OpenGL - TD 02

Dessin, Objets canoniques et Transformations

Lors de cette séance, nous allons commencer à comprendre le mécanisme permettant de dessiner. Nous dessinerons des formes basiques. Nous verrons ainsi les notions d'initialisation et de rendu. Nous aborderons ensuite un des concepts les plus simples de la modélisation en 2D (et en 3D) : La construction d'objets canoniques et leur placement.

Exercice 01 – Premier points

Au précédent TD, nous avons mis en place la fenêtre et vu les interactions hommes machines, et nous allons pouvoir maintenant dessiner ! Dans un premier temps nous nous contenterons de dessiner des points pour rapidement passer à autre chose.

Dessiner dans des applications 3D se fait globalement en 2 temps, d'une part l'initialisation qui va permettre de définir et charger les données graphiques (les modèles 2D/3D à afficher), et d'autre part, le rendu qui se fait en permanence (dans la boucle de rendu). Afin de ne pas rentrer dans le détail de la bibliothèque OpenGL, vous exploiterez une surcouche faite par nos soins qui « cache » toute la partie technique d'OpenGL qui sera vu ultérieurement. Cette surcouche est une bibliothèque appelée *glbasimac* que vous trouverez dans le répertoire *third_party*. Vous pouvez regarder les fichiers hpp (dans le répertoire *glbasimac*) et aussi les cpp mais n'essayez pas de comprendre les fonctions propres à OpenGL. Par exemple, vous y trouverez une structure appelée `GLBI_Engine` qui nous sert de (petit) moteur de rendu. Ainsi ce moteur défini actuellement un espace de coordonnées allant de -1 à 1 (voir TD précédent).

Nous allons donc utiliser notre surcouche et en particulier une structure appelée `GLBI_Set_Of_Points` pour dessiner nos points. Nous avons déjà déclaré une variable de cette structure appelée `thePoints` (je vous laisse trouvez où). Pour simplifier, dans un premier temps, nous allons travailler directement dans la fonction `main`. Nous verrons ensuite comment faire (un peu) mieux.

Enfin, pour définir et stocker les coordonnées des points, nous utiliserons des tableaux unidimensionnel (nous utiliserons pour ce faire des `std::vector` évidement). Ces tableaux stockerons, dans l'ordre, les coordonnées de chacun des points que nous souhaiterons afficher. Nous procéderons de la même manière pour les couleurs de ces points.

A faire :

01. Dans la partie initialisation (à trouver), créez un `vector` de `float` contenant les coordonnées du point origine (donc 0.0/0.0). Puis trouvez la première fonction `initSet` de notre structure afin d'initialiser un point blanc à l'origine. Pour le moment, vous ne verrez rien apparaître à l'écran (vous n'avez fait que la première étape, l'initialisation)

02. Dans la partie rendu (à trouver), insérez les lignes suivantes :

```
glPointSize(4.0);
thePoints.drawSet();
```

La fonction `glPointSize` défini le « diamètre » de dessin du point (en nombre de pixel). La fonction `drawSet` de notre structure demande simplement à OpenGL le dessin des points. Désormais, vous devriez voir apparaître un point blanc au centre de l'écran. Par curiosité vous pouvez faire varier la taille de dessin des points. La taille de dessin des points est un des « états » de la bibliothèque OpenGL qui en comporte de nombreux.

03. Modifiez votre code d'initialisation pour dessiner des points blancs aux coordonnées suivantes : (0.5 / 0.0), (0.0 / 0.5) et (-0.5 / -0.5)

04. A l'aide de l'autre fonction d'initialisation du set de points, modifiez votre code pour dessiner le premier point en blanc, le second en rouge, le troisième en vert et le dernier en violet.

Exercice 02 – Architecture logicielle

Cet exercice consiste surtout à organiser notre fichier d'exercice. En effet, mettre toutes nos commandes dans le main n'est en général pas une bonne idée.

A faire :

01. Comme d'habitude, copier votre fichier *ex01.cpp* en *ex02.cpp*

02. Créer deux fonctions, une appelée `initScene` (ne renvoie rien et n'a pas d'argument) et l'autre appelée `renderScene` (idem) dans le fichier *ex02.cpp*. `initScene` devra être appelée après l'initialisation du moteur (où est-ce d'après vous?) et `renderScene` au sein même de la boucle de rendu.

03. Insérez vos instruction d'initialisation et de rendu dans les fonctions associées et vérifiez que le résultat reste identique à l'exercice 1.

