GLEDEL Clément
DUGNE Quentin

COMPTE-RENDU CIN

Question 1 :

Afin de rendre notre langage simple d'utilisation, nous avons décidé de le faire prendre une forme de langage de structuration balisé tel que le XML. Nous avons donc pensé à associer les balises structurant notre langage aux différents types de noeuds possible dans un graphe de scène, soit : la racine, les transformations (translation, rotation, mise à l'échelle), les groupes de transformations ou de géométries et enfin les différents types de géométries (cylindres, cubes, cônes…).

Le langage XML représentant un arbre, chaque géométrie à placer dans la scène sera située dans des balises de transformations et/ou de groupes afin d'appliquer les transformations à la géométrie en question. La description d'un graphe de scène commencera obligatoirement par l'ouverture d'une balise "<Root>" et se terminera obligatoirement par la fermeture de cette même balise "</Root>".

Lors d'une ouverture de balise autre que "Root", tel que "Transform", "Group" ou "Geometry" , les attributs nécessaires à l'initialisation d'une telle balise devront être écrits à la suite séparés par des points virgules avant la fermeture de la balise, comme montré ci-dessous :

1) <Group**;** nom = "..."**;**>
2) <Transform**;** type = Translation{x,y,z}**;** nom = "..."**;**>
3) <Geometry**;** type = Cube{size}**;** nom = "..."**;**>

La création d'une géométrie peut se faire de 2 manières différentes, comme en XML la fermeture de cette balise est faisable en faisant <Geometry; …> </Geometry>, ou encore <Geometry; … />. La deuxième manière est propre à la création d'une géométrie car elle est considérée comme une feuille de l'arbre représenté par le graphe de scène.

Exemple de structure de graph de scène :

```
<Root>
    <Transform;  … ;>
        <Transform; … ;>
            <Geometry; … ;/>
        </Transform>
    </Transform>
    <Transform;  … ;>
        <Group; … ;>
            <Geometry; … ;/>
            <Geometry; … ;/>
        </Group>
    </Transform>
</Root>
```

Question 2 :

Afin d'écrire notre grammaire sur ANTLR, nous avons tout d'abord créé plusieurs règles de bases, telles que :

- root : règle racine et principale permettant la détection de la balise "Root" et l'appel de la règle "syntaxe" présente une ou plusieurs fois dans notre description.
- syntaxe : règle permettant la détection des balises de transformation (appel de la règle : "transform"), de groupe (appel de la règle : "group"),et de géométrie (appel de la règle : "geometry"). Les balises de transformation et de groupe doivent contenir une ou plusieurs syntaxes afin de finir la structure sur des feuilles (géométries).
- transform : règle permettant la détection du type de transformation "translation", "rotation" et "scale".
- group : règle permettant la détection de la détection de création d'un groupe avec son nom associé.
- geometry : règle permettant la détection du type de géométrie "cone", "sphere", "cylinder", "cube", "teapot", "tetrahedron" et "torus".
- translation, rotation, scale : règles permettant la détection du nom du type de transformation ainsi que ses différentes coordonnées obligatoires afin d'effectuer la transformation.
- cone, sphere, cylinder, cube, teapot, tetrahedron, torus : règles permettant la détection du nom du type de géométrie ainsi que ses différents attributs obligatoires afin de créer une géométrie.

Question 3 :

Afin de construire notre graphe de scène lors de l'analyse du texte d'entrée, nous commençons par ajouter dans les @members la déclaration d'un noeud "currentNode" correspondant à la racine "Root" que l'on prend pour l'initialisation de notre graphe de scène et une pile "stack" vide. Lors de la détection de notre racine dans la règle "root" on commence par ajouter notre noeud racine dans la pile "stack".

A chaque appel de règles de transformation, de group ou de géométrie, on ajoute à notre "currentNode" un fils correspondant à un nouveau noeud créé correspondant à la règle actuelle. Si la règle appelée est une règle correspondant à une transformation ou un groupe, nous modifions le noeud courant "currentNode" par le nouveau noeud, et l'ajoutons sur la pile.

A la détection d'une balise de fermeture de transformation ou de groupe nous retirons une élément de la pile puis nous actualisons la valeur de "currentNode" au dernier élément sur la pile. Lorsque la description arrive à sa fin nous retirons le dernier élément de la pile afin de la vidé ce qui indique la fin de notre graphe de scène.

Question 4 : *(Il faut régler le projet Eclispe au préalable (importations de bibliothèques etc))*

Lorsque l'on compile notre grammaire, 4 fichiers java viennent se créer dans un dossier output. Ces fichiers représentent les tokens, le Parser, le Lexer et notre fichier test. Nous devons donc les récupérer et les glisser dans notre projet Eclipse dans un package que l'on nommera parser. Par la suite nous devons indiquer au préalable dans le Java Renderer, la racine de notre arbre. Cela se fait par l'intermédiaire de cette commande :

```
private Lib3d.SceneGraph.Group m_root = XGDParser.root;
```

Elle indique la première règle qui sera appelée et qui contiendra notre racine.

