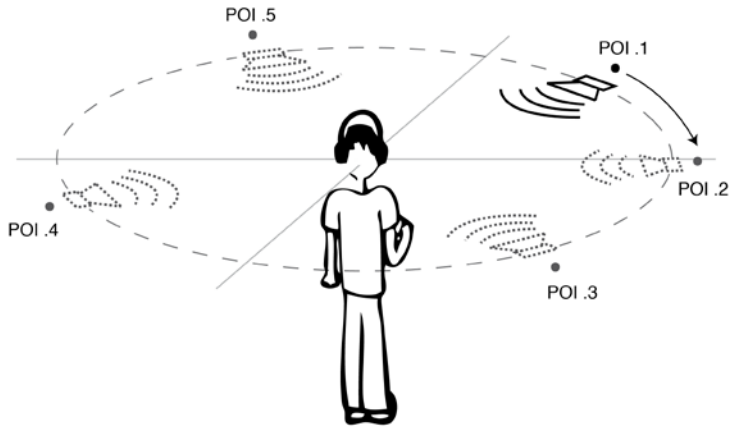

Figure 14: Système de découverte audio des POIs environnants

Comme illustré sur la figure précédente, en mode découverte, chaque POI indique sa position par un indicateur audio spatialisé, l'un après l'autre. Cela permet à l'utilisateur de situer l'orienta-

tion et le nombre de POIs avant de passer en mode interrogation pour obtenir plus de renseignements sur ceux-ci.

Nous allons maintenant voir plus en détail comment l'ensemble de ce système a été mis en œuvre pour tirer parti de la plateforme mobile et des divers développements réalisés dans le cadre du projet.

3.3.2 Mise en œuvre et intégration avec la librairie libmaud

Le moteur d'interrogation se base ainsi sur l'ensemble des outils développés dans le cadre de la conception du logiciel de navigation : orientation de l'utilisateur à partir des capteurs du téléphone, outils de cartographie pour les POIs et moteur audio pour le rendu.

Ce moteur a été développé sous la forme d'une API intégrant les divers outils précédents. Le pré-filtrage des POIs déterminant d'environnement de l'utilisateur basé sur la distance, la thématique et les préférences de ce dernier ne seront pas détaillées ici puisqu'il relève d'une autre partie du projet. L'API va ainsi se concentrer sur le rendu audio de cet environnement que l'on va donc considérer ici prédéterminé.

Tout d'abord, le groupe de POIs constituant le panorama audio dans lequel l'utilisateur va naviguer sera décrit par une composition de deux formats utilisés dans le projet: OSM et MAUDL. Les POIs sont ainsi groupé dans une relation OSM, chaque POI disposant d'une description audio décrite dans le format MAUDL (voir annexe 4.3). Le moteur d'interrogation charge donc ces documents en entrée pour opérer le rendu audio en sortie.

Ensuite, activé à la demande de l'utilisateur, le moteur d'interrogation de l'environnement va sonoriser les POIs lorsqu'ils seront pointés par l'utilisateur, selon la description audio fournie en entrée. Si le POI dispose d'informations complémentaires plus détaillées, l'utilisateur peut alors déclencher leur lecture s'il le souhaite, tant que le POI est sélectionné (pointé).

Enfin, le rendu audio est délégué à la libraire libmaud, à laquelle les fragments de documents MAUDL sont confiés. Le moteur d'interrogation déclenche alors seulement les événements correspondants aux actions sur les POIs.

Il est important de noter que le moteur d'interrogation vient s'intégrer au cœur du système de navigation et qu'il est donc de ce fait complètement dépendant des informations fournies par les sous-systèmes composant cette application : position de l'utilisateur via les centrales d'inerties embarquée, cartographie intégrée et rendu sonore configuré par feuille de style audio XML.

