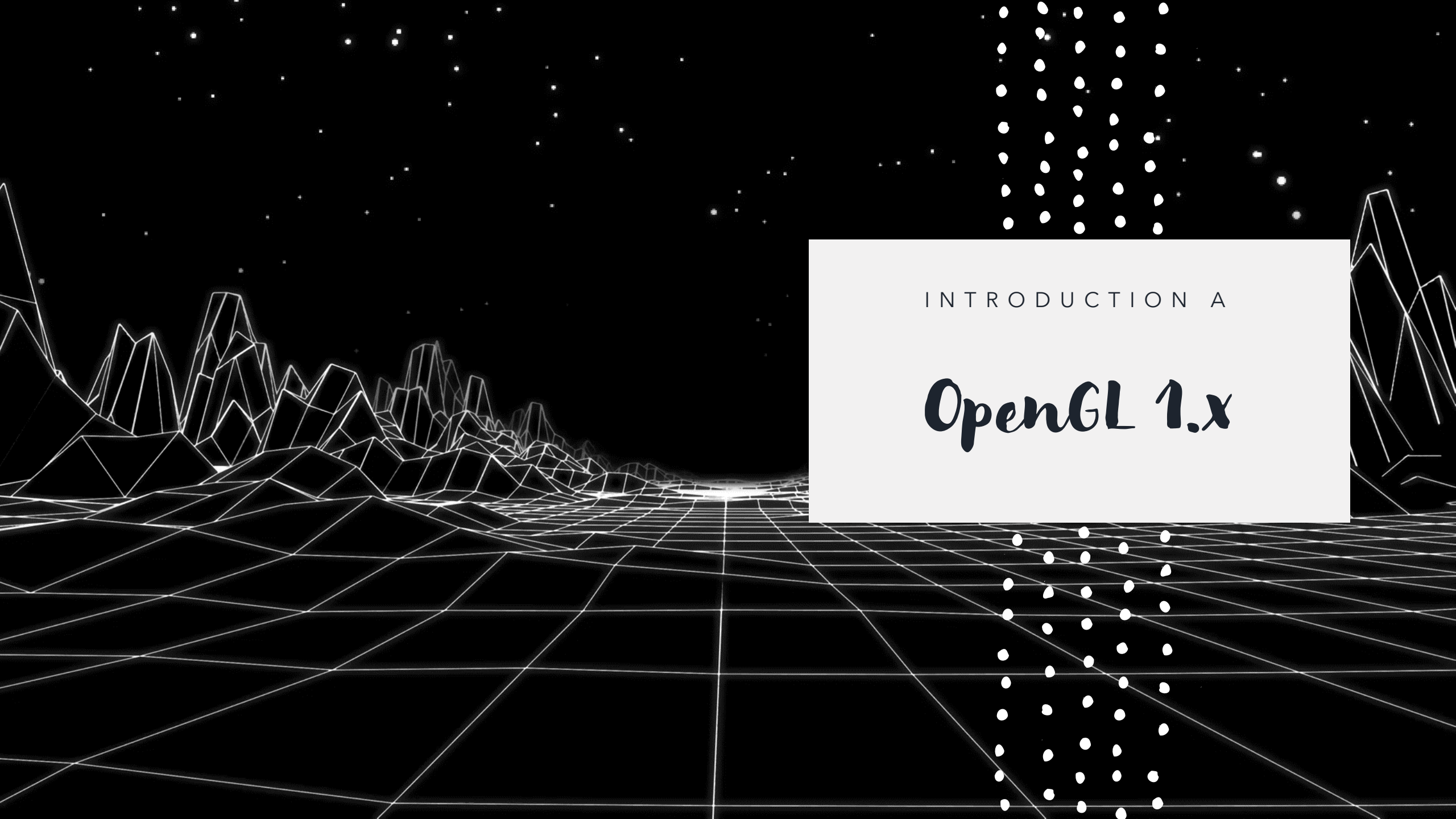


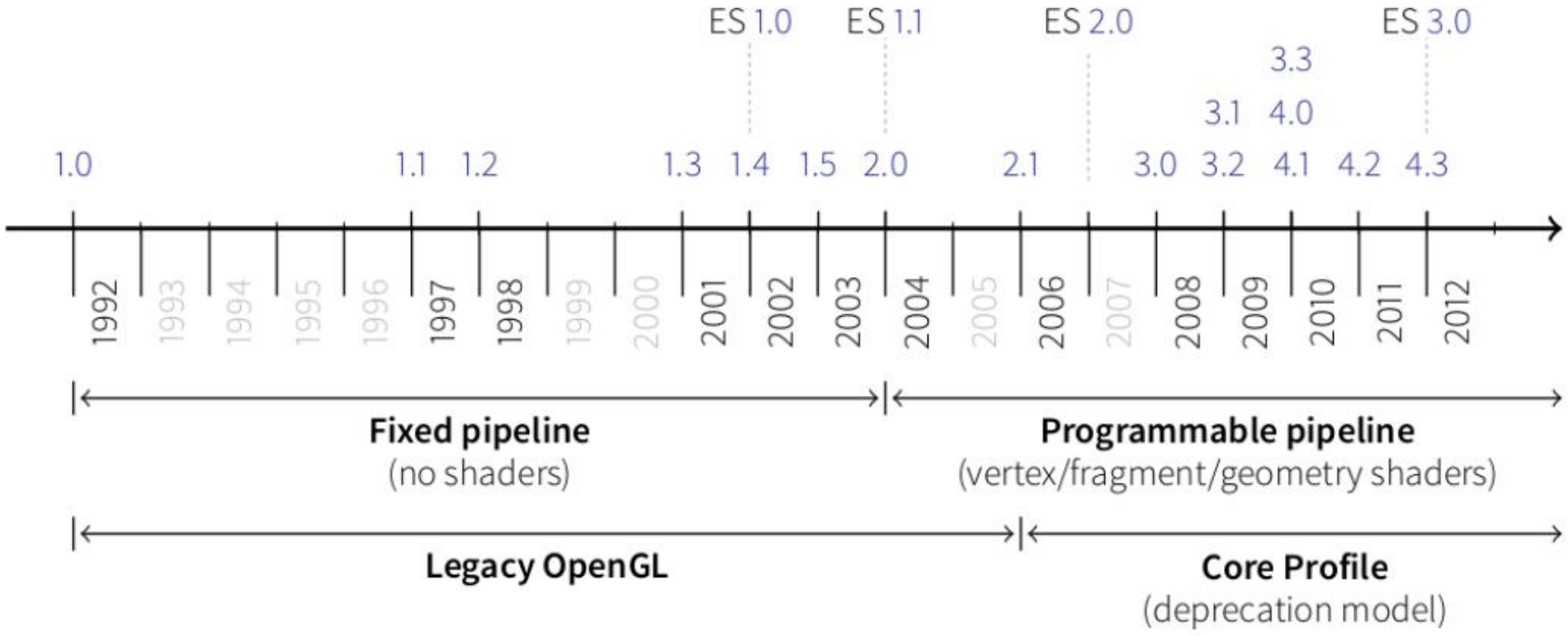
INTRODUCTION A

OpenGL 1.x

Historique

- OpenGL est une bibliothèque graphique contenant des fonctions de calcul d'images 2D et 3D.
- Elle a été développé par la société américaine Silicon Graphics en 1992. (*constructeur de stations de travail et du processeur de la Nintendo 64*)
- On l'utilise pour réaliser des jeux vidéo 3D et des applications de conceptions et de modélisation (CAO).
- Grâce à cette bibliothèque, on peut déclarer la géométrie d'un objet sous forme de points, de vecteurs, de polygones, de bitmaps et/ou de textures.
- Elle effectue ensuite des calculs de projection afin de déterminer l'image à afficher à l'écran. (distance d'affichage, orientation, ombres, transparence , cadrage, etc)

Evolution d'OpenGL



GLUT

- Dans le cadre de cette introduction à OpenGL, on utilisera la bibliothèque utilitaire GLUT (OpenGL utility toolkit).
- Elle facilite la gestion des fenêtres OpenGL et des events (clavier, souris, etc).
- On inclura la bibliothèque OpenGL à un projet via des #include (en particulier GL/gl.h et GL/glu.h).
- Le compilateur aura alors besoin qu'on lui indique les liens vers la bibliothèque via les mots clés –lglut, -lGL et -lGLU pour créer un exécutable
- Attention cependant, GLUT est déprécié de nos jours.

