Projet µSynth

Rapport de groupe

Benoit Gagnon
Francis Bertrand
Sébastien Bérubé

présenté à Gaspard Petit
dans le cadre du cours IMN638 :
Interactions Visuelles Numériques

Université de Sherbrooke
10 décembre 2008

Table des matières

Table des matières....2
Introduction....3
Fiche descriptive....4
  Objectif....4
  Méthodes....4
  Résultats souhaités....4
Architecture....5
  Construction....5
  Communication....5
Module Vision....7
  Responsabilités....7
  Implémentation....8
  Description détaillée de l'algorithme....9
    Suivi d'objets....12
    Résultats....14
Module Audio....15
  Description....15
  Dépendances....15
  Méthodes....15
Module Visuel....18
  Contraintes....18
    Disques de garde et cercle d'introduction....18
    Contraste et résolution limitée....18
  Open Scene Graph....19
    Utilisation de shaders....19
    Synchronisation....19
    Calibration....19
Conclusion....21

Introduction

Dans le cadre du cours IMN638, les étudiants sont amenés à entreprendre un projet visant à mettre en commun les connaissances acquises durant la session afin d'appliquer une application interactive.

Notre équipe a choisi de combiner cette opportunité avec un autre projet, celui-ci du cours IMN697 (*Projet d'intégration et de recherche*). Le projet est un système interactif audiovisuel permettant le contrôle de composantes sonores à l'aide d'objets tangibles disposés sur une table graphique. L'application se nomme *µSynth*.

Ce document ne se veut pas une référence complète du projet. Nous couvrirons certains aspects de chaque module. Nous porterons une attention particulière sur l'implémentation logicielle et aux améliorations techniques que le cours IMN638 nous a permis d'apporter.

Fiche descriptive

Objectif

Système interactif audiovisuel permettant le contrôle de composantes sonores à l'aide d'objets tangibles disposés sur une table graphique.

Méthodes

- Détection de formes (objets tangibles)
- Synthèse et traitement de son à partir de paramètres provenant du module de détection
- Visualisation divertissante générée à partir de composantes sonores

Résultats souhaités

- La détection des objets, de leur position et de leur orientation devra se faire en temps réel
- Le module de vision devra gérer les occlusions partielles, le retrait et l'ajout d'objets
- La synthèse et le traitement du son se feront en temps réel
- Le module de son offrira assez d'outils à l'utilisateur pour produire une musique agréable
- Le module de visualisation devra opérer en temps réel
- Les images produites seront attrayantes
- Les images doivent servir d'interface secondaire aux objets tangibles et donc augmenter la réponse à l'usager (feedback)
- Le module visuel doit fournir une vue d'ensemble du système

Architecture

Construction

Le système tel que présenté pour ce projet est composé d'une caméra vidéo numérique, d'un projecteur DLP, de trois ordinateurs, de câbles réseau, d'une *switch* et d'une table d'environ 4' de hauteur et d'environ 36″ par 24″ d'envergure.

La surface de la table comporte une feuille d'acrylique parfaitement transparent d'environ ¼" d'épaisseur ainsi que d'une feuille de papier calque blanc non granulé. Le papier calque sert de surface de projection mais aussi de filtre du point de vue de la caméra ; les objets ne sont pas visibles tant qu'ils ne font pas contact avec la surface.

Sous la table, un miroir incliné à 45°permet de rediriger les rayons du projecteur vers le dessous de la surface, tandis qu'un plus petit miroir parallèle à la surface permet de filmer le dessous de la table du point de vue de la caméra qui est fixée au châssis. La caméra est reliée à un ordinateur par un câble *FireWire*. Le projecteur est placé face au miroir incliné, parallèle au sol. Afin de couvrir toute la surface sans utiliser de lentille grand angle, il est à environ deux pieds derrière la table. Il est relié à un autre ordinateur par un câble DVI ou VGA.

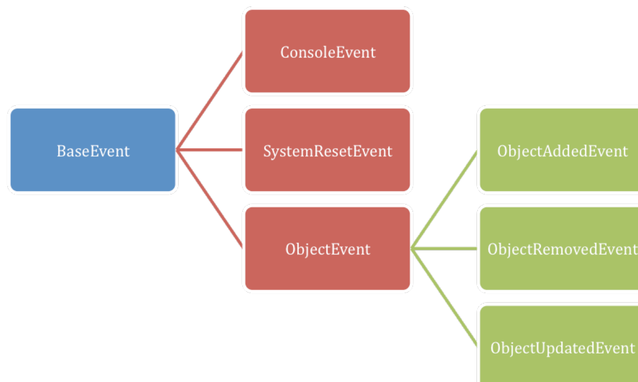
Le troisième ordinateur est relié par réseau aux deux autres et est branché sur un système de son conventionnel.

Communication

La partie logicielle de µSynth est formé de trois applications distinctes; le module de vision, le module audio et le module visuel. Ces trois applications communiquent entre elles par l'intermédiaire d'événements transmis sur réseau local UDP. Une telle architecture n'empêche toutefois pas d'exécuter tous les modules sur un seul ordinateur ; il suffit d'utiliser l'adresse IP locale et des ports de communication différents.