4 Annexes

4.1 Schéma de validation Relax-NG du format MAUDL

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   Relax-NG validation schema for the Mobile Audio Language [maudl]
   - Version 1.2.5
   - Last modification: 30/09/2011
   - Author: Lasorsa Yohan, INRIA Research Center, WAM Project
-->
<grammar ns="http://wam.inrialpes.fr/iaudio/maudl/1.0" xmlns="http://relaxng.org/ns/structure/1.0"
datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
   <start>
      <element name="maudl">
         <!-- The unique ID of the document. An ID must be defined to be able to dynamically unload the
```

```
            document. -->
         <optional>
            <attribute name="id">
               <data type="ID"/>
            </attribute>
         </optional>
         <!--
            Sounds are the playable audio elements.
            The sounds container can configure the properties of the 3D rendering for the sounds
         -->
         <optional>
            <element name="sounds">
               <!-- The global 3D settings. -->
               <ref name="global3DSettings"/>
               <!-- The priority class order settings. -->
               <ref name="prioritySettings"/>
               <!-- The default voice synthesis settings. -->
               <ref name="voiceSettings"/>
               <zeroOrMore>
                  <ref name="sound"/>
               </zeroOrMore>
            </element>
         </optional>
         <!--
            Queues allow to play an ordered list of sounds or queues, one after the other.
         -->
         <optional>
            <element name="queues">
               <zeroOrMore>
                  <ref name="queue"/>
               </zeroOrMore>
            </element>
         </optional>
         <!--
            Mixers allow to perform the submix of a set of sounds, queues and mixers.
            A sound or mixer can only have one parent mixer.
         -->
         <optional>
            <element name="mixers">
               <!-- Master mix settings. -->
               <ref name="mixAttributes"/>
               <zeroOrMore>
                  <ref name="mixer"/>
               </zeroOrMore>
            </element>
         </optional>
      </element>
   </start>

   <!-- Global 3D settings -->
   <define name="global3DSettings">
      <!-- The listener 3D position. -->
      <optional>
         <attribute name="listenerPos">
            <list>
               <data type="double"/>    <!-- x, default = 0.0 -->
               <data type="double"/>    <!-- y, default = 0.0 -->
               <data type="double"/>    <!-- z, default = 0.0 -->
            </list>
         </attribute>
      </optional>
      <!-- The listener 3D orientation (the direction he is looking at). -->
      <optional>
```

```
        <attribute name="listenerLookAt">
          <list>
            <data type="double"/>    <!-- vx, default = 0.0 -->
            <data type="double"/>    <!-- vy, default = 0.0 -->
            <data type="double"/>    <!-- vz, default = 1.0 -->
          </list>
        </attribute>
      </optional>
      <!-- The attenuation of the sounds when the listener is not facing them in 3D mode. -->
      <optional>
        <attribute name="listenerRearAttenuation">
          <data type="double"/>    <!-- default = 0.0 (range from 0.0 to 1.0) -->
        </attribute>
      </optional>
      <!-- The angle in degrees before the listener rear attenuation begins in 3D mode. -->
      <optional>
        <attribute name="listenerFrontAngle">
          <data type="double"/>    <!-- default = 180.0 (range from 0.0 to 360.0) -->
        </attribute>
      </optional>
      <!-- The rolloff model for the volume used in 3D rendering. -->
      <optional>
        <attribute name="rolloff">
          <choice>
            <value>log</value>      <!-- default realistic logarithmic attenuation model -->
            <value>linear</value>   <!-- linear attenuation model -->
          </choice>
        </attribute>
      </optional>
      <!-- The rolloff attenuation scale factor. -->
      <optional>
        <attribute name="scale">
          <data type="double"/>    <!-- default = 1.0 -->
        </attribute>
      </optional>
    </define>

    <!-- 3D settings of an audio source -->
    <define name="mix3DSettings">
      <!-- The audio source 3D position. -->
      <optional>
        <attribute name="pos">
          <list>
            <data type="double"/>    <!-- x, default = 0.0 -->
            <data type="double"/>    <!-- y, default = 0.0 -->
            <data type="double"/>    <!-- z, default = 0.0 -->
          </list>
        </attribute>
      </optional>
      <!-- Render the audio source in 3D (if disabled, the audio source will be rendered using the standard
           2D model). -->
      <optional>
        <attribute name="render3D">
          <choice>
            <value>no</value>     <!-- Do no use 3D rendering (default value). -->
            <value>pan</value>    <!-- Simple positional 3D rendering without distance attenuation. -->
            <value>simple</value>     <!-- Simple positional 3D rendering with distance attenuation. -->
            <value>advanced</value><!-- Advanced positional 3D rendering with distance and rear
                                                attenuation . -->
          </choice>
        </attribute>
      </optional>
      <!-- The audio source's minimum volume distance. When using the standard log rolloff model, this will
```