Initialisation

- On initialise OpenGL avec les fonctions **glutInit**, **glutInitDisplayMode**, **glutInitWindowSize** et **glutInitWindowPosition**. On crée la fenêtre de l'application via **glutCreateWindow**. **glutDisplayFunc** et **glutKeyboardFunc** appelleront les callbacks renseignées. **glutMainLoop** lance la boucle infinie principale.
- Lien documentation GLUT

```
void Refresh(void);
void Keyboard(unsigned char key, int x, int y);

int main(int argc, char *argv[])
{
	glutInit(&argc, argv);			// GLUT initialization.
	glutInitDisplayMode(GLUT_RGBA);		// DisplayMode initialization.
	glutInitWindowSize(800, 600);		// WindowSize Initialization.
	glutInitWindowPosition(0, 0);		// WindowPosition Initialization.

	glutCreateWindow("My GLUT Window"); // Creation of the window.

	glutDisplayFunc(&Refresh);		// Window's callback
	glutKeyboardFunc(&Keyboard);		// Keyboard's callback
	glutMainLoop();				// Infinity loop

	return 0;
}
```

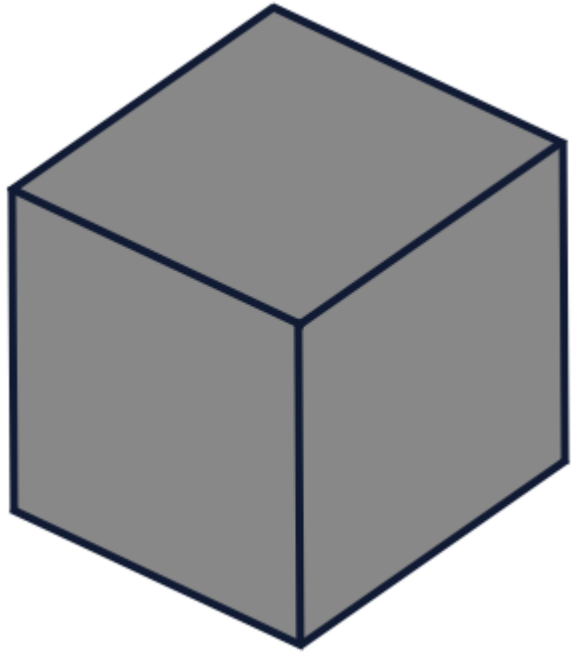
Maillage

On appelle maillage (mesh en anglais) la partie graphique d'un objet.

Ce maillage est constitué de plusieurs éléments, appelés primitives.

On subdivise un maillage en

- sommets (vertices en anglais, vertex au singulier)
- arêtes (edges en anglais)
- faces (composées de polygones convexes simples pour simplifier le rendu)



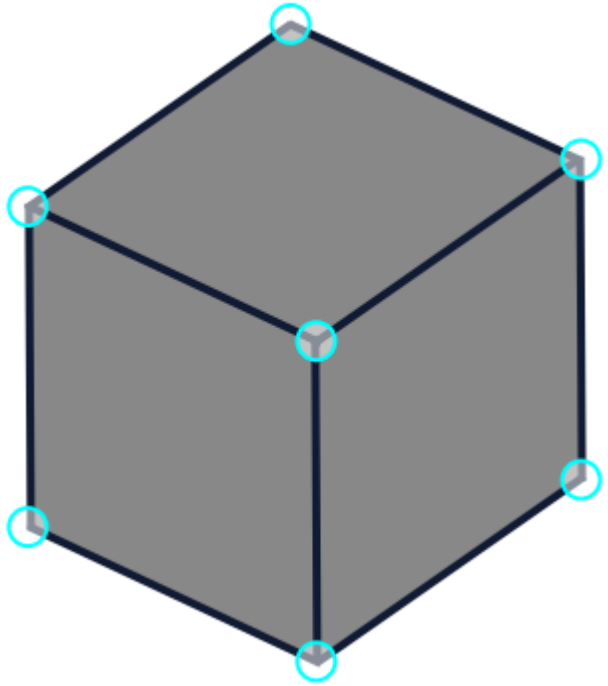
Maillage

On appelle maillage (mesh en anglais) la partie graphique d'un objet.

Ce maillage est constitué de plusieurs éléments, appelés primitives.

On subdivise un maillage en

- sommets (vertices en anglais, vertex au singulier)
- arêtes (edges en anglais)
- faces (composées de polygones convexes simples pour simplifier le rendu)



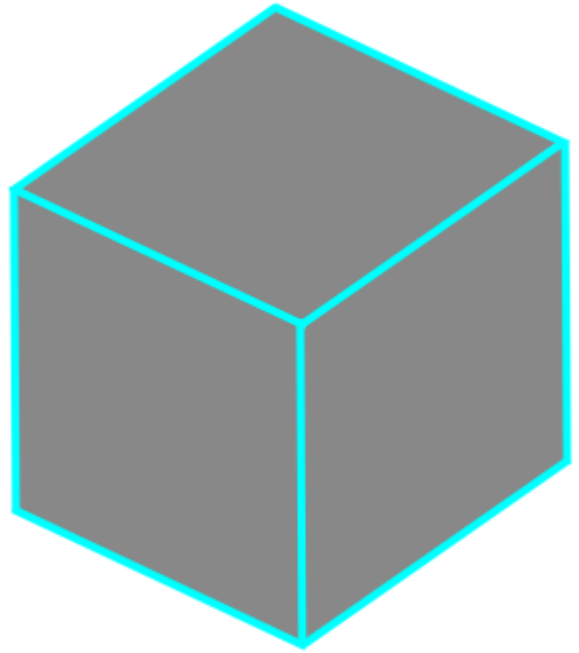
Maillage

On appelle maillage (mesh en anglais) la partie graphique d'un objet.

Ce maillage est constitué de plusieurs éléments, appelés primitives.

On subdivise un maillage en

- sommets (vertices en anglais, vertex au singulier)
- arêtes (edges en anglais)
- faces (composées de polygones convexes simples pour simplifier le rendu)



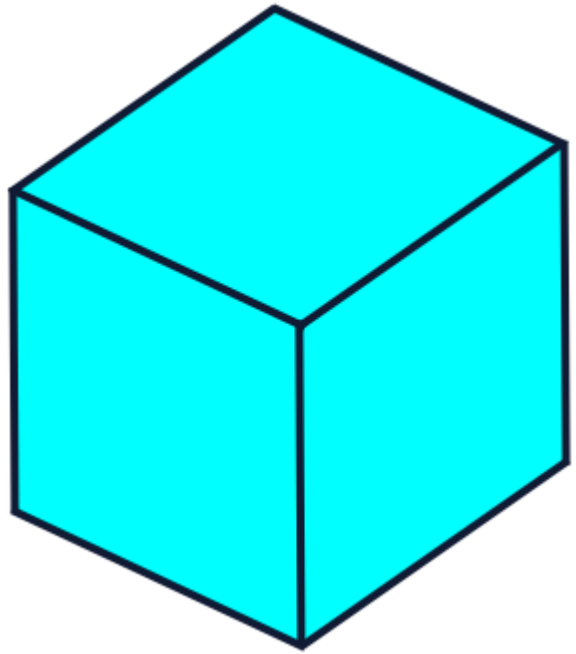
Maillage

On appelle maillage (mesh en anglais) la partie graphique d'un objet.

Ce maillage est constitué de plusieurs éléments, appelés primitives.