Pour communiquer l'état des objets tangibles sur la table, le module de vision envoie des événements qui sont reçus par le module visuel ainsi que le module audio. Ces deux modules ne produisent aucun événement en sortie, pas même une confirmation à la réception des événements. Pour cette raison, il est important de ne pas perdre de paquets dans la transmission. Dans notre installation, trois ordinateurs étaient reliés par des câbles ethernet via une *switch* gigabit. Le débit de transmission étant ultimement liée au nombre d'images par seconde captées et analysées par le module de vision, le problème de perte de paquets s'est avéré négligeable.

On compte trois événements principaux dans le système, soit l'ajout d'un objet, le retrait d'un objet ainsi qu'une modification d'un objet (position et/ou orientation). De plus, un événement simple de réinitialisation est disponible. Aussi, un événement de texte générique permet de transmettre un message à la console d'affichage de la surface.



Concrètement, ces événements forment une hiérarchie de classes C++. Un mécanisme de sérialisation permet de convertir tout événement en chaine de caractères. Inversement, un mécanisme de désérialisation permet de reconstruire un objet d'événement à partir d'une chaîne de caractères. Ces mécanismes utilisent la fonctionnalité RTTI offerte par le C++ pour déterminer la classe d'événement à partir d'un pointeur de celui-ci. Les *smart pointers* de la librairie Boost sont utilisés pour simplifier la gestion de mémoire de ces objets volatiles.

La transmission en temps que telle est cachée sous un objet de haut niveau (un conduit) par lequel entre et sortent des instances d'événements. Les appels cet objet sont tous synchrones et bloquant. Il n'y a aucune attente active dans les processus de communication, ce qui réduit énormément l'impact de cette couche présente dans les trois applications du système. Pour la portabilité, on utilise les *sockets* de la librairie Qt qui fonctionne sous Linux, Windows et Mac OS X.

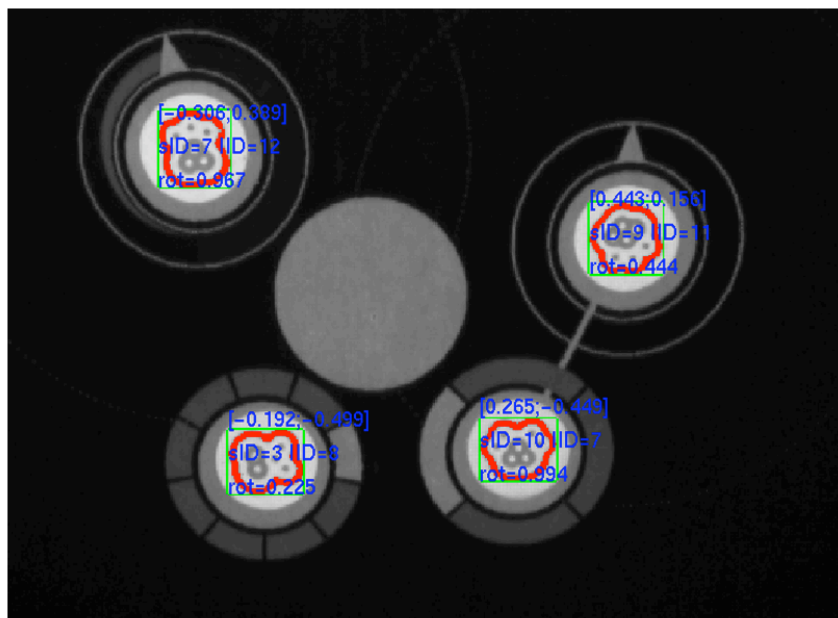
Module Vision

Responsabilités

Le module de vision s'occupe de la détection et du suivi d'objets. C'est aussi à ce module que revient la tâche de générer et de filtrer les évènements d'ajout, de retrait et de mise à jour des objets lors des mouvements. Les évènements de mise-à-jour générés contiennent les informations suivantes :

- Numéro du type d'objet
- Numéro d'instance de l'objet détecté
- Position de l'objet, en coordonnés logiques rectifiées
- Orientation de l'objet
- Dérivée de la position et de la rotation
- Statut de l'objet

Afin de réduire le nombre d'évènements, aucun évènement n'est envoyé pour un objet stationnaire. De plus, les coordonnées de position, d'orientation et de leurs dérivées sont filtrées afin d'augmenter la précision et la stabilité du système. L'utilisation d'un numéro d'instance permet de différencier les instances différentes d'un même type d'objet détecté à plus d'un endroit à la fois dans une image de la vidéo.



Implémentation

Le module de vision utilise les API et librairies suivantes :

- PortVideo 0.3 - Capture des images de la caméra
- OpenGL - Affichage des traitements et de la vidéo, en temps réel
- Trolltech Qt 4.4.2 - Interface usager

L'implémentation du module de vision est fortement inspirée des deux articles suivants :

[1] *Improved Topological Fiducial Tracking in the reacTIVision System*
Ross Bencina, Martin Kaltenbrunner and Sergi Jordà, *ICMC2005,* 2005

[2] *reacTIVision: A Computer-Vision Framework for Table-Based Tangible Interaction*
Kaltenbrunner, M. & Bencina, R, TEI07, 2007

Toutefois, nous n'avons utilisé aucun code ou API existant afin d'implémenter notre algorithme de vision. La lecture des articles mentionnés ci-haut nous a servi de ligne directrice, mais notre algorithme final en diverge quelque peu.

