Exploitation de relevés Scanner 3D dans un SIG

L'exemple des Transports publics genevois

Joris Vidalenche,
ESGT/HKG Géomatique.

La Suisse (ou officiellement, la Confédération suisse) est organisée en vingt-six cantons, possédant chacun ses propres lois. Chaque canton est ensuite subdivisé en communes. Situé à l'extrémité ouest du lac Léman, le canton de Genève regroupe quarante-cinq communes. Il est l'un des plus peuplés (460 534 habitants en 2011 selon l'Office Fédéral Suisse de la Statistique), malgré le fait qu'il soit également l'un des plus petits avec ses 280 kilomètres carrés environ. La ville de Genève et ses alentours attirent avec force les travailleurs du canton de Vaud, au Nord-Est, mais également les travailleurs français de l'Ain et de la Haute-Savoie.

Les TPG (Transports publics genevois) souhaitant disposer d'une base de données afin de gérer les éléments aériens des lignes de tramway et de trolleybus de Genève, la société *HKD Géomatique* a été mandatée pour réaliser le levé des voies du canton traversées par ces lignes et mettre en place une base de données sous forme de SIG (système d'information géographique) 3D. Le bureau a réalisé en janvier 2013 le relevé de soixante-dix kilomètres de corps de rues par MLS (*mobile laser scanning*), puis a modélisé les lignes aériennes des TPG sous forme d'entités ponctuelles (aiguillages, antennes, boîtiers, contrepoids, croisements, feux d'aiguillages, isolateurs, isolateurs de haubans, panneaux, parafoudres, scellements et sectionneurs) et linéaires (câbles électriques, consoles, *feeders*, haubans, lignes de contact, pinces de courbe, protections, suspensions et tubes).

La base de données saisie consiste en la modélisation des éléments aériens en 3D au moyen du logiciel *Trimble Trident-3D Analyst*, en utilisant comme support le nuage de points et les photographies acquis par la mission de MLS. Cette étape va ainsi poser une première problématique : l'accrochage des éléments se faisant sur le nuage de points et non sur les entités elles-mêmes, les erreurs de topologie sont inévitables.

Par ailleurs, la symbologie utilisée lors de cette modélisation reste basique, et n'est pas adaptée à l'exploitation de la base de données en tant que SIG, ce qui constitue une deuxième problématique : concevoir une symbologie 3D en adéquation avec l'utilisation que l'on fera de cette *Géodatabase*.

Par la suite, ce SIG sera finalement livré aux TPG, ce qui soulève de nouvelles interrogations. Premièrement, ceux-ci ayant l'habitude d'utiliser des logiciels de CAO, ils ont besoin que le passage du monde du SIG vers celui de la DAO soit proposé, sous la forme d'un outil d'export. Toujours dans les outils d'export, il leur serait également utile d'avoir à disposition la possibilité de créer des vues en PDF. Il conviendra donc d'intégrer ces deux outils dans notre SIG.

Enfin, la base de données livrée devra être régulièrement mise à jour, au fur et à mesure de l'entretien du réseau. Cependant, cette tâche revient aux techniciens des TPG, peu rompus à

cette pratique. Il est donc indispensable de leur proposer un outil simple pour la réaliser.

Création d'un outil de contrôle topologique en 3D

En SIG, le respect de la topologie est un critère de qualité majeur. En effet, un SIG a pour vocation de renseigner l'utilisateur sur les caractéristiques intrinsèques de l'objet étudié (attributs, position absolue), mais aussi sur les relations qui existent avec son voisinage. Il est donc important de contrôler, et de corriger, les relations topologiques entre les diverses entités du produit. Dans notre cas, il faudra s'assurer, par exemple, que chaque panneau soit bien suspendu à un hauban et ne « *lévite* » pas. Cependant, il faut bien comprendre que détecter et corriger ces erreurs manuellement, sur les soixante-dix kilomètres de lignes relevées, représenterait un travail colossal, d'autant plus qu'il est à effectuer dans les trois dimensions : il convient donc d'automatiser le processus. Pour donner un aperçu de la quantité d'informations présente dans notre base de données, les tableaux 1 et 2 récapitulent quelques statistiques issues de celle-ci.

Éléments ponctuels	*Nombre*	*Pourcentage*
Isolateurs de haubans	20 824	54,06
Isolateurs	13 384	34,75
Scellements	2 550	6,62
Panneaux	810	2,10
Croisements	311	0,81
Sectionneurs	210	0,55
Aiguillages	145	0,38
Contrepoids	93	0,24
Feux	74	0,19
Boitiers	55	0,14
Parafoudres	33	0,09
Antennes	28	0,07
Total	**38 517**	**100,00**

Tableau 1 :
Statistiques - éléments ponctuels

Éléments linéaires	*Longueur (km)*	*Pourcentage*
Lignes de contact	207,318	48,42
Haubans	152,095	35,52
Feeders	35,366	8,26
Câbles électriques	10,628	2,48
Tubes	9,157	2,14
Suspensions	5,382	1,26
Pinces de courbes	5,121	1,20
Consoles	2,751	0,64
Protections	0,363	0,08
Total câbles	**410,790**	**95,94**
Total autres	**17,392**	**4,06**
Total	**428,182**	**100,00**

Tableau 2 :
Statistiques - éléments linéaires

État de l'art

Avant de se lancer dans cette automatisation, il faut déjà se pencher sur les outils existants pour étudier les relations topologiques entre deux entités.

La base de données sera traitée sous *ArcGIS*. On peut donc se renseigner sur les solutions déjà proposées par *Esri*. Comme l'a remarqué Bernard Lachance (2005), *ArcGIS* ne dispose que d'outils de topologie en 2D. Selon lui, bien que ces outils restent fonctionnels dans un environnement 3D, ils travaillent toujours en 2D, comme s'ils plaquaient les entités sur le plan (O, x, y) avant de les traiter.

Gaëtan Givord, ingénieur chez *HKD Géomatique*, a déjà proposé une piste de recherche : elle consiste à utiliser ces outils 2D sur la base de données telles-quelles, puis de leur faire subir une rotation autour des abscisses (ou des ordonnées) de manière à réutiliser les outils de contrôle topologique 2D dans un plan vertical (et enfin réaliser la rotation inverse pour revenir dans le système d'origine). Cette solution est certes efficace, mais oblige à transformer l'ensemble des données, même celles qui ne devront subir aucune correction ; elle est donc lourde à mettre en œuvre, et peut présenter des risques pour le positionnement des objets suite aux transformations effectuées (chaque rotation constitue un nouveau calcul et peut entraîner une erreur d'arrondi). Il est donc préférable de réaliser les contrôles directement dans un espace en trois dimensions.

Hormis cette première approche, il semble *a priori* que de tels outils n'existent pas (y compris dans d'autres logiciels), les SIG étant traditionnellement planimétriques : ils devront donc être entièrement développés. À noter qu'ils seront nécessairement simples : la base ne contient que des entités ponctuelles et linéaires, et les relations topologiques seront uniquement du type « *se touchent* » / « *ne se touchent pas* ».

Étant arrivé à des conclusions similaires, Florian Gandor, lors d'un stage chez *HKD Géomatique*, a cherché à résoudre le problème via le langage *Python* et le module *pyshp*, permettant de traiter des *Shapefiles*. Pour ce faire, il a développé un *script* qui procède par comparaison de coordonnées : pour deux entités données, l'une est définie comme fixe, tandis que l'autre est considérée comme mobile ; la distance qui les sépare est alors calculée à partir de leurs coordonnées. En dessous d'un seuil de tolérance défini par l'utilisateur, l'entité mobile est corrigée. Le *script* procède ainsi pour chaque couple d'entités devant être topologiquement cohérentes.

Sur la figure 1, l'entité à corriger est la polyligne rouge. Pour le premier nœud de cette dernière, le *script* cherche un point fixe à proximité (en bleu) et corrige la polyligne. Il procède ensuite de même avec le point suivant.

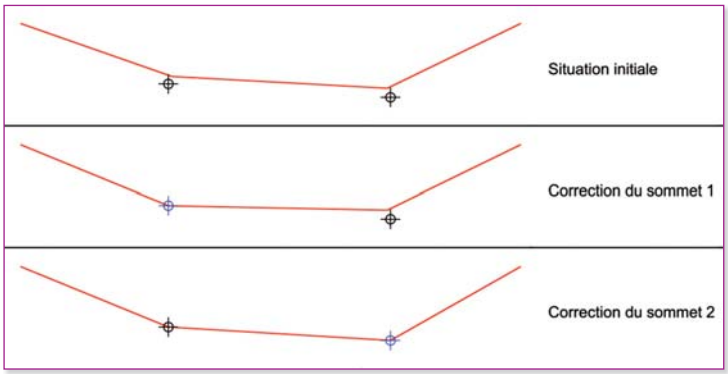

Figure 1 : Fonctionnement du script de Florian Gandor.

Réalisation

Ce premier *script* propose un fonctionnement à la fois simple et efficace. Cependant, il possède quelques défauts qui doivent être corrigés pour un traitement de qualité. Ces premiers ajouts constituent des modifications mineures facilement implémentées dans le code d'origine.

Pour commencer, la solution initiale traite les différents couples dans un ordre arbitraire. Or, nous avons dit que lors de chaque correction, une entité est considérée comme fixe tandis que l'autre est définie comme mobile. Il faut donc que l'entité fixe, qui sert de référence, n'ait pas à être corrigée par la suite, d'où l'importance de déterminer un ordre précis pour effectuer les corrections.