On subdivise un maillage en

- sommets (vertices en anglais, vertex au singulier)
- arêtes (edges en anglais)
- faces (composées de polygones convexes simples pour simplifier le rendu)

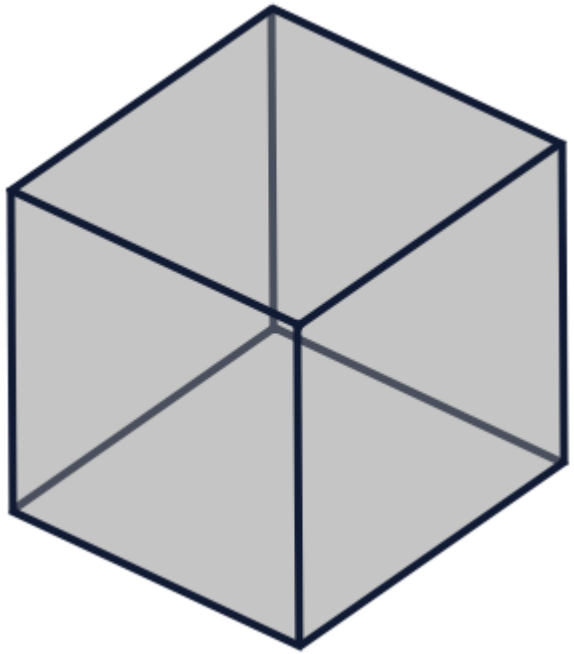


Topologie : quadrilatères

Dans le cas où ses faces sont des « quads »

Un cube peut être composé de :

- 8 vertices (sommets)
- 12 edges (arêtes)
- 6 faces (faces).

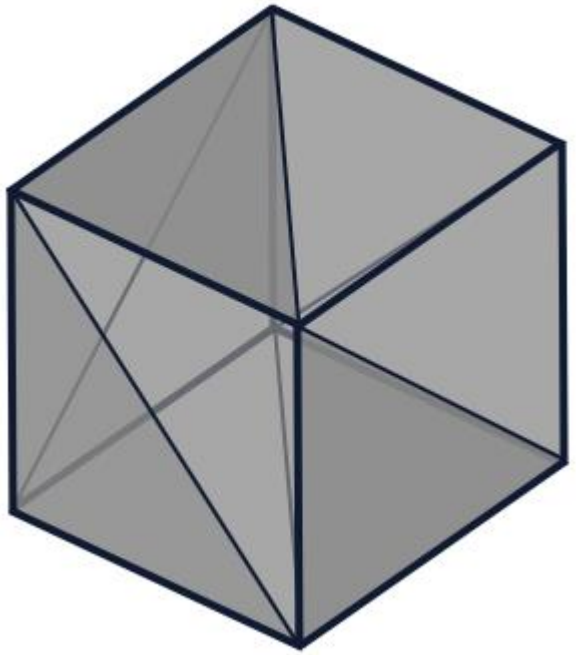


Topologie : triangles

Dans le cas où ses faces sont des « triangles »

Un cube peut être composé de :

- 8 vertices (sommets)
- 18 edges (arêtes)
- 12 faces (faces).



Dessiner

- Pour afficher un objet avec OpenGL il faut dessiner ses primitives.
- La fonction **glBegin** crée un groupe de primitives lié à un objet.
- Elle prend en paramètre le type de polygone souhaité.
- La fonction **glEnd** ferme le groupe en question.
- La fonction **glFlush** force l'exécution des éléments déclarés (***optionnel***).
- A l'intérieur d'un groupe, de nombreuses variantes de fonctions permettent de créer un vertex.
- Par exemple, **glVertex3f** permet de créer un vertex 3D en utilisant des valeurs flottantes comme coordonnées.

```
glBegin(GL_TRIANGLES);

glVertex3f(0.0f, 0.0f, 0.0f);
glVertex3f(0.0f, 1.0f, 0.0f);
glVertex3f(1.0f, 0.0f, 0.0f);

glEnd();

glFlush();
```

Nomenclature

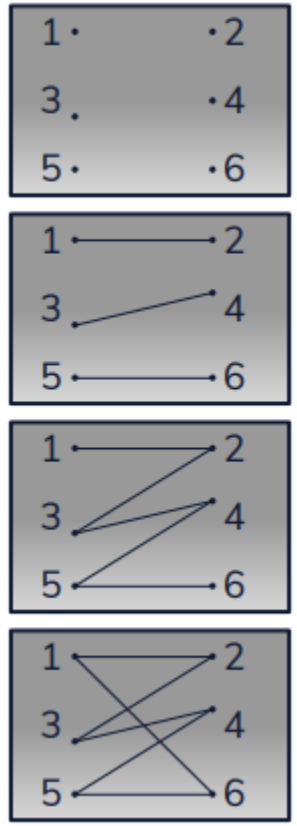
- Toutes les fonctions commencent par **gl** (en « **camel-case** »)
- Les valeurs internes par **GL_** (en **majuscule**)
- Certaines fonctions ont un suffixe composite, par ex:
- **glVertex3f** :
    - ‘glVertex’ est le nom de la fonction
    - ‘**f**’ indique que les paramètres sont des ***float*** (**réels** simple précision)
    - ‘3’ indique que la fonction prend 3 paramètres (ou **composantes**)

Nomenclature

- ***glVertex*** fait partie des fonctions agissant sur la **topologie**.
- On parle d'**attributs**, tel que **Vertex**, **Normal**, **Color**, **TexCoord**, …
- Chaque attribut dispose de 1 à 4 **composantes** (x, y, z, w)
- Les types des composantes peuvent être des entiers '**i**', entiers courts (short) '**s**', des réels simple (float) 'f' ou double précision 'd'
- De plus, les composantes peuvent également être passées par pointeur 'v'

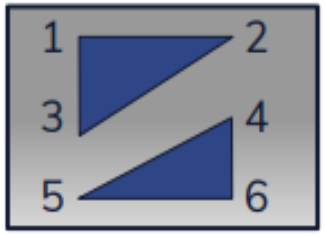
Primitives : lignes et points

- **GL_POINTS** : Dessine un point pour chaque vertex transmis
- **GL_LINES** : Dessine des lignes par groupe de deux vertices
- **GL_LINE_STRIP** : Dessine un groupe de lignes connectées entre chaque vertex
- **GL_LINE_LOOP** : Dessine un groupe de lignes connectées entre chaque vertex formant une boucle

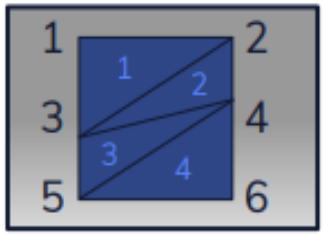


Primitives : triangles

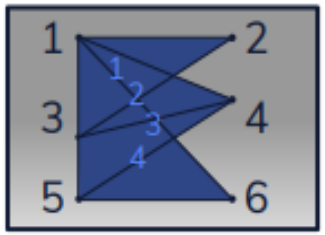
- **GL_TRIANGLES** : Dessine un triangle par groupe de trois vertices



- **GL_TRIANGLE_STRIP** : Dessine un triangle pour chaque vertex défini après deux autres

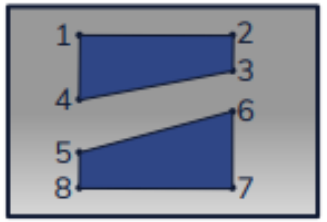


- **GL_TRIANGLE_FAN** : Dessine un triangle par groupe de trois vertices, dont le premier vertex défini sera à chaque le point de départ

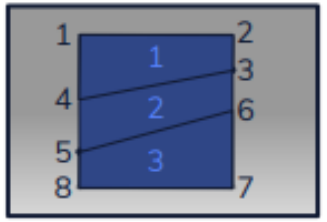


Primitives : quads et polygones

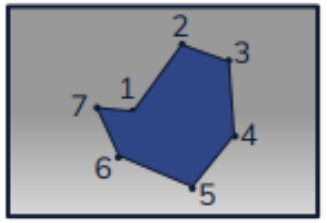
- **GL_QUADS** : Dessine un quad par groupe de quatre vertices