Notre implémentation est adaptée aux besoins et contraintes spécifiques au projet *uSynth*. Contrairement à la *reacTable*, nous n'utilisons pas d'illuminateur infrarouge ni de caméra infrarouge, et nous ne pouvons nous fier qu'à l'illumination dispensée par le projeteur. Cette contrainte affecte le module de vision selon 2 aspects, soit l'absolue nécessité d'afficher une couleur unie suffisamment claire sous les symboles, ainsi que le niveau d'interférence supplémentaire qu'implique l'utilisation de la caméra et du projecteur dans une même plage du spectre de la lumière.

Fig. I (gauche) Image de symbole avec du « *motion blur* » horizontal. (droite) Reconnaissance de la position et du contour du symbole malgré la segmentation imparfaite. Le symbole ainsi altéré ne peut toutefois pas être décodé.

La solution à ces deux problèmes n'est pas couverte par les articles sur lesquels nous avons basé notre système.

Ainsi, les deux plus grands facteurs de risque d'échec du module de vision étaient les suivants :

- Incertitude quant au traitement du nombre de régions supplémentaire généré par l'utilisation de la caméra dans le spectre visible de la lumière.
- Incertitude relative à la synchronisation de la caméra et du projecteur.

Les facteurs de risques mentionnés précédemment expliquent les distinctions opérationnelles de *µSynthVision* comparativement à *reacTIVision*:

Avantages

- Vitesse d'exécution très peu influencée par le nombre de régions de l'image segmentée, contrairement à *reacTIVision*.

- Possibilité de continuer le suivi de position d'objets malgré la corruption partielle du symbole segmenté.

Inconvénients

- Valide pour une profondeur de champ plus limité que *reacTIVision*.
- Contraintes plus strictes relativement au design des symboles, en particulier concernant la nécessité d'établir une aire constante de la région constituant la bordure des symboles.

Description détaillée de l'algorithme

Seuillage (diffère très légèrement de l'algorithme de *reacTIVision*)

L'étape de seuillage est très semblable à celle décrite dans l'article *« Improved Topological Fiducial Tracking in the reacTIVision System »*. L'image est d'abord divisée en tuiles (25 x 25 pixels). Le niveau de gris maximal et minimal interne à chaque tuile est identifié, et un seuil est fixé en calculant simplement la moyenne des deux extremums. Chaque pixel de la tuile est ensuite étiqueté à l'aide d'un seuillage binaire. Pour des considérations de performances, aucune vérification du voisinage n'est effectuée. Tel que mentionné dans l'article [1], il est important d'utiliser des tuiles de calcul de seuil légèrement plus grandes que celles servant à appliquer le seuillage. Pour l'implémentation de *SynthVision*, nous avons utilisé des tuiles de calcul de seuil débordant de 2 pixels relativement aux tuiles d'application du seuillage. Un tel débordement des tuiles de calcul du seuil permet aux tuiles d'application de mieux s'intégrer les unes aux autres et règle certains problèmes de fusion de régions, particulièrement dans le cas où des tuiles voisines présentent une grande différence de contraste.

Segmentation (diffère légèrement de l'algorithme de *reacTIVision*)

Tout comme pour *reacTIVision*, la segmentation des régions se fait à l'aide d'un algorithme non-récursif procédant à une analyse ligne par ligne de l'image seuillée. Lors de cette procédure, des données caractérisant chaque région telles que le nombre de pixels de la région et le rectangle englobant sont extraites. L'utilisation de tampons de régions pré-alloués permet d'éviter toute forme d'allocation dynamique. Chaque pixel se voir attribuer une étiquette de région, formant ainsi une image d'étiquettes. Toutefois, aucune relation d'adjacence n'est extraite à ce moment dans l'algorithme, et c'est à ce niveau que *µSynthVision* diverge de l'algorithme original de *reacTIVision* pour lequel les graphes d'adjacences sont générés en même temps que la segmentation des régions.

Sélection des régions (diffère totalement de l'algorithme de *reacTIVision*)

Pour plusieurs raisons incluant avant tout le coût associé au traitement de chacun d'eux, les arbres d'adjacence ne sont crées qu'à l'intérieur des régions représentant de bon candidats de bordure de symboles. Considérant la morphologie relativement uniforme des symboles utilisés, il est très facile d'effectuer un tri préalable sur les

régions extraites afin d'éliminer la presque totalité des régions ne présentant pas les caractéristiques recherchées.

Pour *µSynthVision*, ces caractéristiques recherchées sont les suivantes :

- Aire de la région (nombre de pixels)
- Hauteur de la région
- Largeur de la région

Aussi simple que cela puisse paraître, une sélection des régions présentant un nombre de pixels, une hauteur et une largeur respectant les tolérances dictées par des valeurs-étalons constitue effectivement un discriminant extrêmement efficace, même en milieu fortement bruité. Les valeurs-étalons peuvent être fixées à l'aide d'une étape de calibration pour laquelle l'utilisateur doit sélectionner une région de contours de symbole. Ainsi, seules les régions candidates passeront à l'étape suivante, soit la génération du graphe d'adjacence. Ce tri constitue l'élément-clef permettant à l'algorithme de *µSynthVision* d'afficher une vitesse d'exécution non-proportionnelle au nombre de régions générées lors de la segmentation. C'est aussi l'élément qui limite la profondeur de champ pour laquelle les symboles peuvent être reconnus, ce qui ne nous gêne toutefois pas le moindrement dans le contexte de la reconnaissance de symboles sur surface fixe tel que pour le projet *µSynth*.

