Programmation objets, web et mobiles (JAVA)

Programmation OpenGL

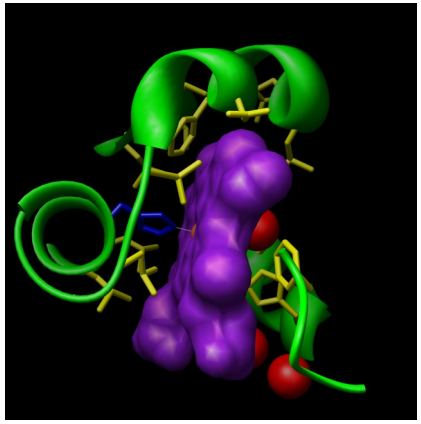
Licence 3 Professionnelle - Multimédia

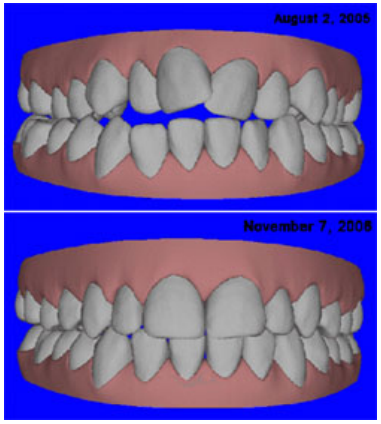
Philippe Esling (esling@ircam.fr)

Maître de conférences – UPMC

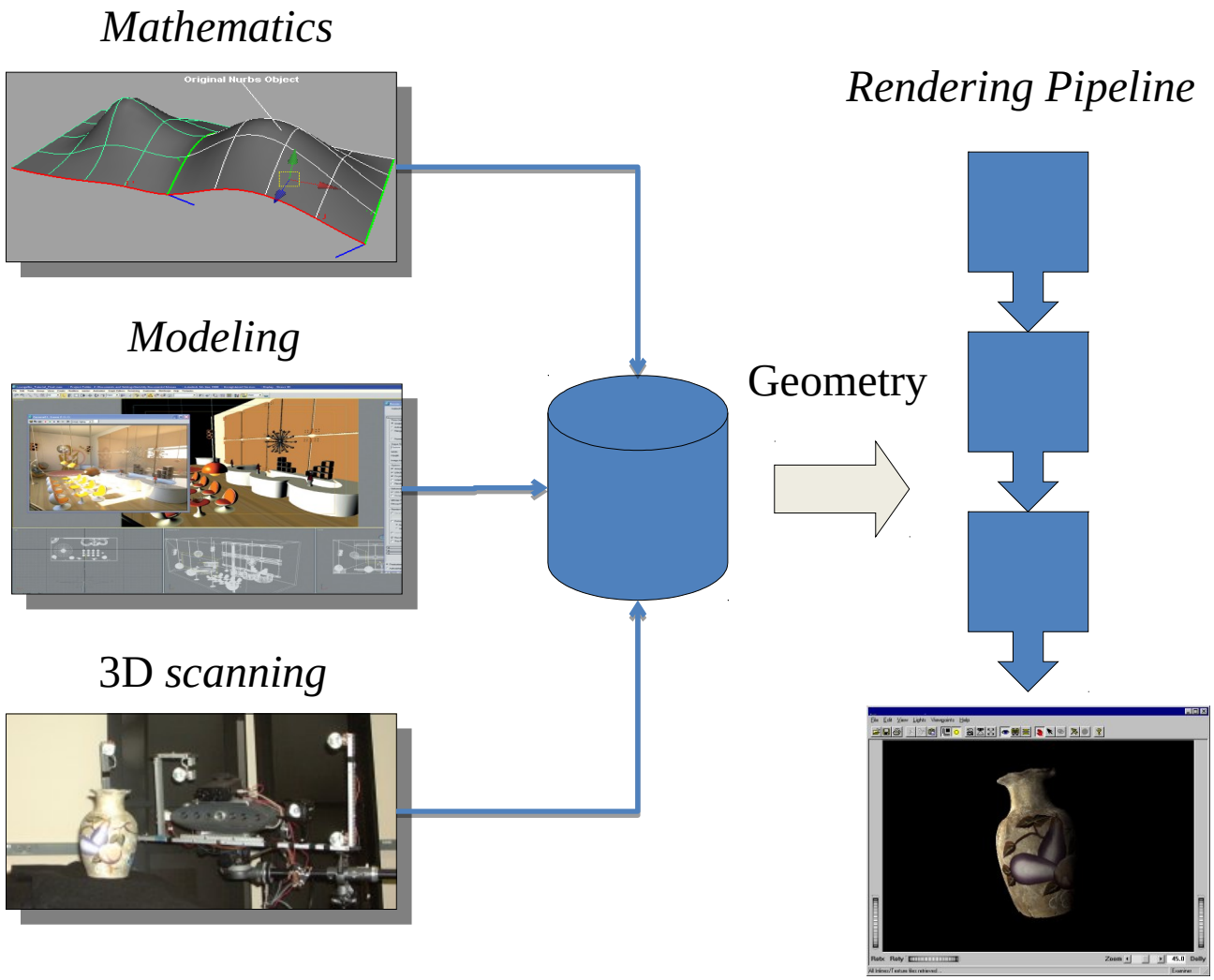
Equipe représentations musicales (IRCAM, Paris)

Synthèse d'images 3D

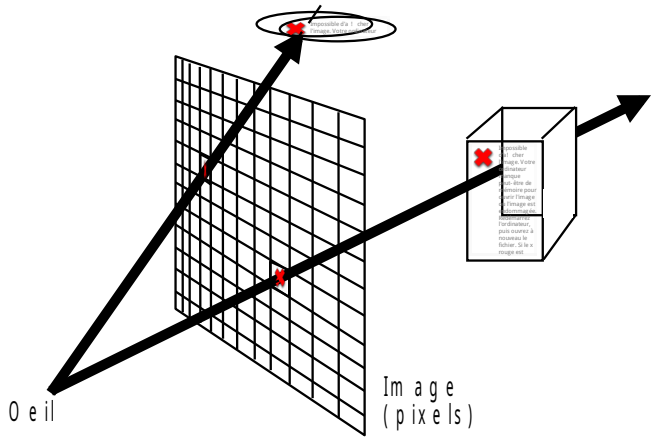




Du modèle à l'image

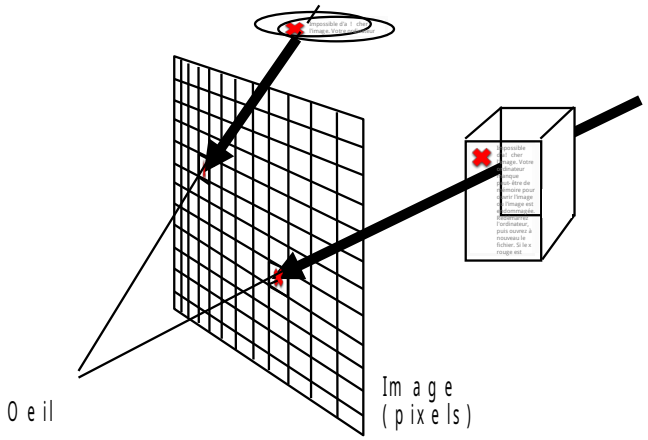


Synthèse d'images



Des rayons sont lancés depuis l'œil vers la scène en passant par un pixel

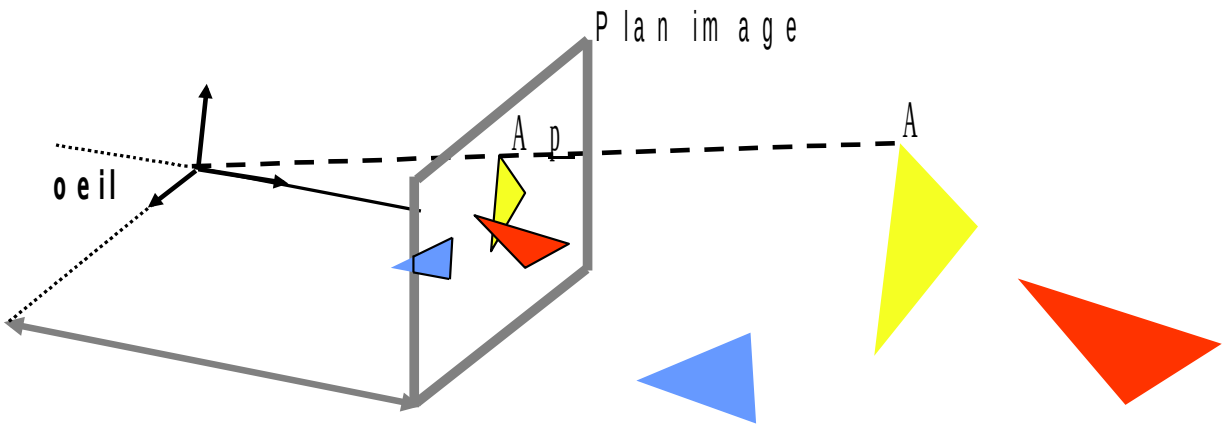
Ray-tracing
- Image réaliste
- Lent



Les objets sont projetés sur l'écran dans la direction de l'œil.

Rendu projectif (cablé sur les cartes graphiques modernes -> temps réel)

Rendu projectif

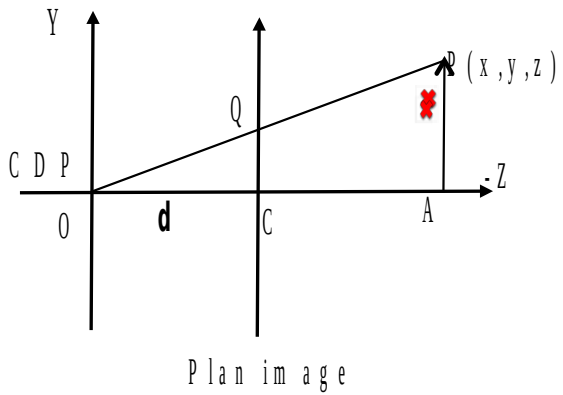


Pipeline

1. Clipping des polygones en 3D suivant la pyramide de vue
2. Projection des points sur le plan image
3. ! Remplissage des triangles (Rasterizing) dans l'image
   a. ! Suppression des parties cachées : Z-Buffer
   b. ! Calcul de la couleur : illumination

Projection perspective

- Besoin de perspective
- Configuration simple :



$$\frac{CQ}{AP} = \frac{CO}{AO} \Leftrightarrow CQ = y' = \frac{y.d}{z}$$

$$x' = \frac{x.d}{z}$$

Si $d = 1 \Rightarrow y' = \frac{y}{z}$ et $x' = \frac{x}{z}$

d = distance focale

Matrice de projection

$$M_{I \leftarrow C} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix}$$

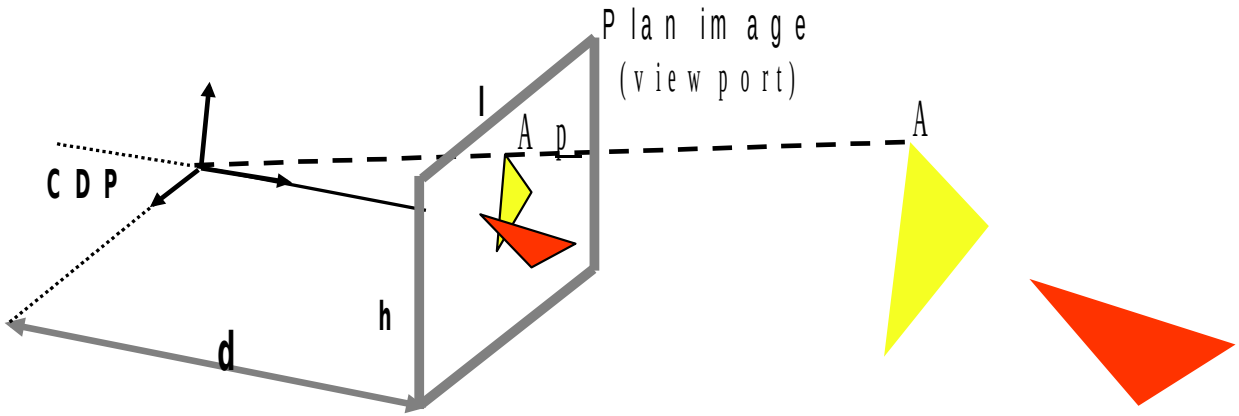
- Soit un point dans l'espace de la camera

$$(x, y, z, 1)$$

- Résultat : un point dans l'espace Image

$$(x, y, z, z/d) = \left(\frac{xd}{z}, \frac{yd}{z}, d, 1\right)$$

Rendu projectif

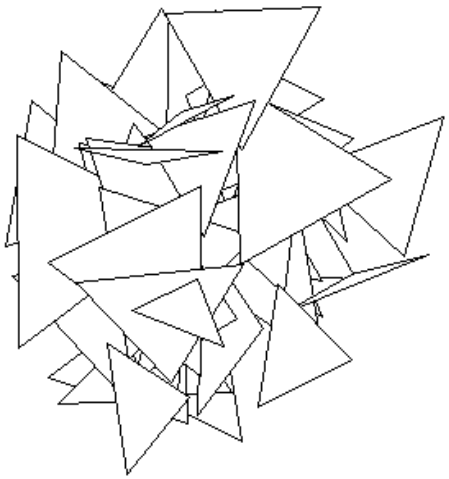


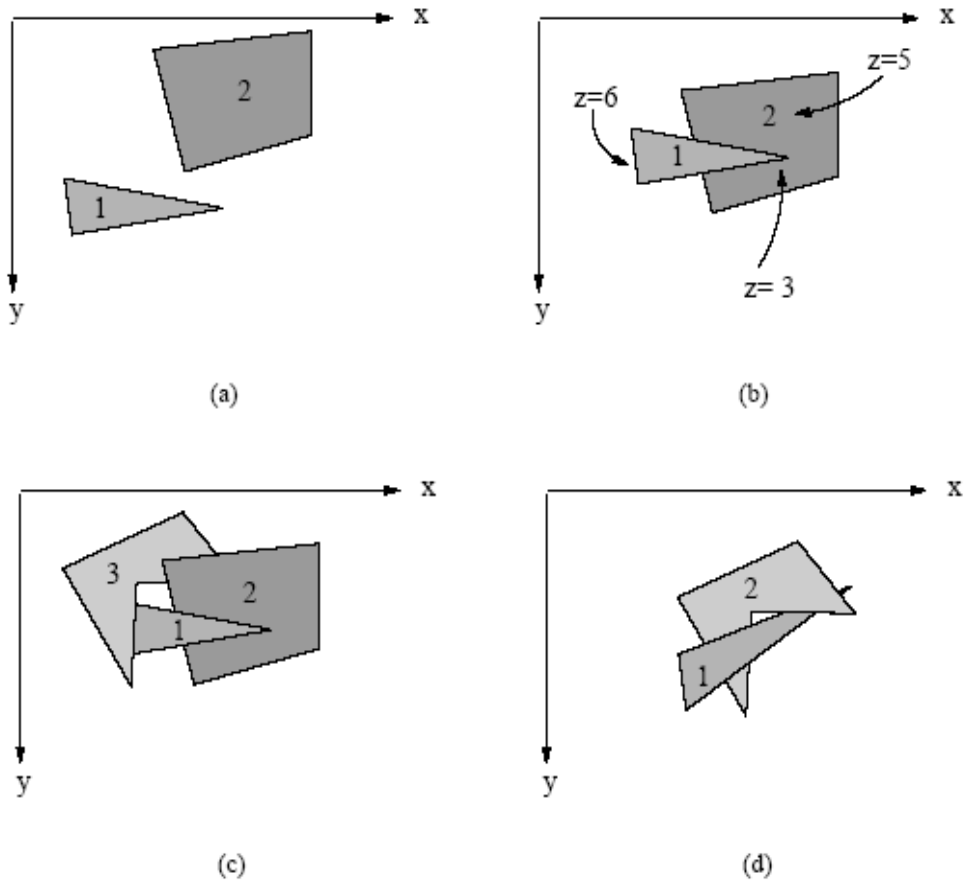
1. Projection des points sur le plan image
2. Clipping
3. ! Remplissage des triangles (rasterisation) dans l'image

Suppression des parties cachées ?

