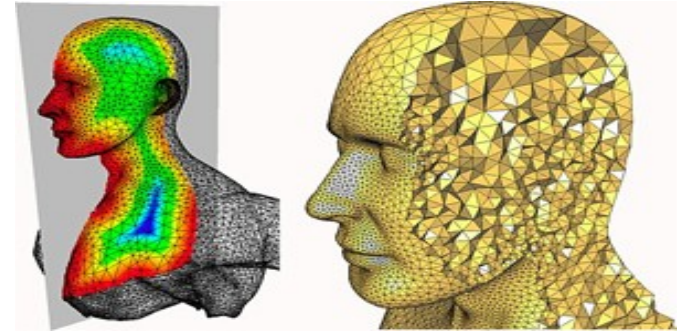
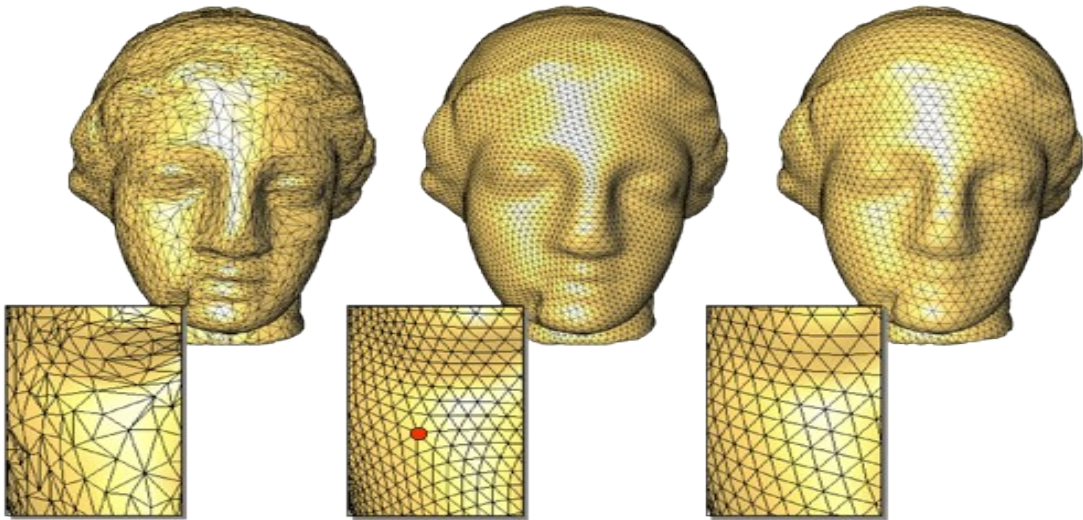


Plans, polygones et maillages

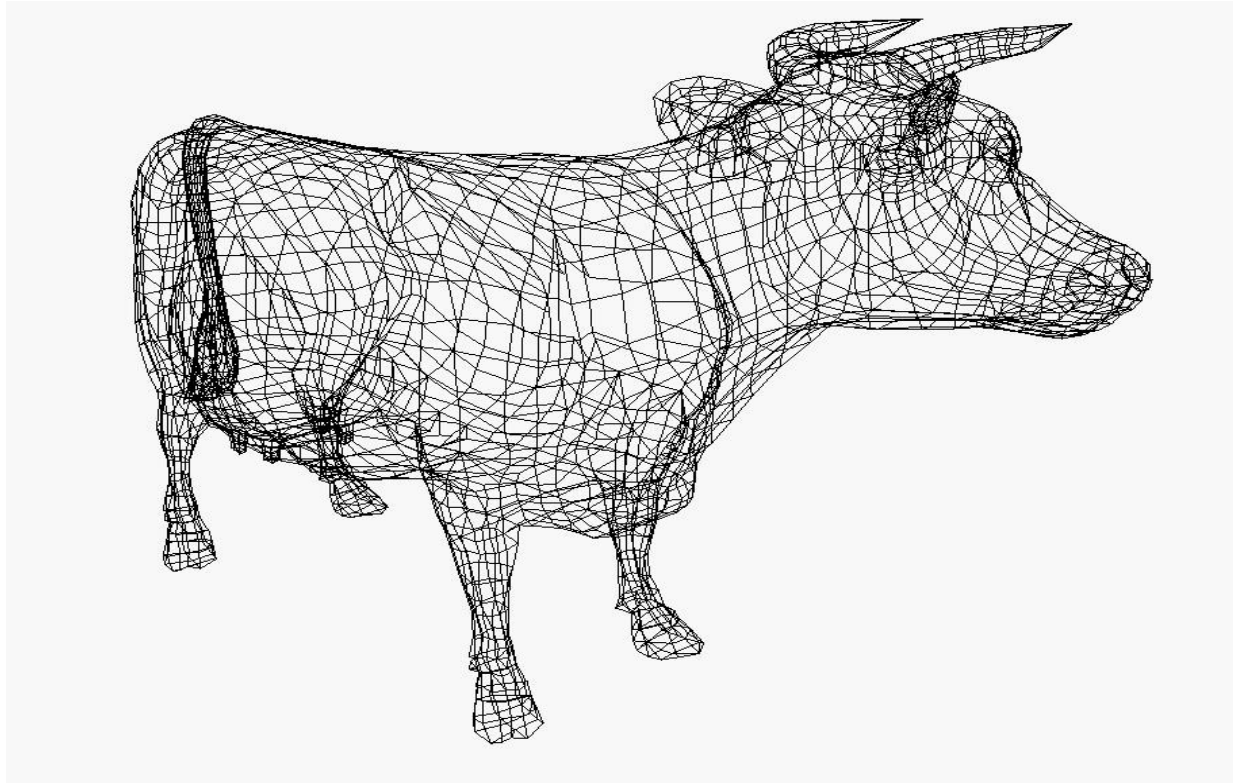


Florence Zara (semestre automne)
LIRIS-ORIGAMI, Université Lyon 1

Plan du cours

- Polygones (2D)
- Polyèdres (3D)
- Format de fichier
- OpenGL
- Code sous gkit (bibliothèque développée par JC Iehl - utilisée en TP)

Maillages polygonaux



Les maillages polygonaux sont la représentation la plus commune

Polygones (2D)

- Un polygone (face) Q est défini par une série de points :

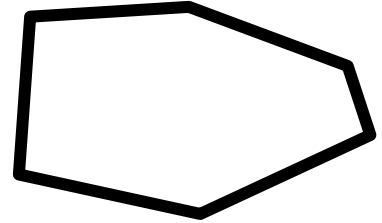
$$\left[p_0, p_1, p_2, \dots, p_{n-1}, p_n \right]$$
$$p_i = (x_i, y_i, z_i)$$

- Les points doivent être **co-planaires**
- 3 points définissent un plan
- Un 4e point ne sera pas forcément sur ce plan

Convexe / concave

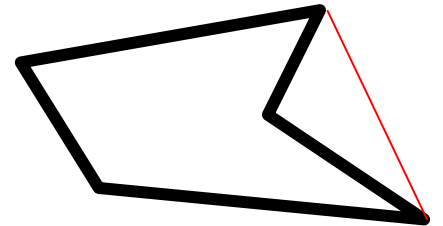
- Convexe

- un polygone est convexe s'il n'est pas croisé et si toutes ses diagonales sont entièrement à l'intérieur de la surface délimitée par le polygone



- Concave

- En SI les polygones convexes sont préférés
- En SI les triangles sont préférés !!
 - Conversion facile d'un polygone convexe en triangles
 - Difficile pour les concaves



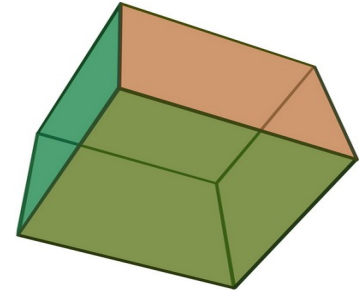
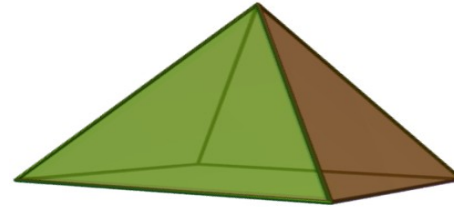
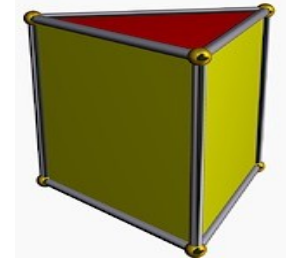
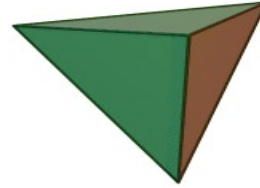
Polyèdres (3D)

- Les polygones sont souvent groupés pour former des polyèdres
 - Une arête joint 2 sommets
 - Une arête joint 2 faces / polygones
 - Les faces ne s'intersectent pas
- Ils sont nommés selon leur nombre de faces

Polyèdres (3D)

- Exemple de polyèdres :