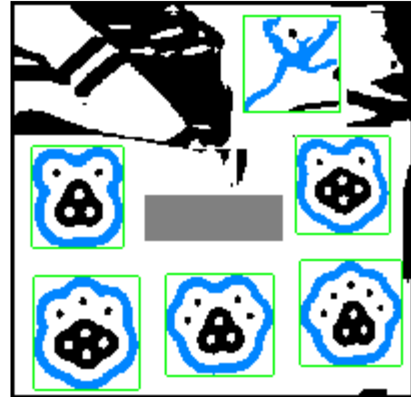


Fig. II Les régions candidates présentant les caractéristiques recherchées sont identifiées. Ces régions représentent potentiellement la bordure du symbole. Les faux-positifs sont rares et ne ralentissent pas significativement l'algorithme.

Génération des arbres d'adjacence (diffère de *reacTIVision*)

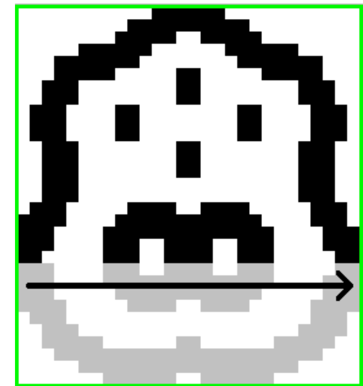


Fig. III Les relations hiérarchiques de régions sont obtenues par une simple vérification horizontale unilatérale. Chaque nouvelle région orpheline d'un pixel de droite est nécessairement l'enfant de la région précédente du pixel de gauche.

Lors de la génération des arbres d'adjacence, le rectangle englobant de chaque bordure de région potentielle sélectionnée est utilisé comme limite d'analyse. Tout comme la segmentation, la génération de l'arbre d'adjacence se fait ligne par ligne, pixel par pixel, de gauche à droite. Cette fois par contre, c'est l'image d'étiquettes qui est balayée. L'algorithme est initialisé dans un coin du rectangle englobant, et les transitions d'étiquettes entre deux pixels voisins sont identifiées comme étant un lien de parenté entre deux régions de l'arbre. Pour chaque région, une liste d'enfants ainsi que l'unique parent sont conservés. L'existence de l'adjacence mutuelle est vérifiée à chaque fois qu'un changement d'étiquette est observé entre 2 pixels voisins. Parce que chaque région du symbole peut possiblement avoir plusieurs enfants, mais seulement un parent, la vérification de l'existence d'un lien entre deux régions peut se faire en temps contant, sans avoir à parcourir la liste d'enfants à chaque vérification.

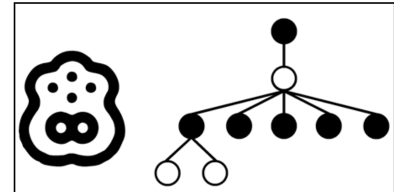


Fig. IV Arbre d'adjacence associé au symbole qu'il représente

En effet, il suffit de vérifier que la région associée à l'étiquette du pixel de droite s'est déjà vue attribuer un lien vers sa seule et unique région mère. Lorsque ce lien n'est pas encore établi, alors il est possible d'en déduire que la région associée au pixel de droite est un enfant de la région associée au pixel de gauche. Cette simplification est possible grâce à la constatation du fait qu'on ne peut atteindre une région « orpheline » qu'en passant par sa région mère d'abord.

En d'autres mots, toute ligne qui traverse le symbole de bord en bord, lorsqu'on la suit d'un bout à l'autre, doit nécessairement passer par le parent avant d'atteindre l'enfant. Effectivement, la nature morphologique des symboles implique que chaque région enfant soit entourée dans toutes les directions par sa région mère. Ainsi, une simple vérification horizontale unilatérale permet d'identifier tous les liens d'un arbre de symbole. Ces constatations sont importantes et permettent aussi bien de réduire le nombre de comparaisons de voisins que de réduire la complexité algorithmique, donnant lieu à un algorithme rapide de génération d'arbre fonctionnant en temps linéaire du nombre de pixels présents à l'intérieur des régions candidates.

Décodage des arbres d'adjacence (identique à *reacTIVision*)

Une procédure de décodage telle que celle proposée dans l'article [1] est effectué. Il est à noter qu'une représentation particulière des arbres doit être utilisée afin de faire abstraction de l'ordonnancement des enfants de chaque région. Pour y parvenir, les arbres sont transformée en équivalence « *left-heavy* », où les branches les plus lourdes et profondes sont déplacées vers la gauche. Suite à cette transformation, l'arbre est parcouru afin de générer le « *left-heavy depth sequence* ». Cette séquence représente l'identifiant unique du symbole. La séquence obtenue est comparée aux entrées d'un dictionnaire de séquences connues.

