École Polytechnique de Montréal

Département de génie informatique

HumanLocator

Vision artificielle pour oeuvres d'art interactives

Rapport de projet de fin d'études soumis
comme condition partielle à l'obtention du
diplôme de baccalauréat en ingénierie.

Présenté par: Samuel Audet
Matricule: 1049949
Directeur de projet: Bastien Beauchamp
Entreprise: Freeset Interactive Entertainment

Date: 15 avril 2003

Sommaire

Ce que j'ai essayé de bâtir avec ce projet est un système de peinture interactive. Il s'agit en gros d'un système de vision artificielle qui serait probablement utile pour beaucoup d'oeuvres d'art interactive. L'art interactif est un art dans lequel on utilise de la technologie pour permettre à une oeuvre de réagir en fonction de diverses actions accomplies par les observateurs actifs.

Il y a plusieurs façon de recueillir de l'information sur l'environnement, mais la façon que j'ai préconisé est l'utilisation d'une caméra ordinaire et d'un traitement et d'une analyse de l'image assez standards. C'est la méthode qui permettait d'avoir à la fois d'avoir accès à une image que l'on pourrait pas la suite projeter et de mettre le plus à profit mes connaissances informatiques. C'est également la méthode qui offre le plus de possibilités quant à l'ajout de nouvelles fonctionnalités.

Le système est ensuite divisé en deux parties. La première partie, HumanLocator, reçoit en provenant de la caméra les images et procède au traitement et à l'analyse. Le traitement et l'analyse se divise en cinq grandes étapes: la soustraction de l'arrière-plan, la binarisation, l'érosion et la dilatation, la segmentation en blobs et l'analyse du mouvement. Les informations que cette partie calculent sont ensuite acheminés à la deuxième partie, celle qui doit projeter des animations. Dans notre cas, il s'agit de Macromedia Director aidé d'un Xtra que j'ai développé. Le tout a été programmé dans Windows NT 5 (Windows 2000) à l'aide Microsoft Visual C++ 6.0, la trousse de développement de DirectX 8.1b et la bibliothèque XML nommée Xerces X++ 2.20.

Table des matières

Remerciements

Je tiens à remercier mon directeur de projet Bastien Beauchamp pour son support financier et artistique au projet. Il fut de plus un merveilleux compagnon de travail. Je tiens de plus à remercier mon co-directeur Michel Dagenais pour avoir endossé mon projet et m'avoir supporté durant mes démarches d'ingénierie, de même que Paul Cohen pour m'avoir accordé de son temps pour m'entretenir de l'analyse stéréoscopique et de d'autres types d'analyses d'images.

Liste des tables

Liste des figures

Glossaire

API: Application programming interface, ou interface de programmation de l'application est une description des différentes fonctions et objets qu'une bibliothèque logiciel possède, permettant à un programme d'y avoir accès.

blob: Un blob est un ensemble de pixels qui se trouvent connectés les uns aux autres dans une image binaire donnée.

filtre IIR: Le filtre « infinite impulse response » est un type de filtre qui utilise les valeurs précédemment calculées de façon récursive. Il possède donc en théorie une réponse infinie à une impulsion. Même si ce n'est pas le cas lors de calculs numériques, il est quand même préférable de se satisfaire d'un certain pourcentage de la valeur à atteindre lors du calcul du temps de réponse.

FOV: Field of view, ou le champ angulaire d'une caméra représente l'angle du champ de vision de la caméra.

IPC: Inter-process communication, ou communication inter-processus est nécessaire comme les zones mémoire de différents processus sont séparés. Plusieurs méthodes, dont les pipes et la mémoire partagée permettent cette communication.

langage script: Un langage script est langage simple de programmation normalement non typé. Un langage que même des gens qui ne sont pas programmeurs, comme des artistes, pourraient utiliser.

MFC: Microsoft Foundation Classes est une trousse d'objets qui font un emballage objet des fonctions Win32. Elles sont fortement intégrées à Microsoft Visual C++ 6.0 et donc la

programmation MFC réside moins dans le design de classes que dans la conception visuelle de l'interface graphique elle-même.

pattern matching: Le pattern matching est le processus par lequel d'après un patron A, il est possible de localiser sa position dans une image B.

plug-in: Un plug-in est programme qui vient se greffer à l'espace mémoire d'un autre programme en utilisant un protocole défini par ce dernier. Il est alors possible de rajouter des fonctionnalités à ce programme à l'aide d'un plug-in.

servo: Un servo est un organe du moteur qui sert a contrôler sa direction et sa vitesse.

STL: Standard Template Library, une bibliothèque de classes modèles standard au C++. Elle possède beaucoup d'algorithmes intéressants, et dans le cas de HumanLocator, les algorithmes de tris sont particulièrement utiles.

table de conversion: Une table de conversion, ou look-up table, permet de calculer une liste de résultats prédéterminés à l'avance et dont le but est d'éviter de recalculer le même résultat des milliers de fois.

Win32: Win32 est l'API du système d'exploitation Windows NT. Un processus roulant sur Windows NT doit donc se conformer à ce qu'il peut trouver dans Win32.

Introduction

Longtemps les arts et la science ont vécu séparés, mais depuis une nouvelle façon d'utiliser la technologie a fait naître les arts interactifs. L'utilisation de la technologie pour l'art permet la création de nouvelles oeuvres qui ne fut pas possible de créer auparavant. D'autant plus que ces oeuvres sont la plupart du temps interactives, car voilà le but principale de la technologie dans les arts interactifs. Dans ce projet, nous avons tenté de bâtir un système de peinture interactif.

Le premier chapitre, la problématique, décrit avec plus de détails la position sur les arts interactifs et la description artistique du système de peinture interactif que nous avons comme vision. Il y sera aussi question des principaux problèmes que nous avons dû appréhendé et la motivation de l'entreprise d'un tel projet.

Le deuxième chapitre, la méthodologie, décrit tout d'abord les différentes façons que la technologie actuelle nous donne comme moyens pour recueillir de l'information sur un environnement. Les manières que nous avons analysé dans ce deuxième chapitre sont: les sonars, les capteurs infrarouges, les lidars, les capteurs de mouvement 3D, les analyses stéréoscopiques, les caméras infrarouges, les tapis avec capteurs de pression et l'utilisation d'une caméra normal avec un traitement et une analyse standard de l'image. J'ai décrit pour chacun leurs avantages et leurs inconvénients. Nous allons de plus voir pourquoi j'ai décidé d'utiliser une caméra normal et le traitement et l'analyse standard d'image. Finalement, je justifie mon choix de configuration logiciel et j'indique de plus structure général que j'ai choisi pour le système global, soit une partie qui s'occupe du traitement et l'analyse de l'image, HumanLocator, et une autre

partie qui s'occupe d'afficher des animations en réponse aux événements envoyés par la première partie, Macromedia Director et un Xtra.

Le troisième chapitre, les résultats, expose en détails toute les parties logicielles du système. Il est d'abord question du langage de programmation, des outils et des bibliothèques utilisées. Toute l'information pour permettre à quelqu'un de rapidement commencer avec le code source s'y trouve. Ensuite, j'articule les détails d'implantation du programme principale, HumanLocator. On y retrouve un diagramme de classe avec une description de chaque classe de même pour les attributs et les méthodes associés que je trouvais important de décrire. Puis, j'ai pris une capture d'écran de l'interface usager que je présente avec une table des fonctionnalités. Par la suite, les algorithmes complets qui sont utilisés par les traitements et les analyses des images sont décrits. Nous retrouvons la soustraction de l'arrière-plan, la binarisation, l'érosion et la dilatation, la segmentation en blobs et l'analyse du mouvement. Ces opérations de traitement et d'analyse de l'image sont bien connus, mais la façon dont je les ai implanté et mis ensemble demande explication. De plus, quelques informations sur le développement de l'Xtra pour Macromedia Director sont présentées. Finalement, je liste quelques paragraphes sur comment faire la correspondance entre les coordonnées sur les images en provenance de la caméra et les coordonnées de l'environnement réel. Il ne s'agit que de la simple géométrie.

1 Problématique

1.1 Les arts interactifs

Les arts interactifs sont des arts en tant que tel qui utilisent la technologie pour permettre aux visiteurs d'interagir avec les oeuvres. Il s'agit d'un mariage entre la technologie et les arts. Par exemple, une oeuvre pourrait posséder un capteur qui ferait en sorte que la projection d'un certain film débute lorsqu'une personne entre dans la salle. Les possibilités de ce genre d'oeuvre ne sont pas seulement limitées par l'imagination des artistes, mais aussi par la technologie. Cependant, dernièrement, la technologie numérique s'est avérée très intéressante et les applications possibles de celle-ci dans le domaine des arts interactifs sont devenues incalculables.

Il s'est déjà fait beaucoup d'oeuvres qui utilisent un projecteur pour projeter des couleurs, des patrons, des images et des animations sur des peintures ou des sculptures. Certaines ont aussi un degré d'interactivité limité grâce à l'utilisation de capteurs permettant de récupérer de l'information sur la présence et la position des personnes et des objets. On peut ainsi par exemple passer notre corps ou notre main à certains endroits pour déclencher certaines fonctions. Il est aussi possible de placer des capteurs sur le plancher et de faire réagir l'image projetée en fonction de la position d'objets lourds (des humains) qui s'y trouvent.

1.2 Système de peinture interactive

Le but de ce projet est de pousser cette interactivité un peu plus loin. Il s'agit de permettre à un système de voir plus de choses, d'être plus intelligent en quelque sorte. Dans le cadre de ce projet, je me limite à un « système de peinture interactive ». Un système informatique sera branché à un projecteur qui diffusera des animations sur une toile déjà peinturée. La toile pourra avoir des espaces laissés intentionnellement blancs pour permettre la projection des animations telles quelles ou on pourra jouer avec les couleurs de la toile et celles du projecteur. Il y a beaucoup de paramètres et de problèmes reliés à la projection sur une toile pour peinture, mais ces considérations ne sont pas à l'étude dans ce rapport.

Le système informatique devra recevoir des informations de son environnement à l'aide de capteurs ou de caméras. L'analyse de l'information devra permettre au système de trouver la position des gens qui se trouvent en face de la peinture et de comprendre leurs mouvements. Éventuellement, il serait aussi souhaitable d'être en mesure de percevoir les gestes de ces personnes, ce qui laisserait la porte encore plus grande ouverte à l'imagination des artistes.

1.3 Problèmes reliés à l'implantation

Pour un ordinateur, il y a deux méthodes principales pour recueillir de l'information sur l'environnement. Il y d'une part les capteurs et d'autre part les caméras. Il y a plusieurs types de capteurs: les lidars, les sonars, les radars, les capteurs de pression, et beaucoup d'autres qui ne

pourraient être utiles pour notre projet. Cependant, les capteurs retournent souvent une information ponctuelle et il est souvent difficile d'obtenir une vue globale de l'environnement. De plus, il serait probablement nécessaire de bâtir notre propre matériel, car il ne s'agit pas de produits habituellement utilisés dans l'industrie, sauf à des fins très spécifiques.

En ce qui a trait aux caméras, de celles qui nous seraient utiles, il y a les caméras infrarouges et les caméras ordinaires. L'analyse de l'image avec une caméra infrarouge serait probablement plus facile, car les points chauds de l'image correspondraient pour la plupart à la position des humains dans la salle. D'une façon ou d'une autre, il faudrait procéder à l'analyse des images. La technologie informatique permettant d'analyser de plus en plus d'informations de plus en plus rapidement est devenue impressionnante certes, mais la capacité d'analyse visuelle des ordinateurs est encore de loin exécrable si on la compare avec celle des humains. Comme les caméras autant celles infrarouges que celles ordinaires sont très largement répandues, le principale problème pour cette solution serait l'analyse de l'image.

1.4 Motivation de l'entreprise

Il n'y a que quelques décennies qui se sont écoulées depuis que les technologies ont vraiment commencées à se tailler une place parmi les arts. Par exemple, le cinéma est né de cette union, mais il y a aussi les arts plus traditionnels qui peuvent profiter des nouveaux moyens mis à notre disposition. Depuis peu, l'avènement des technologies numériques et multimédias a permis aux entrepreneurs artistes désireux de créer de toute pièce des types d'oeuvre qui n'avaient jusqu'alors jamais existé.

Le monde des mathématiques, de l'informatique, du multimédia et des arts est un monde captivant dans lequel l'innovation ne s'arrête pas. Dans le cadre de mes études à l'École Polytechnique de Montréal, j'ai étudié beaucoup de concepts mathématiques et de techniques informatiques. Ce projet me permet de mettre à profit ces nouveaux atouts en plus de contribuer à un projet artistique de plus grande envergure.

2 Méthodologie

2.1 Capture d'informations sur l'environnement

Comme mentionné auparavant, plusieurs possibilités s'offrent à nous en terme d'implantation. Il y a tout d'abord la gamme des capteurs, les caméras infrarouges et les caméras ordinaires de même que les différentes façon de les utiliser. Chacune de ces façons sont décrites dans les paragraphes qui suivent. Les avantages et les inconvénients y sont aussi précisés. Le système parfait ne devrait pas demander aux usagers de porter quoi que ce soit et pourrait aussi fournir une image de la personne en vu d'une utilisation artistique ultérieur.

2.1.1 Sonar, capteur infrarouge et lidar

Les genres de capteurs qui nous seraient utiles sont pour la plupart ceux utilisés en robotique. Il nous faut quelque chose pour voir ce qu'il y a en face de nous. Il y a les sonars, les capteurs infrarouges et les lidars.

Le sonar (« sound navigation ranging ») est un capteur qui fonctionne avec les ultrasons. Tout d'abord un émetteur envoie une onde sonore. Celle-ci est réfléchie sur un objet en face du sonar et revient vers le capteur qui reçoit ensuite l'onde ainsi réfléchie après quelques millisecondes. Cela permet de savoir à quelle profondeur se trouve un objet en face. Le sonar fonctionne mieux sous l'eau, car les ondes sonores voyagent mieux sous l'eau. La porté d'un sonar dans l'air est au maximum d'environ 10 mètres, ce qui n'est pas un problème pour une salle intérieure. De plus, les courants perturbent beaucoup la précision de l'appareil [9], mais comme nous voulons l'utiliser à l'intérieur, ce n'est pas un problème. Toutefois, la précision d'un tel appareil laisse beaucoup à désirer. La résolution est en effet limitée à un angle d'environ 2 degrés [9] . Il y a beaucoup d'interférences dans ce genre de système et il est compliqué de réussir à avoir des résultats viables. Il faudrait de plus un appareil capable de faire un balayage horizontal mécanique pour trouver la position horizontal de l'objet en face.

Le coût de ce genre de système est de quelques centaines de dollars (voir un marchand tel que http://www.robot-electronics.co.uk/) , ce qui est acceptable, mais tous ces appareils demandent l'utilisation d'un servo, ce qui requiert le montage d'un système électrique et mécanique. De plus,

le nombre d'images par seconde serait limité au maximum à environ 10 images pour un angle de vue d'environ 30º à cause du balayage mis en place.

Les capteurs infrarouges fonctionnent comme le sonar, mais émettent une onde infrarouge au lieu d'une onde sonore. La précision de ce genre d'appareil en ce qui a trait à l'angle de vue n'est pas non plus impressionnante et il n'est de plus pas utilisé pour ce genre d'application. On utilise plutôt le lidar (« light detection and ranging »), aussi appelé ladar (« laser detection and ranging ») et laser rangefinder, qui utilise un laser infrarouge comme source de lumière, ce qui réduit de beaucoup l'interférence d'une source infrarouge ordinaire. Ainsi, le lidar offre une bonne précision d'environ 0,05 degrés [3]. Toutefois, ce genre de système est encore assez dispendieux (voir un marchand tel que http://www.internationaltool.com/leica.htm) et coûte dans les 1000$. En outre, il demande aussi à être manuellement monté sur un servo.

2.1.2 Capteur de mouvement 3D

Les « motion tracker » sont des systèmes qui permettent de trouver la position exacte de la tête ou de diverses parties d'une personne. On doit porter un casque, des gangs ou d'autres vêtements nécessaires pour le système, ce qui est un critère avec lequel on peut déjà rejeter cette solution. La porté de ce genre de système est de l'ordre de quelques mètres, ce qui serait suffisant. Les prix de ce genre de système commencent à environ 3000-7000. Les manufacturiers les plus populaires sont:

Polhemus http://www.polhemus.com/

Ascension http://www.ascension-tech.com/

Logitech http://www.logitech.com/

2.1.3 Caméra infrarouge

Une caméra infrarouge nous permettrait de facilement trouver la position des gens dans une salle, car ces derniers seraient les objets chauds de l'environnement. Toutefois, une telle caméra coûte au minimum 5000$. La raison de ce coût si élevé est que les lentilles traditionnelles en verre ne sont pas perméables à la lumière infrarouge que le corps humain émet (longueur d'onde d'environ 10 um). Une lentille en germanium, silicon ou selenide de zinc [16] doit être utilisée.

2.1.4 Analyse stéréoscopique

Les concepts de la stéréoscopie nous permettraient de faire une analyse sur deux images en provenance de deux caméras séparées par une certaine distance et de retrouver les coordonnées en 3D des objets de la salle. Toutefois, selon Paul Cohen, un expert en intelligence artificielle et en vision artificielle, les techniques ne sont pas encore assez mûres. Dans le cadre d'un tel projet, l'utilisation de l'analyse stéréoscopique ne serait pas justifiée vu la quantité de développement et la puissance de traitement requise pour obtenir un résultat qui ne serait peut-être pas si intéressant.

2.1.5 Caméra ordinaire

L'utilisation d'une caméra ordinaire nous donnerait la possibilité de voir exactement ce qu'un humain verrait. De plus, en postant la caméra à quelques mètres de haut et en la faisant regarder

par en bas, il serait possible de trouver d'une façon satisfaisante pour le projet les coordonnées 3D d'objets placés à la verticale dans une scène. Toutefois, pour segmenter les objets d'intérêt de la scène et pour par la suite faire une analyse sur ces objets, il faudrait développer un programme complexe pour traiter l'image. Une idée proposée par Paul Cohen serait de forcer le port de botte jaune ou d'un autre accoutrement par les visiteurs, pour donner une chance au système de facilement repérer les personnes. Une petite Webcam qui pourrait peut-être bien fonctionner assez bien coûte 150$, alors qu'une bonne caméra industrielle, au moins 1500$.

2.1.6 Tapis avec capteurs de pression

Une autre chose qui serait praticable serait de fabriquer un tapis qui possède des capteurs de pression. Cela entraverait peu avec les usagers, mais le système ne pourrait être mis à jour avec de nouvelles fonctionnalités comme l'analyse et la détection de gestes. De plus, il ne permettrait pas d'avoir une image de la personne.

