Maquette 3D et programmation

Jean-Luc Ancey

15 décembre 2004

Moulinettes et scripts sous GNU/Linux
pour la production d'images de synthèse

1 Un mot sur la démarche de l'auteur

L'auteur s'intéresse en amateur à la 3D informatisée depuis plus de quinze ans, mais il a rarement pu y consacrer plus de quelques heures par mois depuis cette époque. Or, une maquette détaillée en 3D informatisée demande à peu près autant de soin qu'une maquette classique du type de celles qu'on réalise dans les clubs de modélisme. Il est donc souvent nécessaire à un amateur consciencieux d'étaler ce travail minutieux sur des semaines, des mois, parfois même des années. Entre-temps, les machines évoluent, les éditeurs informatiques font faillite ou sont rachetés, les programmes et les formats de fichier qui leur sont associés sont revus en profondeur, et cela oblige très souvent, malheureusement, à reprendre à zéro ou encore plus souvent à abandonner des maquettes qui étaient parfois déjà très élaborées. En outre, les logiciels de 3D, qui visent un public de professionnels, sont généralement très chers, et leurs utilisateurs sont véritablement rançonnés à coups de mises à jour.

Pour ces raisons, l'auteur n'a pas cessé au fil de ces quinze années d'essayer de mettre au point des techniques à la fois totalement indépendantes d'un logiciel précis, et facilement adaptables à tout logiciel de 3D qui lui passait entre les mains. La 3D se prête tout à fait à cette démarche un peu abstraite, parce que tous les logiciels de 3D, les anciens comme les modernes, les commerciaux comme les libres, ont recours à des principes géométriques qu'on envisage encore aujourd'hui à peu près exactement de la même façon que du temps d'Euclide et de Pythagore. La 3D est l'un des domaines informatiques où il est le plus facile de lutter contre les incompatibilités (ce qui n'empêche pas d'importants éditeurs informatiques de réaliser d'énormes profits grâce à ces incompatibilités). Les outils logiciels que l'on a besoin de mettre au point pour éviter cet inconvénient sont dans la plupart des cas de tout petits programmes de conversion, dont la mise au point n'est pas du tout hors de portée d'un amateur comme l'auteur, même quand il n'est que débutant et commence seulement à manipuler des langages de scripts du type Perl ou Python (l'auteur lui-même a fait ses premiers pas en Basic, sous MS-DOS).

Cet atelier est organisé autour de petits programmes de ce type – qu'on appellera ici des *moulinettes*. Ce sont très souvent des convertisseurs permettant de transformer en fichiers 3D, puis en images de synthèse, des séries de données géométriques – facilement compréhensibles par toute personne ayant écouté ses cours de maths jusqu'à la troisième, même d'une oreille très distraite. Il ne faut surtout pas se faire un monde de la complexité mathématique mise en jeu : l'auteur lui-même était médiocre en maths et franchement mauvais en géométrie.

Un autre aspect de la démarche de l'auteur, c'est qu'il a graduellement cessé d'utiliser des logiciels commerciaux spécialisés dans la 3D pour se rapprocher des logiciels libres les plus robustes et les plus courants, quitte à les utiliser d'une

façon très particulière parce qu'ils n'étaient pas vraiment faits pour la 3D. On verra en particulier qu'une partie très importante du travail de modélisation d'une forme peut être effectuée avec un simple logiciel de retouche d'images (cet atelier emploie Gimp, mais n'importe logiciel de retouche ferait l'affaire).

Une dernière chose qu'il faut savoir sur l'auteur, c'est qu'il a commencé à mettre au propre ses recherches alors qu'il cherchait à fonder un club informatique à Cochabamba, en Bolivie. C'est la raison pour laquelle le format qu'il utilise s'appelle CBB... et emploie une demi-douzaine de mots espagnols aussi compliqués à comprendre que *"puntos"* et *"color"*.

2 Exigences

2.1 Exigences matérielles

Cet atelier ne requiert aucune configuration matérielle particulière – sauf bien sûr une carte graphique pour afficher les images (et non pour les calculer : l'utilisation d'une carte accélératrice 3D n'est nullement requise). Les exigences en matière de puissance sont faibles, et la démarche exposée ici peut parfaitement être employée sur une machine vieille de plus de cinq ans, avec 48 Mo de mémoire vive. A vrai dire, une grande partie des moulinettes employées ici fonctionnaient déjà... sous MS-DOS.

2.2 Exigences logicielles

Cet atelier a recours à des outils GNU/Linux d'usage si courant qu'ils sont généralement installés en standard :

- le shell **bash** ;
- les langages de script classiques **Perl, Python,** mais aussi **Tcl/TK** (pour l'affichage des images uniquement) ;
- quelques programmes de conversion de fichiers liés au format ouvert **PPM**.

Tous ces outils sont disponibles en standard avec la plupart des distributions, et notamment la Knoppix 3.6.

La production des images de synthèse proprement dite exige l'installation préalable de **POV** (dit aussi Povray ou *Persistence of Vision*). Elle ne pose pas de problème particulier, mais elle n'est jamais réalisée en standard par les distributions Linux commerciales, pour des raisons juridiques.

L'installation d'Open GL n'est pas requise.

2.3 Connaissances requises

Les connaissances requises pour cet atelier sont réduites, voire nulles, mais il est quand même préférable de n'être pas effrayé par les manipulations les plus courantes lorsque l'on travaille avec un shell de Linux, notamment :

- `ls` pour voir la liste des fichiers d'un répertoire et éventuellement leur taille ;
- `cp` pour effectuer des copies de fichiers ;
- `mv` pour renommer des fichiers ;
- `cd` pour se déplacer dans l'arborescence du disque dur ;
- `rm` pour détruire des fichiers.

Les connaissances géométriques requises se limitent à la compréhension d'un repère x,y,z. Si en plus on a compris le théorème de Pythagore... c'est Byzance.

Enfin, il faut savoir se servir d'un éditeur de texte et d'un éditeur bitmap. L'éditeur de texte **kwrite** est très suffisant pour cet atelier, mais on peut bien sûr employer **vi** ou **emacs** si on les connaît mieux (l'auteur emploie et recommande emacs). Pour l'édition bitmap, une connaissance élémentaire de **Gimp** est appréciable, mais d'autres éditeurs même très sommaires peuvent convenir.

3 Installation des fichiers nécessaires à l'atelier

3.1 Décompression

Tous les fichiers nécessaires au bon déroulement de cet atelier sont inclus dans une archive compressée d'environ 1,4 Mo, nommée **atelier_06_nov_2004.tgz**. Ce fichier étant recopié dans le répertoire **home** de l'utilisateur, on le décompresse de la façon la plus classique :

```
tar -zxvf atelier_06_nov_2004.tgz
```

après quoi on peut théoriquement l'effacer (mais il peut être préférable de le garder quelque temps à titre de sauvegarde, pour se prémunir contre des erreurs de manipulation).

La décompression a créé un répertoire **atelier3d** contenant lui-même six sous-répertoires :

– **bin,** qui contient les moulinettes employées au cours de l'atelier, ainsi que quelques fichiers Ascii d'initialisation (d'extension **.ini**) ;
– **cbd**, qui contient une série d'objets en 3D, donnés à titre d'exemple ;
– **fabrscen** (fabrication de la scène), qui sera le répertoire de travail pour la production d'images de synthèse ; il ne faut pas hésiter à réaliser des copies de ce répertoire pour des usages particuliers ;
– **objtrond**, qui contient quelques exemples de scripts élémentaires pour la création d'objets en 3D ;
– **pantin**, qui contient un objet en 3D particulier, constitué de toute une hiérarchie de fichiers permettant l'articulation de ses éléments ;
– et enfin **textface**, qui sert à la production de mentions textuelles susceptibles d'être manipulées en 3D (par exemple pour placer des panneaux indicateurs dans une scène).

3.2 Accès direct aux moulinettes de l'atelier

Pour faciliter les manipulations, il est recommandé d'ajouter le répertoire des moulinettes à la variable PATH, selon une manipulation très classique :

```
cd atelier3d
cd bin
pwd
```

L'écran va alors afficher une mention du genre **/home/knoppix/atelier3d/bin** et il faudra donc écrire :

```
PATH=$PATH:/home/knoppix/atelier3d/bin
```

4 Production de la première image (le rôle du Makefile)

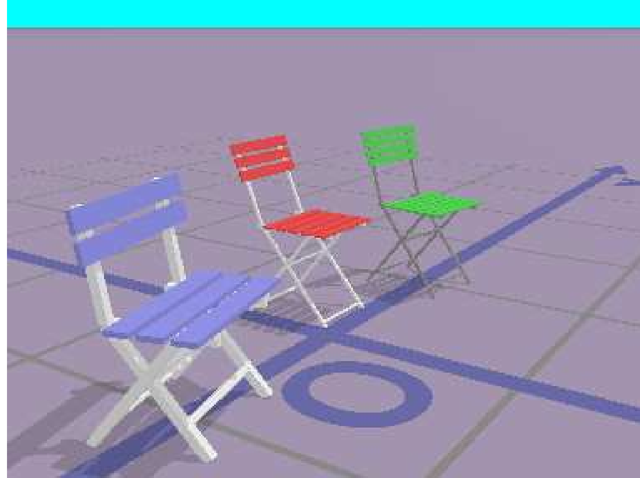

FIG. 1 – *La première image produite, avec les réglages de base.*

Pour produire la première image de synthèse (et sous réserve que POV ait été installé correctement), il suffit de se placer dans le répertoire **fabrscen**, et de taper

```
make
```

ce qui déclenchera la série d'opérations nécessaires à la production d'une image de synthèse de POV, mais aussi des quatre représentations classiques en mode "fil de fer" : vue de dessus, de face et de profil, et axonométrie. Si l'on a bien inclus le répertoire **bin** dans la variable d'environnement PATH, ces cinq images peuvent être visualisées très simplement en tapant

```
visuimag images/*
```

On peut aussi les ouvrir avec Gimp. Elles se trouvent dans le sous-répertoire **fabrscen/images**.

C'est le fichier **Makefile** qui permet d'automatiser les opérations que la commande `make` lance l'une après l'autre (ça n'a rien de particulier à cet atelier, c'est l'un des aspects les plus génialement simples de la puissance des Unix en général et de GNU/Linux en particulier).

La logique de la commande `make` est de relancer la production de certains fichiers (ici les images) lorsque le **Makefile** signale que ces fichiers sont dépendants

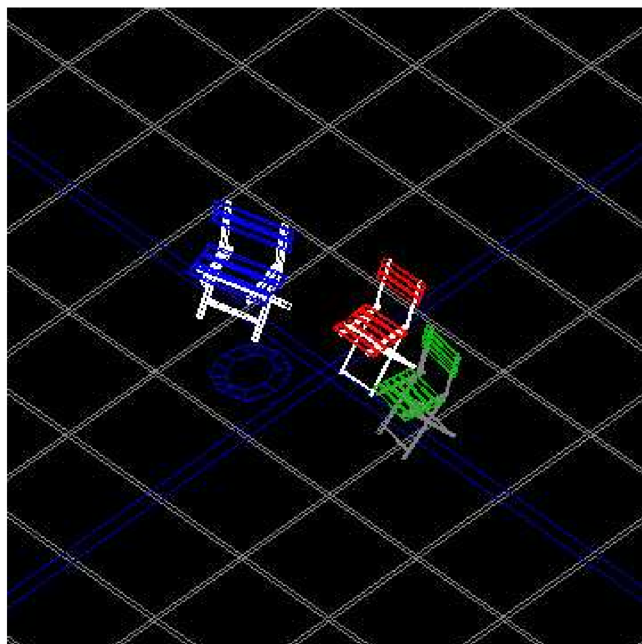

FIG. 2 – *La scène de base en perspective axonométrique.*

de certains autres. On comprendra sans trop de peine ce qui se passe en lisant le **Makefile** avec un éditeur Ascii. Une instruction de `make` occupe généralement dans le **Makefile** deux lignes dont la forme la plus courante est la suivante :

```
fichier_a_mettre_a_jour : fichier_dont_il_depend
        moulinette_de_conversion \
fichier_dont_il_depend fichier_a_mettre_a_jour
```

Nota : La mise en pages nous a contraint à couper la seconde ligne avec la barre de fraction inverse *(backslash).*

La deuxième ligne commence par une tabulation. A ce détail près, il s'agit très exactement de la ligne de commande qu'il faudrait taper pour mettre à jour **fichier_a_mettre_a_jour** dans l'hypothèse où **fichier_dont_il_depend** aurait été modifié. Cette ligne ne constitue pas forcément l'appel à un convertisseur, il peut aussi s'agir du lancement d'un script ou d'un autre programme.

La commande `make` est rendue particulièrement puissante par le fait que la mise à jour d'un fichier peut dépendre de la modification d'un ou plusieurs autres fichiers qui peuvent eux-mêmes dépendre d'un ou plusieurs autres fichiers, etc.

Dans le cas qui nous occupe, l'image de synthèse finale **(images/visu.png)** sera mise à jour si l'on modifie le descriptif de la scène (la liste des objets à représenter, incluse dans le fichier **source.cbb**), mais aussi si l'on change la position

de l'observateur (exprimée dans le fichier **reglages.ini**), et ce alors même qu'il faudra passer par une étape intermédiaire de production du fichier source de POV (**scene.pov** ; si l'on entre dans les détails, on verra que c'est encore nettement plus complexe que cela). La production des vues de dessus, face et profil passe elle aussi par plusieurs étapes intermédiaires, et tout cela est géré sans effort grâce à un **Makefile** même pas bavard.