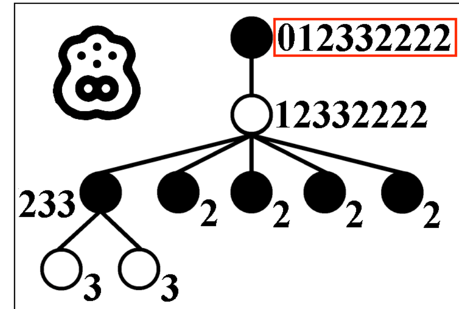


Fig. V – Séquence identifiant l'arbre d'adjacence et le symbole.

Calcul de position et d'orientation (diffère légèrement de *reacTIVision*)

Pour chaque arbre d'adjacence décodé valide, la position et l'orientation peuvent ensuite être déterminés. La position est simplement approximée comme étant le centre du rectangle englobant de la bordure du symbole. L'orientation, quand à elle, est calculée tel que décrit dans l'article [1], soit en trouvant le vecteur qui relie le centre de masse des feuilles blanches au centre de masse des feuilles noires. Les coordonnées obtenues sont normalisées selon la correspondance de la zone d'affichage du projecteur avec celle de la caméra. Une transformation perspective est utilisée afin de corriger les déformations introduites par le miroir.

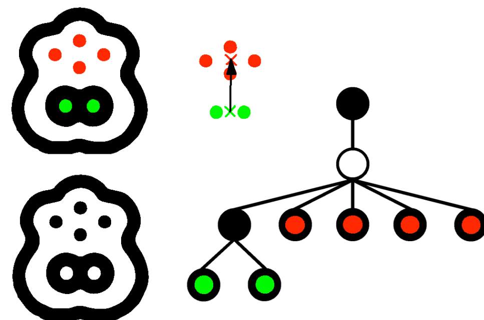


Fig. VI – Obtention du vecteur d'orientation par l'utilisation du centre de masse des feuilles noires et du centre de masse des feuilles blanches.

Suivi d'objets (diffère beaucoup de reacTIVision)

Plusieurs objectifs motivent la mise en place d'un algorithme de suivi d'objet. En plus de connaitre la position et l'orientation de chaque objet, il est important de savoir à quel moment un nouvel objet est introduit, retiré ou a simplement bougé. De plus, afin de réduire le trafic d'événements, l'état stationnaire d'un objet doit aussi pouvoir être reconnu afin d'éviter d'envoyer continuellement de l'information qui le concerne, alors que son état ne change pas. Ainsi, pour chaque objet identifié, un historique des dernières informations de position et d'orientation est construit. De plus, on enregistre la dérivée de l'orientation et de la position dans chaque entrée de l'historique. Une fois l'historique en place, il est facile de déterminer des règles pour

l'activation et la désactivation de chaque objet. De manière plus précise, l'activation d'un objet peut se faire lorsqu'un certain nombre de détections positives sont rencontrées. Pour notre implémentation, nous avons fixé ce seuil d'activation à 15 détections positives. À l'inverse, la désactivation d'un objet se fait lorsque qu'aucune détection positive n'est trouvée pour un objet donnée depuis un certain délai. Nous avons fixé arbitrairement ce délai à 750ms, afin de conserver une certaine robustesse dans le cas où la détection d'un objet échoue momentanément. L'état stationnaire est quand à lui interprété du fait qu'un objet ne présente pas de variation significative de rotation ou de position. Cette tolérance est fixé à +/-1 pixel et à +/- 1 degré pour les 5 dernière entrées de l'historique de l'objet.

En plus de gérer l'activation, la désactivation et l'état stationnaire de chaque objet, le module de suivi d'objet doit aussi gérer l'association des instances de chaque objet décodé aux instances d'historique. L'association des instances se fait par des critères de proximité et de code de symbole. Il est à noter qu'il est possible d'introduire deux objets identifiés par le même symbole. C'est ainsi l'instance de l'objet qui doit les différencier. De plus, la gestion des instances permet de renforcer la détection de la position des objets, car dans l'éventualité où un objet est victime d'une erreur de segmentation, il est possible de retracer la position de la région de contours à proximité de la dernière entrée de l'historique d'un objet et de continuer à suivre son mouvement (voir Fig.1) même si le décodage de l'arbre d'adjacence échoue.

Enfin, chaque objet se voit attribuer une dérivée temporelle de mouvement et de rotation. Cette dérivée est filtrée à l'aide d'une moyenne des dérivées des 5 dernières entrées de l'historique d'un symbole. Cette information est utilisée par le module visuel.

Résultats

L'intégration des divers algorithmes utilisés donne un résultat relativement rapide et efficace. L'algorithme intégré occupe 40% du processeur à une cadence de 30 fps, ce qui est un peu plus lent que les résultats obtenus selon les publications faites sur le *reacTIVision*.

Librairie	Utilisation CPU	Fréquence
µSynthVision	40%	30fps
libfidtrack (*reacTIVision*)	18%	30fps
libdtouch (*reacTIVision*)	82%	30fps
dtouch_old	100%	10fps

Table 1 : Utilisation CPU et fréquence d'acquisition pour le suivi de 12 symboles à une résolution de 640x480. Les tests pour *µSynthVision* ont été effectués sur un Pentium M 1.86GHz sous Windows XP alors que les autres tests ont étés effectués sur un Pentium IV 2.6GHz sous Fedora Linux 3.