Voici la méthodologie suivie pour déterminer cet ordre :

- La liste exhaustive des couples à contrôler est consignée dans un tableau à trois colonnes : rang (vide pour l'instant), entité fixe et entité mobile. Les entités sont colorées selon qu'elles interviennent comme toujours fixes (en vert), toujours mobiles (en bleu), ou les deux suivant le contrôle considéré (en rouge) ;
- Le tableau est ensuite trié pour regrouper les contrôles ayant la même entité mobile. Si, pour un de ces groupes, les éléments de la colonne « *entité fixe* » sont tous verts, le contrôle peut être effectué. On place le groupe en tête du tableau et on y associe le rang 1. Les entités mobiles correspondantes passent alors en vert dans les deux colonnes ;
- L'étape précédente est alors répétée pour le rang 2, puis 3, etc. jusqu'à ce que toutes les entités soient colorées en vert.

L'outil développé par Florian Gandor n'était pas utilisable directement, mais devait être exécuté depuis la suite logicielle *Anaconda,* qui est un environnement de développement *Python* ; de plus, l'exécution s'effectuait en ligne de commande. On a donc à la fois procédé à la création d'une interface ainsi qu'à la « *compilation* » de l'application.

L'interface rend la saisie des paramètres de traitement plus rapide et plus intuitive ; de plus, elle présente également l'avantage de regrouper plusieurs outils réalisés par la suite. Elle a été entièrement codée avec le module *TkInter* présent dans la distribution de *Python* 2.7.

Concernant la « *compilation* », elle permet non seulement à l'utilisateur de travailler sans accéder au code source, et sécurise ainsi ce dernier contre toute modification involontaire, mais également d'utiliser l'application depuis n'importe quel poste informatique, de manière autonome. À noter que le terme « *compilation* » est à mettre ici entre guillemets. En effet, *Python* est un langage interprété et non compilé (l'utilisation d'un script *Python* nécessite un interpréteur qui « *lit* » les lignes de code et les exécute à la volée). De ce fait, le compilateur utilisé (*py2exe* dans notre cas) encapsule dans un exécutable le code source et un interpréteur *Python*, sans réaliser une véritable compilation (à aucun moment le code source n'est traduit en langage machine). De plus, l'ensemble des bibliothèques nécessaires à l'exécution du script est rassemblé dans le même dossier que cet exécutable. Ainsi, on obtient une application qui n'est plus dépendante d'une installation particulière. Pour pouvoir la faire fonctionner, il suffit de copier l'ensemble du dossier contenant l'exécutable sur le poste de l'opérateur.

On remarquera qu'un fichier *.pyc* est généré au moment de l'exécution d'un script *Python*. Il correspond à une version semi-compilée (*bytecoded*) de celui-ci, dont l'exécution est plus rapide. Il est dit « *semi-compilé* » car c'est un code intermédiaire entre code source et langage machine. Sa distribution aurait permis de résoudre la contrainte de protection du code (celui-ci n'est alors plus compréhensible par l'utilisateur et le rebute à effectuer des modifications), mais nécessite toujours une installation de *Python* sur le poste utilisé. Cette solution n'aurait donc pas entièrement rempli l'objectif fixé.

Modifications majeures du script

Les modifications qui suivent représentent des améliorations plus conséquentes, portant sur le fonctionnement lui-même du

script initial. Pour plus de clarté, un nouveau script a été réalisé, s'inspirant du premier mais implémentant ces améliorations dès sa conception.

La première correction majeure du *script* s'est portée sur l'implémentation de ce que nous appellerons la notion de « *plus proche* ». C'est-à-dire que, dans le cas où plusieurs corrections sont possibles, le *script* devra choisir d'effectuer celle qui se rapporte à l'entité la plus proche. Le *script* d'origine ne prend pas en compte cette notion, et corrige l'entité mobile par rapport à chaque objet fixe intervenant dans le contrôle (à condition bien sûr que l'écart entre les deux soit sous le seuil de tolérance). Ainsi, à la fin du contrôle, c'est la dernière correction qui l'emporte. Cette manière de fonctionner entraîne des erreurs de correction.

La figure 2 présente les conséquences de cette modification : les images du haut représentent le fonctionnement du script d'origine, tandis que celles du bas correspondent au script révisé. À l'itération 1, la correction va être calculée par rapport au premier point codé dans le *Shapefile*. À la suivante, ce sera le deuxième point.

L'ajout de cette notion implique une modification importante du fonctionnement du *script*. En effet, plutôt que de corriger l'entité chaque fois que le seuil de tolérance est respecté, il est nécessaire de stocker la correction sans l'appliquer, ainsi que l'écart entre les deux entités, de sorte que, chaque fois qu'un contrôle est positif (c'est-à-dire qu'une entité doit être corrigée), les écarts sont comparés et les coordonnées corrigées les plus proches de celles initiales sont conservées. L'entité mobile est ensuite corrigée une fois que les contrôles avec toutes les entités fixes concernées ont été effectués.

Après concertation avec l'équipe de saisie, il était apparu que les objets ponctuels étaient plus facilement modélisables que les entités linéaires et, de ce fait, qu'ils étaient saisis avec plus de fiabilité. Florian Gandor avait donc conçu son *script* pour considérer tous les ponctuels comme précis, et ne faire porter les corrections que sur les linéaires. Ainsi, il avait défini deux types de contrôles : le test d'un sommet (extrémité ou sommet intermédiaire) de linéaire par rapport à un point, et le test d'un sommet de linéaire par rapport au sommet d'un autre linéaire.

Cependant, ces deux types de contrôles ne permettent pas de corriger les cas où un sommet de linéaire n'a pas été inséré à proximité de l'entité fixe (dans ce cas, il s'agit d'une faute de modélisation). Il a donc fallu rajouter ces nouveaux types au *script*.

L'algorithme fonctionne comme suit :

- La distance entre le point ou le sommet fixe et le linéaire mobile est calculée par projection orthogonale ;
- Cette distance est contrôlée pour respecter le seuil de tolérance et la notion de plus proche ;
- Lorsque les deux étapes précédentes ont été réalisées pour tous les points ou sommets candidats pour la correction, un sommet est inséré dans la portion du linéaire concerné, à l'emplacement de l'objet fixe retenu.

À l'utilisation, il s'est avéré que cet ajout réalisait bien la fonction attendue, mais générait également un nombre important de « *fausses corrections* ». En effet, il est courant que certains éléments linéaires passent à proximité d'entités ponctuelles sans

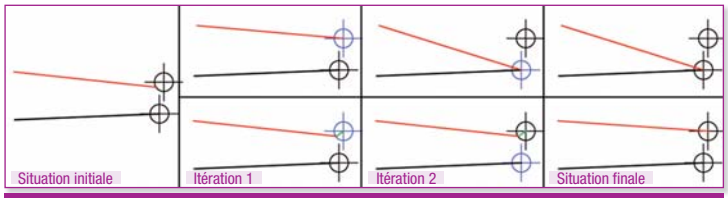

Figure 2 : Notion de « plus proche ».

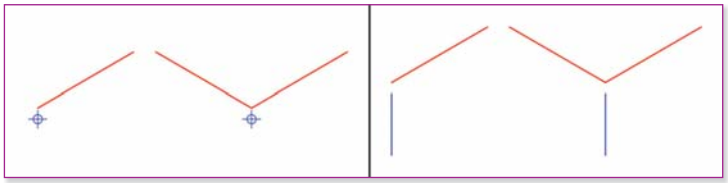
Figure 3 : Les deux types de contrôles implémentés par Florian Gandor.

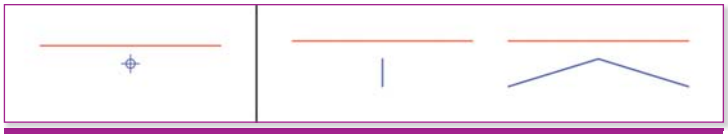
Figure 4 : Nouveaux types de contrôles implémentés.

y être attachés dans la réalité. L'application, qui ne peut faire la différence, va réaliser une modification qui n'a pas lieu d'être. La figure 5 est extraite d'*ArcScene*. Le carré noir signale une potentielle erreur : d'après notre outil, le hauban (ligne rouge) devrait être en contact avec l'isolateur (point bleu). C'est une fausse détection : en réalité, cet isolateur attache le *feeder* (ligne jaune) à la console (ligne verte).

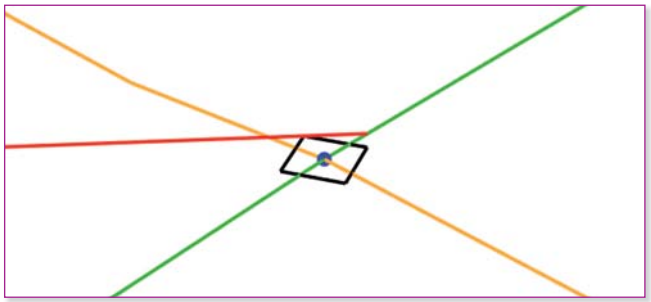

Figure 5 : Fausse détection visualisée dans ArcScene.

Face à ce nouveau problème, la solution trouvée a été de réduire cette fonction à une simple détection, sans correction automatique. Les zones positives à cette détection sont consignées dans un *Shapefile* qui permet ensuite à l'opérateur de visualiser rapidement l'emplacement d'éventuelles erreurs et de les corriger manuellement.

Corrections par rapport au bâti et aux mâts TPG

Outre les corrections topologiques internes à la base de données, il faut savoir que le produit doit être cohérent avec l'ensemble des mâts TPG et bâti 3D présents dans le cadastre 3D du canton de Genève.

L'implémentation de base de cette fonction reprend le même principe que les contrôles précédents, mais seules les composantes X et Y sont corrigées. D'une part, cela suffit, car les façades des bâtiments sont toutes strictement verticales, et d'autre part, cette simplification permet de ne traiter que des intersections de lignes plutôt que des intersections de lignes et de faces (ce qui se traduit par un *script* plus léger et plus performant). Cependant, deux problèmes apparaissent :

- Certains mâts ne semblent pas être aux bons emplacements ;
- Des scellements sont situés très loin des façades du bâti et nécessitent un seuil de tolérance élevé pour être corrigés.

