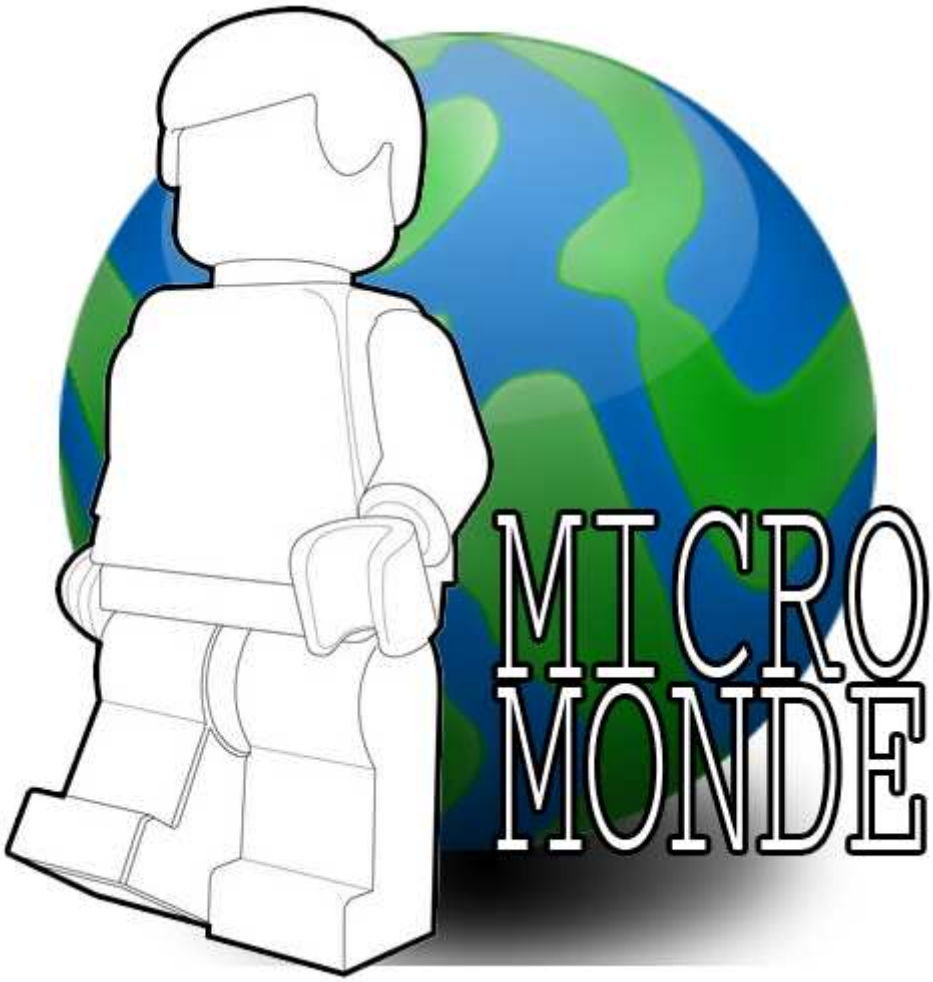
Master 2 INFORMATIQUE 2006/2007
Technologie Avancée du Web



RAPPORT 'MICRO-MONDE' : GROUPE 4

Encadré par M. Pompidor

Bérenger ARNAUD
Sorel DISER
Julien DUTEIL
Anthony GAILHAC
Naïtan GROLLEMUND
Sébastien HALOT

Master 2 Informatique
Année 2007 / 2008

RAPPORT 'MICRO-MONDE' GROUPE 4

Technologie Avancée du Web

Encadré par M. Pompidor

Bérenger ARNAUD
Sorel DISER
Julien DUTEIL
Anthony GAILHAC
Naïtan GROLLEMUND
Sébastien HALOT

SOMMAIRE

1 Introduction
- 1.1 Généralités
- 1.2 Sujet du groupe
- 1.3 Cahier des charges

2 Analyse
- 2.1 Exemple de rendu
- 2.2 Liste des tâches
- 2.3 Répartition des tâches

3 Diagramme des classes

4 Photo des personnages
- 4.1 Prise des photos
- 4.2 Extraction des personnages
- 4.3 Création des photos manquantes par symétrie
- 4.4 Fichier XML des images
- 4.5 Chargement des images : la classe Avatar

5 Objet à afficher
- 5.1 Analyse : calculs des variables
- 5.2 Objets à afficher : classe Acteur et héritage
- 5.3 Affichage des personnages
- 5.4 Affichage du mobilier

6 Création de la vue 3D
- 6.1 Initialisation
- 6.2 Affichage 3D
- 6.3 Intéraction
- 6.4 Mise a jour

7 Intégration des autres groupes
- 7.1 Groupe 1 : Gestion du panorama
- 7.2 Groupe 2 : Création de la pièce
- 7.3 Groupe 3 : Gestion de la Vue 2D
- 7.4 Groupe 5 : Messages
- 7.5 Groupe 6 : Serveur

8 Problèmes et solutions

9 Organisation

10 Conclusion
- 10.1 Le travail en groupe
- 10.2 La découverte de l'AS3
- 10.3 Notre résultat
- 10.4 Perspectives
- 10.5 Remerciements

INTRODUCTION

Généralités

Ce projet prend place dans notre formation de Master 2 Informatique au sein de l'université Montpellier 2 et plus précisément dans l'unité d'enseignement "Technologie Avancé du Web".

Nous avons utilisé le langage ActionScript 3 qui permet de créer des applications au format Flash (format quasi standard dans le domaine du Web) : Il a donc été nécessaire que nous apprenions ce nouveau langage pour le projet.

Le groupe de travail du projet étant composé de plus de 25 personnes, il était nécessaire de s'organiser pour travailler ensemble. La décomposition en plusieurs groupes formés d'un chef de projet et de développeurs n'était pas habituelle pour nous mais avait l'avantage de simuler un contexte professionnel. Nous avons ainsi pu avoir une organisation relativement simple car le rôle de chaque personne était bien défini à l'avance.

Pour faciliter la communication entre les différents groupes, nous avons utilisé un serveur Subversion ainsi qu'un Wiki où chaque personne pouvait envoyer ses données afin de les rendre disponibles pour tout le monde.

Ce rapport présente notre réflexion, notre travail et nos résultats.

Sujet

Construction de la scène "3D" par intégration des images. Chaque client doit pouvoir voir ce qu'il y a en face de lui (dans une vue d'environ 140 degrés), c'est à dire :

- les avatars en avant plan
- les avatars en arrière plan
- la vue panoramique du fond

Cahier des charges

Composé de 6 personnes notre groupe devait créer une scène en « fausse » 3D où apparaîtrait les personnages et le mobilier placés à partir de leurs coordonnées sur la grille 2D.

La scène se mettrait alors à jour par rapport aux messages reçus par le serveur (nouvelle personne connectée, personne déconnectée, personne qui change d'angle, personne qui change de position).

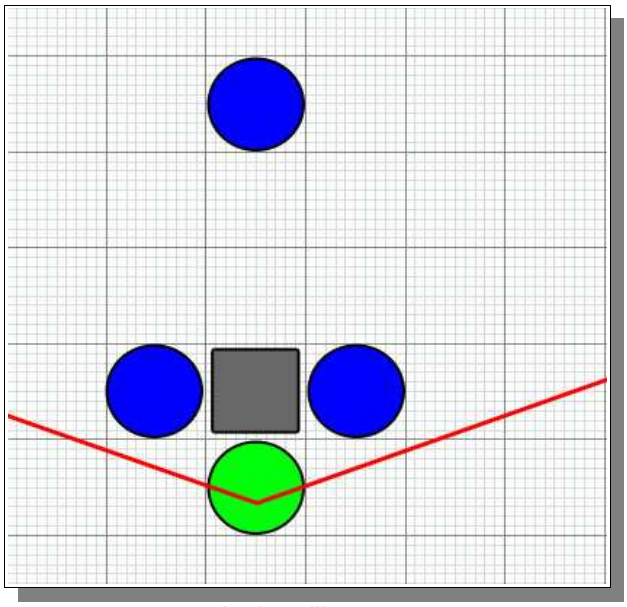
Au niveau intéraction, il a été prévu que :

- Les touches directionnelles fassent tourner le propriétaire de la vue sur lui même, mettant la scène à jour à chaque déplacement.
- La touche 'echap' permettra de passer en vue 2D pour pouvoir se déplacer sur la grille 2D.
- Sélectionner un personnage avec la souris fera afficher l'interface de conversation.

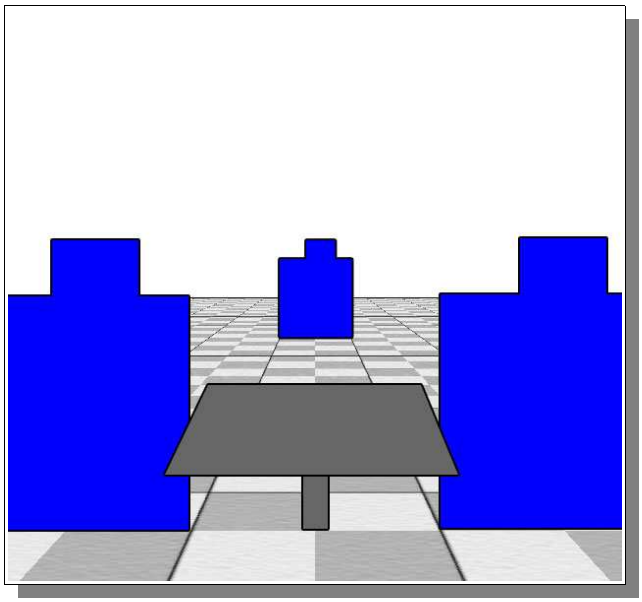
ANALYSE

Exemple de rendu

Afin d'avoir une idée visuelle de ce que devait être notre projet nous avons dessiné un exemple sur un logiciel de graphisme :



Vue de la grille 2D



Vue de la scène 3D

Liste des tâches

- Photographie :
  - Photo prise tous les dix degrés de personnages de Lego. (soit 4 x 36 = 144 photos)
  - Création des personnages en 3D sur un logiciel de graphisme grâce au découpage et assemblage des photos.
  - Intégration au projet.
- Gestion des avatars :
  - Création d'une classe "Personne" contenant toutes les informations : position x,y sur la grille 2D, position x,y,z sur la scène 3D, angle de vue par rapport au propriétaire de la vue, photo vue par le propriétaire, distance entre l'avatar et le

propriétaire...
  - Calcul pour savoir si un avatar est dans l'angle de vision
  - Calcul de l'échelle des avatars vus en fonction de la taille de la pièce, de la distance par rapport à l'utilisateur.
  - Placement de chaque avatar sur un calque pour superposer sur le panorama 3D.
  - Ordre d'affichage de chaque avatar.
- Gestion du mobilier :
  - Création d'une classe similaire à "Personne" pour chaque élément du mobilier qui sera affiché.
  - Dessin du mobilier en 2D puis conversion en 3D.
- Gestion des événements :
  - Touche de déplacement pour faire tourner le propriétaire sur lui même et ainsi modifier la vue avec envoi des nouveaux angles au serveur.
  - Touche 'Echap' permettant de changer de vue: 2D ou 3D.
  - Sélection d'un avatar avec un clic de souris pour communiquer avec son utilisateur.
- Intégration :
  - Récupération des données envoyées par le serveur.
  - Mise en place des fonctions utiles pour autres les groupes.

