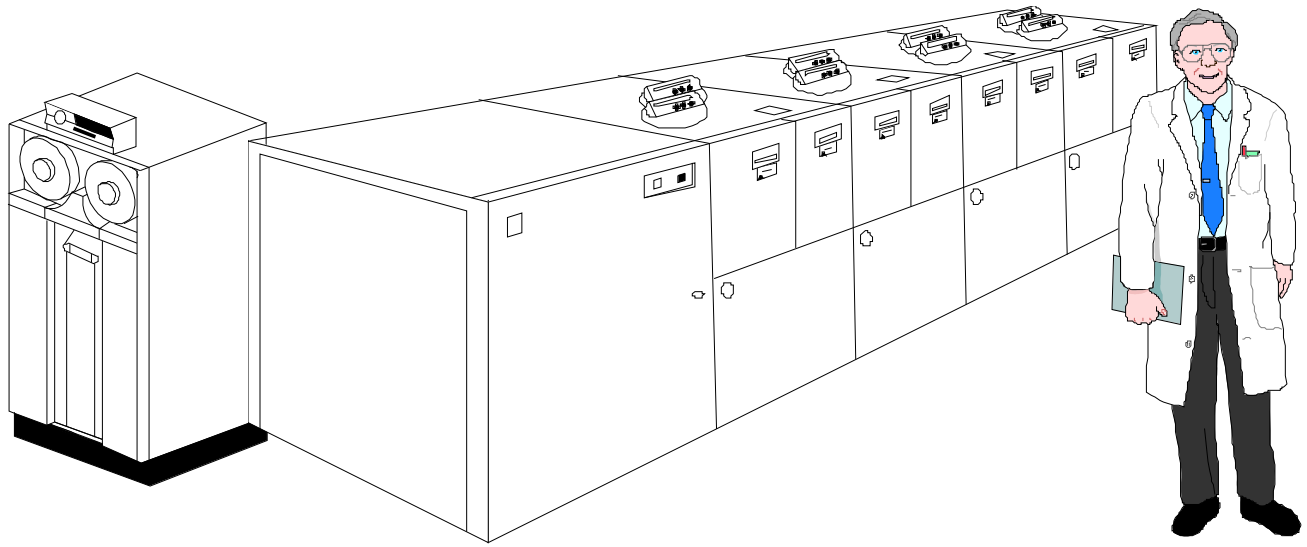
CHAPITRE 1

# Introduction et court historique

# Évolution des systèmes

- Hier (jusqu'aux années 70)
  - le temps d'un ordinateur était beaucoup plus précieux que celui d'un humain
  - l'ordinateur se retrouvait cloîtré et distant du programmeur
  - on travaillait en mode « lot », avec des cartes perforées!
  - l'humain parlait le plus possible le langage de l'ordinateur



# Évolution des systèmes

- Aujourd'hui
  - le coût d'un ordinateur se compare à quelques semaines de salaire d'un programmeur; le principal coût est maintenant l'énergie!
  - le programmeur peut toucher à l'ordinateur
  - on travaille en mode interactif, avec un écran, une souris et un clavier
  - l'ordinateur (et le logiciel) s'efforce de parler le langage de l'humain: p.e. la métaphore du dessus de « bureau ».

# Évolution des librairies graphiques

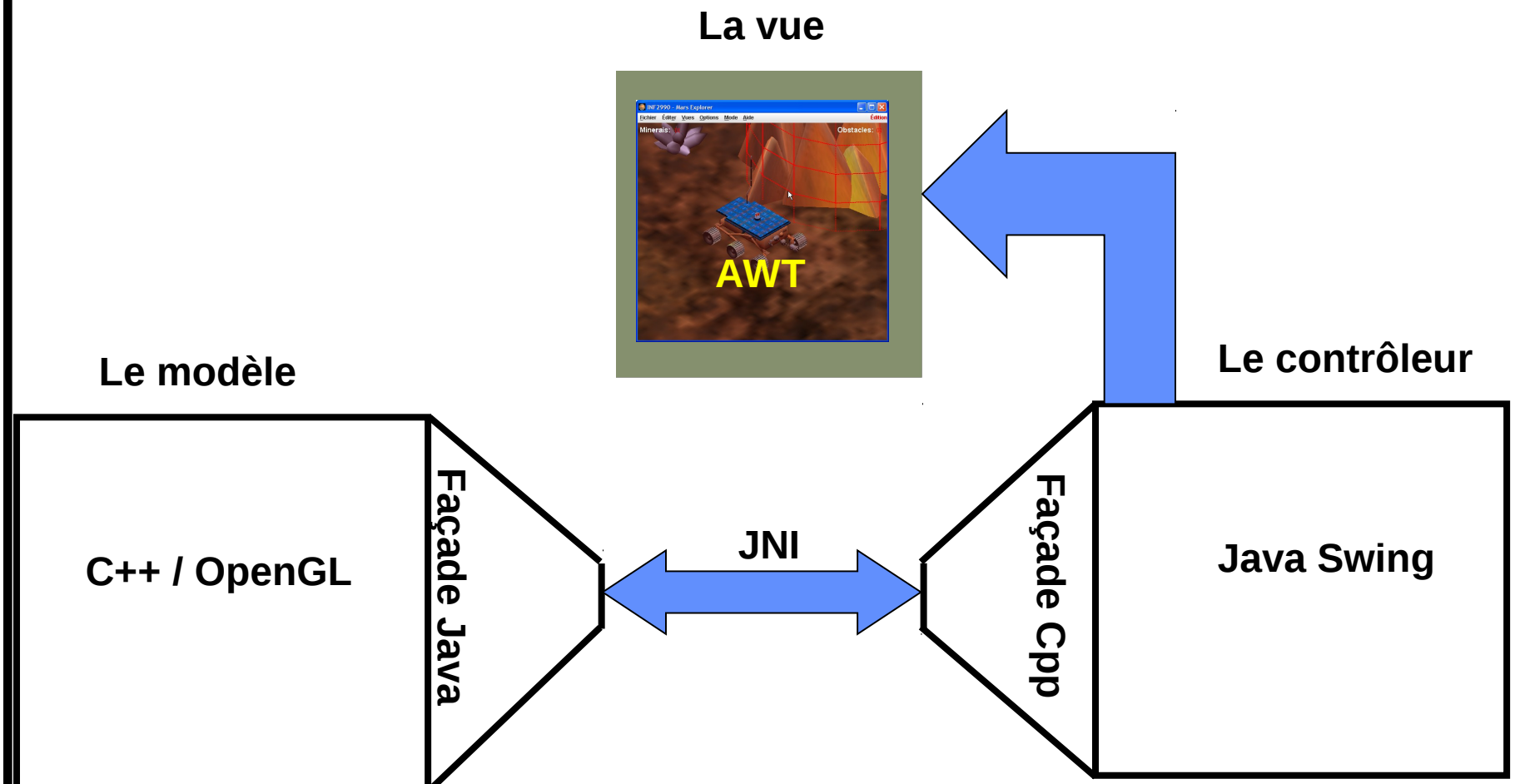
- Au début, chaque compagnie avait une interface spécifique pour son matériel graphique. Ex.: **TCS/PLOT10**.
- Des efforts structurants ont suivis et d'autres librairies plus générales ont vu le jour : **X11, CORE, GKS, GKS-3D, PHIGS**.
- En 1984, la librairie **GL (/IrisGL)** de SGI permet le rendu 3D facile avec menu, événements et affichage de texte.