Cela dit, les opérations automatisées par le **Makefile** ne sont pas toujours indispensables et prennent du temps, donc il peut aussi être utile de savoir lancer certaines opérations sans avoir recours à la commande `make`. En général, on utilise systématiquement `make` quand on débute, mais avec un peu de pratique on finit par préférer taper soi-même à bon escient l'une ou l'autre des lignes de commande incluses dans le **Makefile**.

La commande `make` peut tout à fait ne provoquer aucun traitement si les fichiers dont dépend la mise à jour n'ont pas été modifiés. Ce n'est pas du tout un défaut, bien au contraire, mais on peut souhaiter relancer le processus plusieurs fois de suite pour en comprendre la logique. Pour cela, il suffit d'effectuer la purge des fichiers intermédiaires par la commande

```
sh purge.sh
```

5 Logique employée par les moulinettes de cet atelier

Les moulinettes employées dans cet atelier, sauf exception, sont de l'un de ces deux types : avec ou sans fichier d'initialisation *(voir annexe)*.

La plupart n'utilisent pas de fichier d'initialisation et sont de simples convertisseurs, qui lisent un fichier (généralement Ascii) et en écrivent un autre. Leur syntaxe d'appel est alors invariablement :

```
nom_de_la_moulinette fichier_a_lire fichier_a_ecrire
```

et certaines d'entre elles acceptent en outre un paramètre optionnel `-f` (qui doit être le dernier) pour autoriser l'écrasement éventuel d'une version précédente (supposée caduque) de **fichier_a_ecrire** (faute de quoi le programme demandera l'autorisation d'écraser ce fichier).

Comme les programmes de ce type ne peuvent rien faire sans qu'on leur fournisse des noms de fichier, *l'appel de ces programmes sans argument entraîne l'affichage d'un message* d'erreur qui explique brièvement la mission du programme et sa syntaxe. Appeler ces programmes sans argument est donc un moyen d'en obtenir un *mode d'emploi sommaire* (et presque toujours suffisant).

Les autres moulinettes, plus élaborées, ont besoin de recevoir davantage d'informations, et les cherchent dans un fichier d'initialisation Ascii portant le même nom qu'eux mais d'extension **.ini**, et situé dans le répertoire courant. Par exemple, la commande

```
../bin/velizy2
```

cherchera les informations qui lui sont nécessaires dans un fichier **velizy2.ini** situé dans le répertoire courant (et non pas dans **../bin/velizy2.ini**). Au cas où un tel fichier n'existerait pas dans le répertoire courant, le programme le signalerait. Il faudrait alors placer dans le répertoire courant une copie du fichier d'initialisation-type du sous-répertoire **bin**, en tapant par exemple

```
cp ../bin/velizy2.ini .
```

et éventuellement, adapter ce fichier en l'ouvrant avec un éditeur de texte et en y inscrivant les mentions pertinentes.

La syntaxe de ces fichiers est très simple et n'appelle pas de commentaires.

Signalons toutefois qu'au cas où l'on souhaite que ce fichier d'initialisation soit modifié par un script plutôt que par une modification manuelle dans un éditeur de texte, on dispose pour cela de la moulinette **modifini** (taper `modifini` sans argument pour des explications sur sa syntaxe).

Remarque : Comme un grand nombre de ces moulinettes sont des convertisseurs, leur nom comporte souvent le chiffre "2". C'est une convention informatique très courante, qui s'explique par le fait que le chiffre 2 *(two)* se prononce en anglais de la même façon que la préposition *to,* qui se traduit par "vers". Donc, par exemple, **bmp2vlz** se lit *"bitmap to Vélizy"* et signifie "de bitmap vers Vélizy".

6 Un mot sur POV et son format

L'aspect de l'image de synthèse que l'on vient de produire est entièrement dépendant du contenu du fichier-source **scene.pov** (lui-même produit grâce au **Makefile**). Ce fichier est lui aussi une simple compilation de lignes Ascii ; il est écrit conformément à la syntaxe du programme POV. Rien n'interdit d'ouvrir ce fichier avec un éditeur de texte pour voir à quoi elle ressemble... et se rendre compte de la quantité d'instructions nécessaire à la production d'une scène pourtant assez simple : plus de cinq mille lignes truffées de signes cabalistiques !

Tout l'intérêt des moulinettes présentées dans cet atelier est de permettre d'aboutir de façon relativement simple et rapide à ces milliers de lignes complexes... sans avoir besoin d'en écrire plus que quelques-unes à la syntaxe très simple : les moulinettes de conversion font le reste.

POV (également connu sous le nom de Povray) est l'acronyme de *Persistence Of Vision.* C'est l'un des plus vieux projets de logiciel contributif : il existe depuis une vingtaine d'années et est le résultat des efforts conjugués de dizaines de programmeurs, tous bénévoles. Son code est ouvert, il est sans but lucratif depuis le départ, n'importe qui peut en faire autant de copies qu'il veut et les distribuer à qui bon lui semble... Et cependant, POV n'est pas *à strictement parler* un logiciel libre.

Il n'en serait pas moins ridicule de le confondre avec un programme commercial ou un freeware. Simplement, son développement a été entamé à une époque antérieure à la rédaction de la **licence GPL,** et il se conforme à une licence d'esprit très généreux mais qui comporte une faiblesse que la GPL a plus tard remarquée et corrigée : *il est interdit de le vendre.* Chacun a parfaitement le droit de le donner, mais personne ne peut le vendre – pas même pour le tarif symbolique qui est généralement demandé pour couvrir les frais de la diffusion des logiciels libres dans une revue ou sur un salon informatique.

De ce fait, en dépit de ses grandes qualités, POV n'est inclus dans aucune distribution Linux susceptible d'être vendue, pas même Debian, pas même Knoppix, pas même Slackware, et chacun doit faire l'effort de le télécharger et de l'installer (ce qui ne demande aucune capacité technique : il suffit de recopier un fichier unique en **/usr/bin** ou **/usr/local/bin**).

POV n'est donc pas libre à strictement parler... mais il a assurément été écrit et est bien diffusé dans l'esprit du logiciel contributif non lucratif à code ouvert, de sorte que même les linuxiens les plus intégristes peuvent l'utiliser sans déchoir. Au reste, il est en cours de réécriture pour permettre une diffusion future sous licence GPL.

Signalons enfin pour les malheureux qui ne travaillent pas sous GNU/Linux que POV existe en versions Mac et Windows.

7 Principes généraux de la 3D maillée

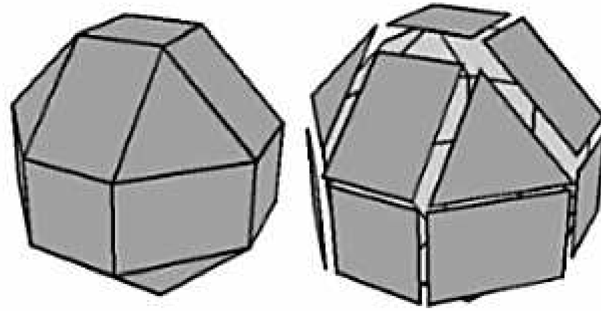

FIG. 3 – *Un volume est en fait géré comme l'assemblage des surfaces constituant son enveloppe.*

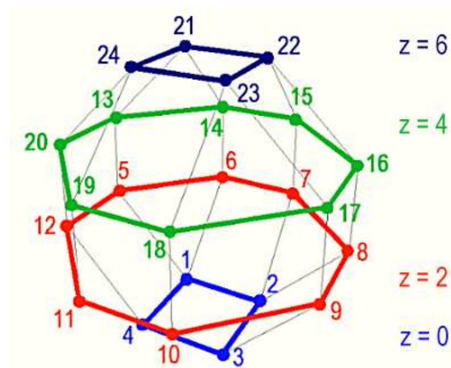

FIG. 4 – *Numérotation arbitraire des sommets d'un volume (ici, de bas en haut).*

La technique utilisée ici pour représenter des objets en 3D est connue sous le nom de **"3D maillée"** (avec faces cachées et ombres portées). Ce n'est pas tout à fait la seule technique envisageable (POV lui-même en connaît quelques autres pour représenter des objets bombés et transparents), mais c'est la plus courante, la plus simple et la plus facile à adapter à n'importe quel générateur d'images de synthèse. Elle consiste à représenter n'importe quel objet *sous la forme d'un assemblage de surfaces polygonales* (et même, en vérité, triangulaires) qui constituent des enveloppes ayant l'aspect extérieur de solides opaques.

En 3D maillée, on définit d'abord par leurs coordonnées x, y et z **les points** situés aux sommets de toutes les faces d'un objet, puis on définit **les faces** elles-mêmes en faisant l'inventaire des points constituant le périmètre de chacune des

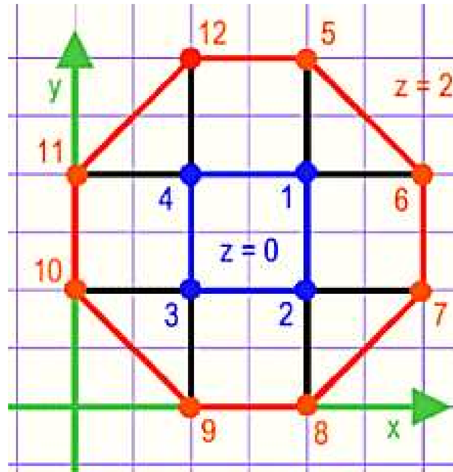

FIG. 5 – *Le même volume, en projection sur le plan horizontal.*

faces polygonales (points que l'on désigne le plus souvent par des numéros arbitraires). Par exemple, le moindre cube oblige à définir huit points (les huit sommets du cube) et six faces carrées (gérées en fait comme six fois deux triangles rectangles isocèles). Chaque face étant affectée d'une **couleur**, il ne reste plus qu'à représenter tout cela en perspective, ce qui est la mission d'un générateur d'images de synthèse comme POV.

Un fichier de 3D maillée n'est somme toute qu'*une longue liste de coordonnées* x, y et z, suivie d'*un inventaire des points* constituant le périmètre de chaque polygone affiché. On intercale toujours à des emplacements bien choisis quelques instructions pour donner des couleurs précises à ces polygones, et on ajoute couramment encore quelques lignes pour *changer l'échelle, faire pivoter ou déplacer* un ou des objets définis par une telle logique.

Le format **CBB** dont nous allons maintenant voir les caractéristiques ne fait rigoureusement rien d'autre, sauf de permettre à un descriptif d'objet d'en inclure un autre *sans qu'il soit nécessaire de le recopier intégralement* (d'où une énorme simplification de la complexité du code informatique).

8 Principes de la syntaxe CBB-VLZ

8.1 Un format hispanophone pour la 3D, francophone pour la 2D

Le **format 3D** que nous utilisons ayant été défini à *Cochabamba* en Bolivie, les fichiers écrits selon sa syntaxe portent l'extension **CBB** (il est utile de connaître ces abréviations pour comprendre à demi-mot la fonction de certaines moulinettes), et ils utilisent quelques mots espagnols : *puntos* (points), *lados* (côtés ; c'est la même étymologie que le mot "latéral"), *color* (couleur), *inclusion* (ça se prononce différemment, mais ça s'écrit comme en français), à quoi il faut ajouter le mot charabia *modifcoord* (modification de coordonnées) et les abréviations *Posi* (position), *Tama (tamaño,* c'est-à-dire taille), *Rotx, Roty et Rotz* (respectivement rotation autour des axes x, y et z). Voilà tout l'espagnol que vous aurez jamais besoin de connaître pour faire de la 3D.

Un fichier de ce type qui a subi un traitement permettant d'en éliminer les instructions d'inclusion de fichiers et de modification de coordonnées (on verra plus tard pourquoi) porte l'extension **CBD**.

Le **format 2D** que nous utilisons ayant été défini à *Vélizy* dans les Yvelines, les fichiers écrits selon sa syntaxe portent l'extension **VLZ**, et ils utilisent quelques mots français... ou à peu près : *points2D* (points), *traits, faces, couleur, inclusion,* à quoi il faut ajouter là encore le mot charabia *modifcoord* (modification de coordonnées) et les abréviations *Posi* (position), *Tail* (taille) et *Rota* (rotation).

Un fichier de ce type qui a subi un traitement permettant d'en éliminer les instructions d'inclusion de fichiers et de modification de coordonnées porte l'extension **VLD**.

CBB et **VLZ** sont des formats similaires, mais ils ne doivent pas être mélangés dans la même hiérarchie de fichiers. Si l'on veut inclure dans une scène CBB un contour plat défini en VLZ, il faudra au préalable le convertir (avec les moulinettes **vld2cbb** ou **extrude**, voir plus bas).

En CBB comme en VLZ, le séparateur décimal est le point et non la virgule... comme dans à peu près tous les fichiers informatiques. Par souci d'unification, nous respectons la même convention même dans les parties rédigées de ce document.

Les unités de mesure employées sont affaire de convention (de toute façon, des mises à l'échelle sont toujours possibles), mais il est d'usage d'*exprimer les mesures en mètres*. Les angles sont exprimés en degrés décimaux (45.5 degrés signifie 45 degrés et 30 minutes), ils peuvent être positifs ou négatifs... et même supérieurs à 360.0 si on aime se compliquer la vie.