1ère idée: algo du peintre

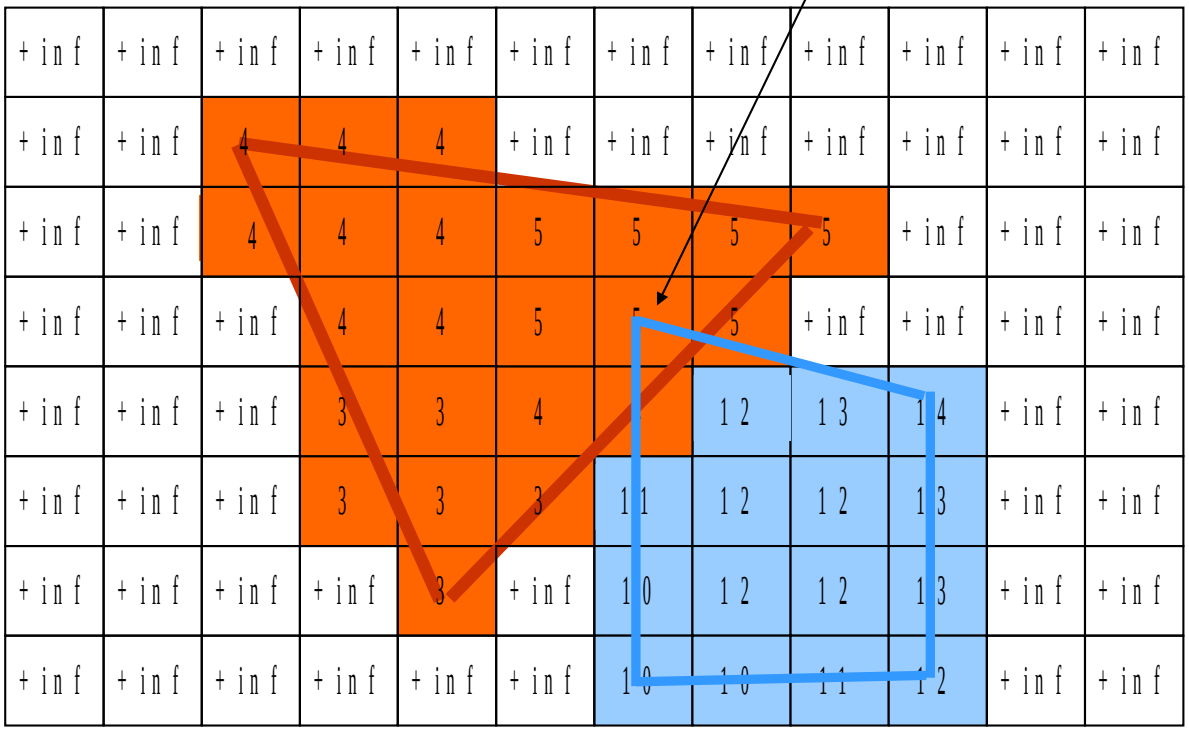
Ambiguïtés





Z-Buffer

11 < 5 donc caché



Exemple en OpenGL

- Effacer le buffer et le zbuffer entre chaque image

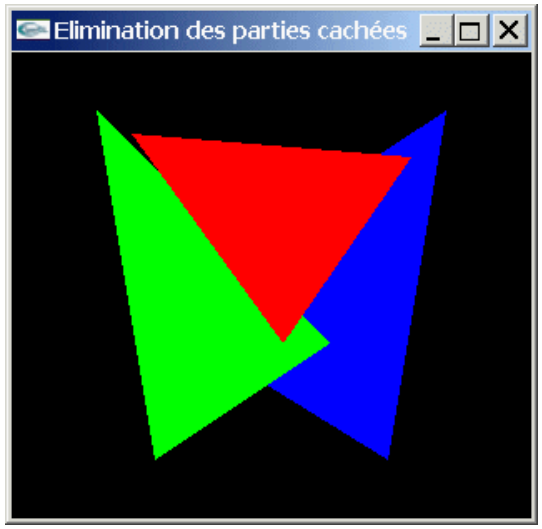
```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

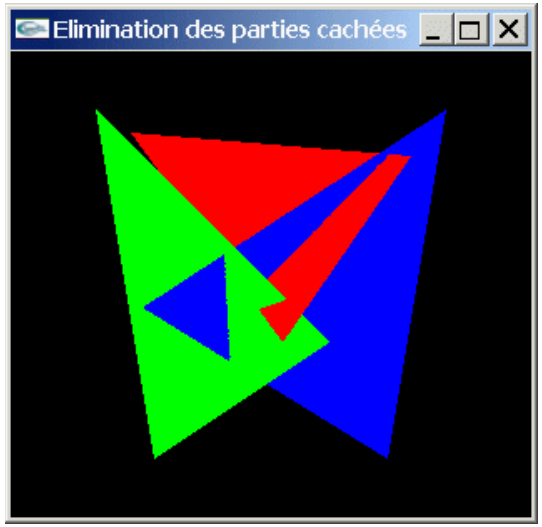
- Active le test des Z avec le Z-buffer