2.1.7 Méthode choisie

Finalement, la méthode choisie est l'utilisation d'une seule caméra ordinaire. Même si cela requiert un traitement et une analyse intense de l'image, il ne saura pas pour autant question de demander aux usagers de porter un vêtement quelconque. Je considère que mes compétences en informatique dans ce domaine sont à un niveau suffisant pour me permettre d'atteindre le but visé même dans ces conditions.

Les raisons de ce choix sont multiples. Tout d'abord, le montage d'un sonar ou d'un lidar demanderait une grande compétence en mécanique et en électrique, ce que je ne possède pas. Ensuite, une caméra infrarouge coûte beaucoup trop cher pour le gain qu'elle pourrait nous apporter par rapport à une caméra normal. L'utilisation d'un capteur de mouvement 3D demanderait aux usagers de porter quelque chose pour que le système fonctionne. Finalement, le tapis avec des capteurs de pression ne nous permettrait pas de mettre à jour le système avec de nouvelles fonctionnalités comme la détection de gestes. Enfin, tous ces systèmes ne nous permettent pas non plus d'avoir une image de la personne en face.

2.2 Configuration logiciel

Comme nous le verrons dans la section *Projection d'images et d'animations*, j'ai décidé d'utiliser Macromedia Director comme outil qui a comme tâche l'affichage d'animations intéressantes. Comme ce logiciel ne fonctionnait que sous MacOS et Windows, le choix de la plateforme de développement était limité. J'ai choisi Windows NT, car c'était un bon système d'exploitation en tant que tel et de plus, Bastien Beauchamp, son équipe et moi-même ne possédions pas de Macintosh.

Le système logiciel est donc séparé en deux parties. Il y a tout d'abord la partie qui doit faire le traitement et l'analyse de l'image. Cette partie s'occupe de recevoir les images en provenance de la caméra et permet de détecter la position des personnes en face de la peinture, et de comprendre leurs mouvements et, dans l'avenir, leurs gestes. Ensuite, la deuxième partie reçoit ces

informations et génère en temps réel des animations appropriées qui sont alors projetées sur la toile.

2.2.1 Le traitement et l'analyse de l'image

Comme nous utilisons une simple caméra pour recueillir de l'information sur l'environnement, il est nécessaire de faire un traitement et une analyse sur les images en provenance de la caméra. Ainsi, tout d'abord j'ai procédé à une recherche afin de trouver des solutions d'imagerie logicielle permettant de facilement recevoir les images en provenance de la caméra et de leur faire subir un traitement approprié le plus simplement possible. On peut retrouver en annexe la liste des produits que j'ai trouvés de même les prix des trousses pour développeur et des bibliothèques d'exécution qui sont utilisées pour les machines qui doivent rouler le produit développé. J'ai contacté quelques entreprises pour connaître ces prix et en général, on peut voir qu'une licence de développeur coûte entre 2000$ et 4000$ et qu'une licence pour la bibliothèque d'exécution, entre 300$ et 2000$. Le budget pour le projet étant quelque chose d'important, je me suis donc demandé si je ne serais capable de faire le traitement et l'analyse nécessaires par moi même. J'ai aussi évalué des bibliothèques « open source », IPL 98 et CVIPTools en particulier, mais ces bibliothèques ne possèdent pas de module pour permettre l'accès à la caméra. Après avoir analysé mes besoins, j'ai décidé d'y aller avec VideoOCX, car il ne coûtait pas cher, il me permettait de facilement accéder à la caméra et de plus, il possédait quelques fonctionnalités de traitement d'images. Toutefois, au milieu de la phase de développement, j'ai été obligé de laisser tomber VideoOCX, car il ne permettait pas de faire l'interface avec les caméras FireWire.

Comme il y a des bonnes chances que des caméras FireWire soit utilisées, j'ai dû utiliser DirectShow directement pour les faire fonctionner.

Pour ce qui est du traitement et de l'analyse de l'image, j'ai établi une procédure de base me permettant de trouver des « blobs » dans une scène, en espérant que ces blobs soient bel et bien des humains. La procédure n'utilise pas de technique faisant un usage significatif des statistiques, des états ou de l'intelligence artificielle. Cela m'a permis de grandement simplifier le développement du logiciel. Toutefois, il reste à voir si cela permet d'obtenir un système fonctionnant de façon satisfaisante. Nous en discutons plus tard dans ce rapport.

La façon dont les images en provenance de la caméra sont traitées est décrite dans la figure 2.1. Avant d'être en mesure d'extraire une information utile, l'image subit diverses traitements divisés en plusieurs étapes. Cette méthode de faire provient en grande partie de [15].

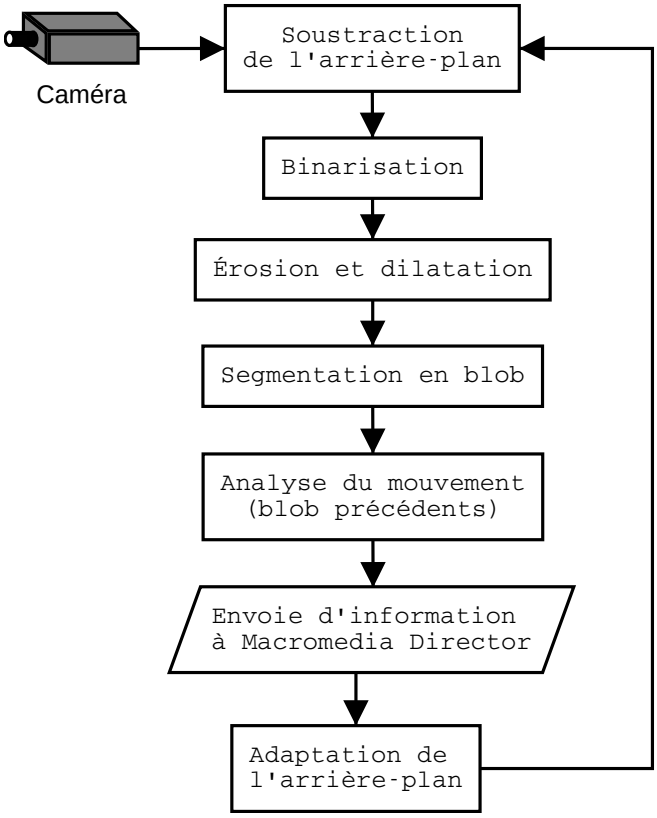


Figure 2.1 - Étapes de traitement et d'analyse

Tout d'abord, on soustrait l'arrière-plan de l'image, ce qui extrait de celle-ci beaucoup de détails non pertinents à l'analyse en faisant abstraction de ce qui est considéré comme l'arrière-plan. On peut voir cela comme la mémoire « cognitive » de l'ordinateur pour ce qui est de l'arrière-plan. Ensuite, on fait une binarisation pour réduire la quantité d'informations nécessaires à analyser dans les étapes suivantes. La binarisation devrait prendre compte autant de la différence des couleurs que de la différence de luminance. Puis nous devons faire une érosion de l'image binarisée pour faire disparaître le bruit restant, et ceci suivie par une dilatation pour boucher les trous qui peuvent se trouver dans l'image binarisée. Finalement, une segmentation en blobs permet de trouver tous les points de l'image qui ont une valeur non nulle et qui sont attachés à d'autres points ayant aussi une valeur non nulle. Nous espérons que ces blobs soient des

personnes. Les blobs doivent posséder des proportions minimales définies à l'exécution pour que le système ne pas tiennent pas compte d'objets trop petits. Ensuite, on fait une analyse du mouvement à l'aide de la distance entre les blobs de l'image courante et ceux de l'image précédente. Finalement, l'image représentant l'arrière-plan est ajustée de façon graduelle en assimilant les pixels de l'image courante qui n'ont pas été classés comme blobs. Cela devrait permettre au système de s'adapter graduellement aux changements de luminosité ou aux déplacements de petits objets. En outre, l'information extraite est ensuite acheminée à l'application de projection d'images et d'animations comme définie plus bas.

Spécifications requises

En dernier lieu, il est important de décrire quel genre de performance que l'on attend d'un tel programme. Comme il s'agit de bâtir un système interactif, il faut qu'il puisse calculer et répondre à la personne en face de la peinture le plus rapidement possible. Nous espérons de plus utiliser le système sur des ordinateurs personnels, ce qui réduirait d'autant plus le coût matériel. Ainsi, pour qu'un humain considère une réponse comme « rapide », elle doit se faire sentir dans les 100~200 ms [7]. Si le temps de réponse est trop lent, on perd la fluidité et par la même occasion l'attention des gens. Il faudrait donc que le programme puisse traiter au moins 5 à 10 images par seconde. Il faudrait de plus être en mesure d'obtenir cette performance sur un ordinateur personnel qui peut rouler Windows NT, c'est à dire possédant un CPU d'AMD ou d'Intel, et de modèle AMD Athlon XP 2200 ou Intel Pentium 4 2.4 Ghz, si on tient compte de leur performance et de leur coût actuellement raisonnable, ou plus rapide. Il faudrait de plus

laisser environ la moitié de la puissance du CPU à Macromedia Director, comme décrit dans la section suivante, afin de pouvoir afficher des animations intéressantes.

2.2.2 Projection d'images et d'animations

La première chose qui fut considérée pour permettre au système de projeter des images est le développement d'un nouvel environnement de travail. Il s'agissait de créer un langage script, ou de se baser sur des langages existants, qui aurait permis à l'artiste d'afficher les choses qu'il aurait voulu que le système affiche à la suite de divers événements en provenance de la partie traitement et analyse de l'image. Il existe quelques langages scripts qui permettent de manipuler des objets multimédias, comme des séquences vidéos, ce qui est le plus important dans notre cas. Avisynth [23] a tout d'abord été considéré, mais il ne permet pas d'afficher différentes séquences vidéos à différents endroits dans une fenêtre, ce qui est la principale fonction dont nous avons besoin. Ensuite, nous avons considéré SMIL [29] (Synchronized Multimedia Integration Language) qui permet de faire ce que nous voulons à l'aide d'attributs décrits en XML. Cela semblait le parfait outil pour le travail, mais SMIL était encore en développement et ne possédait pas une bonne base de lecteur et d'éditeur SMIL. Le meilleur lecteur SMIL, possiblement le seul qui aurait pu être utile, était RealOne Player. De plus, Internet Explorer 6.0 supportait XTML+SMIL, mais de part sa vocation il est plus orienté pour la navigation Internet que pour la présentation multimédia, ce qui aurait sûrement compliqué les choses lors de son utilisation. Finalement, peu d'éditeur SMIL populaire existe encore. Les artistes n'auraient sûrement pas été confortables de développer dans une tel environnement. De plus, les possibilités d'extension d'un éditeur et d'un lecteur SMIL (à l'aide d'un plug-in ou de d'autres moyens) sont encore mal définies, ce qui aurait

rendu d'autant plus complexe la tâche de communiquer les informations de HumanLocator vers les outils SMIL.

Plus tard, nous avons parlé de Macromedia Director [24] et Macromedia Flash[26], les outils que les artistes du multimédia utilisent souvent. En regardant les spécifications de Macromedia Director et Macromedia Flash, il s'est avéré que Director possédait un système de plug-in, les Xtra, et un langage script, le Lingo. Il était ainsi possible de faire communiquer Director avec le monde extérieur à l'aide d'un Xtra et de permettre au designer d'utiliser l'information de HumanLocator à l'aide de Lingo. Pour ce qui est de Flash, il ne possédait pas de moyen de communication avec l'extérieur, alors c'est une option que nous avons dû laisser tomber.

En somme, l'outil choisie pour permettre l'affichage d'images et d'animations fut Macromedia Director. Non seulement c'est un outil fort populaire dans le monde artistique, mais il possède déjà les deux conditions nécessaires à son adoption, soit la possibilité de communiquer avec le monde extérieur et d'écrire des scripts pour pouvoir utiliser l'information reçue.

2.3 Le système d'ensemble préconisé

Le système au complet consiste à l'utilisation d'un ordinateur de calcul (sans moniteur, clavier ou souris), d'une caméra et d'un projecteur par oeuvre. Comme on peut le voir dans la figure 2.2, les ordinateurs sont reliés en réseau et il y a un ordinateur central, avec moniteur, clavier et souris, qui sert de poste de contrôle pour les autres.

Chaque ordinateur de calcul procède à une analyse des images reçues en provenance des caméras. Cette analyse permet de trouver dans la salle la position des personnes, leurs mouvements et peut-être même leurs gestes. Les informations recueillies de cette analyse, sont d'abord acheminées à Macromedia Director à l'aide du plug-in Xtra, ce qui permet ensuite à un artiste d'afficher des images et des animations en fonction de la position, des mouvements et des gestes des observateurs en face de la peinture.

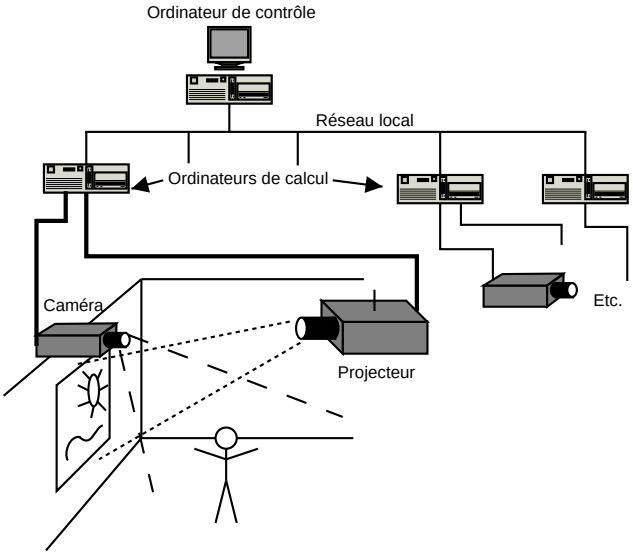


Figure 2.2 - Représentation physique du système

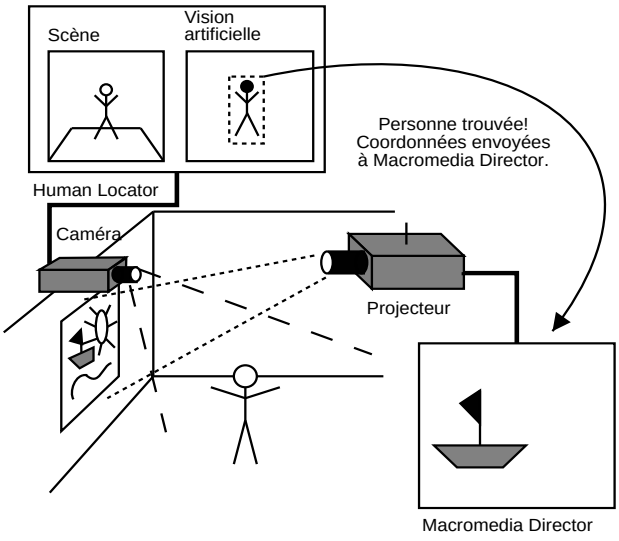


Figure 2.3 - Représentation fonctionnelle du système

Le système HumanLocator procède à une analyse de l'image de la façon suivante et illustré dans le figure 2.3. Tout d'abord, une image de l'arrière-plan doit être mise en mémoire. En enlevant l'arrière-plan de l'image courante, l'ordinateur peut savoir lorsqu'il y a un objet d'intérêt qui entre dans la scène. À l'aide de limites empiriques des changements possibles de la luminosité et de la couleur d'une zone d'intérêt, le système est en mesure de discriminer ces zones du reste de l'image. Ensuite, on compense pour le bruit de la caméra avec une procédure d'érosion et de dilatation. Il

est alors possible de segmenter des régions où se trouvent des objets de la taille requise. De plus, avec une analyse des objets de la scène précédente, le système est capable de comprendre le mouvements des gens. Une analyse plus poussée permettra aussi de tenir compte des gestes des personnes.

3 Résultats

Ce chapitre porte sur les résultats obtenus du projet. Les résultats concrets, c'est à dire le programme de HumanLocator et de l'Xtra, se trouve sur le CD inclu dans la pochette avec ce rapport. Toutefois, les sections suivantes ont pour but de faire mieux comprendre la façon dont le tout a été développé et de faire mieux comprendre le fonctionnement de HumanLocator et de l'Xtra. Le contenu du CD est le suivant décrit dans la table 3.1.

Nom du fichier ou du répertoire	***Description***
HumanLocator.pdf	ce rapport
HumanLocator.exe	l'application principale de HumanLocator
HLPipeXtra.x32	l'Xtra de Macromedia Director
xerces_c2_2_0.dll	la bibliothèque de Xerces C++
HLSystem\	répertoire contenant le code source de HumanLocator.exe et HLPipeXtra.x32
rapport_source\	répertoire contenant tous les fichiers sources de ce rapport (fichiers OpenOffice 1.0 et Dia 0.90)
xerces_c2_2_0-win32\	répertoire contenant le code source de la bibliothèque de Xerces C++

Table 3.1 - Contenu du CD

Tout d'abord, je donne une courte description du langage de programmation, des outils et des bibliothèques que j'ai utilisés. Cela permettra à qui veut modifier les programmes de facilement

s'armer des bons outils lui permettant de rapidement commencer la tâche. Les deux sections suivantes concernent les détails d'implantation de ces deux programmes. Il est question de la description des classes et des concepts et des algorithmes utilisés.

3.1 Langage de programmation, outils et bibliothèques

Tout d'abord, le système d'exploitation utilisé pour le développement a été de Windows 2000, ou Windows NT 5. C'était un système d'exploitation stable et il contenait toutes les nouvelles fonctionnalités requises pour le multimédia. De plus, il fallait pouvoir rouler Macromedia Director en parallèle. Celui-ci ne roule que sur Windows NT et Mac OS, comme mentionné auparavant.

Le langage de programmation choisi est le C++. C'était le langage le plus pratique pour accéder aux fonctionnalités systèmes de Windows NT comme DirectX et Win32. Il était de plus rapide et la nature objet orienté du C++ m'a permis de mieux organiser certaines parties du projet. Par ailleurs, l'environnement de développement utilisé fut Microsoft Visual C++ 6.0 [28]. Il s'agissait de l'environnement le plus utilisé dans le domaine sous Windows NT et je le connaissais bien.