```
            define how loud is the sound in the 3D space. -->
      <optional>
         <attribute name="min">
            <data type="double"/>    <!-- default = 1.0 -->
         </attribute>
      </optional>
      <!-- The audio source's maximum volume distance after which the volume stops attenuating. -->
      <optional>
         <attribute name="max">
            <data type="double"/>    <!-- default = 100.0 -->
         </attribute>
      </optional>
      <!-- Affects the balance between the panning effect rendering in 3D rendering mode and the pan
            value. -->
      <optional>
         <attribute name="pan3DFactor">
            <data type="double"/>   <!-- default = 1.0 (range from 0.0 [full 2D] to 1.0 [full 3D]) -->
         </attribute>
      </optional>
   </define>

   <!-- Priority settings -->
   <define name="prioritySettings">
      <!-- The priority classes. -->
      <optional>
         <attribute name="classOrder">
            <list>
               <oneOrMore>
                  <data type="string"/>    <!-- Names of classes ordered by highest to lowest priority. -->
               </oneOrMore>
            </list>
         </attribute>
      </optional>
   </define>

   <!-- Voice synthesis settings -->
   <define name="voiceSettings">
      <!-- The default voice synthesis parameters. -->
      <optional>
         <attribute name="voice">
            <!--
               Voice synthesis parameters, in the form: <language_code>[:<pitch>[:<rate>]]
               - language_code : the language code to use, in the ISO form "en-US" (default: en-US,
                     supported: en-US, en-GB, de-DE, es-ES, fr-FR, it-IT)
               - pitch         : the pitch of the voice (default = 100, range 50-200)
               - rate          : the speech rate of the voice (default = 100, range 20-500)
            -->
            <data type="string"/>
         </attribute>
      </optional>
   </define>

   <!-- Attributes of a mixable audio object -->
   <define name="mixAttributes">
      <!-- The volume of the audio object, from 0.0 (silence) to 1.0 (full volume). -->
      <optional>
         <attribute name="volume">
            <data type="double"/>   <!-- default = 1.0 -->
         </attribute>
      </optional>
      <!-- The pan of the audio object, from -1.0 (full left) to 1.0 (full right). Ignored when 3D rendering is
            used. -->
      <optional>
```

```
      <attribute name="pan">
        <data type="double"/>   <!-- default = 0.0 -->
      </attribute>
    </optional>
  </define>

  <!-- Attributes of a playable audio object -->
  <define name="playableAttributes">
    <!-- The priority class of the audio object. If the class does not appear in the classOrder attribute, its
         priority is considered as the lowest, equally with the others of its kind. -->
    <optional>
      <attribute name="class">
        <data type="string"/>
      </attribute>
    </optional>
      <!-- Determines the behavior of the audio object when it is already playing and a new play request
        has been received. -->
      <optional>
        <attribute name="replay">
          <choice>
            <value>restart</value><!-- Restart playing the audio object from start (default value). -->
            <value>ignore</value><!-- Ignore the new play request. -->
            <value>stop</value>   <!-- Stop audio object on new play request. -->
          </choice>
        </attribute>
      </optional>
    <!-- Space-separated list of events that will start the playback of the audio object. -->
    <optional>
      <attribute name="play">
        <ref name="eventList"/>
      </attribute>
    </optional>
    <!-- Space-separated list of events that will pause the playback of the audio object. -->
    <optional>
      <attribute name="pause">
        <ref name="eventList"/>
      </attribute>
    </optional>
    <!-- Space-separated list of events that will reset the playback of the audio object. -->
    <optional>
      <attribute name="reset">
        <ref name="eventList"/>
      </attribute>
    </optional>
    <!-- Space-separated list of events that will stop the playback of the audio object. -->
    <optional>
      <attribute name="stop">
        <ref name="eventList"/>
      </attribute>
    </optional>
    <!--
       Space-separated list of events that will enqueue the audio object.
       The events have additional parameters as follows:
         <event>:<queue_id>[:<max_delay>[:<max_distance>]]
       - event        : the event that will trigger the enqueue action
       - queue_id     : the id of the target queue
       - max_delay    : the maximum acceptable time before playing the audio object, in seconds (default
         = -1, which means indefinite time)
       - max_distance : the maximum acceptable distance from from initial position at the time of the
         event the listener has moved before playing the audio object, in distance unit (default = -1, which
         means indefinite distance)
    -->
    <optional>
```

```
        <attribute name="enqueue">
          <ref name="eventList"/>
        </attribute>
      </optional>
    </define>