```
glEnable(GL_DEPTH_TEST);
```

activé

non activé





Technologie 3D en Java

- **OpenGL**
  - Open source avec beaucoup d'implémentations
  - Incroyablement bien designé et constante évolution
  - Intégralement cross-platform
- **DirectX/Direct3d**
  - Bien moins versatile (et mauvais design)
  - Microsoft™ only
    - DX 10 *requires* Vista!
- **Java3D**
  - Bizarrement très mauvais cross-platform
  - Licence GPL; community-developed

OpenGL

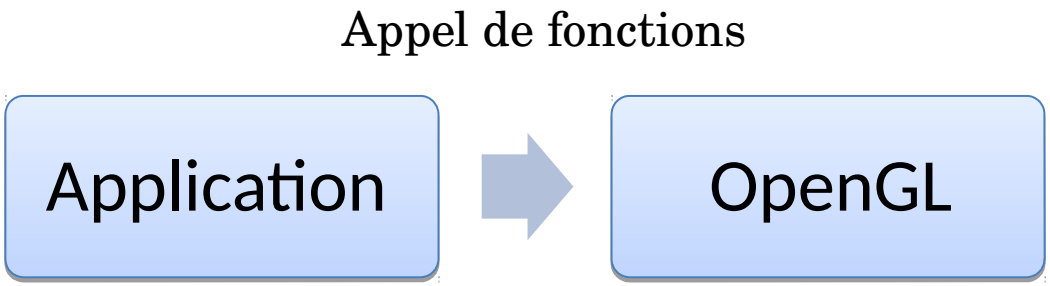
- OpenGL est …
  - hardware-independent
  - operating system independent
  - Gratuit et open-source

- OpenGL est un moteur de rendu par une *machine à états*
  - Mettre en place un état
  - Passer la donnée
  - La donnée est modifiée par l'état existant
  - Très différent du modèle POO ou la donnée "transporte" son propre état

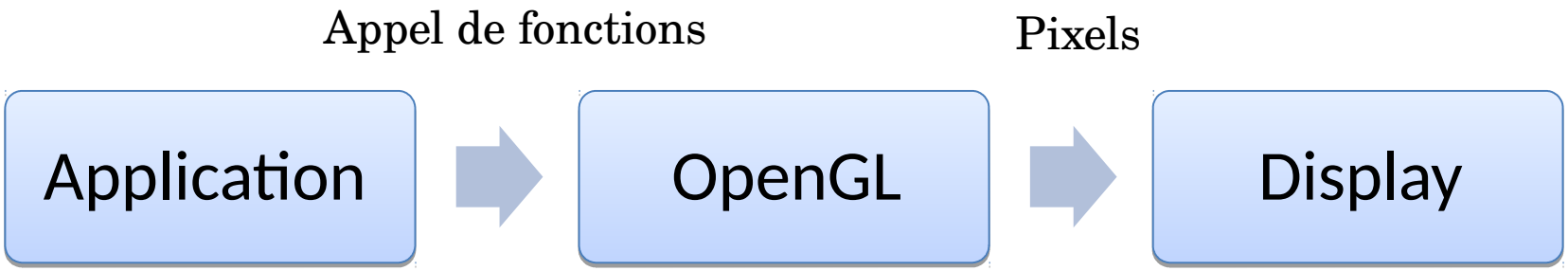
La machine à états OpenGL

Application

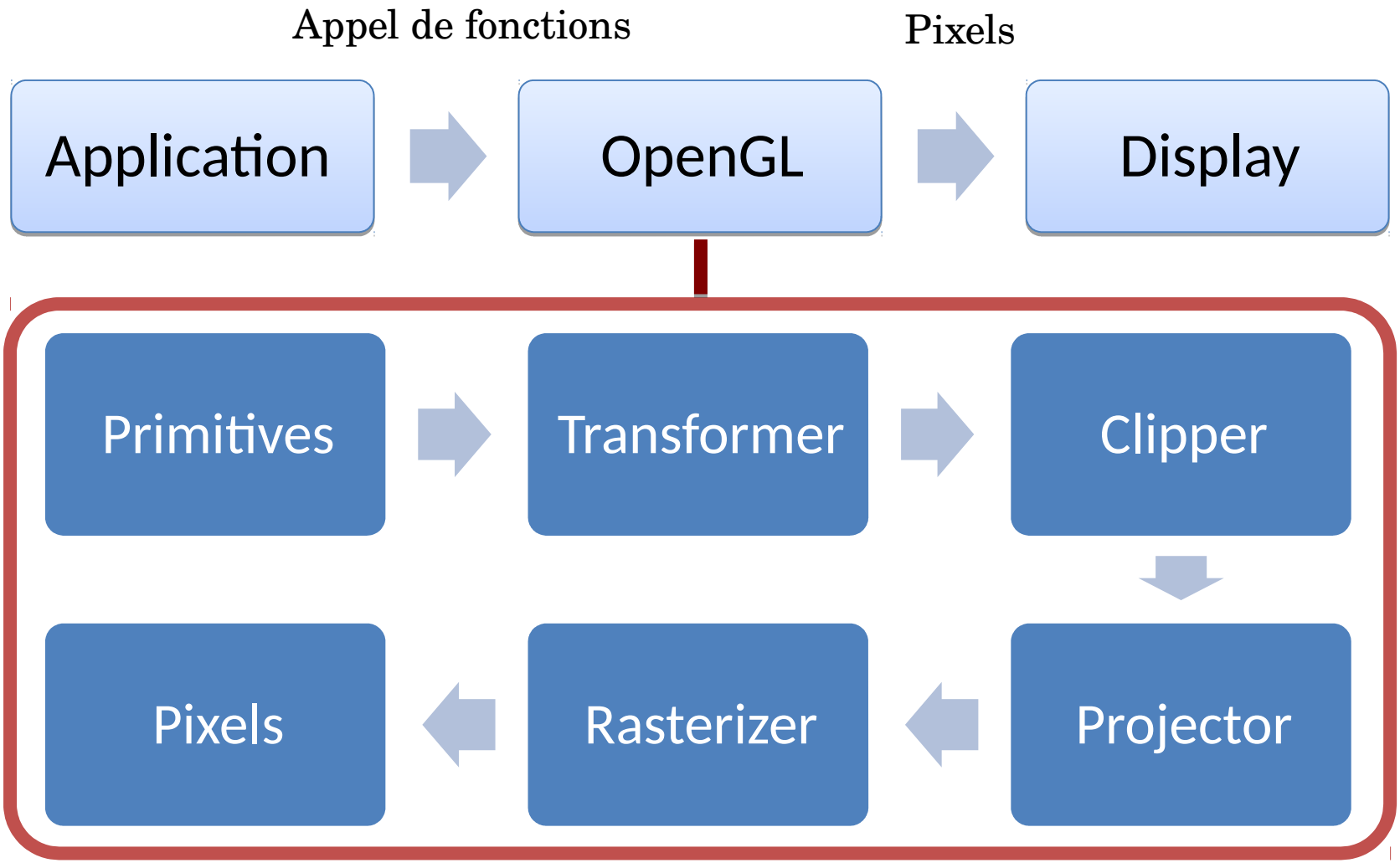
La machine à états OpenGL



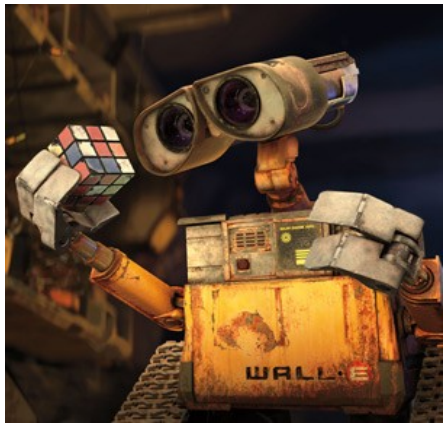
La machine à états OpenGL



La machine à états OpenGL



Niveau hardware



Two players:

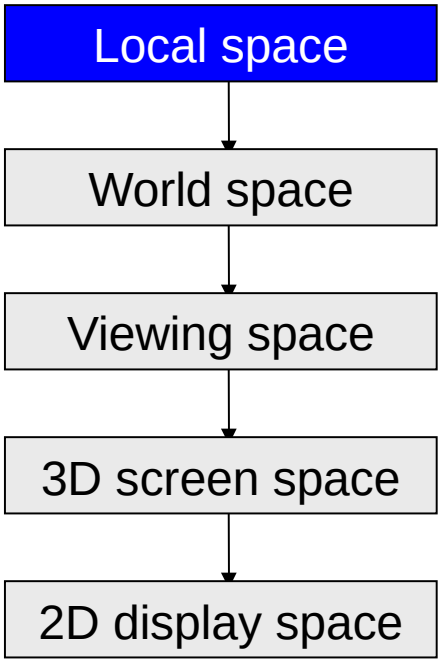
- The CPU, your processor and friend
- The *GPU* (*Graphical Processing Unit*)

or equivalent software

Le CPU envoie des flux de points et de données au GPU.

- Le GPU traite les données en fonction de *l'état* mis en place.
- Le GPU reçoit des points, couleurs, textures et autres, construits les polygones, puis affiche sur l'écran pixel par pixel
- Ce processus s'appelle le *rendering pipeline*.

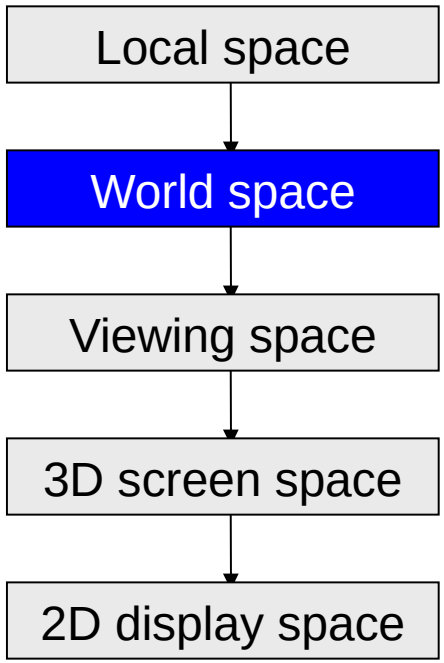
Rendering pipeline: anatomie



1. La géométrie est définie dans un **espace local** (en anglais *local space)*.

- Les points et coordonnées d'une surface sont spécifiées en relatif à l'origine locale.
- Ceci encourage la réutilisation et réapplication de la géométrie (réduit aussi la quantité de maths pour les transformations).
- Ainsi changer la position d'un objet n'implique que des matrices 4x4 au lieu de changer tous les points d'un objet !

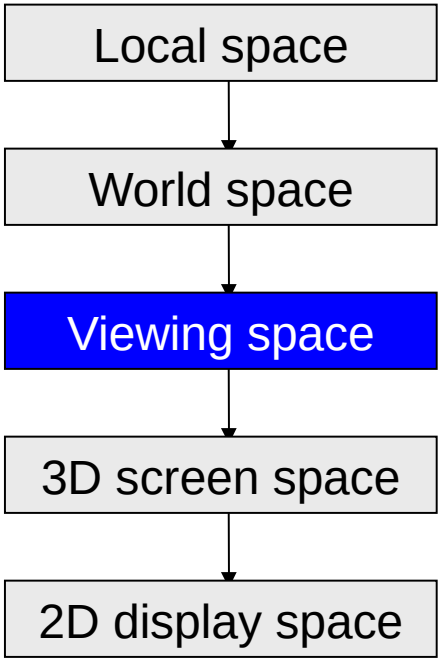
Rendering pipeline: anatomie



2. Le pipeline transforme les points et surfaces d'un espace *local* vers celui du monde (*world*)

- Une série de matrices sont concaténées pour former une unique transformation à appliquer à chaque point
- Le moteur de rendu est responsable d'associer l'état qui transforme chaque groupe de points vers les valeurs finales.

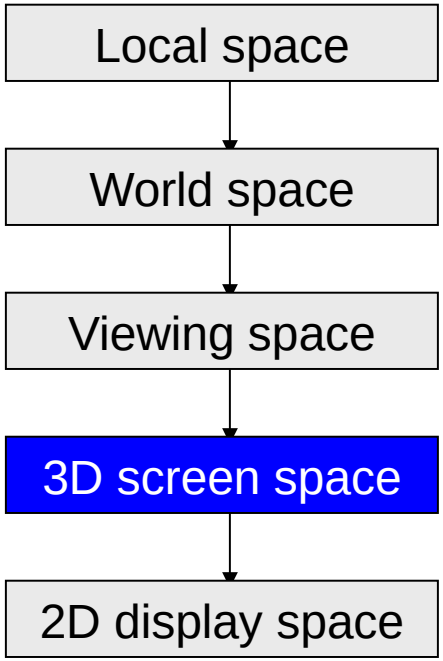
Rendering pipeline: anatomie



3. Les rotations et translations de la géométrie depuis le monde vers l'espace *de vue* ou de *camera* (viewing)

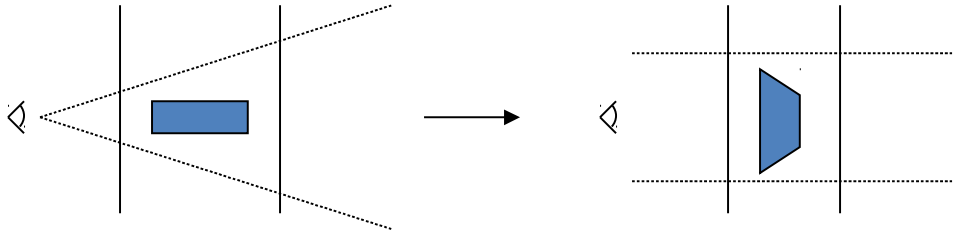
- A ce niveau, tous les points sont positionnés relativement au points de vue de la caméra
- Cela rend les opérations telles que le clipping ou la suppression de faces caches plus rapides

Rendering pipeline: anatomie

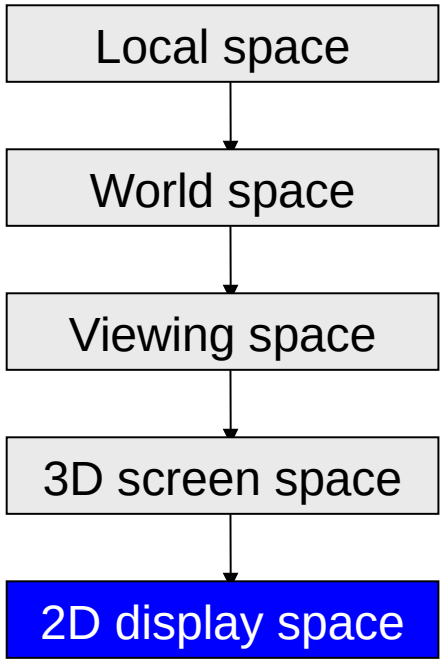


4. Perspective: Transformer l'espace de vue en une boîte alignée sur l'axe ayant un axe et des limites de visions de clipping avec z=0 et z=1

- Cette transformation n'est pas affine, les angles et échelles changent
- La suppression de partie cachées sera accélérée par ce clipping

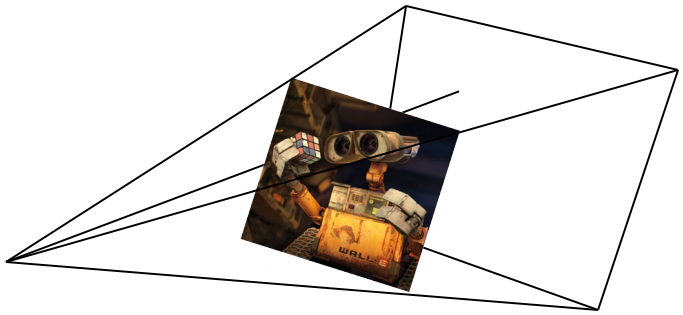


Rendering pipeline: anatomie



4. Aplatir la boîte sur un plan grâce aux informations de profondeur (Z-buffer) et suppression de parties cachées

- Mise à l'échelle du buffer final
- Eventuel post-processing niveau image

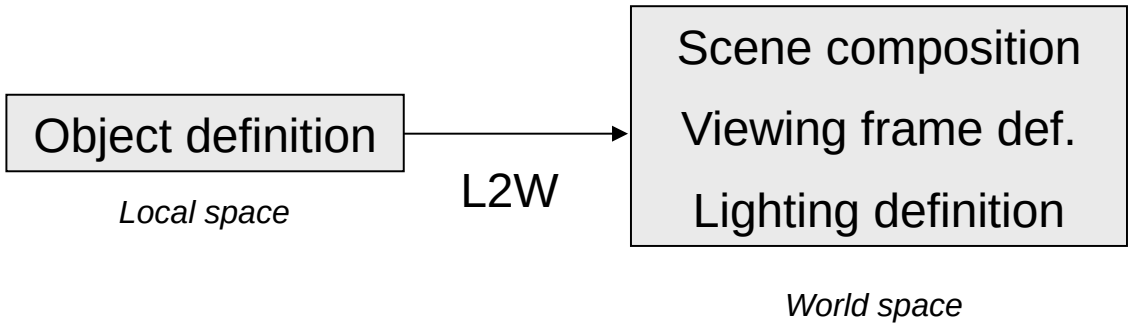


Rendering pipeline: ensemble

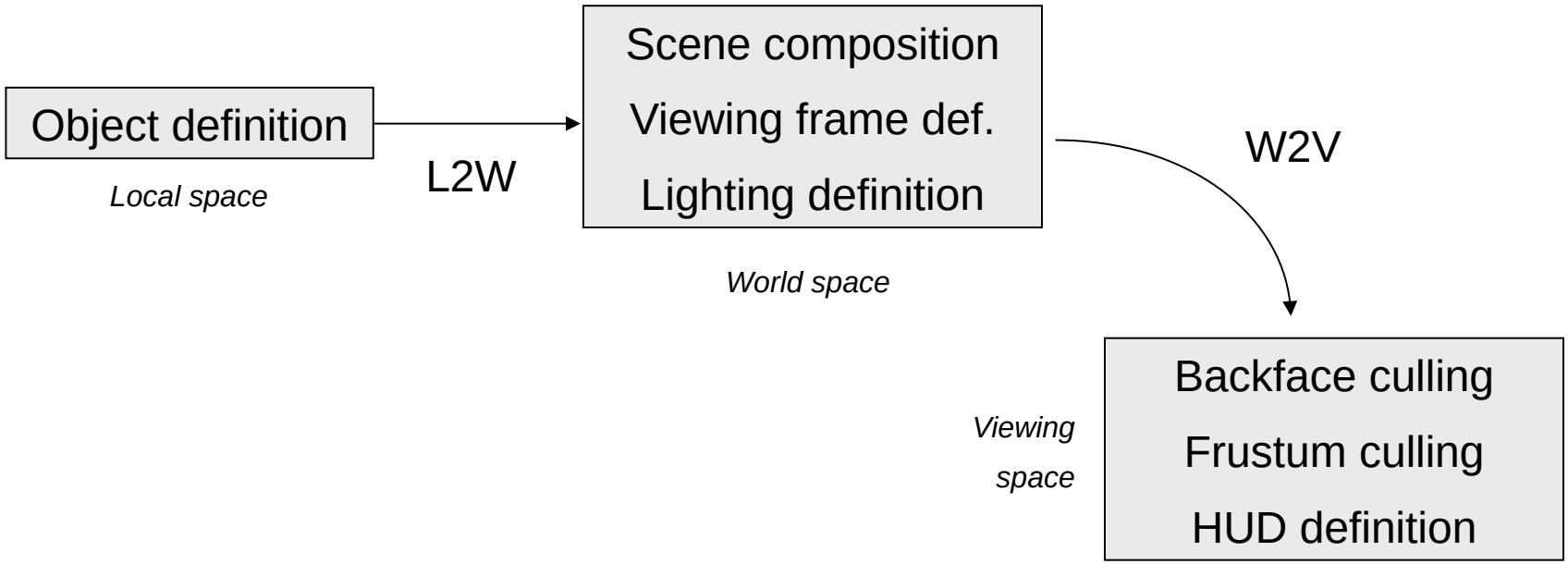
Object definition

Local space

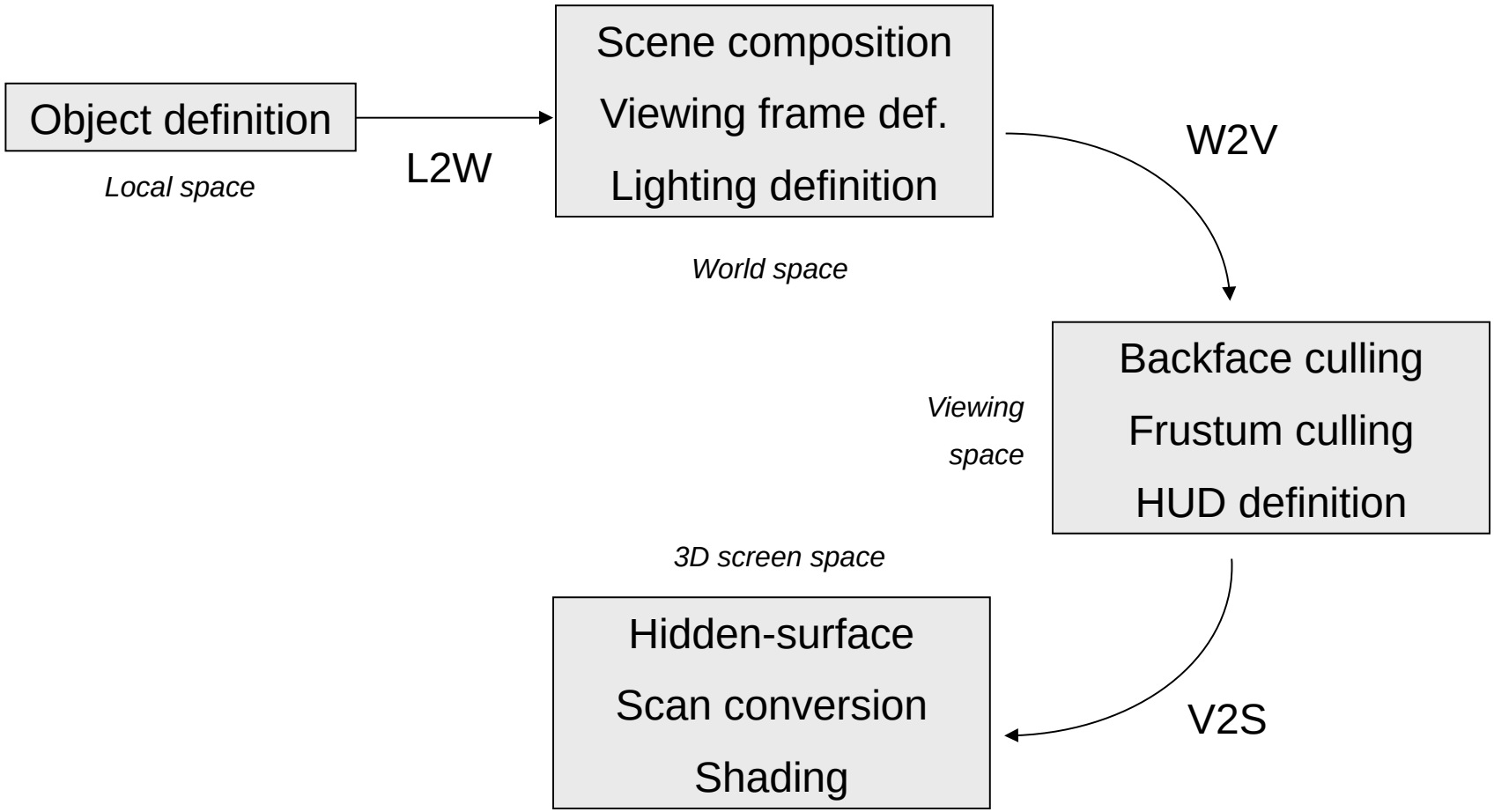
