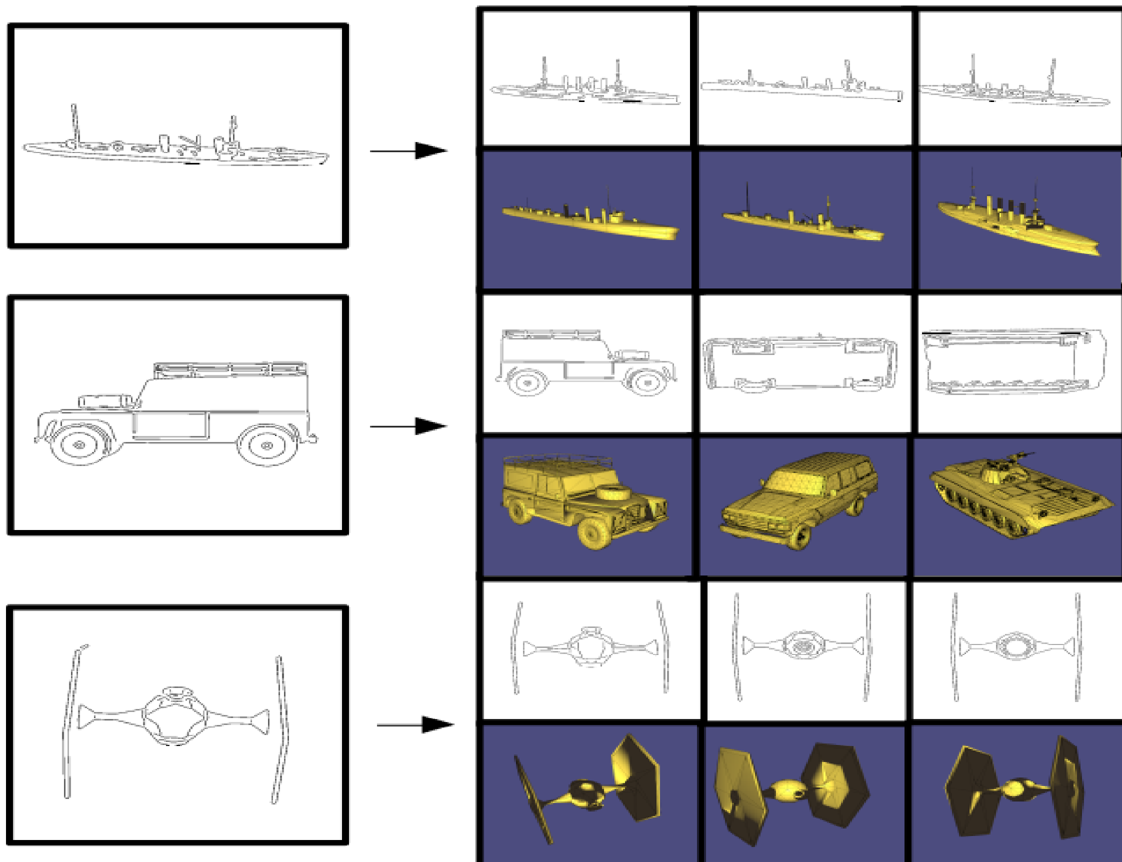
ÉCOLE POLYTECHNIQUE
IP PARIS

Tom Marty
Pierre Tessier

Venderdi 11 décembre 2020

PROJET INF574
SKETCH-BASED SHAPE RETRIEVAL

https ://github.com/PierreTsr/Shape_Retrieval



1 Présentation du projet

L'objectif de notre projet est de permettre une recherche efficace dans une banque de modèles 3D à partir de simples dessins faits à la main. Dans certaines situations, pour des très gros dataset, il arrive que la recherche de modèle 3D par mots-clefs soit laborieuse, tandis que nous avons toujours une image en tête bien précise de l'objet que l'on cherche. Un tel projet a pour but de rendre la recherche de modèles plus pertinente. Nous nous basons sur le papier *Sketch-Based Shape Retrieval* publié à la conférence SIGGRAPH 2012. Les auteurs de la publication originale ont fourni beaucoup d'efforts pour essayer de comprendre qu'elle était la représentation 2D qu'un humain aurait le plus tendance à faire d'un objet 3D, et leurs conclusions ont façonnées de nombreux choix d'implémentations dans le projet.