Ensuite, pour accéder à la caméra depuis HumanLocator, il a fallu utiliser DirectShow à l'aide du DirectX SDK 8.1b [27] et utiliser les interfaces appropriées comme ISampleGrabber pour obtenir les images de la caméra. La programmation DirectShow était très complexe, mais comme il ne s'agissait que de trouver et de programmer les bonnes interfaces, aucune information

supplémentaire sur la programmation DirectShow ne sera donnée. D'ailleurs, pour obtenir les fonctionnalités requises dans HumanLocator, la presque totalité du code provient des échantillons mentionnés ci-dessous et qui se trouvaient dans le SDK. S.V.P veuillez vous référer à la documentation du SDK et aux répertoires d'échantillon suivants pour des exemples sur l'utilisation de DirectShow en ce qui concerne la capture d'image:

```
samples\Multimedia\DirectShow\Capture\AMCap
samples\Multimedia\DirectShow\Capture\DVApp
samples\Multimedia\DirectShow\Editing\SampGrabCB
```

Ensuite, pour la programmation de l'Xtra pour Macromedia Director, j'ai utilisé le Macromedia Directory 8.5 SDK [25], gratuitement téléchargeable. Pour faciliter la communication entre HumanLocator et l'Xtra, j'ai décidé d'utiliser les « named piped » systèmes de Windows NT et d'encoder les informations en XML à l'aide de Xerces C++ Parser [22]. Celui-ci étant déjà compilé pour Windows NT, simple d'utilisation et gratuit lorsqu'utilisé pour des produits non commerciaux, il était parfait pour la tâche.

3.2 Développement de HumanLocator

HumanLocator est le programme qui reçoit les images de la caméra et procède au traitement et à l'analyse de ces images. Il est programmé en C++ et possède plusieurs classes, mais n'est pas un programme fortement objet orienté pour autant. La raison de cela est que la plupart du traitement consiste en des algorithmes mathématiques qui se prêtent bien à la programmation structurée telle quelle, sans objet. HumanLocator possède de plus une interface graphique qui permet à l'usager de choisir sa caméra et les options qui y sont associées. Il permet de plus de voir le résultat du

traitement à la suite des différentes étapes de traitement. Il autorise aussi l'usager à changer la plupart des options affectant les opérations de traitement pour permettre une optimisation plus simple des valeurs.

Le traitement et l'analyse consistent en la plus grosse partie du travail. Les différentes étapes, soit la soustraction de l'arrière-plan, la binarisation, l'érosion et la dilatation, la segmentation en blobs et l'analyse du mouvement, sont décrites en détails dans les sections suivantes. Pour les expliquer, il sera aussi question de formules mathématiques et d'algorithmes.

Les fichiers se rapportant au code source de HumanLocator se trouvent dans les répertoires suivants du CD:

```
HLSystem\HumanLocator
HLSystem\common
```

`HLSystem\HumanLocator\HumanLocator.dsw` est le fichier projet de Microsoft Visual C++ 6.0.

3.2.1 Diagramme des classes + description des classes

HumanLocator n'est pas un programme très objet orienté. Il n'y a pas d'héritage de classe et la plupart des classes ne possèdent que des méthodes statiques. Ces classes agissent plus comme modules que comme autre chose. La raison de cela est que les algorithmes de traitement et d'analyse utilisés se rapportent plus à des fonctions mathématiques qu'à des objets logiciels. Ainsi, la structure du programme se rapproche plus à celle d'un simple programme structuré.

Cela dit, la figure 3.1 montre un diagramme des classes simplifié pour mieux comprendre la structure globale du programme. CHumanLocatorDlg est la classe mère, c'est à dire celle qui est née en premier, et est de même la classe qui est responsable de l'interface usager et qui gère la connexion avec la caméra. Chacune des classes est décrite dans les sous-sections suivantes. Je donne une courte description de la classe ainsi que des descriptions pour les attributs et les méthodes pour lesquels j'ai jugé bon de préciser certaines choses.

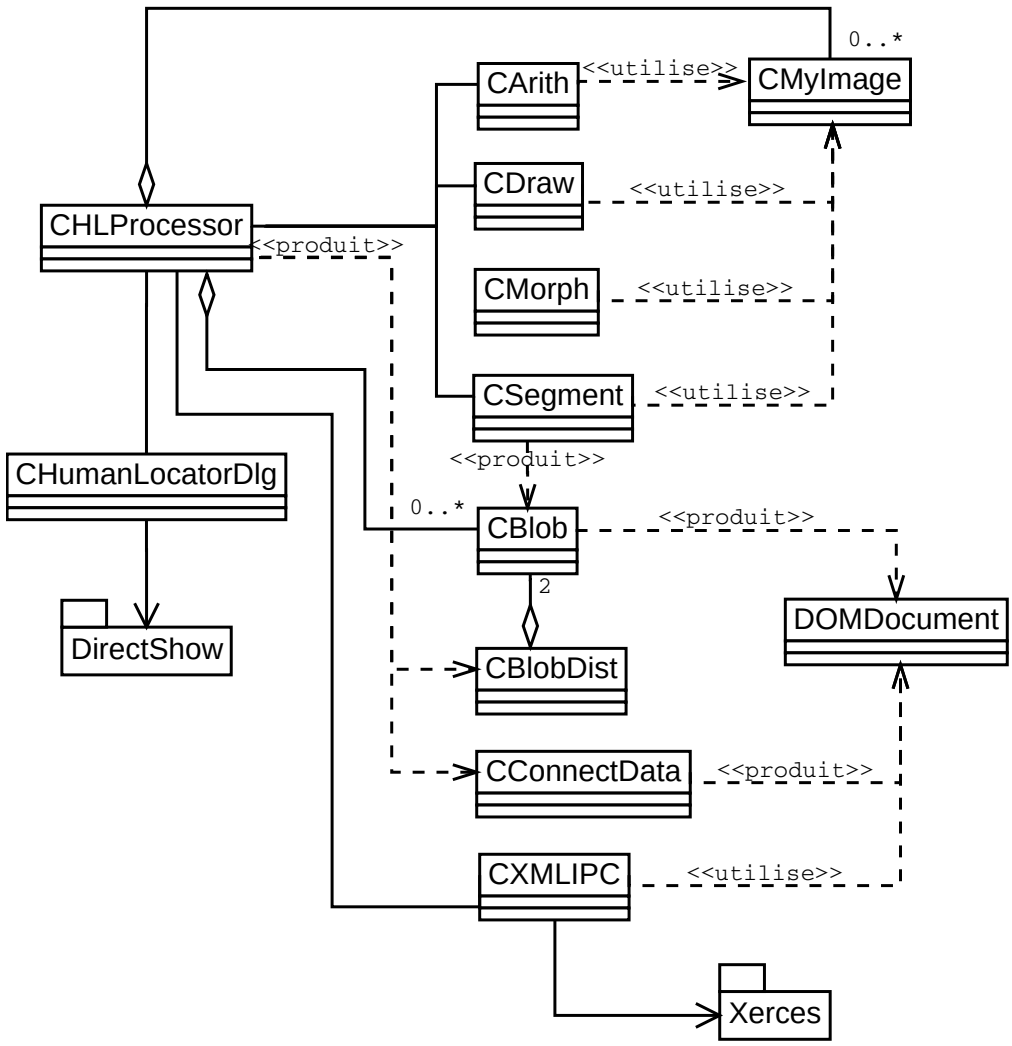


Figure 3.1 - Diagramme des classes de HumanLocator

Classe: CHumanLocatorDlg

Fichiers: HumanLocatorDlg.h, HumanLocatorDlg.cpp

Cette classe englobe toute la fonctionnalité ayant rapport à l'interface graphique et à l'accès à DirectShow pour la caméra. La partie MFC (Microsoft Foundation Classes) est mieux comprise par la description fonctionnelle de la section suivante et la compréhension de MFC elles-mêmes.

Il en est de même pour la partie DirectShow qui ne fait qu'accéder à la caméra, qui quoi que complexe, est basé sur les API de DirectShow et les exemples qui viennent avec la trousse de développement. Ces parties ne seront pas décrites en plus de détails.

Méthode: DisplayImage

Permet d'afficher dans une fenêtre une image contenue dans un objet CMyImage. La seule raison pour laquelle cette méthode est ici est qu'il n'y a pas vraiment d'autre endroit où la mettre.

Paramètres:

HWND hwnd (entrée) - handle de la fenêtre dans laquelle afficher l'image

CMyImage &image (entrée) - image à afficher

Valeur de retour: aucune

Classe: CHLProcessor

Fichiers: HLProcessor.h, HLProcessor.cpp

Cette classe contient la méthode principale de l'application: AnalyzeAndProcess(). CHumanLocatorDlg en crée une copie lorsqu'on commence une session avec le bouton « Start » et la détruit lorsqu'une session termine avec le button « Stop ».

Constructeur: CHLProcessor

Le constructeur doit recevoir en paramètre l'objet CHumanLocatorDlg pour avoir accès à la méthode DisplayImage(). De plus, il doit recevoir en paramètre la hauteur et la largeur des images qu'il aura à analyser, car il doit garder en mémoire l'image de l'arrière-plan.

Paramètres:

CHumanLocatorDlg *dlg (entrée) - objet avec la méthode DisplayImage()

long iWidth (entrée) - la largeur des images à analyser

long iHeight (entrée) - la hauteur des images à analyser

Méthode: AnalyzeAndProcess

Cette méthode fait tout le travail de traitement et d'analyse de l'image comme décrit dans la section *Configuration logiciel - Le traitement et l'analyse de l'image*. Elle fait donc référence aux méthodes de CArith, CDraw, CMorph, CSegment, CBlobDist et contient des CBlob. De plus, c'est elle qui s'occupe de communiquer avec l'Xtra à l'aide de le classe CXMLIPC et de Xerces C++.

Paramètres:

CMyImage &image (entrée) - image à analyser

Valeur de retour: aucune

Classe: CArith

Fichiers: Arith.h, Arith.cpp

Cette classe ne contient que des méthodes statiques qui s'occupent de faire des opérations arithmétiques sur des objets CMyImage.

Méthode: static sub

Fait l'opération de soustraction suivante sur une image à niveaux de gris:

`imageDst := imageDst – imageSrc` comme décrite dans la section plus bas *Soustraction de l'arrière-plan.*

Paramètres:

CMyImage &imageDst (entrée/sortie) - image source et de destination (niveaux de gris)

CMyImage &imageSrc (entrée) - image source (niveaux de gris)

Valeur de retour: aucune

Méthode: static binarize

Fait l'opération de binarisation sur une image à niveaux de gris comme décrite dans la section plus bas *Binarisation*.

Paramètres:

CMyImage &imageDst (entrée/sortie) - image source et de destination (niveaux de gris)

BYTE cThreshold (entrée) - la valeur de seuil

BYTE cBinValue (entrée) - la valeur à utiliser pour 1 (habituellement 128 ou 255)

Valeur de retour: aucune

Méthode: static subAbs

Fait l'opération de soustraction suivante sur une image à niveaux de gris:

`imageDst := | imageDst – imageSrc |` comme décrite dans la section plus bas *Soustraction de l'arrière-plan.*

Paramètres:

CMyImage &imageDst (entrée/sortie) - image source et de destination (niveaux de gris)

CMyImage &imageSrc (entrée) - image source (niveaux de gris)

Valeur de retour: aucune

Méthode: static subAbsAndBinarize

Cette fonction fait d'une pierre deux coups en faisant la soustraction en valeur absolue de deux images à niveaux de gris et la binarisation de l'image résultante, qui sera ensuite placée dans imageDst.

Paramètres:

CMyImage &imageDst (entrée/sortie) - image source et de destination (niveaux de gris)

CMyImage &imageSrc (entrée) - image source (niveaux de gris)

BYTE cThreshold (entrée) - la valeur de seuil

BYTE cBinValue (entrée) - la valeur à utiliser pour 1 (habituellement 128 ou 255)

Valeur de retour: aucune

Méthode: static binarizeColor

Fait l'opération de binarisation sur une image couleur comme décrite dans la section plus bas *Binarisation*.

Paramètres:

CMyImage &imageDst (sortie) - image de destination (niveaux de gris)

CMyImage &imageSrc (entrée) - image source (couleur)

BYTE cShaTresh (entrée) - la valeur de seuil pour les ombres

BYTE cThreshold (entrée) - la valeur de seuil

BYTE cBinValue (entrée) - la valeur à utiliser pour 1 (habituellement 128 ou 255)

Valeur de retour: aucune

Méthode: static subColor

Fait l'opération de soustraction suivante sur une image couleur:

`imageDst := imageDst – imageSrc` comme décrite dans la section plus bas

Soustraction de l'arrière-plan.

Paramètres:

CMyImage &imageDst (entrée/sortie) - image source et de destination (couleur)

CMyImage &imageSrc (entrée) - image source (couleur)

Valeur de retour: aucune

Méthode: static subAbsColor

Fait l'opération de soustraction suivante sur une image couleur:

`imageDst := | imageDst – imageSrc |` comme décrite dans la section plus bas

Soustraction de l'arrière-plan.

Paramètres:

CMyImage &imageDst (entrée/sortie) - image source et de destination (couleur)

CMyImage &imageSrc (entrée) - image source (couleur)

Valeur de retour: aucune

Méthode: static add

Fait l'opération d'addition suivante sur une image à niveaux de gris:

`imageDst := imageDst + imageSrc` . Il y a saturation à 255. Cette opération n'est actuellement pas utilisée dans HumanLocator.

Paramètres:

CMyImage &imageDst (entrée/sortie) - image source et de destination (niveaux de gris)

CMyImage &imageSrc (entrée) - image source (niveaux de gris)

Valeur de retour: aucune

Méthode: static RGBtoHSV

Converti un pixel couleur de l'espace RGB à l'espace HSV. Cette opération n'est actuellement pas utilisée dans HumanLocator.

Paramètres:

BYTE *h (sortie) - composante H

BYTE *s (sortie) - composante S

BYTE *v (sortie) - composante V

BYTE r (entrée) - composante R

BYTE g (entrée) - composante G

BYTE b (entrée) - composante B

Valeur de retour: aucune

Méthode: static HSVtoRGB

Converti un pixel couleur de l'espace HSV à l'espace RGB. Cette opération n'est actuellement pas utilisée dans HumanLocator.

Paramètres:

BYTE *r (sortie) - composante R

BYTE *g (sortie) - composante G

BYTE *b (sortie) - composante B

BYTE h (entrée) - composante H

BYTE s (entrée) - composante S

BYTE v (entrée) - composante V

Valeur de retour: aucune

Classe: CDraw

Fichiers: Draw.h, Draw.cpp

CDraw est une classe qui regroupe des méthodes statiques qui permettent de dessiner sur des objets CMyImage.

Méthode: static drawTriangle

Permet de dessiner un triangle qui se retrouvera limité par un rectangle de centre (x,y) et de taille (dx,dy). De plus, il sera dessiner avec une rotation si autre que 0 comme angle est passé en paramètre. À 0 degré, le triangle pointe vers le haut.

Paramètres:

CMyImage &Image (entrée/sortie) - image sur laquelle dessiner (niveaux de gris)

long x (entrée) - position du centre en x

long y (entrée) - position du centre en y

long dx (entrée) - position du centre en dx

long dy (entrée) - position du centre en dy

float rotation (entrée) - rotation du triangle en degrés

Valeur de retour: aucune

Méthode: static drawLine

Permet de dessiner une ligne qui part de (x1,y1) et qui va jusqu'à (x2, y2).

Paramètres:

CMyImage &Image (entrée/sortie) - image sur laquelle dessiner (niveaux de gris)

long x1 (entrée) - début de la ligne en x

long y1 (entrée) - début de la ligne en y

long x2 (entrée) - fin de la ligne en x

long y2 (entrée) - fin de la ligne en y

Valeur de retour: aucune

Méthode: static drawArrow

Permet de dessiner une flèche à l'aide d'une ligne et d'un triangle. Elle partira de (x1,y1) jusqu'à (x2, y2).

Paramètres:

CMyImage &Image (entrée/sortie) - image sur laquelle dessiner (niveaux de gris)

long x1 (entrée) - début de la flèche en x

long y1 (entrée) - début de la flèche en y

long x2 (entrée) - fin de la flèche en x

long y2 (entrée) - fin de la flèche en y

Valeur de retour: aucune

Méthode: static drawRectangle

Permet de dessiner un rectangle, à l'aide de lignes, qui aura comme quatre limites: left, top, right et bottom.

Paramètres:

CMyImage &image (entrée/sortie) - image sur laquelle dessiner (niveaux de gris)

long left (entrée) - limite gauche du rectangle

long top (entrée) - limite supérieure du rectangle

long right (entrée) - limite droite du rectangle

long bottom (entrée) - limite inférieure du rectangle

Valeur de retour: aucune

Classe: CMorph

Fichiers: Morph.h, Morph.cpp

Méthode: static dilate

Procède à une dilatation surplace en utilisant un algorithme ordinaire sur une image à niveaux de gris comme décrite dans la section *Érosion et dilatation.*

Paramètres:

CMyImage &image (entrée/sortie) -image sur laquelle faire une dilatation (niveaux de gris)

BYTE *pMask (entrée) - l'élément structurant avec lequel faire la dilatation

long imWidth (entrée) - la largeur de l'élément structurant

long imHeight (entrée) - la hauteur de l'élément structurant

BYTE cBinValue (entrée) - valeur à utiliser dans l'image de destination comme un 1 (habituellement 128 ou 256)

Valeur de retour: aucune

Méthode: static erode

Procède à une érosion surplace en utilisant un algorithme ordinaire sur une image à niveaux de gris comme décrite dans la section *Érosion et dilatation.*

Paramètres:

CMyImage &image (entrée/sortie) - image sur laquelle faire une érosion (niveaux de gris)

BYTE *pMask (entrée) - l'élément structurant avec lequel faire l'érosion

long imWidth (entrée) - la largeur de l'élément structurant

long imHeight (entrée) - la hauteur de l'élément structurant

BYTE cBinValue (entrée) - valeur à utiliser dans l'image de destination comme un 1 (habituellement 128 ou 256)

Valeur de retour: aucune

Méthode: static dilateFast

Procède à une dilatation rapide comme décrite dans l'algorithme de la section *Érosion et dilatation.* La largeur et la hauteur de l'élément structurant doit être <= 15.

Paramètres:

CMyImage &image (entrée/sortie) -image sur laquelle faire une dilatation (niveaux de gris)

BYTE *pMask (entrée) - l'élément structurant avec lequel faire la dilatation

long imWidth (entrée) - la largeur de l'élément structurant (<= 15)

long imHeight (entrée) - la hauteur de l'élément structurant (<= 15)

BYTE cBinValue (entrée) - valeur à utiliser dans l'image de destination comme un 1 (habituellement 128 ou 256)

Valeur de retour: aucune

Méthode: static erodeFast

Procède à une érosion rapide comme décrite dans l'algorithme de la section *Érosion et dilatation.* La largeur et la hauteur de l'élément structurant doit être <= 15.

Paramètres:

CMyImage &image (entrée/sortie) - image sur laquelle faire une érosion (niveaux de gris)

BYTE *pMask (entrée) - l'élément structurant avec lequel faire l'érosion

long imWidth (entrée) - la largeur de l'élément structurant (<= 15)

long imHeight (entrée) - la hauteur de l'élément structurant (<= 15)

BYTE cBinValue (entrée) - valeur à utiliser dans l'image de destination comme un 1 (habituellement 128 ou 256)

Valeur de retour: aucune

Classe: CSegment

Fichiers: Segment.h, Segment.cpp

Cette classe ne contient qu'une seule méthode statique: findBlobs() qui permet de trouver des blobs dans une image binaire.

