Synthèse d'images

Rendu par point d’arbre CSG

2018 / 2019

Elise HARDY
Quentin COUMES

Sommaire

I. Manuel d’Utilisation

I.1. Compilation

Le *Makefile* présent dans le dossier *code.3d/* permet entre autre de compiler les sources ainsi que la documentation :

- Pour les sources : *make*
- Pour la documentation : *make doc*
- Supprimer les fichiers objets : *make clean*
- Supprimer les fichiers objets ainsi que l’executable et la documentation : *make cleanall*
- Afficher les information du processeur et de compilation : *make info*

Les sources sont compilées en un executable *csg*, la documentation est quand à elle compilé dans le dossier *docs/html*, ouvrable avec le fichier *docs/index.html*.

I.2. Utilisation

Pour lancer le programme, utilisez la commande :

./csg [script]

La création de l’arbre de scène peut prendre un certain temps pour les scripts compliqués.
Plusieurs scrits d’exemples sont disponible dans le dossieer *code.3d/script/*.

I.3. Script

Le script utilise la notation polonaise inverse : les formes primitives sont écrites avant les opérations. Par exemple, pour l’union d’un cube et d’une sphère :

obj : cube

obj : sphere

op : union

A noter que le script n’est pas sensible à la casse, et que la présence ou non d’espace n’est pas importante.

Toutes lignes commençant par le caractère # seront interprétées comme des commentaires.

I.3.A. Primitives

L'ajout de primitives à la scène se fait grâce à une ligne :

obj : [primitive] [points]

Où *points* correspond à la résolution de l'objet, la valeur donnés étant mis au carré dans le programme (*obj : cube 200* correspond donc à un cube de 40 000 points). Cette options est facultative, la valeur par défaut de *point* étant 500.

Il existe 5 primitives :

- *Cube*
- *Sphere*
- *Cylinder*
- *Torus*
- *Cone*

I.3.B. Opérations

L'opération entre deux objets (un objet pouvant être une primitive ou le résultat d'une opération) se fait grâce à la ligne :

op : [operation]

Il existe quatre opérations :

- *Union*
- *Inter* (Intersection)
- *Sous* (Soustraction)
- *Equal*

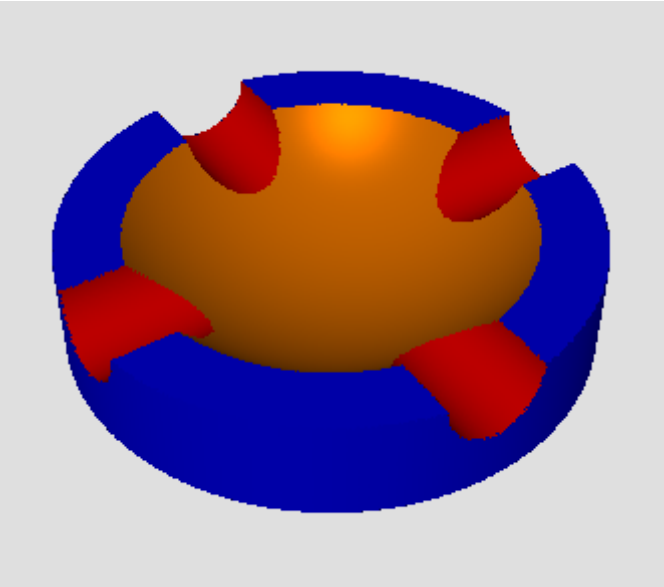
I.3.C. Transformations

Toutes transformations s'applique à la première primitive/opération au dessus.

