Projection adaptée d'images sur surfaces non-planes

Membres de l'équipe :

Elouan ARGOUARCH

Julien BRICAUD

Axel KELHETTER

Farid KARAM

Tuteur : Titus Zaharia, professeur et directeur du département ARTEMIS : Advanced Research and Techniques for Multidimensional Imaging Systems

Table des matières

I Introduction : Contexte du projet

I - 1 Cahier des charges

Le but de notre projet informatique est d'adapter la forme et les dimensions d'une image à une surface non plane afin d'observer une projection correcte de l'image traitée. Pour cela on peut étudier le principe du « texture mapping » : faire un plaquage intelligent qui permet de conserver la géométrie d'une image lorsqu'on plaque celle-ci sur une surface non plane. Il sera intéressant de se lancer dans la modélisation 3D de la structure sur laquelle on souhaite projeter. Pour cela il existe différentes méthodes informatiques (deep learning, ou utilisation de la même image selon deux prises de vue différentes « 3D-R2N2: 3D Recurrent Reconstruction Neural Network").

L'objectif actuel du projet est de déterminer un modèle 3D d'une structure pour pouvoir calculer les déformations à appliquer à une image que l'on souhaiterait projeter sur la structure.

De plus, puisque nous n'avons pas le matériel nécessaire à la projection de l'image, on souhaite réaliser une simulation informatique de la projection d'image sur le modèle 3D précédemment déterminé.

I - 2 Première approche de l'application

Le travail à réaliser se découpe en trois parties, qui représentent les trois fonctions que devra réaliser notre application :

La première fonction de l'application est de modéliser une surface 3D à partir d'une ou plusieurs images : la reconstitution du modèle 3D à partir d'image 2D se fera soit grâce à l'utilisation d'un algorithme utilisant du deep learning, appelé 3D-R2N2 développé à l'université de Stanford soit par une étude des profondeurs dans les images (en se basant sur des points repères ou sur l'éclairage dans les images).

La deuxième fonction de l'application est la déformation d'une image pour l'adapter à la surface lors de sa projection. cette déformation de l'image se basera sur des calculs géométriques et mettra en oeuvre des procédés de déformation préalablement codés.

La troisième fonction est la réalisation d'un programme de simulation de projection de l'image déformée sur la modèle 3D de surface. Elle servira à joindre les 2 fonctions précédentes et consistera à mettre en oeuvre la reconnaissance du support de projection et la déformation de l'image à projeter. Quant à la simulation de projection elle se fera grâce aux fonctionnalités des logiciels employés.

Lors de ce projet nous avons choisi de développer nos programmes en langage Python.

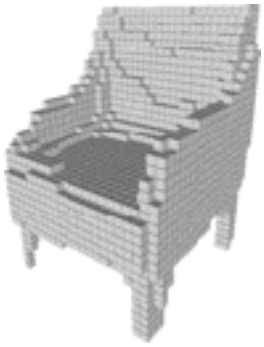
II Réalisation du projet

II - 1 Étapes du développement

II - 1.1 Modélisation 3D

La première étape du projet consiste en la modélisation d'un bâtiment à partir de photo. Notre tuteur nous a aiguillé vers un algorithme de Deep-Learning développé par l'Université de Stanford nommé R^2N^2. A partir des différents codes proposés, nous avons réussi à faire tourner cet algorithme pour les modèles test.





Cependant, le réseau de neurones n'est, à priori, entraîné que pour des objets particuliers tels que des fauteuils ou des avions, c'est pourquoi les modèles 3D résultants d'images de bâtiments ne sont absolument pas satisfaisant. Nous avons donc cherché à entraîner le réseau de neurones sur des bases de données de constructions et leurs modèles 3D associés. Nous avons donc écrit des fonctions nous permettant de traiter des images afin qu'elles puissent être traitées par l'algorithme de l'Université de Stanford. D'une part une fonction permettant de redimensionner des images à la taille voulue (fonction edit_image(name,w,h) du programme resize_image.py). D'autre part les programmes présents dans le dossier « Binary Matrix » nous ont permis d'extraire une base de données d'images depuis une archive au format binary matrix présente sur le site https://cs.nyu.edu/~ylclab/data/norb-v1.0-small/.