8.2 Une ligne par instruction

En CBB comme en VLZ, il y a toujours *une ligne et une seule* pour chaque élément complet d'information (avec séparation éventuelle des sous-éléments par des virgules). Par exemple,

```
3, 2.1, 7.28, 12.444
```

désigne le point 3 dont les coordonnées sont x = 2.1, y = 7.28 et z = 12.444 (c'est donc un point en trois dimensions, dans un fichier CBB ; dans un fichier VLZ, il y aurait une coordonnée de moins).

```
1, 7, 4, 8
```

désigne un quadrilatère ayant pour coins les points 1, 7, 4 et 8 (désignés dans l'ordre d'un parcours du périmètre, et de préférence dans le sens inverse de celui des aiguilles d'une montre).

```
Rota 20.0
```

commande la rotation de 20.0 degrés (dans le sens inverse de celui des aiguilles d'une montre) d'un élément Vélizy (s'il s'agissait d'un élément Cochabamba, ce serait `Rotz 20.0`, par exemple).

```
Tama 1.2, 1.2, 1.0
```

modifie les proportions d'un objet CBB : largeur et épaisseur sont augmentées de 20 %, tandis que la hauteur (troisième valeur, donc coordonnées z) reste inchangée.

8.3 Balises ouvrantes et fermantes

Pour éviter les confusions entre les différents types d'informations, les lignes dont nous venons de parler sont encadrées par des *lignes de balises :* on écrit d'abord une ligne pour la balise ouvrante, on fait suivre un inventaire de lignes d'informations, et on termine par une ligne avec une balise fermante. Cette syntaxe de balises est très classique car employée par les différents formats *SGML* (HTML ou XML notamment) : *la balise ouvrante est encadrée par les signes inférieur et supérieur,* et la balise fermante respecte la même syntaxe, mais *intercale une barre de fraction entre le signe inférieur et le nom de la balise*. Tout cela a l'air bien compliqué, et ce sera beaucoup plus clair avec l'exemple suivant, qui définit un carré jaune à contour bleu au format VLZ :

```
<points2D>
1,  0.4, -0.2
2,  0.4,  0.3
3, -0.1,  0.3
4, -0.1, -0.2
</points2D>

<couleur>
jaune
</couleur>

<faces>
1,2,3,4
</faces>

<couleur>
bleu
</couleur>

<traits>
1,2,3,4,1
</traits>
```

Remarquons que ce que nous appelons pompeusement "inventaire de lignes d'information" peut tout à fait se réduire à une seule ligne. C'est même toujours le cas pour les couleurs.

Attention : bien que les balises des formats CBB et VLZ aient l'aspect de balises SGML, elles n'en respectent pas tout à fait l'esprit, et doivent rester sur des lignes séparées. On ne pourrait pas écrire sur la même ligne

```
<couleur>bleu</couleur>
```

ce qui serait pourtant tout à fait admis en SGML.

8.4 Hiérarchie de fichiers

Les fichiers CBB (ou VLZ) peuvent s'inclure les uns les autres. Par exemple, si l'on veut inclure une table et un tabouret dans un fichier CBB représentant une salle à manger, on écrira dans le fichier **salle_a_manger.cbb** des lignes ressemblant à celles-ci :

```
<inclusion>
table.cbb
</inclusion>

<inclusion>
tabouret.cbb
</inclusion>
```

Dans la plupart des cas, cependant, ce ne sera pas suffisant pour obtenir l'effet recherché : la table et le tabouret seront vraisemblablement placés en plein milieu de la salle à manger, alignés avec les murs comme à la parade, le tabouret étant en outre placé juste sous le milieu de la table et non devant elle. Il faudra donc modifier un peu les coordonnées de la table et du tabouret pour un positionnement plus judicieux et des modifications d'orientation des meubles rendant la scène plus crédible. C'est l'objet du point suivant.

8.5 Modification de coordonnées en deux temps

Toute modification de coordonnées (déplacement, rotation, mise à l'échelle) s'opère *en deux temps :* on la définit avant qu'elle ait lieu, et *on l'annule après* que le code informatique a passé les objets qu'elle concerne, de peur que cette modification ne se répercute au-delà de ce que l'on souhaiterait. Pour reprendre l'exemple précédent, si l'on souhaite faire pivoter de 20 degrés la table et de -35 degrés le tabouret, il ne faut pas écrire

```
<modifcoord>
Rotz 20.0
</modifcoord>
<inclusion>
table.cbb
</inclusion>

<modifcoord>
Rotz -35.0
</modifcoord>
<inclusion>
tabouret.cbb
</inclusion>
```

car alors le tabouret ne pivoterait que de -15 degrés (20 - 35 = -15). La façon correcte d'écrire cela est :

```
<modifcoord>
Rotz 20.0
</modifcoord>
<inclusion>
table.cbb
</inclusion>
<modifcoord>
Fin
</modifcoord>

<modifcoord>
Rotz -35.0
</modifcoord>
<inclusion>
tabouret.cbb
</inclusion>
<modifcoord>
Fin
</modifcoord>
```

Chaque modification de coordonnées doit donc être suivie un peu plus loin d'un groupe de trois lignes annulant cette modification :

```
<modifcoord>
Fin
</modifcoord>
```

Bien entendu, on a tout intérêt à employer le copier-coller pour mettre en place ce groupe de trois lignes, que l'on rencontre très fréquemment dans les fichiers CBB ou VLZ.

8.6 Un coup d'œil sur la scène que nous venons de produire

Ouvrez avec un éditeur de texte le fichier **source.cbb** qui a servi à produire notre image de synthèse. Vous y verrez deux objets inclus sans modification de coordonnées :

```
<inclusion>
../cbd/sol.cbd
</inclusion>
```

```
<inclusion>
../cbd/reperes.cbd
</inclusion>
```

sol.cbd est la grande surface violacée qui sert à la fois à recevoir les ombres portées et à faire apparaître une pseudo-ligne d'horizon. **reperes.cbd** est le quadrillage bleu sombre et gris (un carreau représente un mètre carré). Ces deux objets (si on peut les appeler ainsi) sont inclus sans aucune modification de coordonnées.

En revanche, les trois chaises subissent chacune des modifications de coordonnées minimes, visant à donner à la scène un aspect un peu moins mathématique. A titre d'exemple, replacez la chaise rouge (la deuxième) à son emplacement "naturel", en remettant à zéro les deux lignes de modification de coordonnées. Remplacez donc

```
<modifcoord>
Rotz 180.0
Posi 0.25,0.3,0.0
</modifcoord>
<inclusion>
../cbd/chaiserouge.cbd
</inclusion>
<modifcoord>
Fin
</modifcoord>
```

par

```
<modifcoord>
Rotz 0.0
Posi 0.0,0.0,0.0
</modifcoord>
<inclusion>
../cbd/chaiserouge.cbd
</inclusion>
<modifcoord>
Fin
</modifcoord>
```

Enregistrez la nouvelle version de **source.cbb,** quittez l'éditeur, revenez à la ligne de commande et retapez `make`. Voyez le résultat (`visuimag images/visu.png`). En faisant quelques essais du même genre, vous devriez rapidement comprendre comment on obtient le positionnement que l'on souhaite avec `modifcoord`. Mais nous reprendrons ce type de manipulations un peu plus tard. Avant de passer aux exercices pratiques, il nous reste à voir un dernier point important.

8.7 Simplification d'une hiérarchie de fichiers

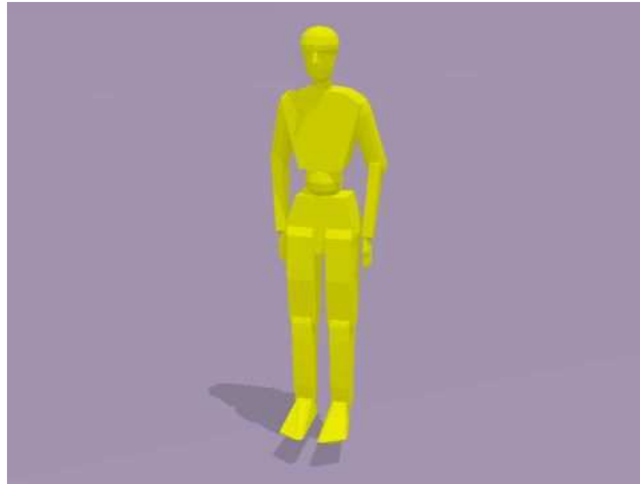

FIG. 6 – *Ce pantin articulé est constitué presque exclusivement de formes définies à main levée en deux dimensions. Ces formes ont été transformées en volumes par extrusion (comme par exemple le thorax) ou par la technique du tour de potier (comme la tête).*

Bien que cette faculté de disposer les éléments d'une scène dans une hiérarchie de fichiers, et de modifier leurs coordonnées à cette occasion, soit extrêmement pratique, elle pose des problèmes pour convertir ces données à un autre format. C'est la raison pour laquelle il est utile de pouvoir *convertir une hiérarchie de fichiers CBB (ou VLZ) en un fichier unique,* dépourvu des instructions d'inclusion de fichiers et de modification de coordonnées (ce qui suppose donc que l'ensemble des coordonnées soient recalculées en fonction de toutes les instructions de modification qui pourront se rencontrer dans la hiérarchie).

Par convention, on donne au fichier produit par cette simplification l'extension **CBD** (ou **VLD**, respectivement). On peut considérer à titre mnémotechnique que ce D final signifie "dépourvu d'instruction de modification de coordonnées et d'inclusion de sous-fichiers" (on a eu raison d'abréger !).

Le programme de simplification s'appelle **syntcbb** (ou **syntvlz**, respectivement). Il doit être paramétré par un fichier d'initialisation **syntcbb.ini** (ou **syntvlz.ini**, respectivement), placé dans le répertoire courant. Voici les deux lignes les plus importantes d'un fichier **syntcbb.ini** classique :

```
Source = source.cbb
Resultat = nom_de_l_objet.cbd
```

Tous les objets du sous-répertoire **cbd** ont été produits de cette façon, et aucun d'entre eux ne fait donc appel à des sous-fichiers. Par exemple, le fichier **cbd/pantin_garde_a_vous.cbd**, qui comprend plus de mille trois cents lignes, est le résultat de la synthèse des trente-quatre fichiers CBB hiérarchisés du sous-répertoire **pantin**, tous dépendants du fichier **pantin/source.cbb**. On pourra à titre d'exercice recommencer la production de ce fichier de synthèse, et jeter un œil sur le fichier **syntcbb.log** pour avoir une idée de la complexité des opérations réalisées au moment de cette synthèse.

9 Quelques manipulations sur la scène de base

9.1 Logique générale

Avant de nous attaquer au gros morceau (la définition même des formes en 3D), nous allons un peu nous familiariser avec le format CBB et la production des fichiers graphiques en apportant des modifications minimes à la scène déjà utilisée pour produire une image. Il ne faut pas avoir peur de l'endommager : dans le pire des cas, il suffira de repartir du fichier **.tgz** initial pour retrouver la configuration de départ.

Il est très facile de tester l'effet de modifications minimes des fichiers définissant l'image dans le dossier **fabrscen**, puisque après chacune de ces modifications, il suffira de taper `make` pour se rendre compte du résultat (éventuellement, faire suivre de `visuimag images/*` pour voir chacune des cinq images produites). Cependant, cela demandera à chaque fois un léger délai que l'on peut juger excessif. Il faut bien être conscient qu'en situation réelle, ces délais seraient considérablement réduits pour trois raisons : on ne travaillerait généralement pas sur l'ensemble d'une scène, mais seulement sur un objet précis (celui qu'on est en train d'essayer de modéliser) ; de toute façon, on ne recalculerait pas à chaque fois les cinq images, on se contenterait la plupart du temps d'une seule vue en perspective "fil de fer" (produite beaucoup plus rapidement qu'une image de synthèse ; voir plus bas) ; enfin, avec un peu de pratique, on écrit le code CBB avec très peu d'erreurs, et l'on juge donc les vérifications visuelles superflues pour les opérations simples comme le positionnement d'un objet.

Il faut garder à l'esprit qu'*il est très facile de faire disparaître involontairement un objet de la scène* si l'on inscrit des paramètres erronés dans les modifications de coordonnées : on peut facilement déplacer l'objet trop bas (et donc *sous le sol,* qui le masquera), hors du champ de la caméra virtuelle (derrière l'observateur, à côté ou au-dessus de lui), ou encore rendre son épaisseur nulle en oubliant l'un des trois paramètres de la commande `Tama` (mise à l'échelle) : `Tama 0.5` ne sera pas interprété comme une multiplication de toutes les dimensions par 0.5, mais considéré comme équivalent à `Tama 0.5, 0.0, 0.0`... donc toutes les coordonnées y et z seront annulées, et l'objet n'aura plus ni hauteur ni épaisseur – ce qui revient à le transformer en un segment de droite sans épaisseur, et donc à le faire disparaître.

Un peu plus complexe, il faut prendre en compte le fait que les rotations se font *autour du point O* et que par conséquent, si l'objet en a déjà été éloigné, une rotation même faible peut le déplacer fort loin alors qu'on pensait à tort qu'il allait pivoter sur lui-même : c'est l'erreur de débutant la plus classique.

9.2 Inclusion de nouveaux objets

A titre d'exercice, on peut essayer de remplacer les mentions `chaiseverte.cbd`, `chaisebleue.cbd` et `chaiserouge.cbd` du fichier **source.cbb** par les noms de trois autres objets de taille comparable : `biblio.cbd, fauteuil.cbd` et `voiture.cbd` (qui a bien l'encombrement d'une voiture, mais dont la forme est simplifiée à l'extrême : cela permettra de la placer assez loin du point O sans qu'elle devienne ridiculement petite).