Il faut maintenant gérer la partie affichage, pour cela nous créons tout d'abord une fonction run() qui se chargera d'initialiser notre parser, notre lexer, nos tokens ainsi que notre racine dans le fichier JavaMain. Pour le lexer nous devons lui indiquer le chemin contenant notre fichier de test.

Une fois cela terminé, nous devons créer une fenêtre qui accueillera la scène de dessin. Cette scène de dessin sera créée via l'initialisation d'un canvas. Nous utilisons ensuite une méthode appelée addGlEventListener qui nous permettra d'utiliser OpenGL avec notre surface de dessin. Il ne nous reste plus qu'à relier ensemble la fenêtre et la surface avec frame.add(canvas). En ce qui concerne l'animation, cela sera géré par le display qui est appelé en continue dans la boucle while du programme. Les fonctions start() et stop() permettent de gérer l'animator (ici displayT) afin de continuer ou non l'animation.

Une fois tout cela mis en place, nous pouvons lancer le programme avec JavaMain. Celui-ci tourne à l'infini puisqu'il attend une connexion "physique" avec notre grammaire. Afin d'établir cette connexion nous allons dans antlrworks et nous faisons un Debug Remote et nous connectons notre exécution de grammaire au bon port. La grammaire se lance et une fois que le fichier test a été parcouru une fenêtre apparaît contenant notre scène.

Question 5/6 :

Nous avons rajouté l'association de la couleur et la texture à une géométrie en commençant par créer une map "mapmat" qui contiendra le nom d'un matériau associé avec son matériau. Cette map permettra de stocker le matériau afin de pouvoir le réutiliser pour plusieurs géométries sans avoir à redéfinir ses paramètres à chaque fois. Nous initialisons en même temps une map "mapi" permettant de stocker toutes les géométries selon une clef correspondant à leurs noms, ainsi lors de la création d'une géométrie nous ajoutons la géométrie dans notre map "mapi".

Nous avons décidé de traiter l'initialisation d'un matériau lors de son premier appel avec une géométrie dans le graph de scène, ainsi lors de la création d'une autre géométrie récupérant le même matériau, il y aura juste à indiquer le nom du matériau. Afin de rendre cela possible lorsque la règle d'une géométrie est appelée nous ajoutons un autre attribut (séparé par un point virgule) avant la fin de la balise d'ouverture.

Cet attribut prend un nom ainsi que ou non l'appel d'une règle "couleur" prenant en paramètre le nom du matériau. Cette règle permet de détecter les nombreux attributs nécessaires à la création d'un matériau afin de le créer ainsi que de l'ajouter à notre map de matériaux. Dans le cas ou l'utilisateur n'appel pas la règle de couleur mais indique seulement le nom du matériau alors nous cherchons si le matériau existe, si oui alors on associe le matériau à la géométrie en allant chercher dans la map "mapi" la géométrie concernée avec son nom en clef de la map. Si le matériau n'existe pas alors la géométrie n'aura donc pas de couleur ni de texture associée.

Exemple d'appel de géométrie avec définition de matériau :

<Geometry; type = Cube{size}; nom = "..."; Material : nom = "...", Ka{x,y,z}, Kd{x,y,z}, Ks{x,y,z}, d = x, Ns = x, textureScale{x,y}, textureFile = "...";>

Exemple d'appel de géométrie avec récupération d'un matériau :

<Geometry; type = Cube{size}; nom = "..."; Material : nom = "...";>

Question 7 :

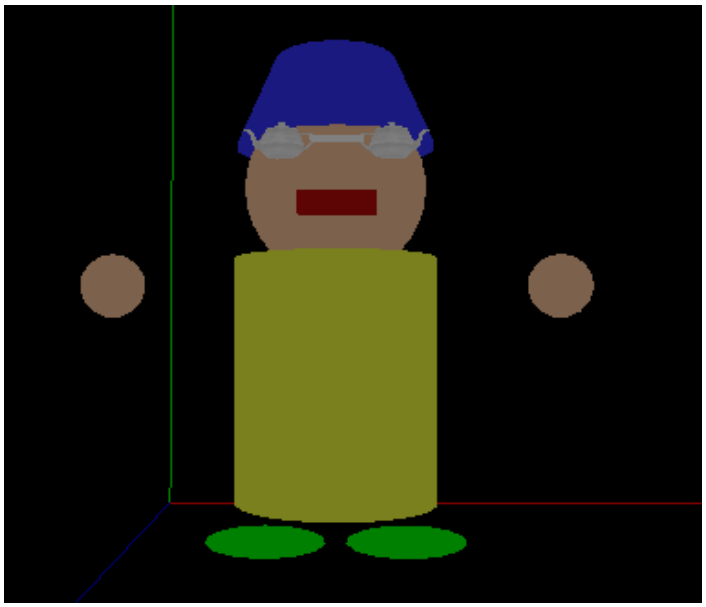
Tous nos noeuds de géométries et de groupes détiennent un nom obligatoire lors de la création de chacun. De plus, lors de la création de ces types de noeuds, nous les ajoutons à une map correspondante à leurs types ("mapi" pour les "Geometry" et "mapgroup" pour les "Group") avec