  C'était la librairie graphique par excellence, allant de pair avec de très bonnes stations graphiques.
- En 1992, **OpenGL** 1.0 est créé par SGI et d'autres manufacturiers. Cette librairie est essentiellement la même du téléphone cellulaire au super-ordinateur.
- Aujourd'hui, avec **OpenGL** et **Direct3D** sont les deux seules librairies générales réellement présentes sur le marché.

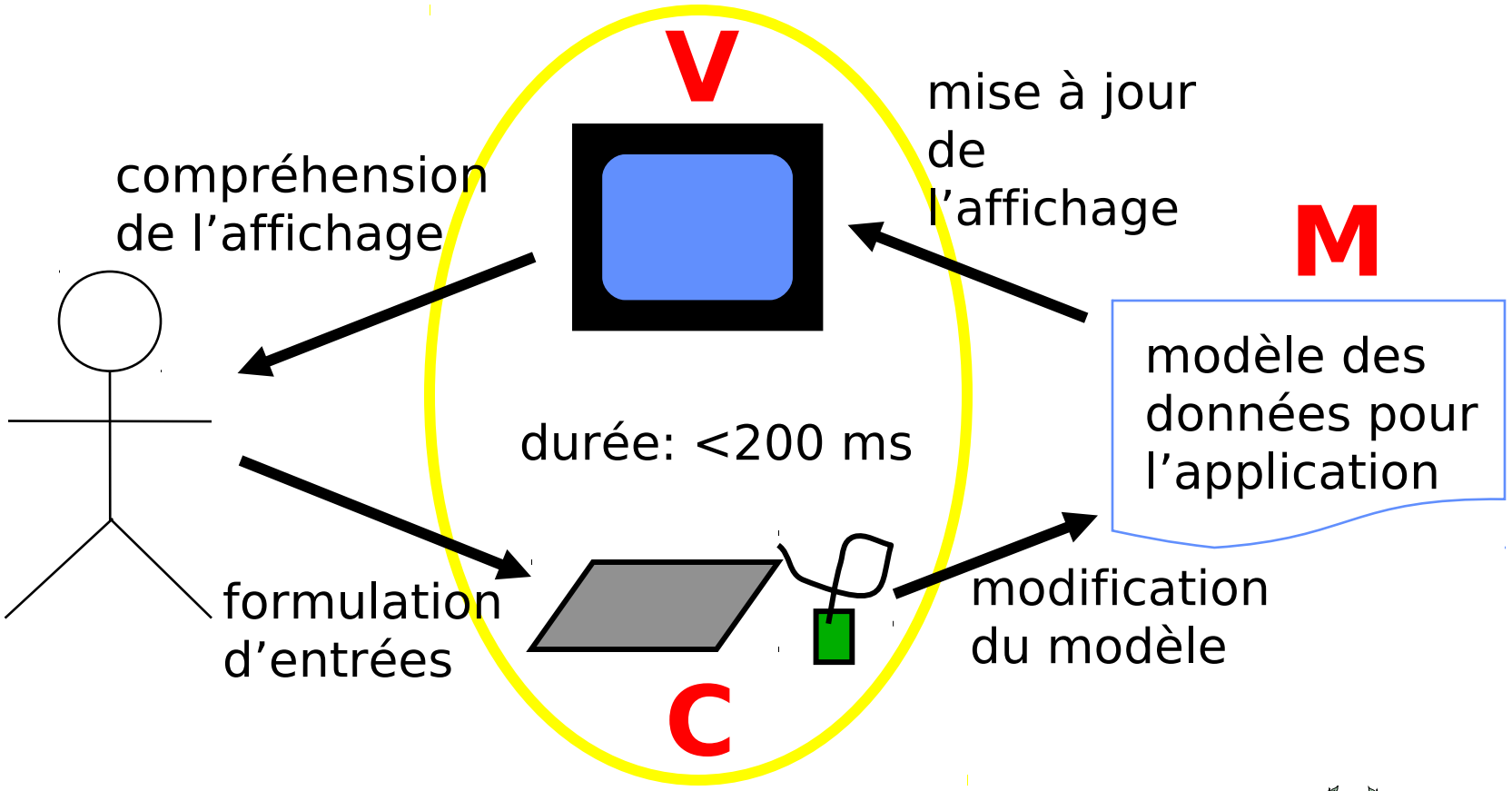
# Physiologie humaine de la vision et infographie