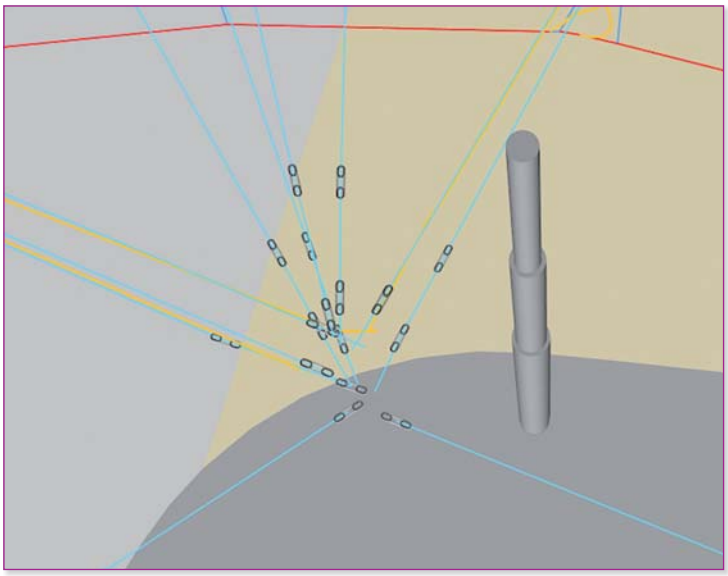

Figure 6 : Mauvais positionnement d'un mât TPG.

Concernant les mâts, les données nous avaient déjà été fournies par les TPG par le passé. Après contact de la personne responsable de la gestion de ces données, il apparaît que des changements de mâts ont eu lieu depuis, d'où les incohérences avec le levé effectué. La dernière mise à jour de ces données date en effet de 2011, et correspond en réalité à une reprojection suite au changement du système de référence suisse (MN03 vers MN95). Le lever des mâts, quant à lui, a été réalisé en 2008. Malheureusement, aucune base de données plus récente n'est disponible à ce jour. Le levé et la représentation des mâts TPG ne faisant pas partie du contrat, la seule solution possible consiste à modifier l'application pour que le contrôle devienne facultatif : il ne s'effectuera pas si l'utilisateur ne spécifie pas l'emplacement du fichier de la base de données des mâts.

Les bâtiments 3D sont, quant à eux, directement extraits de la base de données du SITG (Système d'Information du Territoire de Genève) ; ils correspondent donc bien à la version la plus récente. Le problème des écarts importants entre scellements et façades provient du niveau de détail de la modélisation du bâti : on classe habituellement les bâtiments en cinq groupes de niveaux de détails :

- Le LOD 0, qui correspond à une modélisation au niveau d'une région (orthophotographie plaquée sur un MNT) ;

- Le LOD 1, qui représente les bâtiments sous forme de blocs avec des toits plats (extrusion) ;
- Le LOD 2, qui rajoute la modélisation simplifiée de la toiture, et parfois une texture ;
- Le LOD 3, qui intègre les détails architecturaux comme les portes ou les fenêtres ;
- Le LOD 4 qui représente également l'intérieur du bâtiment.

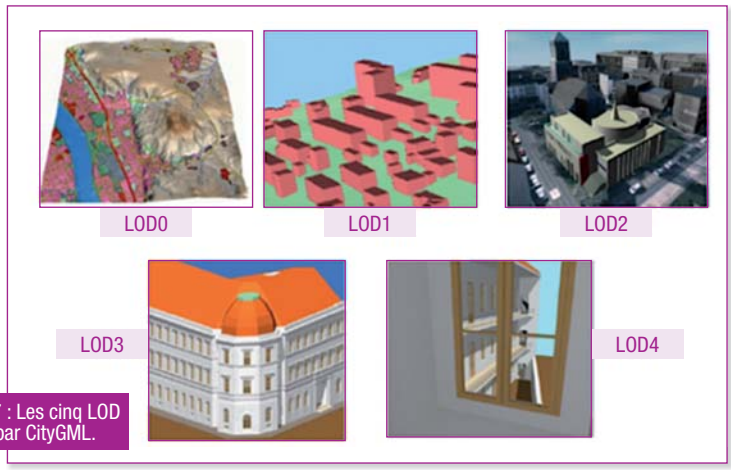

Figure 7 : Les cinq LOD définis par CityGML.

Dans notre cas, on se situe au niveau du LOD 2. Comme Laurène Menu (2011) l'avait remarqué, les bâtiments ont été obtenus par extrusion depuis leur empreinte cadastrale. De ce fait, certains éléments sur lesquels reposent les scellements ne sont pas modélisés (les balcons par exemple), d'où des écarts parfois importants. La figure 8 illustre le souci (le mur réel est en rouge tandis que la modélisation est en bleu ; le point noir représente le scellement).

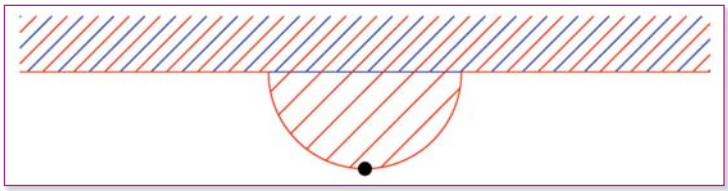

Figure 8 : Erreur topologique entre un scellement et une façade.

Il convient donc d'introduire un deuxième seuil de tolérance dans le *script*, permettant d'avoir une tolérance assez large sur le contrôle des scellements, et plus fine sur le reste de la base.

Cependant, il apparaît que certains écarts sont toujours trop importants : avec un seuil de tolérance fixé à cinquante centimètres, plusieurs scellements restent non corrigés. La question que l'on se pose est alors : « *Faut-il corriger ces scellements à tout prix pour respecter la continuité avec le bâti 3D, ou conserver volontairement ces erreurs de topologie pour conserver la précision de notre modélisation ?* » La décision s'est portée sur le deuxième choix, mais pour pallier un possible changement d'avis, la même mesure que pour les mâts a été adoptée (à savoir, laisser à l'utilisateur le choix de corriger ou non la base selon le bâti).

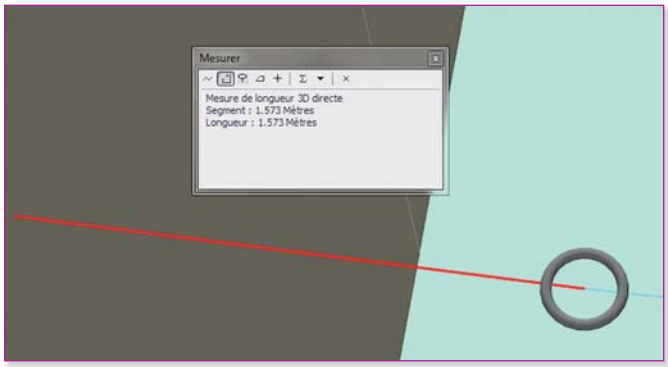

Figure 9 : Écart important entre un scellement et une façade.

Conclusion

Quoique le *script* final fonctionne correctement, il n'est pas des plus optimisés. En effet, il a nécessité environ huit heures pour traiter l'ensemble des données, sans effectuer les corrections selon les mâts et le bâti, sur un ordinateur haut de gamme (processeur Intel Core i7, 32 Go de mémoire vive, disque SSD…). La faute au langage *Python*, qui n'est pas le plus rapide à exécuter (mais un des plus intuitifs à utiliser). Cependant, il faut considérer que cette application n'a d'utilité qu'en amont pour constituer la base de données. Il n'est donc pas anormal de privilégier la rapidité de développement plutôt que la vitesse d'exécution.

Conception de la symbologie

Un SIG n'est rien sans une symbologie adaptée. Ceci est d'autant plus vrai lorsque celui-ci est en trois dimensions, où des objets situés sur plusieurs plans peuvent se superposer. Ainsi, il est important de mettre au point une symbologie qui sera parlante pour l'utilisateur final, et représentative de la réalité.

État de l'art

De manière à respecter les contraintes énoncées précédemment, les TPG souhaiteraient que

cette symbologie soit inspirée de la réalité. Autrement dit, il faudrait qu'un panneau soit modélisé par un symbole en forme de panneau plutôt que représenté par un point coloré par exemple. Par chance, la société *Kummler+Matter SA*, qui fournit les différents éléments aériens des lignes, a accepté que les symboles de leur catalogue soient réutilisés dans le cadre de ce contrat ; ce catalogue est facilement accessible sur *Internet* et en PDF.

Un premier essai, consistant à insérer directement ces symboles (sous forme d'images 2D) dans la modélisation, avait déjà été mené avec un résultat peu probant, soulignant la nécessité d'utiliser de véritables symboles en 3D. Ceci est possible avec *ArcGIS*, qui gère des modèles 3D dans divers formats (*Collada*, *SKP*, 3DS, etc.).

Concernant la modélisation des symboles elle-même, *SketchUp* paraissait parfaitement adapté car gratuit, simple d'utilisation, et produisant des fichiers *SKP* directement pris en charge par *ArcGIS*.

Dessin des symboles

Comme présenté précédemment, le logiciel *Trimble SketchUp* a été choisi pour modéliser les symboles en 3D. Ce travail ne sera effectué que pour les entités ponctuelles, la symbologie des linéaires étant définie directement dans *ArcGIS* par une simple attribution d'épaisseurs et de couleurs aux polylignes.