Rendering pipeline: ensemble



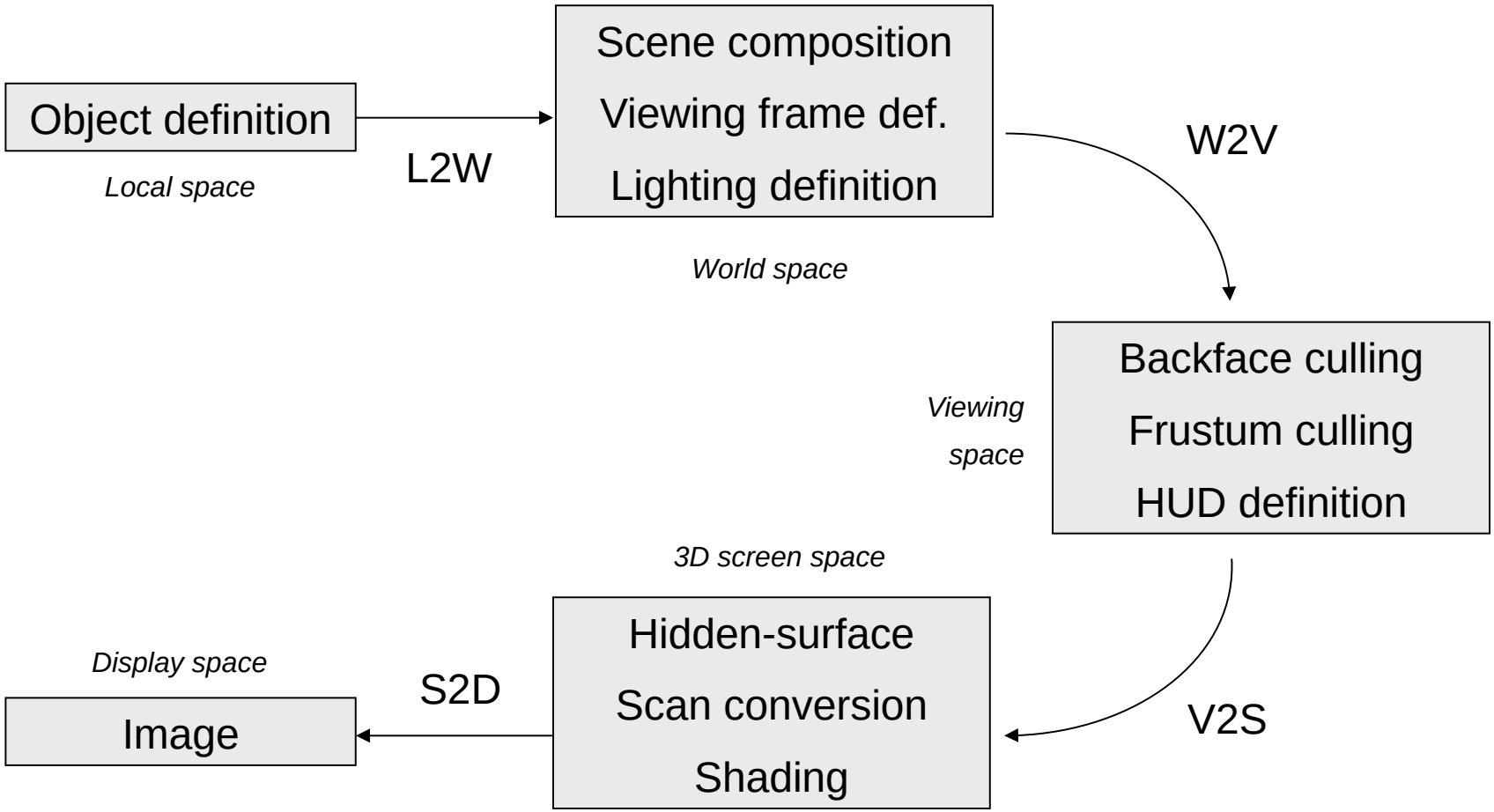
Rendering pipeline: ensemble



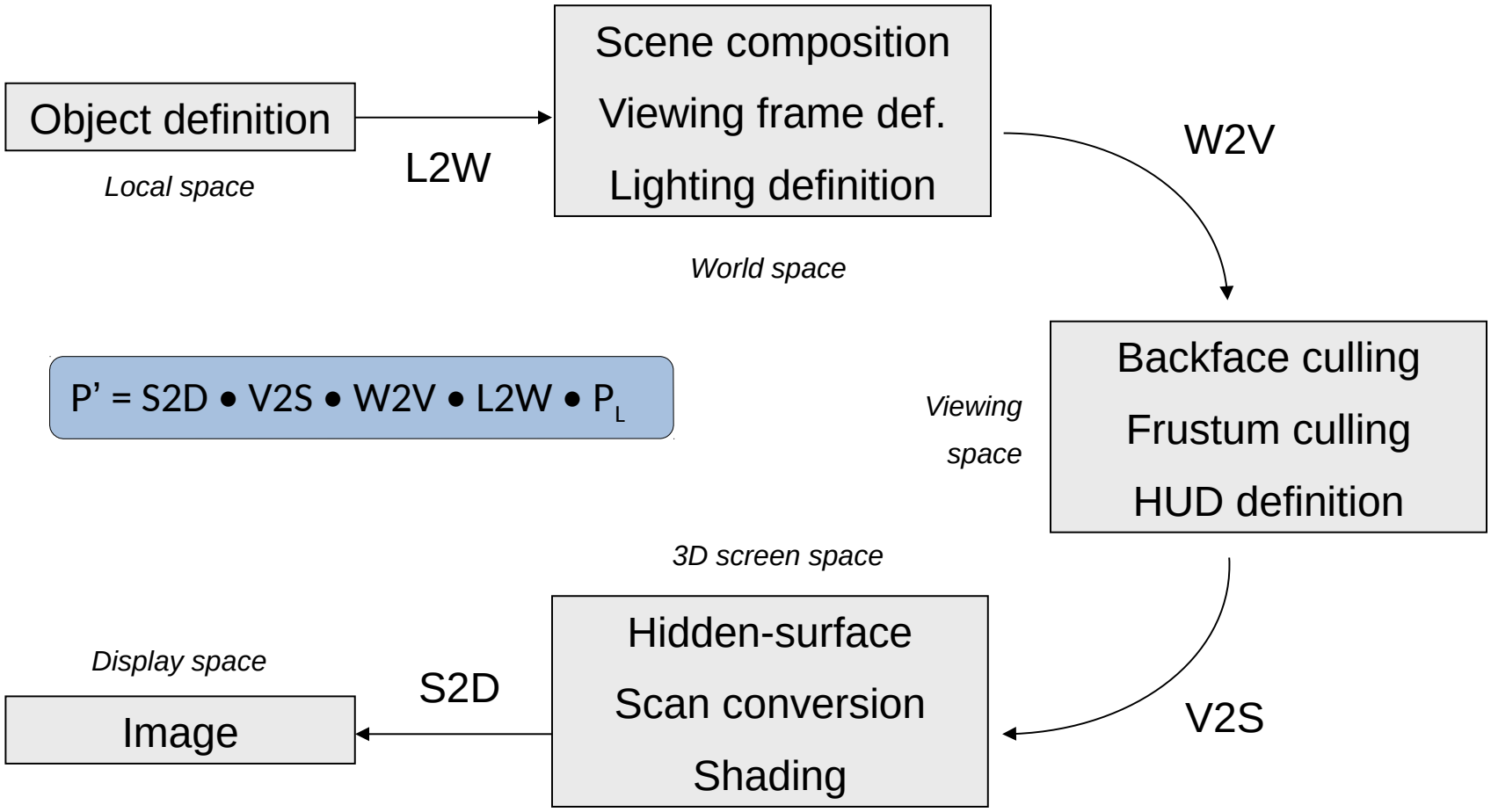
Rendering pipeline: ensemble



Rendering pipeline: ensemble



Rendering pipeline: ensemble



OpenGL

- OpenGL est platform-independent, mais ses implémentations sont dépendantes des librairies pour chaque plateforme
  - Excellent support pour Windows, Mac, linux, etc
  - Support pour mobiles avec OpenGL-ES
    - Including Google Android!
- Accélère les opérations graphiques 3D usuelles
  - Clipping (pour les primitives)
  - Suppression des parties cachées (Z-buffering)
  - Textures, alpha blending (transparence)
  - NURBS et autres primitives avancées (GLUT)

OpenGL en Java: *JOGL*

- *JOGL* est un binding Java pour OpenGL.
  - Les apps JOGL apps se déploient comme applications ou applets.
- Utiliser JOGL:
  - Télécharger les fichiers jar sur http://jogamp.org/jogl/www/
  - (Choisir la “current release build”)
    - Ajouter le contenu du .zip aux variables système CLASSPATH and PATH
    - Dans eclipse:
      - *Project->Properties->Java Build Path->Libraries*
      - Ajouter *jogl.jar* and *gluegen-rt.jar*.
  - Pour déployer un applet, il faudrait utiliser les Sun JNLP wrappers
  - Nous allons mettre en place une fenêtre basique OpenGL
  - Suivre les tutorials de *Neon Helium (Nehe)*: **http://nehe.gamedev.net/**

OpenGL

- **Primitive** functions
  - Geometric : polygons
  - Discrete : bitmap
- **Attribute** functions
  - Attributs des primitives.
  - Color, line type, light sources , textures
- **Viewing** functions
  - Determine les propriétés de la camera.
- **Input** functions
  - Permet le contrôle des fenêtres, du clavier et de la souris
- Control functions

  Permet le contrôle du programme et des features

OpenGL: mise en place

- Le constructeur gère les fenêtres
  - Canvas, JPanel
- **Main** crée simplement une instance
- **Init** mets en place des états de départ
- **Reshape** gère les transformations de vues
  - *Modelview*, *projection* matrices and *viewport*
- **Display** spécifie géometrie et transformation

OpenGL libraries

- **GLUT** (OpenGL Utility Toolkit) simplifie windowing, menus, input-handling.
- **SDL** (Simple Direct Media Layer) permet le multimedia cross-platform .
- **GLFW** windowing, OpenGL context et gestion des entrées.

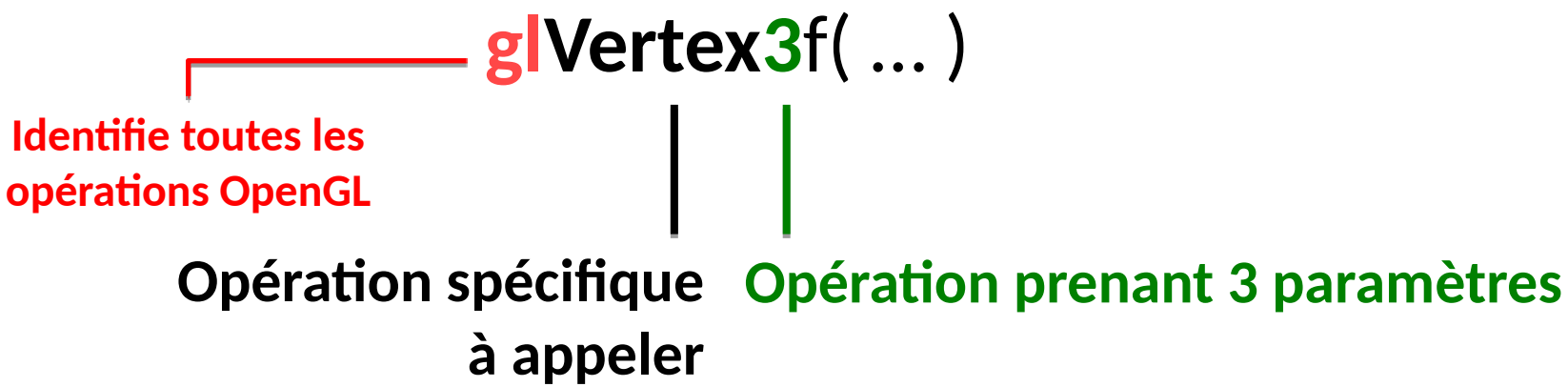
Syntaxe des fonctions OpenGL

glVertex3f(…)

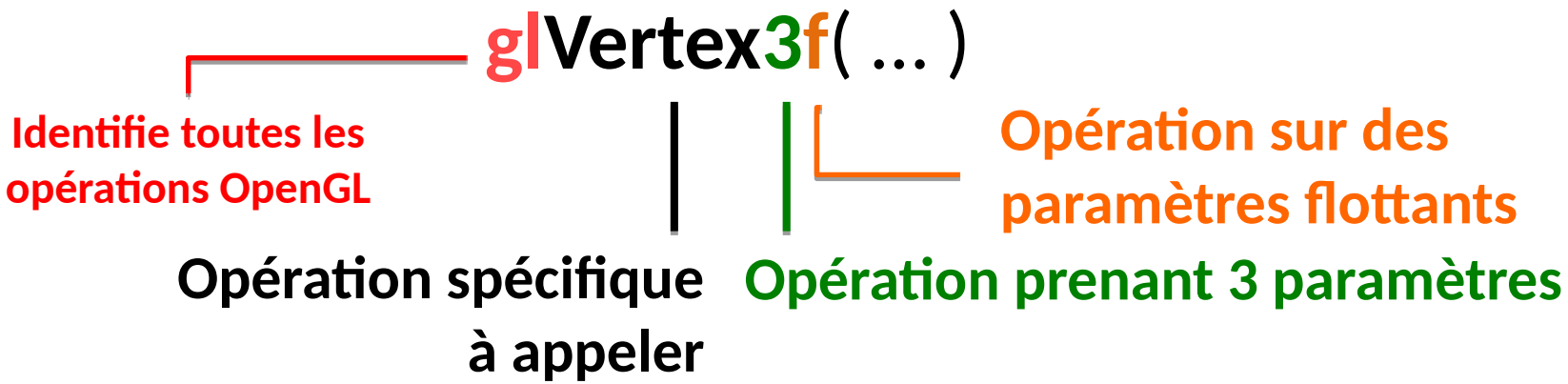
Syntaxe des fonctions OpenGL

Syntaxe des fonctions OpenGL

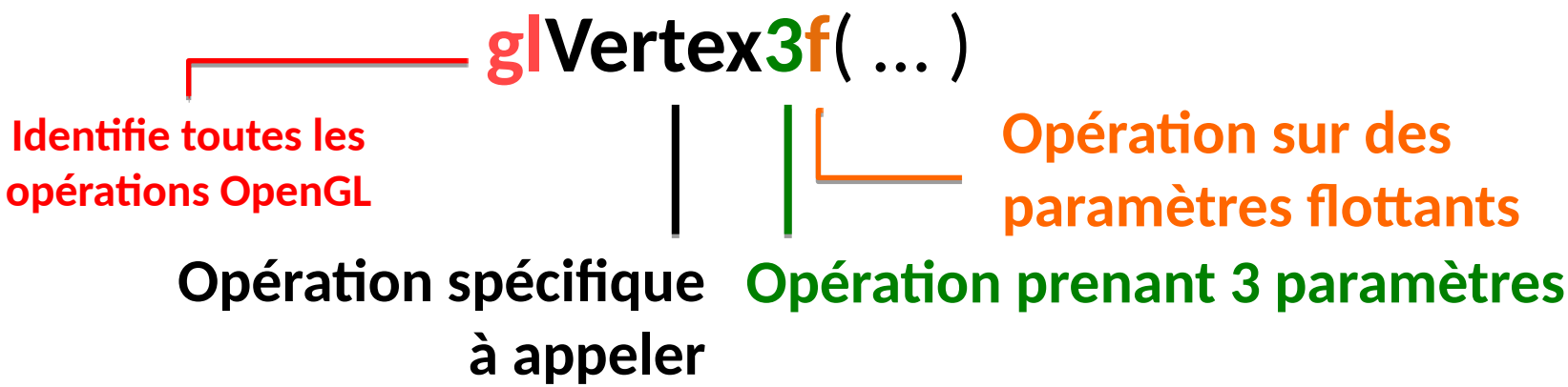
Syntaxe des fonctions OpenGL



Syntaxe des fonctions OpenGL

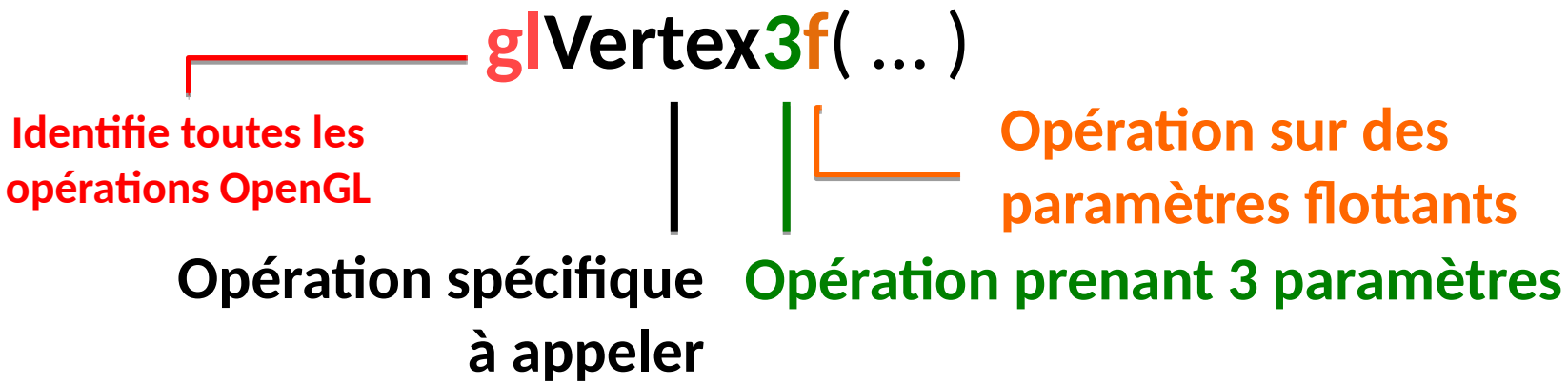


Syntaxe des fonctions OpenGL



Excepté pour la librairie GLUT (*très* pratique)

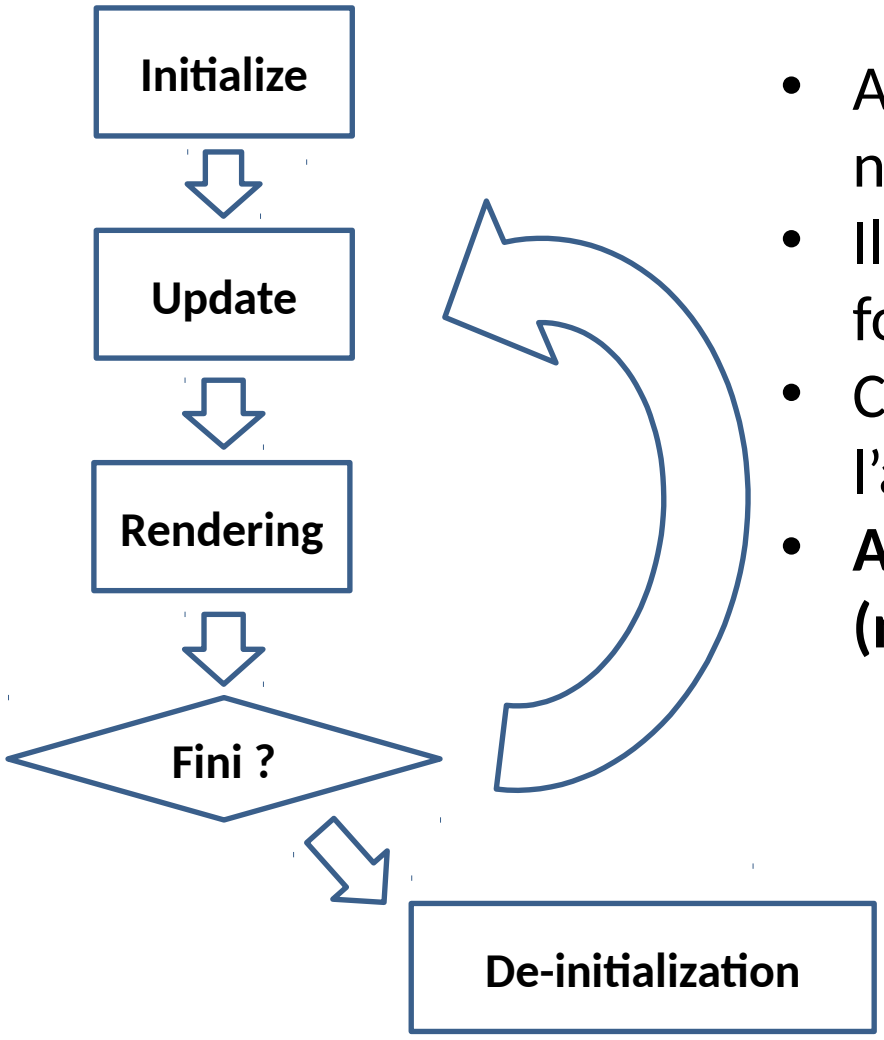
Syntaxe des fonctions OpenGL



Excepté pour la librairie GLUT (*très* pratique)

Initialize	`void glutInit(int argc, char **argv)`
Create window	`int glutCreateWindow(char * title)`
Display	`void glutDisplayFunc(void (*func)(void))`
Main Loop	`void glutMainLoop()`

OpenGL



- Attention le principe OpenGL n'est **pas bloquant**
- Il est nécessaire de gérer une forme de boucle infinie
- Celle-ci mets à jour et rends l'affichage graphique
- **Attention au temps de calcul (max ~40ms)**

Primitives

- Les primitives géométriques (basiques) permettent le dessin 3D, partant des points, lines, triangles, quads etc…
- Primitives OpenGL
  - Immediate mode (OpenGL 1.0)
  - Vertex arrays (OpenGL 1.1)
  - VBO (Vertex Buffer Objects) (OpenGL 1.5)

Mode immédiat

- Le mode de dessin immédiat est encadré par glBegin()/glEnd() :

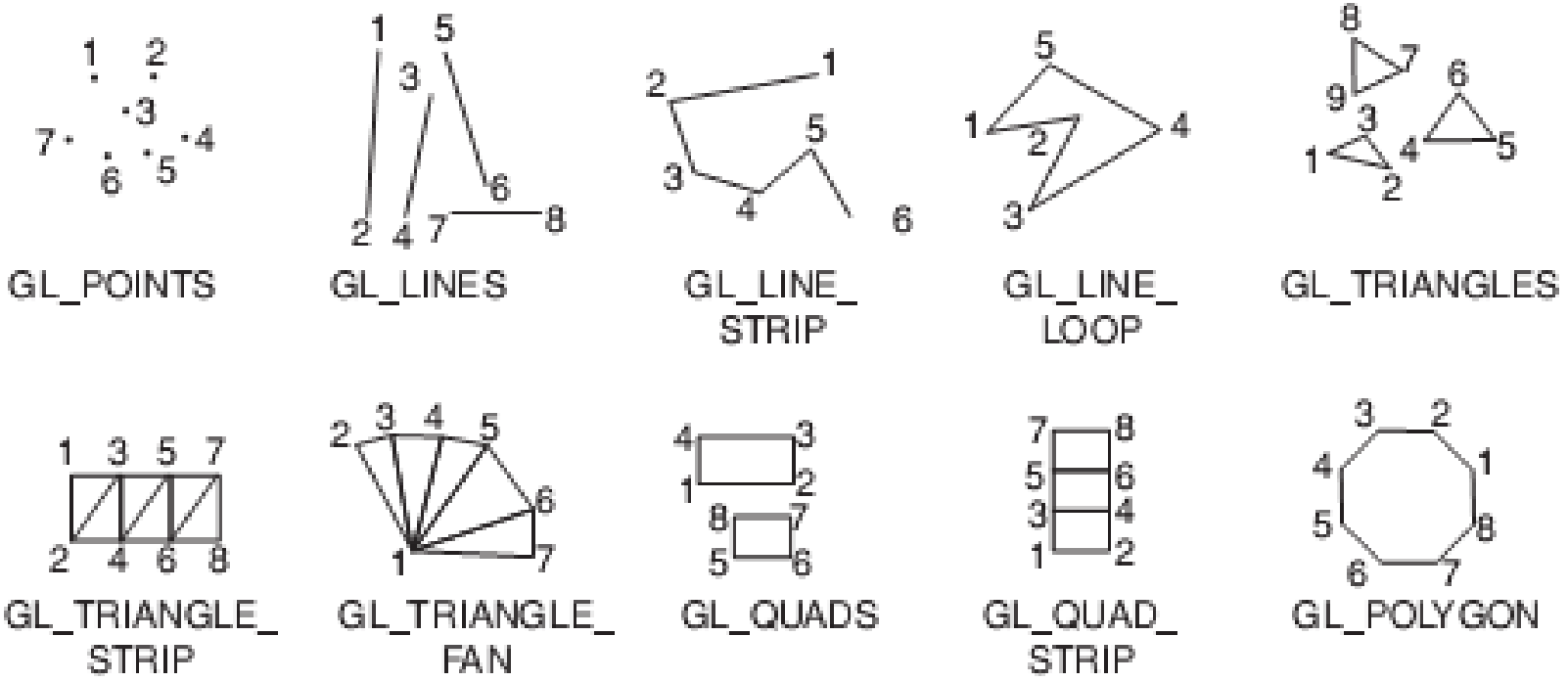
```
glBegin(Glenum mode)