Méthode: static findBlobs

Permet de trouver tous les blobs d'une image. Les étiquettes ainsi trouvées se retrouvent en sortie dans une « image de pointeurs » qui contient comme pixels des pointeurs vers un objet CBlob. Ces mêmes objets se retrouvent aussi en sortie dans le vecteur pBlobs. L'algorithme est décrit dans la section *Segmentation en blobs.*

Paramètres:

CBlob **pBlobImage (sortie) - une image de pointeurs vers des objets CBlob

CMyImage &image (entrée) - l'image à analyser

vector<CBlob *> *pBlobs (sortie) - un tableau contenant les nouveaux CBlob

Note: l'appelant est maintenant le propriétaire des objets CBlob crées

Valeur de retour: aucune

Classe: CBlob

Fichiers: Blob.h, Blob.cpp

Cette classe représente en soit une étiquette ou un blob comme décrit dans la section *Segmentation en blobs.* Elle est utilisée par CSegment:findBlobs() pour trouver toutes les

étiquettes d'une image. Les étiquettes équivalentes sont ensuite fusionnées. Ce qu'il reste représente les blobs de l'image. Un blob contient les attributs suivants:

long iRight (accès lecture) - la limite droite du blob

long iLeft (accès lecture) - la limite gauche du blob

long iBottom (accès lecture) - la limite inférieure du blob

long iTop (accès lecture) - la limite supérieure du blob

CBlob *pMotherBlob (privé) - le blob mère de ce blob

CBlob *pPrevBlob (accès lecture/écriture) -le blob de l'image précédente qui lui est associé

long iCustom (public) - un attribut qui peut-être utilisé comme on veut

Constructeur: CBlob

Initialise iRight, iLeft, iBottom et iTop à 0. Initialise pMotherBlob et pPrevBlob à NULL.

Paramètres: aucun

Constructeur: CBlob

Initialise iRight, iLeft à x, et iBottom et iTop à y. Initialise pMotherBlob et pPrevBlob à NULL.

Paramètres:

long x (entrée)

long y (entrée)

Méthode: addPixel

Repousse les limites iRight, iLeft, iTop et iBottom pour englober le nouveau pixel (x,y).

Paramètres:

long x (entrée) - x du nouveau pixel à rajouter

long y (entrée) - y du nouveau pixel à rajouter

Valeur de retour: aucune

Méthode: setMother

Rajoute l'étiquette reçue en paramètre dans la chaîne d'étiquettes équivalentes. Voir la section *Segmentation en blobs* pour plus d'information.

Paramètre:

CBlob *pBlob (entrée) - étiquette mère à rajouter à la chaîne d'étiquettes

Valeur de retour: aucune

Méthode: getMother

Retourne l'étiquette mère qui se trouve au bout de la chaîne. Voir la section *Segmentation en blob* pour plus d'information.

Paramètre: aucun

Valeur de retour: un CBlob * qui est la mère absolue de ce blob

Méthode: merge

Repousse les limites iRight, iLeft, iTop et iBottom pour englober pBlob.

Paramètre:

CBlob *pBlob (entrée) - blob avec qui fusionner

Valeur de retour: aucune

Méthode: getMiddle

Paramètre: aucun

Valeur de retour: un CMyPoint ayant comme coordonné ((iRight + iLeft)/2, (iBottom + iTop)/2)

Méthode: getDistance

Trouve la distance du milieu de pBlob jusqu'ici. Utilisé par CBlobDist pour calculer les distances entre blobs.

Paramètre:

CBlob * pBlob (entrée)

Valeur de retour: this->getMiddle().getDistance(pBlob->getMiddle())

Méthode: fillDOMDocument

Sauvegarde l'état de cet objet dans doc. Utile avec CXMLIPC pour transférer les données du blob à l'Xtra.

Paramètre:

DOMDocument *doc (sortie) - le document DOM

Valeur de retour: aucune

Méthode: loadDOMDocument

Récupère l'état de cet objet avec l'information contenue dans doc. Utile avec CXMLIPC pour transférer les données du blob à l'Xtra.

Paramètre:

DOMDocument *doc (entrée) - le document DOM

Valeur de retour: aucune

Classe: CBlobDist

Fichiers: BlobDist.h, BlobDist.cpp

Cette classe sert lorsque nous sommes arrivés à vouloir calculer les distances entre un blob courant et un blob de l'image précédente comme décrit dans la section *Analyse du mouvement* plus bas. Elle contient deux attributs:

CBlob *pBlobs[2] (accès lecture/écriture) - deux blobs quelconques

Note: setBlobs() recalcule la valeur de iDist avec CBlob::GetDistance()

long iDist (accès lecture) - la distance entre eux

Méthode: operator<

Méthode utilisée par l'algorithme de tri de la STL. Voir la section *Analyse du mouvement* pour comprendre l'utilité d'un tri.

Paramètre:

CBlob &blobDist (entrée) - le CBlobDist avec qui comparer

Valeur de retour: true si iDist < blobDist.iDist, sinon false

Classe: CConnectData

Fichiers: ConnecData.h, ConnectData.cpp

Cette classe ne comporte que deux attributs:

long iWidth (public) - la largeur de la résolution des images

long iHeight (public) - la hauteur de la résolution des images

Et deux méthodes:

void fillDOMDocument(DOMDocument *doc)

Sauvegarde l'état de cet objet dans doc.

void loadDOMDocument(DOMDocument *doc)

Récupère l'état de cet objet avec l'information contenue dans doc.

Cet objet est utilisé lors de la connexion d'un Xtra à HumanLocator. Il est envoyé par HumanLocator tout de suite après l'établissement d'une connexion à un client. Du côté du serveur, il est utilisé lors de la création avec CXMLIPC::createServerPipe() qui le requiert.

Classe: CXMLIPC

Fichiers: XMLIPC.h, XMLIPC.cpp

Cette classe fait le lien entre une pipe nommée de Windows NT et Xerces C++. Les données d'un objet DOMDocument sont envoyées sur la pipe, puis reçues et interprétées aussi à l'aide d'un objet DOMDocument.

Constructeur: CXMLIPC

Initialise tous les attributs privés assurant le bon fonctionnement de la pipe de Windows NT et initialise Xerces C++.

Paramètre: aucun

Destructeur: ~CXMLIPC()

S'assure que la pipe est fermée et désinitialise Xerces C++.

Méthode: createServerPipe

Ouvre une pipe système nommée « pName » et enverra connectData comme premier message à chaque client qui se connectera à la pipe. Cette fonction est utilisée par le serveur.

Paramètre:

char *pName (entrée) - le nom de la pipe à crée (doit commencer par « \\.\pipe\ »)

CConnectData *connectData (entrée) - les données à envoyer à chaque connexion de client

Valeur de retour: true s'il n'y a pas eu de problème, sinon false

Méthode: openPipe

Cette fonction est utilisée par le client.

Paramètre:

char *pName (entrée) - le nom de la pipe sur laquelle se connecter

(doit commencer par « \\.\pipe\ »)

Valeur de retour: true s'il n'y a pas eu de problème, sinon false

Méthode: closePipe

Ferme autant la pipe du serveur que celle du client.

Valeur de retour: true s'il n'y a pas eu de problème, sinon false

Méthode: sendToPipe

Envoie sur la pipe les données contenues dans doc.

Paramètre:

DOMDocument *doc (entrée) - l'objet DOMDocument

Valeur de retour: long

< 0 - erreur

0 - rien à envoyer (erreur ?)

autre - la quantité d'octets envoyés

Méthode: recvFromPipe

Reçoit les données de la pipe et les stocke dans DOMDocument.

Paramètre:

DOMDocument **doc (sortie) - l'objet DOMDocument

Note: L'appelant ne devient ***pas*** propriétaire de l'objet crée après l'appel.

Cependant, il est valide tant que le CXMLIPC correspondant existe.

bool bBlock (entrée) - true: recvFromPipe va bloquer tant qu'il n'y a rien sur la pipe

false: recvFromPipe va retourner avec 0 s'il n'y a rien à lire

Valeur de retour: long

< 0 - erreur

0 - plus rien à lire (erreur ?)

autre - la quantité d'octets lus

Méthode: createDOMDocument

Retourne un objet DOMDocument vide avec un élément racine portant la marque de XMLIPC et qui peut être rempli comme bon semble avant de l'utiliser avec sendToPipe().

Paramètre: aucun

Valeur de retour: un DOMDocument tout neuf

Note: L'appelant devient propriétaire du DOMDocument.

DOMDocument::release() doit être appelé pour qu'il soit détruit.

Classe: CMyImage

Fichiers: MyImage.h, MyImage.cpp

La classe CMyImage est une classe qui représente une image dans une matrice bidimensionnelle implantée à l'aide des attributs suivants:

BYTE *pImage (accès lecture) - le tableau contenant les pixels

long iWidth, iHeight (accès lecture) - la largeur et la hauteur de l'image

bool bOurBuffer (accès lecture) - indique si le buffer pImage nous appartient

enum MYIMAGE_TYPE type (accès lecture) - indique le type de l'image

(MYIMAGE_GRAY8 ou MYIMAGE_RGB24)

Note: le pixel (0,0) correspond au coin gauche bas de l'image.

Constructeur: CMyImage

Initialise l'image à un type invalide.

Paramètre: aucun

Constructeur: CMyImage

Initialise l'image au type et à la taille désirés.

Paramètres:

long iWidth (entrée) - largeur désirée

long iHeight (entrée) - hauteur désirée

enum MYIMAGE_TYPE type (entrée) - type désiré

(MYIMAGE_GRAY8 ou MYIMAGE_RGB24)

Constructeur: CMyImage

Initialise l'image au type et à la taille désirés à l'aide d'un buffer déjà existant. Le buffer n'appartiendra ***pas*** à CMyImage.

Paramètres:

BYTE *pImage - buffer à utiliser

long iWidth (entrée) - largeur désirée

long iHeight (entrée) - hauteur désirée

enum MYIMAGE_TYPE type (entrée) - type désiré

(MYIMAGE_GRAY8 ou MYIMAGE_RGB24)

Destructeur: ~CMyImage

Détruit pImage s'il nous appartient.

Méthode: getPixel

Retourne le pixel (x,y) d'une image MYIMAGE_GRAY8.

Paramètres:

long x (entrée)

long y (entrée)

Valeur de retour: BYTE *

Méthode: getPixel

Retourne le $i^{ième}$ pixel d'une image MYIMAGE_GRAY8.

Paramètres:

long i (entrée)

Valeur de retour: BYTE *

Méthode: getPixelRGB

Retourne le pixel (x,y) d'une image MYIMAGE_RGB24. La raison pour laquelle ces fonctions sont en double est pour une question d'optimisation. Le code pour accéder au buffer est beaucoup plus optimal lorsqu'on peut utiliser une constante pour y accéder. Il faudrait probablement convertir CMyImage en une classe template pour mieux implanter cette fonctionnalité.

Paramètres:

long x (entrée)

long y (entrée)

Valeur de retour: BYTE *

valeur[0] = bleu

valeur[1] = vert

valeur[2] = rouge

Méthode: getPixelRGB

Retourne le $i^{ième}$ pixel d'une image MYIMAGE_RGB24..

Paramètres:

long i (entrée)

Valeur de retour: BYTE *

valeur[0] = bleu

valeur[1] = vert

valeur[2] = rouge

Méthode: operator=

Opérateur d'affectation. Le pBuffer est complètement copié.

Paramètre:

CMyImage &image (entrée) - l'image à copier

Valeur de retour: CMyImage &

l'objet courant (this)

Méthode: setImage

Initialise l'image au type et à la taille désirés.

Paramètres:

long iWidth (entrée) - largeur désirée

long iHeight (entrée) - hauteur désirée

enum MYIMAGE_TYPE type (entrée) - type désiré

(MYIMAGE_GRAY8 ou MYIMAGE_RGB24)

Valeur de retour: aucune

Méthode: setImage

Initialise l'image au type et à la taille désirés à l'aide d'un buffer déjà existant. Le buffer n'appartiendra ***pas*** à CMyImage.

Paramètres:

BYTE *pImage - buffer à utiliser

long iWidth (entrée) - largeur désirée

long iHeight (entrée) - hauteur désirée

enum MYIMAGE_TYPE type (entrée) - type désiré

(MYIMAGE_GRAY8 ou MYIMAGE_RGB24)

Valeur de retour: aucune

Classe: CMyPoint

Fichiers: MyPoint.h, MyPoint.cpp

Cette classe représente un point (x,y) à l'aide de deux attributs:

long x,y (public)

Méthode: getDistance

Paramètre:

CMyPoint p2 (entrée) - un deuxième point

Valeur de retour: long

distance entre le point courant et p2

Méthode: operator*

Multiplie le point avec une matrice. Cette méthode n'est pas actuellement utilisée dans HumanLocator.

Paramètre:

CMatrix<float> &matrix (entrée) - une matrice de float

Valeur de retour: CMyPoint

le résultat de l'opération

Classe: DOMDocument

Cette classe fait partie de la bibliothèque Xerces C++. S.V.P. veuillez faire référence à la documentation de celle-ci pour en savoir plus.

3.2.2 Interface graphique

HumanLocator possède une interface graphique qui permet à l'usager de visualiser les images à différentes étapes de traitement et d'analyse, de changer la caméra et ses options, de même que de changer diverses valeurs qui sont utilisées par les algorithmes de traitement et d'analyse de l'image. Une capture d'écran de l'interface usager se trouve à la figure 3.2.

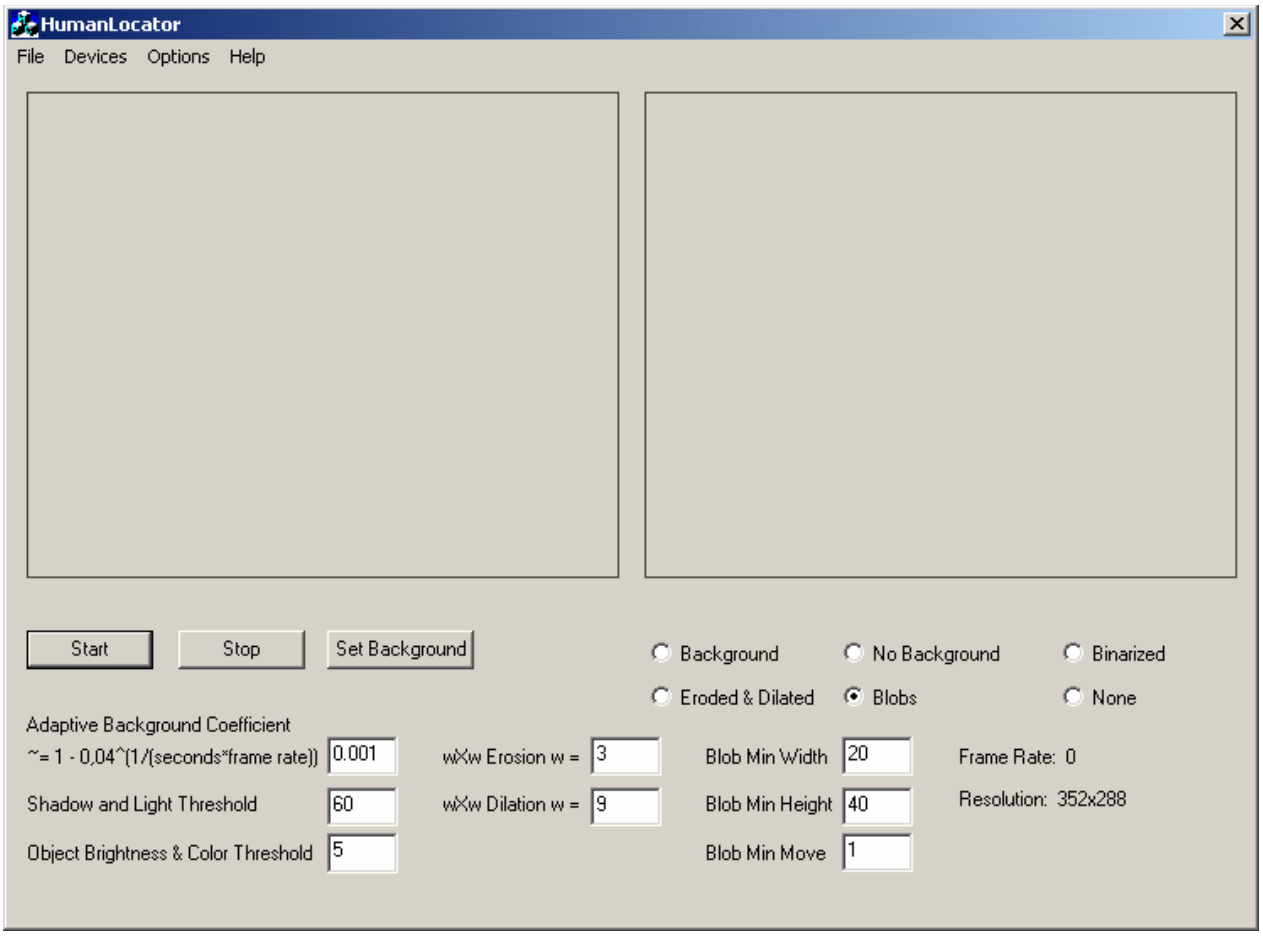


Figure 3.2 - Interface graphique de HumanLocator

Pour chacun des éléments de l'interface la table 3.2 donne une description fonctionnelle de celui-ci.