On peut distinguer 2 parties très différentes dans l'exécution de l'algorithme. Une partie asynchrone où l'on calcule de nombreuses informations sur notre base de modèles 3D. Une partie synchrone où l'image fournie par un utilisateur va être traitée pour trouver les meilleures correspondances dans la base de données. Le plus gros du travail est donc effectué en amont dans la première partie, et c'est donc sur cette portion que nous avons passé le plus de temps.

Le pipeline de la partie « offline »consiste, à partir du dataset de modèles 3D de l'université de Princeton, à générer de nombreux rendus différents pour chaque modèle. Ces rendus doivent être le plus proche possible de dessins humains pour que l'analyse soit pertinentes. Il faut aussi utiliser de nombreux angles de vue différents pour chaque modèle afin d'être capable de reconnaître un modèle quel que soit l'angle choisi par un utilisateur. Dans un premier temps nous nous sommes contentés de générer les 100 vues uniformément réparties sur la sphère unité. Les auteurs du papier de référence ont dans un second temps déterminés quelles étaient les vues privilégiées pour le dessin afin de limiter le nombres de rendu, mais leur résultats ne se sont pas avérés significatifs.

Ces dessins sont ensuite analysés à l'aide d'un filtre de *Gabor* afin de reconnaître des « features »caractéristiques sur chaque image. Une fois l'image analysée, on encode l'image comme un ensemble de $32 * 32$ sous-images, dont les centres sont uniformément répartis sur une grille. Pour s'assurer que notre algorithme est invariant en la taille de l'image, chaque patch recouvre une portion constante p de la surface totale de l'image. En pratique $p = 0.2$, ce qui veut dire que chaque patch recouvre 20% de l'image. Une attention toute particulière est portée aux patchs se trouvant sur les bords de l'image.

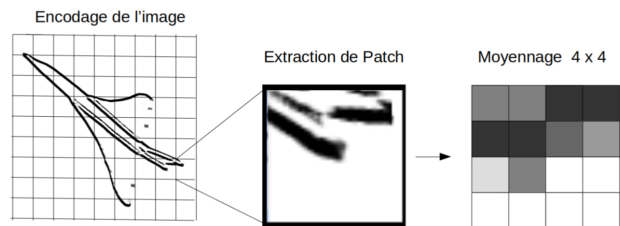


FIGURE 1 – Pipeline d'extraction des Features

Ces features sont moyennées et assemblées en vecteurs de dimension $4*4*k$, où k est le nombre de filtre de Gabor. Chaque image est donc représentée par un *Bag of Features* de 1024 features, soit une matrice réelle de dimension 1024×64.

On échantillonne ensuite 1 000 000 de ces features aléatoirement parmi le dataset afin de créer un vocabulaire de ces caractéristiques. Pour cela on utilise simplement un algorithme de clustering (*Lloyd-Max*) sur les échantillons, et les centres des clusters servent ensuite de mots dans notre vocabulaire. Lors de l'analyse d'une image, on calcule l'ensemble de ses features, puis on trouve le centre du cluster le plus proche pour chaque feature.

On calcule alors un histogramme des mots présents dans l'ensemble des patchs de chaque image. L'utilisation d'un vocabulaire local assure la bonne robustesse de l'algorithme. Cet histogramme forme alors une signature de l'image, et la proximité entre deux histogrammes est fortement corrélée à la proximité des dessins originaux. Une fois tous les histogrammes du dataset calculés il suffit alors à partir d'une nouvelle image de trouver les histogrammes les plus proches selon une certaine métrique pour trouver les modèles 3D susceptibles de correspondre à ce dessin.

Le pipeline de la partie « online »est alors évident. On demande à l'utilisateur de fournir un dessin du modèle voulu, on effectue l'analyse de

Gabor sur ce dessin afin de trouver les features de cette image. On trouve à quels clusters de notre vocabulaire appartiennent ces features ce qui nous donne l'histogramme de notre dessin et permet de faire une recherche dans notre base de donnée

2 Détails de notre implémentation

Nous allons détailler dans cette section essentiellement le pipeline de la partie « offline »car c'est là que se situe l'essentiel du travail. De nombreuses parties de ce pipeline n'étaient pas où peu détaillées et nous avons donc dû trouver des solutions nous-mêmes pour parvenir à nos fins. Nous ne nous attarderons pas trop sur les parties assez évidentes de notre code.