Répartition des tâches

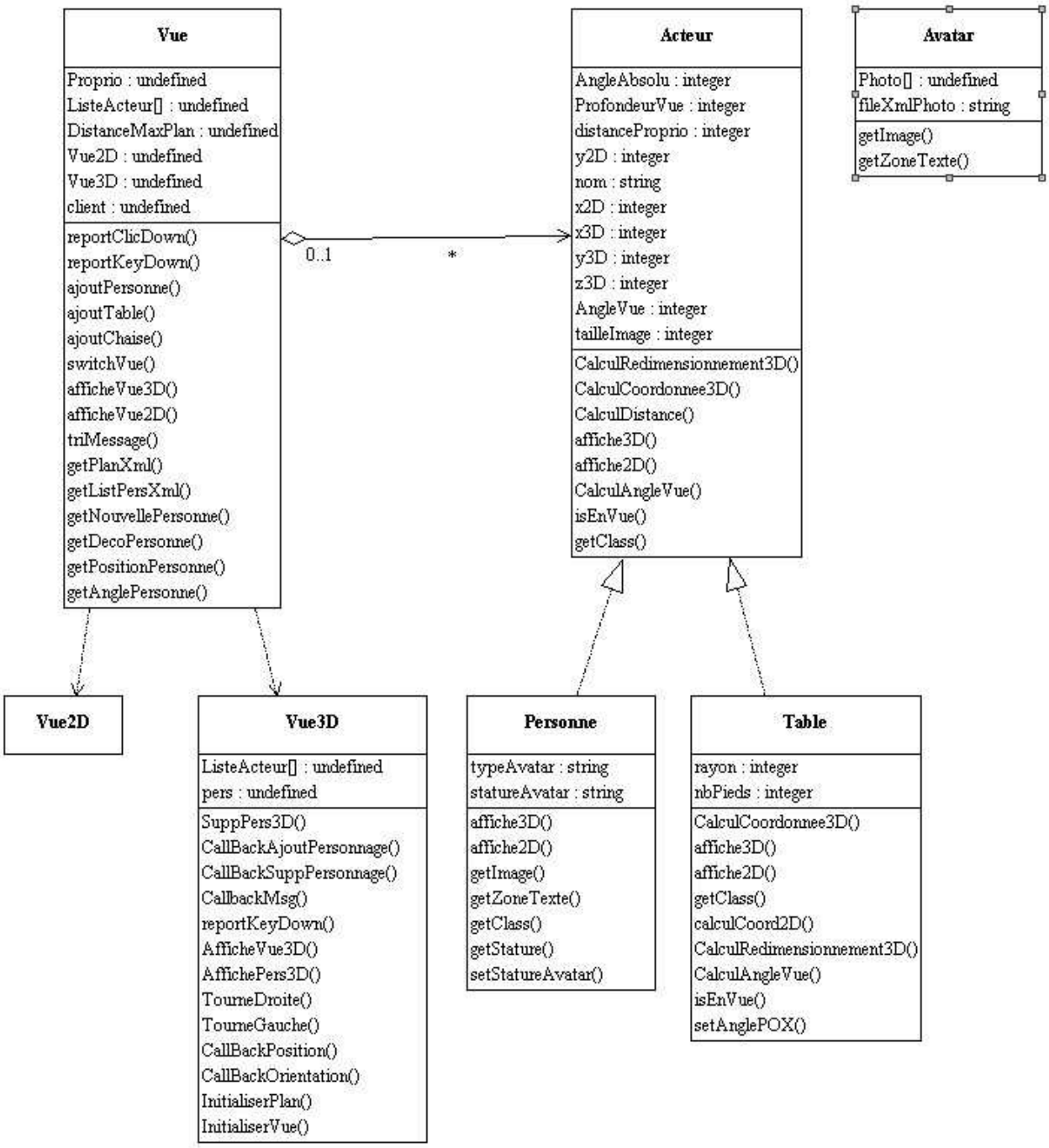
Naïtan GROLLEMUND : Chef de projet
- Coordination et communication entre les groupes
- Analyse
- Création de la structure à partir du diagramme de classe.
- Rédaction du rapport
- Dépannage et aide à la programmation
- Intégration des photos au logiciel par fichier XML
- Intégration des autres groupes (Vue 2D, Serveur...)

Sébastien HALOT : Photographie
- Prise des photos
- Retouche des images

Bérenger ARNAUD, Anthony GAILHAC : Gestion des Avatars et Vue 3D
- Calcul de chaque variable des Personnes
- Implémentation des les fonctions de placement 3D
- Création de la vue 3D et implémentation des callback
- Intégration des autres groupes (Panorama, Plan...)

Sorel DISER, Julien DUTEIL : Gestion du mobilier
- Dessin en 2D du mobilier
- Calcul des variables nécessaire à l'affichage en 3D
- Recalcul des fonctions de placement 3D pour le mobilier
- Aide à la création du fichier XML des images

Diagramme des classes du Groupe



Détails :

La classe avatar sera statique et permettra de charger les photos via un fichier XML.

La classe Acteur représente les objet à afficher, les classes Personne et Table en héritent.

La classe Vue est la classe Client qui va créer les vues 2D ou 3D, il n'y a pas d'héritage.

Chacune des Classes et décisions prises seront expliquées plus en détail dans les parties suivantes.

PHOTOGRAPHIE DES AVATARS

Prise des photos

L'énoncé proposait d'utiliser des personnages de Playmobil, nous avons plutôt choisit de prendre des personnages de Lego car ils ont un avantage: ils sont plus modulables (on peut changer le torse, la tête ou les jambes) ce qui autorise à l'avenir de créer de nouveaux personnages dérivant des personnages actuels (même corps mais on change la tête ou les jambes...). De plus, à l'instar des personnages de Playmobil, les personnages de Lego ont un torse non courbé ce qui rend plus simple l'affichage des messages.

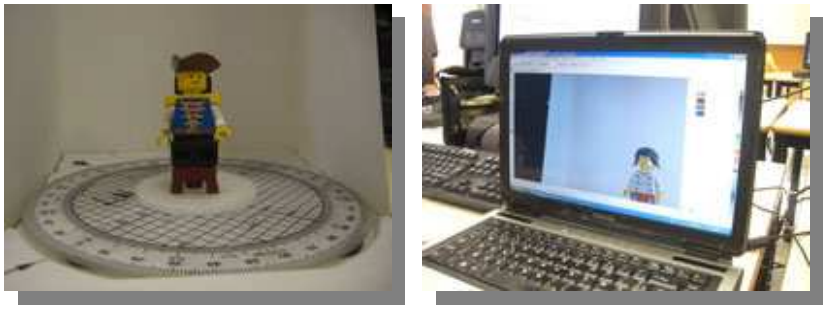
les modèles

Notre objectif était d'obtenir deux personnages différents (homme ou femme) dans deux postures différentes (assise ou debout). Des photos de chaque modèle tous les 10° ont été nécessaires pour arriver à ce résultat. Soit un total de 2 * 2 * 36 = 144 photos.

En utilisant le fait que les personnages sont symétriques selon l'axe y, seules 72 photos ont été réellement prises, les photos restantes étant générées par symétrie sur ordinateur.

Afin de prendre des photos précises et d'ajuster au mieux la lumière, nous avons construit un « mini studio photo » pour nos personnages :

mini studio photo (1)



mini studio photo (2)

Sur chaque photo le personnage devait être pris à un angle juste, la marge d'erreur étant très faible. Pour un meilleur résultat, nous placions toujours l'appareil photo au même endroit grâce à un repère et le personnage était au centre d'un plateau tournant gradué en degrés.

De plus nous savions que notre personnage serait extrait de la photo grâce à un découpage logiciel : L'utilisation d'un fond unicolore a donc été nécessaire.

Extraction des personnages

Le traitement étant le même pour chaque photo, nous avons décidé dans un premier temps de faire un script (ensemble d'actions prédéfinies exécutées dans un ordre donné) sous Adobe Photoshop pour automatiser le découpage du personnage.