Il est toutefois à noter que *µSynthVision* a été développé dans l'optique d'obtenir un système de vision dont le temps d'exécution est insensible au nombre de régions généré par le procédé de segmentation. Il est tout à fait envisageable d'effectuer de la détection de symboles utilisant *µSynthVision* avec de la vidéo complètement remplie de petites régions fortement contrastées, ce qui ne semble pas être le cas avec *reacTIVision* selon ce qui est mentionné dans l'article [1] :

« One property of our system which allowed us to select an efficient algorithm was that correct thresholding is only required on image areas containing clean, well lit fiducial images within a known scale range. The segmentation process tends to take time proportional to the number of black and white regions in the image, so ideally our thresholder should output non-fiducial areas in a single colour. »

Un comparatif plus élaboré des deux algorithmes devrait toutefois être effectué avant de pouvoir tirer cette conclusion.

Module Audio

Description

Le module audio de *µSynth* permet à l'usager d'interagir à l'aide des objets tangibles avec certaines composantes sonores. L'immersion de l'usager est primordiale et c'est pourquoi nous avons opté pour une interface simple possédant un comportement facile à interpréter.

Dépendances

Tout d'abord, le système de génération de son utilise la librairie [1]STK (*Synthesis ToolKit*) développée par Perry R. Cook de l'Université de Princeton et Gary P. Scavone de l'Université de McGill. STK est une librairie *open source* permettant le traitement de signal audio et présentant des algorithmes de synthèse sonore. Développée en C++, STK permet la génération de son en temps réel et sur plusieurs plateformes (Linux, Mac OS X, Windows).

De plus, afin de continuer dans l'optique que notre système soit multiplateforme, nous avons opté d'utiliser la libraire QT[1] qui permet, entre autre, le développement d'interface graphique multiplateforme. C'est pourquoi, à travers le module audio, certaines fonctionnalités ont été développées avec l'aide des utilitaires QT.

Méthodes

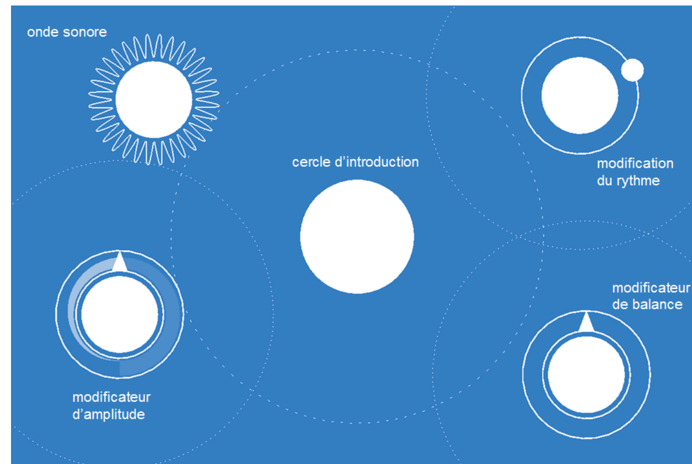
Afin d'atteindre notre objectif d'offrir une interface facile d'utilisation, il était clair que l'interaction entre les objets devait être intuitive. C'est pourquoi d'ailleurs que le module visuel avait comme objectif d'augmenter la réponse à l'usager.

Premièrement, les objets tangibles permettant la mixture des composantes sonores sont classés en 2 types distincts :

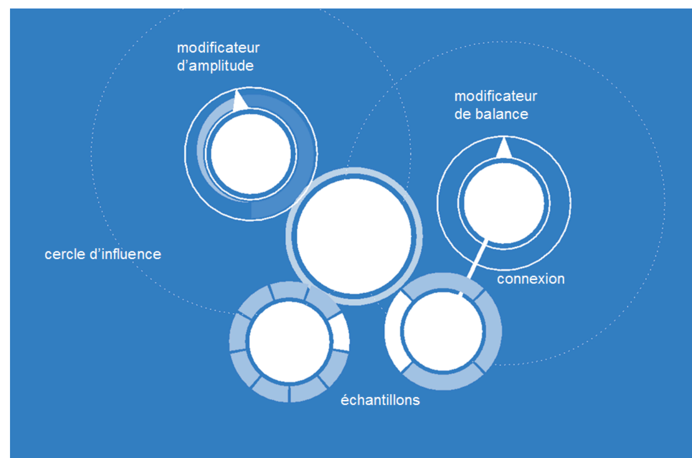
- Objets générateurs de son
- Objets modificateurs de son

L'interaction entre les objets est basée sur deux bases simples : l'orientation et la position des objets. Les générateurs de son détiennent un paramètre modifiable déterminé par leur orientation. Afin de les activer, il suffit de les placer dans le cercle d'activation disposé au centre de la table. Tous les autres paramètres sont modifiables à travers les objets modificateurs. En effet, afin d'altérer les sons générateurs, il suffit de connecter les modificateurs à ceux-ci en les disposant à une certaine distance des générateurs (cercle d'influence). Suite à cette disposition, une connexion se fera et l'orientation permettra de transformer les composantes sonores.