// Valid glBegin()/glEnd() functions

glEnd()
```

Dessin de primitives OpenGL

- Opération **glBegin**() pour le dessin.
- Prends en paramètre un type de primitive.



Mode immédiat

- Instructions valides entre glBegin()/glEnd() :
  - **glVertex*()** Sets vertex coordinates
  - **glColor*()** Sets the current color
  - **glSecondaryColor()** Sets the secondary color
  - **glIndex*()** Sets the current color index
  - **glNormal*()** Sets the normal vector coordinates
  - **glTexCoord*()** Sets the texture coordinates
  - **glMultiTexCoord*()** Sets texture coordinates for multitexturing
  - **glFogCoord*()** Sets the fog coordinate
  - **glArrayElement()** Specifies attributes for a single vertex based on elements in a vertex array
  - **glEvalCoord*()** Generates coordinates when rendering Bezier curves and surfaces

Mode immédiat

- **glEvalPoint*()** Generates points when rendering Bezier curves and surfaces
- **glMaterial*()** Sets material properties (affect shading when OpenGL lighting is used)
- **glEdgeFlag*()** Controls the drawing of edges
- **glCallList*()** Executes a display list
- **glCallLists*()** Executes display lists

```
glVertex*() → glVertex{234}{dfis}{v}()
```

Mode immédiat

- Exemple de dessin d'un triangle

```
glBegin(GL_TRIANGLES);
   glVertex3f(-1.0f, -0.5f, 0.0f);
   glVertex3f(1.0f, -0.5f, 0.0f);
   glVertex3f(0.0f, 1.0f, 0.0f);
glEnd();
```

OpenGL

- Résumé du dessin
  - `glBegin(Glenum mode)`
  - `glEnd()`
  - `void glVertex3f()`
  - `void glClear(Glbitfield mask)`
  - `void glFlush()`
- Changer les défauts GLUT defaults
  - `void glutInitDisplayMode`
  - `void glutInitWindowSize`
  - `void glutInitWindowPosition`
  - `Void gluPerspective`

OpenGL: **mise en place**

1. Télécharger la librairie JOGL
   1. Télécharger les fichiers jar sur http://jogamp.org/jogl/www/
   2. Ajouter le contenu du .zip aux variables système CLASSPATH and PATH
   3. Dans eclipse: *Project->Properties->Java Build Path->Libraries*
      1. Ajouter *jogl.jar* and *gluegen-rt.jar*.
2. Créer un projet éclipse (en ajoutant JOGL)
3. Créer une fenêtre Swing
4. Ajouter un objet de type `GLCanvas`
5. Créer une classe implémentant `GLEventListener`
6. Dans cette classe ajouter les fonctions
   1. `public void init(GLAutoDrawable glD)`
   2. `public void display(GLAutoDrawable glD)`
   3. `public void reshape(GLAutoDrawable, int x, int y, int width, int height)`
7. Dessiner un carré.

Solution

```
public class HelloSquare {
  public static void main(String[] args) {
    new Thread() {
      public void run() {
        Frame frame = new Frame("Hello Square");
        GLCanvas canvas = new GLCanvas();

        // Setup GL canvas
        frame.add(canvas);
        canvas.addGLEventListener(new Renderer());

        // Setup AWT frame
        frame.setSize(400, 400);
        frame.addWindowListener(new
    WindowAdapter(){
          public void windowClosing(WindowEvent e)
    {
            System.exit(0);
          }
        });
        frame.setVisible(true);

        // Render loop
        while(true) {
          canvas.display();
        }
      }
    }.start();
  }
}
```

```
public class Renderer implements GLEventListener {
  public void init(GLAutoDrawable glDrawable) {
    final GL gl = glDrawable.getGL();
    gl.glClearColor(0.2f, 0.4f, 0.6f, 0.0f);
  }

  public void display(GLAutoDrawable glDrawable) {
    final GL gl = glDrawable.getGL();
    gl.glClear(GL.GL_COLOR_BUFFER_BIT);
    gl.glLoadIdentity();

     XX
     ??
     DESSIN CARRE
     ??
     XX
  }

  public void reshape(GLAutoDrawable gLDrawable,
      int x, int y, int width, int height) {
    final GL gl = gLDrawable.getGL();
    final float h = (float)width / (float)height;

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    (new GLU()).gluPerspective(50, h, 1, 1000);
    gl.glMatrixMode(GL.GL_MODELVIEW);
  }
}
```

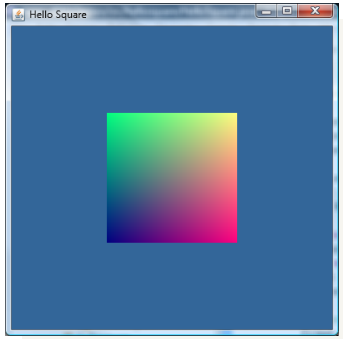
OpenGL

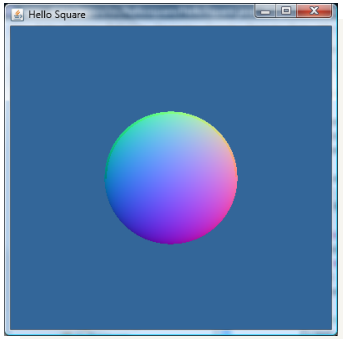
- Setting Colors
  - `void glColor*()`
  - `void glClearColor( Glclampf r, GLClampf g, Glclampf b, Glclampf a )`
- Two Dimensional Viewing
  - `void gluOrtho2D( Gldouble left, Gldouble right, Gldouble bottom, Gldouble top)`
- Coordinate Systems and Transformation
  - `glMatrixMode( GL_PROJECTION );`
  - `glLoadIdentity();`
- Points
  - `glPointSize(2.0);`
- Lines
  - `GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP`
    `in glBegin( GL_LINES );`
- Enabling OpenGL features
  - `void glEnable( Glenum feature)`
  - `void glDisable( Glenum feature )`

OpenGL

```
public void vertex(GL gl,
    float x, float y, float z) {
  gl.glColor3f(
    (x+1)/2.0f,
    (y+1)/2.0f,
    (z+1)/2.0f);
  gl.glVertex3f(x, y, z);
}
```

1) Shaded square



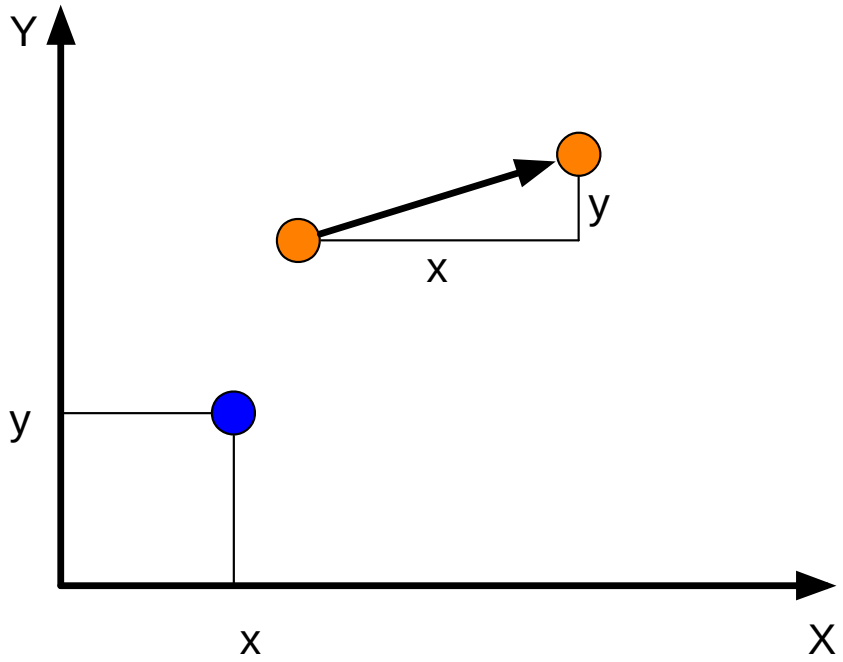