Il est six type de transformations :

- Couleur – *col:r;g;b;a (*où chaque composante est un flottant entre 0 et 1, la couleur par défaut est rouge)
- Translation – *T:x;y;z (*où chaque composante est un flottant entre $-\infty$ et $+\infty$)
- Homothétie – *H:x;y;z (*où chaque composante est un flottant entre $-\infty$ et $+\infty$)
- Rotation sur X – *Rx : degree*
- *Rotation sur Y – Ry : degree*
- *Rotation sur Z – Rz : degree*

1.3.D. Exemple



La figure ci-dessus peut être obtenue à partir du script suivant :

```
obj: cylinder 1500
col: 0;0;1;0
H:   1;1;0.2

obj: sphere 1500
col: 1;0.5;0;0
T:   0;0;0.85

obj: cylinder 1500
H:   0.2;0.2;2
T:   0;0.2;0
Rx:  90

obj: cylinder 1500
H:   0.2;0.2;2
T:  -0.2;0;0
Ry: 90

op:  union
col: 1;0;0;0

op: union

op: sous
```

II. Architecture

II.1. Primitives

Chacune des primitives possèdes un fichier source et un header qui lui corresponds, mais elles se servent toutes de la structure communes (définit dans *object.h*) :

```
/**
 * @brief Group vertex, normal and color together, allowing less array iteration and
more efficient use of qsort().
 */
typedef struct _drawData {
    G3Xpoint vertex;        /**< Vertexes of the Object. */
    G3Xvector normal;       /**< Normals of the Object. */
    G3Xcolor color;         /**< Color of the Object. */
} DrawData;

/**
 * @brief Represents a 3D canonic Object.
 */
typedef struct _obj {
    Shape shape;             /**< Shape of the Object. */
    DrawData *drawData;      /**< Information used to draw the object. */
    bool (*pt_in)(G3Xpoint); /**< Function checking if a vertex is inside. */
    int size;                /**< Number of vertices/normals in this object. */
    int n, p;                /**< Custom parameters. */
} Object;
```

Cela permet à toutes ses formes d'être affiché grâce à la même fonction. Chaque primitive associe aussi une fonction *pt_in* indiquant si un point se trouve dans celles-ci. Cette fonction est utilisé afin de savoir qu'elle points doivent être affichés lors des opérations.

L'ensemble des primitive est créé avec une distribution aléatoire uniforme des points, qui permet un rendu plus agréable.

II.2. Opérations

Les opérations entres différentes primitives sont représenté à l'aide d'un arbre binaire : les feuilles sont les primitives et les nœuds les opérations. Les transformations peuvent aussi bien être appliqué à une feuille qu'à un nœud, celle-ci sont alors appliqué récursivement jusqu'au feuilles.

La structure utilisé pour l'arbre est :

```
/**
 * @brief CSG Tree used to represents composite Objects.
 */
typedef struct _node {
    Operator op;   /**< Operator used by this node.*/
    Object *obj;   /**< Object used by this node. */
    bool *visible; /**< Boolean array telling whether a point I should be drawn. */
    double *mi;    /**< Inverse matrix to move a point into the canonical object */
    struct _node *left, *right; /**< Sons of this node */
} Tree;
```

Lors de la création d'un nouveau nœuds, un nouvelle objet est crée, utilisant la même structure que les primitives. Ce nouvelle objet crée un nouveau tableau correspondant à la concaténation des tableaux des deux fils de ce nœud avant d'appliquer l'opération correspondante.

Le tableau de booléen associé au nœud (*visible*) est ensuite mis à jour en appelant récursivement la fonction *pt_in* sur les fils du nœud et en interprétant le résultat suivant l'opération choisie.

Une fois le tableau *visible* mis à jour, celui-ci est trié avec le tableau *drawData* de l'objet de façon à ce que tout les points visibles se retrouvent au début, et tout les points invisibles à la fin.
La partie visible du tableau est ensuite trié selon la couleur du point, de façon à minimiser le nombre d'appel à *g3x_Material()*.

Tout cela à pour but d'accélérer la fonction d'affichage des objets.

A noté qu'une fois le tableau correspondant à la concaténation des tableaux des fils est créé, celui des fils est libéré et les pointeurs redirigés sur le bonne emplacement du nouveau tableau. De cette manière, aucune données n'est dupliquée.