    <!-- A sound object -->
    <define name="sound">
      <!--
        A sound is an implicit mixer for 1 or more subsounds.
        Subsounds are introduced to allow for sound variations, but are only seen as being part of one
          sound.
      -->
      <element name="sound">
        <!-- The unique ID of the sound. An ID must be defined for event interactions and explicit mixing of
          this sound. -->
        <optional>
          <attribute name="id">
            <data type="ID"/>
          </attribute>
        </optional>
        <!-- Number of times the sound should repeat playback. -->
        <optional>
          <attribute name="loopCount">
            <data type="int"/>      <!-- Default = 0. If set to -1, the sound will repeat undefinitely. -->
          </attribute>
        </optional>
        <!-- Determines how the subsounds are picked. -->
        <optional>
          <attribute name="pick">
            <choice>
              <value>ordered</value>            <!-- Subsounds are played in order, one after the other
                                                (default value). -->
              <value>reversed</value>           <!-- Subsounds are played in reverse order. -->
              <value>random</value>             <!-- Subsounds are played in random order. -->
              <value>omitMostRecent</value>     <!-- Subsounds are played in reverse order, excluding
                                                 the most recently played subsound. -->
              <value>fixed</value>              <!-- The currently active subsound does not change
                                                (unless explicit active request from a subsound). -->
            </choice>
          </attribute>
        </optional>
        <!-- The playable attributes of the sound. -->
        <ref name="playableAttributes"/>
        <!-- The mix attributes of the sound. -->
        <ref name="mixAttributes"/>
        <!-- The 3D mix setting of the sound. -->
        <ref name="mix3DSettings"/>
        <!-- The audio source file of the implicit first subsound. -->
        <optional>
          <attribute name="src"/>
        </optional>
        <zeroOrMore>
          <ref name="subsound"/>
        </zeroOrMore>
      </element>
    </define>

    <!-- Subsound object -->
    <define name="subsound">
      <element name="subsound">
        <!-- The unique ID of the subsound. An ID must be defined for event interactions with the
          subsound. -->
```

```
            <optional>
               <attribute name="id">
                  <data type="ID"/>
               </attribute>
            </optional>
            <!--
                The audio source of the subsound. It may be either an audio file or a text to be synthesized.
                * In the case of an audio file, the value of this attribute is a file path.
                * In the case of text to be synthesized, the value is formed as follows:
                        voice:[(<language_code[:<pitch>[:<rate>]])]:<text>
                  The parameters are explained below:
                    - language_code : the language code to use, in the ISO form "en-US" (supported: en-US,
                          en-GB, de-DE, es-ES, fr-FR, it-IT)
                    - pitch         : the pitch of the voice (range 50-200)
                    - rate          : the speech rate of the voice (range 20-500)
                    - text          : the text to be synthesized.
                  If one of the optional parameters <language_code>, <pitch> or <rate> is missing, the
                           corresponding value specified by the
                  voice attribute of the sounds element is then used.
            -->
            <attribute name="src"/>
            <!-- Space-separated list of events that will set the current sound as the active one. -->
            <optional>
               <attribute name="setNext">
                  <ref name="eventList"/>
               </attribute>
            </optional>
            <!-- The mix attributes of the subsound. -->
            <ref name="mixAttributes"/>
         </element>
      </define>

      <!-- Queue object -->
      <define name="queue">
         <element name="queue">
            <!-- The unique ID of the queue. -->
            <attribute name="id">
               <data type="ID"/>
            </attribute>
            <!-- Time base of the queue. -->
            <optional>
               <attribute name="timeBase">
                  <choice>
                     <value>playtime</value> <!-- The time is based on the addition of the duration of the
played audio objects, which means pausing the queue also pause the time for the queue (default value). -->
                     <value>realtime</value> <!-- The time is based on the continuous absolute time, which
means pausing the queue has no effect on the time for the queue. -->
                  </choice>
               </attribute>
            </optional>
            <!-- Sorting of the audio objects. -->
            <optional>
               <attribute name="sort">
                  <choice>
                     <value>class</value>       <!-- Sort the incoming audio objects by their priority class
                                                     (default value). -->
                     <value>fifo</value>        <!-- Do no sort the incoming audio objects, the oldest entered
                                                     audio object will always be played first. -->
                  </choice>
               </attribute>
            </optional>
            <!-- Automatically starts playing the queue as new audio objects come in. -->
            <optional>
```