Actions nécessaires :

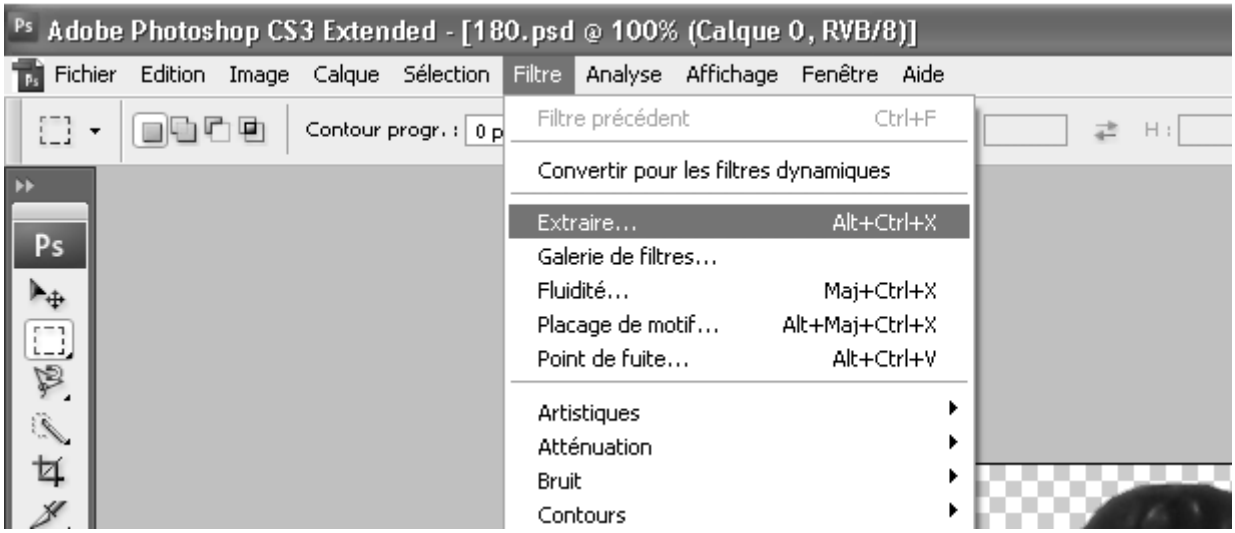
1. Recadrer l'image : on sélectionne le centre de l'image avec un rectangle de sélection d'une taille donnée, pour supprimer les contours inutiles.
2. Sélectionner le fond uni : on utilise l'outil « baguette magique » de Photoshop sur un point défini comme appartenant toujours au fond unicolore de la photo.
3. Ajouter le socle à la sélection : avec un rectangle de sélection on rajoute à la sélection le bas de l'image (pièce Lego servant de support) .
4. Sélectionner le personnage : on inverse la sélection actuelle qui est composée du fond et du socle, on obtient donc le personnage.
5. Copier la sélection.
6. Créer un nouveau fichier avec un fond transparent puis on y insère la sélection.
7. Enregistrer l'image au format GIF (format gérant la transparence et moins lourd que le PNG).

exemple de script d'extraction du personnage d'une photo

L'outil « baguette magique » donne parfois des résultats inattendus (tâches sur la photo, zone de pixels mal compressée, etc.) et donc fausse la sélection du personnage. On notait également que la sélection avec l'outil « baguette magique » est généralement peu précise et laisse des pixels sur les bords du personnages, ce qui posait problème.

L'utilisation du script a donc été abandonnée et nous avons décidé de modifier toutes les images manuellement.

Nous avons utilisé un autre outil qui facilite le détourage d'un personnage : l'outil « extraire »



A l'aide d'une tablette graphique nous avons modifié chaque photo.

Cela a été un travail très minutieux, qui a pris beaucoup de temps et d'attention mais le résultat est incontestable :

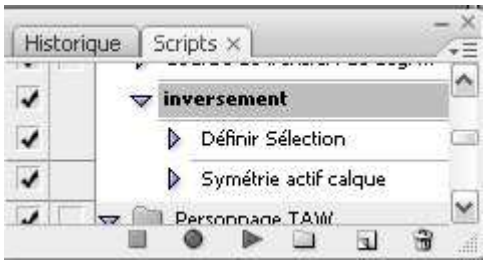
Détourage avec le script

Détourage avec l'outil extraire

Création des photos manquantes par symétrie

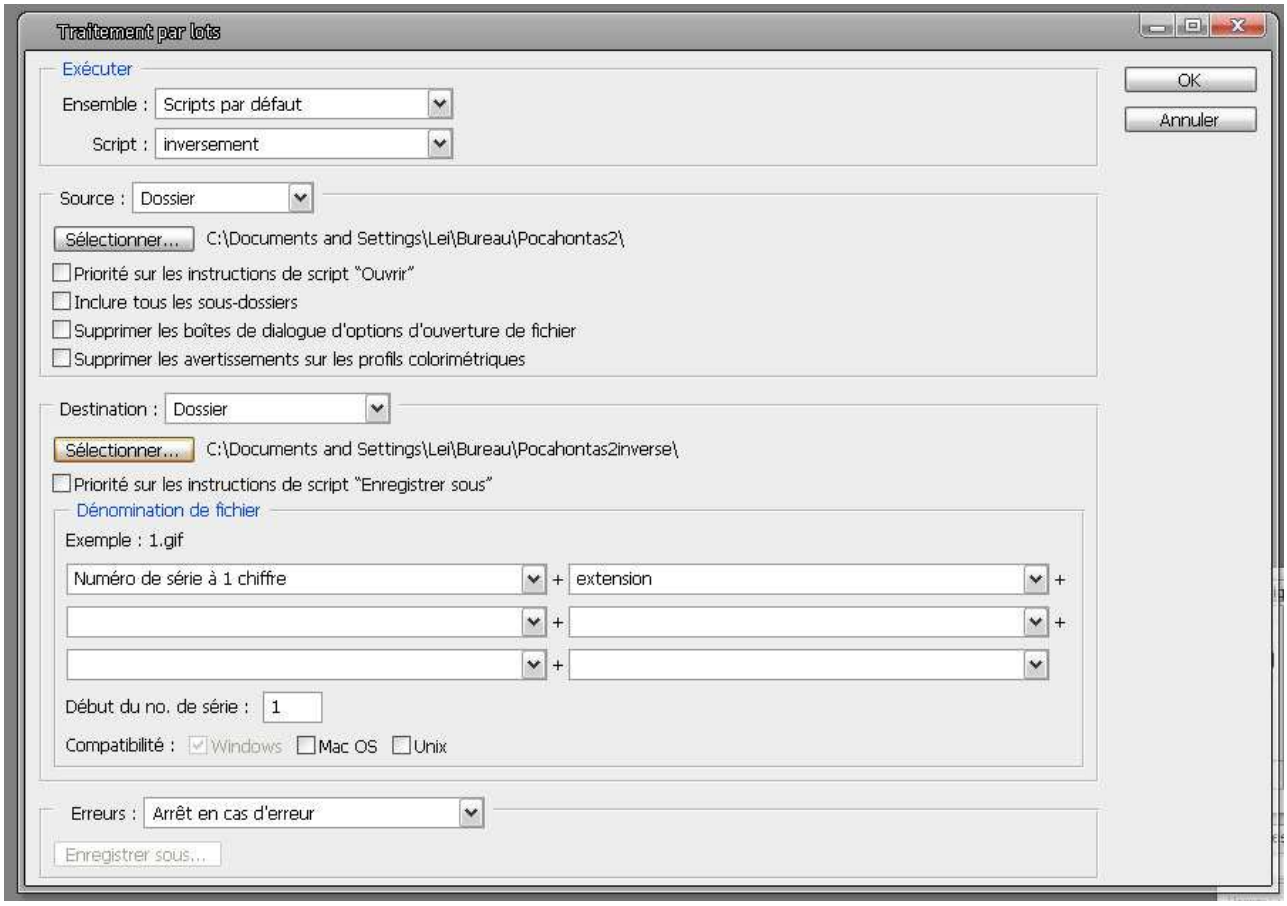
Nous avions pour chaque série 18 photos correspondant aux personnages vus de 0° à 180°, il restait donc à obtenir les images de 190° a 360°.

Cette fois nous avons utilisé un script sans craintes, pour chaque image le traitement est exactement le même : Transformation de type « Symétrie axe vertical ». Ce traitement est sans risque car ne prends pas en compte les éventuels défauts de l'image.



script de symétrie verticale

Ce script a été appliqué automatiquement à chaque photo, nous avons utilisé l'outil de traitement par lot de Photoshop : pour toutes les images d'un dossier on applique un script et on enregistre le résultat avec incrémentation du nom.



Nous avons ainsi obtenu notre série d'avatars vus sous chaque angle, découpés et prêts à être utilisés dans le logiciel.

Fichier XML des images

Nous ne voulions pas que les images soient chargées à partir de leurs adresses directement dans le logiciel, car si le le nom d'une image ou d'un dossier changeait cela nous obligeait à tout recompiler. Nous avons donc décidé d'utiliser un fichier XML où serait stocké les urls relatives de chaque photo. De plus nous y avons aussi stocké les points qui définissent le torse (représentant la zone Texte : là où les messages s'affichent).

Le fichier s'appelle « ImagePersonnage.xml » et voici sa structure :

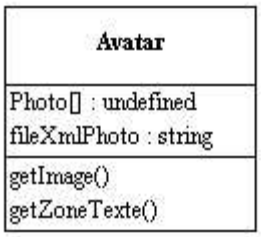
```
<?xml version="1.0" encoding="utf-8" ?>

<ListeAvatar>
  <avatar typeAvatar="pocahontas" statureAvatar="debout" >
        <orientation degres="0">
              <url>Image Personnage/Pocahontas/debout/000.gif</url>
              <zonetexte>
                 <x1>0</x1><y1>0</y1>
                 <x2>0</x2><y2>50</y2>
                 <x3>50</x3><y3>50</y3>
                 <x4>50</x4><y4>0</y4>
              </zonetexte>
        </orientation>
        <orientation degres="10">
              <url>Image Personnage/Pocahontas/debout/010.gif</url>
              <zonetexte>
                 <x1>0</x1><y1>0</y1>
                 <x2>0</x2><y2>50</y2>
                 <x3>50</x3><y3>50</y3>
                 <x4>50</x4><y4>0</y4>
              </zonetexte>
        </orientation>
```

Chargement des images : la classe Avatar

Il ne restait plus qu'à faire une classe qui allait parser le fichier XML et renvoyer les informations.

Nous avons donc créé la classe statique Avatar. Le fait de la rendre "static" permet de ne l'initialiser qu'une seule fois et de ne pas faire suivre un objet de type Avatar en permanence.