En parallèle de ce travail, nous nous sommes formés à l'utilisation de la Librairie Python PyOpenGL qui nous a permis de réaliser la simulation d'une projection d'image sur un objet dans l'espace. Nous avons trouvé un script qui réalise une projection sur un ensemble de cube, ce qui nous a permis de comprendre comment définir un objet 3D avec OpenGL, comment manipuler des sources de lumière et des textures.

Enfin on peut aussi remarquer que le résultat de l'algorithme de modélisation 3D est formé de cubes. Nous verrons dans la suite que nous avons fait une approche des structures 3D par des carrés avec OpenGL, ce qui permettrait d'utiliser nos programmes de projection d'image sur les modèles 3D du premier algorithme.

Pour modéliser une surface en 3D nous avons utilisé la bibliothèque OpenGL qui permet d'afficher des cubes de tailles variable. On construit donc une surface composée de nbface*nbface carré dont la distance selon l'axe z varie, nbface étant une variable définissant le nombre de carrés qui compose le modèle de structure. Les carrés composant la surface ont tous la même taille en pixel qui est égale à carre/nbface, carre étant la taille en pixel du plus grand carré inscrit dans l'image que l'on souhaite projeter.

En outre les carré ont également tous la même longueur h qui est une longueur OpenGl, c'est à dire qu'elle définit la taille des carré lors de l'affichage.
Pour chaque carré, on définit la distance selon l'axe z à l'aide d'une matrice que nous avons nommé lrd, cette matrice est définie avec une double boucle sur les carré. On peut définir par exemple une surface Gaussienne:

```
sigma = 0.1
lrd=[[-np.exp(-0.5*(((i-nbface/2)**2+(j-nbface/2)**2)/
(5000*sigma**2)))/(sigma**2*2*np.pi) for i in range(nbface)] for j
in range(nbface)]
```

C'est ce modèle de surface Gausienne que nous utilisons dans la plupart de nos tests car elle met en valeur les déformations de l'image que nous devons corriger.
Ensuite, nous avons défini dans le script animation.py des fonctions permettant d'afficher les faces des carrés, que nous utilisons dans la fonction main avec les valeurs de la matrice lrd définie.

Initialement nous avions prévu d'utiliser une surface modélisée par un ensemble de carrés dont le format est .obj car c'est le format des objets retournés par l'algorithme 3D-R2N2 mais étant donné que nous n'avancions pas du coté de l'algorithme de machine learning nous nous sommes rabattus sur une surface modélisée par des carrés OpenGL. Cette approche se base sur le fait qu'il est possible d'échantillonner n'importe qu'elle surface 3D avec des carré. Ainsi lorsque le nombre de carré augmente le modèle se raproche de la structure 3D initiale. A titre indicatif, dans notre script animation.py nous avons inclus le chemin path_to_object ainsi que la fonction points. Le chemin était supposé être le répertoire où se trouvait le modèle de structure et la fonction servait à remplacer la liste lrd pour définir les profondeurs des carrés.

II - 1.2 Mapping

Pour projeter une image sur notre structure OpenGL nous avons d'abord supposé que le projecteur se trouvait à une distance infinie de la structure. Cela permet de négliger les variations de profondeur des faces OpenGL suivant l'axe z. On fait alors correspondre à chaque pixel de l'image un point sur la structure OpenGL, cela s'appelle du Texture Mapping.
Notre algorithme de Texture Mapping commence par définir le plus grand carré qu'il est possible de couper dans l'image d'origine. On divise ensuite notre image carré en nbface*nbface petit carré tous de mêmes tailles. Le programme division.py enregistre toutes les petites images dans un même dossier qui porte le nom de l'image d'origine. Le programme animation.py va alors coller chaque petit carré sur les faces OpenGL correspondantes.