```
            <attribute name="autoPlay">
               <data type="boolean"/>          <!-- Default value is false. -->
            </attribute>
         </optional>
         <!-- Maximum number of audio objects allowed in the queue. -->
         <optional>
            <attribute name="maxElements">
               <data type="int"/>              <!-- Values <= 0 means no limit (default = 0). -->
            </attribute>
         </optional>
         <!-- The time delay between playing audio objects in the queue. -->
         <optional>
            <attribute name="delay">
               <data type="double"/>           <!-- Value must be > 0 (default = 0). -->
            </attribute>
         </optional>
         <!-- The mix attributes of the subsound. -->
         <ref name="playableAttributes"/>
      </element>
   </define>

   <!-- Mixer object -->
   <define name="mixer">
      <element name="mixer">
         <optional>
            <choice>
               <!-- The unique ID of the mixer. An ID must be defined for using this mixer as an input of
                     another mixer. -->
               <attribute name="id">
                  <data type="ID"/>
               </attribute>
               <!-- References an existing mixer ID. If a mixer with the specified ID is found in the audio
                     context, it will be used else a new one will be created. -->
               <attribute name="ref">
                  <data type="ID"/>
               </attribute>
            </choice>
         </optional>
         <!-- The mix attributes. -->
         <ref name="mixAttributes"/>
         <!--
            Coma-separated list of ID references of sounds or mixers to input in this mixer.
            An audio source (sound or mixer) can only have 1 mixer parent.
         -->
         <text/>
      </element>
   </define>

   <!-- A list of events. -->
   <define name="eventList">
      <list>
         <oneOrMore>
            <data type="string"/>
         </oneOrMore>
      </list>
   </define>

</grammar>
```

4.2 Exemple de document audio MAUDL

```
<?xml version="1.0" encoding="UTF-8"?>
<maudl xmlns="http://wam.inrialpes.fr/iaudio/maudl/1.0" id="example">
```

```
        <sounds listenerPos="1.0 0.0 0.0" listenerLookAt="0.0 1.0 0.0" listenerRearAttenuation="0.5"
         listenerFrontAngle="180" rolloff="log" scale="1.0" classOrder="high low"
        voice="en-US:80:100">
                <sound id="music" src="bleep.wav" loopCount="-1" pan="0.0" volume="1.0"
                 play="app.start_music" stop="app.stop_music stop" reset="app.reset_music"
                 replay="ignore"/>
                <sound id="sfx" src="bleep.wav" pick="random" render3D="advanced"
                 pan3DFactor="1.0" pos="1.0 0.0 0.0" play="app.sfx">
                        <subsound src="bleep2.wav" pan="1.0" volume="0.7"/>
                        <subsound src="bleep3.wav" setNext="event.useThisBleep"/>
                </sound>
                <sound id="voice1" src="voice:Testing voice 1 2 3" loopCount="0" pan="0.0"
                volume="1.0" enqueue="app.voice1:ui_voice:5:-1" class="high"/>
                <sound id="voice2" src="voice:(fr-FR:110:90):Test de la voix francaise" loopCount="0"
                 pan="0.0" volume="1.0" enqueue="app.voice2:ui_voice" class="low"/>
        </sounds>

        <queues>
                <queue id="ui_voice" autoPlay="true" timeBase="playtime" maxElements="5"
                 delay="2.0" sort="class"/>
        </queues>

        <mixers volume="0.7">
                <mixer id="submix" volume="0.7">music, sfx</mixer>
                <mixer id="premix">submix</mixer>
        </mixers>