II.3. Transformations

Les transformations peuvent s'appliquer sur n'importe que partie de l'arbre :

- Si c'est un nœud, la transformation est propagée récursivement jusqu'aux feuilles.
- Si c'est une feuille, la transformation est appliquée directement aux points et aux normals, et la matrice correspondant à la transformation inverse est stockés.

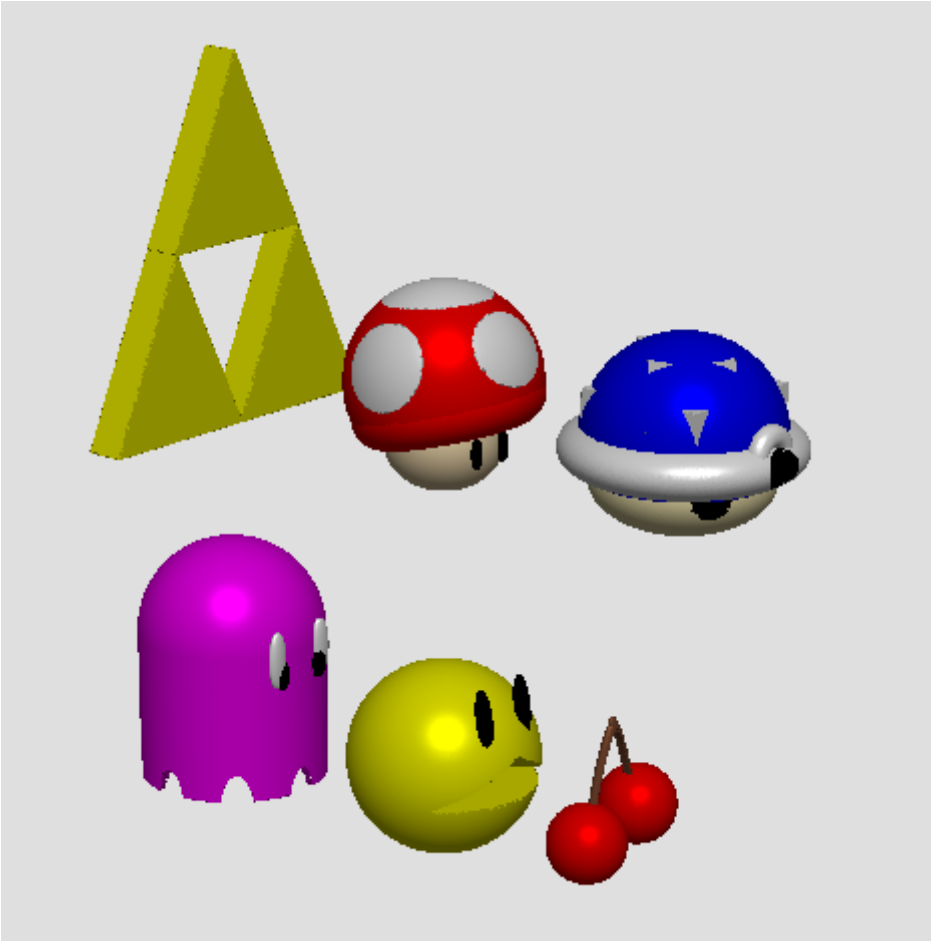
Le stockage de la matrice inverse permet de mettre un points dans le repère correspondant à l'objet canonique avant l'appel à *pt_in*.

II.4. Affichage

L'ensemble de nos points étant stocké à la racine de l'arbre, et triés de manière à ce que tout les points visibles se trouvent au début du tableau, la fonction d'affichage (*tree.c – drawNode()*) ne consite qu'à parcourir le tableau de points de la racine jusqu'à la fin de celui-ci ou la rencotnre d'un points invisible.

III. Exemples

III.1. *script/complex/game.txt*



III.2. *script/complex/scene.txt*