- **GL_QUAD_STRIP** : Dessine un quad pour chaque vertex défini après deux autres



- **GL_POLYGON** : Dessine un polygone de vertices



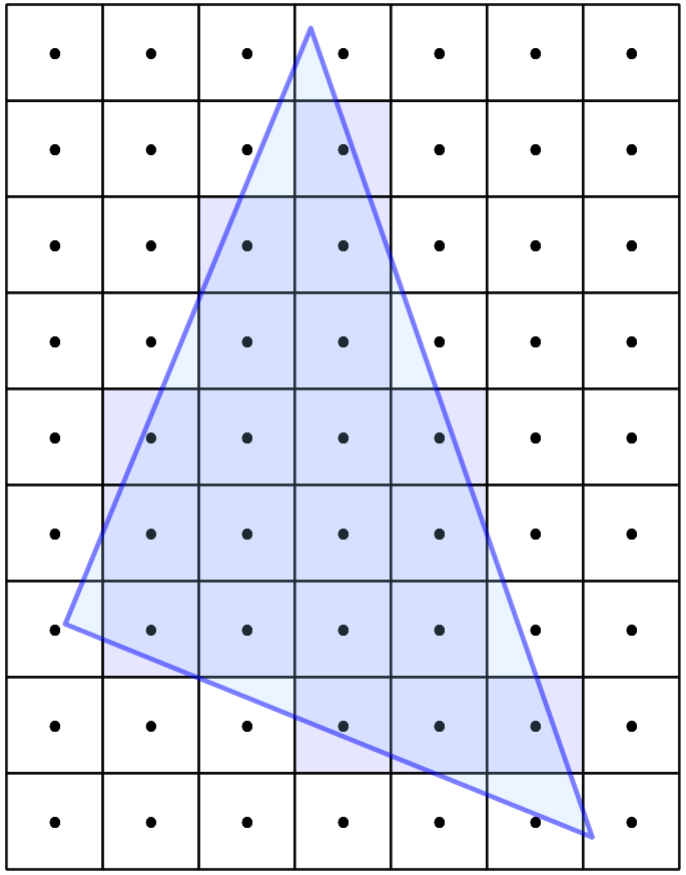
- **ATTENTION !** Ces primitives sont DEPRECIEES en OpenGL 2+
  (mais toujours disponibles sur la majeure partie des pilotes Desktop)

« Rasterization »

- Le dessin des primitives à l'écran se fait par une étape appelée « rasterizer ».

- Les lignes horizontales sont **interpol**ées (*LERP*) **linéaire**ment à partir des arêtes des primitives.

- Ces lignes appelées « *rasters* » suivent une convention d'allumage des pixels spécifique à OpenGL.



Fonctions usuelles

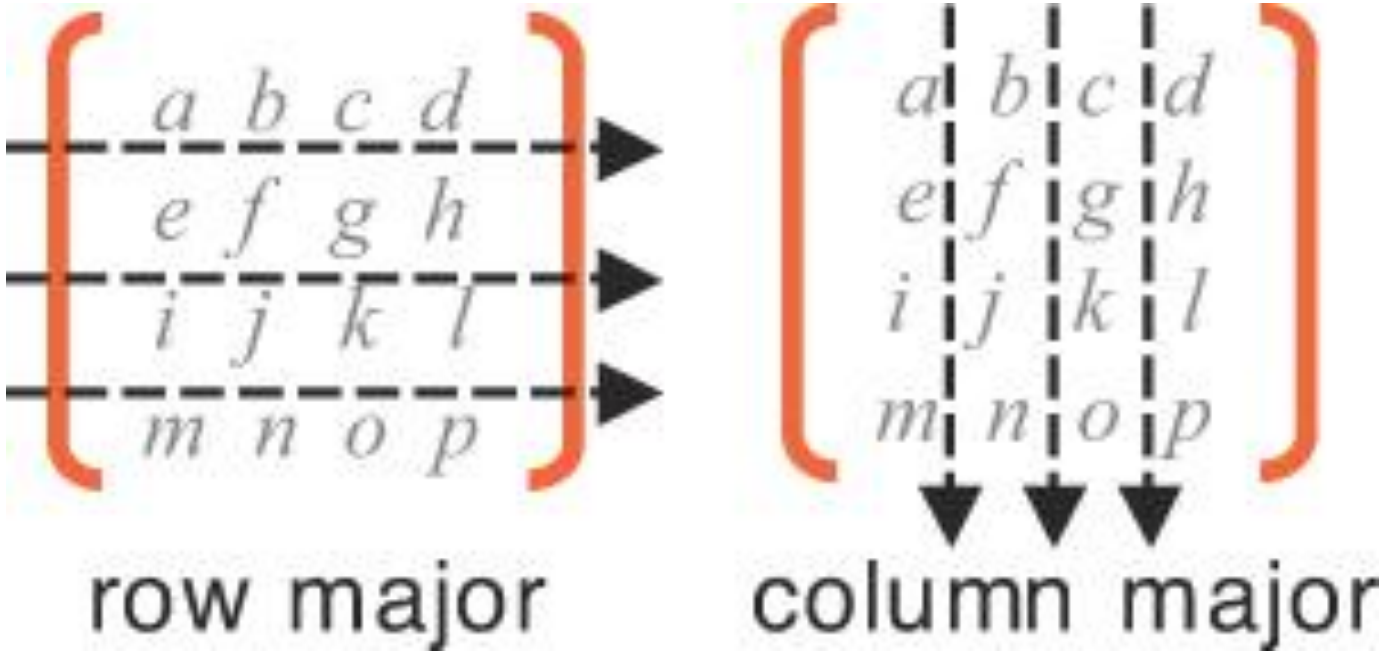
- La commande **glClear** sert à effacer tout ce qui est présent à l'écran.
- On l'utilisera généralement avant de redessiner des éléments en mouvement.
- On verra que l'on peut dessiner (et donc effacer) dans plusieurs zones appelées « buffer » (tampon)

- La fonction **glClearColor(r, g, b, a)** permet quand à elle de modifier la couleur avec laquelle l'écran sera effacé lorsque l'on utilise glClear.

- **glViewport(originx, originy, width, height)** permet de mettre à jour les infos concernant la fenêtre de rendu. (**coordonnées** du coin inférieur gauche, **dimensions** de la zone de rendu). La fenêtre de rendu peut être d'une dimension différente de la fenêtre de l'application.

Conventions : matrices

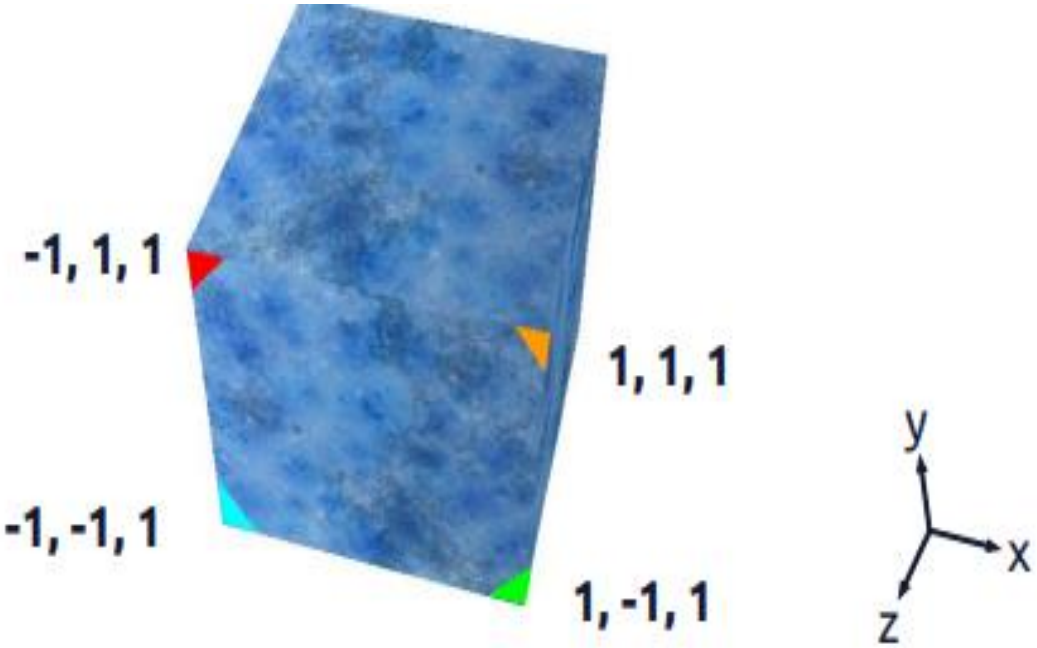
- Les matrices sont des matrices **carrées** 4 lignes 4 colonnes en coordonnées **homogènes**.
- Les matrices sont stockées en **colonne-d'abord** (« *column major* »)