9.3 Rotations

Garder à l'esprit que les rotations ont toujours lieu *autour du point O* et non de l'axe vertical de l'objet, qui peut déjà avoir été déplacé. Dans la plupart des cas, une rotation s'opère autour d'un axe vertical (commande `Rotz`), mais à titre d'exercice on peut aussi essayer `Rotx` et `Roty` (attention à ne pas faire disparaître l'objet sous le sol !).

9.4 Déplacements

Les déplacements, effectués avec la commande `Posi`, ne posent pas de problème particulier. Garder à l'esprit que les valeurs sont normalement exprimées en mètres (à moins qu'il y ait déjà eu une mise à l'échelle dans la hiérarchie des modifications de coordonnées).

9.5 Mises à l'échelle

Attention, une mise à l'échelle (commande `Tama`) requiert toujours trois valeurs : facteur de multiplication des coordonnées X, Y et Z. Le plus souvent, ces trois paramètres sont égaux en valeur absolue afin de conserver les proportions de l'objet, mais ils peuvent être affectés du signe moins pour engendrer des symétries.

Noter qu'une instruction `Tama 2.0, 2.0, 2.0` appliquée à un objet placé en 1.2, 2.5, 0.0 n'aura pas seulement pour effet de doubler la taille de l'objet, mais aussi... de le déplacer en 2.4, 5.0, 0.0 : une mise à l'échelle n'est rien d'autre qu'une multiplication de toutes les coordonnées.

9.6 Modification des visualisations en projection

Toutes ces modifications de coordonnées peuvent conduire à placer l'objet au-delà de la zone affichée dans les vues de face, dessus et profil et dans la projection axonométrique. Cela peut donc nécessiter des réglages par une modification du

fichier Ascii **reglages.ini**. Pour ces quatre vues, les paramètres `xObs, yObs` et `zObs` sont indifférents (ils ne servent que pour les vues en perspective). En revanche, le point virtuel de coordonnées `xDest, yDest` et `zDest` est toujours placé au centre de chacune de ces quatres vues (ainsi d'ailleurs que de la vue en perspective). Le paramètre `largeur` peut être modifié pour permettre d'élargir la vue ou au contraire d'agrandir les représentations. Enfin, le paramètre `unite` (fixé au départ à 1.0) correspond au quadrillage affiché sur les vues de face et de profil (sur la vue de dessus, ce quadrillage risque d'être masqué par la représentation de la grille **reperes.cbd**, qui est un véritable objet en 3D).

9.7 Modification de la visualisation en perspective

Pour régler le cadrage de l'image de synthèse POV, il est théoriquement possible de modifier les paramètres du fichier **reglages.ini** (notamment le paramètre `angle`, qui correspond au facteur de zoom : 45.0 désigne une focale normale, 30.0 donne une vision grand-angulaire, 25.0 correspond à l'usage d'un téléobjectif). Mais pour des réglages plus subtils, on peut lancer la commande `ordrprsp`, qui permet un ajustement plus fin avec une interface en ligne de commande très rudimentaire, mais quand même tout à fait utilisable... et par ailleurs très facile à programmer.

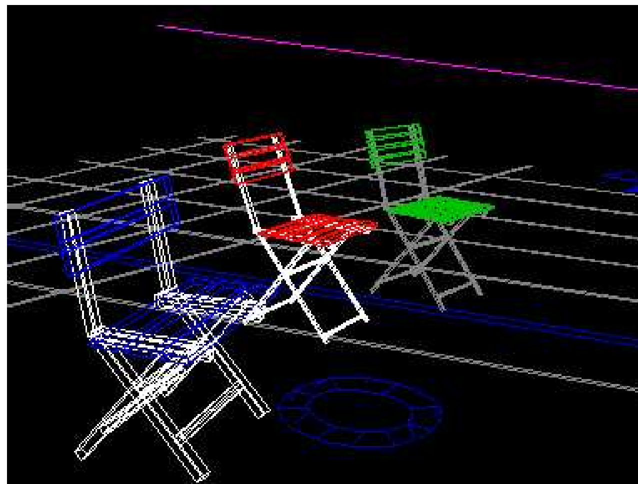

FIG. 7 – *Visualisation en perspective et en "fil de fer" avec le programme* `ordrprsp`.

Les commandes d'`ordrprsp` sont fort nombreuses mais très faciles à retenir :

– `image` fabrique (mais n'affiche pas !) un fichier **images/visu.bmp** correspondant aux paramètres actuels de visualisation en perspective ; comme cette commande fait appel à des utilitaires extérieurs, elle est sensiblement plus lente que les autres ;

– `visu` affiche (mais ne fabrique pas !) le fichier **images/visu.bmp** ; il faut bien sûr enchaîner les deux commandes : `image`, puis `visu` (on peut se demander pourquoi les deux opérations n'ont pas été enchaînées automatiquement, ce qui serait bien sûr envisageable : c'est dû à des problèmes de compatibilité avec d'autres systèmes d'exploitation que GNU/Linux, avec lesquels la commande `visu` plante le programme en raison de l'appel à quelques outils logiciels qui n'existent pas sous ces systèmes) ;

– `exporte` modifie le fichier **reglages.ini** conformément à la représentation que l'on a déterminée ; en d'autres termes, cette commande enregistre le résultat, et il suffira de taper `make` après avoir quitté `ordrprsp` pour provoquer la fabrication de l'image POV correspondant au cadrage que l'on vient de régler ;

– `quitte` permet de sortir d'`ordrprsp` (attention : si `quitte` n'est pas précédé d'`exporte`, les réglages seront perdus, et cela n'est pas signalé !) ;

– `gauche` oriente la caméra plus à gauche ; `poilgauche` oriente la caméra légèrement plus à gauche ;

– `droite` oriente la caméra plus à droite ; `poildroite` oriente la caméra légèrement plus à droite ;

– `nord, sud, est, ouest, ne, no, se, so` orientent la caméra dans la direction adéquate (rappelons qu'une direction nord-sud est parallèle à l'axe des y et non des x – attention –, et que c'est une direction est-ouest qui est parallèle à l'axe des x) ;

– `plushaut` oriente la caméra vers le haut ; `poilhaut` oriente la caméra légèrement vers le haut ;

– `plusbas` oriente la caméra vers le bas ; `poilbas` oriente la caméra légèrement vers le bas ;

– `horizontal` remet la caméra à l'horizontale ;

– `zoom` augmente la focale ; `poilzoom` augmente légèrement la focale ;

– `mooz` diminue la focale ; `poilmooz` diminue légèrement la focale ;

– `angle45` reprend une focale moyenne (correspondant à une vision à 45 degrés) ;
– `plusvite` double la valeur d'un déplacement éventuel ;
– `moinsvite` divise par deux la valeur d'un déplacement éventuel ;
– `vitesseun` redonne au déplacement éventuel la valeur d'un mètre ;
– `avance` déplace la caméra vers l'avant à l'horizontale (la valeur du déplacement étant déterminée par ailleurs) ;
– `serapproche` déplace la caméra dans la direction visée, pas forcément à l'horizontale ;
– `recule` déplace la caméra vers l'arrière à l'horizontale ;
– `seloigne` déplace la caméra à l'opposé de la direction visée, pas forcément à l'horizontale ;
– `monte` déplace la caméra vers le haut, comme dans un ascenseur, sans en changer l'orientation (on fait généralement suivre de la commande `plusbas`) ;
– `descend` déplace la caméra vers le bas sans en changer l'orientation (on fait généralement suivre de la commande `plushaut`) ;
– `crabegauche` effectue un déplacement latéral vers la gauche, sans changer l'orientation de la caméra (on fait souvent suivre par la commande `droite`) ;
– `crabedroite` effectue un déplacement latéral vers la droite, sans changer l'orientation de la caméra (on fait souvent suivre par la commande `gauche`) ;
– enfin, `pointo` replace la caméra au point O (c'est-à-dire, entre autres, au niveau du sol ! il faut faire suivre de la commande `monte`) ;
– tous les autres ordres sont ignorés (en d'autres termes, les fautes de frappe restent sans conséquence).

La bonne compréhension d'un ordre par `ordrprsp` est toujours signalée par une brève réponse du programme (généralement, elle indique la valeur actuelle d'un ou plusieurs paramètres que l'on vient de modifier). Une absence de réponse est le signe que le dernier ordre passé était incompréhensible, le plus souvent à cause d'une faute de frappe.

Si l'on a exporté (commande `exporte`) les résultats, il suffira de quitter `ordrprsp` (commande `quitte`), puis de taper `make`, pour obtenir une image de synthèse POV correspondant exactement au cadrage choisi.

L'utilisation d'`ordrprsp` présente un gros inconvénient : comme il modifie **reglages.ini** et que ce fichier sert aussi à la production des vues de face, de dessus et de profil et de la perspective axonométrique, ces quatre dernières vues ne correspondent souvent plus à grand-chose d'utile après l'emploi d'`ordrprsp`.

Mais l'utilisation d'`ordrprsp` présente l'énorme avantage de permettre d'examiner la scène sous plusieurs angles très rapidement. En fait, dans l'immense majorité des cas, au lieu d'avoir recours à `make` en cas de modification de la scène, on se contente d'enchaîner

```
sh fabrscen.sh
ordrprsp
```

ce qui fait gagner un temps fou.

Remarque : Les images en perspective produites avec `ordrprsp` suivent *chronologiquement* le déroulé de la hiérarchie de fichiers CBB : les traits représentant le dernier objet appelé chronologiquement seront toujours tracés par-dessus les traits représentant les autres objets, sans que cela ait le moindre rapport avec la perspective (ce n'est pas une représentation en faces cachées). On aura donc parfois l'impression qu'un objet éloigné se trouve au premier plan, s'il a été appelé parmi les derniers dans la hiérarchie CBB. Il suffira de taper `make` pour se rendre compte que cet inconvénient n'a aucune influence sur l'image de synthèse produite au final.

10 Création de descriptifs 2D avec un éditeur bitmap

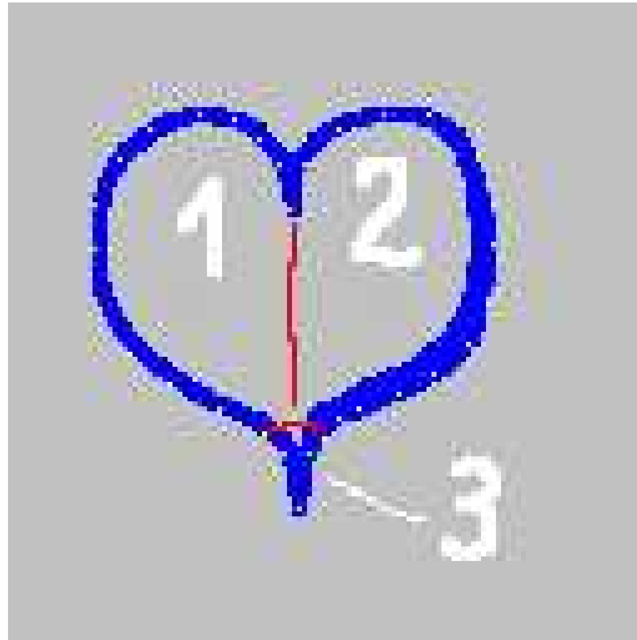

FIG. 8 – *Saisie manuelle des points caractéristiques du périmètre d'une forme en deux dimensions, à l'aide d'un logiciel de retouche d'images quelconque.*

Il est tout à fait possible d'utiliser un éditeur bitmap quelconque (en l'occurrence, nous utiliserons **Gimp**) pour élaborer des données vectorielles en 2D (qu'on pourra ensuite transformer en données 3D, voir plus bas). Le secret est tout simplement de rendre certains pixels faciles à repérer sur une image bitmap, et d'exploiter ces données avec une moulinette de conversion.

C'est la fonction du petit programme **bmp2vlz** (ses variantes **bmp2pfl** et **bmp2txtr** utilisent d'ailleurs une logique très similaire), qui est capable de repérer des pixels jaunes (parfaitement jaunes, c'est-à-dire d'une couleur dont la composante rouge est à 255, la composante verte à 255 aussi, et la composante bleue à 0) dans une image enregistrée au format BMP.

Pour placer de tels pixels avec Gimp, il faut employer l'*outil crayon* (et non le *pinceau,* qui n'appliquerait pas des couleurs pures) et la brosse la plus fine (qui a justement la forme d'un pixel, et que nous appellerons donc la *brosse pixel*). Les dimensions en pixels de l'image sont indifférentes pour **bmp2vlz**, mais les pixels seront plus faciles à mettre en place visuellement sur une très petite image (par exemple un carré de 128 pixels de côté) affichée à 400 %.