Dans le constructeur on parse le fichier XML. Le parsing XML en ActionScript 3 est très simple : Il existe déjà un type XML et toutes les fonctions associées. On stocke les données lues dans un tableau : cela évite de devoir recharger et reparser le fichier XML à chaque accès aux photos, l'accès à un tableau est moins lourd et plus rapide.

tabPhoto[type][stature][orientation]["loader"] : URL de l'image
tabPhoto[type][stature][orientation]["zonetexte"]["x1"] : point bas gauche du torse
tabPhoto[type][stature][orientation]["zonetexte"]["y1"] : point haut gauche du torse
tabPhoto[type][stature][orientation]["zonetexte"]["y2"] : point haut droite du torse
tabPhoto[type][stature][orientation]["zonetexte"]["x2"] : point bas droite du torse

Structure du tableau

Les méthodes getImage() et getZoneTexte() renvoie respectivement l'url de l'image et un tableau contenant les points de la zone texte.

OBJET A AFFICHER

Analyse

Tous les objet vont être afficher sur la scène 3D, ils seront vu par le personnage au centre de la vue, celui que contrôle le client. Nous le nommerons le **propriétaire** de la vue car la vue est créé a partir de sa vision.

Chaque objet à afficher possède :

- Des coordonnées 2D pour l'affichage dans la vue 2D (fourni par le groupe de la vue 2D ou à partir du serveur)
- Des coordonnées 3D pour les placer dans la scène 3D, elles seront calculé à partir des coordonnées 2D.
- Un Angle, on en déduira l'angle vue par le propriétaire et donc la photo ou l'image à afficher.
- A partir de l'angle et des coordonnées du propriétaire que l'on va comparer à ceux de l'objet on va pouvoir calculer si cet objet est dans le champ de vision.
- Une distance entre le propriétaire et l'objet afin de savoir dans quel ordre on va afficher les objets. Le plus éloigné sera affiché en premier et le plus proche en dernier afin qu'ils se superposent correctement. Cette distance permettra aussi de calculer les coordonnées 3D.
- Une taille, calculée afin de donner une impression de perspective.

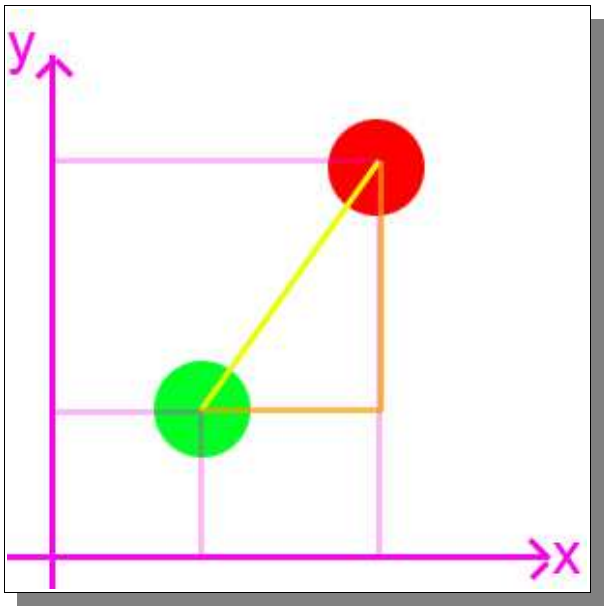
Il faut donc trouver comment calculer toutes ces variables :

- Distance entre un objet et le propriétaire de la vue :

On calcule la distance entre un objet et le propriétaire de la vue à partir de leurs coordonnées 2D.

Cette distance va permettre d'ordonner la pile d'affichage du plus éloigné au plus proche et donc de les superposer correctement. De plus cette distance va aussi permettre le calcul des coordonnées 3D.

Cette distance est propre à chaque client car elle dépend du propriétaire.



On obtient la formule suivante :

Avec l'origine de la vue : x_{vue} et y_{vue}

Avec la position d'un objet quelconque : $x_?$ et $y_?$

$$\sqrt{(x_? - x_{vue})^2 + (y_? - y_{vue})^2}$$

- Passage des coordonnées 2D aux coordonnées 3D :

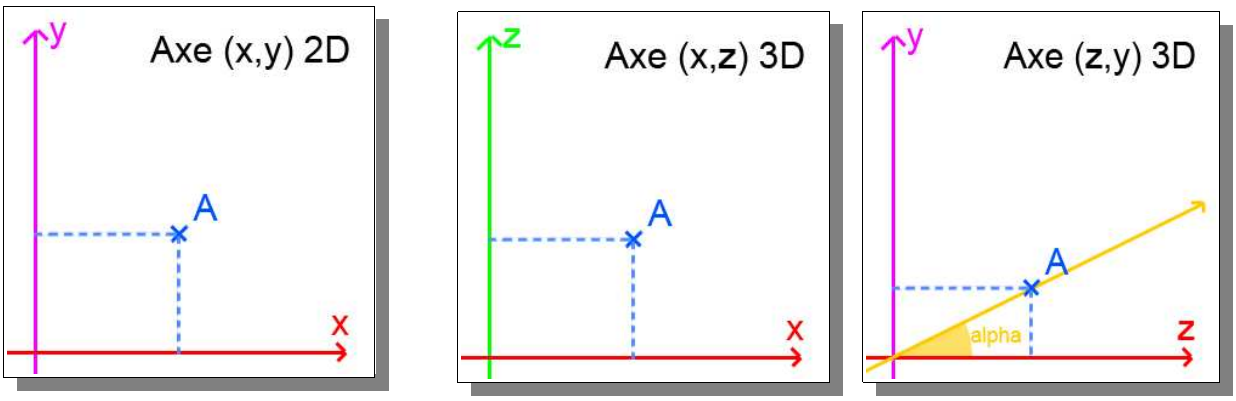
Les coordonnées 3D vont permettre de placer l'objet sur la vue 3D dans la vision du propriétaire.

Les coordonnées 3D d'un objet ne seront pas les mêmes pour tout les propriétaires car elles sont calculées à partir des coordonnées 2D de l'objet, mais aussi de celle du propriétaire et leur distance entre eux deux. Elles devront être recalculées pour chaque modification sur la vue 3D (déplacement, nouveau personnage...) et ne pourront pas être stockées sur le serveur.

On part donc des coordonnées 2D : A ($x2D_a$, $y2D_a$)

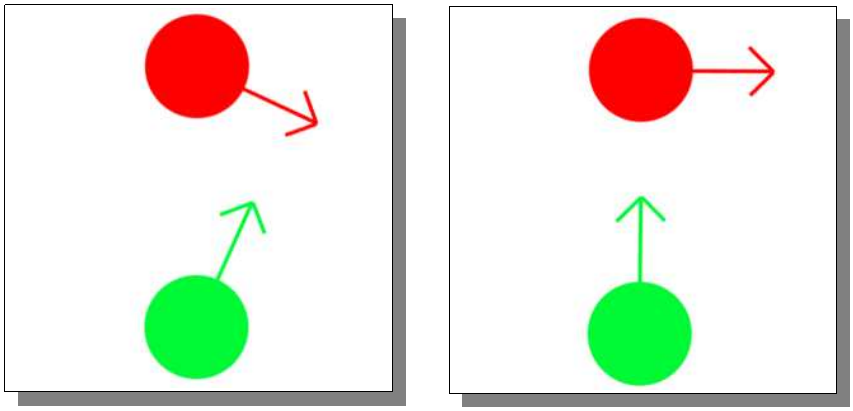
Pour le passage aux coordonnées 3D :

- on pose axe Y (2D) = axe Z (3D) et axe X (2D) = axe X (3D)
- d'où en 3D : A($x2D_a$, y3D, $y2D_a$)
- on pose un axe de perspective dans le repère 3D X,Z avec un angle α représentant l'inclinaison du plan de perspective.
- On trouve ensuite Y à partir de X et α : A($x2D_a$, $x2D_a$ * tan α , $y2D_a$)



- Angle de l'objet vue par le propriétaire de la vue :

Les avatars regardent dans une direction qui leur est propre. On cherche ici à calculer l'angle vue par le propriétaire :



On trouve la formule suivante :

Avec r_{vue} l'angle absolu du propriétaire

Avec $r_?$ l'angle de l'objet

$$(180° - r_{vue} - r_?) \bmod 360°$$

On arrondi cet « angle vue » à la dizaine prés car les photos sont prises tous les dix degrés, ainsi on à un « angle vue arrondi » qui correspond à l'image à faire afficher.

- Savoir si l'objet dans dans le champ de vision du propriétaire de la vue :

On cherche la formule à partir des coordonnées d'un objet et des coordonnées du propriétaire qui permet de savoir si un objet est vu ou pas.

On suppose :

- L'origine de la vue (propriétaire de la vue) : x_{vue} et y_{vue}
- La direction de la vision (en degré) : r
- L'angle de vision : 140°
- La position d'un objet : $x_?$ et $y_?$

Il faut respecter ces deux conditions pour qu'un avatar soit vu :