Conventions : repère 3D

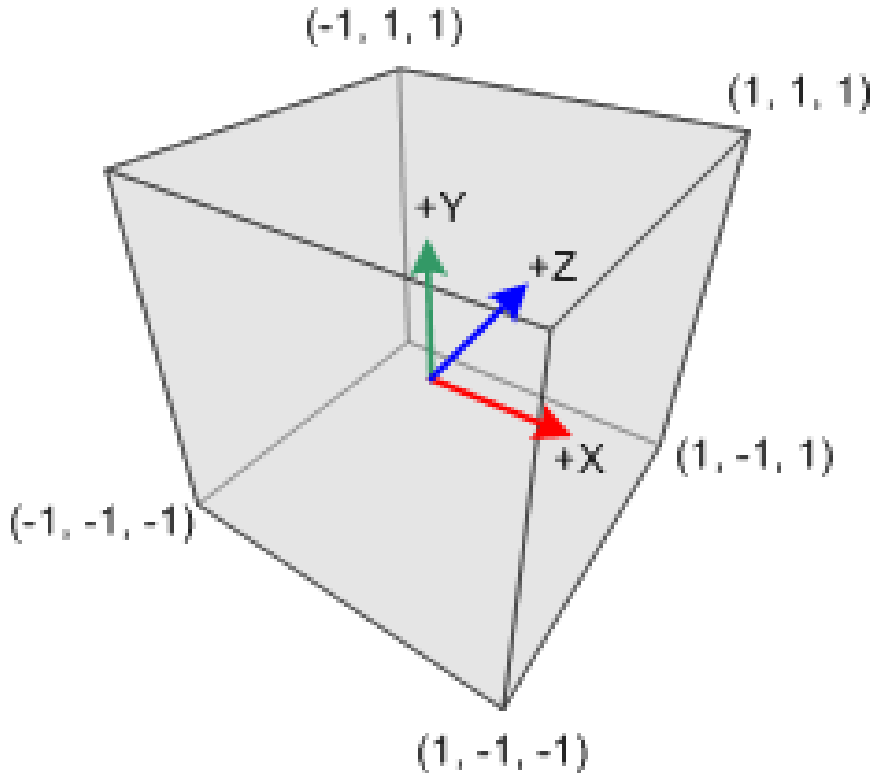
- Le repère d'OpenGL (sa base, ou « frame » en anglais) est un repère main-droite

- Par défaut ce repère est orthocentré, l'axe des **ordonnées** pointant vers le **haut** et l'axe de **profondeur** pointant **hors écran**.



Conventions : NDC

- Lorsque toutes les matrices sont à l'**identité**
- Ou bien lorsque OpenGL doit finalement « **rasterizer** » (dessiner à l'écran)
- le repère d'OpenGL est aligné sur celui du contexte de rendu GL qui est un repère orthonormé main-gauche !



On parle alors de **NDC** : « *Normalized Device Coordinates* »

Matrices de transformation

- **glMatrixMode** permet de sélectionner une des trois matrices OpenGL :

• **GL_PROJECTION** : Utilisée pour paramétrer la transformation des coordonnées 3D en coordonnées 2D.

• **GL_MODELVIEW** : Utilisée pour paramétrer la position du repère de coordonnées dans le monde.

• **GL_TEXTURE** : Utilisée pour paramétrer le rendu des textures qu'on applique sur les faces d'un élément

```
void Timer(int value)
{
        // Your code.
        glutTimerFunc(25, &Timer, 0);
}

void Reshape(int w, int h)
{
        glViewport(0, 0, w, h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45, float(w) / float(h), 0.1, 100);
}

int main(int argc, char *argv[])
{
        // GLUT initialization
        glutReshapeFunc(&Reshape);          // Window's Reshaped callback
        glutDisplayFunc(&Draw);             // Window's callback
        glutTimerFunc(0, &Timer, 0);

        glutKeyboardFunc(&Keyboard);        // Keyboard's callback

        glutMainLoop();                     // Infinity loop

        return 0;
}
```

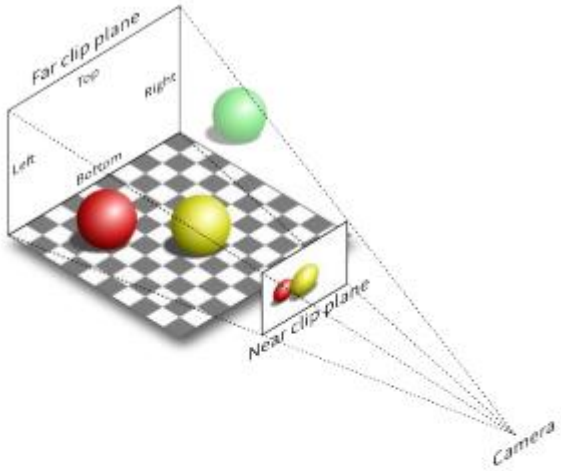
Matrices de transformation

- **glLoadIdentity** réinitialise la matrice OpenGL courante.
- On l'utilise à chaque fois que l'on souhaite dessiner un élément en effaçant les translations précédentes.
- Après avoir effectué une transformation, l'ensemble des éléments dessinés après celle-ci seront affectés.
- Chaque transformation se base sur le repère de coordonnées courant, puis le modifie.
- Le **repère** est donc **local**.

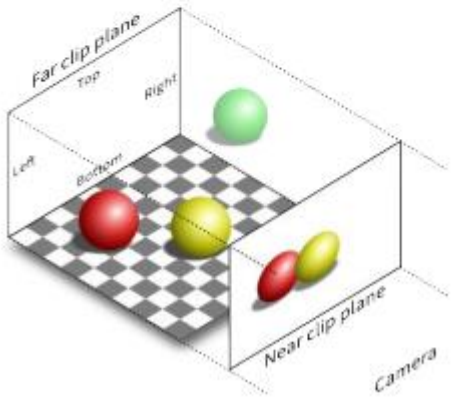
```
void Timer(int value)
{
        // Your code.
        glutTimerFunc(25, &Timer, 0);
}

void Reshape(int w, int h)
{
        glViewport(0, 0, w, h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45, float(w) / float(h), 0.1, 100);
}

int main(int argc, char *argv[])
{
        // GLUT initialization
        glutReshapeFunc(&Reshape);          // Window's Reshaped callback
        glutDisplayFunc(&Draw);             // Window's callback
        glutTimerFunc(0, &Timer, 0);

        glutKeyboardFunc(&Keyboard);        // Keyboard's callback

        glutMainLoop();                     // Infinity loop

        return 0;
}
```

Matrices de projection

- **gluPerspective** modifie la matrice sélectionnée (GL_PROJECTION) afin de lui paramétrer une projection en perspective .