Nous avons choisi de coder ce projet en C++ afin de pouvoir exécuter les grandes quantités de calculs du projet efficacement et d'avoir plus de contrôle sur notre code, ce qui a rendu certaines parties plus difficile en nous privant de bibliothèques existantes en Python par exemple.

Cependant certaines parties de notre projet sont très indépendantes et demandaient des fonctionnalités assez spécifiques et nous avons alors choisi d'utiliser Python - notamment pour le rendu graphique ou pour le clustering.

2.1 Rendu graphique des modèles 3D

Les auteurs du papier original mentionnent plusieurs techniques pour obtenir un dessin à partir d'un modèle 3D ainsi que les résultats obtenus par chacune des méthodes. Ainsi la technique la plus efficace était le *suggestive contouring.* En cherchant les travaux de l'équipe citée sur ce point on trouve en effet une archive C++ permettant de visualiser en temps réels les contours d'un modèle 3D.

Notre première piste sur ce point consistait à réutiliser le projet de *suggestive contouring* cité par l'article. Cependant aucune bibliothèque C++ ou Python ne semblait implémenter cette technique, et nous avons donc essayé d'intégrer directement le code C++ original dans notre projet. Malheureusement entre les nombreuses dépendances obsolètes et notre faible expérience en options de compilation, cette approche s'est révélée désastreuse et très chronophage, et nous avons fini par l'abandonner.

Il nous a donc fallu trouver un autre moyen de générer tous ces dessins rapidement. Malheureusement nous n'avons trouvé aucune implémentation exploitable d'un générateur de dessins - que ce soit en *occluding contours* ou autre - et nous avons dû nous résoudre à implémenter nous-même cette partie.

Comme la génération des dessins se fait en amont, cette partie est très indépendante du reste du projet, et nous avons finalement décidé d'utiliser Python qui permettait de gérer un pipeline de rendu 3D de façon beaucoup plus ergonomique que C++. Après avoir beaucoup tâtonné, nous avons choisi d'utiliser *VTK* pour cette partie et nous nous sommes familiarisé avec la syntaxe pour créer notre pipeline, qui obtient un résultat un peu différent de ce que les auteurs avaient.

En effet, afin de générer le plus facilement possible les 185 130 dessins du dataset, nous avons envisagé cette approche : faire un rendu basique du modèle 3D, avec un éclairage de face sur fond noir, en spécifiant la position de la caméra et en s'assurant de voir l'objet entier ; puis passer l'image ainsi obtenue dans un filtre de *Canny* (dont l'implémentation sur *VTK* était disponible en ligne). Ainsi le dessin obtenu est composé uniquement des lignes apparaissant naturellement lors d'un rendu du maillage.

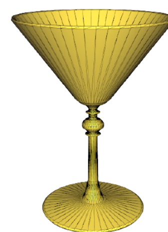


Figure 2 – Rendu 2D sans traitement

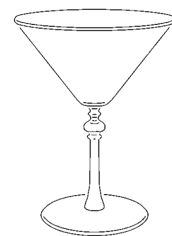


Figure 3 – Rendu 2D après filtre de Canny

Cette approche nous a semblé la plus pertinente, car nous disposions de peu de temps pour générer les images et nous familiariser avec VTK, et elle avait l'avantage d'être simple à implémenter et de proposer des résultats corrects en un temps convenable. En effet, il nous a fallu près de 3 heures de calculs sur GPU pour générer les 102 vues par modèles préconisées par les auteurs.