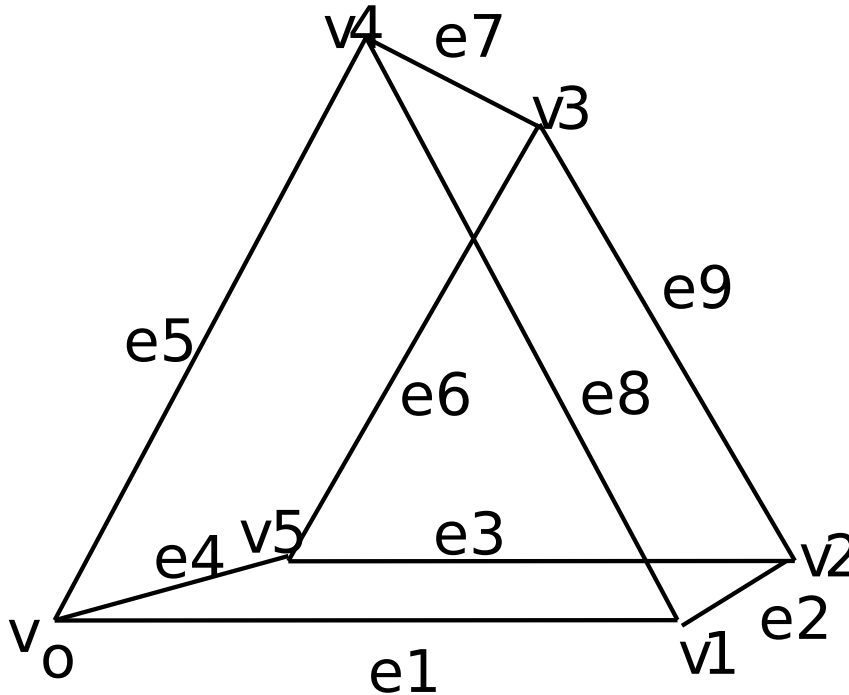
- Tétraèdres (4 faces)
- Pentaèdres (5 faces)
 - Pyramide
- Hexaèdres (6 faces)
 - Hexaèdre régulier : cube



- Caractéristique d'Euler pour les polyèdres convexes

- $V - E + F = 2$ (V= #Vertex, E= #Edge, F= #Face)

Exemple



- $F_0 = v_0v_1v_4$
- $F_1 = v_5v_3v_2$
- $F_2 = v_1v_2v_3v_4$
- $F_3 = v_0v_4v_3v_5$
- $F_4 = v_0v_5v_2v_1$

- $V=6, F=5, E=9$
- $V-E+F=2$ (ok !)

Structure de données

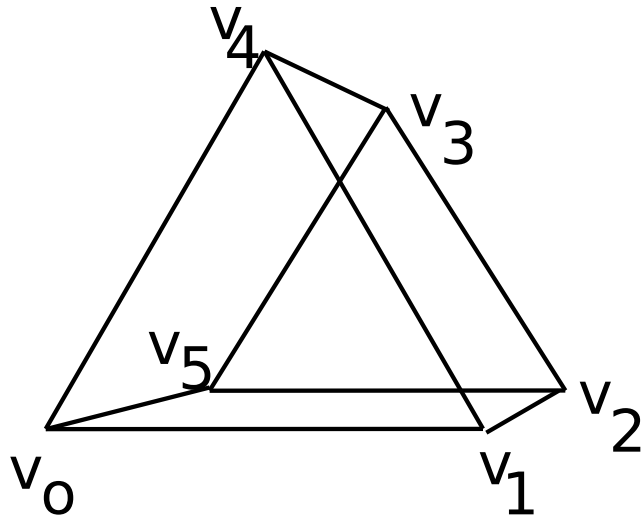
- Exhaustif (Tableau de listes de sommets)
 - faces[1] = $(x_0, y_0, z_0), (x_1, y_1, z_1), (x_3, y_3, z_3)$
 - faces[2] = $(x_2, y_2, z_2), (x_0, y_0, z_0), (x_3, y_3, z_3)$
 - etc.
- Très coûteux en mémoire
 - car chaque sommet apparaît au moins 3 fois !
- Si un point bouge,
 - un trou apparaît car il n'y a pas de notion de connectivité !

Structure de données

Indexed Face Set

- Tableau de sommets (Vertex array)
 - $\text{vertices}[0] = (x_0, y_0, z_0)$
 - $\text{vertices}[1] = (x_1, y_1, z_1)$
 - etc.
- Tableau de faces (liste d'indices dans le tableau de sommets)
 - $\text{faces}[0] = 0, 2, 1$
 - $\text{faces}[1] = 2, 3, 1$
 - etc.

Ordre des sommets



- Polygone $v_0, v_1, v_4 \neq v_0, v_4, v_1$
- Le vecteur normal pointe dans des directions opposées
 $\mathbf{v}_0\mathbf{v}_1 \times \mathbf{v}_0\mathbf{v}_4 = -\mathbf{v}_0\mathbf{v}_4 \times \mathbf{v}_0\mathbf{v}_1$
- Habituellement un polygone n'est visible que depuis les points de son demi-espace positif
- **Back-face culling** (si le point de vue n'est pas devant le polygone, on ne l'affiche pas)

Format de fichier OBJ (AliasWavefront)

- Fichiers OBJ sont au format ASCII
- # Commentaire jusqu'à la fin de la ligne

- ***v float float float***
 - Vertex (sommet). Le 1er vertex a pour numéro 1
- ***vn float float float***
 - Vecteur normal. La 1ere normale a pour numéro 1
- ***vt float float***
 - Coordonnée texture. La 1ere coordonnée a pour numéro 1
- ***f int int int ...***
- ***f intlint intlint intlint ...***
- ***f intlintlint intlintlint intlintlint ...***
 - Face. Les numéros correspondent respectivement au indice de Vertex/CoordTexture/VecteurNormal

Format OBJ : exemple (cube)

Ceci est un cube

8 sommets

v 1 1 1

v 1 1 -1

v 1 -1 1

v 1 -1 -1

v -1 1 1

v -1 1 -1

v -1 -1 1

v -1 -1 -1

6 faces

f 1 3 4 2

f 5 7 8 6

f 1 5 6 2

f 3 7 8 4

f 1 5 7 3

f 2 6 8 4

- Pour la description d'une face des champs peuvent éventuellement être vides

Par exemple :

f 1//7 2/3/5 3// 4/6/8

Formats : OBJ et VRML

- OBJ
 - Format efficace, simple à gérer et standard
 - Trouve facilement des fichiers sur Internet
- VRML (fichier.WRL)
 - VRML=Virtual Reality Modeling Language
 - Egalement, listes de sommets et de polygones

```
Coordinate3 {  
    point [    -2.250000 3.110000 -0.350000,  
              -2.170000 3.070000 -0.520000,  
              ...  
    ]  
}  
IndexedFaceSet {  
    coordIndex [    0, 1, 2, 2, -1,  
                 3, 2, 4, 4, -1,  
                 ...  
    ]  
}
```

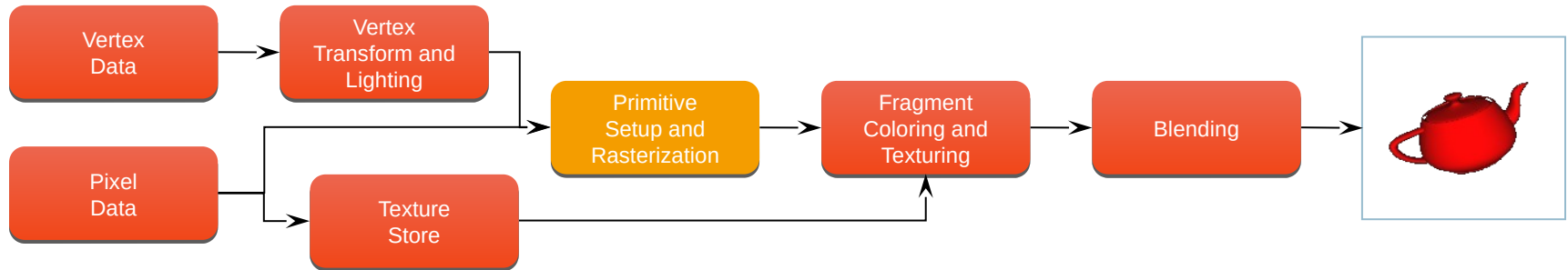


OpenGL : généralité

- OpenGL 1.x est une API 3D (Application Programmer 's Interface)
 - basé sur IRIS GL de Silicon Graphics
 - de bas niveau pour avoir l'indépendance vis-à-vis
 - système de fenêtrage
 - plate-forme matérielle
 - système d'exploitation
- Architecture client-serveur :
 - le client émet les commandes
 - le serveur les exécute

Les débuts ...