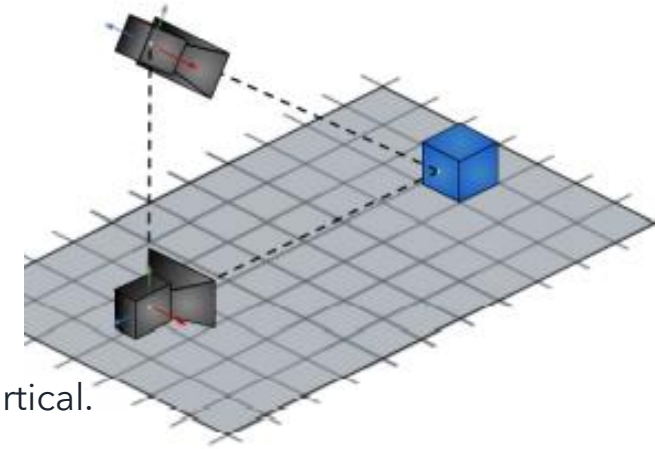
- **gluOrtho2D** modifie la matrice sélectionnée (GL_PROJECTION) afin de lui paramétrer une projection orthogonale .




```
void Timer(int value)
{
        // Your code.
        glutTimerFunc(25, &Timer, 0);
}

void Reshape(int w, int h)
{
        glViewport(0, 0, w, h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45, float(w) / float(h), 0.1, 100);
}

int main(int argc, char *argv[])
{
        // GLUT initialization
        glutReshapeFunc(&Reshape);          // Window's Reshaped callback
        glutDisplayFunc(&Draw);             // Window's callback
        glutTimerFunc(0, &Timer, 0);

        glutKeyboardFunc(&Keyboard);        // Keyboard's callback

        glutMainLoop();                     // Infinity loop

        return 0;
}
```

Matrice de vue

- Pour modifier le point de vue de la caméra dans la scène OpenGL on utilise **gluLookAt**.



- **gluLookAt**(**eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ**)
- La **position** de la caméra est définie par eyeX, eyeY et eyeZ.
- Le **point regardé** par la caméra est définie par centerX, centerY et centerZ.
- L'**axe de référence** de la caméra est défini par upX, upY et upZ qui représentent son vecteur vertical.

- Lorsqu'on l'utilise, l'ensemble des éléments (le monde 3D) est déplacé pour correspondre au point de vue de la caméra.
- Attention à son influence sur la matrice GL MODELVIEW !

GLU

- GLU, pour OpenGL Utility –à ne pas confondre avec GLUT, est un bibliothèque annexe d'OpenGL Cette bibliothèque propose des fonctions utilitaires nous simplifiant le travail avec OpenGL.

- Ces fonctions ont pour préfixe '**glu**'.
- En l'occurrence, gluPerspective et gluLookat permettent de charger la pile de matrice sans avoir besoin de se préoccuper des différentes transformations sous-jacentes.

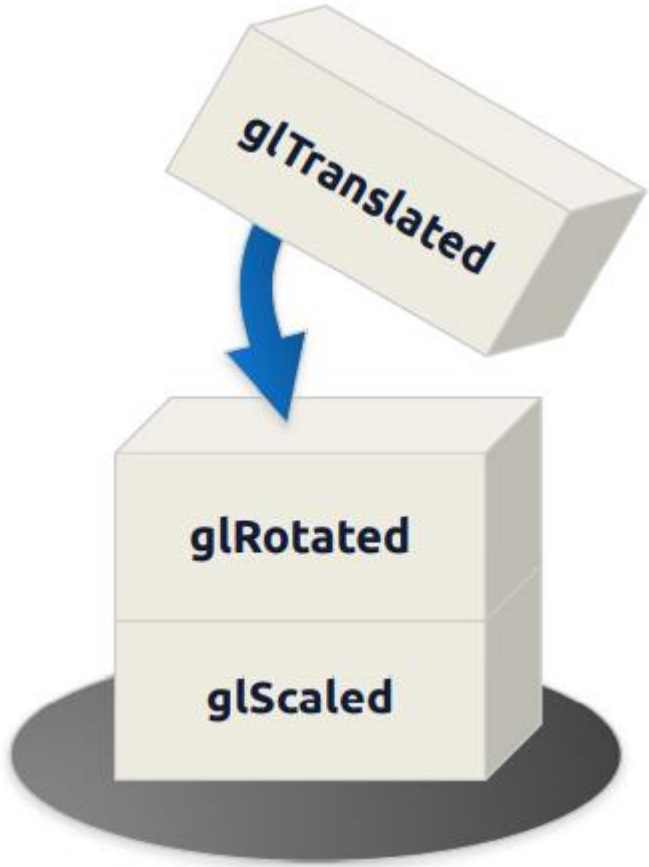
Transformations affines et linéaires

- Pour appliquer une transformation à notre matrice courante, on utilise :

• **glTranslated** pour réaliser une translation

• **glRotated** pour réaliser une rotation

• **glScaled** pour réaliser une homothétie (changement d' échelle)



Ces fonctions existent en version simple précision 'f'

Les matrices « MODELVIEW »

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Identity Matrix

$$\begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glTranslatef(tx,ty,tz)

$$\begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glScalef(sx,sy,sz)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(d) & -\sin(d) & 0 \\ 0 & \sin(d) & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,1,0,0)

$$\begin{pmatrix} \cos(d) & 0 & \sin(d) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(d) & 0 & \cos(d) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,0,1,0)

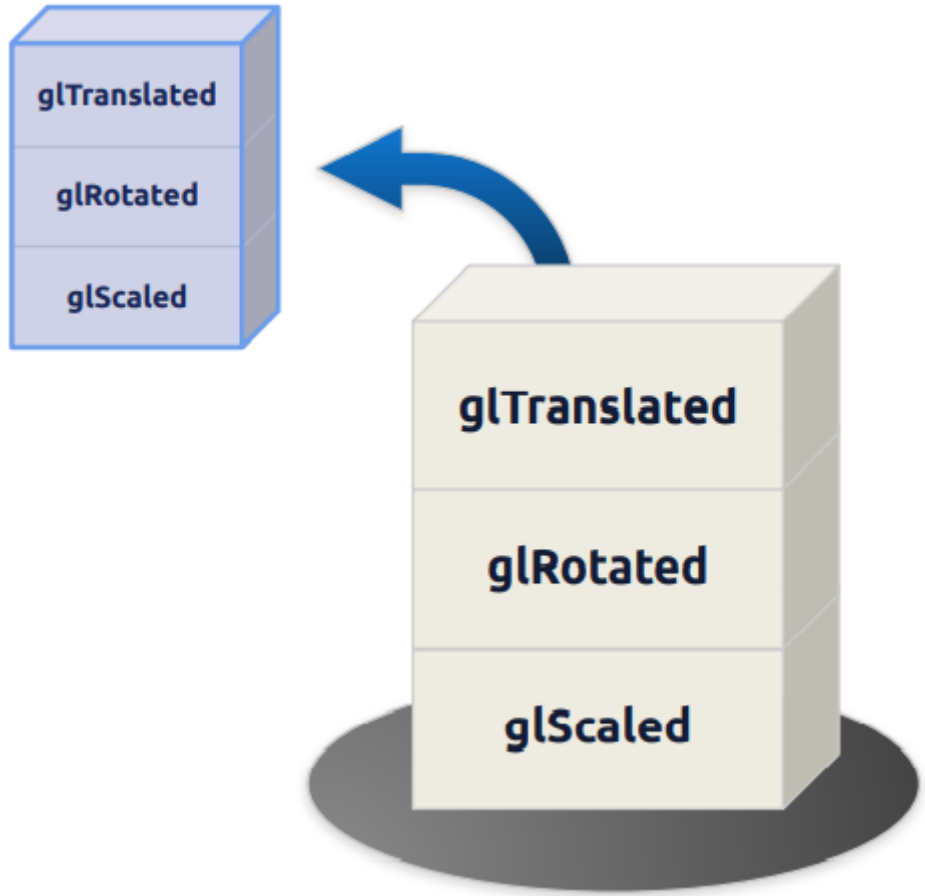
$$\begin{pmatrix} \cos(d) & -\sin(d) & 0 & 0 \\ \sin(d) & \cos(d) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

glRotatef(d,0,0,1)

Pile de matrices : sauvegarde

- On peut sauvegarder une ou plusieurs matrices via la fonction **glPushMatrix**.