```
public void sphere(GL gl,
   double u, double v) {
  vertex(gl, cos(u)*cos(v),
             sin(u)*cos(v),
             sin(v));
  }

for (double u = 0; u <= 2*PI;
    u += 0.1) {
  for (double v = 0; v <= PI;
    v += 0.1) {
    sphere(gl, u, v);
    sphere(gl, u+0.1, v);
    sphere(gl, u+0.1, v+0.1);
    sphere(gl, u, v+0.1);
  }
}
```

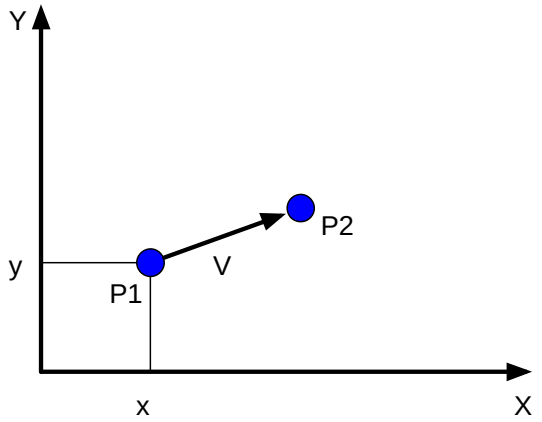
2) Parametric sphere

Repérage dans l'espace 2D



- A l'aide de la classe Point
  - Attributs double x et y
- La même classe nous permet de gérer les points et les vecteurs

Repérage dans l'espace 2D

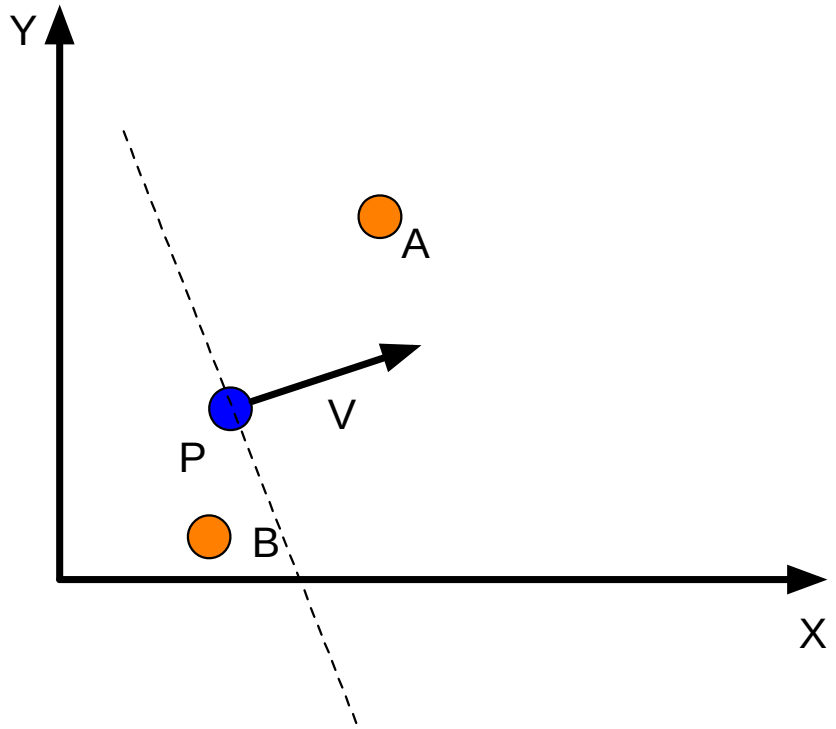


- Déplacements discrets (P : position, V : vitesse) :

$$\overrightarrow{P_2} = \overrightarrow{P_1} + \overrightarrow{V}, \qquad \begin{aligned} P_2.x &= P_1.x + V.x \\ P_2.y &= P_1.y + V.y \end{aligned}$$

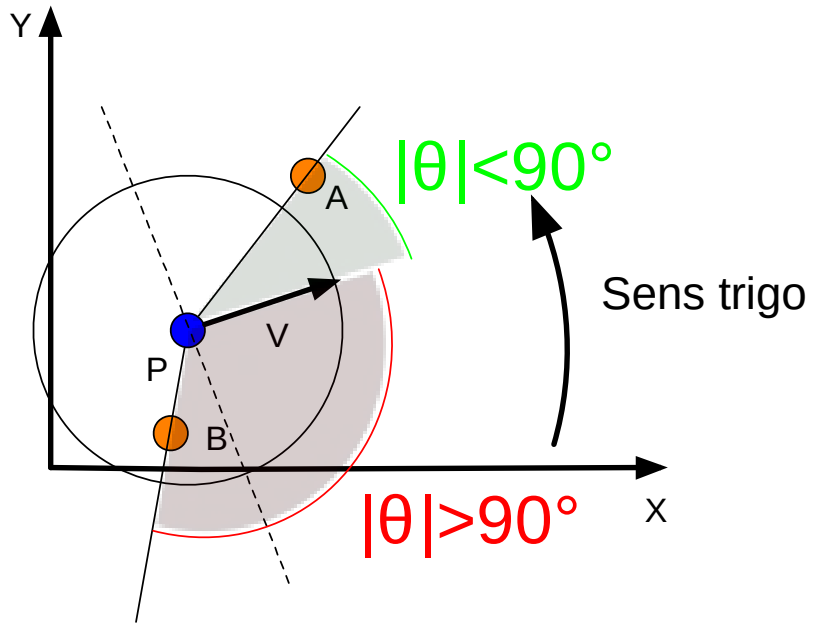
- En physique : $v = \dot{x} \approx \frac{x_{t+1} - x_t}{\delta_t}$, pour nous : $\delta_t = 1$ (unité arbitraire)
- En utilisant des vecteurs suffisamment petits : modélisation d'un déplacement continu

Repérage dans l'espace 2D



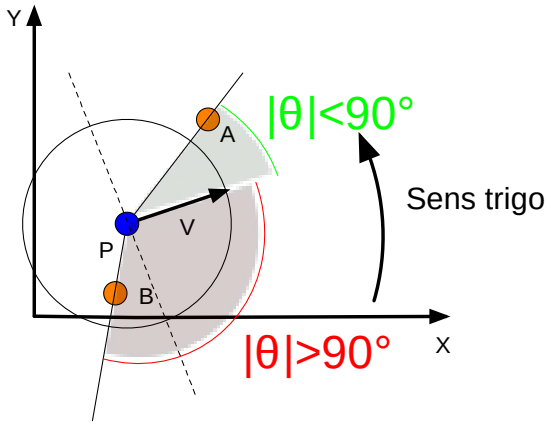
- Un objet est caractérisé par sa position *P* et sa vitesse *V*
- Qu'est ce qui est devant, qu'est ce qui est derrière l'objet ?
- Qu'est ce qui est à droite, qu'est ce qui est à gauche ?

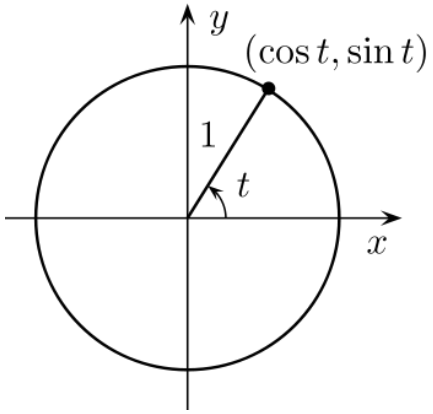
Repérage dans l'espace 2D



- Devant/ Derrière : calculer les angles $\widehat{V, PA}$ et $\widehat{V, PB}$
- Gauche/ Droite : calculer les angles $\widehat{V, PA}$ et $\widehat{V, PB}$ avec le signe

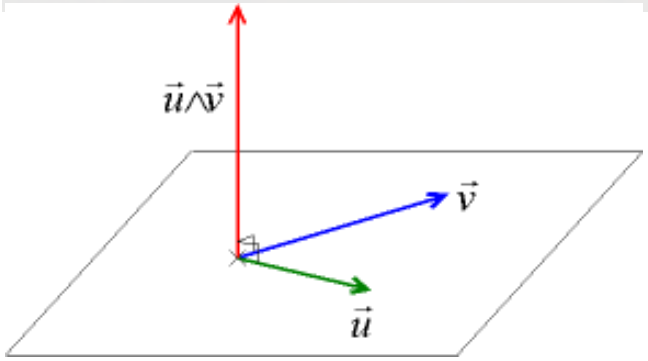
Repérage dans l'espace 2D

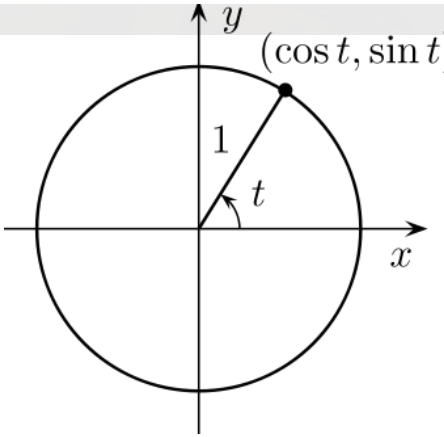




- Produit scalaire
  $U \cdot V = \langle U, V \rangle = \|U\| \|V\| \cos(\widehat{U, V}) = U_x V_x + U_y V_y$
- Corollaire :
  $$\widehat{U, V} = \arccos \frac{U \cdot V}{\|U\| \|V\|}$$
- Attention : $\widehat{U, V} \in [0, \pi]$, pas de signe ici…
- Le signe de $U \cdot V$ permet de résoudre le pb devant/derrière

Repérage dans l'espace 3D





- Produit vectoriel : $\|U \wedge V\| = \|U\|\|V\| \sin(\widehat{U,V})$
- Corollaire :

$$\widehat{U,V} = \arcsin\left(\frac{\|U \wedge V\|}{\|U\|\|V\|}\right) \qquad U \wedge V = \begin{pmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{pmatrix}$$

- L'étude de $u_1 v_2 - u_2 v_1$ permet de connaitre le signe de l'angle…

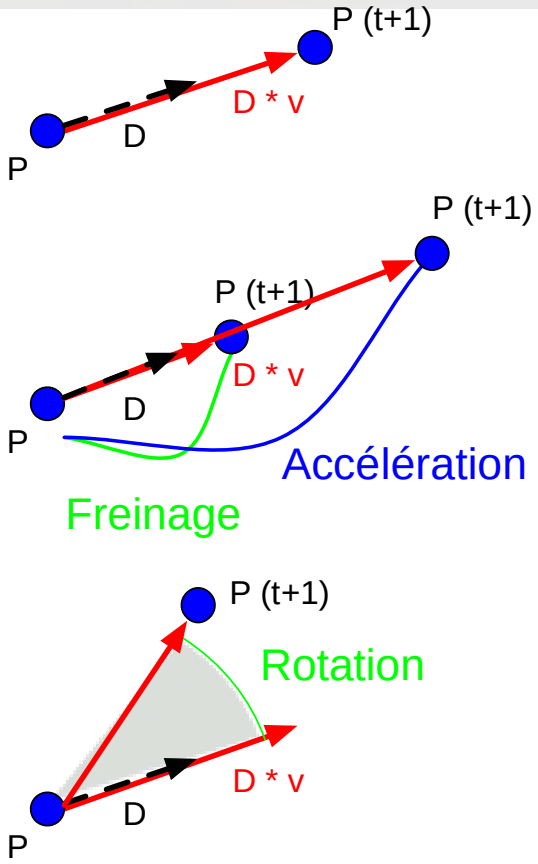
Exemple: **Course de voitures**

La voiture sera définie géométriquement par :

- Sa position : *P*
- Sa direction (vecteur unitaire) : *D*
  Conservation de la direction même à l'arrêt
- Sa vitesse (scalaire) : *v 2* [0, *vmax*]

La commande de la voiture se fera sur 2 axes :

- Accélération/ Freinage : modification de *v*
- Commande de direction : modification de *D*



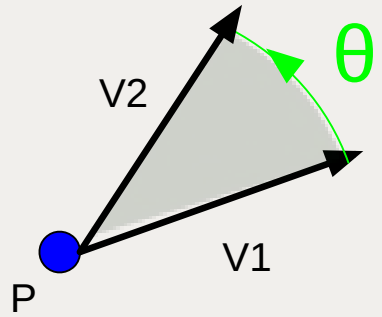
Repérage dans l'espace 3D

Soit un vecteur $V = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$, la rotation d'angle ✓ est obtenu en utilisant la matrice de rotation :

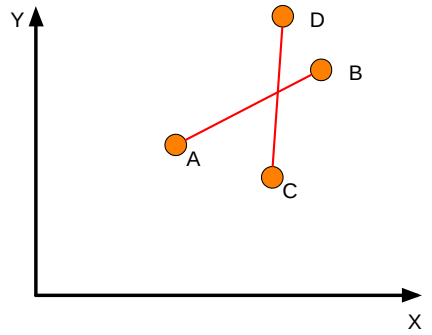
$$R = \begin{bmatrix} \cos(✓) & -\sin(✓) \\ \sin(✓) & \cos(✓) \end{bmatrix}, \quad V^0 = RV$$

C'est à dire en utilisant la mise à jour :

- $v_x^0 = v_x \cos(✓) - v_y \sin(✓)$
- $v_y^0 = v_x \sin(✓) + v_y \cos(✓)$



Exemple: **Collisions**



(Problématique de base dans les cartes graphiques/ moteur physique)

Comment détecter la collision de deux vecteurs ?

- Si C et D sont à gauche et à droite de AB
- ET que A et B sont à gauche et à droite de CD

Résultat :

S'ils sont de part et d'autre, l'un des produit vectoriel est positif, l'autre négatif…

$$(AB \wedge AC)(AB \wedge AD) < 0 \text{ ET } (CD \wedge CA)(CD \wedge CB) < 0$$

Classe vecteur / matrice

- Addition, soustraction
  - génération d'un nouveau vecteur
  - auto-opérateur
- produit scalaire
- produit vectoriel (composante en z)
- multiplication par un scalaire
- rotation
- calcul de la norme
- clonage
- test d'égalité (structurelle)

OpenGL

- Même le dessin en 3D perspective de cubes devient vite inutile
- Comment le faire bouger ?
- La matrice de Model View contient les transformations appliquées aux objets
  - Changer cette matrice l'applique à tous les objets dessinés par la suite (persistance).
  - Il suffit donc d'appeler la fonction d'affichage time tweak model view matrix

OpenGL

- Dans la fonction de dessin appeller
  - **gl.glRotatef**(0.5f, 1.0f, 1.6f, 0.7f);
  - “Tourne de 0.5 degrés autour du vecteur {1,1.6,.7}”
  - Plus précisément, “calcule la matrice pour cette rotation et multiplie la Model View par celle-ci
- D’autre fonctions de transformations
  - gl.glTranslatef(x,y,z)
  - gl.glScalef(x,y,z)

OpenGL

- Exercice de hiérarchie
  - Dessiner un premier grand cube
  - Dessiner un second plus petit cube
    - Posé sur le premier
    - Tourné de 30°
  - Dessiner un troisième plus petit cube
    - Posé sur le second
    - Tourné de 60°
  - Animer les 3 cubes (rotations à vitesse variable)

OpenGL

