Les transformations géométriques

Rappels: transformations, vecteurs et matrices.

Les vecteurs et matrices sont ici codés avec la librairie glm
Une surcouche de simplification syntaxique est fournie dans *matrices.h*

Les vecteurs
Vecteurs x,y,z,w : **Vec4** (glm::vec4)
Vecteurs x,y,z : **Vec3** (glm::vec3)
Vecteurs x,y : **Vec2** (glm::vec2)

On peut construire un vecteur plus petit à partir d'un plus grand:

```
Vec4 u(3,4,5,6);
Vec3 v = Vec3(u); // u vaut 3,4,5
```

On peut construire un vecteur plus grand à partir d'un plus petit, il faut juste lui donner ce qui manque

```
Vec3 u(3,4,5);
Vec4 v = Vec4(u,1);
```

Les Matrices 4x4 : **Mat4** (glm::mat4)
`Mat4 m ;` // initialisée à l'identité
Attention: stockage en column-major (m[0] : première colonne)
Produit de 2 matrices :

```
m3 = m1 * m2;
```

Produit matrice x vecteur :

```
v = m * u;
```

Les transformations:
Transformation affine 3D <=> Matrice 4x4 (cf cours)

```
Mat4 m1= translate(x,y,z);
Mat4 m2= rotateZ(alpha);//autour de l'axe X/Y/Z, angle en degré
Mat4 m3= scale(sx,sy,sz);
```

Application à un vecteur:
Soit une transformation f1 codée par une matrice m1
(x',y',z') = f1((x,y,z)) → (x',y',z',1) = m1 * (x,y,z,1)

```
Vec3 v = Vec3(m1*Vec4(u,1));
```

Composition:
appliquer f1 puis f2 puis f3 sur u:
→ f3(f2(f1(u)))
→ f3 ◦ f2 ◦ f1(u)
→ m3 * m2 * m1 * u; // on multiplie par la gauche !

```
Vec4 u4 = Vec4(u,1);
Vec4 v4 = (m3 * m2 * m1) * u4;
Vec3 v = Vec3(v4);
ou:
Vec3 v = Vec3(m1*m2*m3*Vec4(u,1));
```

Utilisation par OpenGL:
Les shaders (code exécuté par le GPU) utilisent toujours au moins 2 matrices:

- *model-view :* place les objets face à la caméra (en fonction de la boite englobante, des manipulations de la souris, etc ...)
- *projection :* projette les points 3D sur l'écran 2D

Pour déplacer un objet dans la scène rendue sans avoir à modifier les coordonnées de tous les sommets, il suffit de multiplier la matrice model-view par une matrice de transformation, avant l'appel corrrespondant à l'affichage de cet objet.

Trasnformations et repères locaux:
Toute transformation affine est équivalente à un repère local
Les colonnes de la matrice donnent les vecteurs X,Y,Z et la position de l'origine.

Code & Interface

Le développement se fera sous QtCreator.

Le projet Geom3D contient les sous-projets suivants:
- OGLRender (rendu en OpenGL 3.3 fourni)
- QGLViewer (version custom de la libQGLViewer qui permet de manipuler une scène à la souris)
- Transfos : un TP (2 séances) sur les transformations
- Revolution : un TP (2 séances) sur les polygones, les maillages et l'algorithme de révolution
- Projet_modeling : Le projet de modélisation par extrusion à partir d'un cube

Le Projet Transfos

La classe **Primitives** permets de tracer des cubes, spheres, cylindres et cones.
Elles sont toutes centrées en 0,0,0, et de tailles 1.
Voir les commentaires dans primitives.h

La classe **Viewer** gère l'interface
- clavier
- affichage
- animation

Voir les commentaires dans viewer.h & viewer.cpp

TP à réaliser:

Commencer par prendre en main l'interface en jouant avec les événements clavier, le draw, les primitives, etc...
Pour l'affichage des primitives (regarder la méthode Viewer::draw_basic())

Les tests suivants seront fait à la fin de la methode Viewer::init()

Testez la création des différentes matrices de transformations en les affichant

Appliquez ces transformations au point 2,2,2

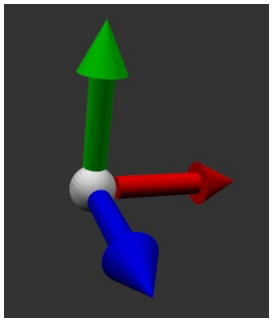
Faites une composition de Translation et de Rotation/Z

Vérifier que Translation + Rotation $\neq$ Rotation + Translation

Coder la création d'un repère direct (**O,X,Y,Z**)