Le catalogue de *Kummler+Matter SA* propose, pour chaque objet, un(e) à quelques schéma(s) et/ou photographie(s), ainsi que certaines cotes. À partir de ces données, il est donc possible de reproduire chaque élément sous *SketchUp*, moyennant quelques simplifications. L'objet n'étant pas toujours dimensionné en totalité, il est souvent nécessaire d'estimer les cotes manquantes par des mesures à l'écran puis par une règle de trois par rapport aux mesures connues. Ces modèles estimés ne reproduisent pas parfaitement la réalité, mais suffisent à s'en faire une bonne représentation.

Import et attribution des symboles

Pour pouvoir attribuer correctement les symboles, il convient au préalable de catégoriser chaque classe d'entités selon un ou plusieurs champs (par exemple, les panneaux seront catégorisés selon le type de véhicule, le type de panneau et la vitesse – qui n'a de réelle utilité que pour les panneaux de type « *vitesse* »). Il suffit ensuite d'importer le bon symbole pour chaque catégorie de chaque classe d'entités. Pour ce faire, il faut, lors de la définition du symbole, choisir le type « *symbole ponctuel 3D* » et sélectionner le fichier approprié. La procédure est donc très simple.

Initialement, l'idée était de garder les symboles au format natif *SketchUp*, qui est censé être reconnu par *ArcGIS*. Cependant, ce dernier ne prend pas en charge toutes les versions du format *SKP*, ce qui complique singulièrement la manœuvre. Heureusement, *SketchUp* exporte au format *Collada*, qui est aussi reconnu par *ArcGIS* ; en outre, ce format est également géré par un grand nombre de logiciels de modélisation et de CAO, ce qui en fait finalement un choix judicieux si ces modèles devaient être réutilisés ou retravaillés par la suite.

Gestion de l'orientation des symboles

À ce stade, la symbologie est bien affectée à chaque entité de la base. En revanche, l'orientation de nos symboles reste celle par défaut et ne correspond pas à la réalité.

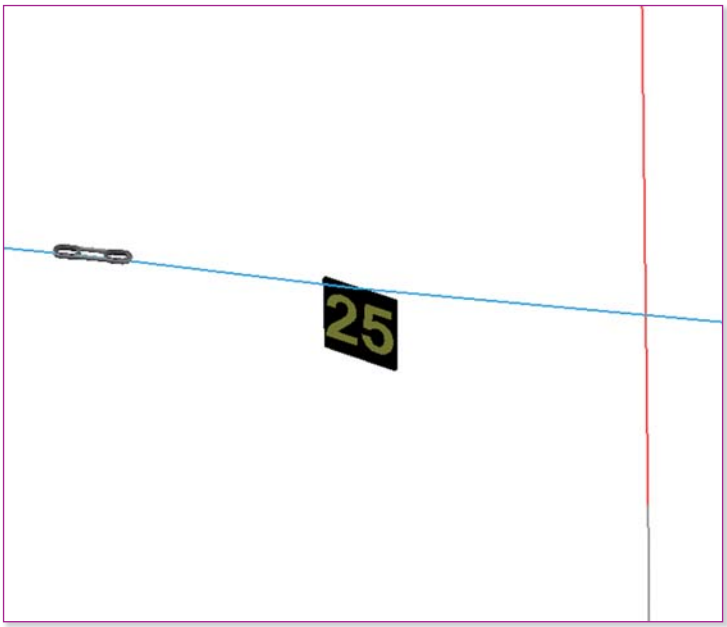

Figure 10 : Orientation par défaut d'un panneau.

De même que pour les contrôles topologiques, orienter chaque symbole manuellement représenterait un travail titanesque. De ce fait, automatiser la procédure apparaît à nouveau comme une évidence. Après quelques recherches parmi les options de symbologie d'*ArcGIS*, il ressort que le logiciel offre d'orienter automatiquement les symboles selon un champ. Il suffit donc de développer une application qui se chargera de calculer l'orientation de chaque symbole et de l'enregistrer dans un champ approprié. Le langage *Python* ayant fait ses preuves pour la manipulation des *Shapefiles* lors des contrôles topologiques, il est de nouveau utilisé dans ce cadre.

Le fonctionnement du *script* est relativement simple : pour chaque ponctuel ayant besoin d'être orienté (isolateurs, isolateurs de haubans, sectionneurs, aiguillages, croisements, antennes, panneaux, feux d'aiguillages, boucles isolantes et guides-perches), un nouvel attribut est créé : *rot_z*, qui correspond à l'orientation du symbole selon l'axe (O, z) (c'est donc un gisement). Chaque valeur est ensuite calculée ainsi : le sommet de linéaire sur lequel l'entité est fixée est identifié, puis le gisement de ce sommet vers le suivant est calculé (ou vers le précédent lorsque l'entité à orienter est située sur le sommet final du linéaire).

Les isolateurs de haubans représentent un cas particulier, ceux-ci devant également être orientés selon leur axe (O, x) (équivalent à une distance zénithale). Un attribut *rot_x* est donc créé pour cette classe d'entités et est complété d'une manière similaire au calcul des attributs *rot_z* (la différence réside dans le calcul de distances zénithales plutôt que de gisements).

Pour finir, cette étape faisant partie du travail de conception du SIG au même titre que les contrôles topologiques, et étant tout deux développés dans le même langage de programmation, les deux outils se voient regroupés dans une interface commune.

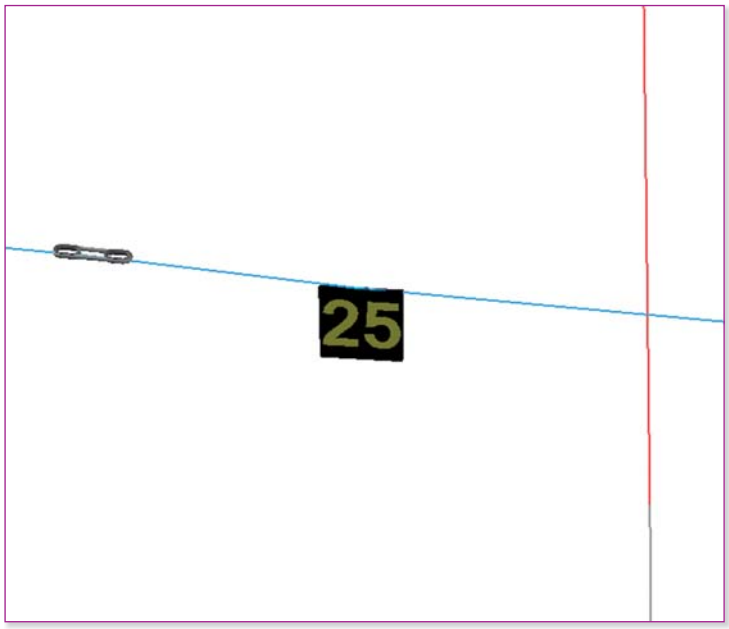

Figure 11 : Orientation d'un panneau après calcul.

Figure 12 : Interface du programme « Traitement des lignes TPG ».

Gestion de l'affichage 2D/3D

À l'usage, une faiblesse de notre modélisation apparaît. En effet, il faut savoir qu'en plus de la visualisation en trois dimensions, *ArcScene* propose également une vue orthographique (donc en 2D, vue du dessus). Or, si notre symbologie est parfaitement adaptée à la vue 3D (perspective), il en est tout autrement lors du passage en vue orthographique : les symboles apparaissent trop petits, et parfois mal orientés (les informations des panneaux ne sont plus visibles par exemple).

Figure 13 : Symbologie 3D et vue orthographique.

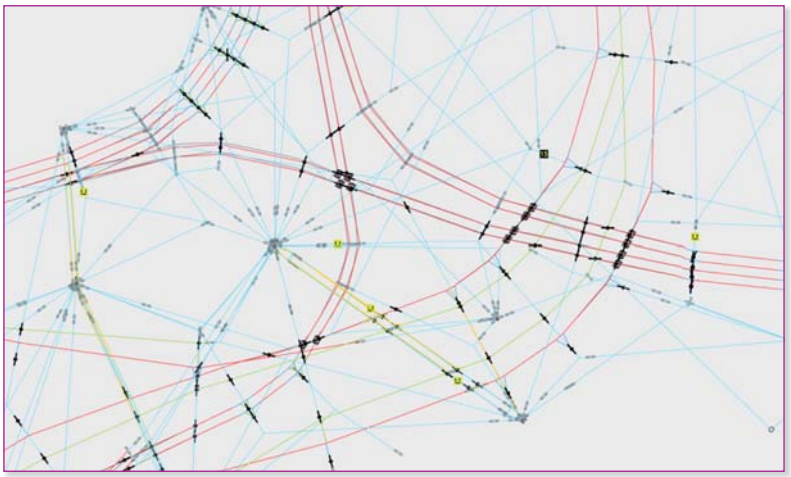

Figure 14 : Symbologie adaptée à la vue orthographique.

La solution consiste en la création d'un outil qui permet de jongler entre les deux types de visualisation et d'adapter la symbologie à la vue choisie. Il faut également savoir que celui-ci doit être disponible directement dans *ArcScene*, intégré dans une barre d'outils. Il est donc nécessaire de créer une extension. Pour réaliser une telle extension, deux solutions sont possibles :

- La programmation avec le *Framework*.NET (*C#* ou *VB.NET*) ou en *Java* depuis la version 10.0 ;
- La programmation en *Python* à partir de la version 10.1.

HKD Géomatique disposant de licences pour les deux versions, il est décidé d'utiliser le langage *C#*, qui permettra le développement d'une extension compatible avec toute version d'*ArcGIS* à partir de la 10.0. De plus, *C#* permet de créer des interfaces relativement élaborées (avec la gestion de divers événements survenant dans les boîtes de dialogues notamment, chose qui n'est pas possible avec *TkInter*), ce qui nous sera très utile par la suite. Pour ce faire, il est nécessaire d'installer *ArcObjects SDK*, qui permet non seulement d'utiliser les fonctionnalités d'*ArcGIS* dans le script *C#*, mais aussi de compiler la solution au format *.esriaddin* (format des extensions *ArcGIS*).