Éléments de l'interface	***Description fonctionnelle***
Menu	
File	Possède l'option « Exit » pour fermer le programme.
Devices	Liste les caméras disponibles sur la machine et permet d'en choisir une.
Options	Possède des options pour régler la caméra. Les options varient selon les caméras.
Help	Ne possède qu'une boîte de dialogue « À propos de »

Éléments de l'interface	*Description fonctionnelle*
Espace	
de gauche	Si l'option de « Preview » du menu option est activée, affiche ce qui est en provenance de la caméra.
de droite	Permet de voir l'image après une étape de traitement que l'on choisie grâce aux cases d'option en dessous de celle-ci.
Bouton	
Start	Démarre une session de traitement et d'analyse des images en provenance de la caméra choisie.
Stop	Arrête cette session.
Set Background	Fixe l'image de l'arrière-plan à l'image courante (voir la section plus bas *Soustraction de l'arrière-plan*)
Case d'option	Affiche dans l'espace de droite:
Background	• l'image de l'arrière-plan
No Background	• l'image courante soustraite de l'arrière-plan
Binarized	• l'image binarisée
Eroded & Dilated	• l'image une fois érodée et dilatée
Blobs	• l'image avec des rectangles entourant les blobs trouvés ainsi que des flèches qui suivent le mouvement des blobs
None	• rien
Zone d'entrée	
Adpative Background Coefficient	• le coefficient de l'arrière-plan adaptatif. Calculez la valeur à mettre avec cette équation: $1 - 0{,}04^{\frac{1}{(secondes * vitesse\ de\ défilement\ des\ images)}}$. (voir la section plus bas *Soustraction de l'arrière-plan*)
Shadow and Light Threshold	• la valeur seuil pour les ombres (voir la section *Binarisation*)

Éléments de l'interface	*Description fonctionnelle*
Object Brightness & Color Threshold	• la valeur seuil de la binarisation (voir la section *Binarisation*)
wXw Erosion	• la grandeur de l'élément structurant de l'érosion (voir la section *Érosion et dilatation*)
wXw Dilation	• la grandeur de l'élément structurant de la dilatation (voir la section *Érosion et dilatation*)
Blob Min Width	• la largeur minimale d'un blob
Blob Min Height	• la hauteur minimale d'un blob
Blob Min Move	• le mouvement minimale d'un blob i.e: Le mouvement ne sera pas rapporté tant qu'il n'atteint pas la valeur minimale.
Zone d'information	
Frame Rate	• la vitesse de défilement des images.
Resolution	• la résolution (largeur x hauteur) des images qui proviennent de la caméra

Table 3.2 - Description des éléments de l'interface

3.2.3 Soustraction de l'arrière-plan

La première étape qu'une nouvelle image en provenance de la caméra subit est la soustraction de l'arrière-plan. L'implantation de cette fonctionnalité est faite dans la méthode CHLProcessor::AnalyzeAndProcess(). La valeur de la soustraction est en fait la valeur absolue de la différence entre chaque composante (rouge, verte et bleue) de chaque pixel, comme illustré par l'algorithme de la figure 3.3. La nouvelle image ainsi obtenue ne contient que les éléments différents de l'arrière-plan.

```
pour tout pixel à i dans image
   image[i] := valeur_absolue(image[i] – arrière_plan[i])
```

Figure 3.3 - Algorithme pour la soustraction d'images

L'arrière-plan est en fait une image précédente qui est conservée en mémoire, agissant un peu comme la « conscience » de l'environnement de la machine. L'opérateur doit bien sûr être celui qui doit décider lorsque qu'une image est bonne candidate pour l'arrière-plan et appuyer sur le bouton « Set Background » de l'interface usager pour transférer l'image courante à l'arrière-plan. Cette méthode s'apparente à la détection du mouvement, mais je voulais essayer autre chose que seulement la détection du mouvement, car une fois qu'une personne ou un objet arrête de bouger, il n'est plus détectable par le détecteur de mouvement. Les problèmes reliés à l'utilisation d'une image arrière-plan sont toutefois tout aussi importants. Dès qu'un changement de l'arrière-plan se produit, on doit fixer à nouveau l'image reliée à l'arrière-plan. Cela peut facilement survenir lorsqu'il y a un changement de la luminosité par exemple, car les caméras ne s'adaptent pas aussi bien que l'oeil humain aux changements de lumière. Pour compenser en partie pour un changement de lumière graduel, j'ai implanté un simple filtre IIR qui permet à l'arrière-plan de s'adapter à de petits changements d'emplacement physique d'objets de la scène ou à de petits changements d'intensité lumineuse [21].

Adaptation de l'arrière-plan

Pour permettre à l'arrière-plan de s'adapter à de petits changements, on utilise l'algorithme décrit dans la figure 3.4. L'implantation de cette fonctionnalité est aussi faite dans la méthode CHLProcessor::AnalyzeAndProcess().

```
pour tout pixel à i dans image
  si image[i] n'appartient pas à un blob
    arrière_plan[i]:=coefficient*image[i]+(1-coefficient)*arrière_plan[i]
```

Figure 3.4 - Algorithme pour adapter l'arrière-plan à l'aide d'un simple filtre IIR

Le coefficient est une variable qui est spécifiée grâce à la zone d'entrée « Adapative Background Coefficient » dans l'interface usager. L'arrière-plan tend donc graduellement vers l'image courante, si celle-ci est statique. De plus, si le pixel appartient à un blob, c'est à dire à quelque chose que l'on croit qui ne fait pas partie de l'arrière-plan, la valeur de ce pixel dans l'arrière-plan n'est pas ajusté. Cette procédure peut ainsi compenser pour les changements graduels d'intensité lumineuse et pour l'ajout, le retrait et le déplacement de petits objets.

Cela peut bien sûr créer des problèmes si justement un trop grand changement d'intensité lumineuse par exemple provoque la trouvaille de blobs qui ne sont pas nécessairement des objets. Le système ne fonctionne alors plus du tout. Il faut fixer à nouveau l'arrière-plan manuellement comme c'est le cas lors du déplacement de la caméra ou lors de l'ajout, le retrait ou le déplacement de gros objets appartenant à l'arrière-plan.

Ensuite, quelle valeur donnée au coefficient? Pour cela, il faut analyser la fonction de transfert du filtre à l'aide de sa transformée en Z [12]. Avec y(n) l'arrière-plan, x(n) l'image courante et α le coefficient nous avons:

$$y(n) = \alpha\, x(n) + (1-\alpha)\, y(n-1) \qquad \textbf{(1)}$$
$$n \in \mathbb{N}$$

Nous pouvons alors trouver la fonction de transfert H(z):

$$y(n)-(1-\alpha)y(n-1)=\alpha\,x(n)$$
$$Y(z)-(1-\alpha)z^{-1}Y(z)=\alpha\,X(z)$$
$$Y(z)(1-(1-\alpha)z^{-1})=\alpha\,X(z)$$
$$H(z)=\frac{Y(z)}{X(z)}=\frac{\alpha}{1-(1-\alpha)z^{-1}} \qquad \textbf{(2)}$$

Nous pouvons ainsi trouver la réponse du système à une impulsion:

$$x(0)=1, x(n+1)=0$$
$$X(z)=1$$
$$Y(z)=H(z)X(z)=H(z)$$
$$y(n)=h(n)=\alpha(1-\alpha)^{n} \qquad (\textbf{3})$$

Toutefois, ce que nous cherchons est le temps que le système va prendre pour partir par exemple de 0 (donc y(-1) = 0) et d'aller jusqu'à 1 si x(n) = 1. Nous avons ainsi:

$$x(n)=1$$
$$X(z)=\frac{1}{1-z^{-1}}$$
$$Y(z)=H(z)X(z)=\frac{\alpha}{1-(1-\alpha)z^{-1}}\times\frac{1}{1-z^{-1}}$$
$$=\frac{A}{1-(1-\alpha)z^{-1}}+\frac{B}{1-z^{-1}}$$

Avec les fractions partielles, on trouve

$$A=\frac{-(1-\alpha)}{\alpha}$$
$$B=\frac{1}{\alpha}$$
$$Y(z)=\left(\frac{-(1-\alpha)}{\alpha(1-(1-\alpha)z^{-1})}+\frac{1}{\alpha(1-z^{-1})}\right)\alpha$$

$$y(n)=(\alpha-1)(1-\alpha)^{n}+1 \qquad \textbf{(5)}$$

Nous avons ainsi avec l'équation (5) la réponse du filtre. Comme il s'agit d'un filtre IIR, il ne faut pas s'attendre à obtenir un changement total de y(n) vers 1 dans le temps désiré. Dans notre cas,

nous pouvons établir qu'une valeur de 245/256 = 0,96 est suffisamment proche de 1. En isolant α dans l'équation (5) et en utilisant un y(n) = 0,96, on obtient la formule suivante:

$$\alpha = 1 - e^{\frac{\ln(1-y(n))}{1+n}}$$

$$\alpha = 1 - e^{\frac{-3,22}{1+n}} \qquad \textbf{(6)}$$

$$\text{si n} >> 1 \Rightarrow \alpha = 1 - e^{\frac{-3,22}{n}}$$

$$\alpha = 1 - 0,04^{\frac{1}{n}} \qquad \textbf{(7)}$$

Comme le n utilisé dans l'équation est toujours beaucoup plus grand que 1, nous avons utilisé l'équation (7). Cette équation se retrouve dans l'interface graphique en dessous de la zone d'entrée « Adpative Background Coefficient ». Cela permet à l'usager de trouver un coefficient utile à ses besoins. Il faut habituellement que l'arrière-plan ne s'adapte pas trop vite, car il deviendrait alors facile d'acquérir trop rapidement des objets qui ne devraient pas faire partie de l'arrière-plan. Comme exemple d'utilisation de la formule, si l'on veut que l'arrière-plan s'adapte en 60 secondes et que la vitesse de défilement des images est de 10 images par secondes, le coefficient sera de:

$$\alpha = 1 - 0,04^{\frac{1}{(60\,s \times 10\,images/\,s)}} = 0,0054$$

En dernier lieu, pour ce qui est de l'implantation dans le code de algorithme de la figure 3.4, il faut mentionner que comme les images sont conservées dans des tableaux d'octets et que comme le coefficient est habituellement de l'ordre de 0,001, il est évident qu'il faut conserver en mémoire l'arrière-plan à l'aide d'un tableau de nombres en virgule flottante pour permettre à un changement de se faire sentir.

En somme, la soustraction de l'arrière-plan est une méthode efficace pour faire abstraction des parties de l'image que l'on identifie comme étant l'arrière-plan. Elle possède toutefois un gros problème: que se passe-t-il quand l'arrière-plan change? J'ai développé une méthode pour compenser pour les petits changements, mais cela ne règle pas complètement le problème.

3.2.4 Binarisation

La prochaine étape du traitement consiste en la binarisation. L'algorithme de base de la binarisation est décrite dans la figure 3.5. Le code relié à la binarisation se trouve dans la classe CArith.

```
pour tout pixel à i dans image
   si image[i] < valeur_seuil
      image_binaire[i] := 0
   sinon
      image_binaire[i] := 1
```

Figure 3.5 - Algorithme pour la binarisation d'une image à niveaux de gris

La valeur seuil est une variable qui est spécifiée grâce à la zone d'entrée « Oject Brightness and Color Threshold » dans l'interface usager de HumanLocator. Elle se situe entre 0 et 255, car les images sont des tableaux d'octets. Cela fonctionne bien dans le cas d'une image originale à niveaux de gris, mais pour une image couleur, il faut soit la convertir en niveaux de gris au préalable, ou rendre l'information sur les couleurs utile. Si la caméra produit beaucoup de bruit dans les couleurs, la conversion en niveaux de gris permet d'éviter ce bruit. Toutefois, un autre problème survient, les ombres. Ce que nous voulons trouver dans l'image ce sont les objets et non leurs ombres. Les ombres ont toutefois une caractéristique intéressante, elles restent habituellement environ de la même couleur que les objets sur lesquels elles se posent. Il ne reste

qu'à discriminer plus sévèrement les pixels, de l'image soustraite de l'arrière-plan, qui conservent les mêmes proportions de couleurs par rapport à leurs luminances. On retrouve dans une image trois composantes, la composante rouge (R), verte (G) et bleue (B) et il est possible de dériver la luminance (s) à partir de celles-ci. Pour y parvenir, j'ai utilisé une méthode qui consiste premièrement à séparer l'information sur la luminance de celle sur les couleurs [5]. Cette méthode utilise les équations suivantes à une constante près pour séparer les informations:

$$s = \frac{R+G+B}{3}$$
$$r = \frac{R}{s}$$
$$g = \frac{G}{s}$$
$$b = \frac{B}{s}$$

Il s'agit ensuite de traiter r, g et b à la manière de R, G et B. Ainsi il faut faire la différence entre deux pixels dans l'espace r, g et b pour savoir si une des composantes à changer. Il faut alors refaire l'étape de la soustraction de l'image de l'arrière-plan de l'image courante dans ce nouvel espace de couleurs. Toutefois, comme ces nouvelles composantes ne tiennent pas compte de la luminance, si nous ne tenions compte que de r, g et b pour la binarisation il est possible qu'un objet assez noir (R=1, G=1, B=1, s=1 et ainsi r=1, g=1, b=1) se présente sur un fond blanc (R=255, G=255, B=255, s=255 et dans ce cas-ci aussi r=1, g=1, b=1) et ne soit pas détecté par cette méthode. Pour compenser pour ces extrêmes, nous allons utiliser l'espace de couleurs r,g,b seulement lorsque la différence de luminance se trouve en deçà d'une valeur seuil. Lorsque que la différence de luminance est trop faible, il peut s'agir d'une ombre ou d'un faible éclairage et il ne faut alors pas utiliser l'information sur la luminance. Si toutefois, la différence est très grande,

on suppose qu'il ne s'agit pas d'une ombre ni d'un éclairage. Cette valeur de seuil est définie par la zone d'entrée « Shadow and Light Threshold » de l'interface graphique.

À la suite de la deuxième étape de soustraction nécessaire, nous avons pour chaque pixel:

$$R_d = R_{arriere\ plan} - R_{image}$$
$$G_d = G_{arriere\ plan} - G_{image}$$
$$B_d = B_{arriere\ plan} - B_{image}$$
$$s_d = s_{arriere\ plan} - s_{image}$$
$$r_d = r_{arriere\ plan} - r_{image}$$
$$g_d = g_{arriere\ plan} - g_{image}$$
$$b_d = b_{arriere\ plan} - b_{image}$$

Il est alors possible de développer un nouvel algorithme, comme décrit dans la figure 3.6, grâce à ces nouvelles valeurs. Une valeur seuil pour l'ombre d'environ 60 et une valeur seuil d'environ 5 donnent habituellement de bons résultats, mais cela varie beaucoup de caméra en caméra.

```
pour tout pixel à i dans image
   image_binaire[i] := 1
   si sd[i] < valeur_seuil_ombre
      si (rd[i] + gb[i] + bd[i]) < valeur_seuil
         image_binaire[i] := 0
```

Figure 3.6 - Algorithme pour la binarisation d'une image couleur

L'utilisation de b_d est discutable, car comme c'est une couleur à laquelle l'oeil humain est très peu sensible, la plupart des caméras négligent cette partie du spectre et on y retrouve souvent beaucoup de bruit. Divers tests avec les caméras qui seront à notre disposition devront être faits.

Enfin, la binarisation permet d'obtenir une image binaire contenant seulement l'information qui nous intéresse. L'image soustraire de l'arrière-plan possède déjà en théorie seulement les parties qui nous intéresse, excluant le bruit, les zones éclairées et les zones d'ombrage. Pour discriminer

ces zones, il importe d'utiliser un espace de couleurs qui ne tienne pas compte des changements de luminance pour avoir une chance de les détecter. J'utilise pour cela les composantes r,g et b. Il serait aussi possible d'utiliser les composantes chromatiques du YUV, YIQ ou du HSV. Il faudrait les étudier dans le contexte actuel de l'application pour voir avec quelle efficacité il serait possible des les utiliser par rapport aux composantes r, g et b.

3.2.5 Érosion et dilatation

Une érosion suivie d'une dilatation ou l'inverse servent à deux choses. Premièrement à enlever du bruit de l'image à l'aide d'une érosion suivie d'une dilatation de même taille. Deuxième à boucher les trous avec une dilatation suivie d'une érosion de même taille [13]. Il y a toutefois une certaine perte dans la résolution de l'image si la dilatation est trop importante. Une dilatation et une érosion se font habituellement sur une image binaire. Dans le cas de HumanLocator, c'est ce qui est fait et son implantation se trouve dans la classe CMorph. Une fois la binarisation de l'image complétée, une érosion de taille 3 par défaut est faite pour enlever le bruit, et une dilatation de 9 est faite pour compenser pour l'érosion et pour boucher quelques trous. Les paramètres de l'érosion et de la dilatation peuvent être changer à l'aide des zones d'entrée « wXw Erosion » et « wXw Dilation » de l'interface usager. Il deviendra peut-être nécessaire de faire subir à l'image une seconde érosion après la dilatation si celle-ci est trop importante, mais cela ne fait pas partie de l'implantation actuel.

Une dilatation est en gros une convolution d'un élément structurant B sur un élément structuré (l'image) A. L'opération est notée $A \oplus B$, ce qui veut dire que l'on exécute un OU logique sur

l'ensemble des pixels de B et les pixels correspondants de A, pour tous les pixels de A. Dans la figure 3.7, on voit une illustration du processus. L'image A a tendance à devenir plus épaisse.

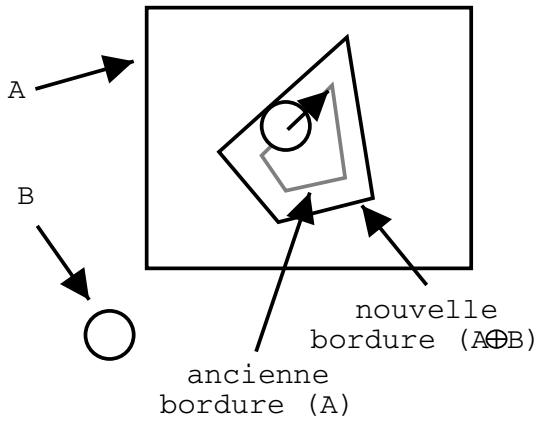


Figure 3.7 - Illustration d'une dilatation

Une érosion est exactement le contraire. Au lieu de faire un OU logique, il s'agit maintenant d'un ET logique, comme illustré sur la figure 3.8. Une érosion est notée A⊖B et a cette fois-ci tendance à amincir l'image A. L'élément structurant peut avoir n'importe quelle forme, un cercle, un carré, etc. Dans le cas de HumanLocator, un carré est utilisé. C'est ce qui a de plus facile à manipuler et il est symétrique. Nous n'avons pas besoin d'autres propriétés pour se débarrasser du bruit et pour boucher les trous.

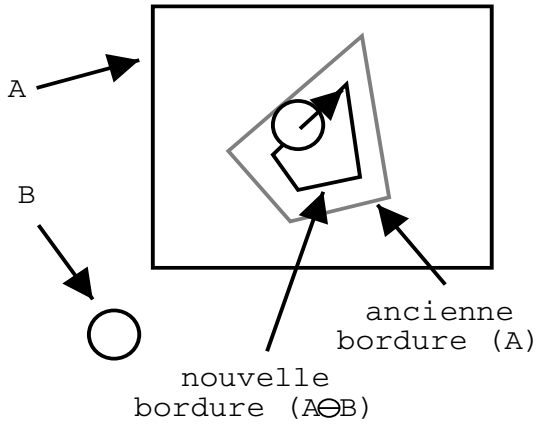


Figure 3.8 - Illustration d'une érosion

Pour donner un aperçu du type de résultat que ce genre d'opération peut donner, on peut observer dans la figure 3.9 le résultat après chaque étape de l'opération érosion/dilatation. La première image est l'image originale, la deuxième a subi une érosion carrée de 3x3, et finalement la troisième une dilatation carrée de 9x9.