- perception de mouvement continu
  - 30 images/seconde dépasse la fréquence d'échantillonnage à la rétine par le système visuel chez l'humain
  - p.e. un film présente 24 images/seconde
  - Pour une application où tous les pixels sont traités d'une image à l'autre, cela laisse peu d'instructions pour le faire
- affichage en continu et animation
  - L'expérimentation a démontré que 5 à 10 mises à jour/seconde est la limite inférieure acceptable pour maintenir l'interactivité
  - Pour faire des animations : il suffit de modifier et retracer l'écran régulièrement pour donner la sensation que la scène est dynamique!
  - Pour afficher des animations, un logiciel graphique réaffiche donc continuellement la scène

# Architecture logicielle: modèle MVC (INF2990)



# Le cycle interactif




**interface personne-machine**

# Le cycle interactif

- *Architecture MVC: modèle vue contrôleur (*« model view controller »)
  - proposé par SmallTalk (Xerox PARC, années ’70)
  - **M**odèle («model»)
    - le modèle de données de l’application
  - **V**ue («view»)
    - l’apparence de l’application à l’écran
    - la disposition des différents éléments visuels
  - **C**ontrôleur («controller»)
    - ce qui permet de faire le raccord entre l’usager et l’application
    - matériel : souris, clavier
    - logiciel : boutons, menus, fonctions de rappel («callbacks») et autres mécanismes

# Notions de base

# Trois principaux acteurs pour afficher une primitive géométrique

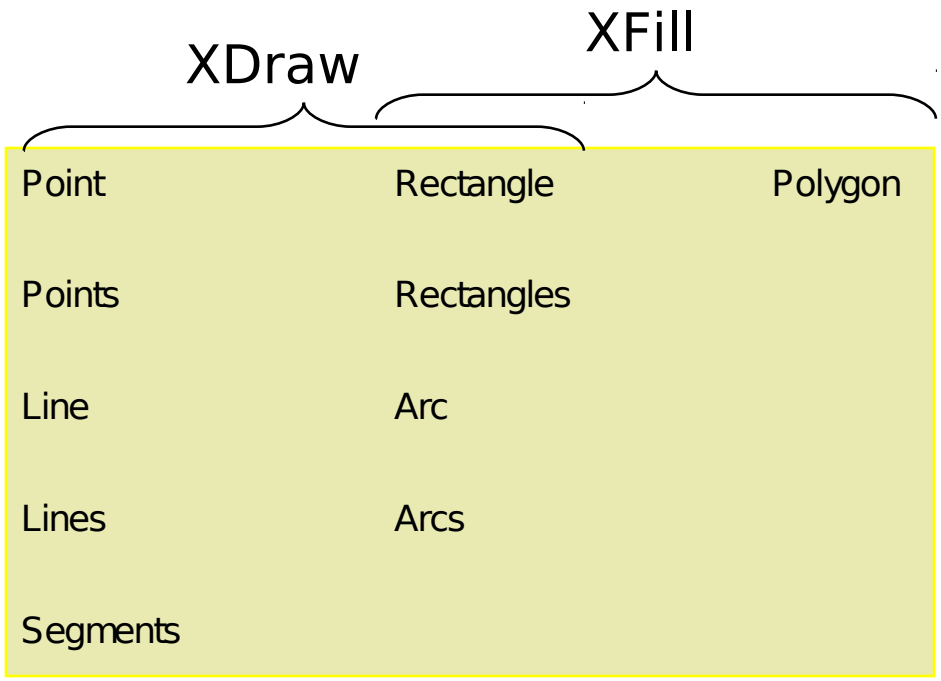
- Primitive graphique ( / géométrique )
  - indique la forme générale de ce qu'on doit dessiner
  - p.e. segment, arc, rectangle, courbe, caractères
- Contexte graphique
  - indique le « comment », avec tous les détails pertinents
  - p.e. épaisseur des traits, couleur, texture
- Possiblement, une table de couleurs (« colormap »)
  - opère la traduction entre valeur de pixel et couleur (triplet RGB)

# Pourquoi séparer l'information géométrique (primitive graphique) du reste (contexte graphique)?

- Beaucoup moins de paramètres à donner pour afficher une primitive
- On souhaite souvent afficher plusieurs primitives dans un même contexte (graphique) ou dans un contexte légèrement différent ; on évite ainsi de répéter l'information
- Dans une architecture client-serveur, si le contexte graphique est conservé chez le serveur, beaucoup moins d'information est transmise à travers le réseau à chaque affichage