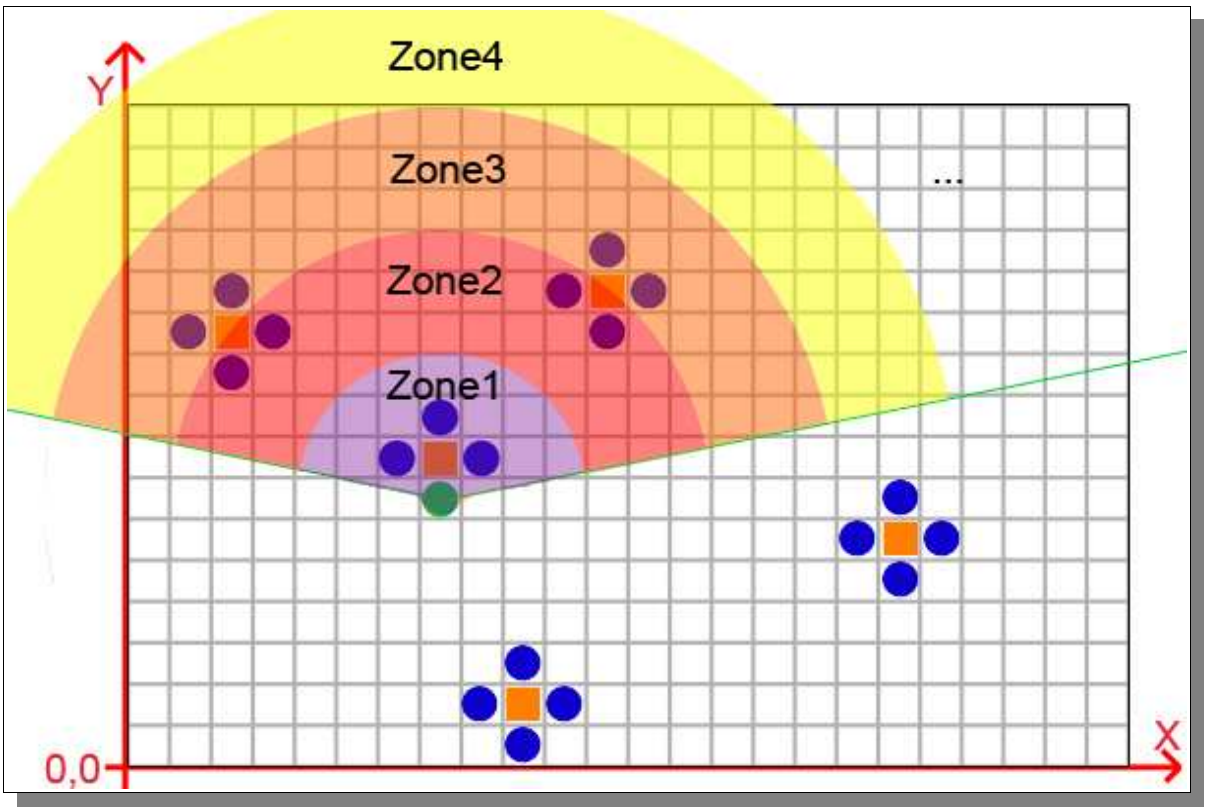
$$\frac{\cos(r_{vue}-\frac{140°}{2})\times y_?-\sin(r_{vue}-\frac{140°}{2})\times x_?}{\cos(r_{vue}-\frac{140°}{2})\times y_{vue}-\sin(r_{vue}-\frac{140°}{2})\times x_{vue}}>0$$

et

$$\frac{\cos(r_{vue}+\frac{140°}{2})\times y_?-\sin(r_{vue}+\frac{140°}{2})\times x_?}{\cos(r_{vue}+\frac{140°}{2})\times y_{vue}-\sin(r_{vue}+\frac{140°}{2})\times x_{vue}}<0$$

- Redimensionnement de l'objet pour l'effet de perspective :

L'énoncé du sujet proposait d'avoir plusieurs tailles prédéfinies et d'afficher une des tailles par rapport à une zone (proche, éloigné, très éloigné...). Nous avons donc défini les zones dans l'image :



Pour savoir dans quel plan est un avatar, on va déjà le classer dans une zone. Par exemple on définit chaque zone large de 5 mètres.

On peut prendre par exemple :

- Premier plan : zone 1
- Deuxième plan : zones 2 et 3
- Dernier plan : zones 4 et plus

On suppose :

L'origine de la vue : x_{vue} et y_{vue}

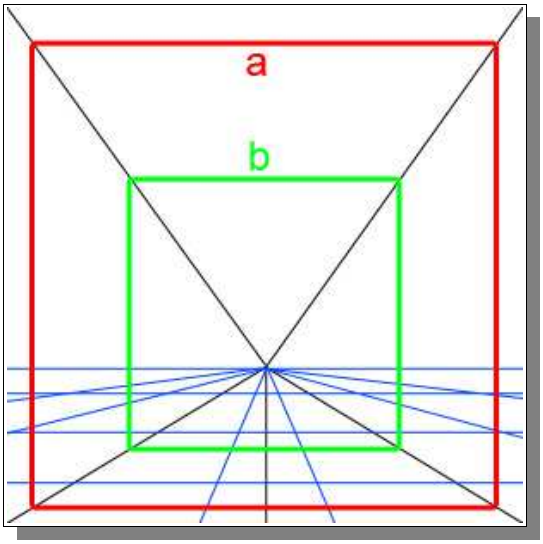
La position d'une objet quelconque : $x_?$ et $y_?$

Pour déterminer la zone, voici une formule "simple" :

$$PartieEntière\left(\frac{\sqrt{(x_? - x_{vue})^2 + (y_? - y_{vue})^2}}{5}\right) + 1$$

Mais cette solution ne donne pas vraiment un effet de perspective donc nous choisissons une autre solution : Modifier la taille de l'objet par rapport à un pourcentage de l'objet initial (comme s'il était très proche).

Les méthodes d'ActionScript permettent de modifier la taille d'une image, on a donc besoin de savoir en quelle proportion cette taille va être modifiée. Nous avons donc changé de stratégie et calculé directement le pourcentage de la taille de l'objet à afficher.



Comme le montre le dessin ci contre la taille de l'objet sera calculé par rapport :

- à la distance entre l'objet et le propriétaire de la vue (distance de A vers B)
- la distance maximale dans le plan (distance entre A et le point de fuite de la perspective)
- à la taille originale (taille de A)

On utilise le théorème de Thalès afin de trouver la taille de la nouvelle image :

$$Taille\ de\ l'image = \frac{(Distance\ max - Distance\ entre\ les\ deux\ personnes) \times (Taille\ originale)}{(Distance\ Max)};$$

La classe Acteur

On remarque que les calculs des objets à afficher, que ce soit la distance objet-propriétaire ou le placement de l'objet dans la scène 3D ou encore le redimensionnement vont être a quelques détails prés les mêmes pour une personne (photo à faire afficher) ou pour le mobilier (dessin en vectoriel).

Nous avons donc décidé de faire une classe abstraite Acteur dont hériteront tous les objets à afficher. Cette classe contient les méthodes de calculs et les variables communes (nom, coordonnées 2D, coordonnées 3D, angle absolu, angle vue...).

Ainsi pour l'affichage de tous les objets sur la scène nous parcourerons une liste d'acteurs, plutôt qu'une liste pour chaque type d'élément.

Cette classe hérite elle même de la classe Sprite propre à ActionScript qui est un objet à faire afficher sur une scène.

Affichage des personnages

On crée une classe personne, héritant de la classe Acteur.

Cette classe contiendra les variables et fonctions propres aux personnes : la stature (assis ou debout), le type d'avatar de la personne (pocahontas ou scaramouche) et les fonctions d'affichages et récupérations de l'image correspondante.

Lors de l'affichage3D on place le Sprite au coordonnées x3D et y3D de l'acteur et on ajoute la photo correspondante récupérer grâce à un appel de la fonction getImage() dans classe Avatar.

Affichage du mobilier

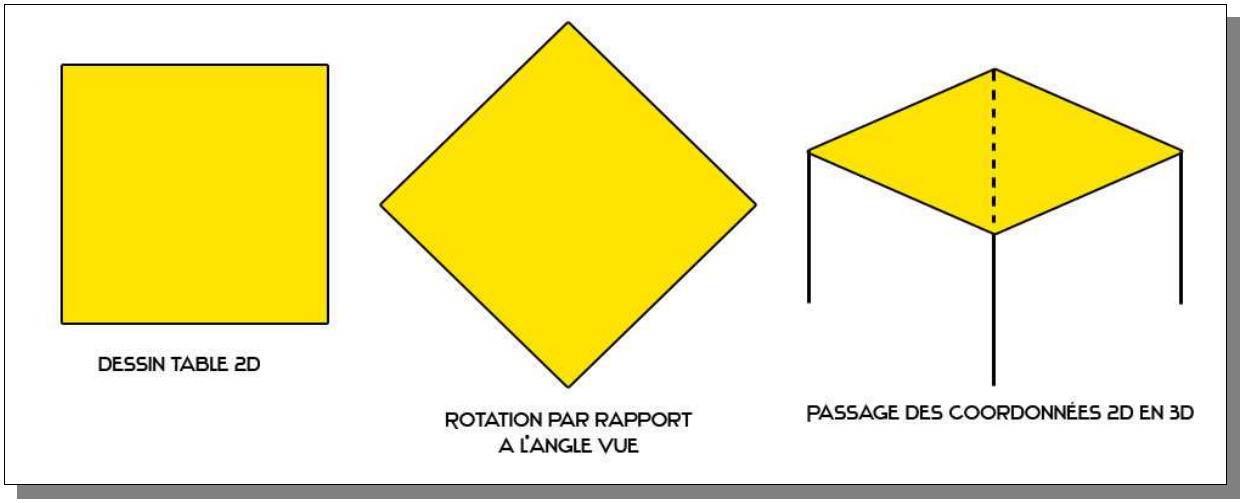
Les objets du mobilier sont plus complexe à réaliser que les personnes car il faut directement dessiner l'objet.

Nous parlerons ici uniquement de la modélisation d'une table car c'est le seul objet du mobilier implémenté actuellement mais avec la même méthode on pourrait rajouter d'autres meubles.

Les tables sont des objet très simple composé d'un carré (le dessus) et quatre segments sur chaque angle (les pieds). Un paramètre permet de choisir un (au centre) ou quatre pieds.

Le principe est de commencer par dessiner l'objet en 2D puis d'appliquer notre fonction pour obtenir les coordonnées 3D de chaque point et redessiner la table en 3D.

- On calcule tout d'abord le redimensionnement de la table en trouvant le rayon correspondant à la distance entre l'objet et le propriétaire à l'aide de la fonction *CalculRedimensionnement3D()*
- La fonction *calculCoord2D()* calcule les coordonnées 2D de chaque point de la table à partir de son centre
- La fonction *CalculAngleVue()* applique une symétrie centrale sur chaque point afin de redessiner la table en fonction de l'angle vue
- Dans cette même fonction on calcule les variables nécessaires au calcul des coordonnées 3D et à la translation : les distances entre les points de la table et la position du propriétaire
- La fonction *CalculCoordonnee3D()* permet ensuite d'obtenir les coordonnées 3D de chaque point de la table par rapport au centre de la vue.
- Pour finir dans la fonction *Affiche3D()* on trace les lignes entre chaque point de façon à obtenir une table.



On note qu'on doit redéclarer beaucoup de fonctions déjà déclarées dans la classe Acteur; en fait on constate,même si le principe de placement pour chaque acteur est le même, qu'il y a des cas particuliers comme les objets pour lesquels on doit faire les mêmes calculs mais pour beaucoup plus de points.