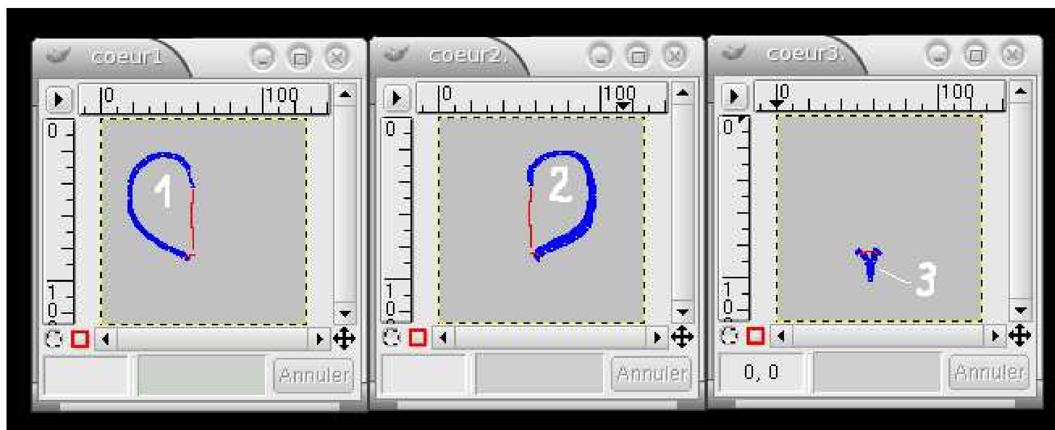

FIG. 9 – *Découpage et enregistrement en plusieurs morceaux plus faciles à gérer.*

Les pixels étant toujours organisés dans une image bitmap en lignes et en colonnes, le programme **bmp2vlz** n'a aucune peine à déterminer les coordonnées entières (la ligne et la colonne) de chaque pixel jaune, puis à convertir ces données en nombres à virgule exprimant les coordonnées x et y (l'ensemble de l'image étant supposé de centre O et de largeur 1.0).

Il faudra également placer un pixel cyan (composante rouge à 0, composante verte à 255, composante bleue à 255) à peu près au centre du périmètre défini par les pixels jaunes. La moulinette **bmp2vlz** classera les pixels jaunes dans le sens inverse de celui des aiguilles d'une montre, en fonction de leur position par rapport à ce pixel cyan, et rédigera un fichier VLZ où le pixel cyan deviendra la pointe d'une série de triangles ayant deux pixels jaunes pour autres sommets. Au total, la figure tracée aura très grossièrement l'aspect d'une tarte ou d'un fromage découpés en parts.

Dans l'hypothèse d'une forme complexe non susceptible d'être ainsi interprétée en un seul morceau comme une tarte découpée en parts, il faudra travailler sur des fractions d'image ayant des formes très grossièrement circulaires, quitte à réunir ensuite les fichiers VLZ produits dans une structure hiérarchique, avec les balises `<inclusion>`.

En dehors des pixels jaunes et du pixel cyan, les autres pixels peuvent prendre n'importe quelle valeur. On peut donc dessiner librement sur l'image pour la rendre plus compréhensible, ou encore y inscrire des numéros qu'on exploitera pour nommer d'éventuels sous-fichiers et comprendre plus facilement à quoi correspond chacun d'entre eux (**1.bmp, 2.bmp, 3.vlz**, etc.).

On peut aussi exploiter une photographie ou un document scanné (à la condition de l'avoir converti en niveaux de gris avant de le reconvertir en couleurs, pour

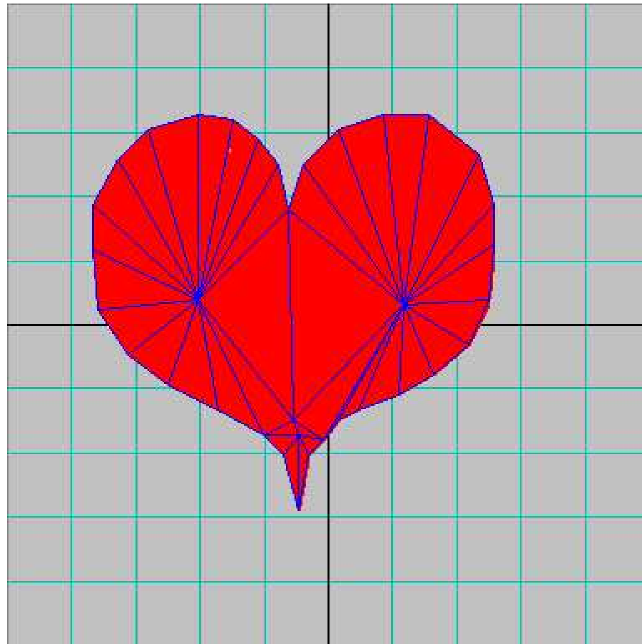

FIG. 10 – *Traduction de la forme de départ en un assemblage de triangles.*

être sûr d'avoir éliminé tous les pixels jaunes et cyan indésirables).

Récapitulons :

1) créer une image de petite taille (on peut aussi rééchantillonner une photo) ;

2) l'afficher à 400 % ;

3) choisir l'outil crayon, la brosse pixel et la couleur jaune ;

4) dessiner un périmètre de pixels jaunes très grossièrement circulaire ;

5) choisir la couleur cyan ;

6) placer un pixel cyan grossièrement au centre du périmètre de pixels jaunes ;

7) enregistrer au format bitmap ;

8) utiliser la moulinette `bmp2vlz` :

```
bmp2vlz image.bmp dessin.vlz
```

9) vérifier, intégrer dans une hiérarchie et éventuellement corriger le fichier VLZ produit grâce au programme Vélizy2 *(voir plus loin).*

11 Quelques manipulations 2D (formats VLZ-VLD)

Un fichier VLZ (ou une hiérarchie de fichiers) peut être très rapidement transformé en image bitmap par le programme **velizy2**. L'aspect de l'image (et en particulier, le cadrage de la zone affichée) peut être réglé par le paramétrage du fichier d'initialisation **velizy2.ini**, notamment ses paramètres `xCentre, yCentre, largeurReelle` (à ne pas confondre avec `LargeurPixels`) et `UniteAxes` (qui détermine le quadrillage).

Il est à noter que les images **dessus.bmp, face.bmp, profil.bmp** et **axonom.bmp**, produites automatiquement par la commande `make` dans le sous-répertoire **fabrscen/images** (ainsi d'ailleurs que **visu.bmp**), sont elles-mêmes fabriquées par ce programme Vélizy2.

Outre son utilité propre, Vélizy2 peut être employé comme *outil d'initiation à la syntaxe* des fichiers CBB, puisque VLZ et CBB utilisent absolument la même logique pour l'inclusion des sous-fichiers et les modifications de coordonnées. Une fois qu'on s'est bien entraîné sur des fichiers VLZ 2D avec Vélizy2, on perd beaucoup moins de temps en manipulations inutiles pour produire des fichiers CBB 3D avec `make` dans le répertoire **fabrscen**.

12 De la 2D à la 3D

12.1 Simple traduction

La moulinette **vld2cbb** transforme un fichier VLD (fichier 2D sans inclusion de sous-fichiers ni modification de coordonnées) en fichier CBB. Les instructions de couleur sont ignorées (le fichier CBB ne contiendra pas d'instructions de couleur... à moins qu'on ne les ajoute manuellement). L'objet produit reste une surface plane, incluse dans le plan xy, et dont toutes les coordonnées z sont égales à 0.0. Ce qui, bien sûr, n'interdit pas d'éventuels déplacements et rotations.

Pour que cette surface soit bien visible si on la plaque contre une autre (pour représenter un dessin tracé sur un mur, par exemple), il est nécessaire de l'en détacher ne fût-ce que d'une toute petite distance (un demi-millimètre suffit amplement).

12.2 Extrusion

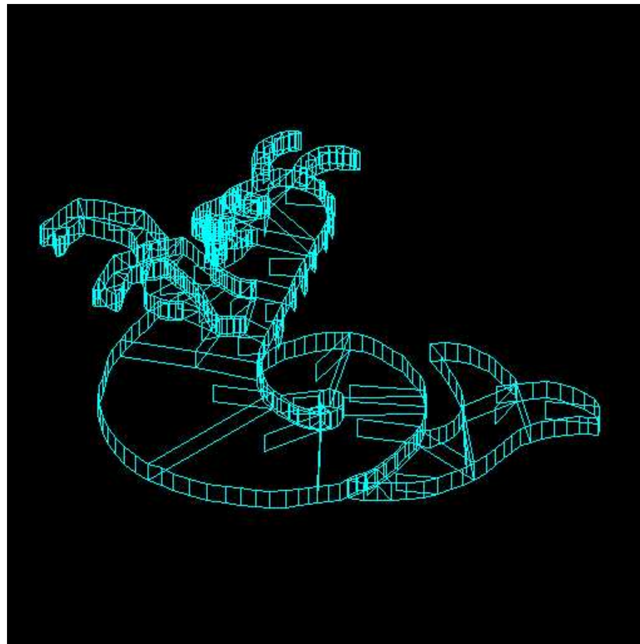

FIG. 11 – *Une forme 2D complexe à laquelle on a conféré une épaisseur par extrusion.*

La moulinette **extrude** transforme elle aussi un fichier VLD en fichier CBB, mais lui donne cette fois une épaisseur de 1.0 (ce qui est généralement beaucoup

trop : il faudra utiliser une mise à l'échelle pour imposer une autre dimension). Il est très facile avec cette moulinette de transformer des contours Vélizy en objets 3D ressemblant à des morceaux de bois découpés à la scie sauteuse.

12.3 Tour de potier

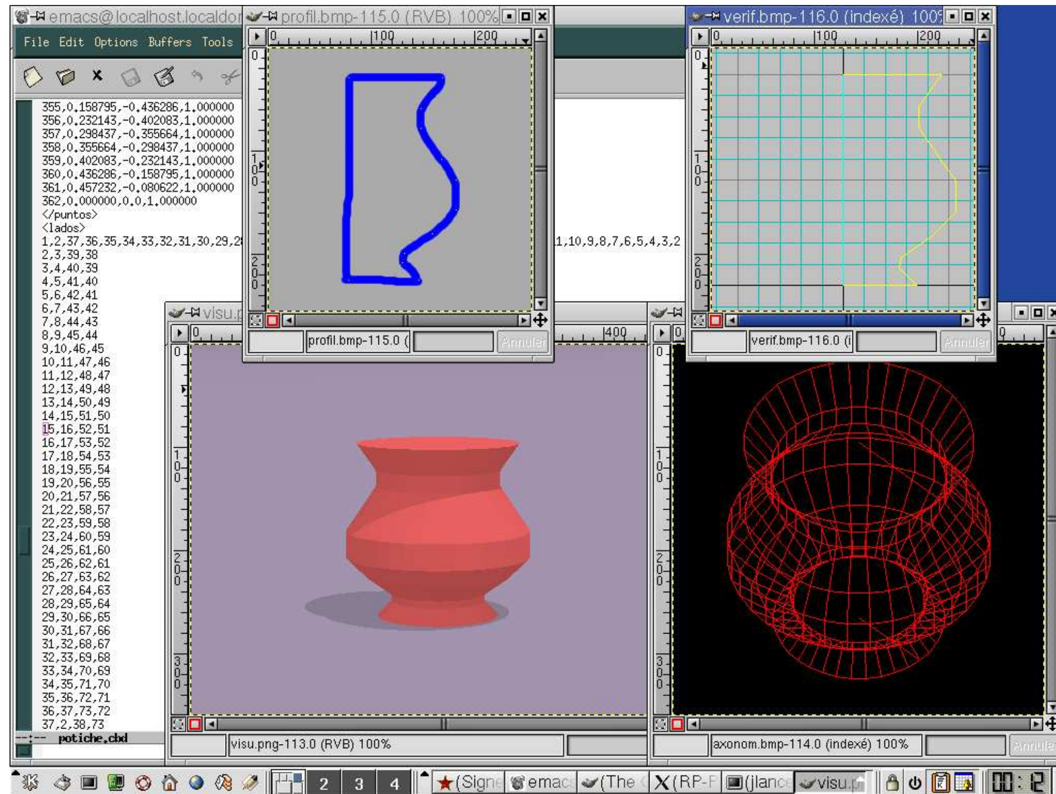

FIG. 12 – *Elaboration d'une forme en "tour de potier" à partir d'un profil 2D dessiné manuellement. En arrière-plan, le code informatique produit automatiquement.*

La moulinette **potier** transforme en objets circulaires (ayant par exemple l'aspect d'un vase ou d'une potiche) des profils définis en VLZ. La façon la plus simple de produire de tels profils VLZ est d'utiliser la moulinette **bmp2pfl**, assez semblable dans son principe à **bmp2vlz** *(voir plus haut),* à quelques détails près : l'image bitmap qu'elle exploite doit comporter non pas un, mais *deux pixels cyan, placés sur une même ligne verticale* (ou à peu près) ; cette ligne constituera l'axe central de l'objet ; d'autre part, l'objet produit aura toujours une hauteur de 1.0,

quelle que soit la taille du fichier bitmap et la position des pixels sur l'image, et il sera centré sur l'axe des z (on pourra bien sûr modifier ensuite tout cela avec des `modifcoord`). Les contours de l'objet épouseront la forme décrite par les pixels jaunes (pris en compte dans l'ordre, de bas en haut).

Bien que les objets réalisés par la technique du tour de potier soient faciles à produire et assez agréables à regarder, *il ne faut pas en abuser* car de tels objets sont très encombrants en mémoire, et une simple potiche trop bien lissée peut demander autant de points et de triangles qu'un immeuble de cinq étages – pour un intérêt visuel nettement moindre. En général, on se limitera donc à des objets non pas parfaitement ronds, mais seulement hexagonaux ou octogonaux.

On peut obtenir quelques précisions supplémentaires sur cette technique en lisant les messages affichés en cas d'appel de `bmp2pfl` et de `potier` sans argument.

13 La 3D créée par programmation

Bien que la production de fichiers 3D par programmation soit le sujet même de cet atelier, on ne donnera pratiquement aucune explication ici, puisqu'il s'agit pour chacun des participants à l'atelier d'inventer ses propres méthodes de travail pour créer ses propres objets 3D. Au cours de l'atelier, les débutants complets pourront bien sûr demander à l'auteur toutes les explications orales nécessaires.