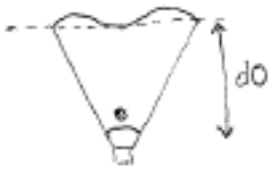
II - 1.3 Projection

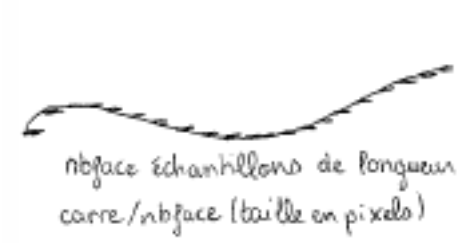
Pour une application concrète, la solution de texture mapping n'était pas suffisante car elle est équivalente à une projection à distance infinie, les rayons lumineux étant parallèles entre eux. Nous avons donc cherché un moyen de pouvoir simuler une projection réelle. La bibliothèque OpenGl proposait des moyens de projections, mais avec cette solution il était très compliqué de modifier l'image par la suite. Nous avons donc opté pour une utilisation du texture mapping pour simuler un projecteur. La première étape est de définir la position du projecteur par un vecteur décrivant ses coordonnées dans l'espace. Le projecteur est également caractérisé par un angle theta caractérisant l'ouverture des faisceaux lumineux émis par le projecteur. On définit également une distance caractéristique du projecteur noté d0, à laquelle l'image projetée et la surface ont la même taille. On calcule cette distance selon:
`d0=carre/(2*tan(theta/2))`, carre est la longueur du coté de la surface
Généralement, pour tous nos tests ultérieurs, nous avons placé le projecteur aux coordonnées: x=Carre/2; y=Carre/2; z=d0, c'est à dire que nous avons placé le projecteur au centre de la surface et à la distance caractéristique d0.

Ensuite, pour réaliser une projection réelle, pour chaque cube de la structure nous avons multiplié la taille de l'image à découper dans l'image que l'on souhaite projeter par distance/d0 où distance est la distance du cube considéré relativement au projecteur. Elle est calculée selon:
`distance=projecteur[2]-lrd[i][j]`

II - 1.4 Traitement de l'image

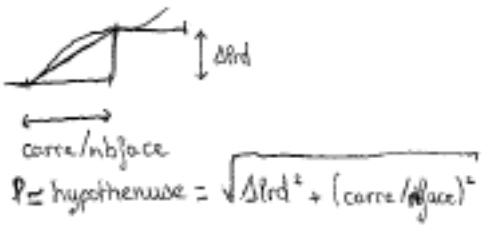
Pour modifier l'image et qu'elle paraisse non déformée une fois projetée sur la surface, nous avons d'abord commencé par réfléchir à un problème à une dimension, où la surface est donc équivalente à un lacet.





Etant donné que nous travaillons avec des surfaces constituées de cubes OpenGl, nous effectuons nos calculs sur l'échantillonnage du lacet qui correspond à notre ensemble de carrés.

On émet ici l'hypothèse que les coté des cubes que nous utilisons ont une longueur très petite devant la longueur du lacet sur lequel on souhaite projeter l'image. Alors on peut approximer la portion du lacet sur laquelle on souhaite projeter l'image comme étant l'hypoténuse du triangle rectangle dont la base est le coté du cube et dont la hauteur est la différence de profondeur des deux cubes adjacents (Δlrd). On note cette longueur hypoténuse.



La première étape est d'adapter la taille de l'image que l'on souhaite projeter sur le lacet, on définit la longueur du lacet comme étant la somme pour tous les cubes du lacet de la longueur hypoténuse.
Il est important que le nombre de cubes qui constitue le côte de notre surface soit impair pour pouvoir utiliser le cube central comme référence, c'est à dire le cube duquel on part pour réaliser notre projection en découpant l'image.

Ainsi pour chaque carré, en partant du carré central, on découpe dans l'image redimensionnée un carré de longueur correspondant à la longueur hypoténuse relative au cube sur lequel on projette notre portion d'image. Afin de tester ce premier algorithme, nous avons effectué nos différents tests avec la conditions que tous les cubes d'une même colonne sur la structure soient tous situés à distance égale du projecteur pour pouvoir considérer chaque ligne comme un lacet.