```
gl.glPushMatrix();
    gl.glScalef(1,.3f,1);
    drawCube(gl); //just draws standard cube
    gl.glPushMatrix();
        gl.glTranslatef(.5f,2f,0);
        gl.glScalef(0.3f,1f,0.3f);
        gl.glRotatef(15,0,1,0);
        drawCube(gl);
    gl.glPopMatrix();
    gl.glPushMatrix();
        gl.glTranslatef(-.5f,2f,0);
        gl.glScalef(0.2f,1f,0.2f);
        drawCube(gl);
    gl.glPopMatrix();
gl.glPopMatrix();
```

Transformations

- Translation

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotation by t around Y

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

- Scaling

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Perspective

$$\begin{pmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & f/(f-d) & -df/(f-d) \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(Watt, pp.149—153)

Pourquoi des matrices 4x4

- Tout est fait en jeu de *coordonées homogènes*.
  - $[X, Y, Z, W]_H \rightarrow [X/W, Y/W, Z/W]$
  - $[A, B, C] \rightarrow [A, B, C, 1]_H$
- Pourquoi?
  - Translation

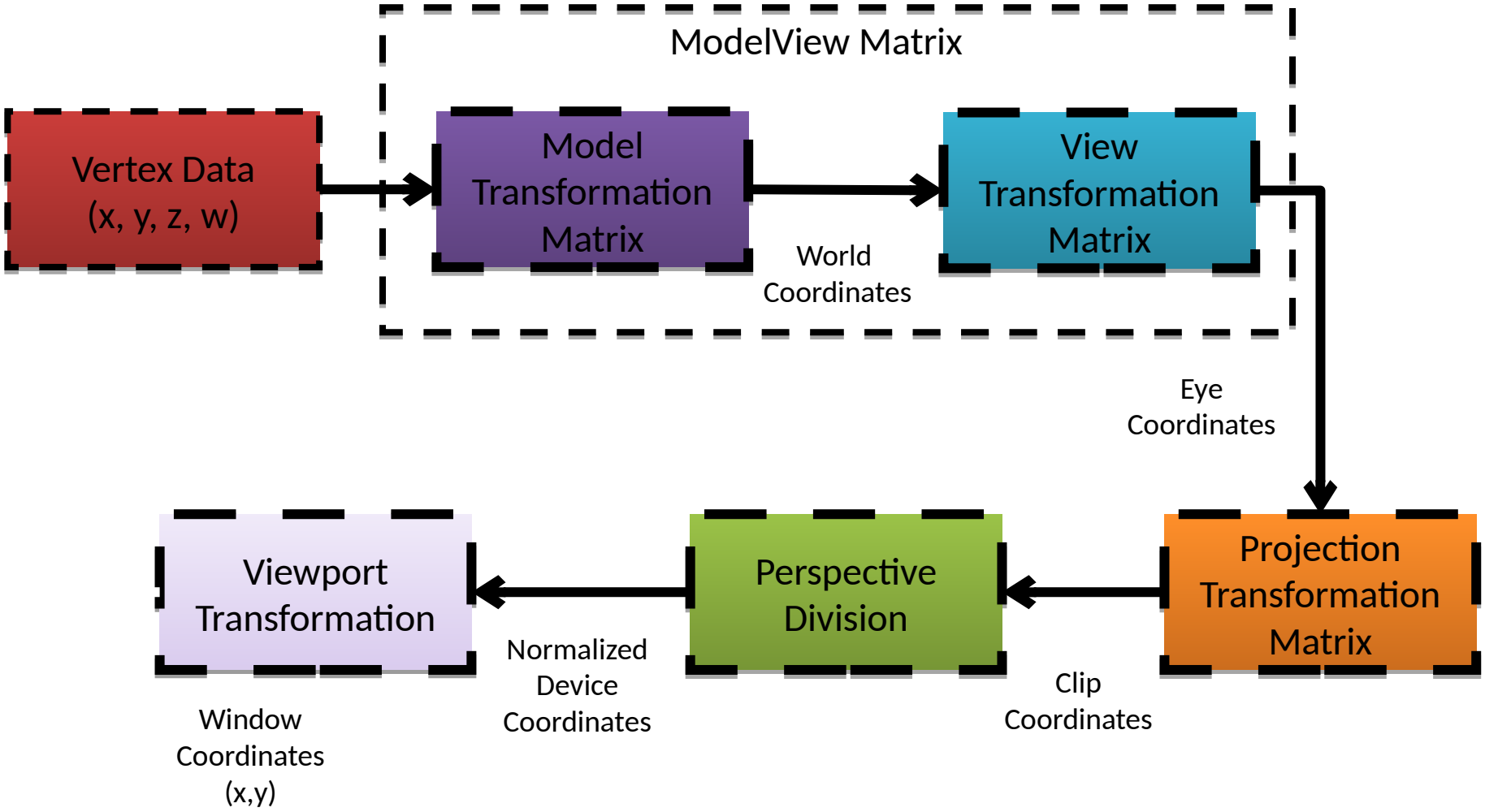
$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ 1 \end{pmatrix} = \begin{pmatrix} a+x \\ b+y \\ c+z \\ 1 \end{pmatrix}$$

  - Perspective – yields X/Z, Y/Z. (Try it!)

OpenGL

- Un programme peut agir par transformation sur différentes parties du *vertex transformation pipeline* :
  1. Viewport Transformation
  2. Viewing Transformation
  3. Projection Transformation
  4. Modelling Transformation

Vertex Transformation Pipeline



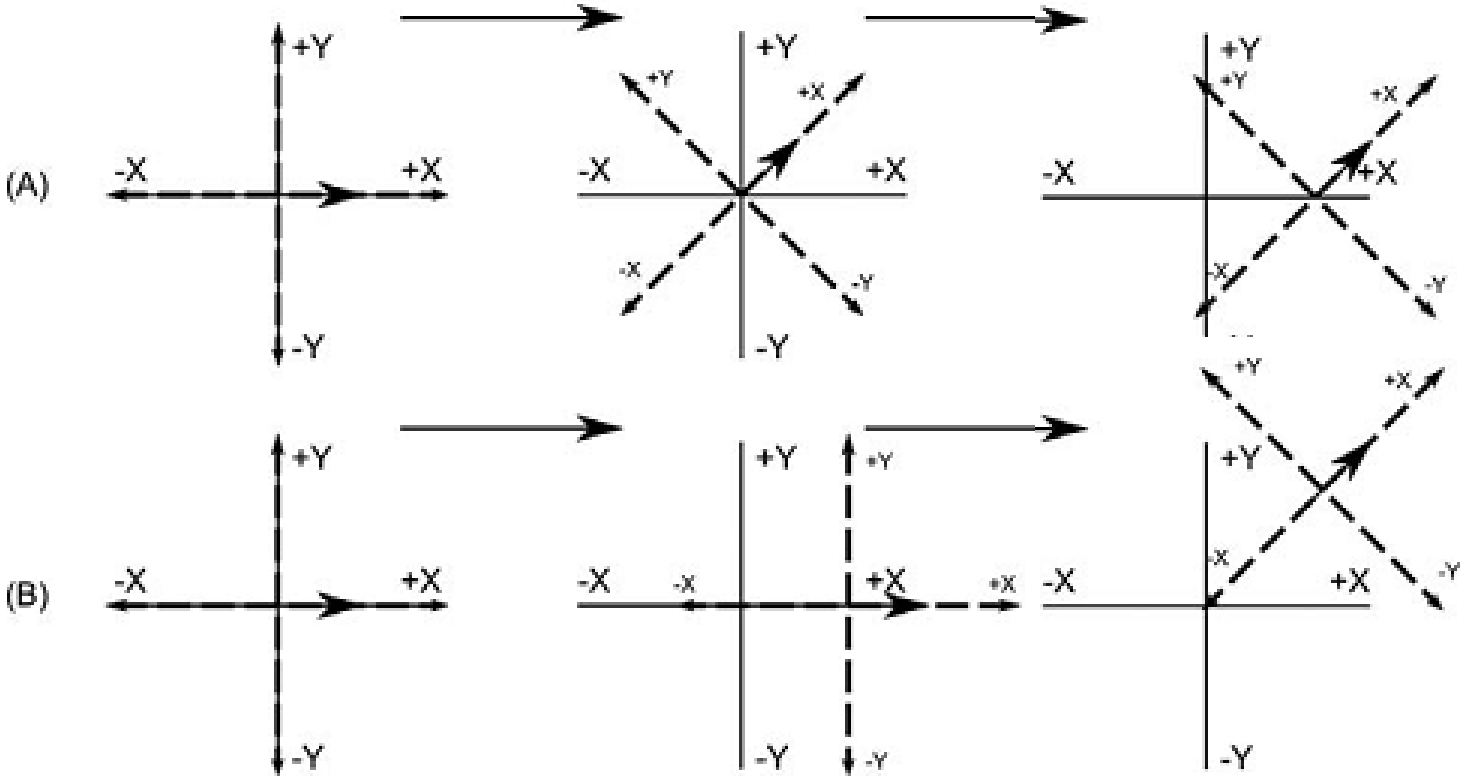
Modelling transform

- Comment le coder

1. Choisir la ModelView Matrix :
   glMatrixMode(GL_ModelView);
2. Vider la ModelView Matrix :
   glLoadIdentity()
3. Appliquer différentes transformations
4. Relancer le rendu de l'objet

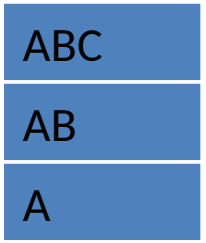
Transformations OpenGL

- Problématique de l'ordre des transformations



La pile des matrices OpenGL

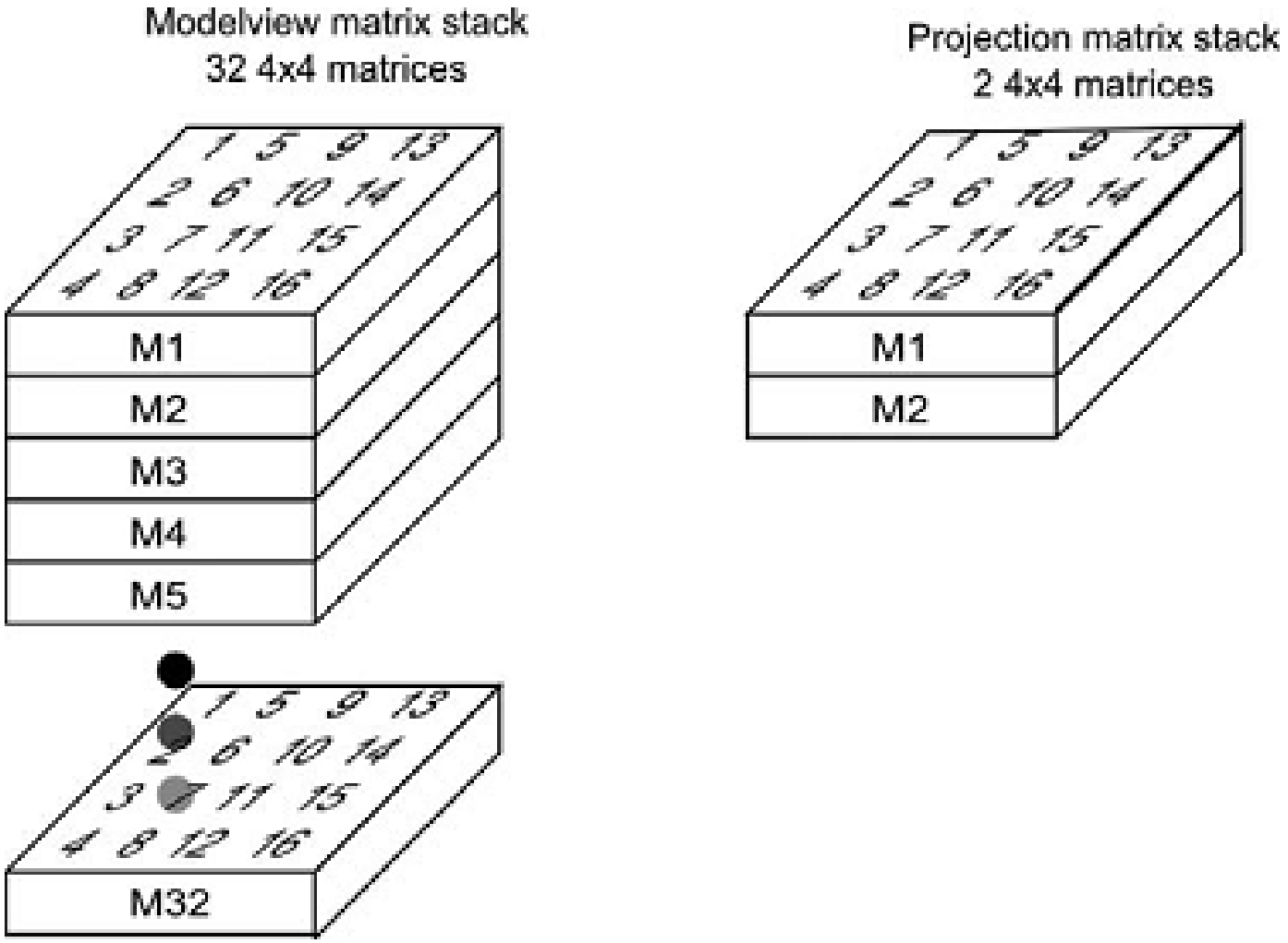
- Rappel: la multiplication de matrices est associative mais non commutative.
  - ABC = A(BC) = (AB)C ≠ ACB ≠ BCA
- Pré-multiplier les matrices qui seront utilisées plus d'une fois est plus rapide que de multiplier de nombreuses matrices à chaque rendu d'une primitive.
- OpenGL utilise des *piles de matrices* pour stocker les transformations ou la matrice en haut de pile est (généralement) le produit de toutes celles inférieures.
  - Ceci permet de construire une frame local de référence. Espace local — puis transformations depuis cet espace.



Piles de matrices

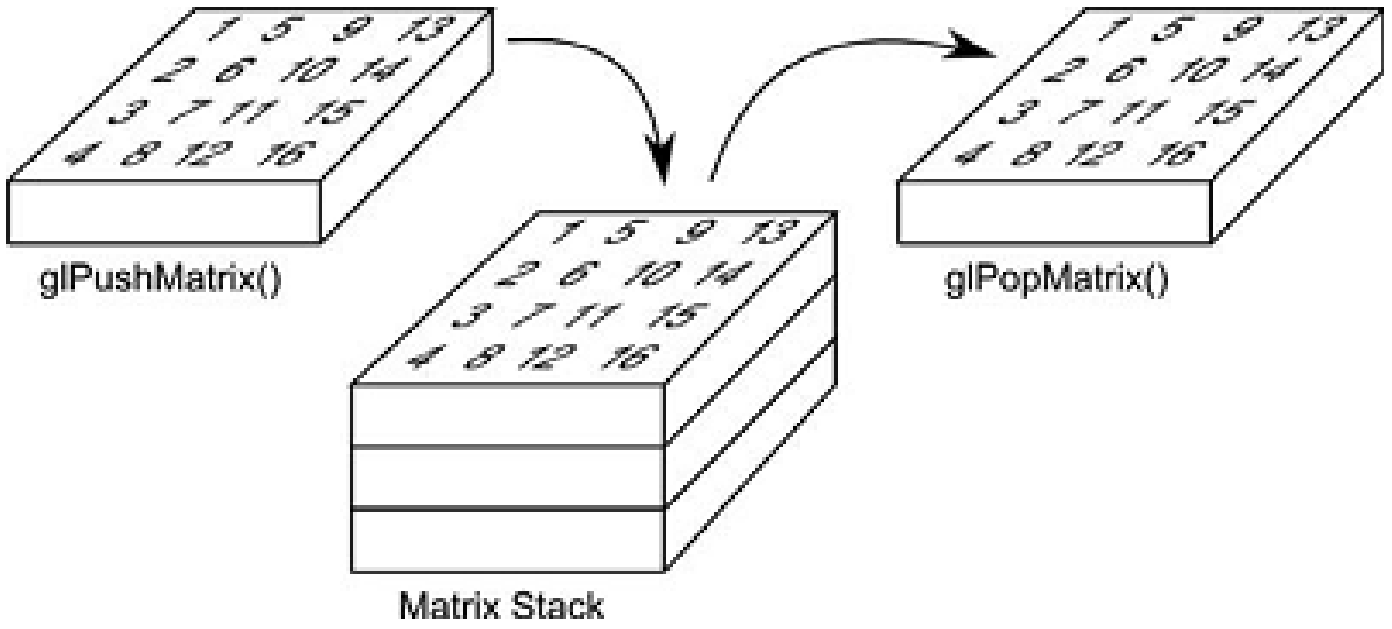
- Principe de pile de matrices est omniprésent
- Evident pour les tranformations spatiales
- Mais 4 types de piles de matrices existent
  1. Modelview matrix stack
  2. Projection matrix stack
  3. Color matrix stack
  4. Texture matrix stack

OpenGL pile des matrices



OpenGL

- Ajouter matrice courante en haut par **glPushMatrix()**
- Retirer le haut de pile par **glPopMatrix()**.



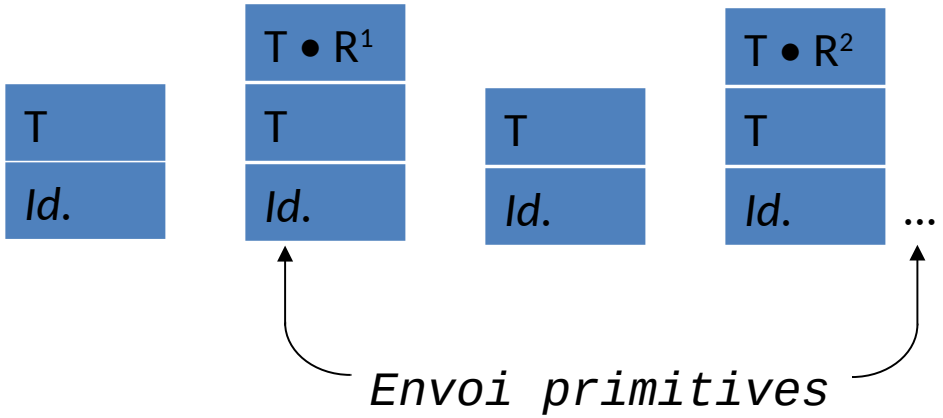
Exemple de rendu hiérarchique