Méthode draw_repere()

- Commencer par réaliser une flèche
- Utiliser 3 x la flèche pour faire le repère.
- Utiliser le paramètre *global* pour placer le repère ou l'on veut

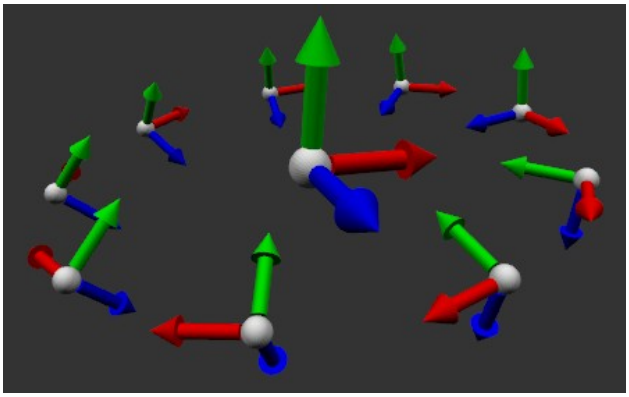


Afficher plusieurs repères qui tournent autour de l'axe Y, l'axe x de chaque repère est tangent à la trajectoire:

- inclinez le plan de le trajectoire de 10° /Z

- faites tourner les repère sur eux même (/X)

- animer avec en faisant varier les angles à l'aide d'événements clavier puis avec animate.

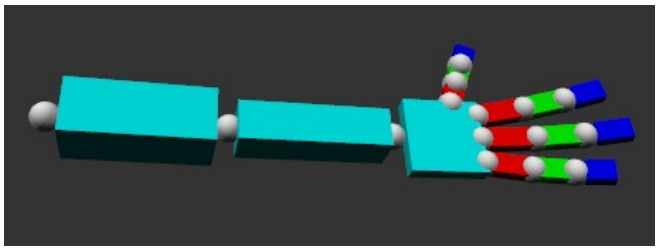


Création & animation d'une main, d'un bras

Méthode draw_main()

Vous utiliserez uniquement des cubes & des sphères

Commencez par faire un doigt



Vous pouvez le paramétrer par sa longueur

Utilisez plusieurs fois doigt pour faire une main

Ajoutez la main

Ajoutez le bras

….

Animez (pliez les doigts et le bras en faisant varier les angles à l'aide d'événements clavier puis avec animate.

Un peu de c++

Référence

- Une “ref” est un alias sur une variable. Le nom est différent mais on accède à (ou modifie) la même donnée.
- Syntaxe: TYPE& var. Exemple: `Vec3& Q = P;` Q est maintenant un alias de P
- Attention, on ne peut pas réutiliser le nom: `Q = R;` Q et donc P, sont remplacés par R
- C'est un pointeur déguisé: pas de * ni de →, il est forcément initialisé
- Une référence constante (const) est un alias sur une variable que l'on ne peut pas modifier
- Sert surtout pour passer des paramètres:

```
bool intersect(const Vec3& P, const Vec3& Dir, Vec3& inter);
. . .
Vec3 B;
bool i = intersect(A, V, B);
```

A et V ne sont pas modifiables dans *intersect*, alors que B oui, il contiendra ici le résultat de l'intersection (si on renvoit vrai)

Affichage dans la console (plus de printf !)

```
#include<iostream>
std::cout << "X=" << x << " / Y = "<<y<< std::endl;
```

Math

```
#include <cmath>
std::sin(a); std::pow(3,4); …
```

Fonction locale (lambda, c++11)

Permet de définir localement une fonction, afin d'éviter d'ajouter à la classe une méthode qui ne sert que dans une autre méthode. La syntaxe est la suivante:

auto nom_fonction = [&] (paramètres) - > type_retour
{ … code … };

[&] signifie que l'on peut accéder aux variables de l'environnement (scope + classe)

```
float b=2.2f;
auto fonction_locale = [&] (float a) -> void
{
    std::cout<<"param a="<<a<<" global b="<<b<<std::endl;
};

//appel
fonction_locale(1.1f);
```

Types et constantes

0.1 → double
0.1**f** → float
13 → int (32bits)
13**u** → unsigned int
13**l** → long int (64bits)
13**ul** → unsigned long int

Souvent optionnel (warning), mais parfois obligatoire (par exemple dans Vec3 u = v * 2.0f;)

Remarque: ne jamais commencer une constante entière par 0 (car on utilise le mode octal, la base 8)

023 → 23 en octal = 2*8+3 = 19
0x23 → 23 en hexa = 2*16+3 = 35
0b10101 → 21 en binaire