CREATION DE LA VUE 3D

L'affichage de la scène 3D est une des phases clé du projet. C'est sur la scène que les personnages vont s'afficher, que les objets du mobilier vont s'afficher et qu'on va pouvoir discuter.

La vue 3D est créé à partir du personnage client, il peut tourner sur lui même pour voir l'intégralité de la scène, cliquer sur un autre personnage pour lui parler et s'il désire se déplacer il lui suffit de passer a la vue2D où il choisira sa nouvelle position.

Initialisation de la vue

La scène est créé après que le client ait choisi sa position sur la vue 2D.

Lors de l'initialisation de la scène, on doit calculer plusieurs variables :

- On calcule la distance maximale possible entre deux points dans le plan, cette distance va permettre de calculer le redimensionnement des acteurs.
- On parcours la liste d'acteurs et pour chacun on calcule :
    - La distance entre le propriétaire de la vue et l'acteur.
    - Le redimensionnement de l'acteur.
    - L'angle vue de l'acteur par le propriétaire.
    - Les coordonnées 3D de l'acteur.
- On trie ensuite la liste d'acteurs en fonction de la distance par rapport au propriétaire afin de les afficher avec la bonne superposition.

- On ajoute ensuite chaque Acteur (le sprite) à la vue 3D.

On appelle ensuite la fonction Affiche3D() qui va s'occuper d'afficher seulement les acteurs que le propriétaire peut voir dans son angle de vue.

Affichage 3D

L'affichage 3D ne consiste qu'à parcourir tout les Acteurs et les afficher seulement s'ils sont dans l'angle de vue du propriétaire.

On utilise pour cela la fonction *isEnVue()* et les calculs cités dans la partie analyse des objets à afficher.

La liste est parcourue en entier et lorsqu'un acteur est vu alors le Sprite de l'acteur devient *visible.*

Interaction

On gère ici les interactions sur la vue 3D. Il y en a principalement trois :

- Tourner à droite ou à gauche

Lorsque le propriétaire appuie sur la touche directionnelle droite ou gauche du clavier alors la vue du propriétaire change et on met à jour la scène 3D.

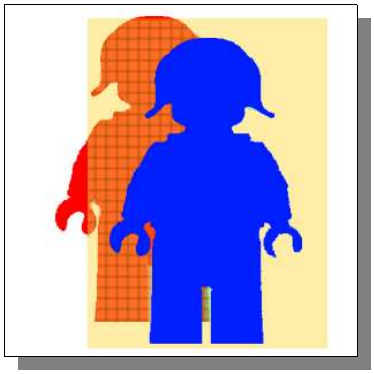
- On commence par changer l'angle absolu du propriétaire.
- On parcours la liste d'acteurs et on recalcule les coordonnées 3D pour chacun.
- On réaffiche la vue 3D avec la fonction d'affichage.

- Cliquer sur une personne pour lui parler

Pour envoyer un message à une personne il faut cliquer sur son avatar. On doit donc gérer les clics sur les personnages.

On détecte les clic dans la vue. Pour chaque clic on va parcourir tout les acteurs et détecter sur lequel on a cliqué.

La difficulté vient de la superposition des avatars. En effet, les personnages sont sur des images donc rectangulaires et les personnages sont au milieu. Le contour est donc transparent et si on clique sur un personne derrière une autre, que leurs images se superpose et qu'on clic sur la partie transparente alors c'est la première personne qui sera sélectionnée.



Si on veut cliquer sur le personnage rouge dans une des parties quadrillées.

On est en fait dans l'image du personnage bleu.

On détecte donc la couleur du pixel cliqué sur la personne.

On doit donc convertir l'image en Bitmap et détecter la couleur sur l'image à l'endroit cliqué, si c'est transparent alors on continue à parcourir la liste, sinon on s'arrête et on récupère le nom de la personne cliquée.

On appel sur le propriétaire une fonction avec le nom qui va afficher l'interface de discussion.

- Basculer vers la vue 2D

La vue 3D n'autorise que le propriétaire à tourner lui même. S'il veut se déplacer dans la vue il doit repasser à la Vue 2D.

Lorsque la touche « Echap » sera pressée on basculera entre les deux vues : on appelle pour cela la fonction *SwitchVue().*

Cette fonction supprime la vue 3D de la scène et ajoute la vue 2D. Et inversement pour basculer de la vue 2D à la vue 3D.

Mise à jour

Il faut prévoir de mettre à jour la vue 3D en fonction des réactions des autres personnes de la vue.

Lorsqu'une autre personne va tourner sur elle même ça doit se voir sur la vue 3D, de même lorsqu'une personne se connecte, se déconnecte ou encore change de position.

C'est le serveur qui doit nous prévenir qu'il y a un changement sur une personne de la vue, nous verrons la communication avec le serveur dans une autre partie, nous nous intéresserons ici à la répercussion de ces changements sur la vue 3D.

Ce sont les fonctions de *callback* qui vont s'occuper des modification sur les vues :

- *CallBackAjoutPersonnage* : personne se connectant et arrivant dans la vue
- *CallBackSuppPersonnage* : personne se déconnectant et quittant la vue
- *CallBackPosition* : personne changeant de position (x2d et y2d)
- *CallBackOrientation* : personne changeant d'angle absolu

Ces fonctions seront présente dans la vue 3D comme dans la vue 2D, car celle ci aussi doit se mettre à jour, nous voyons ici le fonctionnement dans la vue 3D :

Lorsqu'une nouvelle personne se connecte, on va l'ajouter à la vue 3D, on l'ajoute dans la liste d'acteurs et on calcule chaque variable de placement et d'affichage. On ajoute ensuite le Child si nécessaire. C'est ce qui explique qu'on ne réaffiche pas tous les personnages dans la fonction *Affiche3D()* : il est plus léger d'afficher les personnages un par un quand ils changent que de les réafficher tous quand il n'y a qu'un seul changement.

Quand une personne se déconnecte on supprime son entrée dans la liste d'acteurs et le Child correspondant.

Quand une personne change de position on détecte de quel acteur il s'agit et on recalcule ces coordonnées 3D.

On fait de même quand une personne change d'angle.

INTEGRATION DES AUTRES GROUPES

Groupe 1 : Gestion du panorama

Le groupe 1 avait pour tâche de créer un panorama se déformant en fonction du plan de la pièce, l'idée étant d'avoir une réelle impression de 3D, le panorama représentant le mur.

Pour rajouter le panorama à la vue de notre propriétaire, il faut d'abord savoir quelle est la partie du mur qui est dans le champ de vision.

Pour pouvoir déformer le panorama et donner l'effet 3D, les photos sont découpées en « bandelettes ». Une des difficultés est de trouver quelles bandelettes devons nous faire afficher et donc laquelle est la première à afficher, on pourrait se baser sur le point de départ de l'image mais c'est impossible; en effet comme il s'agit d'un panorama on ne peut pas savoir quelle est la « bandelette » a l'origine, la première. Elles sont collées une à une et bouclent.

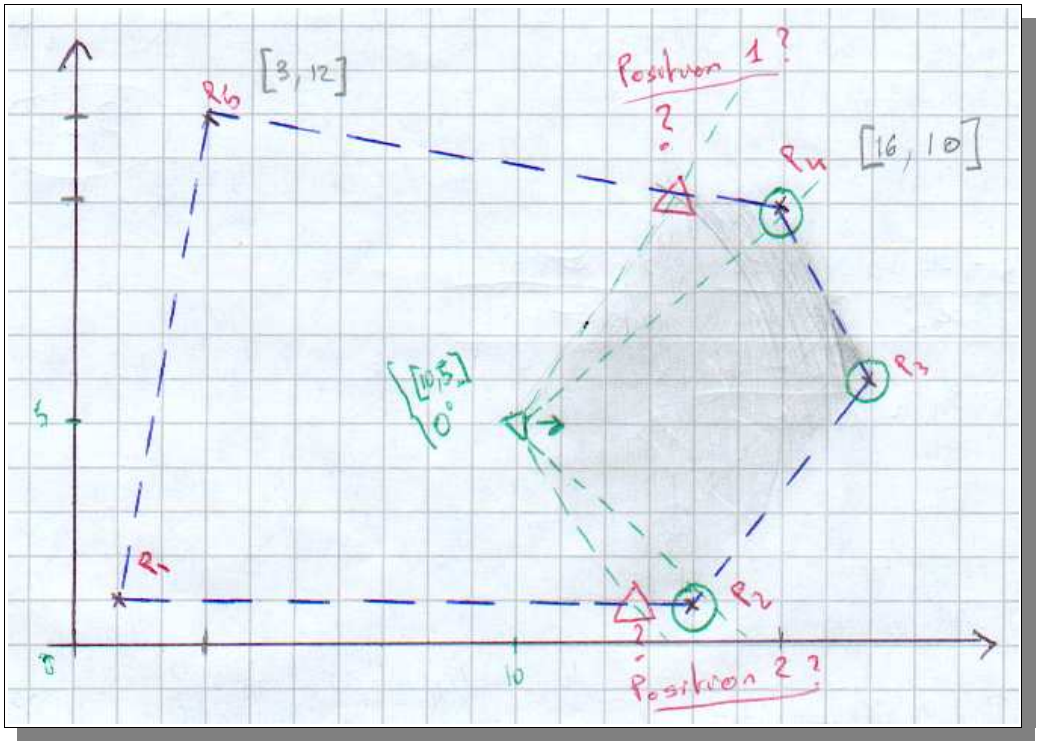
On va donc se repérer par rapport aux angles de la pièces, grâce a une coordination avec le groupe 2 qui s'occupe de la création de la pièce on sait quelle « bandelette » correspond à un angle.

Ensuite si l'on sait la distance entre un angle et un coté de notre vision, si l'on connaît la largeur d'une « bandelette », et si on sait quelle largeur de mur on voit alors on sait quelle image demander à la classe du groupe 1.

Pour calculer tout cela nous avons implémenter les calculs suivant dans les fonctions *Intersection(), Equation1(), DistanceduPoint(), EstDansLeChampDeVision()* :

On a deux cas : soit il y a au moins un angle dans le champ de vision soit il n'y en a pas. On différencie les deux cas en utilisant la fonction *isEnVue()* pour chaque angle de la pièce.