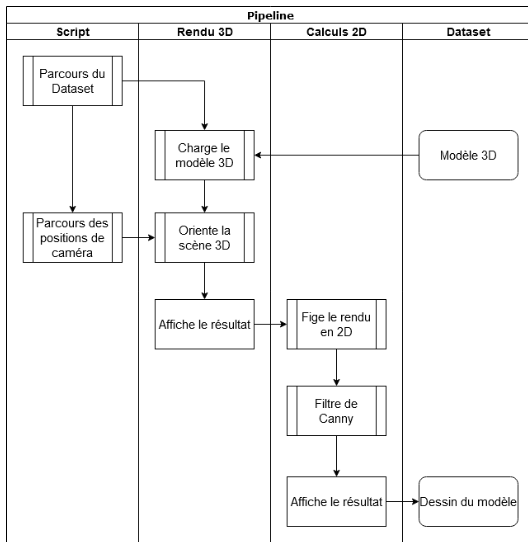


FIGURE 4 – Pipeline de rendu des modèles 3D

2.2 Convolution par un filtre de Gabor

La première étape importante du pipeline ensuite, est la génération des features à partir des rendus stylisés des modèles 3D. Afin d'identifier les caractéristiques pertinentes des images, on utilise k filtres de *Gabor* avec des orientations différentes. $(k = 4)$ $\theta \in \left\{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}\right\}$. Un filtre de *Gabor* est la convolution d'un noyau gaussien de la transformée de Fourier d'une image. Il correspond dans le *domaine spatial* à un noyau sinusoïdal modulée par un fonction gaussienne. Pour certains chercheurs, le filtre de *Gabor* traite l'image d'une manière similaire au système de traitement de l'image du monde vivant, ce qui explique pourquoi les filtres de *Gabor* sont aujourd'hui largement répandus en analyse d'image.

Il est défini dans le *domaine spatial* par la fonction :

$$g(x, y) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \cos\left(2\pi\frac{x'}{\lambda} + \psi\right)$$

On applique un noyau de convolution de taille [11,11] sur l'ensemble de l'image.

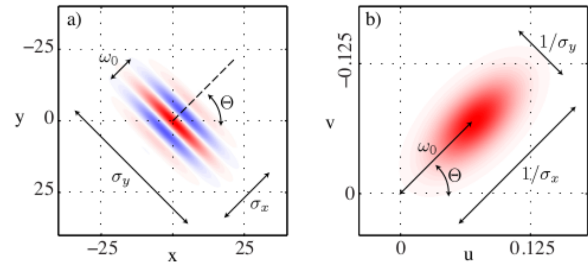


FIGURE 5 – Réprésentation du filtre de Gabor dans les domaines spatial et fréquentiel *Tiré de la publication originale*

λ représente la longueur d'onde des features que l'on souhaite extraire (ie. l'épaisseur des traits), x' et y' sont les coordonnées spatiales d'un pixel dans le repère local orienté d'un angle θ, σ_x et σ_y sont les écarts type de la gaussienne dans le repère local, et ϕ est le déphasage spatial de la modulation sinusoïdale.

Nous avons dans un premier temps utilisé les paramètres utilisés dans le papier de recherche : $\lambda = 0.3$, $\sigma_x = 12$, $\sigma_y = 40$ et $\phi = \frac{\pi}{2}$.

Nous avions tenté initialement de passer l'image dans le *domaine fréquentiel* et de lui appliquer un noyau de convolution gaussien, comme le conseillaient les auteurs. Les résultats n'ont cependant pas été concluant et cette approche s'est révélée très laborieuse. Nous avons donc décidé de rester dans le domaine spatial et d'établir la relation entre les paramètres donnés par le papier de recherche exprimés dans le *domaine fréquentiel* et les paramètres utilisés par la librairie *Opencv* exprimés dans le domaine spatial.

Le filtre est réglé spécifiquement pour notre générateur d'image 2D, et devra être réglé à nouveau si l'on décidait de changer la technique de rendu stylisé. Les résultats sont présentés dans la section suivante.

2.3 Calcul d'une feature

Une fois l'image filtrée par l'un des k filtres de *Gabor*, on effectue un moyennage local sur $n * n$ sous-portions du patch. Chaque feature est un vector de taille $n*n*k$, où n est la taille de la matrice de moyennage et k le nombre de filtres de *Gabor*. On obtient en pratique un vecteur de dimension 64.