</maudl>
```

4.3 Exemple de panorama audio en OSM

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6">

   <!-- Audio Panorama (Group of POIs) -->
   <relation id="environment">
      <tag k="name" v="current environment"/>
      <tag k="type" v="iap"/>
      <member type="node" ref="init" role="from"/>
      <member type="node" ref="poi1" role="to"/>
      <member type="node" ref="poi2" role="to"/>
      <member type="node" ref="poi3" role="to"/>
      <member type="node" ref="poi4" role="to"/>
      <member type="node" ref="poi5" role="to"/>
   </relation>

   <!-- Auditor initial location and global audio settings -->
   <node id="init" lat="0" lon="0" ele="0" visible="false">
      <tag k="audio:type" v="text/xml"/>
      <tag k="audio:resource" v="./audio/init.xml"/>
   </node>

   <!-- POIs with associated audio stylesheet -->
   <node id="poi1" lat="-1.0" lon="-0.4" ele="0.0" visible="true">
      <tag k="audio:type" v="text/xml"/>
      <tag k="audio:resource" v="./audio/poi1.xml"/>
   </node>
   <node id="poi2" lat="0.4" lon="-0.6" ele="0.0" visible="true">
      <tag k="audio:type" v="text/xml"/>
      <tag k="audio:resource" v="./audio/poi2.xml"/>
   </node>
   <node id="poi3" lat="0.7" lon="0.3" ele="0.0" visible="true">
```

```
    <tag k="audio:type" v="text/xml"/>
    <tag k="audio:resource" v="./audio/poi3.xml"/>
  </node>
  <node id="poi4" lat="0.0" lon="1.0" ele="0" visible="true">
    <tag k="audio:type" v="text/xml"/>
    <tag k="audio:resource" v="./audio/poi4.xml"/>
  </node>
  <node id="poi5" lat="-0.5" lon="0.5" ele="0" visible="true">
    <tag k="audio:type" v="text/xml"/>
    <tag k="audio:resource" v="./audio/poi5.xml"/>
  </node>

</osm>
```

5 Références

[1] iOS, mobile operation system by Apple, http://www.apple.com/ios/

[2] Interactive Audio Rendering Guidelines Level 2, *Interactive Audio SIG*, http://www.iasig.org

[3] Synchronized Multimedia Integration Language, *W3C*, http://www.w3.org/TR/SMIL2/

[4] Interactive XMF, *Interactive Audio Special Interest Group*, http://www.iasig.org./wg/ixwg/

[5] *An Interactive Audio System for Mobile*, Y. Lasorsa, J. Lemordant, 127th AES Convention.

[6] FMOD, *Firelight Technologies Pty, Ltd.*, http://www.fmod.org

[7] Thomas Knott. CORONA - Implementation and Evaluation of Continuous Virtual Audio Spaces for Interactive Exhibits. Master's thesis, RWTH Aachen University, 2009.

[8] S. Sandberg, C. Håkansson, N. Elmqvist, P. Tsigas, F. Chen. Using 3D Audio Guidance to Locate Indoor Static Objects. In Proceedings of the Human Factors and Ergonomics Society 50th Annual Meeting, pp. 1581-1584, 2006.

[9] OpenAL (Open Audio Layer), http://connect.creativelabs.com/openal/default.aspx

[10] Measuring HRIR, Listen HRTF Database, IRCAM, http://recherche.ircam.fr/equipes/salles/listen/system_protocol.html

[11] Listen HRTF Database, IRCAM, http://recherche.ircam.fr/equipes/salles/listen/

[12] OpenSL ES, http://www.khronos.org/opensles/

[13] Android, open source mobile operating system, http://www.android.com/

[14] Evaluation of Spatial Displays for Navigation without Sight, J. Marston, J. Loomis and all, ACM Transactions on Applied Perception, V3, N2, 2006.

[15] Open augmented reality browser, http://www.layar.com/

[16] OpenStreetMap, a collaborative world map in XML, http://www.openstreetmap.org/

6 Sommaire

1 Introduction 3
2 Format pour le guidage audio 3
2.1 Motivations 3
2.2 Audio interactif et navigation 4
2.2.1 Formats existants et limitations 4
2.2.2 Objectifs d'un format adapté au guidage 5
2.3 Le format MAUDL 6
2.3.1 Description 6
2.3.2 Utilisation et exemples 9
2.4 Pointeur audio 3D 14
2.4.1 Objectifs 14
2.4.2 Techniques de spatialisation 3D 15
2.4.3 Mise en oeuvre 19
2.4.4 Utilisation d'un head tracker pour améliorer la spatialisation 20
2.5 Rendu environnemental 21
3 Gestionnaire de sons 23
3.1 Présentation et organisation 23
3.2 La librairie libmaud 23
3.2.1 Fonctionnement de l'API 23
3.2.2 Utilisation et exemples 24
3.2.3 Fonctionnalités avancées 26
3.2.4 Dépendances et limitations actuelles 28
3.3 Interrogation de l'environnement 28
3.3.1 Problématique et solution proposée 28
3.3.2 Mise en œuvre et intégration avec la librairie libmaud 30
4 Annexes 30
4.1 Schéma de validation Relax-NG du format MAUDL 30
4.2 Exemple de document audio MAUDL 37
4.3 Exemple de panorama audio en OSM 38
5 Références 39