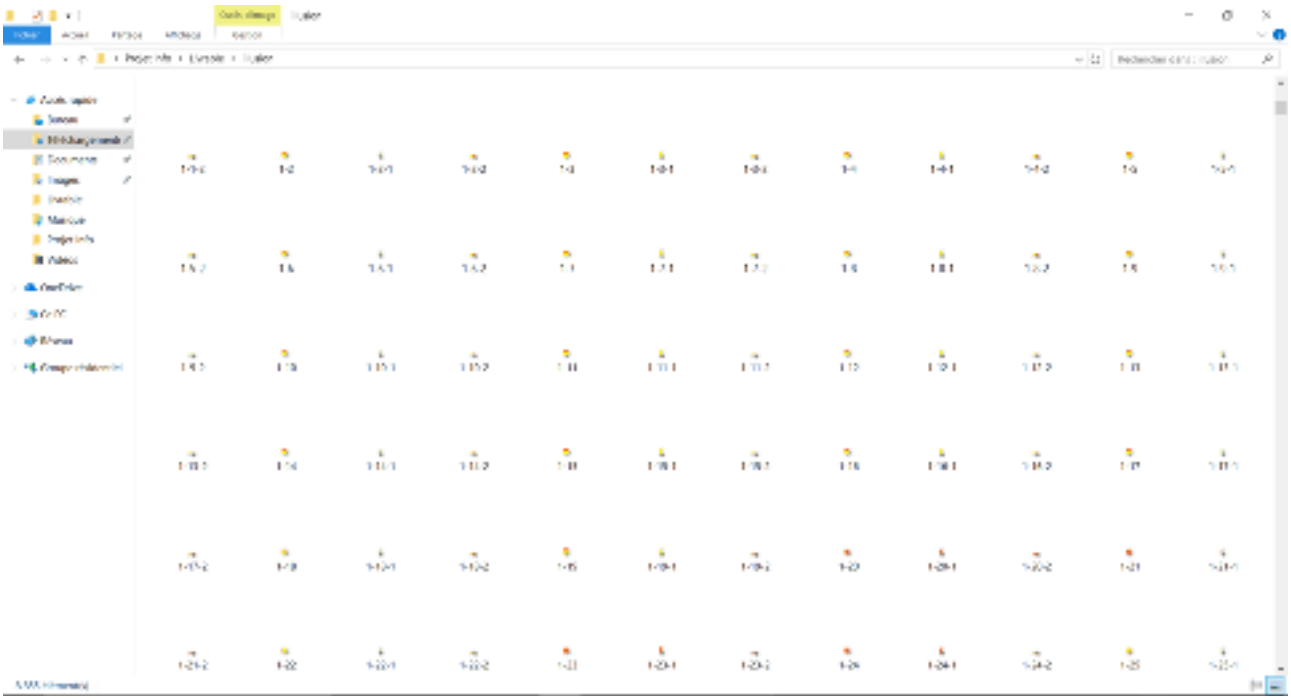
Dans le cas d'une structure 3D, le problème possède deux dimensions. Il faut utiliser le même raisonnement que dans le cas du problème à une dimension et le répéter suivant les axes x et y. Au lieu de couper des carrés de la longueurs de l'hypoténuse on va découper des rectangles dont la largeur est égale à l'hypoténuse suivant l'axe des x et la hauteur est égale à l'hypoténuse suivant l'axe des y.

Pour chaque face OpenGl perpendiculaires aux rayons lumineux du projecteur (les faces de devant) il est possible de définir une face au dessus, en dessous, à droite et à gauche de sorte à faire un cube. La structure OpenGL peut ainsi être définie sans trou. Pour définir l'image à coller sur une de ces faces perpendiculaires aux faces de devant, on considère les deux images adjacentes et on étend la dernière ligne ou colonne de pixel pour faire en sorte que les deux images se rejoignent. Cette technique produit des images de faible résolution mais cela n'est pas un problème car les faces perpendiculaires sont par hypothèse de faible largeur.

II - 2 Tests

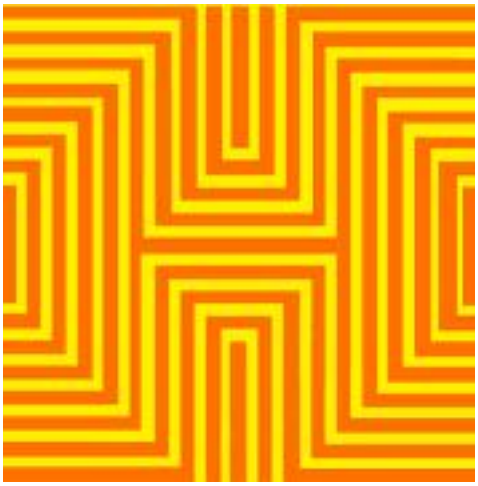
II - 2.1 Découpage de l'image

En exécutant la fonction permettant de découper l'image nous obtenons le résultat suivant :
Les différentes parties de l'image se trouvent dans un dossier.