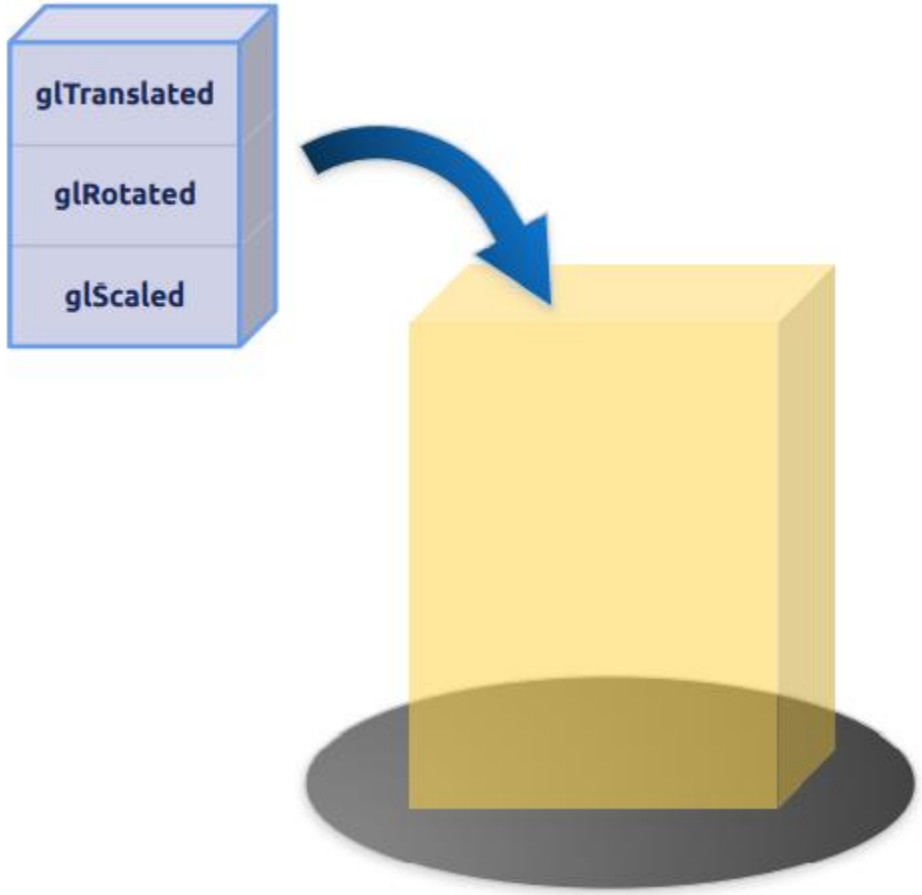
- La pile de matrices est ainsi conservée pour un usage ultérieur.



Pile de matrices : restitution

- Après avoir réinitialisé notre matrice via la fonction **glLoadIdentity**,

- on peut restituer la pile de matrices sauvegardée grâce à la fonction **glPopMatrix**.



Textures

- Une **texture** est la représentation interne d'une **image (*bitmap*)**.
- Grâce à **glGenTextures** on peut générer un tableau de textures.
- Les textures seront accessibles via un tableau d'IDs correspondants.
- Cette opération évite de transférer plusieurs fois une même texture (lors de changement d'objet par exemple).
- On sélectionne la texture de son choix grâce à la fonction **glBindTexture**.
- Celle-ci met à jour la texture courante en se servant d'un ID transmis en paramètre.
- **glTexImage2D** permet de paramétrer la texture courante. Dans le cas où l'on affiche une texture qui évolue au cours du temps, on peut utiliser **glTexSubImage2D** afin de transférer uniquement les parties modifiées.

Paramètres de texture

- On peut définir de nombreux paramètres pour gérer nos textures à l'aide de la fonction **glTexParameteri** : (voir documentation)

• **échantillonnage** de la texture (GL_REPEAT, GL_MIRRORED_REPEAT GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER)

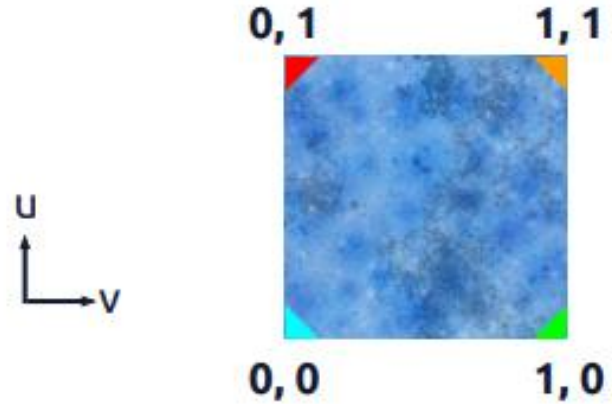
• **filtrage** de la texture (GL_LINEAR, GL_NEAREST, etc.)

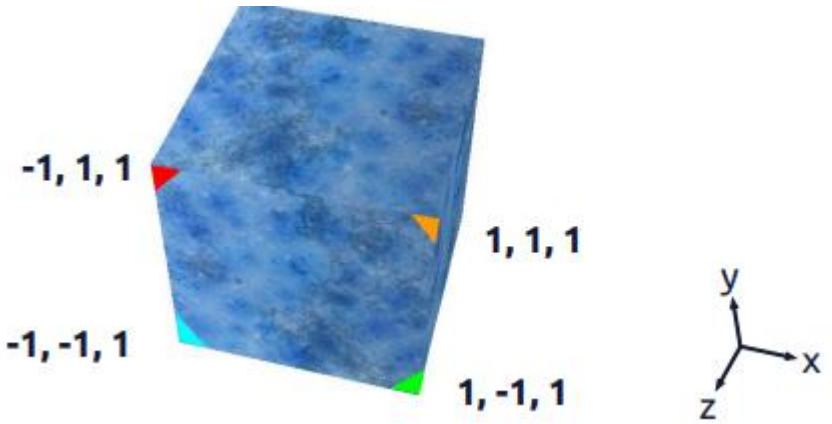
Par défaut la gestion est désactivée, il faudra donc penser utiliser la fonction **glEnable**(GL_TEXTURE_2D).

Coordonnées de texture

- On utilise **glTexCoord2f(u,v)** juste avant de dessiner un vertex pour indiquer sur quel sommet le coin de la texture doit être. plaqué.

```
glBegin(GL_QUADS);

        glTexCoord2f(0.0F,0.0F);
        glVertex3f(-1.0F, -1.0F, 1.0F);

        glTexCoord2f(1.0F ,0.0F);
        glVertex3f(1.0F, -1.0F, 1.0F);

        glTexCoord2f(1.0F, 1.0F);
        glVertex3f(1.0F, 1.0F, 1.0F);

        glTexCoord2f(0.0F, 1.0F);
        glVertex3f(-1.0F, 1.0F, 1.0F);

        // Others faces

glEnd();
```





Illumination

- Par défaut OpenGL propose un éclairage dit ambiant. Toutes les faces de chaque élément de votre scène sont ainsi éclairées de manière homogène, sans variation d'intensité lumineuse.

- En réalité on peut **activer** un **éclairage** plus poussé via la fonction **glEnable**(GL_LIGHTING).

- Il existe **GL_MAX_LIGHTS** (constante défini par les capacités de l'ordinateur) **sources** de **lumière** utilisable dans une scène OpenGL.

- On peut **activer** une source de **lumière** via **glEnable**(GL_LIGHTi), où ' i ' représente une valeur entre 0 et GL_MAX_LIGHTS-1.

Source lumineuse

- Une fois activé, une source de lumière est positionnée par défaut en (0, 0, 1).

- Vous pouvez désactiver cette source à tout moment via la fonction **glDisable**(GL_LIGHTi).