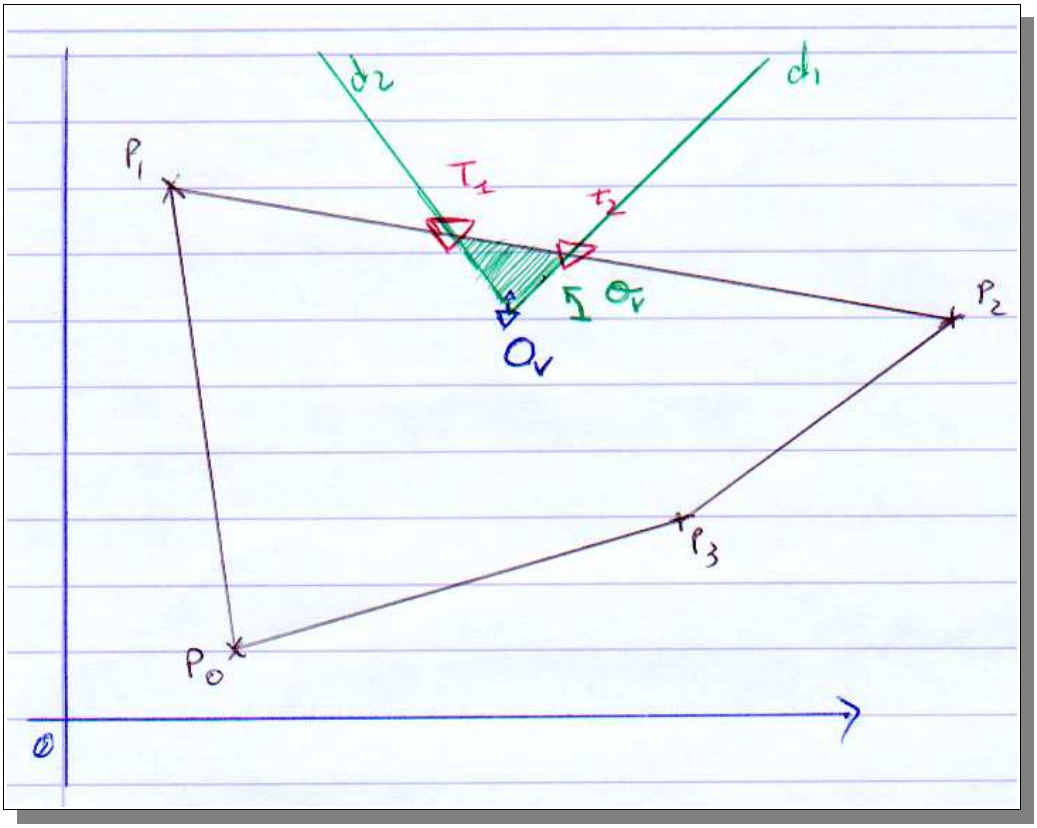
Dans le premier cas : Angles en vue :



Lorsqu'on connaît les angles vues on sait sur quels segments de mur est notre vision (*position 1* et *position 2* sur le dessin). On connaît les coordonnées des angles avant et après ceux qui sont dans notre vision, on a donc facilement les coordonnées du segment. On n'a plus qu'à calculer les coordonnées de l'intersection entre la droite de notre vision et le segment du mur.

On obtient ainsi les coordonnées de début et de fin de notre vision. En sachant quelle bandelette correspondant à un angle et la largeur d'une bandelette par rapport au mur on peut savoir l'image correspondant à notre vision.

Dans le deuxième cas : Aucun angle en vue :



Le principe est à peu près le même, trouver les intersections (*T1* et *T2* sur l'image) afin d'avoir les bandelettes à afficher. Mais il n'y a plus les angles pour se repérer.

On trace les droites *d1* et *d2* correspondant à la vision du propriétaire. Il s'agit ensuite de trouver le mur sur lesquels se trouve les intersections avec d1 et d2, on parcours chaque segment de mur de la pièce. Une fois que l'on connaît le mur correspondant, similairement au cas 1 on calcule les coordonnées des intersections et on trouve l'image du panorama à afficher.

Groupe 2 : Création de la pièce

Le groupe 2 devait faire une interface pour créé le plan de la pièce et produire un fichier XML.

Pour placer nos acteurs et appliquer un effet de perspective nous devons calculer la distance maximale entre deux point dans le plan, cette distance servira ensuite de ligne de fuite pour la perspective.

Cette distance est calculée à partir du fichier XML du plan : On sait qu'il s'agit forcément d'une distance entre deux angles. On a aussi une fonction pour calculer la distance entre deux points. Pour chaque angle du plan on calcule la distance avec les autres angles et on enregistre la plus grande.

Groupe 3 : Création de la vue 2D

Le groupe 3 s'occupait de la grille sur lesquels se positionneraient les personnages. L'intégration de leur travail était très important car nous partagions certaines classes (Vue, Acteur, Personne, Table...)

Par chance ils avait utilisé une architecture similaire à la notre et donc géraient leurs objets à afficher quasi de la même façon que nous, avec un héritage. Nous avons donc seulement du nous mettre d'accord sur les noms des fonctions et des variables.

La classe Vue est une classe commune, car elle fait la transition entre nos deux vues (2D et 3D). C'est aussi cette classe qui va communiquer avec le serveur et recevoir les informations de modification d'un acteur. La classe récupère les messages reçus par le serveur, les lis, les interprète et appelle les fonctions de mise à jour (Callback) correspondantes dans les deux vues.

Il reste la transition entre les deux vues : la fonction *Switch()* que nous avons décrit plus haut va être appelé et ajoutera ou supprimera la vue courante pour passer à l'autre vue. Les vue 3D et 2D ne connaissent pas directement la Vue mais peuvent y accéder grâce a l'attribut *this.parent*.

Groupe 5 : Gestion des messages

Le groupe 5 était chargé de gérer l'envoie et la réception de message. L'intégration de leur travail était simple : Appeler sur fonction sur le propriétaire lorsque le client clique sur une personne.

La gestion de clic sur les personnes était déjà prévu (voir événements souris plus haut) et donc il s'agissait juste d'appeler la fonction qu'ils nous fournissaient.

Ils ont travaillé sur leur propre version de la classe Personne et y ont ajouté leurs classes pour la gestion des Messages.

Pour l'intégration il a simplement fallu faire un *diff* entre les deux classes Personne et intégrer leur code. La majeure partie de cette intégration a été faite par le groupe 5.

Groupe 6 : Le serveur

Le groupe 6 avait pour rôle de créer le serveur et donc la communication avec le client. Ils ont décidé de créer un serveur en Java et les messages transitant serait au format XML.

Le serveur stocke le plan du micro-monde reçu par le groupe 2 (gestion du plan) au au format XML ainsi que les photos du monde.

Lorsque le client se connecte (constructeur de la Vue) on doit envoyer un message au serveur pour qu'il nous envoie le plan et la liste des personnes actuellement connectées. On envoie une *demande* :

```
<demande nom='pseudo' />
```

On doit attendre d'avoir reçu le plan et la liste des personnes pour créer la Vue2D, on utilise pour cela un minuteur qui se rappelle tant que le plan et la liste n'ont pas été reçus.

Dans la vue 2D on choisit la position et l'orientation du propriétaire et on passe ensuite à la vue 3D par la fonction *switch()* vu précédemment. Lors de cette transition on doit avertir le serveur qu'une nouvelle personne s'est connectée afin qu'il prévienne tout les autres clients pour qu'ils se mettent à jour. On envoie le message *user* :

```
<user pseudo="toto">
    <x>1</x>
    <y>2</y>
    <orientation>90</orientation>
    <stature>debout</stature>
    <type>pocahontas</type>
</user>
```

Lorsque le propriétaire tourne sur lui même dans la vue 3D, on doit avertir le serveur afin qu'il avertisse les autres clients pour qu'ils mettent à jour la photo de l'avatar correspondant a leur vision. On envoie le message *position* :

```
<position pseudo="zorro">
    <x>5</x>
    <y>5</y>
    <stature>assis</stature>
</position>
```

Lorsque le propriétaire se déconnecte il envoie aussi un message au serveur qui va prévenir les clients. On envoie le message *deco* :

```
<deco pseudo="zorro" />
```

On envoie donc des messages au serveur qui va les rediriger vers les clients. Il faut prévoir l' écoute des messages : on utilise la fonction triMessage(). Cette fonction va être appeler à chaque fois que l'on va recevoir un message du serveur grâce à un écouteur sur l'objet client.

On tri les messages reçus a partir de leur racine et on appelle les fonctions correspondantes à chaque message. Ces fonctions vont ensuite appeler les fonctions *callback* dans les vue 2D et 3D.

Cette intégration a été difficile car il a fallu être en étroite collaboration entre le groupe 6 et le groupe 3 pour décider des types de message et s'entraider pour le debogage.

PROBLEMES ET SOLUTIONS

Malheureusement tout ne s'est pas passé exactement comme prévu et nous avons rencontré plusieurs problèmes. Certains ont été résolu et d'autres pas encore.

C'est l'intégration qui a causé le plus de problème, les autres étant souvent du à des erreurs d'inattention lors du codage.

Au niveau de la programmation

Certaines erreurs ont trouvées rapidement des solutions, il s'agissait en général de problèmes purement de programmation, elles étaient dûes à la méconnaissance du langage que nous avons découvert au fur et à mesure.

Par exemple, la communication avec le serveur a posé problème car même si les codes de la version test et la version intégré correspondaient exactement ils ne pouvaient pas se connecter. En fait, le problème venait d'ActionScript qui semble s'exécuter de façon asynchrone, il exécutait le reste du code pendant qu'à coté il se connectait avec le serveur et comme le reste du code nécessitait des informations du serveur alors le programme plantait. Nous avons résolu ce problème en modifiant la connexion au serveur de façon à rendre le déroulement de l'exécution synchrone.