pour clef leurs noms. Afin de gérer réutilisation de ces noeuds, nous avons rajouté un moyen d'appeler ces géométries ou groupes. En définissant une géométrie ou un groupe dans le graph de scène avec seulement un nom une vérification d'existence de ce noeud est faite sur la map correspondante, s'il existe déjà alors on récupère l'existant afin d'effectuer les liaisons dans l'arbre du graph de scène.

Si l'utilisateur défini par la suite une géométrie ayant le même nom qu'une déjà existante (en définissant de nouvelles caractéristiques (sans utiliser l'appel avec seulement le nom)), l'ancienne géométrie se verra remplacée par la nouvelle.

Nous obtenons le résultat suivant avec le fichier texte test1 mis en pièce jointe. Cette exemple fait appel à plusieurs forme géométrique que l'on a modifié avec des scales, des translations, des géométries.

Par exemple en modifiant une géométrie avec un Scale nous pouvons obtenir une une forme en 2D. Les paramètres initials d'un cylindre permet d'obtenir des formes plus précises, comme dans le cas ci-dessous. La base est plus grande que le Top ainsi nous obtenons un objet ressemblant à un chapeau.



"Les Teapots-Lunettes reviennent à la mode"

Question 8/9/10:

Il est possible de définir une trajectoire aux géométries que nous avons implémentées dans la scène, pour cela nous devons prendre une expression mathématique sous forme de String au lieu d'une valeur finie en paramètre pour les transformations (ex: Translation{ 20*t,5,6}). Il nous faudra donc la calculatrice qui a été implémentée lors du TP1 afin de calculer les valeurs que renverront ces expressions.

Cependant la trajectoire se fait selon une variable de temps que l'on nommera t. Ainsi pour chaque t il va falloir calculer le résultat de l'expression, mettre la valeur correspondante dans les paramètres de la transformation, puis afficher l'évolution avec le projet éclipse. Cette étape et à faire pour toutes les transformations dépendant du temps.

Une des possibilitées serait donc de charger note grammaire avec le projet Eclipse et une valeur différente à chaque fois pour t. Puis nous devons modifier le noeud de la transformation, ainsi

lors de la boucle du display il effectue le même parcours d'arbre à l'exception que les noeud transformations ont changés de valeurs ce qui donnera un effet de déplacement. Le t sera incrémenté par un Timer et mis dans une map afin que l'on y accède lors des calculs des coordonnées de la transformation. Il nous reste donc à accéder aux valeurs renvoyées par la calculatrice depuis le projet Eclipse. Pour cela nous implémentons 3 maps, une pour chaque type de transformation. Dans chacune d'entre elle nous y insérons le type de la transformation utilisée (exemple Translation) puis la valeur qui lui est associée sous la forme d'un tableau de String. Ce tableau de String représentera les expressions de chaque coordonnées.

La syntaxe pour appeler ces transformations ne change pas énormément. L'utilisateur devra insérer des Strings au lieu de Floats/INT, l'emploie de la variable t est primordiale, tout autre variable ne sera pas reconnu:

<Transform; type = Translation{“5”,”30*t”,”12.3”}; nom = "...";>

Au cours de ce TP nous avons rédigé la grammaire associée à cette méthode, mais nous n'avons pas réussi à implémenter le code dans JavaMain, nous avions une erreur d'adressage: Address already use : JVM_BIND

(Voir le code dans JavaMain (en commentaire) et la grammaire pour plus de compréhension)

Conclusion:

Notre grammaire fonctionne correctement tout en gardant une structure solide dû à l'inspiration du langage structurant balisé de XML, et facile d'utilisation grâce aux formes similaires d'initialisation de chaque type de noeud dans la description du graphe de scène. L'animation consiste à remplacer les noeuds de transformations au cours du temps. Pour cela une calculatrice peut être utilisée pour exprimer les coordonnées des transformations; Cette animation sera réalisé par un affichage répétitif du display.