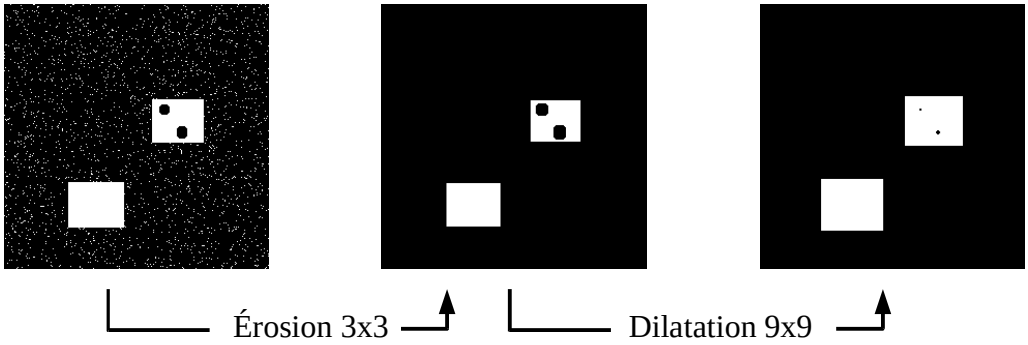


Figure 3.9 - Effets d'une érosion et d'une dilatation sur une image binaire

L'algorithme pour une érosion et une dilatation ressemble beaucoup à celui d'une convolution normale comme décrite dans l'algorithme de la figure 3.10.

Un gros problème de l'opération d'érosion et de dilatation que j'ai rencontré est sa grande demande de calculs. Avec une matrice de convolution de 3x3, tout va, mais dès que l'on dépasse

```
-- algorithme pour une dilatation
pixel := 0
pour tout i dans image
   pour tout j dans élément
      pixel := pixel | (élément[j] & image[i+j])
   image2[i] := pixel
image := image2

-- algorithme pour une érosion
pixel := 1
pour tout i dans image
   pour tout j dans élément
      pixel := pixel & (!élément[j] | image[i+j])
   image2[i] := pixel
image := image2
```

Figure 3.10 - Algorithme d'une dilatation et d'une érosion fait à l'aide d'une convolution

cette taille, le temps de calcul sans optimisation de cet algorithme devient inacceptable sur un ordinateur personnel. On se retrouve loin du 10 images par seconde. Il a alors fallu procéder à une optimisation de l'algorithme. L'idée d'utiliser MMX [1] pour obtenir des résultats acceptables m'est venue à l'esprit, mais il est possible d'arriver à de bons résultats sans avoir recours à ces instructions non portables à l'aide des instructions logiques de base de n'importe quel microprocesseur. Comme une érosion ou une dilatation binaire ne demande que des comparaisons logiques binaires, il est possible d'en faire 32 à la fois sur un processeur 32 bit. Il y a donc une augmentation de la vitesse de l'ordre de 32 fois avec cette optimisation! Ce n'est pas mal.

C'est alors que j'ai développé l'algorithme suivant, figure 3.11. Avec cet algorithme, la seule limitation est que la taille de l'élément structurant doit être <= 15. Cette optimisation fut en fin de compte suffisante et une implantation MMX n'a donc pas été réalisée.

```
-- algorithme de dilatation rapide avec image binaire
pour toutes les lignes j de l'image
  pour toutes les colonnes i de 32 bit de l'image
    sortie := 0X00000000
    pour toutes les lignes l de l'élément

      entrée := (image[j+l][i] décalé de (largeur de l'élément/2)
        vers la droite) + (image[j+l][i-1] décalé
        32-(largeur de l'élément/2) de vers la gauche)

      entrée2 := faire la même chose avec image[j+l][i+1] et image[j+l][i]

    pour toutes les colonnes k de l'élément
      sortie := sortie | (élément[l][k] & entrée)
      entrée := entrée décalé de 1 vers la gauche +
        entrée2 décalé de 31 vers la droite
      entrée2 := entrée2 décalé de 1 vers la gauche

    image2[j][i] := sortie

image := image2
```

Figure 3.11 - Algorithme de dilatation rapide utilisant les instructions logiques du processeur

Ensuite, il faut mentionner que cet algorithme utilise des images binaires avec 32 pixels sur un octet et que les images que j'utilise dans le reste du programme sont de 1 pixel par octet. Le compromis que j'ai décidé de prendre fut de faire une conversion vers la représentation binaire de 32 pixels sur un octet avant le traitement et de faire une autre conversion vers le format de 1 pixel par octet après le traitement. On perd un peu de performance, mais la vitesse globale du système change très peu.

De plus, comme nous l'avons vu auparavant, seulement 2 seules lignes changent entre un algorithme pour une érosion et un pour une dilatation. Dans ce cas-ci, pour avoir une algorithme rapide pour l'érosion, il ne suffit que de prendre de l'algorithme de la figure 3.11 et changer la ligne

```
sortie := 0X00000000
```

par

```
sortie := 0XFFFFFFFF
```

et la ligne

```
sortie := sortie | (élément[l][k] & entrée)
```

par

```
sortie := sortie & (élément[l][k] | entrée) .
```

Cela porte bien sûr à croire qu'il y a une grande redondance et que ce code ne devrait pas se retrouver en double dans le code source pour faciliter la modification de code et le débogage. La première différence ne consiste qu'en une initialisation, il n'y a aucun problème. Toutefois, la deuxième différence m'a apporté un défi intéressant. Cette ligne se retrouve au milieu de trois boucles imbriquées et peut facilement devenir le goulot d'étranglement si codée incorrectement. On pourrait être tenté d'utiliser l'appel indirect d'une fonction et de changer le pointeur selon que la méthode ait été appelée pour une dilatation ou une érosion, mais cela nuit beaucoup à la performance. Pour remédier à ce problème sans doubler le code inutilement, j'ai trouvé deux solutions en C++. L'une est d'utiliser la macro #define. Il s'agit de coder la moitié supérieure de la fonction dans une macro #define, et de coder la partie inférieure dans une autre macro #define. Nous aurions un code qui ressemble à celui de la figure 3.12. Ce qui n'est pas très élégant.

```
#define HAUT \
  char sortie;
   for(int j = 0; j < hauteur; j++) \
   { \
      for(int i = 0; i < largeur; i++) \
      { \
   ...

#define BAS \
   ...
      } \
   } \
   return; \
} \

void dilate(char *image, int hauteur, int largeur, char *element)
char entree = 0x00000000;
#define HAUT
sortie = sortie | (element[l][k] & entree);
#define BAS

void erode(char *image, int hauteur, int largeur, char *element)
char entree = 0xFFFFFFFF;
#define HAUT
sortie = sortie & (element[l][k] | entree);
#define BAS
```

Figure 3.12 - Essai d'optimisation en C à l'aide de macros #define

L'autre méthode implique l'utilisation d'une classe modèle et c'est cette méthode que j'ai utilisée. Cette méthode n'est pas non plus très élégante, mais la vérification des erreurs de syntaxe pour les classes modèles est de loin supérieur aux macros, ce qui rend le codage plus simple et le code plus facile à corriger. Voici l'astuce pour en arriver à ce tour de force avec les classes modèles. La figure 3.13 montre un bout de code possible pour l'implantation. Il y une classe modèle, que nous allons nommer Morph. Cette classe contient une méthode non statique Morph::erosionOuDilatation() qui contient le gros de l'algorithme. Deux autres classes normales cette fois-ci existe aussi: Erosion et Dilatation. Chacune de ces classes contient une méthode statique ou non faireOperation(). Dans cette méthode, on retrouve une seule ligne, ce qui diffère entre une érosion et une dilatation. Ainsi, lorsqu'on veut exécuter une dilatation, on crée une

instance de la classe Morph avec la classe Dilatation comme paramètre. Dans Morph::erosionOuDilatation(), faireOperation() est appelée et comme la classe modèle est crée à la compilation, la méthode faireOperation() est automatiquement incorporée dans erosionOuDilatation comme optimisation du compilateur. C'est cette méthode qui est implantée dans la classe CMorph de HumanLocator. Il y a toutefois un bogue dans Visual C++ 6.0 qui empêche l'appel de la fonction statique de Erosion ou Dilatation si on ne crée aucune instance de la classe. Toutefois, même si un objet est crée, l'optimisation, encore une fois, enlève cette fausse création.

```
class Dilatation
{
public:
   static inline int faireOperation(int sortie, int element, int entree) {
     return sortie | (element & entree);
   }
};


class Erosion
{
public:
   static inline int faireOperation(int sortie, int element, int entree) {
     return sortie & (element | entree);
   }
};


template <class T> class Morph
{
public:
void erodeOrDilate(char *image, char *element, long hauteur, long largeur)
{
   for() {
     for()  {
       for()  {
         for()  {
   ...
             // cet appel sera incorporé lors de la compilation
             // avec optimisation
             sortie = T::faireOperation(sortie,element[l][k],entree);
   ...
         }
       }
     }
   }
}
};

void dilate(char *image, char *element, long hauteur, long largeur){
   Morph<Dilatation> dilatationOp;
   dilatationOp.erosionOuDilatation(image, element, hauteur, largeur);
}
```

Figure 3.13 - Optimisation en C++ avec à l'aide d'une classe modèle

En somme l'érosion et la dilatation permet de se débarrasser de bruit et de boucher des trous qui seraient présents dans un gros objet. Toutefois, ces opérations utilisent beaucoup de temps processeur. Heureusement, elles sont aussi très facilement optimisées, car les instructions qu'elles utilisent sont très simples: des instructions binaires logique. De plus, j'ai développé une méthode me permettant de ne pas dupliquer du code inutilement tout en gardant la même

performance en utilisant de façon inhabituelle une classe modèle. De cette façon, on obtient du code plus facilement gérable que celui où on utiliserait des macros #define.

3.2.6 Segmentation en blobs

Suite à l'érosion et à la dilatation, l'opération suivante consiste à la segmentation de l'image en blobs. Pour exécuter cette opération, il faut faire une analyse de l'image en plusieurs étapes. Les blobs sont par définition des pixels qui sont reliés ensemble pour former une tache dans l'image. Chaque blob doit correspondre à un seul objet en particulier pour qu'il soit ainsi considéré correctement par le système. Ainsi, deux personnes une à côté de l'autre pourraient causer problème.

Quoi qu'il en soit, il faut un algorithme qui va nous permettre de trouver ces blobs. Il y a tout d'abord le choix du nombre de pixels voisins à vérifier. Ceci affecte aussi bien la performance que la précision du rendu. Les choix les plus populaires consistent à vérifier quatre, six ou huits voisins comme illustré dans la figure 3.14.

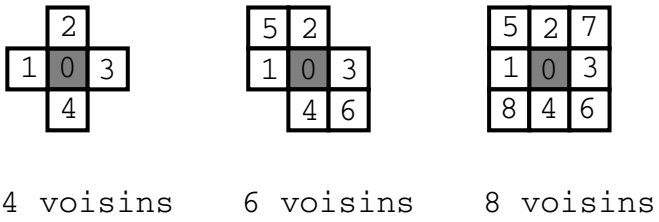


Figure 3.14 - Illustration des pixels voisins possibles pour trois algorithmes différents de segmentation

Question de performance, j'ai décidé d'utiliser la méthode avec quatre voisins. J'ai de plus considéré que dans une image réel, dû à leur instabilité fondamentale, les quelques pixels qui pourraient peut-être faire la différence si on utilisait la méthode à huit ou six voisins n'avaient pas beaucoup de chance de faire une grosse différence. L'algorithme considéré à alors été le suivant, figure 3.15. Il s'agit d'une adaptation de l'algorithme décrit dans [13]. L'implantation de cet algorithme se retrouve dans la méthode CSegment::findBlobs().

```
pour tous les pixels pixel_0 de l'image
  si pixel_0 := 0
    rien faire
  sinon si pixel_1.étiquette et pixel_2.étiquette n'existent pas
    pixel_0.étiquette := nouvelle étiquette
  sinon si pixel_1.étiquette existe, mais pixel_2.étiquette n'existe pas
    pixel_0.étiquette := pixel_1.étiquette
  sinon si pixel_2.étiquette existe, mais pixel_1.étiquette n'existe pas
    pixel_0.étiquette := pixel_2.étiquette
  sinon si pixel_1.étiquette = pixel_2.étiquette
    pixel_0.étiquette := pixel_1.étiquette ou pixel_2.étiquette
  sinon si pixel_1.étiquette != pixel_2.étiquette
    pixel_0.étiquette := pixel_1.étiquette ou pixel_2.étiquette
    pixel_1.étiquette.ajouter_étiquette_mère(pixel_2.étiquette)
```

Figure 3.15 - Algorithme pour la segmentation en blobs à 4 voisins

On se retrouve ainsi avec une image contenant une étiquette pour chaque pixel qui n'est pas égale à 0, autrement dit, égale à 1, comme il s'agit d'une image binaire. Toutefois, beaucoup de ces étiquettes sont probablement équivalentes. La situation qui peut produire ce genre d'équivalence est décrite dans la figure 3.16. Lorsque que deux étiquettes jusqu'alors distinctes se rencontrent plus bas dans l'image, elles ne doivent ensuite ne former qu'une, mais revenir en arrière coûte cher en performance, alors on les ajoute à un tableau d'équivalence.

En fait, au lieu de les ajouter à un tableau d'équivalence en tant que tel, j'ai décidé d'utiliser une méthode que j'ai appelé « l'étiquette mère ». Nous n'avons ainsi pas à parcourir un tableau

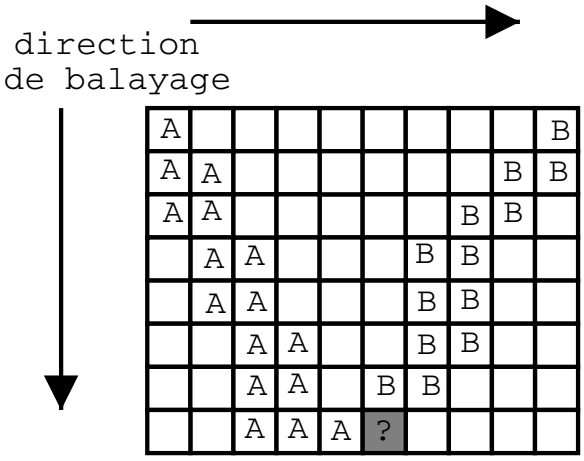


Figure 3.16 - Illustration du problème des étiquettes équivalentes

d'équivalence pour construire séparément une table de conversion ou un arbre permettant de trouver cette « étiquette mère ». Nous avons déjà un arbre prêt à être utilisé. Nous allons tout d'abord décrire l'algorithme de la deuxième étape, la réduction des équivalences, vu dans la figure 3.17. Les algorithmes des méthodes `étiquette::ajouter_étiquette_mère()` et `étiquette::trouver_étiquette_mère()` sont décrites plus loin.

```
liste_blobs := { }
pour tous les pixels pixel de l'image
   si pixel.étiquette n'existe pas
      rien faire
   sinon
      étiquette_mère := pixel.étiquette.trouver_étiquette_mère()
      pixel.étiquette := étiquette_mère
      si liste_blobs.trouver(étiquette_mère) = faux
         liste_blobs.ajouter(étiquette_mère)
```

Figure 3.17 - Algorithme pour la réduction des équivalences

Après l'exécution de cet algorithme, on trouve alors dans le tableau `liste_blob` une liste de blobs contenant des étiquettes uniques. De plus, chacun des pixels de l'image fait maintenant référence à « l'étiquette mère ». Pour que cela fonctionne, nous avons besoin des méthodes `étiquette::ajouter_étiquette_mère()` et `étiquette::trouver_étiquette_mère()` qui

sont d'ailleurs décrites dans la figure 3.18. L'implantation de ces algorithmes se retrouve dans les méthodes `CBlob::getMother()` et `CBlob::setMother()`.

```
-- Nom de méthode:   étiquette::ajouter_étiquette_mère
-- Paramètres:       étiquette_mère (entrée)
-- Valeur de retour: rien
-- Description:  Ajoute de façon récursive l'étiquette mère
--               à la chaîne d'étiquette équivalente dont elle fait partie.
méthode étiquette::ajouter_étiquette_mère(étiquette_mère)
     si mon_étiquette_mère != étiquette_mère -- empêche les mères récursives
       si mon_étiquette_mère n'existe pas
          mon_étiquette_mère := étiquette_mère
       sinon
         -- remonter l'arbre
         mon_étiquette_mère.ajouter_étiquette_mère(étiquette_mère)

-- Nom de méthode:   étiquette::trouver_étiquette_mère
-- Paramètres:       aucun
-- Valeur de retour: étiquette_mère
-- Description:  Trouve de façon récursive l'étiquette mère
--               à la chaîne d'étiquette équivalente dont elle fait partie.
méthode étiquette::trouver_étiquette_mère()
     si mon_étiquette_mère existe
       retourner mon_étiquette_mère
     sinon
       -- remonter l'arbre
       retourner mon_étiquette_mère.trouver_étiquette_mère()
```

Figure 3.18 - Algorithme pour les méthodes ajouter_étiquette_mère et trouver_étiquette_mère

On pourrait se demander si cet algorithme est optimal. Il est impératif tout d'abord de revisiter chaque pixel de l'image pour changer leurs étiquettes à l'étiquette mère. L'algorithme ne peut donc pas être d'ordre de complexité plus bas que O(m) où m est le nombre de pixels de l'image. Ensuite, le pire cas adviendrait si nous avions un seul blob avec n étiquettes qui s'enchaînent toutes. Dans ce cas, il faudrait parcourir en moyenne n/2 étiquettes dans la chaîne d'étiquettes pour chaque pixel, donc un ordre de complexité de $O(m \times n/2)$. Pour réduire ce facteur, il faudrait bâtir une table de conversion reliant une étiquette à sa mère de façon direct. Nous aurions donc un algorithme optimal d'ordre O(m). Toutefois, le besoin ne s'en est pas fait sentir et il n'a pas été implanté dans HumanLocator pour le moment.

Enfin, pour segmenter des blobs dans une image, nous parcourons l'image au complet en fixant des étiquettes pour chaque pixel et on essaie de réunir en une seule toutes les étiquettes qui appartiennent au blob. Je procède pour cette deuxième opération à la technique de l'étiquette mère, ce qui me permet de facilement choisir une étiquette qui remplacera toutes celles équivalentes.

3.2.7 Analyse du mouvement

Finalement, une fois que nous avons trouvé les blobs de l'image, il faut faire une analyse de leurs mouvements, c'est à dire les associer à des blobs d'images précédentes. Dans le cas du HumanLocator actuel, il n'est question que d'associer un blob avec un blob de l'image précédente. Pour ce faire, il ne suffit que d'associer le blob de l'image précédent au blob courant le plus près. Ensuite, il faudrait s'assurer d'avoir le bon blob en faisant appel au pattern matching. Il s'agit de voir si le contenu des blobs associés est suffisamment similaire pour les considérer comme le même objet. Il faudrait bien sûr faire cette analyse avec les quelques blobs les plus près pour avoir plus de chance de le trouver. Toutefois, cette analyse n'est pas encore implantée dans HumanLocator. Pour le moment, il ne fait qu'une analyse sur la distance entre deux blobs.