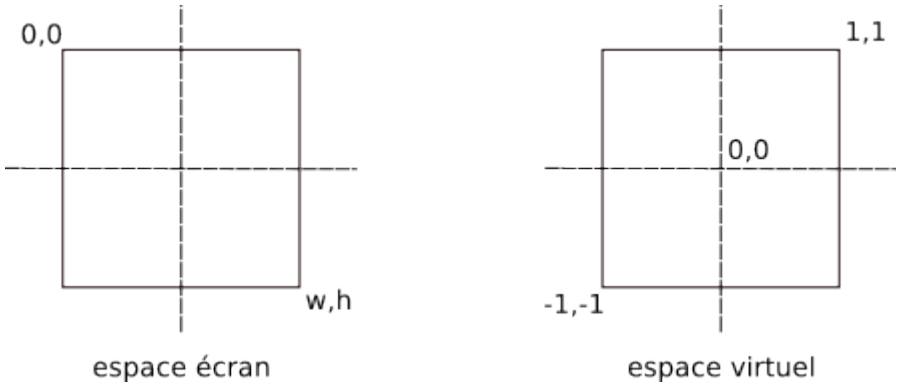
Exercice 03 – Dessin à la souris

Nous voudrions maintenant placer un sommet à chaque clic dans la fenêtre. L'objectif de cet exercice est donc de redessiner tous les points cliqués à chaque « tour » de la boucle principale. Pour cela, vous devrez enregistrer ces derniers en mémoire au moment d'un clic.

A faire :

01a. Faites en sorte que lorsque l'utilisateur clique, les coordonnées du vertex à dessiner soient stockées dans votre `GLBI_Set_Of_Points`. Pour ce faire, vous pourrez exploiter la fonction `addAPoint` de votre structure. Attention, cette fonction demande en paramètre **deux pointeurs**, qui sont les « adresses » de base de vos tableaux (`vector`). Vous pouvez demander à votre chargé de TD quel est cette notion de pointeur, mais pour convertir vos `vector` en pointeur, vous pouvez utiliser la fonction `data` des `vector`.

Ayez en tête que l'origine du repère de la fenetre GLFW est en haut à gauche de cette dernière, là ou l'origine du repère de la scène est au centre de la fenêtre. Pour cette raison, vous devrez convertir les coordonnées de l'espace écran à l'espace virtuel que nous avons défini (ici il va de -1 à 1 sur chacun des axes si `GL_VIEW_SIZE` est à 2.0 par exemple, à vous de voir ce que vous voulez mettre).



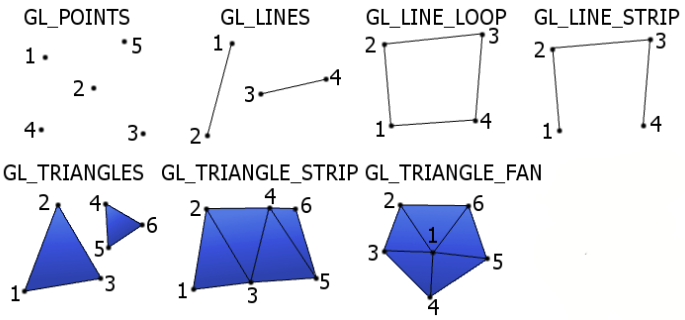
Il vous faudra donc trouver la bonne conversion. Dans un premier temps en considérant une fenêtre carrée (comme ci-dessus). Puis vous devrez exploiter `aspectRatio`, selon l'orientation de la

fenêtre (portrait ou paysage), pour obtenir le bon « mapping » dans le cas d'une fenêtre rectangulaire. La fonction `onWindowResized` peut vous aider.

01b. Assurez vous ensuite qu'à chaque tour de boucle le contenu de la fenêtre soit effacé et que tous les sommets enregistrés dans votre structure soient redessinés à l'écran. Vous devriez maintenant pour voir les points se rajouter à chaque clic.

Note : Pour le moment, la « primitive graphique » que nous utilisons sont les points. Ils correspondent à un état OpenGL nommé GL_POINTS. Il est évidemment possible de dessiner d'autres primitives graphiques. Voici la liste des primitives graphiques acceptables par OpenGL :

- `GL_POINTS`
- `GL_LINES`
- `GL_LINE_STRIP`
- `GL_LINE_LOOP`
- `GL_TRIANGLES`
- `GL_TRIANGLE_STRIP`
- `GL_TRIANGLE_FAN`



A chaque fois que vous « envoyez » (demandez le dessin) d'un point, `GL_POINTS` dessine un point (simple). Pour `GL_LINES`, il faut deux points. A chaque nouveau point rencontré, `GL_LINE_STRIP` et `GL_LINE_LOOP` dessinent un segment. Enfin, il faut trois points pour chaque triangle de `GL_TRIANGLES` et un seul point suffit (après les deux premiers) pour dessiner un triangle pour `GL_TRIANGLE_STRIP` et `GL_TRIANGLE_FAN`. N'hésitez pas à demander des explications à votre chargé de TD (ou de CM).