Pour commencer, la symbologie 3D initiale est enregistrée dans un ensemble de fichiers *.lyr* (un pour chaque classe d'entités) afin de la réutiliser plus tard sans avoir besoin de la redéfinir. On adapte ensuite la symbologie à l'affichage en 2D : la taille des symboles est doublée et l'orientation des panneaux et des feux d'aiguillage est modifiée, de manière à ce qu'ils soient vus de face par l'utilisateur. Cette nouvelle symbologie est alors sauvegardée dans un deuxième jeu de fichiers *.lyr*.

Deux outils sont alors codés : l'un permet de gérer le passage à la vue en perspective, et l'autre gère l'affichage en 2D. Chacun de ces outils devant utiliser respectivement les deux jeux de symboles définis précédemment (les fichiers *.lyr*), il est nécessaire de les définir comme ressources incorporées plutôt que de les utiliser tels quels. Ainsi, l'extension compilée intègrera elle-même ces fichiers et constituera un fichier unique, comme c'est le cas habituellement pour une extension *ArcGIS*. Le code en lui-même reste assez simple puisqu'il suffit d'utiliser les outils d'*ArcGIS* disponibles via *ArcObjects SDK* pour appliquer la symbologie et changer la vue. La difficulté résulte dans le fait que l'extension ne peut utiliser directement ces ressources (hormis certains types de fichier comme les images). Il faut donc auparavant les extraire. Pour cela, l'extension crée un répertoire temporaire, y extrait la symbologie à utiliser, l'applique à la base de données, et supprime finalement le répertoire d'extraction.

Pour finir, il faut préciser que ce travail a été initialement réalisé pour être exploité sous *ArcScene*, en raison de sa prise en charge de la 3D. Cependant, si l'utilisateur souhaite travailler dans un environnement 2D, il n'est pas exclu qu'il se dirige vers le module *ArcMap*. On doit alors faire face à un nouveau problème : bien que les symboles ponctuels 3D soient bien reconnus, l'affichage n'est pas convaincant (symboles trop petits, mauvaise orientation). Par manque de temps, ce problème n'a malheureusement pas encore pu être traité, et reste donc à résoudre.

Mise en place d'outils d'export

L'outil que constituera ce SIG pour les TPG pourra servir dans diverses applications. Les TPG souhaitent ainsi disposer de fonctions permettant l'export de cette base de données, plus particulièrement vers le monde du DAO. L'export en PDF est également intéressant pour extraire diverses vues de la modélisation.

État de l'art

Au premier abord, cette question semble trouver une réponse assez simple, *ArcGIS* intégrant nativement un outil d'export vers les formats *DWG* et *DXF*. Cependant, il s'avère que celui-ci n'est pas suffisant. En effet, bien que la conversion des linéaires soit fidèle, il n'en est pas de même avec les ponctuels, car

aucune symbologie n'est appliquée pendant le traitement. Les points sont finalement représentés par des points, et non par des symboles. Ceci nuit à la lisibilité du plan et entraîne une importante perte d'informations. Nous réaliserons donc un outil qui permettra de pallier ces imperfections.

Les données étant toujours au format *Shapefile*, il est naturel de se tourner une fois de plus vers le langage *Python* et le module *pyshp*. En revanche, le format *DWG* étant un format propriétaire d'*Autodesk*, il est très difficile (voire impossible) de trouver un module pour ce type de fichier. En revanche, le *DXF*, format ouvert, fait l'objet de la bibliothèque *SDXF* qui permet de générer un fichier *DXF* directement depuis un *script Python*. De plus, les plans de pose du réseau aérien que les TPG ont en leur possession sont également au format *DXF*. Le choix de ce format pour notre export permet donc d'uniformiser les deux sources de données.

Par ailleurs, la vocation des plans obtenus par extraction et celle des plans de pose seront similaires. Ces derniers étant en 2D, notre export le sera également, toujours dans ce souci d'uniformité. Il faudra aussi réaliser une nouvelle symbologie, inspirée de celle présente sur les plans de pose. Pour cela, les TPG ont mis à notre disposition un document qui consigne ces symboles. Certains pourront être repris tels quels, mais pour quelques éléments non représentés ces plans, il faudra les créer de toutes pièces.

Il serait également intéressant pour les TPG de disposer d'un outil permettant l'export en PDF, de manière à réaliser des présentations. *ArcGIS* intègre également ce type d'export et le réalise très bien. Une classe *ExportPDF* est d'ailleurs présente dans *ArcObjects SDK* et laisse présager une possible intégration dans notre barre d'outils. Une autre possibilité : il existe des modules pour *Python* qui permettent la lecture et/ou l'écriture de fichiers PDF, de la même manière que *SDXF* avec les DXF. Le PDF est en effet un format ouvert, lui aussi, et donc relativement facile à écrire (en théorie). Enfin, reste la possibilité d'utiliser l'export DXF puis un outil de conversion DXF vers PDF, ce qui entraînerait la perte de la représentation sous *ArcGIS*.

Définition d'une symbologie DXF

Comme on vient de le voir, il est nécessaire de réaliser une nouvelle symbologie spécifique pour le fichier *DXF* issu de l'export. L'idée de départ était de réaliser manuellement des blocs pour ces symboles et de les regrouper dans un même répertoire (en quelque sorte, un travail similaire à celui décrit précédemment), puis de les insérer dans le DXF lors de l'export. Cependant, il se trouve que le module SDXF ne permet pas d'insérer des éléments extérieurs dans le dessin. En revanche, il est tout à fait possible de créer nos blocs directement par programmation dans le script *Python*. Cette solution nécessite cependant de réaliser des tests après l'implémentation de chaque bloc, afin de vérifier si celui-ci est effectivement tracé comme espéré. Pour éviter une perte de temps importante - lancer la totalité du *script* (et donc traiter toute la base de données) - une méthodologie a été mise en place :

- On crée un *script* annexe, dans lequel on code uniquement la définition du bloc en cours de création et son insertion à l'origine du dessin. Ces blocs correspondent à un ensemble de polylignes et de solides ;
- Lorsque le résultat escompté est atteint, cette portion de code est insérée dans le *script* principal.

```
b_Panneau_danger = dxf.Block("Panneau_danger_bloc")
b_Panneau_danger.append(dxf.PolyLine(points = [(-1.6,0.4),(1.6,0.4),(1.6,-0.4),(-1.6,-0.4),(-1.6,0.4)]))
b_Panneau_danger.append(dxf.PolyLine(points = [(-1.2,0.35),(-1.55,-0.35),(-0.85,-0.35),(-1.2,0.35)]))
b_Panneau_danger.append(dxf.Solid(points = [(-1.2314,0.2271),(-1.1568,0.2271),(-1.3247,-0.1302),(-1.2678,-0.0526)]))
b_Panneau_danger.append(dxf.Solid(points = [(-1.2678,-0.0526),(-1.1214,0.0373),(-1.3247,-0.1302),(-1.2117,-0.0689)]))
b_Panneau_danger.append(dxf.Solid(points = [(-1.2117,-0.0689),(-1.1214,0.0373),(-1.2155,-0.2098),(-1.1856,-0.2153)]))
b_Panneau_danger.append(dxf.Solid(points = [(-1.2464,-0.1951),(-1.1513,-0.2125),(-1.2172,-0.3204),(-1.2172,-0.3204)]))
b_Panneau_danger.append(dxf.Text(text = "600 Volts",point = (-0.72,-0.12), height = 0.25))
b_Panneau_danger.append(dxf.PolyLine(points = [(1.2,0.35),(1.55,-0.35),(0.85,-0.35),(1.2,0.35)]))
b_Panneau_danger.append(dxf.Solid(points = [(1.1686,0.2271),(1.244,0.2271),(1.0753,-0.1302),(1.1322,-0.0526)]))
b_Panneau_danger.append(dxf.Solid(points = [(1.1322,-0.0526),(1.2786,0.0373),(1.0753,-0.1302),(1.1883,-0.0689)]))
b_Panneau_danger.append(dxf.Solid(points = [(1.1883,-0.0689),(1.2786,0.0373),(1.1845,-0.2098),(1.2144,-0.2153)]))
b_Panneau_danger.append(dxf.Solid(points = [(1.1536,-0.1951),(1.2487,-0.2125),(1.1828,-0.3204),(1.1828,-0.3204)]))
Dessin.blocks.append(b_Panneau_danger)
```

Figure 15 : Un bloc DXF codé en Python avec le module SDXF.

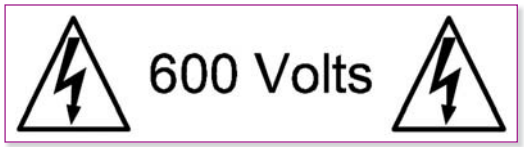

Figure 16 : Visualisation du bloc codé en figure 15.

Développement de l'outil

Grâce aux deux modules précédemment cités, le principe de fonctionnement se dessine : l'idée est de lire la géométrie et les attributs des *Shapefiles* en entrée, puis de traiter ces données pour écrire un fichier *DXF* en sortie.

Le *script* est structuré en trois grandes parties. La première « *initialise* » le traitement : elle s'occupe d'importer les différents modules utilisés, de spécifier les répertoires de travail (celui contenant les données ainsi que celui où sera enregistré le fichier exporté), puis de lire les données contenues dans les *Shapefile*. Plus tard, lors de la définition des couleurs des calques, un dernier élément a été rajouté : le format DXF utilisant un système numérique pour représenter les couleurs (par exemple le

chiffre 1 code le rouge, le 2 le jaune, etc.), les couleurs employées n'apparaissaient pas clairement dans le code, ce qui nuisait à sa lisibilité. On a donc affecté la valeur numérique de quelques couleurs de base à des constantes portant leur nom. Ainsi, plutôt que d'appeler une couleur par son code, elle peut l'être par son nom.