# Primitives géométriques -- *X*

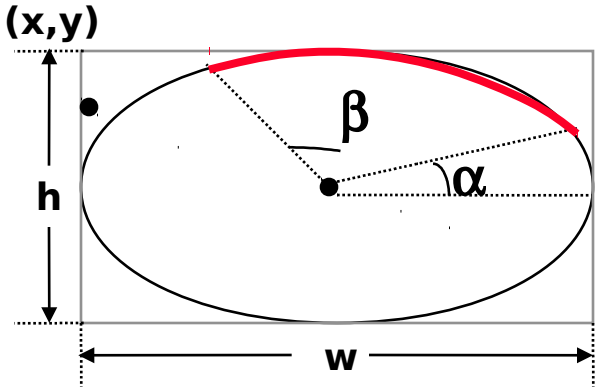
- Syntaxe




- p.e. XDrawPoint(...), XDrawArcs(...), XFillArcs(...), ...

# Primitives géométriques -- *X*

- XDrawArc(<écran>, <fenêtre>, <contexte graphique>, x, y, w, h, angle1, angle2)




  - int x, y : coordonnées du coin supérieur gauche du rectangle englobant
  - int w, h : largeur et hauteur de ce rectangle, respectivement
  - int angle1 : point de départ de l'arc [64 $\times \alpha$ (en degrés)]
  - int angle2 : étendue de l'arc [64 $\times \beta$ (en degrés)]
    - une valeur négative donne un arc en direction horaire

# Primitives géométriques -- *OpenGL*

- Les sommets sont le fondement des primitives géométriques

  `glVertex{2,3,4}{s,i,f,d}[v]()`

- Ils doivent être spécifiés entre un `glBegin()` et un `glEnd()`
- La spécification d'un *mode* indique ce qu'on en fait

```
glBegin(mode);
   glVertex*(coordonnées);
   glVertex*(coordonnées);
.
.
   glVertex*(coordonnées);
glEnd();
```

MODES
GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_POLYGON
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUADS
GL_QUAD_STRIP

# Primitives géométriques -- *OpenGL* : glVertex* (coordonnées)

```
void glVertex{2,3,4}{s,i,f,d}[v] (TYPE coords);
```

*2 (x, y)*
*3 (x, y, z)*
*4 (x, y, z, h)*

*s entier 16 [short]*
*i entier 32 [int]*
*f flottant [float]*
*d double [double]*

*v tableau -- vecteur*

*TYPE -- GLshort -- s*
*GLfloat -- f*
*Gldouble -- d*
*GLint -- i*
*GLxx * -- v*

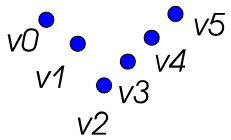
# Primitives géométriques -- OpenGL : glVertex* (coordonnées)

- Exemples

```
glVertex2s( 2, 3 );
glVertex3d( 0.0, 0.0, 3.45 );
glVertex4f( 2.3, 12.0, -4.8, 1.0 );

GLdouble dvect[3]={ 5.0, 9.0, 1435.0 };
glVertex3dv(dvect);

typedef struct {
  GLfloat x, y, z;
} Point3D;

Point3D pt = {0.0, 4.0, 6.0};
glVertex3fv((GLfloat *) &pt); // (pas bien beau!)
```

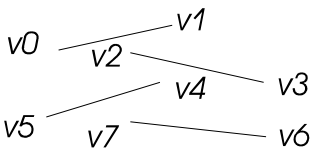
# Primitives géométriques -- OpenGL : modes

- Points

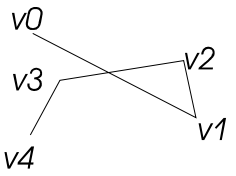



GL_POINTS

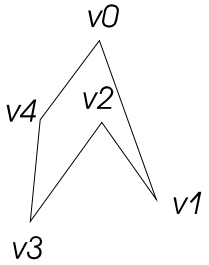
- Segments de droite
  - segments indépendants
  - polylignes
  - polygones vides




GL_LINES

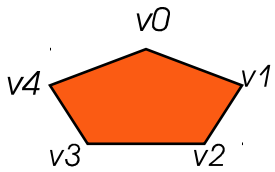



GL_LINE_STRIP

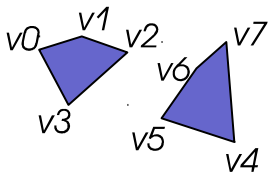



GL_LINE_LOOP

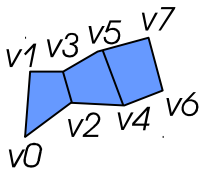
- Remplissage
  - polygones, triangles et quadrilatères indépendants *(simples et convexes)*
  - triangles en éventail, bande de triangles, bande de quadrilatères