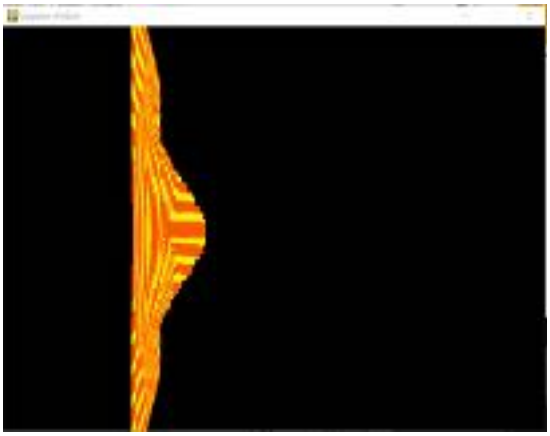
Les images suivantes montrent les résultats des tests de notre algorithme. Ils ont tous été réalisé avec les variables prenant les valeurs suivantes :
h=1, theta=45, le projecteur placé à (nbface/2,nbface/2,d0) et la liste lrd définie comme une gaussienne à deux dimensions (voir II - 1.1).

II - 2.2 Image témoin



Cette image est celle que nous avons utilisé pour la plupart de nos tests car elle présente des lignes horizontales et verticales, facilitant le repérage des éventuelles déformations.
Les tests suivants sont réalisés avec `nbface=35.`

Cette image est le résultat de la première version de notre algorithme qui ne réalisait qu'un simple mapping de l'image sur la structure 3D sans bien compenser les déformations dues aux reliefs.



Ces deux images montrent respectivement le résultat vu de face et de profil de la version finale de notre application. On remarque ici que les formes sont mieux respectées lors de la projection que dans la version précédente.
Néanmoins, on observe que certaines lignes sont toujours courbes. Ce phénomène est du aux approximations géométriques que nous avons faites lors du développement. Nous avons donc

procédé à d'autres tests avec des images plus complexes (des visages) afin de pouvoir mesurer l'influence des déformations sur des lignes déjà courbes.

II - 2.3 Tests avec des photos

Ici, les test ont été effectué pour nbface=25, on constate alors toujours une légère déformation qui n'est pas aussi flagrante qu'avec des formes géométriques.

La variable nbface n'a en réalité que peu d'influence sur la qualité de la projection mais il est nécessaire, pour modéliser une surface réelle (une façade de bâtiment par exemple) de pouvoir l'échantillonner avec suffisamment de précision, c'est à dire de choisir nbface grande. Cependant le temps de calcul devient très long au dessus de nbface=60.

III Bilan

Au début de ce projet nous avions mal apprécié les difficultés auxquelles nous avons dû faire face. En effet nous n'avions pas prévu de nous attaquer à l'étude du deep learning, qui entre dans une partie de notre projet qui nous a été suggérée par notre tuteur. Cette partie a cependant été très intéressante à étudier et avait un bon potentiel (du point de vue des carrés et des cubes) si on avait pu modéliser des façades. Le projet n'aurait eu que la partie deep learning nous aurions peut être réussi à mieux appréhender sa difficulté mais nous avons choisi après une période de plus se concentrer sur le noyau de notre projet : la projection d'image, tout en gardant en tête l'idée de pouvoir réétudier cet aspect problématique plus tard afin de pouvoir compléter le projet au mieux.
Concernant la partie de simulation de projection nous avons d'une part pu être assez autonome dans l'apprentissage de OpenGL et d'autre part notre tuteur nous a bien aidé et guidé en réfléchissant avec nous sur les aspects géométriques et les déformations des images.
Enfin nous ne nous sommes pas intéressés à la projection de vidéos car cela n'apportait pas d'intérêt du point de vue du développement car il suffit d'exécuter le programme sur une succession d'image pour créer le rendu d'une vidéo.

IV Manuel utilisateur

Pour faire tourner l'algorithme, il faut placer l'image que l'on souhaite projeter dans le repertoire courant. Il faut ensuite executer division.py puis animation.py, il faut noter qu'en fonction des versions de Python et/ou des bibliothèques, il est possible qu'il soit nécéssaire d'executer les scripts depuis un terminal. Une fois les deux scripts lancé, une fênetre PyGame s'ouvre montrant le résultat des de la projection. Il est possible de déplacer la caméras avec les touches i,j,k,l; il est possible de zoomer avec les touches b,h, ou la molette de la souries; et enfin il est possible de faire tourner la caméra avec les flèches directionnelles.
Pour modifier les grandeurs h, theta, nbface, la position du projecteur, il est nécéssaire de changer ces valeurs dans les deux scripts animation.py et division.py.
Lorsque vous lancez division.py, si dans le dossier courant se trouve déjà un dossier portant le nom de l'image à projeter, le programme va supprimer ce dossier et va créer un nouveau dossier avec les petites images à projeter sur les faces OpenGL.