La deuxième partie est une étape de préparation du dessin. Elle comprend la création des différents calques et la définition des symboles sous forme de blocs. Il est intéressant de regrouper dans une même partie toutes les définitions de couches et de blocs. Structuré de cette manière, celui-ci gagne en visibilité et facilitera le travail d'un futur développeur si le *script* devait être repris par la suite.

Enfin, la troisième étape traite réellement les données et les trace dans le *DXF*. D'une manière générale, le procédé est le même pour chaque classe d'entités :

- Pour un ponctuel, le *script* récupère les coordonnées du premier élément, insère le bloc correspondant à ces coordonnées, puis passe au suivant ;

- Pour un linéaire, le *script* récupère les coordonnées de chaque point du premier élément et les insère dans une table. Lorsque chaque point a été traité, le linéaire est tracé suivant cette table. L'opération est ensuite réitérée pour les éléments suivants.

Cependant, certains symboles à insérer requièrent des traitements particuliers :

- Pour la plupart des ponctuels, une orientation est à prendre en compte. Il se trouve qu'elle est identique à celle enregistrée dans l'attribut *rot_z* calculé précédemment, et est donc récupérée directement dans la table attributaire ;

- Certains symboles n'ont pas d'équivalent dans le modèle d'origine : les retenues de courbes (cas particuliers de pinces de courbes), les pinces de passage, les croisements de lignes de contact à hauteurs différentes, et les amarrages pour fil de contact ou feeder. Pour ceux-ci, des tests particuliers sont donc à effectuer.

Intégration dans ArcGIS

Chronologiquement, cet outil a été développé avant celui concernant la gestion de l'affichage sous *ArcScene*. De ce fait, l'intégration dans le logiciel n'avait pas encore été abordée, et le développement avec des outils maîtrisés avait été privilégié pour assurer une bonne avancée des recherches. Cependant, comme expliqué précédemment, la création d'une extension *ArcGIS* a nécessité un développement en *C#*, ce qui impliquait de réécrire entièrement l'outil dans ce langage. En plus d'entraîner une perte de temps non négligeable, il aurait fallu trouver un équivalent en *C#* de la bibliothèque *SDXF*. Or, il n'en existe qu'un seul et, selon son développeur, celui-ci génère parfois des erreurs. Cette solution n'est donc pas viable pour réaliser un produit destiné à être livré à un client.

En revanche, il est possible d'exécuter une application extérieure au script *C#* depuis celui-ci. On a donc codé la partie interface de l'outil en *C#*, tandis que la partie traitement restait en *Python*. Cette dissociation pose cependant certaines contraintes : *primo*, il faut que la partie interface « *connaisse* » l'emplacement du module de traitement en toute circonstance ; *secondo*, l'extension doit être fonctionnelle sans installation annexe (autre qu'une installation d'*ArcGIS* bien entendu) ; *tertio*, il faut protéger la partie codée en *Python*, où le code source est exposé à une modification involontaire de la part de l'utilisateur.

Un procédé permet de répondre à ces trois problématiques. Celui-ci consiste à réunir la « *compilation* » du script en *Python* déjà évoqué à l'incorporation de celui-ci en ressource de l'application *C#*. Outre l'exécutable, il faut également ajouter dans ces ressources l'ensemble des fichiers générés par *py2exe*.

La partie interface de l'outil réalise ainsi un certain nombre d'actions en amont du véritable traitement des données. Premièrement, elle propose à l'utilisateur de choisir le répertoire où s'effectuera l'export. Le chemin d'accès à ce répertoire est alors noté dans un fichier *.txt*, lui-même placé dans un répertoire temporaire. Nous verrons l'utilité de cette action par la suite.

Ensuite, les différentes couches présentes dans la scène (pour *ArcScene*) ou dans la vue (pour *ArcMap*) sont exportées au format *Shapefile* dans le répertoire temporaire. Il faut, en effet, convertir les données livrées au format *Géodatabase* (.gdb), en *Shapefile* compréhensible par *pyshp*.

Le *script* de traitement (et tous ses fichiers annexes) est alors

extrait à son tour dans le répertoire temporaire, et est exécuté. Il va commencer par récupérer le chemin d'accès inscrit dans le fichier *.txt* mentionné précédemment afin de savoir où écrire les données en sortie, et réalise enfin l'export. Pour finir, le dossier temporaire et son contenu sont effacés.

Export en PDF

Dans *ArcObjects SDK*, il est nécessaire de passer par la création d'une instance de la classe *ExportPDF,* qui correspond à ce même outil. Toutefois, il apparaît que celui-ci est surtout utilisé dans *ArcMap* et peu (voire pas du tout) sous *ArcScene*. Les différents exemples trouvés (y compris sur l'aide officielle d'*ArcGIS*) ne l'utilisent ainsi qu'avec un environnement *ArcMap*, et font donc appel à des types d'objets qui ne sont pas présents dans *ArcScene*. Bien que la manipulation soit en théorie possible, puisque l'outil intégré est fonctionnel, le manque d'informations sur le sujet n'a pas permis de concrétiser cette piste.

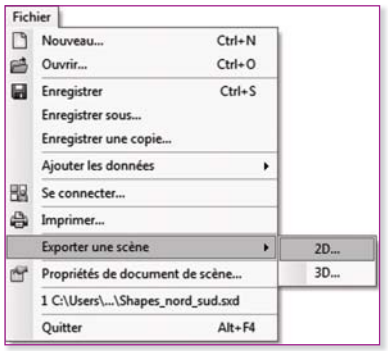

Figure 17 : Outils d'export 2D d'ArcScene.

Une deuxième piste a donc été explorée : réaliser un travail similaire à l'export *DXF*, mais avec une bibliothèque PDF. Il existe en effet des bibliothèques de ce type disponibles sur internet, tel que *pyPDF* ou *pdfrw*. Cependant, celles-ci sont conçues pour traiter principalement du texte, voire des images, mais pas des éléments vectoriels comme il serait nécessaire pour notre projet. De ce fait, pour pouvoir avancer sur cette piste, il convient de réaliser nous-même une bibliothèque adaptée à nos besoins.

Pour ce faire, il faut auparavant étudier la structure d'un fichier PDF. Il existe une norme (ISO 32000-1) qui la décrit. Ce format ayant été créé par *Adobe*, cette dernière met à disposition une version non-officielle et gratuite de cette norme (contrairement à l'originale qui est payante). La structure du PDF est relativement simple, et organisée en plusieurs blocs. Nous ne rentrerons pas ici dans les détails de l'encodage du PDF, mais nous parlerons des difficultés rencontrées. En effet, si les données textuelles sont encodées de manière relativement compréhensible, ce n'est pas le cas pour d'autres types d'objets, tels que les éléments vectoriels (qui sont ceux qui nous intéressent pour notre export).

Un rapide coup d'œil sur un PDF généré depuis *AutoCAD* nous montre que ces éléments sont présents dans un bloc *stream*, totalement indescriptible pour nous. En réalité, l'ensemble des données présentes dans ce bloc ont été compressées afin de minimiser le poids du fichier. La figure 18 montre un aperçu de ce bloc lorsque le fichier est ouvert avec un éditeur de texte.

Alors que les logiciels capables de lire des PDF possèdent les algorithmes nécessaires à la décompression de ces données lors de leur lecture, rares sont les utilitaires qui permettent d'extraire le code initial qui correspond à celles-ci. Il faut également savoir

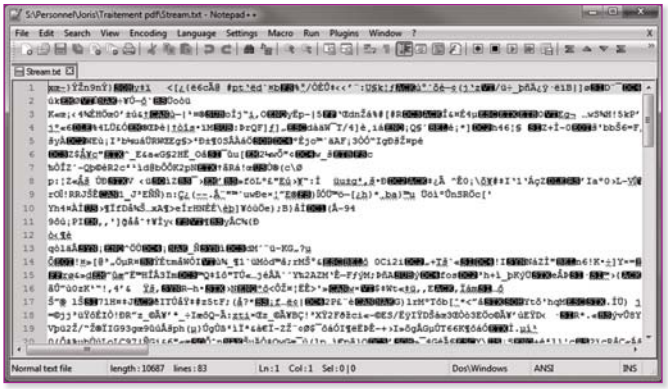

Figure 18 : Extrait d'un stream PDF non décompressé.

que c'est l'équivalent hexadécimal du code qui subit cette étape de compression/décompression. Après plusieurs tests, une bibliothèque *Python* a permis d'y parvenir. Une fois le code hexadécimal décompressé, il est converti en son équivalent *ASCII* qui, après quelques traitements de mise en forme, s'apparente bien à un fragment de fichier PDF.

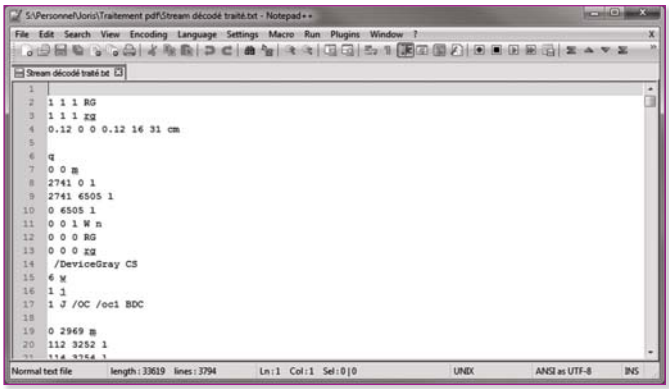

Figure 19 : Extrait d'un stream PDF après décompression.

Cependant, il faut reconnaître que la structure reste encore relativement compliquée à comprendre, et que réaliser manuellement une bibliothèque qui permettrait de générer un tel fichier n'est pas à la portée du premier venu.