Les éventuels lecteurs de ce texte, eux, ne peuvent pas poser de telles questions, mais ils devraient s'en sortir facilement grâce aux tout premiers chapitres d'un manuel d'initiation au langage informatique qu'ils ont adopté : des connaissances absolument élémentaires sont toujours suffisantes pour la 3D par programmation – qui est en fait l'un des domaines informatiques les plus simples à maîtriser.

L'idée générale est d'utiliser un langage pour écrire les lignes Ascii du code CBB sur la sortie standard, de façon à pouvoir récupérer ces instructions dans des fichiers de texte par une simple redirection. Par exemple :

```
./cone1.py > cone.cbb
```

```
./cone2b.pl > cone.cbb
```

Pour aider les débutants à mettre le pied à l'étrier, une série d'exemples très simples leur ont quand même été fournis. Ces exemples sont rédigés avec les deux langages de script aujourd'hui les plus répandus en environnement GNU/Linux : **Perl** et **Python**. A chacun de choisir son outil de prédilection ; l'auteur n'en préconise aucun car pour un usage aussi élémentaire on peut presque les tenir pour interchangeables. Python demandera peut-être un tout petit peu plus d'efforts, mais on peut juger sa syntaxe légèrement plus élégante et pédagogique.

On trouvera ces exemples dans le sous-répertoire **objtrond**. Il est recommandé de ne s'intéresser qu'aux exemples écrits dans un seul langage... quand bien même il faudrait le choisir à pile ou face (ou Perl, ou Python, mais surtout pas les deux). Ils sont l'un et l'autre excellents, mais le meilleur moyen de les juger inutilisables est de confondre leurs deux syntaxes !

On pourra commencer en s'attaquant à la fabrication d'objets contenant des éléments répétitifs : colonnades, marches d'escalier, etc. : la programmation est évidemment l'outil idéal pour les produire. Ensuite, l'exercice le plus classique consiste à construire des façades de maison en portant une attention particulière aux détails des fenêtres : rebord extérieur, menuiserie, volets, dont les dimensions sont intimement liées à celles de la fenêtre elle-même. Le programme sera correctement écrit s'il suffit de changer la largeur de la fenêtre pour que les volets se décalent sur les côtés, par exemple. Cela se traduira par des calculs extrêmement simples, du genre

```
xVoletGauche = ((lrgMur - lrgFenetre) / 2.0) - lrgVolet
xVoletDroit = lrgMur - ((lrgMur - lrgFenetre) / 2.0)
```

La rédaction de telles lignes est non seulement à la portée d'un débutant, mais c'est même l'un des exercices les plus simples auxquels il puisse se livrer.

Dans les scripts donnés à titre d'exemple, le but poursuivi est de représenter un simple cône, de rayon 1.0 et de hauteur 1.0, éventuellement avec un nombre de facettes paramétrable. Cela met en jeu des notions élémentaires de géométrie dont l'usage est extrêmement courant en 3D, raison pour laquelle cet exemple a été choisi.

Le rayon étant supposé égal à 1.0 et le centre confondu avec le point O, les coordonnées x et y des points sont égales au cosinus et au sinus (respectivement) de l'angle désignant la direction de ces points par rapport au centre. La seule difficulté est que la plupart des langages informatiques raisonnent sur des angles exprimés en radians et non en degrés, d'où des lignes de code de ce type (ce sont de simples règles de trois) :

```
angleRadians = angleDegres * pi / 180.0
```

ou encore

```
angleDegres = angleRadians * 180.0 / pi
```

Les concepts les plus complexes mis en jeu par les exemples Perl sont le *hachage,* l'*itération* et la *routine*.

Les concepts les plus complexes mis en jeu par les exemples Python sont à peine plus nombreux : ce sont la *classe,* le *constructeur,* l'*itération,* la *routine* et les *conversions d'un type à un autre.*

Par ailleurs, tous ces listings font appel aux syntaxes de formatage des sorties écran du type *"printf-sprintf"* (codes du type `%f, %d, %s`), qui sont d'usage courant aussi bien en Perl qu'en Python... et presque exactement semblables dans ces deux langages, ainsi d'ailleurs que dans beaucoup d'autres.

Les exemples Perl s'appellent, par degré de complexité croissant, **cone1.pl, cone1b.pl, cone2.pl, cone2b.pl**. Les exemples Python correspondants s'intitulent sans surprise **cone1.py, cone1b.py, cone2.py, cone2b.py**.

Les exemples **cone1** sont à la portée d'un débutant ayant lu les premiers chapitres de son manuel d'initiation : aucun formatage des données, déroulement purement séquentiel, et seulement deux itérations pour que l'emploi de la programmation présente un intérêt.

Les exemples **cone1b** sont presque exactement semblables, ils ajoutent seulement quelques codes de formatage des données, afin que le code CBB produit soit

plus propre. En compensation, le listing est évidemment un peu plus obscur aux yeux d'un débutant (il faudra s'y habituer !).

Les exemples **cone2** sont sensiblement plus élégants : ils utilisent des routines, avec passage d'arguments et retour de valeurs résultats. Ils restent cependant tout à fait à la portée de grands débutants.

Enfin, les exemples **cone2b** sont donnés par acquit de conscience, mais inutiles pour l'apprentissage. Ils constituent seulement une variante légèrement plus orthodoxe (quoique un peu plus complexe) quant au respect scrupuleux de la syntaxe Cochabamba (tous les points décrivant le périmètre d'une face sont énumérés dans le sens inverse de celui des aiguilles d'une montre, ce qui n'était pas le cas dans les autres exemples... mais est de toute façon indifférent pour produire au final des fichiers POV).

Le répertoire **objtrond** comporte en outre deux scripts Perl permettant de produire des cylindres et des sphères. Ils sont inclus dans le répertoire **objtrond** par souci d'exhaustivité, mais ils n'apprendront rien de plus à un débutant que les exemples précédemment cités : ils emploient rigoureusement la même logique et les mêmes méthodes, et sont simplement un peu plus complexes tout en créant un nombre de points un peu plus impressionnant.

Rambouillet, novembre 2004.

14 Annexes

14.1 Extensions

aucune extension

Au moins dans cet atelier, les fichiers sans extension sont tous des exécutables (généralement, il s'agit de moulinettes écrites en langage C).

.bmp

Image bitmap, non compressée. Gimp peut employer le format BMP pour stocker des images en millions de couleurs (trois octets par pixel), et c'est ce qu'il faut faire pour employer les moulinettes **bmp2vlz, bmp2pfl** et **bmp2txtr**. Par ailleurs, la moulinette **velizy2** et ses variantes employées ici produisent des fichiers BMP en 16 couleurs seulement (deux pixels par octet)... ce qui est moins beau mais six fois plus rapide et moins encombrant.

Les images BMP peuvent être affichées par **visuimag**, qu'elles soient en mode seize couleurs ou millions de couleurs.

Pour l'archivage éventuel, il est recommandé d'employer plutôt le format GIF, beaucoup moins encombrant... mais attention à bien régler les paramètres de la conversion si l'on ne veut pas risquer d'endommager les pixels jaunes et cyan. La moulinette **visuimag** peut aussi afficher le format GIF.

.cbb

Descriptif Ascii d'objet 3D (polygones de couleur exclusivement). Un fichier CBB fait couramment appel à des sous-fichiers du même type.

CBB est l'abréviation de "Cochabamba".

.cbd

Descriptif Ascii d'objet 3D, dépourvu d'instruction de modifications de coordonnées et d'inclusion de sous-fichiers (ce qui facilite les conversions). Un fichier CBD est presque toujours le produit de la synthèse d'une hiérarchie de fichiers CBB (synthèse réalisée avec la moulinette **syntcbb**).

.gif

Format d'image compressé utilisant un nombre limité de couleurs (on dit aussi une *palette) :* 16 ou 256 couleurs le plus souvent. On peut avoir recours à ce format pour archiver les images bitmap employées par les moulinettes **bmp2vlz, bmp2pfl** et **bmp2txtr**... au prix de certaines précautions car il ne conserve pas forcément les couleurs jaune et cyan nécessaires à l'emploi de ces moulinettes. GIF peut servir sans problème pour archiver les projections ou perspectives en mode "fil de fer", ou n'importe quelle image produite par **velizy2**.

Les images GIF peuvent être affichées par **visuimag**.

.ini

Fichier Ascii d'initialisation, généralement utilisé par une moulinette portant le même nom. C'est presque toujours un simple inventaire de paramètres de la forme

```
nom_du_paramètre = valeur_du_parametre
```

De tels fichiers sont voués à être modifiés manuellement avec un éditeur de texte pour adapter à des besoins précis les résultats du programme auquel ils correspondent.

.jpg

Format d'image compressé en millions de couleurs. POV peut l'utiliser pour représenter des objets texturés, et la moulinette **bmp2txtr** est écrite dans cet esprit.

Les images JPG peuvent être affichées par **visuimag**.

.log

Compte rendu Ascii créé automatiquement par certaines moulinettes. Les comptes rendus de **syntcbb** et **syntvlz** (**syntcbb.log** et **syntvlz.log**, respectivement) peuvent aider à débusquer les erreurs dans les imbrications d'inclusions de fichiers et de modifications de coordonnées.

.pl

Script Perl (en Ascii).

.py

Script Python (en Ascii).

.png

Image bitmap compressée. C'est le format produit par POV. Gimp peut être employé pour une conversion au format Jpeg, d'usage plus courant.

Les images PNG peuvent être affichées par **visuimag**.

.pov

Fichier source de POV (en Ascii). Comme il est généralement très bavard, il est recommandé de le compresser si on tient à l'archiver.

POV est l'abréviation de *Persistence Of Vision.*

.sh

Script Bash (en Ascii).

.vld

Descriptif Ascii d'objet 2D, dépourvu d'instruction de modifications de coordonnées et d'inclusion de sous-fichiers (ce qui facilite les conversions). Un fichier VLD est presque toujours le produit de la synthèse d'une hiérarchie de fichiers VLZ (synthèse réalisée avec la moulinette **syntvlz**).

.vlz

Descriptif Ascii d'objet 2D (polygones ou traits droits). Un fichier VLZ fait couramment appel à des sous-fichiers du même type.

VLZ est l'abréviation de "Vélizy".

14.2 Exécutables (moulinettes)

ajoutep

Cette moulinette n'a a priori rien à voir avec la 3D, elle sert à encadrer des lignes de texte entre les balises HTML `<p>` et `</p>`. Cependant, l'auteur l'utilise fréquemment en combinaison avec **dublment** pour écrire rapidement des scripts bash pour la conversion de longues séries de fichiers (au prix de quelques recherches-remplacements). Si cela ne vous évoque rien, vous pouvez oublier.

bmp2pfl

Cette moulinette exploite les pixels jaunes et cyan d'une image bitmap pour rédiger le descriptif VLZ du profil d'un objet rond, lequel profil pourra être exploité par la moulinette **potier**.

bmp2txtr

Cette moulinette exploite les pixels jaunes et cyan d'une image bitmap pour rédiger le descriptif CBB d'un polygone texturé (la texture employée ayant le même nom que l'image bitmap, mais l'extension **.jpg**).

bmp2vlz

Cette moulinette exploite les pixels jaunes et cyan d'une image bitmap pour rédiger le descriptif VLZ d'un polygone, lui-même géré comme un assemblage de triangles ayant une pointe en commun, à la façon des parts d'une tarte.

cbd2pov

Cette moulinette convertit un fichier CBD en série d'instructions de POV. Toutefois, pour que POV puisse exploiter ces instructions, il faudra au moins en ajouter quelques-unes relatives à l'éclairage et à la position d'une caméra virtuelle (ce que le **Makefile** du sous-répertoire **fabrscen** fait automatiquement).

cbd2vlz

Cette moulinette convertit les contours des polygones 3D d'un fichier CBD en série de traits 2D au format VLZ (représentation dite "fil de fer"). Elle est appelée par divers scripts du sous-répertoire **fabrscen** pour fabriquer les vues de dessus, de face et de profil et les perspectives en mode "fil de fer".

cbdpersp

Cette moulinette prend en compte la position d'un observateur et la direction dans laquelle il regarde, lit les traits de contour des polygones 3D d'un fichier CBD et rédige un descriptif VLZ permettant de les afficher en perspective si leurs deux sommets sont bien dans le champ visuel de l'observateur (autrement, le trait

de contour est simplement ignoré). L'interface textuelle **ordrprsp** n'a pratiquement pas d'autre fonction que de faciliter l'envoi des ordres à cette moulinette.

comptcbd

Cette moulinette calcule et affiche le nombre de points et de faces gérées par un fichier CBD, pour se faire une idée précise de sa complexité.

decuptxt

Cette moulinette n'est pas particulièrement conçue pour la 3D. Sa fonction est de découper un texte en sous-fichiers en se fondant sur un balisage de la forme :

```
{sous_fichier.txt}
```

(Ici les lignes de texte que l'on souhaite stocker dans sous_fichier.txt)

```
{fin sous_fichier.txt}
```

Cela peut permettre d'utiliser l'*unique* sortie standard d'un script Perl ou Python pour produire *plusieurs* fichiers CBB ou VLZ. Bien entendu, Perl et Python peuvent employer pour cela des techniques moins élémentaires, mais cette moulinette peut rendre service aux débutants. Si cela ne vous évoque rien, vous pouvez oublier.

dublment

Cette moulinette n'a a priori rien à voir avec la 3D, elle sert à dupliquer sur une seule ligne toutes les mentions d'un fichier de texte ("blabla" devient "blabla blabla"). En combinaison avec **ajoutep** et quelques recherches-remplacements, cela peut faciliter l'écriture de scripts de conversion d'une série de fichiers. Si cela ne vous évoque rien, vous pouvez oublier.

extrude

Cette moulinette transforme un descriptif 2D (en VLD) en objet 3D (en CBB) d'une épaisseur de 1.0.