La méthode la plus simple consiste à bâtir un tableau contenant la distance entre chaque blob de l'image courante avec chaque blob de l'image précédente. L'algorithme est décrit dans la figure 3.19 et son implantation se trouve dans la méthode `HLProcessor::AnalyzeAndProcess()`.

```
pour tous les blob_cour de liste_blobs
  blob_cour.blob_pre := NULL

pour tous les blob_pre de liste_blobs_précédent
  blob_pre.deja_associe := faux

liste_distance := { }
pour tous les blob_cour de liste_blobs
  pour tous les blob_pre de liste_blobs_précédent
    liste_distance := liste_distance +
      { blob_cour.distance(blob_pre), blob_cour, blob_pre }

trier(liste_distance) sur liste_distance[0] en ordre croissant

pour toutes les dist de liste_distance
  si dist.blob_cour.blob_pre n'existe pas et
     dist.blob_pre.deja_associe = faux
  {
       dist.blob_cour.blob_pre := dist.blob_pre
       dist.blob_pre.deja_associe := vrai
  }
```

Figure 3.19 - Algorithme pour trouver les blobs précédents les plus rapprochés des blobs courants

On retrouve ainsi à la fin dans `liste_blobs` des blobs avec un blob précédent associé ou non. Les blobs précédents ne sont de plus associés qu'à un seul blob courant. Nous remarquons ensuite que l'ordre de complexité globale est de $O(n^2)$, comme la complexité d'une boucle imbriquée est de cet ordre, que celle d'un tri est de O(n log(n)) et que celle du parcours de la liste de distance est équivalente à la première boucle imbriquée. Il est fort possible qu'un algorithme avec une complexité moindre existe, car par exemple, il serait bizarre pour un cerveau humain d'avoir à considérer tous les objets d'une salle avant de s'apercevoir qu'un objet se trouve à côté d'un autre. Cependant, comme le nombre de blobs dans une scène ne dépasse que rarement la centaine, la complexité de l'algorithme importe peu.

En conclusion, pour analyser le mouvement des blobs, j'essaie de trouver dans l'ensemble des blobs de l'image précédente celui qui se situe le plus près d'un blob de l'image courante. Ceci est

fait pour tous les blobs de l'image courante. Une analyse avec du pattern matching devrait être faite pour nous permettre de s'assurer d'une façon plus certaine que deux blobs correspondent bien. De plus, si nous attendons une grande quantité de blobs, l'analyse qui possède en ce moment une complexité $O(n^2)$ devrait être amélioré.

3.2.8 Résumé

En conclusion, j'ai tout d'abord procédé au développement de HumanLocator en divisant le codage nécessaire en quelques classes pour implanter l'analyse et le traitement de l'image requis. L'analyse et le traitement se divisent en quatre grandes étapes: la soustraction de l'arrière-plan et son adaptation, la binarisation, l'érosion et la dilatation, la segmentation en blobs et l'analyse du mouvement. HumanLocator achemine alors cette information à l'Xtra qui roule dans Macromedia Director. Il possède de plus une interface graphique permettant de choisir notre caméra et d'ajuster divers paramètres. Il nous permet de plus de visualiser le résultat de différentes étapes de traitement pour permettre d'ajuster leurs paramètres et l'environnement dans lequel la caméra se trouve.

Il s'agit tout d'abord de garder en mémoire une image représentant l'arrière-plan de la scène et de soustraire cet arrière-plan de l'image courante. La soustraction de l'arrière-plan est en effet une méthode efficace pour nous permettre de faire abstraction de l'arrière-plan. Toutefois, il peut survenir que cet arrière-plan change dû à un déplacement de la caméra, aux déplacements d'objets dans la scène et aux changements d'illumination de la scène. Pour compenser pour ces changements, j'ai implanté un arrière-plan adaptatif. Il s'agit d'un simple filtre IIR avec un seul

coefficient qui détermine la vitesse à laquelle l'adaptation se fait. L'adaptation ne tient pas compte d'objets qui ont été trouvés par la étape précédente de segmentation. Pour cette raison, de trop grands changements ne se verront pas automatiquement assimilés. Il faudra faire une remise à zéro manuelle. Il ne faut pas non plus que l'adaptation soit trop rapide, sinon on absorbe les objets trop facilement et l'arrière plan changera radicalement à chaque seconde.

Ensuite, l'étape de binarisation essaie de conserver l'information qui provient seulement d'objets d'intérêt. Pour ce faire j'utilise une simple valeur seuil qui est la même pour tous les pixels. En outre, la méthode que j'utilise tient compte de la différence de couleurs pour augmenter la détection des ombres et des zones éclairées, ce qui permet de faire abstraction de ces zones qui ne sont pas des objets.

La prochaine étape consiste en l'érosion et la dilatation de l'image obtenue. Il ne s'agit ici que d'éliminer le bruit et de boucher les trous introduits par l'étape de binarisation. Le gros problème de cette étape est sa faible performance. Toutefois, ces opérations sont facilement optimisées et j'ai réussi à obtenir une performance satisfaisante.

L'étape suivante de l'analyse consiste en la segmentation de l'image résultante en blobs. Durant cette étape, on trouve tous les pixels qui sont connectés ensemble et on assigne à chaque blob une étiquette. Chaque blob, on l'espère, correspond à un objet d'intérêt, un humain dans le cas présent.

En dernier lieu, HumanLocator procède en une analyse du mouvement. Cependant, elle n'est encore que très primitive. Elle ne fait que la recherche du blob précédent qui se trouve le plus près du blob courant. Il faudrait au moins rajouter une étape de pattern matching pour s'assurer que les contenus des deux blobs soient suffisamment similaires.

Finalement, HumanLocator avec l'implantation de tous ces algorithmes rencontre les spécifications de performance requises. Avec une image de l'ordre de 320 pixels par 240 pixels, une résolution suffisante pour facilement localiser des personnes dans une petite salle, le programme atteint une vitesse d'analyse d'environ 10 images par secondes sur un AMD Duron 1200 MHz. Sur un nouveau processeur, au moins 30-40% du temps processeur serait libre pour permettre à Macromedia Director de gérer des animations intéressantes.

3.3 Développement de l'Xtra

Pour pouvoir communiquer avec Macromedia Director, il a fallu programmer un plug-in, ou un Xtra dans les termes de Director. Avec cet Xtra, il est alors possible d'envoyer l'information sur la position et le déplacement de chacun des blobs d'une scène. D'autres informations peuvent aussi être échangées au besoin. Comme le HumanLocator et son Xtra sont programmés dans Windows NT, il fallait trouver le meilleur moyen pour communiquer entre deux processus. Les IPC (inter-process communication) disponibles dans Windows NT sont listés dans la table 3.3 de même que leurs utilités.

IPC	*Notes*
pipe	• permet d'envoyer et de recevoir un flot d'octets • possède aussi un mode par message
socket	• n'est pas efficace en local, car ne supporte pas les socket BSD, seulement les socket IP
mailslot	• envoie de messages d'une manière non-fiable • supporte la diffusion générale et la multidiffusion, mais ce n'est pas utile pour HumanLocator
COM/DCOM/OLE/RPC	• complexe à implanter
DDE/clipboard	• fait pour interagir avec l'usager
memory mapped file (mémoire partagée)	• ne règle que la moitié du problème, car ne possède pas d'événement associé
WM_COPYDATA	• requiert une queue de message, donc utile surtout pour les applications GUI

Table 3.3 - liste des moyens de communication inter-processus de Windows NT

Ainsi, les sockets ne sont pas efficaces, les mailslots sont peu fiables, les RPC de COM sont compliquées à implanter vu le peu d'information à échanger (elles sont de plus non portables), le DDE et le clipboard sont faits pour interagir avec l'usager (une caractéristique non désirée), la mémoire partagée ne règle que la moitié du problème et WM_COPYDATA demande une queue de message qu'un Xtra de Macromedia Director pourrait difficilement acquérir. C'est pourquoi j'ai décidé d'utiliser une pipe. De plus, dans Windows NT, les pipes supportent l'envoie par message rendant la tâche de séparation des messages beaucoup plus simple.

Il restait maintenant à définir le format des données à envoyer. Le choix fut facile vu la popularité de XML depuis quelque temps. Comme mentionné auparavant, l'outil utilisé pour la tâche fut Xerces C++. J'ai ensuite programmé la classe CXMLIPC pour faire le lien entre une

pipe Windows NT et les API de Xerces C++. Pour transférer des données, on appel CXMLIPC::sendToPipe() avec un objet DOMDocument et le processus qui est connecté le reçoit avec CXMLIPC::recvFromPipe(). Un objet peut ainsi avoir une paire de méthode pour lui permettre de sauvegarder ses données dans un DOMDocument et de pouvoir les récupérer à partir d'un DOMDocument. C'est le cas des méthodes `CBlob::fillDOMDocument()` et `CBlob::loadDOMDocument()` comme on veut envoyer à Macromedia Director l'information sur la position et le mouvement (ie.: le blob précédent associé).

Le gros du travail par la suite fut de comprendre les API de la trousse de développement d'Xtra, ce qui ne fut pas une mince affaire. Le plus gros problème que j'ai eu à régler est de comprendre que Director ne possède qu'un seul fil d'exécution et n'en supporte pas l'addition. Il faut pourtant bien dans mon cas que je fasse parvenir les événements jusqu'au coeur de Director. Si on essaie d'appeler les fonctions internes à celui-ci à partir d'un autre fil d'exécution, il s'en suit d'un plantage immédiat. La façon dont Director s'y prend alors est à l'aide d'un événement « temps libre » (Idle Notification) qui est envoyé à tour de rôle à tous les programmes roulant sur Director. Le problème de cette façon de faire est évidemment que la précision d'un événement que je peux générer est limité par le temps que les autres programmes prennent à redonner le contrôle à Director. Il s'agit alors d'un système « multitâche » coopératif au lieu de préemptif.

3.4 Correspondance avec l'environnement réel

À la suite de toute cela, une fois l'information transférée à Macromedia Director, comment l'interpréter? Elle ne s'agit en fait que de coordonnées dans l'image en provenance de la caméra.

Pour trouver la correspondance avec l'environnement réel, il faut faire un peu de géométrie. Nous avons préconisé de placer la caméra à une hauteur (l) et avec un angle de dénivellation par rapport à l'horizontal (c). Cela nous permet ainsi de connaître la distance des objets en face de la caméra à l'aide de la composante verticale de l'image. En effet, ce que l'on recherche ce sont les coordonnées (x,y) comme si on possédait une vue du haut comme illustré dans la figure 3.20. Ensuite, la hauteur (h) des objets nous serait aussi utile.

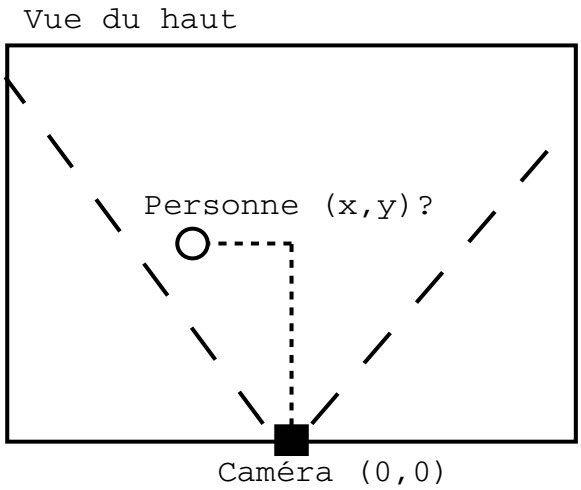


Figure 3.20 - Représentations de coordonnées désirées

Nous possédons une image en provenance de la caméra comme illustré dans la figure 3.21. Il nous ait possible de directement déduire l'angle a et b à l'aide des coordonnées de l'image et la connaissance des champs angulaires (FOV) vertical et horizontal de la caméra. Le champ angulaire de lentilles ordinaires est de 40-50º.

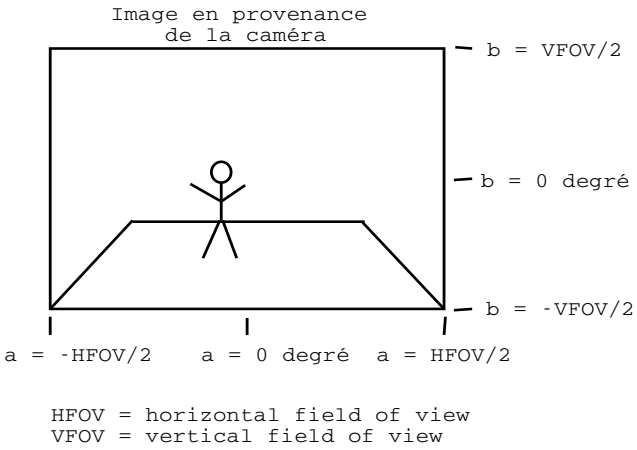
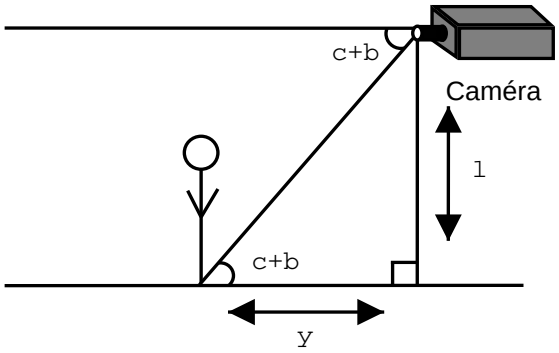


Figure 3.21 - Représentation de l'image en provenance de la caméra

Ensuite, pour trouver la coordonnée (x,y) et la hauteur (h) de l'objet, il ne suffit que de faire un peu de géométrie. Les figures 3.22, 3.23 et 3.24 illustrent les calculs nécessaires.

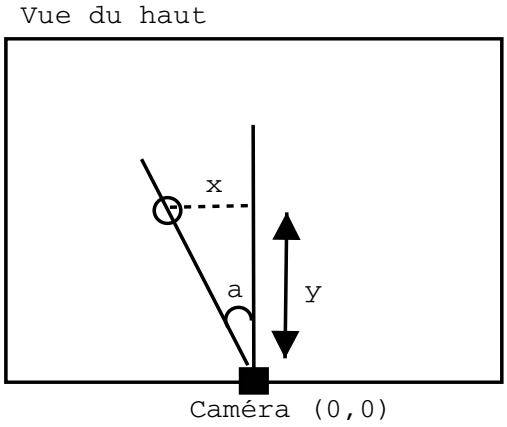


c = angle de dénivellation
de la caméra

$\tan(c+b) = h/y$

$y = h/\tan(c+b)$

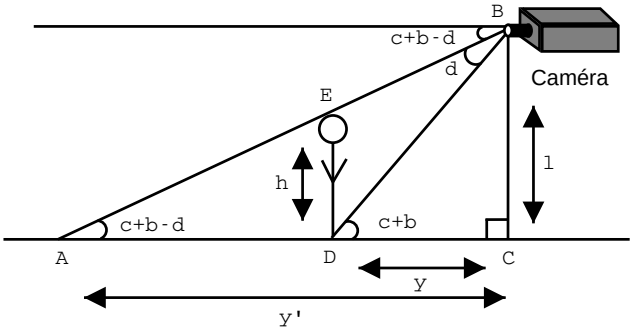
Figure 3.22 - Illustration pour trouver la coordonnée y



tan(a) = x/y

x = tan(a) y

Figure 3.23 - Illustration pour trouver la coordonnée x



c = angle de dénivellation
de la caméra

y = l/tan(c+b)
y'= l/tan(c+b-d)
h/l = y/y'
h = l*tan(c+b-d)/tan(c+b)

Figure 3.24 - Illustration pour trouver la coordonnée h

En résumé, les équations suivantes seront utiles pour trouver les coordonnées (x,y) et la hauteur (h) d'objets qui on suppose sont à la verticale.

$c =$ angle de dénivellation de la caméra
$l =$ hauteur de la caméra
$b = y_{image} - résolution_{image} / 2 \times FOV_{vertical}$
$e = y'_{image} - y_{image}$ où $y'_{image} > y_{image}$

$$y = \frac{l}{\tan(c+b)}$$

$$x = \tan(a)y$$

$$h = l\frac{\tan(c+b-e)}{\tan(c+b)}$$

De même, il est possible de connaître les coordonnées x, y et h maximales en fonction de l'angle de dénivellation de la caméra, de sa hauteur et de ses champs angulaires. Cependant, ces équations ne sont valides qu'en absence de distortion comme définie dans [18]. Si la caméra et la lentille utilisées créent une trop grande distortion, il faudrait alors compenser et tenir compte de sa présence dans les équations.

4 Discussion

Nous avons tout d'abord discuté des arts interactifs et ce qu'ils sont. Les arts interactifs sont des d'arts qui utilisent des moyens techniques pour donner un rôle à l'observateur et ainsi rendre son expérience artistique plus intéressante. Les moyens techniques mis en place sont variés et font habituellement usage de capteurs pour recevoir de l'information sur l'environnement. Dans le cas qui nous préoccupe, nous avons tenté de bâtir un système de peinture interactive. Il s'agit ici d'avoir un système informatique qui reçoit de l'information sur son environnement et de faire varier des animations en fonction de la position, du mouvement et peut-être même des gestes des personnes en face de la peinture.

Pour ce faire, nous avons évalué diverses solutions, incluant l'utilisation de sonars, de capteurs infrarouge et de lidars. Ceux-ci aurait demandé une plus grande compétence que je possédais en électrique et en mécanique, alors nous ne les avons pas utilisés. Pour ce qui est des capteurs de mouvement 3D, ils requéraient tous le port d'un émetteur, une caractéristique non désirée par les artistes. Ensuite, les caméras infrarouges coûtaient beaucoup trop chers, puis l'analyse d'images stéréoscopiques n'était qu'à ses débuts et elle était encore trop lente et complexe à implanter. En dernier lieu, le tapis avec capteurs de pression aurait bien fonctionné, mais il ne nous aurait pas éventuellement permis d'analyser les gestes des personnes. C'est pourquoi nous avons opté pour une caméra ordinaire. À partir des images fournies par celle-ci, nous pouvons faire une analyse et un traitement implantant de cette façon une vision artificielle nous permettant de trouver la position, le mouvement et les gestes de personnes.