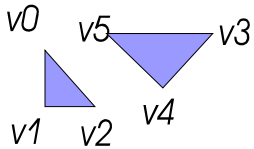

GL_POLYGON



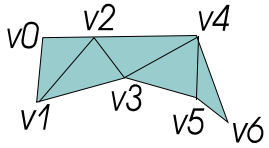

GL_QUADS



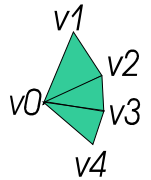

GL_QUAD_STRIP




GL_TRIANGLES



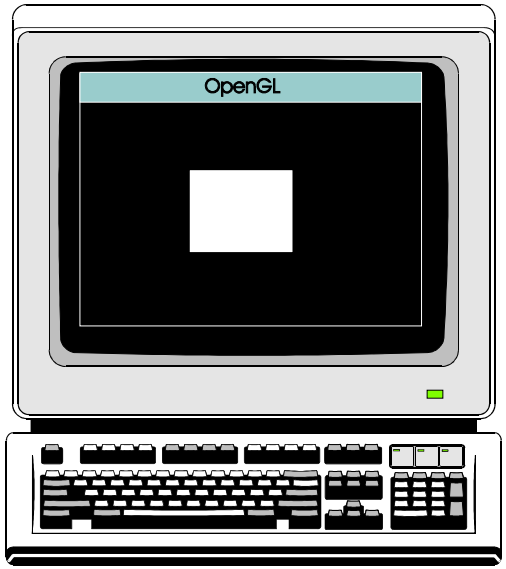

GL_TRIANGLE_STRIP




GL_TRIANGLE_FAN

# Primitives géométriques -- *OpenGL :* petit exemple

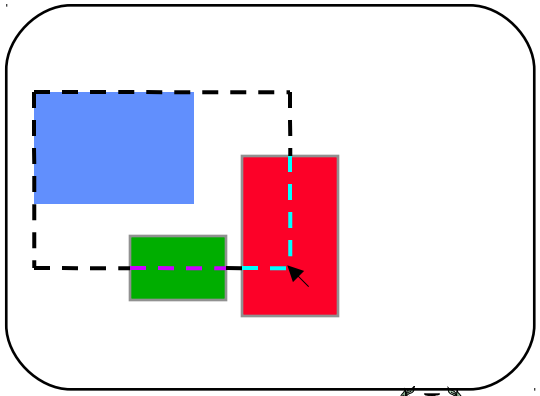
```
#include <GL/glut.h>
int main( int argc, char** argv )
{
   glutInit( &argc, argv );
   glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB );
   glutInitWindowSize( 200, 200 );
   glutInitWindowPosition( 100, 100 );
   glutCreateWindow( argv[0] );

   glClearColor( 0.0, 0.0, 0.0, 0.0 );
   glClear( GL_COLOR_BUFFER_BIT );
   glColor3f( 1.0, 1.0, 1.0 );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   glOrtho( -1.0, 1.0, -1.0, 1.0, -1.0, 1.0 );
   glBegin( GL_QUADS );
        glVertex2f( -0.5, -0.5 );
        glVertex2f( -0.5, 0.5 );
        glVertex2f( 0.5, 0.5 );
        glVertex2f( 0.5, -0.5 );
   glEnd();
}
```



# Contexte graphique

# Contexte graphique

- Après la primitive graphique, le second acteur important est le contexte graphique :
  - sélection des pixels à « allumer »
  - attribution de la « couleur » (valeur des pixels)
  - sélection des plans utilisés (p.e. permet la superposition de graphiques)
  - combinaison logique de la valeur calculée d'un pixel (source) avec la valeur courante du même pixel dans la mémoire de trame (destination)
    - p.e. source, source XOR destination
    - Rectangle élastique



# Contexte graphique -- *OpenGL*

- OpenGL définit plusieurs centaines de **variables d'état** pour établir entre autres le contexte graphique
- variables booléennes
  - p.e. GL_LINE_STIPPLE, GL_BLEND
  - modificateur: `void glEnable(), void glDisable()`
  - accesseur: `GLboolean glIsEnabled()`
- autres variables
  - p.e. GL_CURRENT_COLOR, GL_LINE_WIDTH
  - modificateur: chacune a sa fonction
    - p.e. `void glColor3f(), void glLineWidth()`
  - accesseur: quelques fonctions générales selon le type de données

```
void glGetBooleanv(GLenum nomVar, GLboolean *dest)

void glGetIntegerv(GLenum nomVar, GLint *dest)

void glGetFloatv(Glenum nomVar, GLfloat *dest)
```

# Contexte graphique -- *OpenGL*

- Point
  - Couleur (nous en reparlerons)
  - Taille

Taille

```
void glPointSize(GLfloat taille);
```

où ***taille*** indique la taille en pixels

```
glGetFloatv(GL_POINT_SIZE, &taille);
```

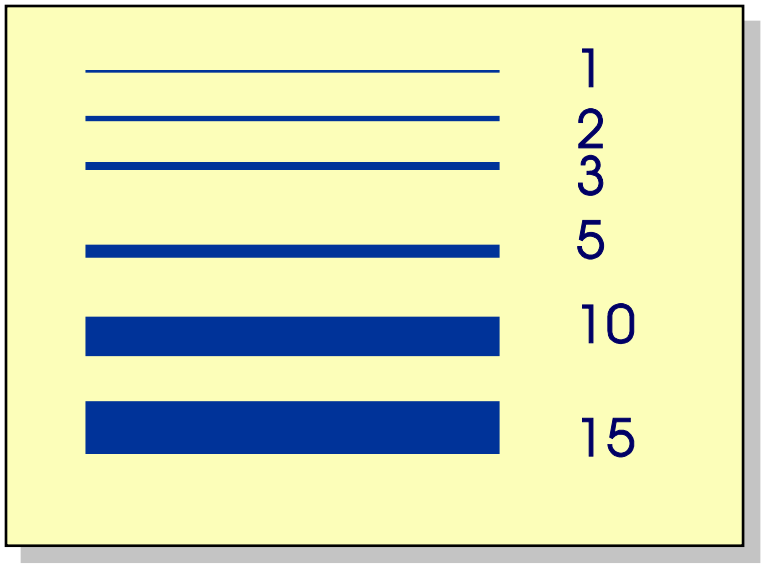
# Contexte graphique -- *OpenGL*

- Épaisseur du trait

```
void glLineWidth(GLfloat largeur);
```

où *largeur* représente la largeur du trait en pixels

```
glGetFloatv(GL_LINE_WIDTH, &largeur);
```