Attention : toutes les couleurs sont éliminées dans cette conversion.

gigogne (et gigogne.ini)

Cette moulinette n'est pas exclusivement conçue pour la 3D. Elle permet de rassembler dans un fichier Ascii unique divers textes disposés logiquement dans une structure hiérarchique, et qui s'incluent les uns dans les autres à la façon de poupées gigognes (d'où le nom de la moulinette). Le balisage à employer est le suivant :

```
<inclusion>
fichier_a_inclure.txt
</inclusion>
```

En fait, cette moulinette n'est rien d'autre qu'une simplification de **syntcbb** et **syntvlz**, permettant un traitement similaire à celui de la synthèse des fichiers CBB et VLZ pour n'importe quel type de texte. C'est grâce à cette moulinette simplifiée qu'est assemblé le fichier **scene.pov** (le seul qui puisse être exploité directement par POV).

majorpnt

Cette moulinette permet de majorer (ou d'ailleurs de minorer) les numéros des points définis dans un fichier VLD ou CBD. Elle ne sert que très occasionnellement, notamment pour faciliter la description en CBB d'un relief dont on a stocké en VLZ les courbes de niveau. Si cela ne vous évoque rien, vous pouvez oublier.

modifini

Cette moulinette permet (notamment à un script) de modifier un fichier d'initialisation sans qu'une manipulation manuelle dans un éditeur de texte soit nécessaire.

ordrprsp

Petite interface en mode texte pour faciliter le positionnement d'une caméra virtuelle. L'utilisation d'**ordrprsp** a fait l'objet d'un long développement dans ce document.

potier

Cette moulinette transforme un profil 2D (en VLD) en objet 3D (en CBB) d'une hauteur de 1.0 et ayant grossièrement l'aspect d'un objet fabriqué sur le tour d'un potier. Dans la plupart des cas, ce profil aura été produit préalablement par **bmp2pfl**.

Attention : il faut penser à compléter le fichier produit pour lui conférer une couleur.

rapontxt (et rapontxt.ini)

Cette moulinette est le symétrique de **decuptxt** : au lieu de découper un fichier Ascii en morceaux, elle rassemble en un fichier Ascii unique une série de petits fichiers, dont les noms sont énumérés dans une liste. Eventuellement (si le fichier **rapontxt.ini** est ainsi configuré), elle balise le texte de synthèse sous la forme

```
{fichier_inclus.txt}
```

(Ici les lignes recopiées du sous-fichier)

```
{fin fichier_inclus.txt}
```

Si cela ne vous évoque rien, vous pouvez oublier... Mais si cela vous évoque quelque chose, vous pouvez aussi l'utiliser avec des textes n'ayant rien à voir avec la 3D !

reglages

Cette moulinette n'est pas vouée à être appelée manuellement. Elle est utilisée par le **Makefile** du sous-répertoire **fabrscen** pour mettre à jour une série de fichiers après la modification de **reglages.ini**. En d'autres termes, vous pouvez l'oublier... mais pas la jeter !

syntcbb (et syntcbb.ini)

Cette moulinette effectue la synthèse d'une hiérarchie de fichiers CBB pour écrire un fichier CBD unique où les instructions de modification de coordonnées éventuelles auront été *prises en compte, mais effacées.* Le fichier CBD résultant est particulièrement facile à exploiter, notamment pour une conversion automatique.

syntvlz (et syntvlz.ini)

Cette moulinette effectue la synthèse d'une hiérarchie de fichiers VLZ pour écrire un fichier VLD unique où les instructions de modification de coordonnées éventuelles auront été *prises en compte, mais effacées.* Le fichier VLD résultant est particulièrement facile à exploiter, notamment pour une conversion automatique.

td101310

Cette moulinette n'a pas particulièrement été conçue pour la 3D, elle vise seulement à adapter un fichier Ascii d'Unix aux habitudes de Windows, pour résoudre une incompatibilité aussi mineure qu'exaspérante.

velizy2 (et velizy2.ini)

Cette moulinette est peut-être la seule de toute cette série qui commence à mériter le nom pompeux de programme (et c'est en tout cas la seule qu'il n'aurait pas été raisonnable d'écrire dans un langage de script). Elle transforme une hiérarchie de fichiers VLZ en une image bitmap en 16 couleurs, et la rend plus lisible en traçant à l'arrière-plan des axes repères. Plus subtile qu'elle en a l'air, elle permet de travailler à *n'importe quelle échelle,* même pour représenter à l'échelle du micron des objets ayant des dizaines de milliers de kilomètres de large. Outre son utilité propre, elle permet au débutant de s'habituer rapidement à la logique des fichiers hiérarchiques et des modifications de coordonnées (déplacement, rotations, mises à l'échelle).

vire1326

Cette moulinette n'a pas particulièrement été conçue pour la 3D, elle vise seulement à adapter un fichier Ascii de MS-DOS ou de Windows aux habitudes des Unix en général et de GNU/Linux en particulier, pour résoudre des incompatibilités aussi mineures qu'exaspérantes.

virebak

Script minuscule pour la purge rapide des exaspérantes copies de sécurité *(backups,* d'où son nom) réalisées automatiquement par certains éditeurs de texte, dont **emacs, vi** et **kwrite**.

viretab

Cette moulinette transforme toutes les tabulations d'un fichier Ascii en série de quatre espaces (ce qui peut entre autres éviter des bugs incompréhensibles lorsque l'on travaille sur un script Python qu'on n'a pas écrit soi-même).

visubmp

Cette moulinette affiche des images BMP. A priori, elle est rendue caduque par **visuimag** – mais elle est seule à fonctionner sur certaines plates-formes récalcitrantes.

visuimag

Cette moulinette aussi rustique que pratique affiche à l'écran les images BMP, GIF, JPG et PNG (et même quelques autres). Pour visualiser l'une après l'autre toutes les images d'un répertoire, on peut taper sans vergogne

```
visuimag *
```

car les fichiers non gérés sont simplement ignorés. On peut même utiliser cette moulinette pour des diaporamas rudimentaires en spécifiant une durée en secondes

comme dernier argument. Par exemple, pour un affichage de deux secondes et demie par image :

```
visuimag * 2.5
```

Cette moulinette a été écrite en *quick and dirty* avec le pied gauche d'un cul-de-jatte. Elle ne peut pas fonctionner dans un répertoire sur lequel l'utilisateur n'a pas de droits d'écriture (et donc notamment pas pour afficher des images stockées sur un CD-ROM). C'est l'une des rares de cette série qu'il serait sûrement très laborieux d'adapter à un autre système d'exploitation : elle a recours à quantité d'outils extérieurs que seul GNU/Linux fournit en standard, en particulier pour la conversion des formats graphiques qu'elle affiche.

visuppm

Cette moulinette n'est pas vouée à être appelée manuellement, mais elle est indispensable au bon fonctionnement de **visuimag**.

vld2cbb

Cette moulinette convertit en CBB les polygones stockés dans un fichier VLD. Bien que le fichier résultat soit techniquement en 3D, et donc facile à intégrer dans une hiérarchie CBB, toutes les formes qu'il décrit sont plates, horizontales et d'épaisseur nulle (et incluses dans le plan xy). Si l'on souhaite leur donner une épaisseur, il faut employer **extrude** et non cette moulinette.

Attention : toutes les couleurs sont éliminées dans une conversion par **vld2cbb**.

14.3 Les couleurs en VLZ et en CBB

Le format VLZ (2D) ne connaît que seize couleurs ainsi nommées : `rouge, jaune, vert, cyan, bleu, magenta, rouge sombre, jaune sombre, vert sombre, cyan sombre, bleu sombre, magenta sombre, noir, blanc, gris, gris sombre`.

Le format CBB (3D) connaît une infinité de couleurs, toutes définies sous la forme **Teinte-Saturation-Niveau de Gris** (on dit en anglais *Hue, Saturation, Grayscale).* La teinte est une valeur en degrés sur le cercle chromatique, comprise entre 0.0 et 360.0 (0.0 = rouge, 60.0 = jaune, 120.0 = vert, 180.0 = cyan, 240.0 = bleu, 300.0 = magenta). La saturation est une valeur comprise entre 0.0 (gris absolu) et 1.0 (couleur aussi pure que possible... au niveau de gris considéré : un rouge pur devient un rose si son niveau de gris est élevé). Le niveau de gris est une valeur comprise entre 0.0 (noir) et 1.0 (blanc).

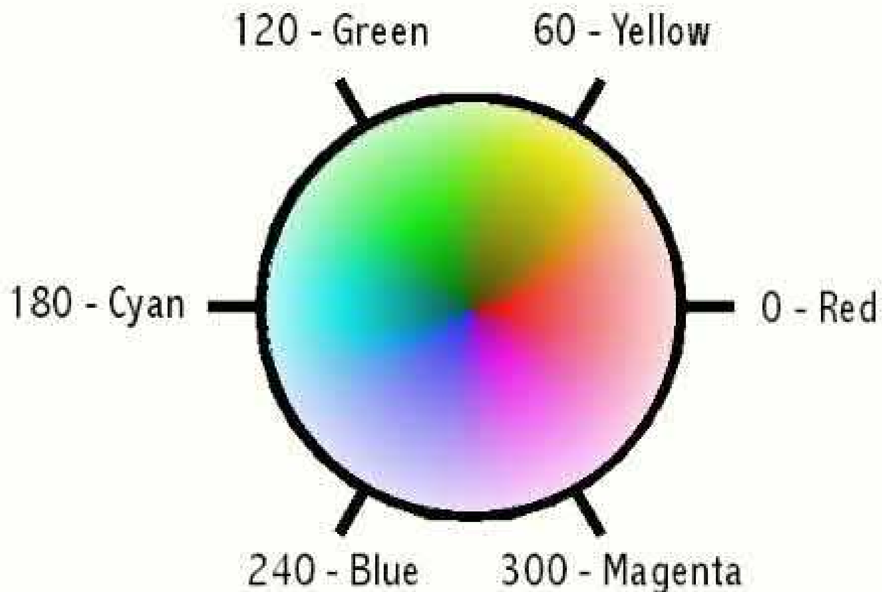

FIG. 13 – *Définition des couleurs par leur position angulaire sur le cercle chromatique.*

Par exemple, `0.0, 1.0, 0.4` désigne du rouge vif ; `60.0, 1.0, 0.85` désigne du jaune vif ; `240.0, 0.45, 0.65` désigne un bleu pastel clair ; et on pourra employer `240.0, 0.0, 0.3` pour un gris sombre. Il faut noter que sur les représentations en 3D de POV, l'éclairage et les ombres portées conduisent souvent à afficher ces couleurs avec des teintes plus sombres – ce qui est absolu-

ment normal, et même essentiel pour que la scène soit lisible sous tous les angles, mais ce qui déconcerte aussi fréquemment les débutants.

Noter que les représentations "fil de fer" d'une scène en 3D (fichiers **visu.bmp, face.bmp, profil.bmp, dessus.bmp, axonom.bmp**) sont en réalité des images VLZ à seize couleurs seulement : par conséquent les couleurs affichées en "fil de fer" sont beaucoup moins subtiles que ce qui est demandé dans la hiérarchie CBB constituant une scène et obtenu sur la représentation POV en image de synthèse. Cependant, la conversion des couleurs en mode fil de fer est toujours suffisante pour que l'image reste compréhensible.

14.4 Du texte en 3D

On a couramment besoin d'intégrer de courtes mentions textuelles dans une scène (par exemple pour y faire figurer des panneaux d'information : *"Sortie", "Attention à la marche",* etc.) Le programme **textvlz2** du sous-répertoire **textface** peut transformer à cette fin une mention textuelle en un assemblage de polygones peu esthétique, mais suffisamment lisible et peu gourmand en mémoire (et aussi, accessoirement, libre des droits qui affectent la plupart des polices de bonne qualité typographique). Le fichier informatique produit est au format VLZ (et même VLD, puisqu'il ne comporte ni modification de coordonnées, ni inclusion de sous-fichiers), et il ne peut donc être inclus dans une scène 3D avant d'avoir été converti avec **vld2cbb**, ou encore avec **extrude** si l'on souhaite lui donner une épaisseur.

La syntaxe d'appel est relativement simple :

```
./textvlz2 "SORTIE"
```

crée un fichier **texte.vlz** dessinant les lettres du mot *SORTIE*. Par défaut, le point O est situé en bas à gauche de la première lettre, mais on peut aussi taper

```
./textvlz2 "SORTIE" -c
```

pour que le bas du texte (en termes typographiques, la "ligne de base") soit centrée sur le point O.

La largeur d'un tel texte est arbitraire, et généralement supérieure à plusieurs unités. On peut imposer une largeur de 1.0 (qui facilitera ensuite les modifications de coordonnées, puisqu'il suffira d'imposer la largeur désirée comme facteur de mise à l'échelle, sans avoir à tâtonner ni à effectuer de règle de trois). Cela se fait avec la syntaxe :

```
./textvlz2 "SORTIE" -1
```

ou encore, pour obtenir un texte compris entre x = -0.5 et x = 0.5

```
./textvlz2 "SORTIE" -1c
```

Enfin, on peut obtenir un texte disposé sur plusieurs lignes en intercalant le code traditionnel de retour à la ligne (la lettre n précédée de la barre de fraction inverse). Par exemple :

```
./textvlz2 "Il est dangereux\nde se pencher au dehors"
```

Dans l'hypothèse probable où l'on voudrait plaquer une telle mention textuelle sur un fond de couleur dans une scène 3D, il ne faudrait pas oublier de l'en détacher ne fût-ce que d'un demi-millimètre, pour éviter des bizarreries d'affichage.