- OpenGL 1.0 : 1994
- Pipeline fixe
 - Toutes les opérations étaient câblées et fixes

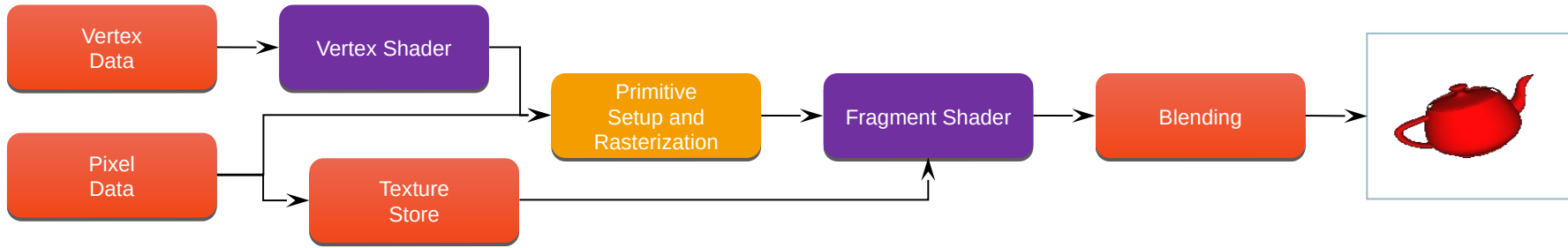


- Le pipeline a évolué
 - Mais en restant fixe au début des évolutions

OpenGL versions 1.1 → 2.0 (Sept. 2004)

OpenGL 3.1

- Fin du pipeline fixe
 - programs = shader
 - Vertex program : transforme les sommets
 - Fragment program : transforme les fragments = pixel

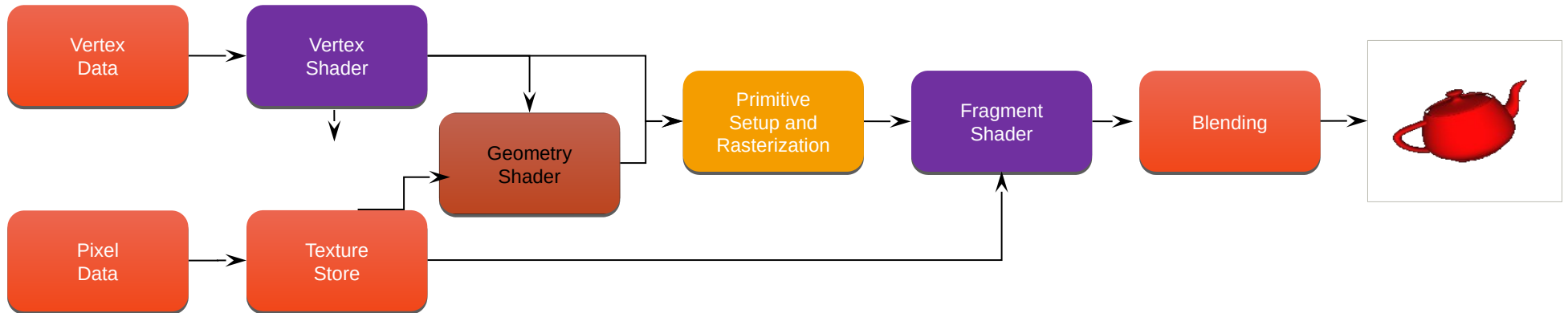


- En plus, les données sont stockées que sur GPU
 - Vertex → buffer objects

Plus programmable

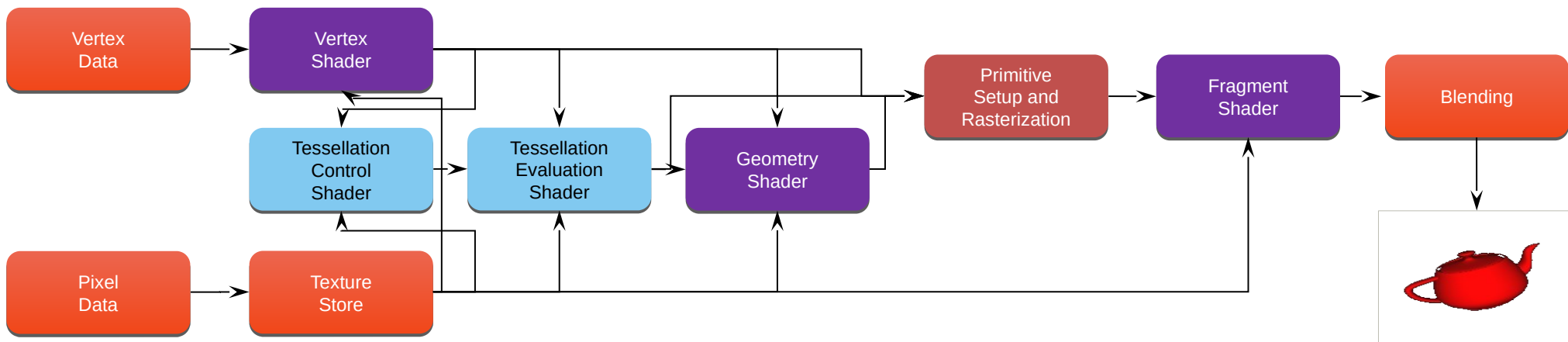
- OpenGL 3.2 (released August 3rd, 2009)
 - Ajoute un étage de shader : geometry shaders

☐ Modifie les primitives géométriques (triangles) dans le pipeline

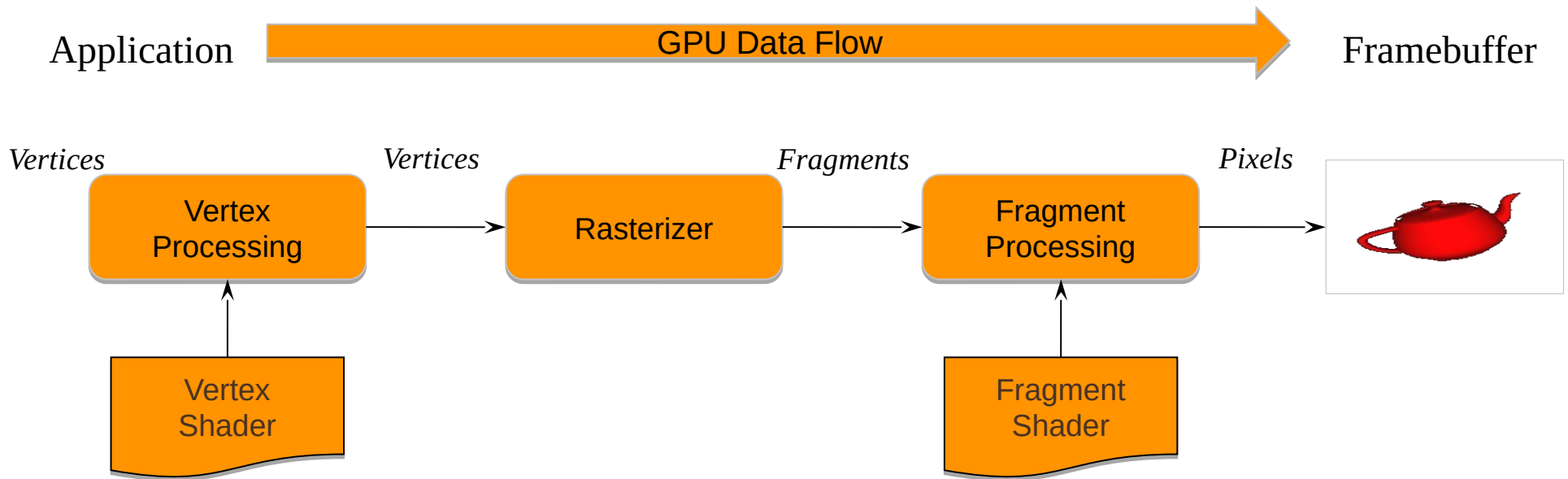


Dernier Pipeline

- OpenGL 4.1 (released July 25th, 2010) ajoute d'autres étages
 - *tessellation-control*
 - *tessellation-evaluation* shaders
- Puis Compute shader



Un pipeline simplifié



OpenGL ES et WebGL

- OpenGL ES 2.0
 - Pour système embarqué comme les mobiles
 - Basé sur OpenGL 3.1
 - Avec Shader
- WebGL
 - JavaScript implementation of ES 2.0
 - Tourne sur des navigateurs récents

OpenGL : bibliothèques associées

Comme OpenGL est indépendant du système de fenêtrage, des bibliothèques additionnelles sont utilisées pour intégrer OpenGL dans de tels systèmes :

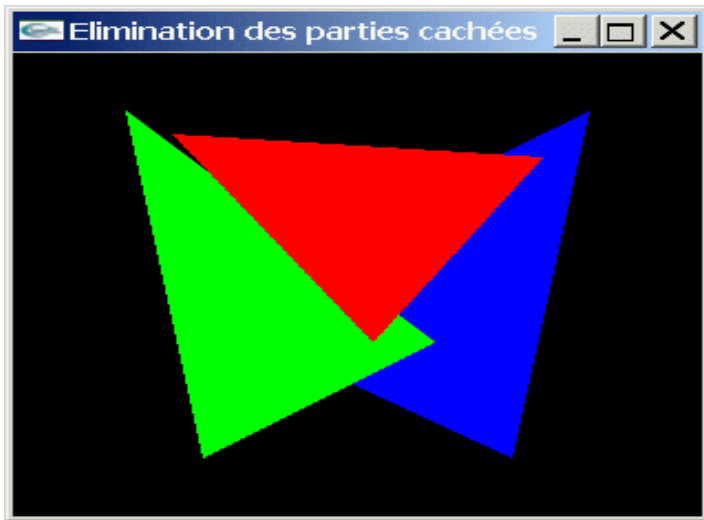
- GLX pour Xwindow
- WGL pour Windows
- GLUT indépendant du système de fenêtrage
- SDL
- Qt
- MFC (windows)