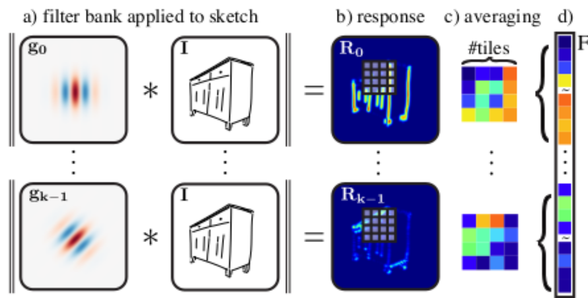


FIGURE 6 – Pipeline d'extraction des features : la réponse moyennée au sein d'une cellule des k filtres gabor forme une feature F *Tiré de la publication originale*

2.4 Calcul du vocabulaire et histogrammes

L'algorithme « offline »créé un vocabulaire de features en identifiant des clusters dans l'ensemble des features des images générées. Ce vocabulaire permet ensuite de générer un histogramme pour chaque image, contenant la fréquence d'apparition de chaque classe associée à chacune des 1024 features de chaque image.

Le vocabulaire est obtenu par clustering d'un échantillon aléatoire de 1 000 000 de features du dataset. Générer cet échantillon à lui seul nécessite près de 3 heures de calculs, car il faut à chaque fois appliquer un filtre de *Gabor* à une image.

On utilise ensuite la librairie Python *scikit.learn* pour le calcul des centroïdes des 2500 clusters. Cela nous donne un vocabulaire de 2500 "mots visuels", qui sera utilisé pour la génération des histogrammes.

La phase de calcul des clusters est particulièrement longue et nos premières tentatives en C++ se sont révélées infructueuses même sur des échantillons beaucoup plus petits. *Scikit* a pris 40 minutes pour le clustering de 100 000 échantillons avec un algorithme KMeans standard et nous avons été obligé d'utiliser un algorithme de KMeans par batchs pour trouver les clusters parmi 1 000 000 d'échantillons en un temps et avec une utilisation de la mémoire raisonnables.

Pour calculer les histogrammes, on applique à la lettre les recommandations des auteurs. On commence par calculer un histogramme classique à partir du clustering précédent et des features de l'image. Puis on applique la transformation suggérée qui vise à pondérer le poids d'une composante de l'histogramme par l'inverse de sa fréquence dans le dataset. Cette étape est assez dirigée et ne pose pas de problème majeur.

La seule subtilité réside dans la structure utilisée pour implémenter un histogramme. Il s'agit d'un vecteur réel dont la dimension est égale au nombre de mots du vocabulaire. Seulement ce vecteur est rempli en grande majorité de 0. Nous avons donc décidé de les représenter par des dictionnaires prenant en clé les indices des composantes non-nulles, et en valeur la masse de cette composante dans l'histogramme.

2.5 Interface graphique de dessin

Nous avons implémenté une interface de dessin très simple permettant à l'utilisateur de dessiner sur une image de $600 * 600$ pixels. L'interface est implémenté à l'aide d'un petit script Python basé sur la librairie *Opencv*. L'image est ensuite enregistré dans le dossier de travail.

2.6 Recherche des instances les plus proches

Pour retrouver les instances les plus proches d'une image donnée, les auteurs suggèrent l'utilisation d'un index inversé, qui permet de retrouver tous les histogrammes partageant une feature donnée. Il faut ensuite faire l'union de tous les histogrammes ainsi retrouvés et calculer le plus proche pour la mesure implémentée précédemment.

Pour cette partie nous avons essayé plusieurs approches. Nous avons d'abord reproduit le processus des auteurs, en écrivant les histogrammes de chaque vue dans un fichier et en utilisant un index inversé.

Puis face à de gros problèmes de performances nous avons essayé de stocker les histogrammes de toutes les vues d'un même modèle dans un fichier unique, afin de limiter les ouvertures de fichiers. Nous avons également essayé de nous passer de l'index inversé.

Enfin pour la recherche nous avons essayé de lire les histogrammes uniquement lorsque nécessaire dans un premier temps, puis de tous les charger dans la RAM en amont de la phase de recherche.