02. Faites en sorte qu'appuyant sur une touche du clavier, une ligne bouclant sur elle même remplace les points dessinés jusqu'ici. Pour ce faire, regardez les fonctions disponibles dans votre structure de points. Le paramètre à rentrer est tout simplement `GL_POINTS` ou `GL_LINE_STRIP`.

Optionnel 03 : En fait, en utilisant les set_of_point et GL_LINES et les bonnes coordonnées, vous pouvez aussi dessiner un repère de taille 1, c'est à a dire :
- Un segment de taille 1 et de couleur rouge sur l'axe horizontal (axe des *x*)
- Un segment de taille 1 et de couleur verte sur l'axe vertical (axe des *y*)

Exercice 04 – Des objets canoniques…

Votre maîtrise des primitives simples (points et lignes) fraîchement acquise, vous allez maintenant pouvoir créer des objets un (petit) peu plus complexes. Ces objets seront de taille unitaires, et centrés sur l'origine. On les considère comme des objets « canoniques » car en les composant on peut obtenir des objets encore plus complexes.

Pour ce faire, nous allons utiliser une autre structure `GLBI_Convex_2D_Shape` permettant de dessiner des formes 2D convexes. Cette structure permet le dessin d'une forme, remplie ou non, à l'aide de points définissant son contour.

A faire :

01. Fixez la taille virtuelle de la fenêtre à 4.0 (l'univers va donc de -2 à 2).

02. Créez trois « objets » (des variables) `carre`, `triangle` et `cercle` mais à l'aide de la structure `GLBI_Convex_2D_Shape` (comme nous l'avions fait à l'exercice précédent pour l'ensemble de point). Pour ce faire, il vous faudra aussi inclure le bon fichier en entête de votre fichier.

03. Dans votre fonction `initScene`, créez le carré unitaire (longueur des cotés = 1) et centré sur l'origine. Pour ce faire, vous devrez utiliser la fonction `initShape` de la structure `GLBI_Convex_2D_Shape`. Faites-en le rendu dans la fonction `renderScene`. Vous devriez voir un carré (non rempli) noir autour de l'origine. Pour changer la couleur de dessin des objets, il faut appeler la fonction `setFlatColor` de la structure `GLBI_Engine` au moment de dessiner (juste avant). Changez la couleur du carré en rouge.

04. Faites de même pour le triangle (ne dessinez plus le carré). Pour ce triangle prenez les coordonnées (-0.5;-0.5)/(0.5;-0.5)/(0.0;0.5). Dessinez-le en jaune.

05. Enfin, dessinez un cercle. Il faut donc calculer tous les points du bord du cercle unitaire. Dessinez le en vert.

06. Remplissez le cercle en appelant la bonne fonction de la structure de forme convexe 2D. Vous pouvez faire de même pour le carré et le triangle.

07. Pour finir, à l'aide d'une touche (de votre choix) on doit 'boucler' sur l'affichage des trois objets canoniques.

Exercice 05 – Des transformations… et un chat !

Les deux objets canoniques (cercle et carré) définis dans l'exercice 04 sont peu intéressants en l'état… Cependant OpenGL à la particularité de pouvoir déplacer et/ou déformer les objets qu'il dessinera par la suite. Cela se fait en utilisant un mécanisme de transformation d'OpenGL fondé essentiellement sur un ensemble de trois transformations principales : la translation, la rotation et l'homothétie. Voici l'ensemble des transformations affines classiques que nous utiliseront :

- La translation (définie sur les 3 axes x,y,z)
- La rotation qui en 3D dépend d'un angle et d'un axe (vecteur)
- L'homothétie (agrandissement, réduction) qui peut se faire sur les 3 axes

Toutes ces transformations se définissent par des matrices 4x4. Pour ce faire nous allons utiliser une structure (`MatrixStack`), présente dans notre `myEngine`, et permettant de stocker et de manipuler les transformations. Cette structure permet donc de stocker et d'accumuler les transformations.