OpenGL : machine à états

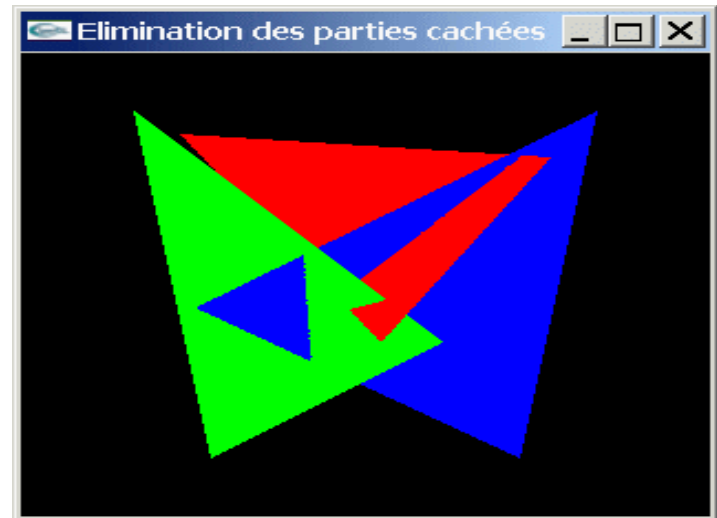
- OpenGL possède un état courant pour tout (repère courant, projection courante, couleur courante, matrice courante, ...)
- On peut activer / désactiver les états avec
 - glEnable()
 - glDisable()
- Tous les états ont des valeurs par défaut (pas besoin de tous les définir en début de programme !)

Exemple avec le Z-buffer

- Pour activer le test des Z avec le Z-buffer
`glEnable(GL_DEPTH_TEST);`



non activé



activé

Utilisation de gkit pour écrire une application OpenGL

Une application gkit-OpenGL est composée de plusieurs éléments :

- une fenêtre pour voir ce qui est dessiné
- un contexte OpenGL pour dessiner
- 4 fonctions appelées dans le main() :
 - **init()** pour créer les objets OpenGL – **création des maillages des objets**
 - **render()** pour afficher les objets créés – **affichage des maillages des objets pour les voir !**
 - **update()** pour mettre à jour les objets OpenGL – **modification des maillages initiaux des objets**
 - **quit()** pour détruire les objets OpenGL à la fermeture de l'application

Code gkit - class Viewer (définit dans fichier [Viewer.h](#))

```
class Viewer : public App
{
public:
    Viewer();          // Constructeur de la class

    int  init();      // Initialise tout

    int  render();   // Fonction d'affichage

    // Mise à jour de la scène au cours du temps
    int  update(const float time, const float delta );

    int  quit();    // Libère tout
```

Protected:

```
// Déclaration de la camera
Orbiter m_camera;

// Déclaration du contexte OpenGL
DrawParam gl;

// Gestion de la camera + lumière
void manageCameraLight();
```

Code gkit - class ViewerEtudiant (définit dans fichier [ViewerEtudiant.h](#))

Le code du TP va être fait dans les fichiers [ViewerEtudiant.h](#) et [ViewerEtudiant.cpp](#)

```
class ViewerEtudiant : public Viewer // classe pour le code du TP
{
    public:
        ViewerEtudiant(); // Constructeur

        int init();        // Pour créer vos objets - appelée qu'une seule fois
        int render();     // Pour afficher vos objets - appelée en continue
        int update(...);  // Pour mettre à jour vos objets - appelée en continue

    protected:
        ...
};
```

Sujet du TP – Création d'objets

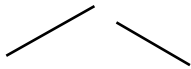
- Création de formes de base (cône, cylindre, sphère, cube)
 1. **Création** du maillage représentant une forme de base
 2. **Affichage** du maillage représentant une forme de base
 - Le maillage peut subir des **transformations géométriques** au préalable
 - Changement d'échelle (agrandir/diminuer la taille de l'objet)
 - Rotation (faire tourner l'objet)
 - Translation (positionner l'objet dans la scène)
Cela permettra d'afficher un objet à plusieurs endroits différents ou de modifier son apparence

Création d'un objet - définir son maillage

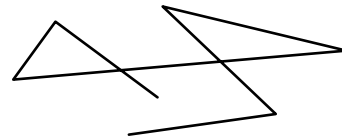
- Chaque objet est composé d'éléments basiques appelés primitives (sommets, faces, etc.)
- Pour créer un objet, on indique ses primitives
- Primitives possibles dans gkit-OpenGL :



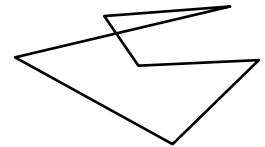
GL_POINTS



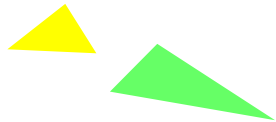
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES



GL_TRIANGLE_STRIP

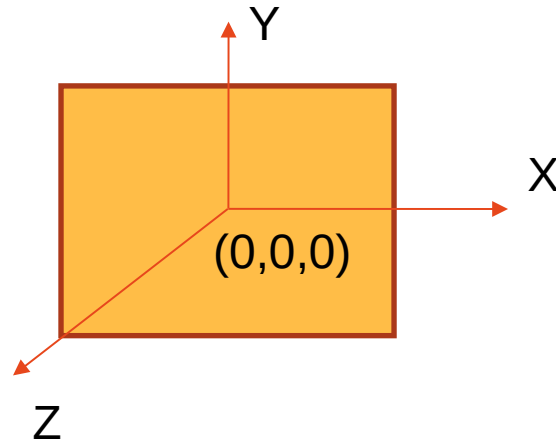


GL_TRIANGLE_FAN

Création d'un objet - définir son maillage

Premier exemple : définir le maillage d'un carré

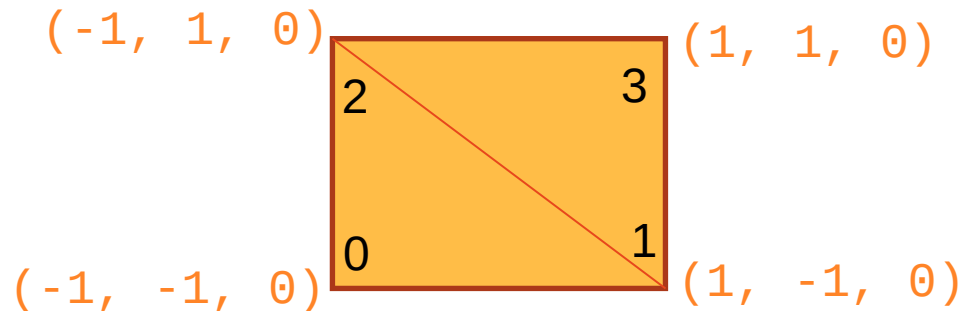
On souhaite créer un carré centré en $(0,0,0)$ et de côté 2



Création d'un objet - définir son maillage

Premier exemple : définir le maillage d'un carré

1. Choix de la **primitive de base du maillage** : triangle (carré = 2 triangles)
2. Définition de la **normale au carré** : $\mathbf{N} = (0, 1, 0)$
La même normale pour les 4 sommets
3. Définition des **sommets** du maillage : coordonnées des 4 sommets
4. Relier les sommets ensemble pour constituer les triangles du carré



Création d'un objet – code gkit d'un carré

Code dans le fichier `Viewer_etudiant.h`

```
class ViewerEtudiant : public Viewer
{
public:
    ViewerEtudiant();

    int init();
    int render();
    int update( const float time, const float delta );

protected:
    Mesh m_quad;    // Déclaration d'une variable de type Mesh représentant le
    carré
    void init_quad(); // Déclaration de la fct créant le maillage (Mesh)
};
```

Création d'un objet – code gkit d'un carré

Code dans le fichier `Viewer_etudiant.cpp`

// Premier exemple : définir le maillage d'un carré

```
void init_quad() // Définition de la fonction créant le maillage (Mesh) du carré
{
    m_quad = Mesh(GL_TRIANGLE_STRIP); // Choix primitive OpenGL pour le maillage

    m_quad.normal( 0, 0, 1 );          // Définition de la normale à la face

    m_quad.vertex( -1, -1, 0 );       // Création du sommet 0
    m_quad.vertex( 1, -1, 0 );        // Création du sommet 1
    m_quad.vertex( -1, 1, 0 );        // Création du sommet 2
    m_quad.vertex( 1, 1, 0 );         // Création du sommet 3
}
```