# Contexte graphique -- *OpenGL*

- Type de trait

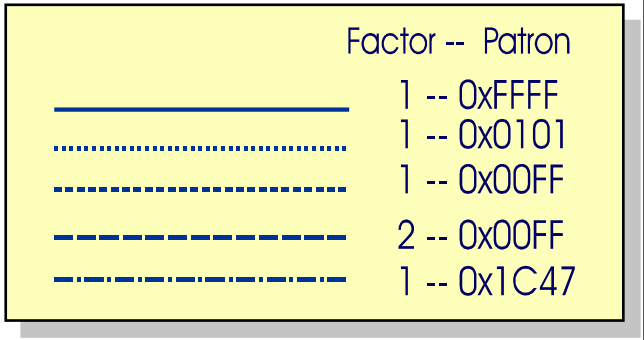
```
void glLineStipple( GLint factor, GLushort pattern );
```

où

*factor* représente un facteur d'étirement du patron

*pattern* patron de 16 bits où 1 =>tracé

0 => pas tracé




***p.e.***

```
glLineStipple(1,0x1C47);
glEnable(GL_LINE_STIPPLE);
glBegin(GL_LINE_STRIP);
glVertex2i(5,5); glVertex2i(10,7); glVertex2i(3,15);
glEnd();
glDisable(GL_LINE_STIPPLE);
```

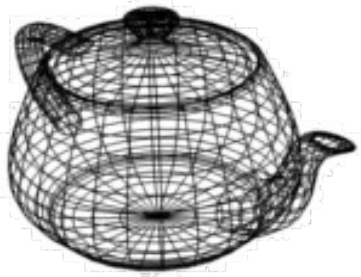
# Contexte graphique -- *OpenGL*

- Type de remplissage

```
void glPolygonMode( GLenum face, GLenum mode);
```

*face* spécifie une face : GL_FRONT_AND_BACK, GL_FRONT, GL_BACK

*mode* indique le niveau de rendu: GL_POINT, GL_LINE, GL_FILL



# Contexte graphique -- *OpenGL:* exemples modificateurs et accesseurs

```
GLboolean valid;
glGetBooleanv( GL_CURRENT_RASTER_POSITION_VALID, &valid );

GLdouble ModelMatrix[16];
GLdouble ProjectMatrix[16];
GLint ViewPort[4];
glGetDoublev( GL_MODELVIEW_MATRIX, ModelMatrix );
glGetDoublev( GL_PROJECTION_MATRIX, ProjectMatrix );
glGetIntegerv( GL_VIEWPORT, ViewPort );

glPolygonMode( GL_FRONT, GL_FILL );
glPolygonMode( GL_BACK, GL_LINE );
GLint vals[2];
glGetIntegerv( GL_POLYGON_MODE, vals );
// vals[] contient alors { GL_FILL, GL_LINE }

glClearColor(0.0, 0.0, 0.0, 1.0);
GLfloat clearcolor[4];
glGetFloatv( GL_COLOR_CLEAR_VALUE, clearcolor );
```

# Contexte graphique -- *OpenGL*

- Affichage sélectif
  - Orientation des faces : Par convention, la *face avant* est celle où l'ordre des sommets est dans le sens anti-horaire (*counterclockwise*)

```
void glFrontFace( GLenum mode );
```

où *mode* indique l'orientation de la face avant: GL_CCW (défaut) ou GL_CW.

  - Élimination de certaines faces

```
void glCullFace( Glenum mode );
```

où *mode* indique l'orientation de la face avant : GL_FRONT, GL_BACK (défaut), ou même GL_FRONT_AND_BACK.

- doit être activé avec `glEnable( GL_CULL_FACE );`

# Couleur -- *OpenGL*

Deux modes, direct et indexé, sont disponibles (mais on ne peut changer dynamiquement)

- Mode direct (RGB*A*)
  - modificateur: `void glColor*(…);`
  - accesseur: p.e. `GLfloat couleur[4];`

    `glGetFloatv( GL_CURRENT_COLOR, couleur );`
  - réinitialiser la mémoire de trame:

    `glClear( GL_COLOR_BUFFER_BIT );`
  - établir la couleur de réinitialisation:

```
void glClearColor( GLclampf rouge, GLclampf vert,
                   GLclampf bleu, GLclampf alpha );
```

# Couleur -- *OpenGL*

## Couleur -- RGBA

```
void glColor3{b,s,i,f,d,ub,us,ui}(TYPE r,TYPE g,TYPE b );
void glColor4{b,s,i,f,d,ub,us,ui}(TYPE r,TYPE g,TYPE b,TYPE a);
void glColor{3 4}{b,s,i,f,d,ub,us,ui}v(const TYPE *v );
```