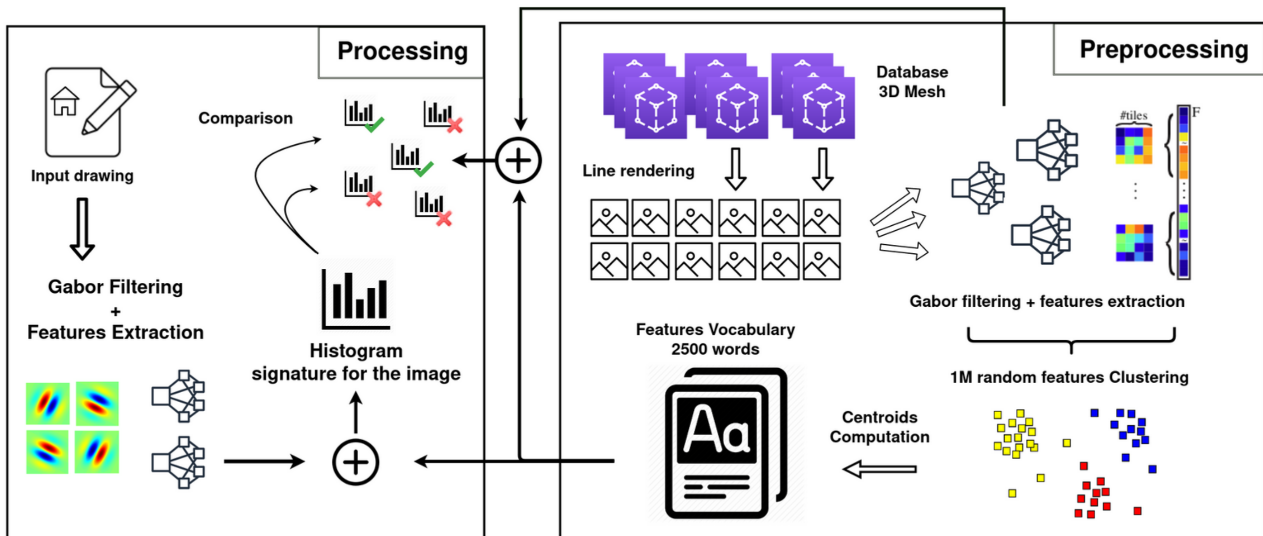


FIGURE 7 – Pipeline complet de recherche de meshs correspondants

3 Résultats

3.1 Résultats du rendu

Nous avons travaillé sur le même dataset que celui utilisé par le papier de recherche composé de 1814 meshs 3D. Pour chaque mesh, on calcule les rendus 2D des 102 vues de l'objet après application du filtre de *Canny*. La plus grande partie de ce travail consistait à déterminer les constantes optimales permettant de se rapprocher le plus possible d'un dessin issu de la main de l'homme. Il a donc fallut trouver le bon compromis sur les paramètres pour à la fois éviter une surabondance de détails non pertinents et conserver les composantes essentielles au dessin.

Le calcul total des rendus stylisés a duré environ 3 heures.

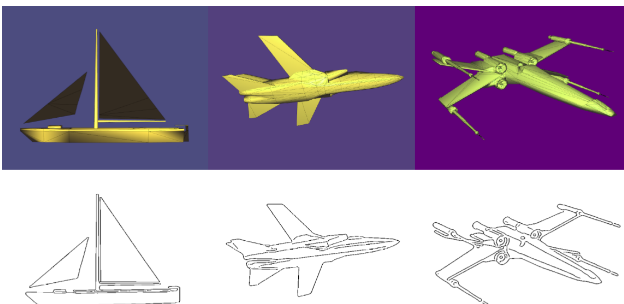


FIGURE 8 – Exemple de rendu stylisé pour différents modèle 3D issus du dataset

Toutefois, il semblerait que malgré nos efforts ces rendus soient trop détaillés pour être comparés aux dessins humains, et notre approche fonctionne probablement un peu moins bien que les *suggestive contours*.

3.2 Résultats sur les filtres de Gabor

Une fois les images générées, elles subissent un traitement par plusieurs filtres de *Gabor*.

La majeure partie du travail sur les filtres de *Gabor* fut sur le réglage des différents paramètres du filtre présenté plus haut. Les paramètres donnés dans le papier n'étaient pas adaptés à nos rendus 2D à cause entre autres de la différence d'épaisseur des arrêtes entre nos rendus et les leurs.