Attention à l'ordre :

1. Définition des coordonnées de la normale
2. Définition des coordonnées du sommet

Création d'un objet - Fichier `Viewer_etudiant.h`

Pour créer un nouvel objet, vous devez :

1. Déclarer dans `class ViewerEtudiant` :

- la variable de type **Mesh** relatif à l'objet

- la **texture** de l'objet si besoin
(cf. CM texture)

- une fonction **init_Objeto()**

```
class ViewerEtudiant : public Viewer
{ ...
// Déclaration d'une variable de type Mesh
// représentant l'objet que nous souhaitons créer
Mesh m_Objeto;

// Déclaration de la texture de l'objet
GLuint m_Objeto_texture;

// Déclaration de la fonction de création du
// Mesh de l'objet
void init_Objeto();
};
```

Code gkit - Fichier `Viewer_etudiant.cpp`

Pour créer un nouvel objet, vous devez :

2 – Définir la fonction qui créera le **Mesh** de l'objet (fonction écrite dans `Viewer_etudiant.cpp`)

```
// Définition de la fonction créant le maillage
void ViewerEtudiant::init_Objet(){
    m_Objet = Mesh(GL_TRIANGLE_STRIP); // Primitive OpenGL

    m_Objet.normal( 0, 0, 1 ); // Normale au sommet 0
    m_Objet.vertex( 0, 0, 0 ); // Création du sommet 0

    m_Objet.normal( 0, 0, 1 ); // Normale au sommet 1
    m_Objet.vertex( 0, 1, 0 ); // Création du sommet 1
}
```

Attention à l'ordre :

1. Définition des coordonnées de la normale
2. Définition des coordonnées du sommet

Code gkit - Fichier `Viewer_etudiant.cpp`

Pour créer un nouvel objet, vous devez :

3. Appeler la fonction `init_Objet()` dans la fonction `ViewerEtudiant::init()`

```
// Creation des Mesh des objets
int ViewerEtudiant::init(){
    ...
    init_Objet();
}
```

Fonction `init()` appelée **une** seule fois

1. **Création** du maillage représentant une forme de base => OK

Code gkit - Fichier `Viewer_etudiant.cpp`

Pour voir l'objet, il faut effectuer l'affichage du maillage représentant l'objet

Ajouter dans la fonction `ViewerEtudiant::render()` les appels pour l'affichage de votre objet

```
int ViewerEtudiant::render(){// Affichage de la scène
    // Texture
    gl.texture(m_Objet_texture); //(cf. CM texture)

    // Transformation géométrique
    gl.model(Translation( -3, 5, 0 ));

    // Affichage du Mesh de l'objet
    gl.draw(m_Objet);
}
```

Fonction `render()` appelée **en continu** pour rafraichir l'écran

Code gkit - Fichier `Viewer_etudiant.cpp`

Pour mettre à jour le **Mesh** de votre objet, vous pouvez préciser les changements dans la fonction **`ViewerEtudiant::update()`** :

```
int ViewerEtudiant::update(const float time, const float delta)
{
    // Changements que vous souhaitez faire sur le Mesh :

    // Changement des coordonnées de texture d'un sommet
    // (cf. CM texture)
    m_Objet.texcoord(numero-du-vertex, new-u, new-v);

    // Changement des coordonnées (x,y,z) d'un sommet
    m_Objet.vertex(numero-du-vertex, new-x, new-y, new-z);
}
```

Code gkit – Création d'objets

Pour créer le maillage, on peut aussi tout simplement charger un fichier OBJ décrivant la géométrie de l'objet :

```
// Création du Mesh (dans la fonction init())  
Mesh m_objet = read_mesh( /* filename */ "..." );  
  
// Affichage du Mesh de l'objet (dans la fonction render())  
gl.draw(m_Objet);
```


Étapes pour la création d'objets

- Création de formes de base (cône, cylindre, sphère, cube)
 1. **Création** du maillage représentant une forme de base
 2. **Affichage** du maillage représentant une forme de base

- Le maillage peut subir des **transformations géométriques** au préalable
 - **Changement d'échelle** (agrandir/diminuer la taille de l'objet)
 - **Rotation** autour d'un axe (faire tourner l'objet)
 - **Translation** (pour positionner l'objet dans la scène)

=> Applique une transformation géométrique sur le Mesh **avant** son affichage

Rappel : transformations géométriques obtenues en multipliant les points par matrice de transformation correspondante

Transformations – code gkit

Transformations géométriques déjà définies dans gkit : type Transform (matrice 4 x 4)

Rotation :

- Rotation(Vector(x,y,z), angle-degrees);
- RotationX(angle-degrees);
- RotationY(angle-degrees);
- RotationZ(angle-degrees);

Changement d'échelle :

- Scale(Sx,Sy,Sz);

Translation :

- Translation(x, y, z);

Transformations – code gkit

Changement d'échelle : Scale(Sx,Sy,Sz);

```
Transform Scale (const float x, const float y, const float z
{
    return Transform(
        x, 0, 0, 0,
        0, y, 0, 0,
        0, 0, z, 0,
        0, 0, 0, 1);
}
```

Transformations – code gkit

Transformations appliquées avant l'affichage des Mesh dans render()

```
ViewerEtudiant::render()
{
    // Definition de la transformation géométrique
    Transform T = Scale(100,100,100);

    // Applique la transformation au contexte OpenGL
    gl.model(T);

    // Affiche le Mesh de l'objet ayant subi la transformation
    gl.draw(m_Objet);
}
```

Transformations – code gkit

Nous pouvons appliquer plusieurs transformations géométriques :

```
Transform T = Translation( -32, 0, -32 )
    * Rotation( Vector(1,0,0), 45)
    * Scale(64.f/192, 0.3f, 64.f/192);

gl.model(T);
gl.draw(m_Objet);
```

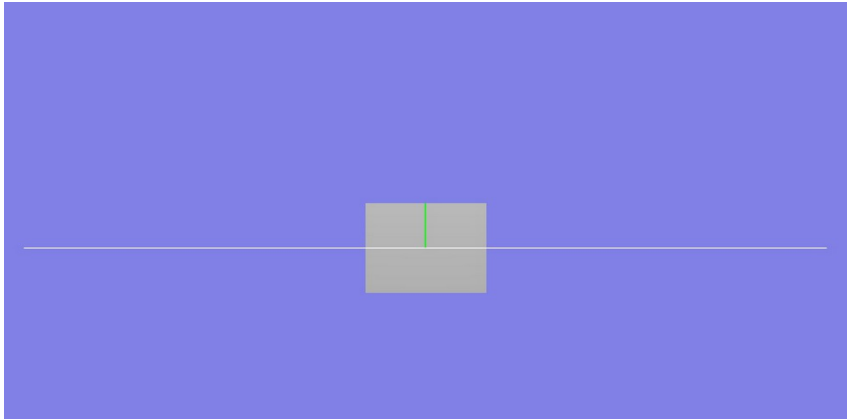
Dans cet exemple, on fait dans l'ordre :

1. Changement d'échelle
2. Rotation
3. Translation

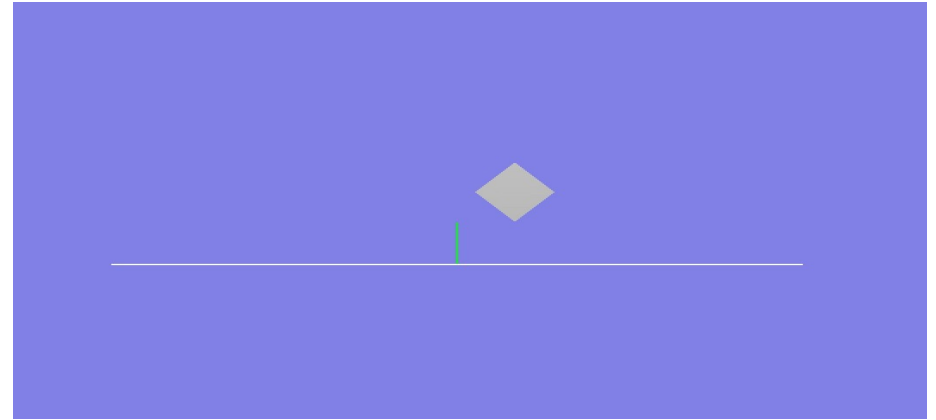
Mais attention à l'ordre des transformations !