Conclusion

Du côté de l'export en *DXF*, l'objectif a été pleinement atteint puisque l'outil est parfaitement fonctionnel. Cependant, il subsiste un défaut dans la méthode employée : le fait de

devoir extraire les données en *Shapefile* ainsi que le programme d'export et de l'exécuter en dehors d'*ArcGIS* plutôt que directement sur les données de la *Géodatabase* demande un temps de traitement relativement important, pendant lequel *ArcGIS* se fige. Ce n'est pas un réel problème dans le sens où le traitement est bien effectué et que le logiciel redevient actif à la fin de l'export, mais il serait toujours profitable de rechercher s'il est possible de réaliser l'export en « *arrière-plan* », de manière à ce que l'utilisateur puisse continuer à travailler. Enfin, l'outil actuel ne permet que d'exporter la totalité de la base de données ; une amélioration judicieuse serait de permettre à l'utilisateur de limiter l'export à une zone sélectionnée.

L'export en PDF, en revanche, n'a pas été intégré à notre extension, malgré les différentes pistes explorées. L'utilisateur d'*ArcGIS* disposera toujours de la possibilité de réaliser cet export simplement avec les fonctionnalités proposées d'origine par le logiciel, il est préférable de ne pas trop s'attarder sur ce point, et de développer les autres outils qui doivent être implémentés dans l'extension.

Développement d'outils de mise à jour

Un bon SIG se doit d'être régulièrement mis à jour. *ArcGIS* propose déjà des outils de mise à jour, que ce soit en 2D sous *ArcMap* ou en 3D dans *ArcScene*. Cependant, rappelons que les personnes susceptibles de devoir les utiliser sont les techniciens des TPG, peu habitués à l'utilisation de logiciels de SIG. Or, ces outils s'avèrent peu intuitifs, et relativement complexes pour une personne non rompue à cette pratique. Ainsi, il sera nécessaire de réaliser un outil tout aussi fonctionnel, mais simple d'utilisation.

État de l'art

Il est intéressant de noter que le logiciel *Trimble Trident-3D Analyst* utilisé par l'équipe de modélisation propose une méthode efficace pour la saisie des attributs en utilisant un système de listes déroulantes pour choisir la valeur de chaque attribut, ce qui permet de les renseigner conformément au modèle de données et d'éviter les variations orthographiques (majuscules/minuscules, singulier/pluriel, etc.). De plus, ce système entraîne un gain de temps conséquent, puisqu'il suffit de sélectionner une valeur d'attribut plutôt que de la saisir manuellement. Notre outil s'inspirera donc de ce procédé, et reprendra le système de listes déroulantes qui a prouvé son efficacité.

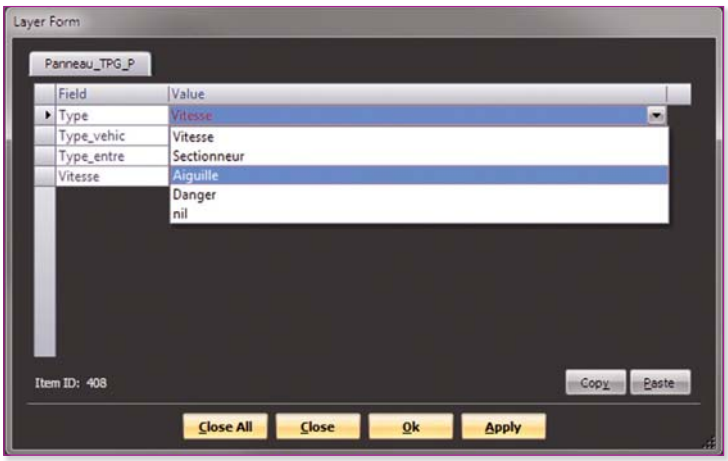

Figure 20 : Système de saisie des attributs dans Trimble Trident-3D Analyst.

Ne réalisant que l'interface de l'outil, le développement peut se faire directement et intégralement en *C#*. En effet, le *framework .NET* permet de réaliser des interfaces au moyen de boîtes de dialogues appelées *WinForms* (pour *Windows Forms*). Chaque *WinForm* peut ensuite contenir divers *widgets*, qui correspondent aux divers éléments graphiques présents dans la boîte de dialogue (boîtes de texte, listes déroulantes, cases à cocher, etc…). Il est ainsi possible de créer une interface totalement adaptée à l'utilisation que l'on veut en faire.

Conception graphique des boîtes de saisie

L'idée est d'utiliser des *WinForms* pour guider entièrement l'utilisateur lors de la saisie en lui demandant de remplir les informations géographiques et attributaires concernant l'objet à ajouter. Lorsque les éléments sont à choisir parmi une liste exhaustive de possibilités (la plupart des attributs), la saisie se fait au moyen de listes déroulantes. Pour tous les autres cas (les coordonnées par exemple), elle est réalisée par remplissage de champs. Initialement, il était prévu de créer un seul *WinForm* dont le contenu aurait été généré dynamiquement selon l'entité à créer. Cependant, le contenu

des boîtes de saisie variait de manière importante d'un élément à l'autre, ce qui se traduisait par un code complexe pour gérer tous les cas. De ce fait, il a finalement été décidé de créer une boîte pour chaque classe d'entités, ce qui oblige à gérer plus d'objets dans notre application, mais qui se révèle être beaucoup plus compréhensible et simple à coder.

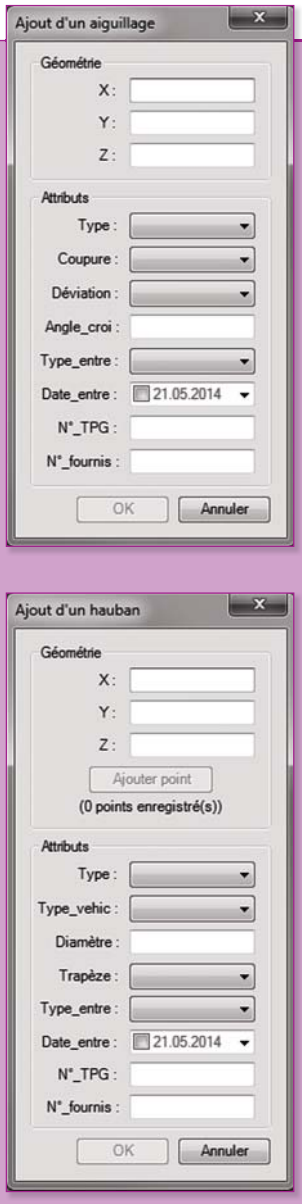

Figure 21 : Exemple de boîtes de saisie de nouveaux éléments.

On remarquera que les boîtes de saisie sont toutes deux divisées en deux parties : « *géométrie* » et « *attributs* ». Le bloc « *géométrie* » diffère suivant le type d'entité : pour un ponctuel, il contient uniquement les champs « *x* », « *y* » et « *z* » à remplir, tandis que pour un linéaire, il comporte également un bouton « *Ajouter point* » et un compteur. En effet, un linéaire étant défini par plusieurs points, il faut que l'utilisateur puisse rentrer l'ensemble des points qui le compose avant de valider la création de l'objet. Par ailleurs, notons que ce bouton n'est pas « *cliquable* » tant que les trois champs n'ont pas été remplis (nous en reparlerons après). Les différents points ainsi enregistrés sont consignés dans une liste qui s'actualise au fur et à mesure de la saisie.

Le cadre « *attribut* » varie bien évidement en fonction de l'entité à insérer. Néanmoins, certains attributs sont communs : *Type_entre*, *Date_entre*, N°_*TPG* et N°_*fournis*. L'attribut *Date_entre* constitue un cas particulier, car il est rempli à l'aide d'un *DateTimePicker*. Ce *widget* permet de sélectionner directement une date dans un calendrier. De plus, le remplissage de cet attribut étant facultatif, il contient une case à cocher pour décider de son activation.

Figure 22 : Saisie d'une date à l'aide d'un DateTimePicker.

Articulation de l'interface

Une interface étant composée de plusieurs boîtes de dialogue, il convient de faire le lien entre chacune d'elles. De plus, pour éviter bon nombre d'erreurs lors du traitement des données renseignées dans celles-ci, il est important de permettre la validation d'une boîte uniquement après que les informations obligatoires ont été correctement remplies.

Lors du lancement de l'outil, il est demandé de choisir quel type d'entité est à créer. Cela permet à l'application non seulement de savoir dans quelle couche elle va devoir tracer l'élément, mais également de présenter à l'utilisateur la boîte de dialogue de saisie adéquate.

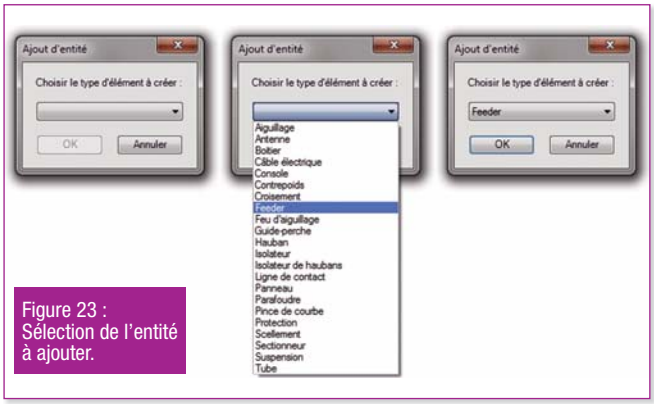

Figure 23 : Sélection de l'entité à ajouter.

La figure 23 montre également l'activation du bouton « *OK* » une fois que le choix a été fait dans la liste déroulante. Ceci est géré à l'aide d'une fonction qui surveille le changement du texte affiché dans la liste, et qui rend le bouton actif uniquement lorsque ce texte n'est pas vide (ce qui correspond à l'événement « un choix a été fait »). La fenêtre de saisie adaptée est alors affichée à l'écran.