In fine, le filtre est capable d'éliminer les arrêtes qui sont orientées suivant l'angle θ du filtre. Ci dessous, on donne en entrée, une image contenant plusieurs arrêtes dans les directions qui nous intéressent : $\theta \in \left\{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}\right\}$.

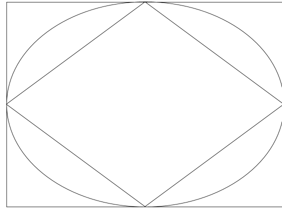


FIGURE 9 – Image d'entrée

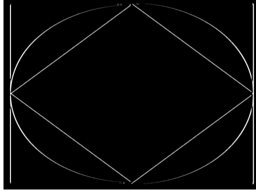


FIGURE 10 – $\theta = 90°$

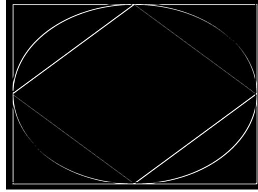


FIGURE 11 – $\theta = 45°$

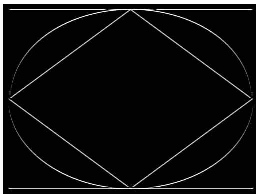


FIGURE 12 – $\theta = 0°$

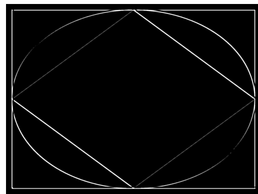


FIGURE 13 – $\theta = 135°$

3.3 Résultat de la phase de recherche

Le premier constat incontournable de toutes nos approches, c'est que ce projet nécessite une puissance de calcul colossale et énormément de données doivent être stockées. Ainsi nous n'avons même pas pu envisager calculer tous les histogrammes des images de notre dataset : en exploitant en parallèle les 16 coeurs de l'ordinateur que nous avons utilisé il nous faut 40 minutes pour générer les histogrammes de 10 000 images. Il nous faudrait donc une demi-journée à ce rythme pour écrire tous les histogrammes. C'est pourquoi nous nous sommes limités dans nos tests à une recherche sur 100 modèles 3D soit, 10 000 images.

Sur ces tests, nous nous sommes rendus compte que l'idée d'utiliser un index inversé n'apportait en fait rien en terme de performance et était même contre-productive. En effet, nos histogrammes comptent en moyennent 400 composantes sur un vocabulaire de 2 500 mots et sur tout nos tests, 100% des histogrammes partageaient au moins une composante et devaient donc être comparés. Nous avons donc rapidement décidé de laisser tomber l'index inversé tel que proposé par le papier de recherche et de comparer un histogramme contre tous ceux du dataset.

Cette approche est relativement efficace à condition de charger tous les histogrammes dans la RAM en amont, ce qui était tout à fait possible vu que nous n'exploitions que 6% du dataset. On note néanmoins qu'il est très important de ne pas répartir les histogrammes dans trop de fichiers pour ne pas trop ralentir cette partie.

Cependant si cette approche peut encore fonctionner avec le dataset complet (il suffirait de réserver 1Gb de RAM aux histogrammes d'après nos calculs), il est évident qu'il sera beaucoup plus difficile de la faire passer à l'échelle avec une plus grande banque de modèles.

3.4 Résultats du pipeline complet

Une fois le dessin soumis, notre algorithme renvoi les modèles 3D associés possédant un histogramme faisant partie des n plus proches voisins de notre requête.

Ci-dessous, notre algorithme nous renvoie les 5 histogrammes les plus proches de celui généré par notre requête :

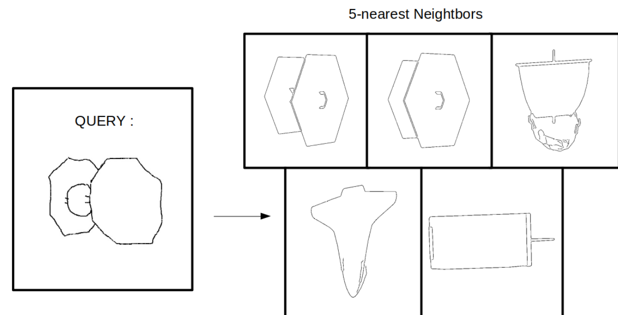


FIGURE 14 – Exemple de résultats pour un dessin de chasseur TIE

Sur des exemples simples comme celui-ci, l'algorithme fonctionne relativement correctement. La cinquième image est un rendu de "tank" vu de dessous. La surface carré peut rappeler la face latérale du chasseur. Il semblerait donc que le style du dessin ait une influence significative dans la recherche.