(issu du fait que multiplication de matrices non commutative : $M_1M_2 \neq M_2M_1$)

Combinaison de transformations



Initialement : quad centré en
 $(0, 0, 0)$ de taille 2×2
Le coin en bas à gauche, se
situe en $(-1, -1, 0)$



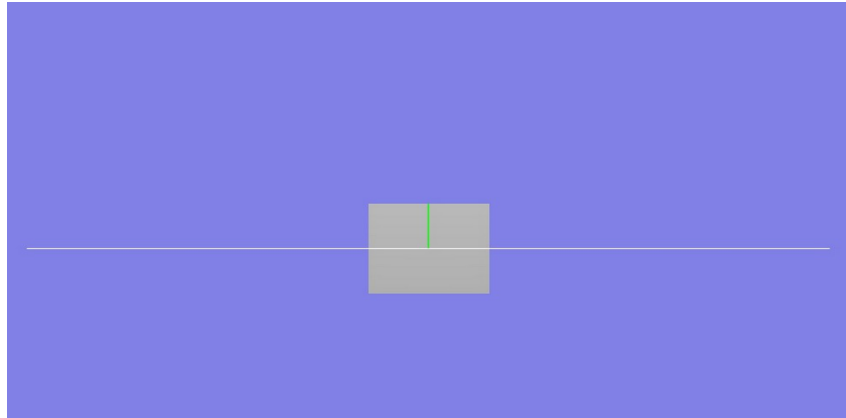
On souhaite : quad de taille 1×1 avec
le coin en bas à gauche en $(1, 1, 0)$ et
une rotation de 45°

On va faire cela en plusieurs étapes

Combinaison de transformations

Étape 0 - Affichage du quad initial

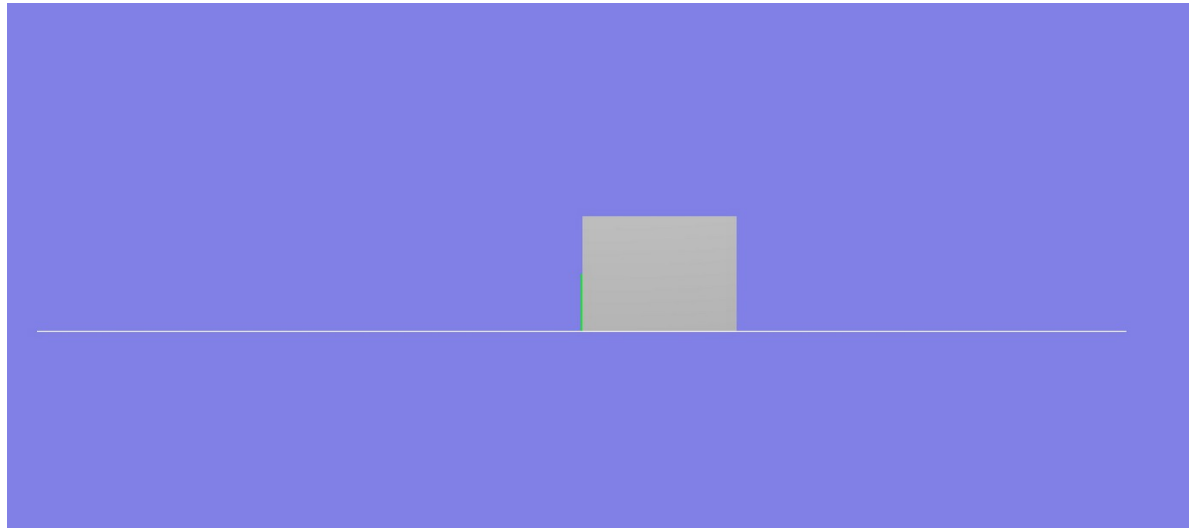
```
gl.model(Identity());  
gl.draw(m_quad);
```



Combinaison de transformations

Étape 1 : translation pour mettre le coin en bas à gauche à l'origine (0, 0, 0)

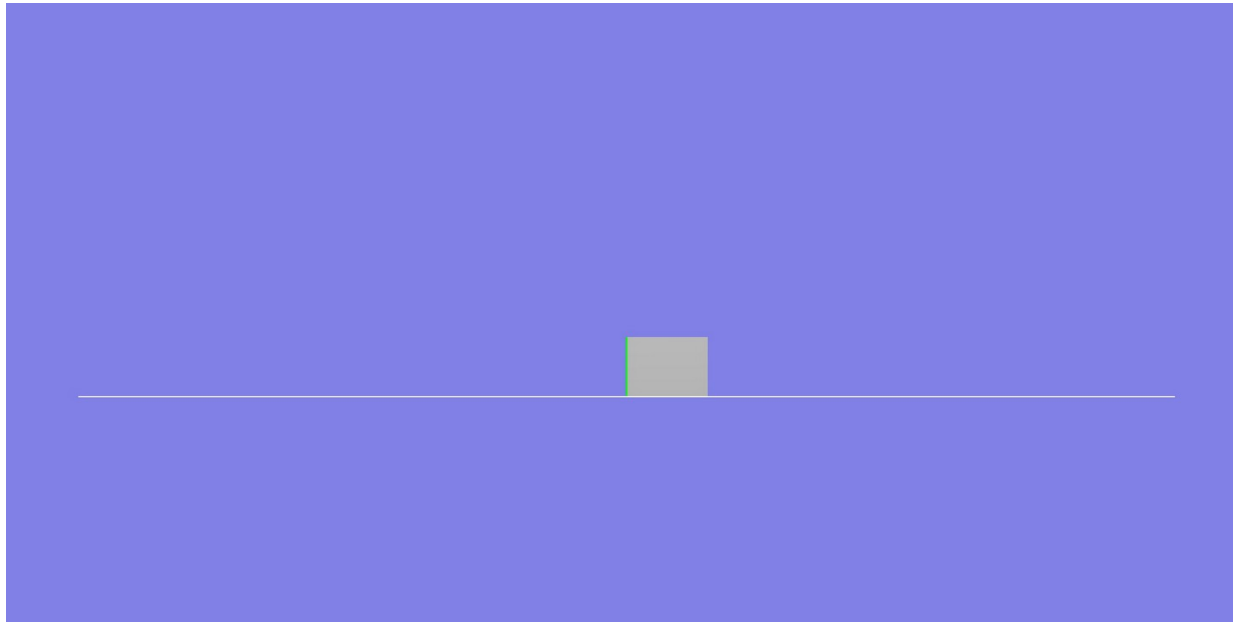
```
gl.model(Translation(1, 1, 0));  
gl.draw(m_quad);
```



Combinaison de transformations

Étape 2 : on ajoute le changement d'échelle de facteur 1/2

```
gl.model(Scale(0.5, 0.5, 0.5) * Translation(1, 1, 0) );  
gl.draw(m_quad);
```

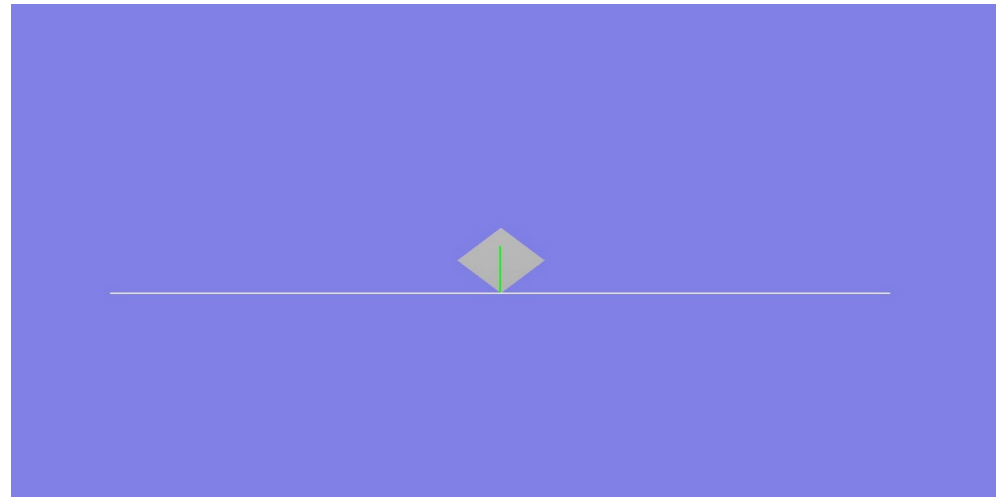


Combinaison de transformations

Étape 3 : on ajoute la rotation de 45 degrés autour des Z

```
gl.model(RotationZ(45)  
  * Scale(0.5, 0.5, 0.5)  
  * Translation(1, 1, 0) );
```

```
gl.draw(m_quad);
```