Ainsi, pour créer une translation, vous exploiterez la fonction (de la structure `MatrixStack`) `addTranslation(const Vector3D& trans)` de la manière suivante :

```
Vector3D tr{0.5,0.0,0.0};
myEngine.mvMatrixStack.addTranslation(tr);
```

Le premier argument défini un vecteur 3D (voir dans le répertoire `tools` la classe – voyez la comme une structure – Vector3D. En 2D, vous mettrez toujours la dernière coordonnée des vecteurs à 0. Pour les rotations et les homothéties, vous exploiterez les fonctions `addRotation(float angle,const Vector3D& axe)` et `addHomothety(const Vector3D& scale)` que vous trouverez dans le fichier `matrix_stack.hpp` dans le répertoire `tools`. **Pour les rotations, le premier argument représente l'axe de rotation, et pour celui-ci, n'utilisez, en 2D, que l'axe z (0.0,0.0,1.0)**. Pour les homothétie, en 2D, vous mettrez la valeur 1 sur la composante z.

Enfin, les modifications que vous faites sur la structure `myEngine` ne se font que du côté **CPU**, et il est nécessaire d'envoyer ces transformations au pipeline graphique. Cela se fera par l'appel suivant :

```
myEngine.updateMvMatrix();
```

Cet appel est indispensable **avant tout dessin**, pour transmettre les transformations au pipeline. Une fois envoyé, cette transformation est celle qui sera utilisé pour modifier les coordonnées de tous les points que vous transmettrez.

Après l'appel au dessin, vous devrez **absolument** réinitialiser les transformations en appelant la fonction de rechargement de l'identité :

```
myEngine.mvMatrixStack.loadIdentity();
```

En effet, si ceci n'est pas fait, les transformations s'accumulent au fur et à mesure ce qui fait « disparaître » rapidement les objets visibles.

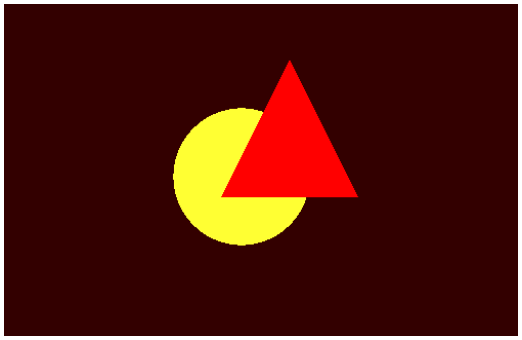
A faire :

01. Reprenez votre exercice de l'exercice précédent. Modifiez la taille de l'espace virtuel (`GL_VIEW_SIZE`) pour qu'il fasse 6 de coté.

02. Dessinez un disque (cercle plein) de la couleur de votre choix et de **diamètre** 1.0 au centre (donc pas de transformation à faire pour le moment).

03. Nous allons décaler ce disque de 1.0 sur la droite (x positif). Implémentez, dans la fonction de rendu, le code nécessaire pour décale le disque. N'oubliez pas de recharger la matrice identité **après** le rendu.

04. Enlevez le décalage précédent du cercle (qui constitue la tête du chat). Vous allez maintenant créez les oreilles du chat. Commençons par l'oreille droite. Il s'agira d'un triangle correctement placé et orienté. **Vous devez reprendre le triangle créé précédemment dans l'exercice 4 (avec les mêmes dimensions).** Tout d'abord, nous allons utiliser une translation. Pour placer l'oreille correctement, il faut mettre la base du triangle à 45° (Pi/4), en haut à droite du disque. Pour le moment, vous devriez avoir ceci (le choix des couleurs est à vous) :



N'oubliez pas de recharger l'identité après le dessin.

05. Maintenant il faut effectuer une rotation de 60° (Pi/3). Effectuez cette transformation, vérifiez et ensuite faites une homothétie de rapport 0.5 sur les axes x et y (toujours avant de dessiner).

06. Nous allons dessiné l'oreille gauche après avoir dessiné l'oreille droite. Pas la peine de recréer un nouveau triangle, vous utiliserez le même ! Après avoir dessiné l'oreille précédente, rechargez la matrice identité puis faites les transformations nécessaires pour l'oreille gauche puis dessinez là (avec la même variable!). N'oubliez pas de recharger l'identité après ce dessin.

Optionnel 07. Je dessinerais bien des yeux (noir par exemple) à ce chat !

Exercice 06 – Vasarely

Attention : Exercice optionnel qui n’est là que pour s’entraîner.

Allez cherchez sur internet des exemples de la série de peinture de Victor Vasarely appelé constellation. Choisissez en une et tentez de la reproduire. Dans les « constellations » les plus simple, il y a « Orange and Green » et « 49 ».

Attention, pour le moment, le dessin se fait plutôt à la manière d’un peintre : toute nouvelle forme dessinée vient recouvrir les formes précédentes. On appelle cela d’ailleurs l’algorithme du peintre.