Ci-dessous un des meilleurs résultats que nous ayons pu obtenir, ici les images générées ont un style se rapprochant beaucoup de notre dessin :

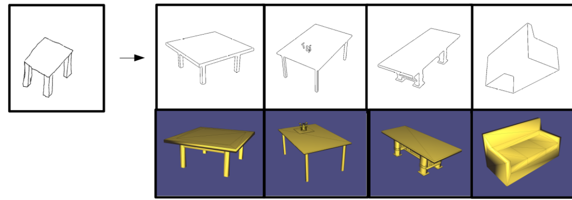


FIGURE 15 – Résultat pour une requête d'un dessin de table

Nous avons cependant remarqué, que, contrairement à ce qu'affirmaient les auteurs, cette approche est très sensible à la quantité de détails dans un dessin. Ainsi esquisser grossièrement une silhouette ne nous a jamais permis de retrouver un objet complexe, probablement car nos rendus étaient trop détaillés.

Nous avons alors décidé de tester le fonctionnement de l'algorithme avec des images générées de la même façon, afin qu'elles aient des styles et une quantité de détails similaires. Les résultats de ces tests sont un peu plus probants.

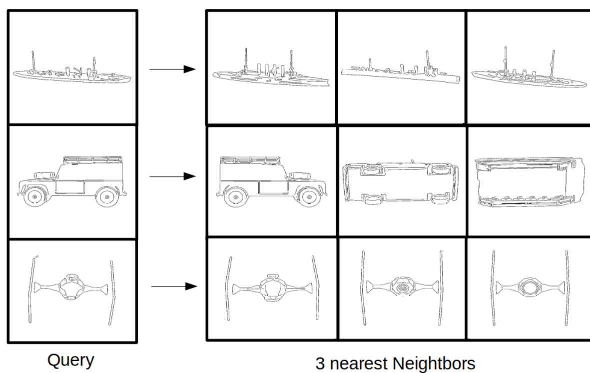


FIGURE 16 – recherche des 3 plus proches voisins d'un dessin issu du dataset

On remarque cette fois-ci que les voisins correspondent d'avantage graphiquement à la requête. Une prochaine étape serait donc de trouver une technique de rendu 2D plus proche de ce qu'un humain a tendance à dessiner, qui ne garde que les détails principaux.

Un autre facteur que nous avons remarqué est l'importance de l'échelle de l'image : une image à laquelle on ajoute des marges blanches va obtenir un histogramme très différent de l'image originale. De même, bien que nous n'ayons pas testé ce point en particulier, les déformations solides en générales semblent impacter fortement la performance de l'algorithme. Ces points ont probablement été résolus par les auteurs par des étapes de prétraitement des images qui ne sont pas mentionnées dans leur publication.

4 Conclusion

En conclusion, ce projet fut particulièrement intéressant car à la croisée de plusieurs sujets abordés par le cours d'INF 574. Nous avons ainsi travaillé avec du rendu stylisé 2D, des filtres de *Canny*, de la convolution, du clustering, et de la gestion de base de données. Nos résultats sont prometteurs. Un axe d'amélioration possible serait de rendre notre algorithme plus robuste au différence d'échelle et d'orientation des dessins. Il serait également possible de travailler sur les performances du préprocessing, qui se sont révélées être particulièrement limitantes dans la phase d'implémentation. Enfin, nos résultats sont très fortement impactés par la quantité de détails des rendus 2D, ce qui laisse supposer que nos choix pour créer nos dessins n'étaient pas optimaux.

De plus, avec les résultats actuels des techniques d'apprentissage profond, construire manuellement un tel processus pour associer un vecteur à une image semble particulièrement complexe, lorsqu'un réseau convolutif parviendrait probablement à un meilleur résultat plus rapidement que nos histogrammes.