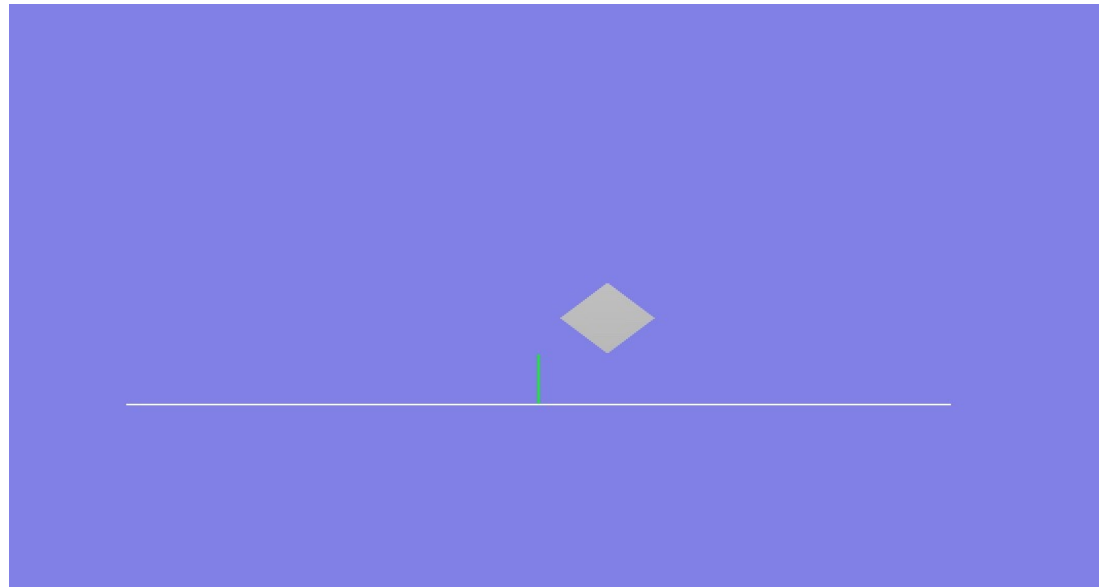


Combinaison de transformations

Étape 4 : on ajoute la translation pour positionner le coin en bas à gauche en (1, 1, 0)

```
gl.model(Translation(1, 1, 0)
* RotationZ(45)
* Scale(0.5, 0.5, 0.5)
* Translation(1, 1, 0) );

gl.draw(m_quad);
```

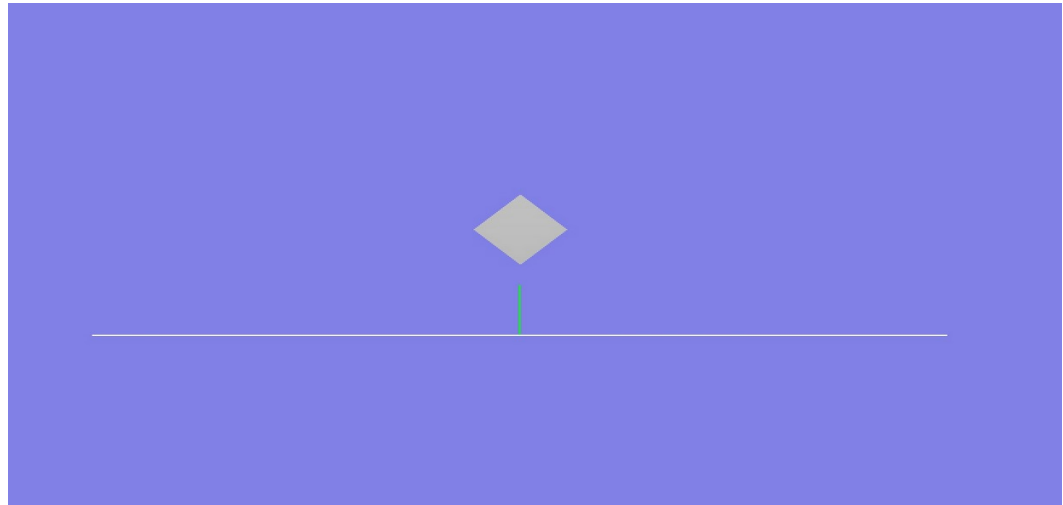


Attention à l'ordre des combinaisons

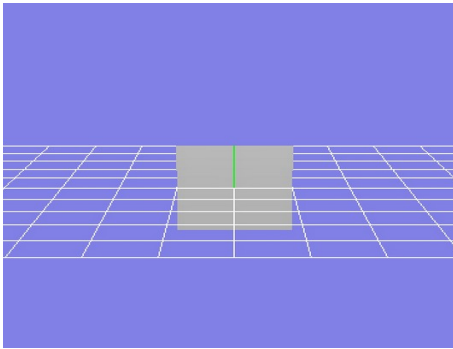
```
gl.model(RotationZ(45)
* Translation(1, 1, 0)
* Scale(0.5, 0.5, 0.5)
* Translation(1, 1, 0) );
```

```
gl.draw(m_quad);
```

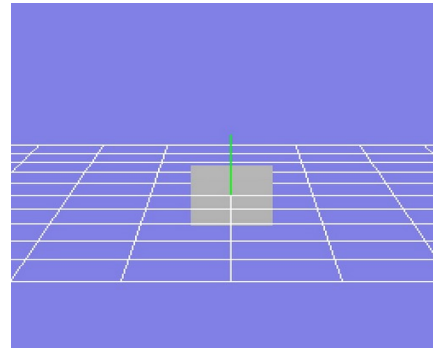
Ordre entre translation et rotation a de l'importance !



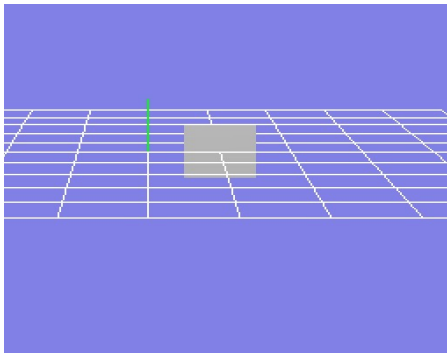
Attention à l'ordre des combinaisons



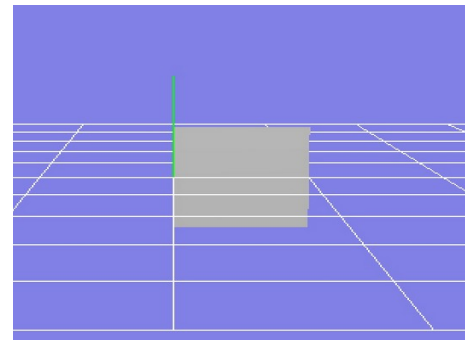
```
gl.model(Identity());
```



```
gl.model( Scale (0.5, 0.5, 0.5) );
```



```
gl.model(Translation(1, 0, 0)  
* Scale (0.5, 0.5, 0.5));
```



```
gl.model(Scale (0.5, 0.5, 0.5)  
* Translation(1, 0, 0) );
```

Attention à l'ordre des combinaisons

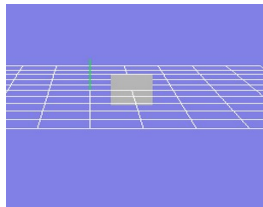
L'ordre des transformations est très important :
un **changement d'échelle suivi d'une translation** est différent d'une **translation suivie d'un changement d'échelle**

$$\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & \Delta x \\ 0 & s_y & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

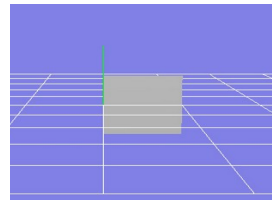
```
gl.model(Translation(1, 0, 0)  
* Scale (0.5, 0.5, 0.5));
```

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & s_x \Delta x \\ 0 & s_y & s_y \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

```
gl.model(Scale (0.5, 0.5, 0.5)  
* Translation(1, 0, 0) );
```



≠

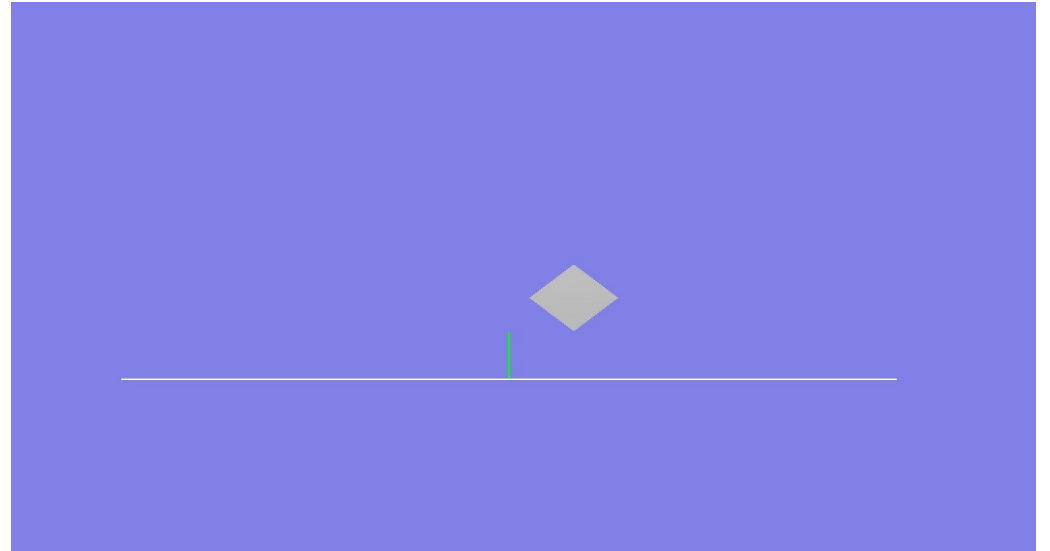


Combinaison de transformations

Sinon en commençant par le Scale...

```
gl.model(Translation(1, 1 + 0.5 * sqrt(2), 0)
  * RotationZ(45)
  * Scale(0.5, 0.5, 0.5));

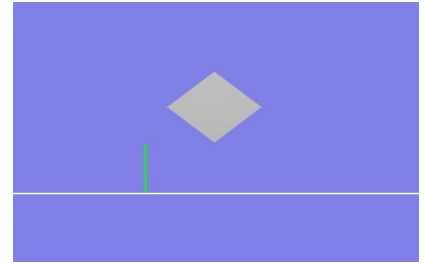
gl.draw(m_quad);
```