On remarque ici le même procédé avec le bouton « *Ajouter point* ». La méthode est identique à celle utilisée dans la boîte de

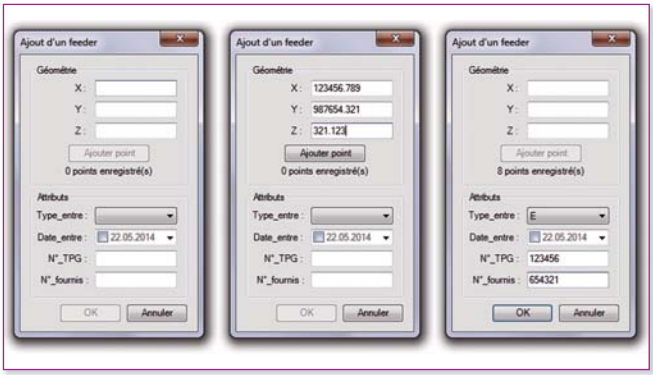

Figure 24 : Saisie d'un nouveau feeder.

dialogue précédente et porte sur les trois champs du cadre « *Géométrie* ». De plus, on vérifie que ceux-ci sont bien remplis avec des caractères numériques puisqu'il s'agit de coordonnées. Pour pouvoir valider la boîte de dialogue, au moins deux points doivent être enregistrés (ou les coordonnées doivent être renseignées pour un ponctuel) et les attributs doivent être complétés (hormis *Date_entre* qui est facultatif).

Après avoir validé une fenêtre de saisie, un nouveau *WinForm* permet à l'utilisateur de choisir s'il veut créer une nouvelle entité de même type que celle qu'il vient de saisir ou non. Ainsi, il n'a pas à relancer la procédure depuis le début de l'outil, ce qui constitue une économie de temps.

Modification d'entités existantes

La modification d'entité est plus compliquée à séparer en deux parties. En effet, l'interface a besoin d'informations sur les entités à modifier pour être fonctionnelle (elle doit par exemple récupérer les valeurs d'attributs de l'objet et les présenter dans une boîte de dialogue).

Pour commencer, l'application vérifie qu'une entité (et une seule) est sélectionnée. Les informations de position et d'attribut sur cet objet sont alors récupérées dans la base de données. La programmation de cette étape représente la difficulté majeure dans le développement de l'outil, car, pour accéder à ces informations, il est nécessaire de parcourir l'ensemble de la hiérarchie des classes d'*ArcObjects*, depuis les couches présentes dans *ArcGIS* jusqu'à l'entité sélectionnée, (diverses classes intermédiaires rendent cette hiérarchie relativement complexe). De plus l'aide en ligne d'*ArcObjects* pour *C#*.NET est relativement peu prolixe sur ce sujet. En revanche, son équivalent pour *VBA* propose un script répondant au problème. Une « *traduction* » de ce dernier a donc été effectuée pour l'implémenter en *C#* dans l'extension.

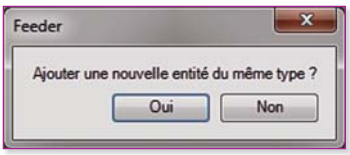

Figure 25 : Saisie d'un élément de même type.

Les boîtes de dialogue de modification d'entité sont alors identiques à celles de saisie, hormis le fait que les divers champs sont déjà remplis. De plus, une modification est apportée pour les entités linéaires afin de visualiser et modifier les informations géométriques facilement : les coordonnées des sommets sont affichées dans un tableau. Pour modifier un sommet, il suffit donc de changer les valeurs des cellules correspondantes. À noter que seules des valeurs numériques peuvent être saisies. Enfin, un clic droit sur ce tableau affiche un menu permettant d'accéder à des fonctions d'ajout et de suppression de sommet(s).

Suppression d'entités existantes

Pour l'outil de suppression d'entités, il suffit d'effacer de la *Géodatabase* les éléments que l'utilisateur aura préalablement sélectionnés. L'interface est de ce fait relativement simple, puisqu'elle consiste simplement en un message d'avertissement et de demande de confirmation (la suppression étant définitive).

Ici, on passe par le *Geoprocessor* « *DeleteFeatures* » d'*ArcObjects*,

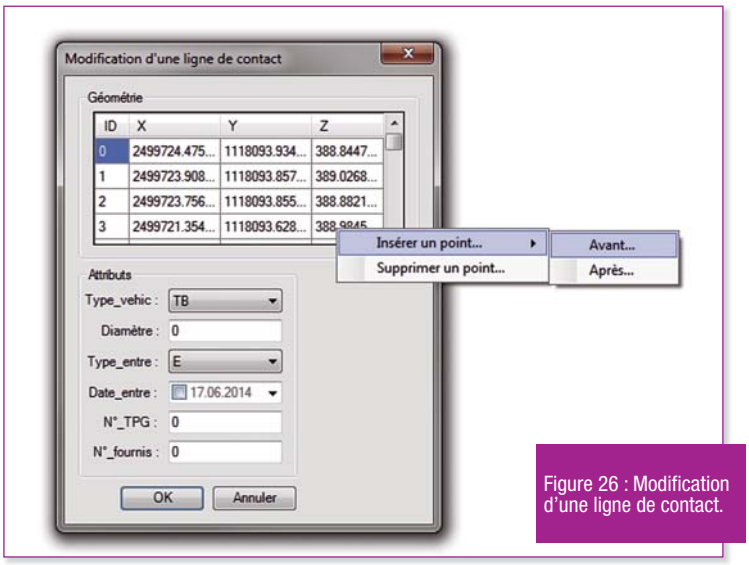

Figure 26 : Modification d'une ligne de contact.

qui prend une couche en entrée. Si une sélection est présente dans cette couche, seules les entités sélectionnées sont supprimées. Dans le cas contraire, tous les éléments sont effacés, ce qui pose problème. Il faut donc détecter en amont les couches comportant une sélection, et n'utiliser le *Geoprocessor* que sur celles-ci. Pour ce faire, on réutilise le procédé d'identification des couches à traiter.

Conclusion

Cet outil demandant un travail important, son développement est toujours en cours. À l'heure actuelle, la suppression d'entités est entièrement fonctionnelle et les interfaces de saisie et de modification sont terminées.

La difficulté principale de cet outil réside dans le fait que la structure des éléments de base de données dans *ArcObjects* est plus complexe que celle présentée habituellement en SIG (y compris *ArcGIS*). Si l'on a l'habitude de dire qu'une entité est contenue dans une couche, pour *ArcObjects*, l'entité fait partie d'une classe d'entité, elle-même représentée par un objet *FeatureLayer*, qui est affiché dans une couche de notre SIG. Autre exemple avec les sélections : dans un logiciel de SIG, on assimile naturellement une sélection à un ensemble d'entités. Avec *ArcObjects*, une sélection est elle-même un objet qui va pointer vers les entités sélectionnées. Ceci est dû non seulement à l'orientation objet de *C#,* mais également à la volonté d'*Esri* de réaliser un outil performant et permettant d'accéder aux fonctionnalités les plus avancées d'*ArcGIS*.

Bibliographie

Adobe Systems Inc. : Document management – Portable document format – Part 1: PDF 1.7, [en ligne]. 2008, 756 p. Disponible sur : http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf (consulté le 20/05/2014).

ESRI : ArcGIS Resource Center – ArcObjects SDK 10 Microsoft .NET Framework [en ligne]. Disponible sur http://help.arcgis.com/en/sdk/10.0/arcobjects_net/ao_home.html (consulté le 05/05/2014).

ESRI : ArcGIS Resource Center – ArcObjects SDK 10 Visual Basic for Applications, [en ligne]. Disponible sur : http://help.arcgis.com/en/sdk/10.0/vba_desktop/ao_home.html (consulté le 02/06/2014).

GeoRezo. http://georezo.net

SDXF – Python Library for DXF. In : Kellbot. Kellbot!, [en ligne]. Disponible sur http://www.kellbot.com/sdxf-python-library-for-dxf (consulté le 15/04/2014).

Kummler+Matter SA. : Catalogue lignes contact, [en ligne]. Disponible sur http://catalog.kuma.ch/fr (consulté le 09/05/2014).

Lachance B. : Développement d'une structure topologique de données 3D pour l'analyse de modèles géologiques [en ligne]. Mémoire de maîtrise en sciences géomatiques, Laval : Faculté des études supérieures de l'Université Laval, 2005, 116 p. Disponible sur http://archimede.bibl.ulaval.ca/archimede/fichiers/22501/22501.pdf (consulté le 05/02/2014).

Lawhead J. : pyshp – Python Shapefile Library, [en ligne]. Disponible sur http://code.google.com/p/pyshp (consulté le 13/05/2014).

Menu L. : Réalisation d'un cadastre aérien en 3D sur le canton de Genève : Mise en application au cas des lignes de Transports Publics Genevois. Mémoire d'ingénieur CNAM spécialité Géomètre et Topographe. École Supérieure des Géomètres et Topographes, Le Mans, 2011, 96 p.

Minéry C. : Des données 3D pour les architectes, urbanistes et paysagistes, [en ligne]. Mémoire d'ingénieur INSA spécialité Topographie, Institut National des Sciences Appliquées, Strasbourg, 2011, 69 p. Disponible sur http://eprints2.insa-strasbourg.fr/928/1/Mémoire_Cedric_MINERY.pdf (consulté le 05/05/2014).

OpenClassrooms : OpenClassrooms, [en ligne]. Disponible sur http://fr.openclassrooms.com (consulté le 09/05/2014).

Retzlaff J. et al. : Python : py2exe, [en ligne]. Disponible sur http://www.py2exe.org (consulté le 05/03/2014).

Stack Overflow. http://stackoverflow.com