Pour ce faire, le système logiciel fut séparé en deux parties. Une partie, appelée HumanLocator, le coeur du projet, fait toute l'analyse et tout le traitement des images en provenance de la caméra. Il en résulte quelques chiffres qui indiquent la position et le mouvement des gens. Pour permettre une interactivité adéquate, un tel système devrait pouvoir analyser au moins 10 images par seconde et laisser du temps processeur à la partie qui doit réagir et afficher diverses animations. Nous avons de plus décidé que cette deuxième partie serait Macromedia Director. Au lieu de programmer un nouvel outil permettant d'afficher des animations, nous avons trouvé qu'il était préférable d'utiliser un outil existant, un outil que les artistes utilisaient déjà abondamment et qui suffisait à la tâche. D'ailleurs Macromedia Director possédait un gestionnaire de plug-in permettant la programmation et l'intégration de plug-in nommé Xtra. Avec la programmation d'un Xtra, il devenait alors possible d'échanger des données entre Director et HumanLocator.

Ensuite, il a fallu développer ces applications, c'est à dire HumanLocator et l'Xtra. Comme Director ne fonctionnait que sous MacOS et Windows et que nous ne possédions pas de Macintosh, nous avons dû utiliser Windows, et plus précisément Windows NT 5 (mieux connu sous le nom de Windows 2000 et Windows XP). Cette plateforme suffisamment stable supportait de plus les services multimédias nécessaires à la capture d'images à partir d'une caméra, c'est à dire DirectShow de DirectX 8. La programmation a été faite avec Microsoft Visual C++ 6.0, la trousse de développement de DirectX 8.1b et la trousse de développement pour les Xtra de Director. De plus, la bibliothèque Xerces C++ 2.20 fut utilisée pour nous permettre de plus facilement échanger des informations entre HumanLocator et l'Xtra.

En ce qui concerne HumanLocator, le développement s'est divisé en quelques classes qui ont implanté l'analyse et le traitement de l'image requis. Il s'agit tout d'abord de garder en mémoire une image représentant l'arrière-plan de la scène. Ensuite, nous faisons une soustraction des images en provenance de la caméra et de l'arrière-plan nous permettant de tenir compte que des objets qui ne font pas partie de l'arrière-plan. De plus, à l'aide des images courantes, cet arrière-plan s'adapte de façon progressive aux variations qui sont trop faibles pour être détectées comme des objets. Par la suite, nous exécutons une binarisation qui tient compte autant de la différence de luminance que de la différence de couleur pour tenter de séparer les variations de luminosité des vrais objets. Cette étape est suivie d'une érosion et d'une dilatation pour enlever du bruit et pour boucher les trous. En quatrième lieu, on segmente l'image en blobs, s'ils existent. Un blob est un groupe de pixels connectés ensemble. Finalement, nous analysons chaque blob de l'image courante et ceux de l'image précédente pour associé un blob courant et un blob précédent, ce qui revient à une analyse du mouvement. HumanLocator achemine alors cette information à l'Xtra qui roule dans Macromedia Director. Il possède de plus une interface graphique permettant de choisir notre caméra et d'ajuster divers paramètres. Il nous permet de plus de visualiser le résultat de différentes étapes de traitement pour nous permettre d'ajuster les paramètres de ces étapes et d'ajuster de même l'environnement dans lequel la caméra se trouve.

Tout d'abord, la soustraction de l'arrière-plan est en effet une méthode efficace pour nous permettre de faire abstraction de l'arrière-plan. Toutefois, il peut survenir que cet arrière-plan change dû à un déplacement de la caméra, à un déplacement d'objets dans la scène ou à un changement d'illumination de la scène. Pour compenser ces changements, j'ai implanté un arrière-plan adaptatif. Il s'agit d'un simple filtre IIR avec un seul coefficient qui détermine la

vitesse à laquelle l'adaptation se déroule. L'adaptation ne tient pas compte d'objets qui ont été sélectionnés par la dernière étape de segmentation. Pour cette raison, de trop grands changements ne se verront pas automatiquement assimilés. Il faudra faire une remise à zéro manuelle. Il ne faut pas non plus que l'adaptation soit trop rapide, sinon on absorbe les objets trop facilement et l'arrière plan changera radicalement à chaque seconde. Puis, il y a probablement des meilleures façons d'implanter l'adaptation de l'arrière-plan, notamment par l'utilisation de méthodes statistiques [5].

Ensuite, l'étape de binarisation essaie de conserver l'information qui provient seulement d'objets d'intérêt. Pour ce faire, j'utilise une simple valeur seuil qui est la même pour tous les pixels. Quoique la méthode que j'utilise tienne compte de la différence de couleur pour rendre meilleure la détection des ombres et des zones éclairées, elle est très simple et n'utilise pas de modèle statistique pour chaque pixel [5]. Il serait en effet peut-être plus efficace d'utiliser un tel modèle, car il permettrait d'être moins dépendant d'une simple valeur seuil pour l'image au complet. Ainsi, les parties plus bruyantes ou avec une résolution moindre n'affecteraient pas les autres parties.

La prochaine étape consiste en l'érosion et la dilatation de l'image obtenue. Il ne s'agit ici que d'éliminer le bruit et de boucher les trous introduits par l'étape de binarisation. Le gros problème de cette étape est sa faible performance. Toutefois, ces opérations sont facilement optimisées et j'ai réussi à obtenir une performance satisfaisante. Si une meilleure performance devient nécessaire, il faudra faire appel à des instructions vectorielles telles que MMX et à des algorithmes appropriés [1].

L'étape suivante de l'analyse consiste en la segmentation de l'image résultante en blobs. Durant cette étape, on trouve tous les pixels qui sont connectés ensemble et on assigne à chaque blob une étiquette. Chaque blob, on l'espère, correspond à un objet d'intérêt, un humain dans le cas présent. Il y a un problème toutefois lorsque deux humains se trouvent trop près l'un de l'autre. Dans ce cas, le système n'est pas capable de les différencier. Pour compenser cela, il serait peut-être possible d'utiliser des méthodes telles que le watershed [2] pour tenter de les séparer en deux entités.

Finalement, HumanLocator procède à une analyse du mouvement. Cependant, elle n'est encore que très primitive. Elle ne fait que la recherche du blob précédent qui se trouve le plus près du blob courant. Il faudrait au moins rajouter une étape de pattern matching pour s'assurer que les contenus des deux blobs soient suffisamment similaires. De plus, il n'y a en ce moment aucun moyen de suivre la trace d'une personne en mouvement. Une analyse plus poussée doit être entreprise pour en arriver à ce niveau. Cela demanderait de garder en mémoire l'état de chaque personne reconnue par le système et demanderait alors le développement d'un algorithme d'intelligence artificielle assez complexe [17].

Par ailleurs, le système ne possède pas encore non plus de routine permettant l'analyse de gestes. Encore une fois, cela demanderait la gestion d'états qui requerrait un autre effort de programmation non négligeable. Pour analyser les gestes de personne, il serait aussi utile d'utiliser le pattern matching pour reconnaître les membres d'un corps [21].

Pour ce qui est de la deuxième partie, l'Xtra, la gros du travail fut de comprendre la trousse de développement de Macromedia Director. Un des concepts importants à comprendre de Director est qu'il ne fonctionne qu'avec un seul fil d'exécution et pour permettre de générer des événements en « tout temps », il faut faire appel à l'événement de temps libre. Je ne trouve pas très élégant le concept de devoir utiliser un événement de temps libre pour en générer d'autres, mais c'est la façon dont il fonctionne.

En dernier lieu, il est possible de faire correspondre les coordonnées de l'image avec les coordonnées de l'environnement. Nous avons décidé que la caméra serait place à une certaine hauteur et regarderait le plancher sous un angle nous permettant de voir d'une façon suffisante les gens qui se trouvent près de la peinture. Avec une série de relations géométriques et en utilisant les coordonnées (x,y) pour lesquelles l'origine est arbitrairement placée à la caméra, il est possible de retrouver les coordonnées d'objets à l'aide de coordonnées de l'image comme s'il s'agissait d'une vue de haut. De plus, si on suppose que les objets sont debout, il est possible de retrouver leurs hauteurs. Une caméra avec un plus grand champ angulaire (60º et plus) nous permettrait d'en voir plus de la scène, toutefois il faut faire attention à la distortion. Si la caméra utilisée possède une distortion importante, il faudra repenser ces équations.

5 Références bibliographiques

1. C. M. MARTINS, Fernando; NICKERSON, Brian R.; BOSTROM, Vareck; HAZRA, Rajeeb. *Implementation of a Real-time Foreground/Background Segmentation System on the Intel Architecture.* Hillsboro, Oregon: Intel Architecture Laboratories - Video and Audio Technologies. http://fizbin.eecs.lehigh.edu/~tboult/FRAME/Martins/martins.pdf (Page consultée le 14 avril 2003)

2. C. RUSS, John. Janvier 2001. « Chapter 6 Section C - Watershed segmentation », *The Image Processing and Measurement Cookbook.* [En ligne]. Asheville, NC: Reindeer Graphics, Inc. http://www.reindeergraphics.com/tutorial/chap6/binary04.html (Page consultée le 14 avril 2003)

3. DOWLING, Kevin. 1996. « Section 10.1.3 Rangefinding devices ». *Robotics: comp.robotics Frequently Asked Questions.* [En ligne]. 90+ pages. http://www.frc.ri.cmu.edu/robotics-faq/ (Page consultée le 14 avril 2003)

4. ÉCOLE POLYTECHNIQUE DE MONTRÉAL. Décembre 2002. *Guide de présentation des citations et des références bibliographiques.* [En ligne]. http://www.polymtl.ca/biblio/citations-guide.pdf (Page consultée le 14 avril 2003)

5. ELGAMMAL, Ahmed; HARWOOD, David; DAVIS, Larry. Juillet 1999. *Non-parametric Model for Background Subtraction.* [En ligne]. College Park, MD: University of Maryland, Computer Vision Laboratory. http://fizbin.eecs.lehigh.edu/~tboult/FRAME/Elgammal/bgmodel.html (Page consultée le 14 avril 2003)

6. FISHER, Bob; PRICE, Sarah; TRUCCO, Emanuele; Wallace ANDREW. Juillet, 1996. « Fundamentals, Models of Geometric Projection », *The Marble Project.* [En ligne]. Edinburgh, Scotland: Heriot-Watt University, Institute for Computer Based Learning; University of Edinburgh, Division of Informatics; Heriot-Watt University, Computing and Electronic Engineering. http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MARBLE/low/fundamentals/geomet.htm (Page consultée le 14 avril 2003)

7. GUIBAULT, François; PESANT, Gilles. Août 1999. « Chapitre 1 - Considérations de conception d'interface ». *INF2700: Interface personne-machine et infographie - Transparents du cours.* Montréal: École Polytechnique de Montréal. p.4. http://www.cours.polymtl.ca/inf2700/transparents/inf2700-01.pdf (Page consultée le 14 avril 2003)

8. KAMPEL, Martin. Septembre 2000. *Stereo-Vision*. [En ligne]. http://www.prip.tuwien.ac.at/Research/3DVision/stereo.html (Page consultée le 14 avril 2003)

9. KLEEMAN, Lindsay. 1999. « Real Time Mobile Robot Sonar with Interference Rejection ». *Sensor Review*. [En ligne]. Australia: Monash University, Intelligent Robotics Research Centre. Vol 19: no 4. p. 214-221. http://www.ecse.monash.edu.au/centres/IRRC/LKPubs/Sensor_review99/sensrev.htm (Page consultée le 14 avril 2003)

10. LOONEY, Carl G. « Appendix 3.A. An Algorithm for Mask Convolution ». *CS 474/674 Digital Image Processing.* [En ligne]. Reno: University of Nevada. http://ultima.cs.unr.edu/cs674/unit3/appendix3ab.pdf

11. MCMANIS, Charles. Novembre 1999. « Cameras ». *Project: 3D Craft.* [En ligne]. http://www.mcmanis.com/chuck/3d/p3dc/p3dc_CMRA.html (Page consultée le 14 avril 2003)

12. MELDRUM, Jason. Mai 2002. *Multimedia for Signal Processing - The Z-Transform.* [En ligne]. Glasgow, Scotland: University of Strathclyde. http://www.spd.eee.strath.ac.uk/~interact/ztransform/ (Page consultée le 14 avril 2003)

13. OWENS, Robyn. Mars 1999. « Lecture 3 - Connected components, and morphological operations ». *233.412 Computer Vision 412.* [En ligne]. Crawley: The University of Western Australia. http://www.cs.uwa.edu.au/~robyn/Visioncourse/Lectures/Lecture3/lecture3.html (Page consultée le 14 avril 2003)

14 PYARAY. Août 1997. *Integer Line Algorithm*. [En ligne]. http://home.pacbell.net/pyaray/articles/lines.htm (Page consultée le 14 avril 2003)

15. RHODY, Harvey. September 2002. « Connected Components Demonstration, Lecture 4 ». *SIMG-782 Introduction to Digital Image Processing.* [En ligne]. Rochester, NY: Rochester Institute of Technology, Center for Imaging Science. http://www.cis.rit.edu/class/simg782/lectures/lecture_4/lec_782_2002_04.pdf (Page consultée le 14 avril 2003)

16. RUDDOCK, Wayne. *Infrared Thermography vs The Visible.* [En ligne]. Advanced Infrared Resources. http://www.infraredthermography.com/irvsvis.htm (Page consultée le 14 avril 2003)

17. SATO, Koichi; AGGARWAL, J. K. *Recognizing two-person interactions in outdoor image sequences.* [En ligne]. Austin: The University of Texas, Computer and Vision Research Center. http://www.ece.utexas.edu/projects/cvrc/koichisato.pdf (Page consultée le 14 avril 2003)

18. VAN KEUREN, Edward; LEEDS, Alicia. Octobre 2002. *Online Optics Reference.* [En ligne]. Washington, DC: Georgetown University, Department of Physics. http://www.physics.georgetown.edu/~vankeu/webtext/TOC.html (Page consultée le 14 avril 2003)

19. VISHNEVSKY, Eugene. *Color Conversion Algorithms.* [En ligne]. http://www.cs.rit.edu/~ncs/color/t_convert.html (Page consultée le 14 avril 2003)

20. WATT, Alan; POLICARPO, Fabio. 1998. *The Computer Image.* Essex, England: Addison-Wesley. 751 pages.

21. WREN, Christopher; AZARBAYEJANI, Ali; DARRELL, Trevor; PENTLAND, Alex. Juillet 1997. « Pfinder: Real-Time Tracking of the Human Body - Modeling the Scene ». *IEEE Transactions on Pattern Analysis and Machine Intelligence.* [En ligne]. Cambridge, MA: MIT Media Laboratory Preceptual Computing Section. Vol 19: no 7. p.780-785 http://vismod.www.media.mit.edu/tech-reports/TR-353/index.html (Page consultée le 14 avril 2003)

Référence logicielle

22. THE APACHE SOFTWARE FOUNDATION. 2001. *Xerces C++ Parser.* Version 2.2.0. [Bibliothèque logicielle]. [En ligne]. http://xml.apache.org/xerces-c/ (Page consultée le 14 avril 2003)

23. *AviSynth.* 2003. Version 2.08. [Logiciel]. [En ligne]. http://avisynth.org/ (Page consultée le 14 avril 2003)

24. MACROMEDIA. 2002. *Director.* Version MX. [Logiciel]. CD-ROM. [En ligne]. http://www.macromedia.com/software/director/ (Page consultée le 14 avril 2003)

25. MACROMEDIA. 2002. *Director SDK.* Version 8.5. [Bibliothèque logicielle]. [En ligne]. http://www.macromedia.com/support/xtras/ (Page consultée le 14 avril 2003)

26. MACROMEDIA. 2002. *Flash.* Version MX. [Logiciel]. CD-ROM. [En ligne]. http://www.macromedia.com/software/flash/

27. MICROSOFT CORPORATION. 2002. *DirectX.* Version 8.1b. [Bibliothèque logicielle]. [En ligne]. http://msdn.microsoft.com/directx/ (Page consultée le 14 avril 2003)

28. MICROSOFT CORPORATION. 1998. *Visual C++.* Version 6.0. [Logiciel]. CD-ROM. [En ligne]. http://msdn.microsoft.com/visualc/ (Page consultée le 14 avril 2003)

29. W3C. Août 2001. *Synchronized Multimedia Integration Language (SMIL).* Version 2.0. [En ligne]. W3C Recommendation. http://www.w3.org/AudioVideo/ (Page consultée le 14 avril 2003)

6 Annexe

Voici un tableau qui résume les trousses de traitement et d'analyse d'images que j'ai trouvé lors de mon investigation. En plus du nom et du lien URL où on peut trouver plus d'information, j'indique le prix des licences pour développeur dans la colonne Coût - D et le prix pour les licences la bibliothèque d'exécution dans la colonne Coût - R. La bibliothèque d'exécution peut être redistribué avec le produit qu'un autre entreprise veut vendre. Les trousses pour lesquelles je n'ai pas trouvé d'information ou pour lesquelles je n'ai pas cherché à savoir, la case est laissée vide. On peut remarquer les coûts exorbitants de telles trousses commerciales.

Nom de la trousse	***Lien URL***	***Coût***	
		D	***R***
Trousses permettant de faire la capture, le traitement et l'analyse d'images			
LEADTOOLS Multimedia Imaging Pro	http://www.leadtools.com/SDK/Multimedia/Multimedia-Imaging.htm	1495 $US	
Matrox Imaging Library D: 3000$ Note: avec pattern matching, runtime = 750$	http://www.matrox.com/imaging/	3000 $CAN	300 $CAN
Common Vision Blox non disponible au Canada	http://en.commonvisionblox.de/		
ADCIS Aphelion	http://www.adcis.net/Home_P.html	3995 $US	1500 $US
MVTec HALCON	http://www.mvtec.com/halcon/		
RobotVisionCAD	http://www.ccs.neu.edu/home/psksvp/lg.htm		0
VYSOR Integration PiXCL 5 Imaging Library	http://www.vysor.com/	650 $CAN	0 si < 20

Nom de la trousse	*Lien URL*	*Coût*	
		D	***R***
VideoOCX Note: traitement de l'image limité	http://www.videoocx.de/	100 $US	 0
Coreco Imaging WiT	http://www.logicalvision.com/		
Trousses ne supportant pas la capture d'images (TWAIN, VFW ou DirectX)			
FSI Automation XCaliper	http://www.fsiautomation.com/		
Optimas Image Analysis Software for Win32	http://www.optimas.com/opdesc.htm		
EureSys eVision	http://www.euresys.com/products/ softwaretools/eVision.asp		
Trousses gratuites sans support pour la capture d'images			
CVIPtools	http://www.ee.siue.edu/CVIPtools/	0	0
Image Processing Library 98	http://www.mip.sdu.dk/ipl98/	0	0
TINA	http://www.niac.man.ac.uk/Tina/	0	0
XMegaWave	http://serdis.dis.ulpgc.es/xmwgus/	0	0
Trousse shareware sans support pour la capture d'images			
NeatVision Note: programmée en Java	http://www.neatvision.com/	100 euro	50 euro

Table 6.1 - Liste des trousses de traitement et d'analyse d'images