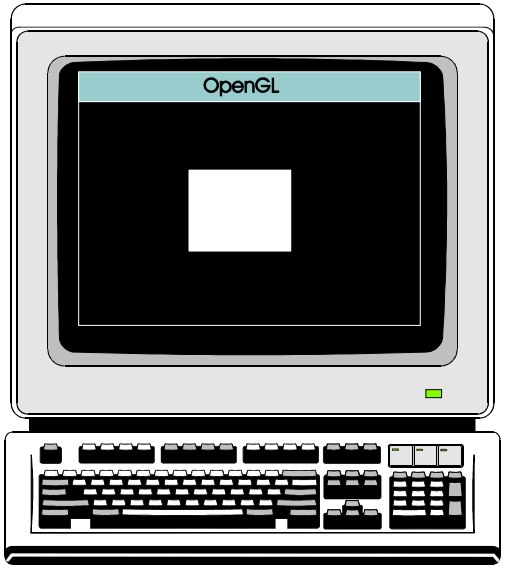
où

*r* rouge
*g* vert
*b* bleu
*a* alpha
*v* vecteur de 3 ou 4 valeurs

*b -- byte*
*s -- short*
*i -- int*
*f -- float*
*d -- double*
*ub-- unsigned byte*
*us-- unsigned short*
*ui-- unsigned int*

# Couleur -- *OpenGL* retour sur le petit exemple

```
#include <GL/glut.h>
int main( int argc, char** argv )
{
   glutInit( &argc, argv );
   glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB );
   glutInitWindowSize( 200, 200 );
   glutInitWindowPosition( 100, 100 );
   glutCreateWindow( argv[0] );

   glClearColor( 0.0, 0.0, 0.0, 0.0 );
   glClear( GL_COLOR_BUFFER_BIT );
   glColor3f( 1.0, 1.0, 1.0 );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   glOrtho( -1.0, 1.0, -1.0, 1.0, -1.0, 1.0 );
   glBegin( GL_QUADS );
    glVertex2f( -0.5, -0.5 );
    glVertex2f( -0.5, 0.5 );
    glVertex2f( 0.5, 0.5 );
    glVertex2f( 0.5, -0.5 );
   glEnd();
}
```



# Couleur -- *OpenGL*

- Mode indexé
  - modificateur: `void glIndex*(…);`
  - accesseur: p.e. `glGetIntegerv(GL_CURRENT_INDEX, &index);`
  - réinitialiser la mémoire de trame:
    `glClear(GL_COLOR_BUFFER_BIT);`
  - établir la couleur de réinitialisation:
    `void glClearIndex(Glfloat index);`

# Utilisation de tableaux de sommets

# Primitives géométriques -- *OpenGL :* utilisation de tableaux

- 1) Au lieu de transmettre les sommets un à un avec glVertex(), on peut utiliser plutôt un tableau de sommets, afin de réduire le nombre d'appels OpenGL pour tracer une primitive

  - Pour tracer les six faces d'un cube avec la primitive GL_QUADS, on aurait toutefois encore besoin de définir 6 faces * 4 sommets/face = tableau de 24 sommets

- 2) Afin d'éliminer la redondance des sommets, on peut définir un tableau de sommets uniques et un tableau de connectivité qui indique comment relier ces sommets

  - Pour tracer les six faces d'un cube avec cette stratégie, on aurait alors besoin d'un tableau de 8 sommets et d'un autre tableau de 6 faces * 4 indices/face = 24 indices

# Primitives géométriques -- *OpenGL :* tableau de sommets

- 1) Exemple d'utilisation d'un tableau de sommets:

```
// définir les sommets
GLfloat sommets[24] = {...}; // les coordonnées (x,y,z)

// activer et spécifier un pointeur vers les sommets
glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer( 3, GL_FLOAT, 0, sommets );

// tracer la primitive
glDrawArrays( GL_QUADS, 0, 24 );

// désactiver l'utilisation des tableaux de sommets
glDisableClientState( GL_VERTEX_ARRAY );
```

# Primitives géométriques -- *OpenGL :* tableau de sommets

Activer ou désactiver l'utilisation de tableaux:

```
void glEnableClientState( GLenum cap );
void glDisableClientState( GLenum cap );
```

Où cap est une des constantes:

- GL_VERTEX_ARRAY
- GL_COLOR_ARRAY
- GL_EDGE_FLAG_ARRAY
- GL_INDEX_ARRAY
- GL_NORMAL_ARRAY
- GL_TEXTURE_COORD_ARRAY

# Primitives géométriques -- *OpenGL :* tableau de sommets

Définir un tableau de sommets :

```
void glVertexPointer( GLint taille, GLenum type,
                      GLsizei pas, const GLvoid *ptr );
```

taille : nombre de coordonnées par sommet (2,3, ou 4)

type : type des données (GL_FLOAT, GL_SHORT, GL_INT, GL_DOUBLE)

pas : distance entre les x de chaque sommet

ptr : pointeur sur les sommets

# Primitives géométriques -- *OpenGL :* tableau de sommets

Effectuer le rendu de la primitive :

```
void glDrawArrays( GLenum mode,
                   GLint debut, GLsizei nombre );
```

mode : choix de la primitive.