[1] Synthesis ToolKit - http://ccrma.stanford.edu/software/stk/index.html



Exemple du cercle d'action



Exemple de connexion entre objets

Objets générateurs de son

- Objet d'échantillons
  Utilisé pour stocker des fichiers sonores (*.wav*). Son activation dépend de la position sur la table. Son orientation détermine le fichier en lecture.
- Objet d'onde sonore
  Utilisé pour représenter une onde sonore. Son activation dépend de la position sur la table. Son orientation détermine sa fréquence.

Objets modificateurs de son

- Modificateur d'amplitude
  Utilisé pour modifier le gain des objets générateurs. Son orientation détermine l'amplitude du son.
- Modificateur d'échantillonnage
  Utilisé pour modifier la fréquence d'échantillonnage des objets générateurs. Les changements d'orientation augmentent ou diminue la fréquence.

- Modificateur de balance
  Utilisé pour modifier le gain relatif gauche/droite des objets générateurs. Son orientation détermine la balance du son.

Afin d'obtenir une réponse profitable à l'utilisateur, il était nécessaire d'obtenir une génération sonore en temps réel. Pour ce faire, une entité (*SoundStream*) a été développée afin de gérer la composition sonore. Cet utilitaire permet définir une fonction spécifique réalisant ces tâches. Cette fonction est directement liée avec la carte de son de l'ordinateur. L'appel se fait automatiquement par STK et les paramètres d'appel permettent d'obtenir le tampon de lecture de la carte et de le modifier à notre guise.

Module Visuel

Contraintes

Une particularité importante du système est que le module de vision opère dans le spectre visible de la lumière, et non dans l'infrarouge comme dans le système *Reactable*. En conséquence de quoi, le module visuel constitue une interférence direct au module de vision. Autrement dit, tout ce qui est projeté peut être vu par la caméra et peut venir perturber le bon fonctionnement de la détection de symbole.

Disques de garde et cercle d'introduction

La calibration des paramètres de détection est faite sous une illumination maximale (blanche) du projecteur. Ainsi, les symboles sont bien détectés uniquement si la zone où ils se trouvent est parfaitement blanche. Pour assurer cette condition, un cercle blanc doit suivre les objets tangibles (des rondelles de hockey) en tout temps. Le disque doit être légèrement plus grand que la rondelle afin de permettre les déplacements tout en tenant compte de la latence du système.

Le premier prototype fonctionnel nécessitait de très grands disques pour permettre les déplacements à vitesse raisonnable. Or, ces disques sont encombrant visuellement et réduisent les possibilités d'affichage.

Pour remédier à ce problème, la taille des disques est constamment ajustée en fonction de l'amplitude du vecteur de vitesse. Cette information est disponible grâce aux dérivées de positions contenues dans chaque événement de mise à jour. Ainsi, les disques peuvent devenir très grand lorsque l'usager déplace rapidement un objet, mais retournent se « serrer » au contour de l'objet lorsque celui-ci ralentit. Fait intéressant, l'effet est peu perceptible par l'utilisateur et peut même être perçu comme un effet purement esthétique.

Toutefois, un problème demeure : où introduire de nouveaux objets si ceux-ci n'ont pas encore de disque de garde ? On a choisi de tricher au jeu de l'œuf et la poule en ajoutant simplement un deuxième œuf : un disque permanent au centre de la table. C'est à cet endroit que les outils peuvent être introduits. À noter que le module de vision ne traite pas différemment cette zone.

Contraste et résolution limitée

La puissance de la lampe du projecteur influence directement le rapport de contraste disponible pour l'affichage. Aussi, la résolution limitée du projecteur (1280x960 dans notre installation) et la grande taille de la surface font que les pixels deviennent sont facilement visibles si on affiche des texture ou de la géométrie trop fine. Les interfaces des outils doivent être facilement identifiables par l'utilisateur.

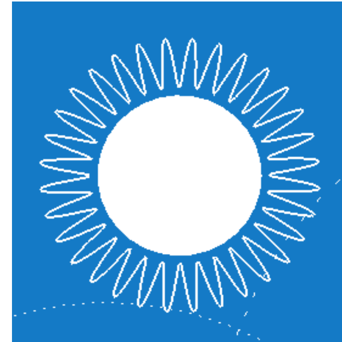
Or, la rétroprojection et le type surface rendent certains effets, autrement peu impressionnant sur des écrans LCD standards, assez impressionnant. Par exemple, l'outil de modification de la couleur du fond, simple changement de la composante *hue* du modèle HSV, est un des outils les plus appréciés bien qu'il ne génère aucun son.

Open Scene Graph

Au cœur du module visuel, on retrouve la populaire librairie Open Scene Graph. Cette librairie multi-plateforme facilite grandement l'affichage et la mise à jour de l'état de la scène. Ce qu'offre la librairie, c'est une riche hiérarchie de classes (des nœuds) que l'on peut placer dans un graphe pour représenter la scène. Les nœuds représentent de la géométrie, des transformations et tout changement d'état dans le rendu OpenGL. Des *callbacks* sont attachés à certains nœuds que l'on désire mettre à jour à chaque image.

Utilisation de shaders

Open Scene Graph rend très facile l'utilisation du GPU sur certaines parties (sous-graphes) du graphe de scène. Par exemple, l'outil de génération d'onde de notre système offre une interface amusante représentant l'onde générée.