Combinaisons de transformations

Il est possible de représenter toutes les combinaisons de transformations par une multiplication d'une matrice et d'un point

```
gl.model(Translation(1, 1, 0)
* RotationZ(45)
* Scale(0.5, 0.5, 0.5)
* Translation(1, 1, 0) );
```



Cela revient à faire :

$$P' = T(1, 1, 0) \text{ Rot_Z}(45^\circ) \text{ S}(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}) T(1, 1, 0) P$$

Dans cet exemple, on fait dans l'ordre :

1. Translation
2. Scale
3. Rotation
4. Translation

Rotation et changement d'échelle

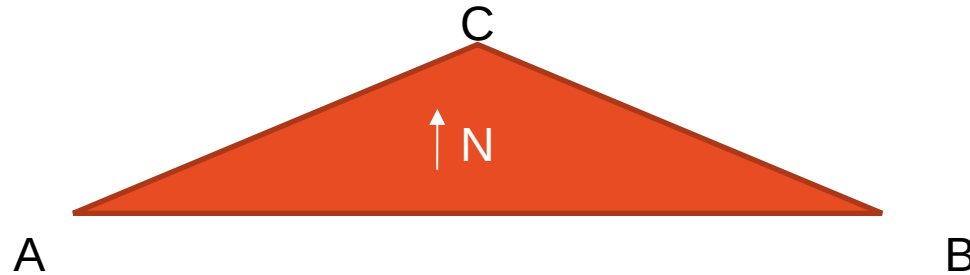
- La rotation et le changement d'échelle sont des opérations qui s'opèrent par rapport à l'origine
- Pour effectuer une de ces opérations par rapport à un point arbitraire, on doit utiliser deux translations :
 - la première pour ramener le point à l'origine et la seconde pour replacer l'objet à sa position originale

$$T(a,b) R(\theta) T(-a,-b)$$

Exercice – Forme de base

- Création d'un triangle

- Il est défini par 3 sommets de coordonnées (x, y, z)



- Il est défini par une normale
 - $N = ???$

Création d'un triangle - Solution

Viewer_etudiant.h

```
class ViewerEtudiant : public Viewer
{
public:
    ViewerEtudiant();

    int init();
    int render();
    int update( const float time, const float delta );

protected:

    Mesh m_triangle; // Déclaration d'une variable de type Mesh

    void init_triangle(); // Déclaration de la fonction de création du maillage
};
```

Création d'un triangle - Solution

Viewer_etudiant.cpp

```
void ViewerEtudiant::init_triangle() // Fonction créant le Mesh du triangle
{
    m_triangle = Mesh(GL_TRIANGLE); // Choix primitive OpenGL

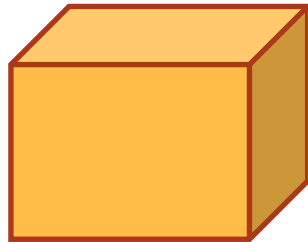
    Point a(1, 0, 0); // Les 3 sommets du triangle
    Point b(3, 0, 0);
    Point c(1.5, 1, 4);

    Vector ab = Vector(a,b); // Calcul de la normal à la face
    Vector ac = Vector(a,c);
    Vector n = normalize (cross (normalize(ab), normalize(ac)) ); // n = AB x AC (produit vectoriel)

    m_triangle.normal(n); // Paramètre = 1 Vector = Normale au triangle
    m_triangle.vertex(a); // Paramètre = 1 Point = Création du sommet 0
    m_triangle.vertex(b); // Paramètre = 1 Point = Création du sommet 1
    m_triangle.vertex(c); // Paramètre = 1 Point = Création du sommet 2
}
```

Exercice – Formes de base

- Création d'un cube (à faire en TD et TP)
 - Cube centré en $(0,0,0)$ et de côté 2
 - Il peut être défini par 6 polygones
 - Il peut être défini avec une structure indexée



Exercice – Formes de base

- Création d'une pyramide (à faire)
 - Avec 4 polygones : base triangulaire
 - Avec des TRIANGLE_STRIP
 - Avec une structure indexée

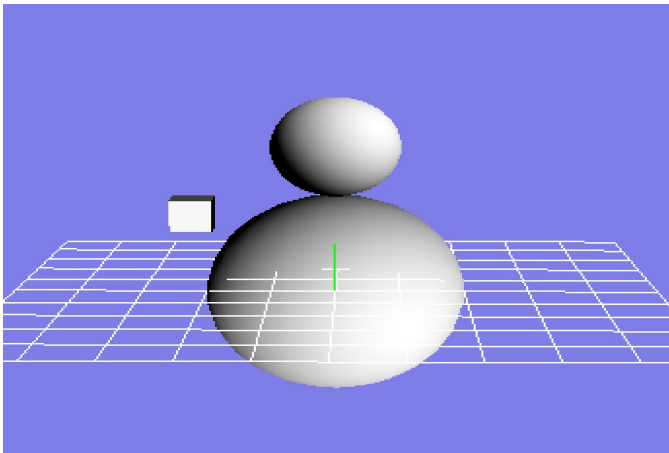
Exercice – Formes de base

- Création d'une sphère (à faire en TD et TP)
 - Centrée en $(0, 0, 0)$
 - De rayon 1
- Création d'un cône (à faire en TD et TP)
 - Base centrée en $(0, 0, 0)$
 - Base de rayon 1
 - Hauteur : 1
- Création d'un cylindre (à faire en TD et TP)
 - Centre : $(0, 0, 0)$
 - Base de rayon 1
 - Hauteur : 1

Il faudra également mettre les normales à ces formes de base

Exercice : affichage d'un objet complexe

- En TD : On souhaite dessiner un « bonhomme de neige »

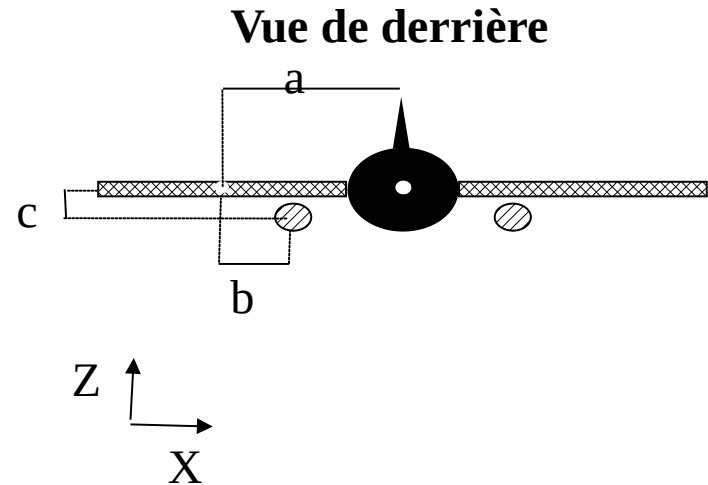
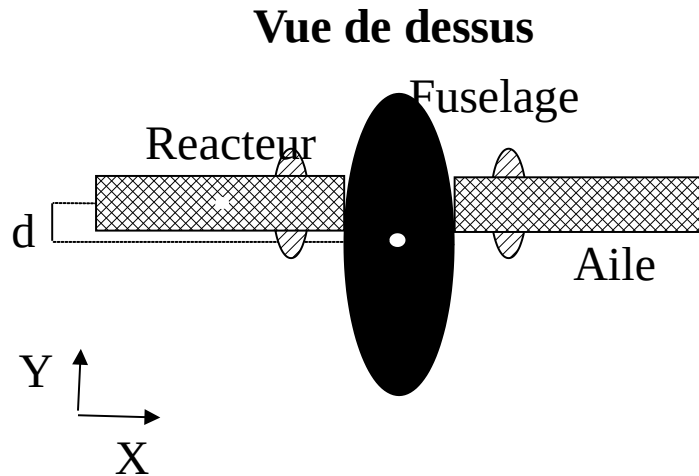


- Un corps blanc
 - de diamètre 4
 - centre en $(0,0,0)$;
- Une tête blanche
 - de diamètre 2 ;

Exercice : affichage d'un objet complexe

En TP : on souhaite dessiner un avion

- On dispose de 2 fonctions
 - `gl.draw(m_cube)` : dessine un cube centré en 0
 - `gl.draw(m_sphere)` : dessine une sphère centrée en 0



Gkit – Pour en savoir plus...

- **Documentation de gkit en ligne :**

<https://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/M1IMAGE/html/index.html>

- **Tutoriel sur les transformations géométriques :**

https://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/M1IMAGE/html/group__transformations.html

- **Pour connaître les opérations définies dans gkit sur les Points, les Vecteurs et les transformations géométriques :**

https://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/M1IMAGE/html/group__math.html