Les mêmes modes que pour glBegin()/glEnd() :

GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES,
GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES,
GL_QUAD_STRIP, GL_QUADS, GL_POLYGON.

debut: la position de la première valeur

nombre : nombre de valeurs à utiliser pour tracer

# Primitives géométriques -- *OpenGL :* tableau de sommets et connectivité

- 2) Exemple d'utilisation d'un tableau de sommets et d'une connectivité:

```
// définir les sommets
GLfloat sommets[8] = {...}; // les coordonnées (x,y,z)
GLuint connec[24] = {...}; // la connectivitê

// activer et spécifier un pointeur vers les sommets
glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer( 3, GL_FLOAT, 0, sommets );

// tracer la primitive
glDrawElements( GL_QUADS, sizeof(connec)/sizeof(GLuint),
                GL_UNSIGNED_INT, connec );

// désactiver l'utilisation des tableaux de sommets
glDisableClientState( GL_VERTEX_ARRAY );
```

# Primitives géométriques -- *OpenGL :* tableau de sommets et connectivité

Effectuer le rendu de la primitive avec un tableau de connectivité :

```
void glDrawElements( GLenum mode, Glsizei nombre,
                     Glenum type, const Glvoid * connec );
```

mode : choix de la primitive (GL_QUADS ou autre)

nombre : nombre d'éléments à utiliser pour tracer

type : le type des indices (GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT)

connec : un pointeur sur la position de la première valeur

# Primitives géométriques -- *OpenGL :* tableaux sur le serveur

- Plutôt que de conserver les tableaux sur le client, on peut aussi envoyer les valeurs sur le serveur.

- Les *Vertex Buffer Object (VBO)* sont conservés sur le serveur et permettent d'améliorer l'efficacité de l'affichage
- On crée
  - un *VBO* pour les sommets et
  - un *VBO* pour la connectivité

- La création d'un *VBO* se fait en trois étapes:
  1. Générer un id d'objet tampon avec glGenBuffers().
  2. Lier l'objet tampon avec glBindBuffer().
  3. Copier les données dans le tampon avec glBufferData().

# Primitives géométriques -- *OpenGL :* tableaux sur le serveur

Générer ou supprimer un(des) identificateur(s) de tampon :

```
void glGenBuffers( GLsizei n, GLuint* tampons );
void glDeleteBuffers( GLsizei n, const GLuint* tampons );
```

`n`: nombre de noms désiré

`tampons`: tableau de n noms

# Primitives géométriques -- *OpenGL :* tableaux sur le serveur

Lier un objet tampon :

```
void glBindBuffer( GLenum type, GLuint tampon );
```

`type:` le type de tampon par l'une des quatre constantes
`GL_ARRAY_BUFFER` *(pour les sommets),*
`GL_ELEMENT_ARRAY_BUFFER` *(pour la connectivité),*
`GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER`

`tampon:` le nom de l'objet tampon

# Primitives géométriques -- *OpenGL :* tableaux sur le serveur

Copier les données dans le tampon :

```
void glBufferData( GLenum type, GLsizeiptr taille,
                   const GLvoid * donnees, GLenum usage );
```

`type:` le type de l'objet tampon par l'une des mêmes quatre constantes

`taille:` la taille en octets

`donnees:` les données à copier dans l'objet tampon (ou NULL si on ne copie rien)

`usage:` l'utilisation attendue de ce tampon

`GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY,`
**`GL_STATIC_DRAW`**`, GL_STATIC_READ, GL_STATIC_COPY,`
`GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, GL_DYNAMIC_COPY`

# Primitives géométriques -- *OpenGL :* tableaux sur le serveur

- Création d'un tableau de sommets sur le serveur (VBO):

```
// le tableau de sommets
GLfloat* sommets = new GLfloat[nsommets*3];
...
// générer le VBO
GLuint vboId1;
glGenBuffers( 1, &vboId1 );
// lier l'objet tampon afin de pouvoir l'utiliser
glBindBuffer( GL_ARRAY_BUFFER, vboId1 );
// charger le tableau de sommets sur le serveur
glBufferData( GL_ARRAY_BUFFER, sizeof(sommets), sommets,
              GL_STATIC_DRAW );
// si on veut, on peut effacer les données après la création du VBO
delete [] sommets;
...
// supprimer le VBO à la fin du programme
glDeleteBuffers( 1, &vboId1 );
```

# Primitives géométriques -- *OpenGL :* tableaux sur le serveur

- Création d'un tableau de connectivité sur le serveur (VBO):

```
// le tableau de connectivité
GLuint* connec = new GLuint[nfaces*4];
...
// générer le VBO
GLuint vboId2;
glGenBuffers( 1, &vboId2 );
// lier l'objet tampon afin de pouvoir l'utiliser
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, vboId2 );
// charger le tableau de connectivité sur le serveur
glBufferData( GL_ELEMENT_ARRAY_BUFFER, sizeof(connec), connec,
              GL_STATIC_DRAW );
// si on veut, on peut effacer les données après la création du VBO
delete [] connec;
...
// supprimer le VBO à la fin du programme
glDeleteBuffers( 1, &vboId2 );
```

# Primitives géométriques -- *OpenGL :* tableaux sur le serveur

- Affichage d'une primitive avec VBO:

```
// lier VBOs pour les tableaux de sommets et de connectivité
glBindBuffer( GL_ARRAY_BUFFER, vboId1 );         // sommets
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, vboId2 ); // connectivité
// activer et spécifier un pointeur 0 vers les sommets
glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer( 3, GL_FLOAT, 0, 0 ); // ptr nul
// tracer la primitive: 6 faces * 4 indices/face = 24 indices
glDrawElements( GL_QUADS, 24, GL_UNSIGNED_BYTE, 0 ); // ptr nul
// désactiver l'utilisation des tableaux de sommets
glDisableClientState( GL_VERTEX_ARRAY );
// défaire le lien avec les VBO
glBindBuffer( GL_ARRAY_BUFFER, 0 );
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, 0 );
```