- Pour paramétrer une des sources de lumière activées, on utilise la fonction :

  void **glLightfv**(GLenum *light*, GLenum *pname*, const GLfloat * *params*)

  light représente la source de lumière choisie (Ex : LIGHT23)

  pname représente un paramètre que l'on souhaite modifier

  param représente les valeurs que l'on souhaite attribuer au paramètre choisi

Paramètres des sources lumineuses

- Le paramètre ***pname*** de la fonction **glLightfv** peut être :

GL_AMBIENT = Intensité ambiante RGBA de la lumière. (Par défaut à (0, 0, 0, 1))

GL_DIFFUSE = Intensité RGBA diffuse de la lumière. (Par défaut à (1, 1, 1, 1) pour GL_LIGHT0 et à (0, 0, 0, 1) pour les autres)

GL_SPECULAR = Intensité RGBA spéculaire de la lumière. (Par défaut à (1, 1, 1, 1) pour GL_LIGHT0 et à (0, 0, 0, 1) pour les autres)

GL_POSITION = Position de la lumière dans les coordonnées homogènes de l'objet. (Par défaut à (0, 0, 1, 0))

GL_SPOT_DIRECTION = Direction de la lumière dans les coordonnées homogènes de l'objet. (Par défaut à (0, 0, -1))

GL_SPOT_EXPONENT = Distribution d'intensité de la lumière. (Valeurs comprises entre 0 et 128. Par défaut à 0.)

GL_SPOT_CUTOFF = Angle d'étalement maximum d'une source lumineuse. (Valeurs comprises entre 0 et 90, ou 180. Par défaut à 180.)

GL_CONSTANT_ATTENUATION,

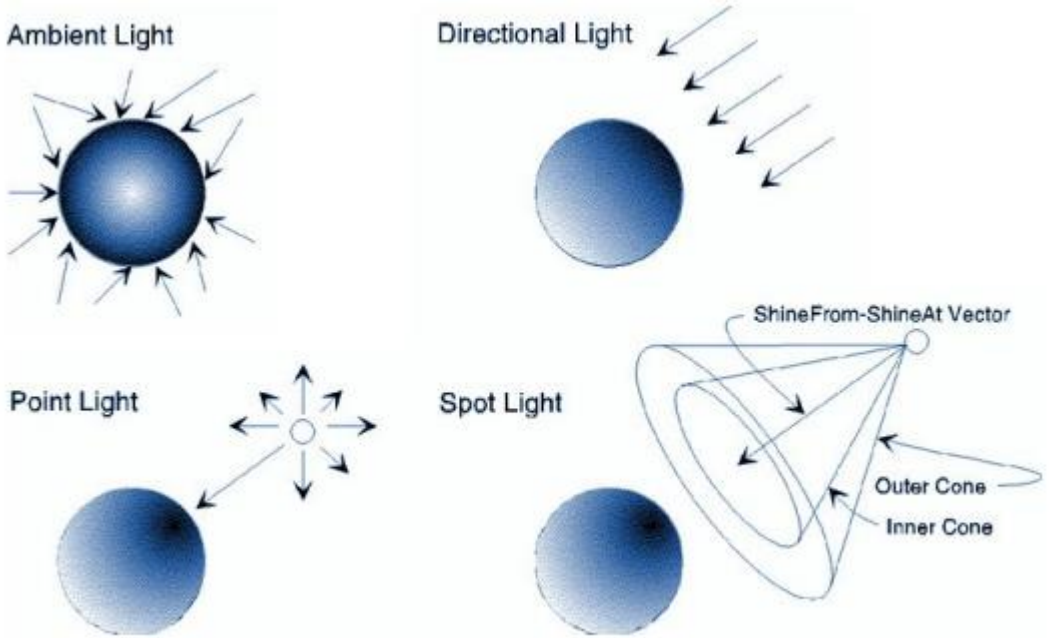
GL_LINEAR_ATTENUATION,

GL_QUADRATIC_ATTENUATION = Spécifie l'un des trois facteurs d'atténuation de la lumière. (Par défaut à (1, 0, 0) : aucune atténuation)

Types de source lumineuse

- Lorsque l'on paramètre une source de lumière, on peut obtenir plusieurs types de lumière

- **Ambiante** (Default Light)
- **Directionnelle** (Directional Light)
- **Ponctuelle omnidirectionnelle** (Point Light)
- **Ponctuelle angulaire** (Spot Light)



Couleurs perceptibles

- Chaque élément de la scène possède 4 paramètres de couleurs distinctes :

● Couleur **ambiante** (visible par défaut, objet sans éclairage)

● Couleur **diffuse** (renvoyé par l’élément lorsqu'il est éclairé)

● Couleur **spéculaire** (reflet lumineux, renvoyé vers l’observateur)

● Couleur d’**émission** (émis directement par l’élément, ne concerne pas les sources de lumière)

Matériaux

- Ces paramètres de couleur sont contenus dans ce que l'on appelle le **matériau** de l'objet.

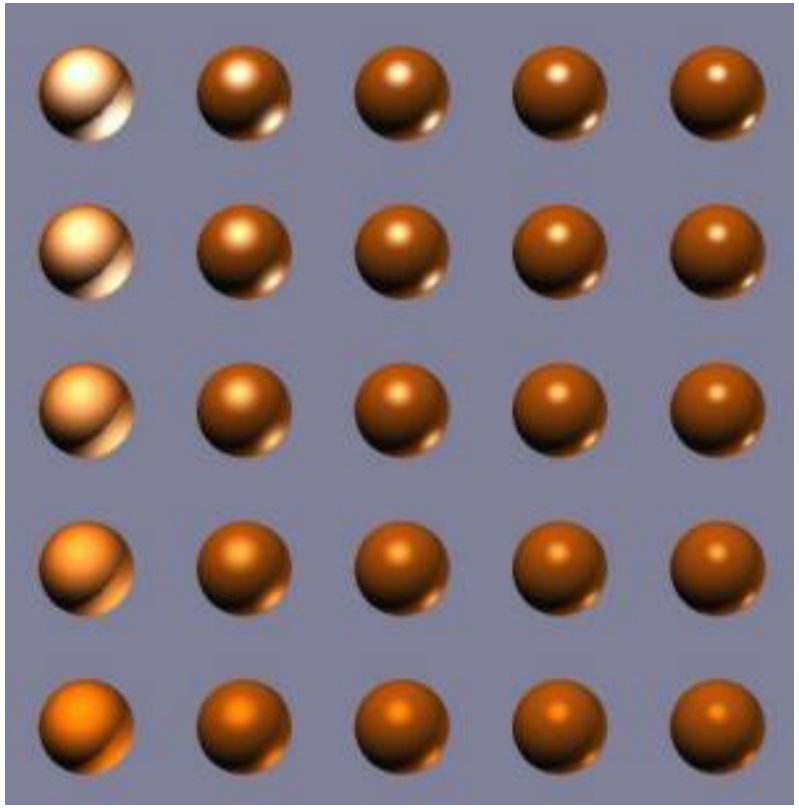
- Pour modifier ce matériau on doit l'activer via **glEnable**(GL_COLOR_MATERIAL).

- On sélectionne le paramètre qui nous intéresse via la fonction **glColorMaterial**(GLenum *face*, GLenum *mode*).

- Puis on modifie sa couleur via la fonction **glColor**. Par défaut glColor modifie la valeur ***GL_AMBIENT_AND_DIFFUSE***.

Paramètres de matériaux

- Pour modifier d'autres paramètres tel que la **brillance** (*shininess*) de la couleur spéculaire du matériau, **absorption** ou la **réflexion** du matériau, on fait appel à la fonction **glMaterial**.



Normales et illumination

- Le rendu général de notre scène nous donne une couleur **ambiante** qui prend en compte la couleur **ambiante de la lumière** et **la couleur ambiante du matériau** de chaque objet.
- De même, pour les couleur **diffuse** et **spéculaire** bien qu'elles prennent en compte également la normale de chaque face d'un élément.
- Lorsque l'on dessine un objet 3D il faudra donc spécifier la normale de chaque face via la fonction **glNormal** de la même façon que glVertex.