Signalons aux curieux que les nombreux petits fichiers VLZ du répertoire **textface** sont eux-mêmes des fichiers Vélizy. Il serait donc relativement facile de modifier le dessin de ces lettres ou d'en créer de nouvelles (le signe de l'euro, par exemple), mais c'est une autre histoire (il faudrait aussi modifier le fichier **chasses.txt**, et peut-être même **textvlz2.ini**).

14.5 Des textures en 3D

Pour faire simple, une texture n'est rien d'autre qu'une photographie informatisée (généralement enregistrée au format Jpeg) représentée dans un espace en trois dimensions (au besoin en lui donnant la taille d'une affiche). Comme n'importe quelle image numérique, la texture est fondamentalement plate et rectangulaire... et le problème est de la représenter en 3D sur un relief qui n'est généralement pas plat ni rectangulaire ! C'est un problème très similaire à celui que l'on rencontre lorsque l'on colle du papier peint sur des angles de murs : il peut être nécessaire de plier le papier, voire d'en découper certains morceaux pour les jeter purement et simplement.

La question des textures est très vaste. La 3D use et même abuse des textures pour donner un aspect plus réaliste aux scènes qu'elle produit, avec des techniques qui ont souvent plus à voir avec la retouche d'images qu'avec la géométrie. Nous ne pouvons pas aborder ici en détail toutes ces questions. Cependant, les notions mises en jeu sont tellement similaires à celles que nous venons de voir, et surtout la technique de repérage de pixels précis sur une image bitmap (que nous avons décrite pour expliquer l'emploi des moulinettes **bmp2vlz** et **bmp2pfl**) simplifie si radicalement cette question complexe qu'il serait dommage de ne pas en toucher un mot.

Nous savons déjà décrire un relief quelconque en indiquant les coordonnées 3D des sommets des polygones constituant son enveloppe. Pour employer facilement des textures, le secret est de définir sur l'image en 2D un nombre de points correspondant, auxquels on donnera les mêmes numéros d'ordre. Quelques moulinettes feront le reste en affichant en perspective des triangles ou des polygones découpés dans cette image, et en leur faisant éventuellement subir des déplacements, des mises à l'échelle ou des rotations, exactement de la même façon que pour des formes non texturées, comme nous l'avons déjà vu.

Les coordonnées définies sur la texture sont bien sûr exprimées en deux dimensions et non en trois (par convention, on nomme généralement ces coordonnées u et v pour ne pas les confondre avec x et y), mais il y a une autre différence importante : les unités employées ne désignent pas des mètres ni d'autres unités de longueur, mais un positionnement proportionnel sur l'image. Le point de coordonnées u = 0.0 et v = 0.0 désigne le coin inférieur gauche de l'image, le point de coordonnées u = 1.0 et v = 0.0 désigne le coin inférieur droit, le point de coordonnées u = 0.5 et v = 0.5 désigne le centre de l'image, et ainsi de suite, sans considérer le moins du monde si l'image a la forme d'un carré ou d'un rectangle horizontal ou vertical.

Cela étant dit, la syntaxe CBB pour l'emploi de textures n'a plus rien pour surprendre. Il suffit de savoir que les balises (hispanophones, évidemment !) `<textura>` et `</textura>` sont inscrites pour encadrer le nom du fichier Jpeg employé

comme texture, et que les balises `<puntosText>` et `</puntosText>` encadrent les coordonnées *u* et *v* des points définis sur l'image.

Voici à titre d'exemple le code nécessaire à la représentation d'une affiche verticale de 1.2 m de largeur et de 1.8 m de hauteur, disposée à partir du niveau du sol dans le plan xz à la verticale du point O (cet exemple type peut être facilement adapté pour intégrer n'importe quelle photographie dans une scène 3D) :

```
<textura>
affiche.jpg
</textura>

<puntosText>
1, 0.0, 0.0
2, 1.0, 0.0
3, 1.0, 1.0
4, 0.0, 1.0
</puntosText>

<puntos>
1, -0.6, 0.0, 0.0
2,  0.6, 0.0, 0.0
3,  0.6, 0.0, 1.8
4, -0.6, 0.0, 1.8
</puntos>

<lados>
1, 2, 3, 4
</lados>
```

Ce code est certes un peu bavard (encore qu'il le soit beaucoup moins que son équivalent en POV !). Avec un peu de pratique, il est néanmoins assez facile à écrire... et encore bien plus à adapter : l'exemple que nous venons de donner est suffisant pour tous les cas où l'on souhaite seulement faire figurer une photo non recadrée dans un rectangle.

Mais une dernière moulinette, **bmp2txtr**, peut rendre cela *considérablement plus simple* encore, en permettant de découper la texture exactement de la même façon que celle que nous avons vue précédemment avec la moulinette **bmp2vlz**. Nous ne donnerons que des explications laconiques, celles que nous venons de fournir étant suffisantes pour comprendre la logique du code CBB produit automatiquement par la moulinette (et qui sera de structure strictement identique à celui que nous venons de voir).

Pour toutes les opérations que nous allons voir, il est plus simple de placer ou d'enregistrer tous les fichiers manipulés (y compris la texture elle-même) dans le répertoire de travail (**fabrscen** ou une copie de celui-ci). Nous supposerons pour l'exemple que la texture est nommée **maTexture.jpg**.

1) Ouvrir **maTexture.jpg** avec Gimp.

2) En enregistrer une copie au format BMP et sous le nom **maTexture.bmp** (le fichier Jpeg original ne doit pas être modifié, et les opérations suivantes seront effectuées sur cette copie).

3) Rééchantillonner l'image pour que sa plus petite dimension soit de 128 pixels (cela n'est pas nécessaire mais permet un positionnement plus facile des pixels jaunes et cyan).

4) Afficher l'image à 400 %.

5) Prendre l'outil crayon, la brosse pixel et la couleur jaune pure (rouge et vert à 255, bleu à 0).

6) Dessiner autour de la forme que l'on veut afficher (grossièrement circulaire pour cet exercice) un périmètre de pixels jaunes.

7) Placer au centre du périmètre grossièrement circulaire un pixel cyan unique (rouge à 0, vert et bleu à 255).

8) Enregistrer, quitter Gimp.

9) Dans le répertoire de travail, lancer la moulinette avec la syntaxe suivante :

```
bmp2txtr maTexture.bmp maTexture.cbb
```

Le fichier **maTexture.cbb** qui vient d'être produit peut être intégré dans la scène de façon classique. Sans modifications de coordonnées, il apparaîtra sous la forme d'un découpage plat et horizontal de taille proche d'un mètre, au niveau du sol, à proximité du point O. Pour éviter des bizarreries à l'affichage, il est recommandé de le décoller un peu du sol, par exemple comme ceci :

```
<modifcoord>
Posi 0.0, 0.0, 0.002
</modifcoord>
<inclusion>
maTexture.cbb
</inclusion>
<modifcoord>
Fin
</modifcoord>
```

14.6 Liens Internet

On peut se procurer **POV**, son code source et l'excellente documentation qui l'accompagne à partir de

`http://www.povray.org`

Le site de l'auteur est

`http://www.amarelia.org`

et il donne accès à... beaucoup de choses inutiles, mais aussi à quelques pages en rapport avec les sujets traités ici. On peut notamment trouver un argumentaire qui chante en français les louanges de la 3D par programmation, avec de nombreuses images de démonstration :

`http://jlancey.free.fr/arguamar/Amarhome.htm`

On peut aussi jeter un œil sur le site du club informatique bolivien (défunt depuis longtemps) pour lequel le format CBB a été créé :

`http://www.geocities.com/SiliconValley/Way/4179/VrCocha.htm`

Bien qu'il s'agisse d'un club bolivien, ces explications sont rédigées... en anglais (en mauvais anglais pour être précis). Elles sont partiellement caduques (l'auteur ne travaille plus en Quick Basic, et il a depuis cette époque supprimé du format CBB de base les balises `<cubos>, <conos>, <cilindros>` et `<esferas>`), mais la logique générale n'a absolument pas changé.

14.7 Quelques ouvrages de référence

Neil Matthew et Richard Stones (2001), *Programmation Linux,* Eyrolles, ISBN 2-212-09129-X.

Tim Parker (1999), *Linux Ressources d'experts,* CampusPress, ISBN 2-7440-0581-9.

Gérard Swinnen (2004), *Apprendre à programmer avec Python,* O' Reilly, ISBN 2-84177-294-2.

Randal L. Schwartz et Tom Christiansen (1998), *Introduction à Perl,* O' Reilly, ISBN 2-84177-041-9.

Alex Martelli (2004), *Python en concentré,* O' Reilly, ISBN 2-84177-290-X.

Jean-Pierre Couwenbergh (1998), *La synthèse d'images,* Marabout informatique, ISBN 2-501.03039-7.

Oscar Riera Ojeda et Lucas H. Guerra (1999), *Modèles virtuels d'architecture,* Evergreen (Benedikt Taschen Verlag GmbH), ISBN 3-8228-7107-9.

Brian W. Kernighan et Denis M. Ritchie (2000), *Le langage C norme ANSI,* Dunod, ISBN 2 10 005116 4.

Bernard Desgraupes (2002), *Tcl/TK Apprentissage et référence,* Vuibert, ISBN 2-7117-8679-X.

Debra Cameron et Bill Rosenblatt (1997), *Introduction à GNU Emacs,* O' Reilly, ISBN 2-84177-015-X.

Benoît Rittaud (2000), *La géométrie classique,* Le Pommier-Fayard, ISBN 2-746-50037-X.

14.8 Licence des moulinettes et de la documentation employées pour cet atelier

Les moulinettes employées dans cet atelier ne sont pas (encore ?) sous licence GPL. La principale raison en est que, jusqu'ici, personne à part l'auteur ne les a jamais employées. La planète informatique n'y perd pas grand-chose, et cet état de faits pourrait se prolonger indéfiniment sans gêner personne, tellement ces outils sont rustiques et seraient faciles à réécrire comme à perfectionner (ils en auraient d'ailleurs bien besoin !). L'auteur croit beaucoup plus à la *validité générale de sa démarche intellectuelle pour la 3D informatisée* qu'aux médiocres listings avec lesquels il a tenté de la traduire.

Par ailleurs, l'auteur souhaiterait encore améliorer quelque peu ses sources avant de les diffuser (notamment pour leur permettre de fonctionner *avec des tubes* plutôt qu'*avec des fichiers temporaires*, ce qui les ralentit considérablement).

Cela dit, une personne (masochiste) qui souhaiterait absolument obtenir *en licence GPL* ces listings dans leur très médiocre état actuel (pour les adapter à un autre système d'exploitation, par exemple, ce qui serait extrêmement facile)... n'aurait sans doute pas à insister beaucoup pour obtenir satisfaction : quelques mails polis mais insistants à `jlancey@rocketmail.com` y suffiraient probablement !

Dans l'immédiat, les exécutables employés dans cet atelier peuvent être considérés comme des freewares, et ce support écrit peut être photocopié, diffusé et amélioré (éventuellement même défiguré !) sans aucune restriction.

Table des matières

1 Un mot sur la démarche de l'auteur **2**

2 Exigences **4**
2.1 Exigences matérielles 4
2.2 Exigences logicielles 4
2.3 Connaissances requises 4

3 Installation des fichiers nécessaires à l'atelier **6**
3.1 Décompression 6
3.2 Accès direct aux moulinettes de l'atelier 6

4 Production de la première image (le rôle du Makefile) **7**

5 Logique employée par les moulinettes de cet atelier **10**

6 Un mot sur POV et son format **12**

7 Principes généraux de la 3D maillée **13**

8 Principes de la syntaxe CBB-VLZ **15**
8.1 Un format hispanophone pour la 3D, francophone pour la 2D . . . 15
8.2 Une ligne par instruction 16
8.3 Balises ouvrantes et fermantes 16
8.4 Hiérarchie de fichiers 17
8.5 Modification de coordonnées en deux temps 18
8.6 Un coup d'œil sur la scène que nous venons de produire 19
8.7 Simplification d'une hiérarchie de fichiers 21

9 Quelques manipulations sur la scène de base **23**
9.1 Logique générale 23
9.2 Inclusion de nouveaux objets 24
9.3 Rotations 24
9.4 Déplacements 24
9.5 Mises à l'échelle 24
9.6 Modification des visualisations en projection 24
9.7 Modification de la visualisation en perspective 25

10 Création de descriptifs 2D avec un éditeur bitmap **29**

11 Quelques manipulations 2D (formats VLZ-VLD) **32**

12 De la 2D à la 3D **33**
12.1 Simple traduction 33
12.2 Extrusion 33
12.3 Tour de potier 34

13 La 3D créée par programmation **36**

14 Annexes **39**
14.1 Extensions 39
14.2 Exécutables (moulinettes) 42
14.3 Les couleurs en VLZ et en CBB 48
14.4 Du texte en 3D 50
14.5 Des textures en 3D 52
14.6 Liens Internet 55
14.7 Quelques ouvrages de référence 56
14.8 Licence des moulinettes et de la documentation employées pour cet atelier 57