Interface de l'outil de génération d'onde

Visuellement, il s'agit d'une onde circulaire qui oscille légèrement en amplitude et qui change de phase dans le sens horaire. Ceci donne une impression de rotation. La phase et la fréquence de l'ondre affichée sont modifiées selon les mises à jour reçues.

On utilise pour ce faire des variables uniformes ainsi qu'un *vertex shader*. La géométrie passée à la carte graphique est un simple cercle avec une grande résolution de points (720 points). Ceux-ci sont éloignés ou rapprochés du centre selon une fonction sinus paramétrée par les variables uniformes. Ces variables sont modifiées d'une image à l'autre, créant l'animation.

Synchronisation

Le rendu est beaucoup plus rapide que le débit de réception des événements. Il est donc primordial de séparer ces deux tâches dans des fils d'exécution distincts. Par contre, il faut faire attention à la façon dont Open Scene Graph parcoure le graphe, sans quoi les crashs sont très faciles à produire. La solution utilisée dans notre projet fut de placer toute demande de mise à jour dans des files d'attentes protégées par des mutexes. On laisse Open Scene Graph appeler les *callbacks* de mise à jour comme bon lui semble. Lorsque le callback est appelée, on verrouille le mutex et on traite tout ce qui se trouve dans la file. De cette façon, la couche de communication n'entre pas en compétition avec les mécanisme de mise à jour ou d'affichage d'Open Scene Graph.

En pratique, les files de modifications sont presque toujours vide car la fréquence de mise à jour d'Open Scene Graph est facilement de plusieurs centaines d'images par seconde. Bien entendu, le projecteur est ultimement limité à 60Hz.

Calibration

Il est assez difficile d'aligner physiquement (en déplaçant le projecteur) l'image produire par le projecteur à la surface de la table. Plutôt que d'essayer d'y parvenir, on a choisi de positionner et de redimensionner l'image de façon logicielle. Ainsi, on cherche d'abord à projeter un rectangle suffisamment grand pour couvrir (et dépasser) la zone semi

transparente de la table. Ensuite, on ajuste la correction trapèze à l'aide du menu du projecteur. Le reste de la calibration se fait à l'aide du clavier.

Il reste simplement à aligner la dimension, les proportions ainsi que l'emplacement logique du *viewport* OpenGL sur l'ensemble de la résolution disponible. Pour faciliter ce travail, une grille de calibration est affichable en pesant sur la touche « g » du clavier. Les informations sont clairement visibles à l'écran. Le ratio largeur/hauteur, par exemple, doit être connu du module audio afin que celui-ci réagisse de la même façon que le module visuel dans les calculs de distances cartésiennes.

Les flèches permettent de déplacer en X et en Y, alors que d'autres touches intuitives permettent d'agrandir, réduire, étirer ou allonger le *viewport*.

Évidemment, la calibration doit être faite de paire avec le module de vision. On cherche à placer quatre cercles le plus près possible des coins physiques de la table tout en étant visibles par la caméra. Un dernier paramètre est la taille nominale des cercles pour l'illumination des outils. Cette fonctionnalité rend le système neutre par rapport à la résolution, à la distance du projecteur et au rayon des objets tangibles.

Une fois la calibration acceptée, elle est sauvegardée en appuyant sur « Return ». Au prochain démarrage de l'application, elle est automatiquement restaurée.

Conclusion

Somme toute, le mot «multi» est à l'honneur dans le système *µSynth* ; multi-sensoriel, multi-processus, multi-thread et multi-plateforme. Les différentes composantes logiciels utilisent plusieurs librairies externes, ce qui a grandement facilité le développement.

Mis à part la portion logicielle, le système final est très réactif et stable. Les différents outils permettent de créer des effets amusants pour les oreilles et les yeux. Bien entendu, on pourrait ajouter beaucoup plus d'effets visuels et sonores.

Par exemple, un prototype de piano virtuel fut développé pour le système mais retiré de la version finale. Un outil permettait de choisir le type d'instrument à synthétiser tandis que des touches projetées sur la surface permettaient de jouer des airs simples tout en intégrant le son d'autres outils. Ce genre d'ajouts au système le rendraient plus riche et plus complet.

Par ailleurs, il serait intéressant de considérer des effets visuels réagissant d'avantage aux sons produits. La visualisation de la musique est très populaire ces temps-ci et ajouterait encore plus de richesse à l'affichage.

N'oublions toutefois pas les limites du système ; la vitesse de déplacement des objets, quoique assez satisfaisante selon les utilisateurs qui en ont fait l'essai, demeure limitée. Aussi, le mixage des sons pourrait être améliorée en ajoutant un ajustement automatique de la cadence en fonction du tempo des échantillons, comme le font les logiciels comme *Garage Band* ou *Soundtrack*. Le module de vision, quant à lui, pourrait être simplifié davantage, notamment par rapport à la calibration.

En conclusion, nous pensons que le projet *µSynth* est très bonne synthèse des techniques vues dans le cours d'interactions numériques, mais aussi dans l'ensemble de notre baccalauréat en Imagerie et Médias Numériques. Nous espérons qu'il puisse inspirer d'autres étudiants à s'inscrire dans ce programme et à développer eux aussi des solutions interactives.