Ce problème de gestion asynchrone nous a posé problème plusieurs fois, pour la lecture du fichier XML des images, ActionScript chargeait le fichier et continuer à s'exécuter en parallèle et on arrivait rapidement a des fonctions qui nécessitaient les informations du fichier. Ici le problème a été résolu avec un système d'*Interval* qui met en attente une fonction tant qu'une variable booléenne ne passe pas à *true*.

Au niveau de la coordination

Un gros problème par exemple est la conversion des coordonnées 2D de la grille en coordonnées 3D en pixel : le groupe du plan a donné des coordonnées qui correspondaient à leur grille de dessin du plan, le groupe de la vue 2D se sert des coordonnées en pixel pour placer les personnages et donc on se retrouve avec des placements aberrants (une table en 10,10 et une personne en 600,450 quand la distance maximale ente deux acteurs est de 10). Cela n'a aucune conséquence pour les travaux des deux groupes mais quand nous ajoutons les deux acteurs dans la vue 3D alors on se retrouve avec des problèmes. De même le placement et le calcul du panorama est impossible sans cette cohérence donc on a un problème. Faisant confiance aux coordonnées des deux groupes nous nous somme aperçu de ce problème qu'une fois l'intégration terminée et donc cette incohérence persiste.

Au niveau de la gestion

S'il reste des erreurs dans le programme rendu c'est majoritairement à cause de notre gestion du temps et je pense que cela a été pareil pour la plupart des groupes. Le travail de chaque groupe a pris beaucoup de temps et du coup l'intégration a été négligée alors que l'on aurait dut prévoir la moitié de temps consacré à la programmation.

De même la partie analyse a peu être été un peu négligée, surtout au niveau de l'entente entre les groupes. Elle a du se faire au fur et à mesure de l'avancement alors que certains points auraient clairement dûs être précisés dés le départ. Par exemple, la coordination sur les classes communes entre notre groupe et le groupe 3 (vue 2D).

ORGANISATION

Tout au long du projet nous avons du rester en contact permanent, vu l'effectif de notre groupe il fallait se tenir au courant en permanence de l'avancement de chaque membre du groupe et des autres groupes.

Pour cela il a été mis en place avec l'accord du tuteur et de tous les membres du projet un serveur subversion hébergé par *Google code* à l'adresse suivante *http://code.google.com/p/pompitheque/*

Un Wiki était aussi à notre disposition avec le SVN il nous a beaucoup servi pour se coordonner.

Malheureusement le SVN a été trop peu utilisé (uniquement par le groupe 4 et 5) et le Wiki n'était pas fréquemment mis à jour. Mais à l'intérieur de notre groupe nous avons joué le jeu et ce fut très utile.

Au niveau de la programmation, nous avons utilisé Adobe Flex Builder est un plugin d'eclipse permettant de faire de l'Action Script.

CONCLUSION

Le travail en groupe

Le projet est composé de 24 personnes reparti sur le 6 groupe ayant chacun des tâches très différentes à accomplir. C'est un gros projet autant au niveau du travail qu'au niveau humain. Il a fallu organiser tout le monde et coordonner les travaux. Notre atout était d'avoir un chef de projet dans chaque groupe qui coordonnait les travaux à l'intérieur de son groupe mais aussi à l'extérieur avec les autres chefs de projet.

A l'intérieur de notre groupe nous nous sommes organisé en nous répartissant les tâches mais nous avons pu observer que tout ne se passait pas comme prévu. Certaines tâches ont pris plus de temps que prévu et d'autres se sont avérées beaucoup plus compliquées, engendrant d'autres tâches. Certaines personnes du groupe aussi n'ont pas pu se tenir à leur tâches, ajoutant ainsi des tâches aux autres membres du groupe. Et de même on s'aperçoit après cette expérience que le rôle du chef de projet n'est pas simplement la coordination et la rédaction du rapport mais aussi un rôle de « couteau-suisse », entre les problèmes de communication et d'organisation, il se doit de pouvoir remplacer un membre sur une tâche ou remplir une tâche sans auteur.

A l'extérieur du groupe il a aussi fallu préparer et exécuter l'intégration. Notre groupe était au centre du projet et tout les autres groupes avaient une partie commune avec notre partie. L'intégration a donc été difficile : nous devions savoir ce que faisait chaque groupe et où se délimitait notre travail, il fallait aussi prévoir des points d'entrée et des points de sortie pour communiquer avec les autres groupes. Nous avons eu des difficultés dans ces tâches là car malgré la présence des chefs de projet et du tuteur parfois les sujets étaient mal délimités et on apprenait que l'on devait faire une tâche au moment où nous en avions besoin. Néanmoins nous avons réussi à nous coordonner et à intégrer chaque partie.

En tant que chef de projet, ce projet m'a permis d'avoir une première expérience de notre futur rôle dans les entreprises, ce fut très enrichissant car nous n'avons pas ou peu été formé au management ou à la gestion de projet et cette première expérience m'a permis de voir que le travail d'encadrement d'un projet était très intéressant.

La découverte d'un nouveau langage : ActionScript3

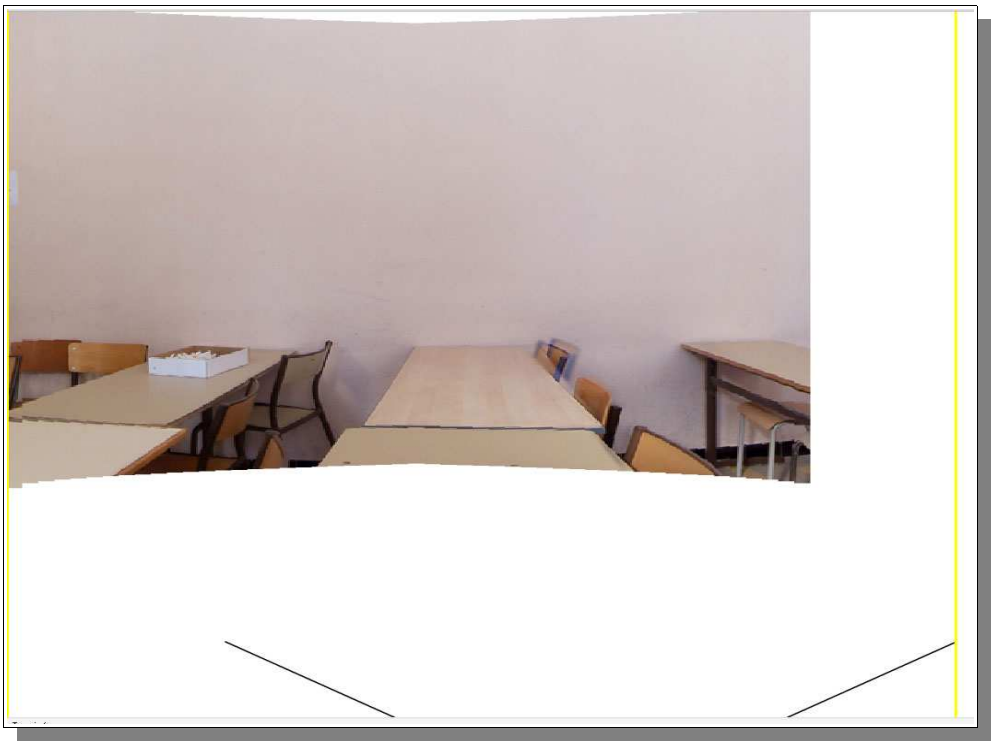
Le projet a été développé en ActionScript3, un langage que, pour la plupart, nous ne connaissions pas. Il a donc fallu rapidement le découvrir et apprendre au fur et à mesure de l'avancement du projet.

Cela nous a permis d'apprendre à nous adapter rapidement à la nouveauté et d'acquérir une grande autonomie dans l'apprentissage d'un langage.

De plus l'ActionScript est un langage générant des programmes Flash, format très utilisé dans le domaine du Web et donc ces connaissances nous serons très utile par la suite dans notre futur emploi pour ceux suivant le parcours DIWEB.

Résultat obtenu

résultat avec plusieurs personnes connectées dans le micro monde



résultat avec une personne seule dans la vue et l'affichage du panorama

Si dans la théorie tout semble correct, en pratique notre résultat n'est pas un produit fini.

La gestion de la « fausse » 3D est correcte mais le résultat est lourd, le chargement des photos ralenti l'application et le calcul du panorama est trop important et alourdi les intéractions. ActionScript ne semble pas optimiser pour de gros calcul mathématique et donc pose problème.
Par manque de temps l'intégration n'est pas totalement fini est donc certaines étapes clé posent quelques petits problèmes lors de l'exécution.

Mais le programme fonctionne, on peut le tester sans problème, le résultat obtenu correspond au cahier des charges : un monde en 3D avec des personnages représentés par des photographies. Le logiciel obtenu est original et inédit et avec des améliorations il pourrait être commercialisable et intéresser les utilisateurs.

Perspectives

Comme précisé au dessus beaucoup d'améliorations sont à apporter. En premier lieu terminer l'intégration et debuguer tout ce qui pose encore problème.
Dans le produit final au niveau de la vue 3D on pourrait rajouter d'autre personnage et même proposer aux utilisateurs d'ajouter leurs propres personnages, on pourrait rajouter le déplacement directement dans la vue 3D avec les touches directionnelles, ou encore on pourrait modifier la vision en rajoutant la possibilité de regarder en haut ou en bas. Pour améliorer l'immersion il faudrait aussi rajouter un sol et un plafond et modéliser d'autres éléments du mobilier.

Certaines améliorations seraient facilement implémentables car nous avions vu les possibilités lors de la programmation (vision haut et bas, ajout de personnage, ajout de mobilier par exemple) et d'autre paraissant pourtant simple serait beaucoup plus long et complexe (plafond et sol notamment).

De plus notre code est très commenté afin de pouvoir être compris et repris facilement et beaucoup d'amélioration sont déjà prévu dans le code. Tout cela rendant notre programme facilement évolutif.

Remerciements

Nous tenons tout d'abord à remercier Mr Pompidor pour son encadrement sur le projet, pour l'attention qu'il nous a porté et son aide tout au long de l'avancement du projet.
Nous remercions aussi chaque membre du groupe pour leurs travaux et leurs aides mutuelles. Merci aussi à tout les chefs de projet et participants qui se sont beaucoup investis.