```
void renderLevel(GL gl, int level, float t) {
  gl.glPushMatrix();
  gl.glRotatef(t, 0, 1, 0);
  renderSphere(gl);
  if (level > 0) {
    gl.glScalef(0.75f, 0.75f, 0.75f);
    gl.glPushMatrix();
      gl.glTranslatef(1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    gl.glPopMatrix();
    gl.glPushMatrix();
      gl.glTranslatef(-1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    gl.glPopMatrix();
  }
  gl.glPopMatrix();
}
```

Graphe de scène et pile de matrices

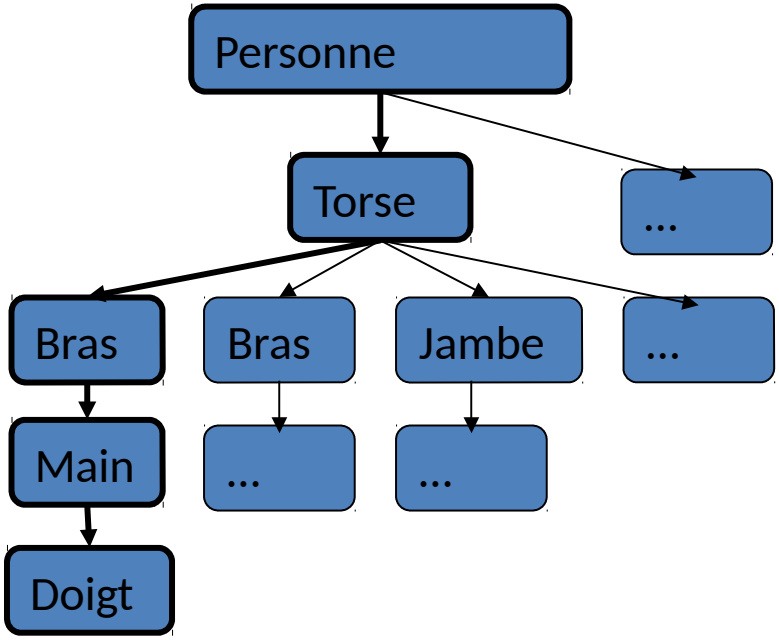
- Pile de matrices crée des **transformations relatives imbriquées**.

```
glPushMatrix();
  glTranslatef(0,0,-5);
  glPushMatrix();
    glRotatef(45,0,1,0);
    renderSquare();
  glPopMatrix();
  glPushMatrix();
    glRotatef(-45,0,1,0);
    renderSquare();
  glPopMatrix();
glPopMatrix();
```



Le graphe de scène

- Un *graphe de scène* est un arbre des éléments de la scène ou la transformation d'un fils est relatif à son parent.
- La transformation finale du fils est le produit ordonné de tous ses ancêtres dans l'arbre.
- La pile des matrices OpenGL se fait en profondeur dans l'arbre de scènes.

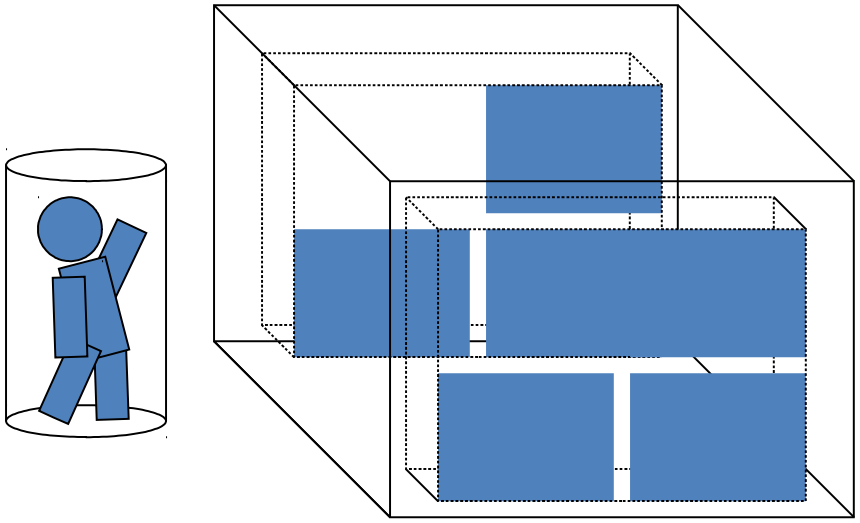


$M_{doigtToWorld} = (M_{person} \bullet M_{torso} \bullet M_{arm} \bullet M_{hand} \bullet M_{finger})$

Utilisation du graphe de scène

- Une optimisation habituelle dérivée du graphe de scène est la propagation des *bounding volumes*.
  - Il sont de plusieurs formes: *bounding spheres*, *axis-aligned bounding boxes*, *oriented bounding boxes*…
- *Nested bounding volumes* permet le culling rapide de larges portions de la géométrie.
  - Test des bounding volumes du haut du graphe de scène et descendant vers les fils.

- Parfait pour…
  - Detection de collision entre éléments.
  - Culling avant rendu
  - Ray-tracing accéléré



Le graphe de scène

- Many 2D GUIs today favor an event model in which events 'bubble up' from child windows to parents. This is sometimes mirrored in a scene graph.
  - Ex: a child changes size, which changes the size of the parent's bounding box
  - Ex: the user drags a movable control in the scene, triggering an update event
- If you do choose this approach, consider using the *model/ view/ controller* design pattern. 3D geometry objects are good for displaying data but they are not the proper place for control logic.
  - For example, the class that stores the geometry of the rocket should not be the same class that stores the logic that moves the rocket.
  - Always separate logic from representation.

Objet de scène possible

```
class Pt {
  float x, y, z;
}
```

floats suffisant ?

```
class Face {
  Pt[] verts;
  Pt normal;
}
```

flat or Gouraud shading

```
class SceneObject {
  SceneObject parent;
  List<SceneObject>
    children;
  Matrix4x4
    transform;
  Face[] faces;
}
```

List au lieu de array: on prévoit de nombreux dynamic updates?

Array au lieu de list: prévoit de pré-charger toutes les données.

4x4 matrix: séquence de transformations affines

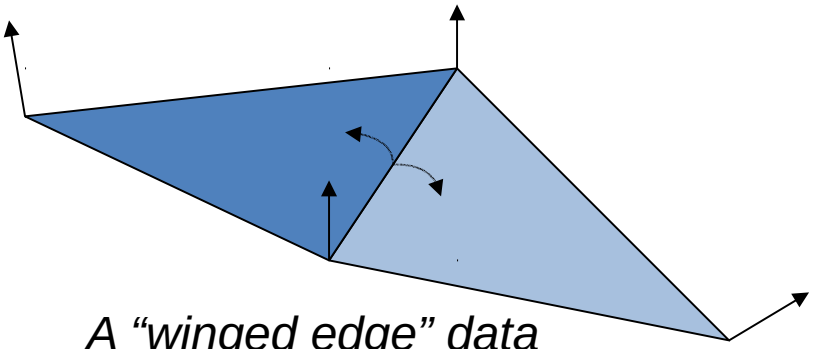
Structure de données

- When designing your data structures, consider *interactions* and *interrogations*.
  - If you're going to move vertices, you'll need to update faces.
  - If you're going to render with crease angles, you'll need to track edges.
  - If you want to be able to calculate vertex normals or curvature, your vertices will have to know about surrounding faces.
  - Is memory a factor? What about the processing overhead of dereferencing a pointer or array?

Améliorer l'objet de scène

- ***L'ordre des points compte***
  - Habituellement antihoraire par rapport à la normale
- Could store normal at the vertex
  - Phong shading
- Vertices could track faces
- Could introduce *edges*, tracking vertices and faces together

- Could store color at the face or at the vertex; depends on lighting model
- Same for other material traits (shading, bump maps, …)
- Texture data has to be at the vertices



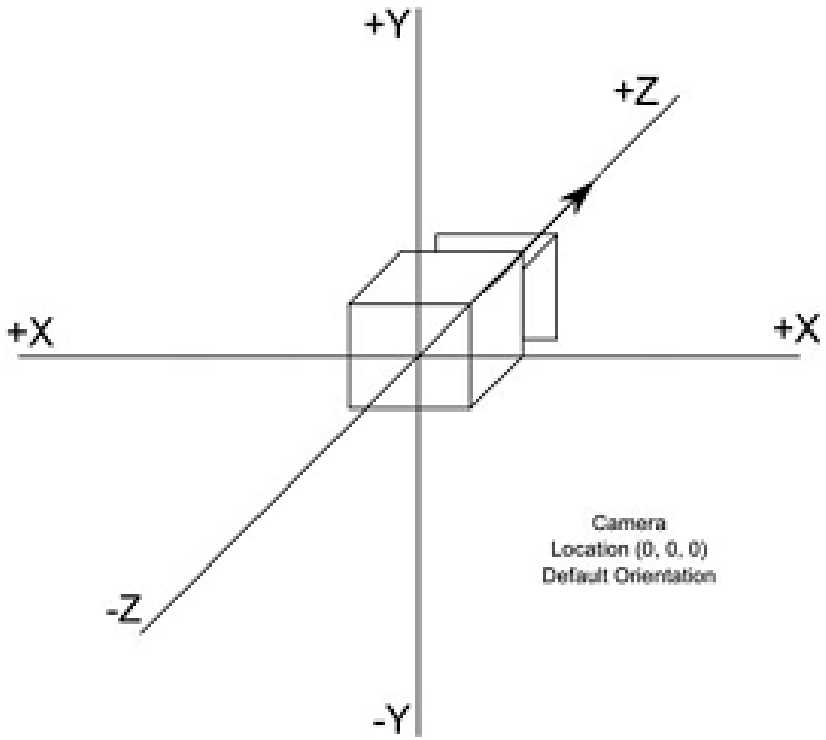
A "winged edge" data structure

Vues en OpenGL

- OpenGL maintient deux matrices de vues:
  - Model view matrix: positions des objets
  - Projection matrix: propriétés de camera
- Les projections peuvent petre
  - Perspective ou orthographique
- La caméra est *toujours* à l'origine regardant *vers les Z négatifs* (-Z).

Viewing transformation

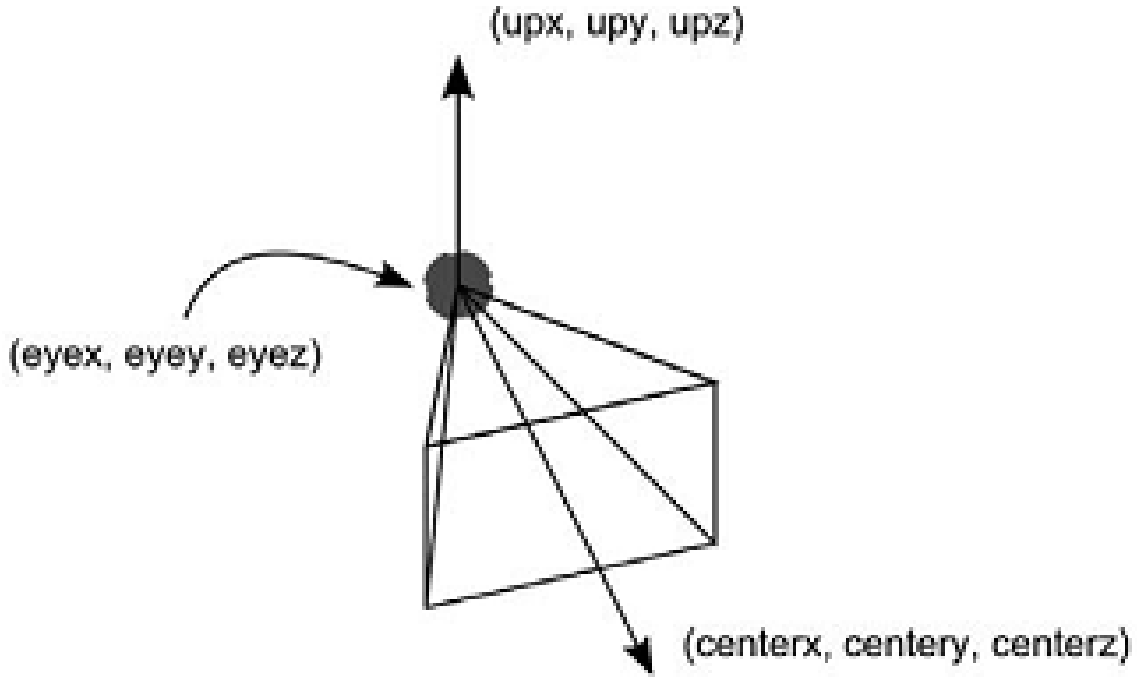
- Par défaut la caméra est une boîte [-1, 1]
- Regarde vers l'axe des Z négatifs



Viewing transformation

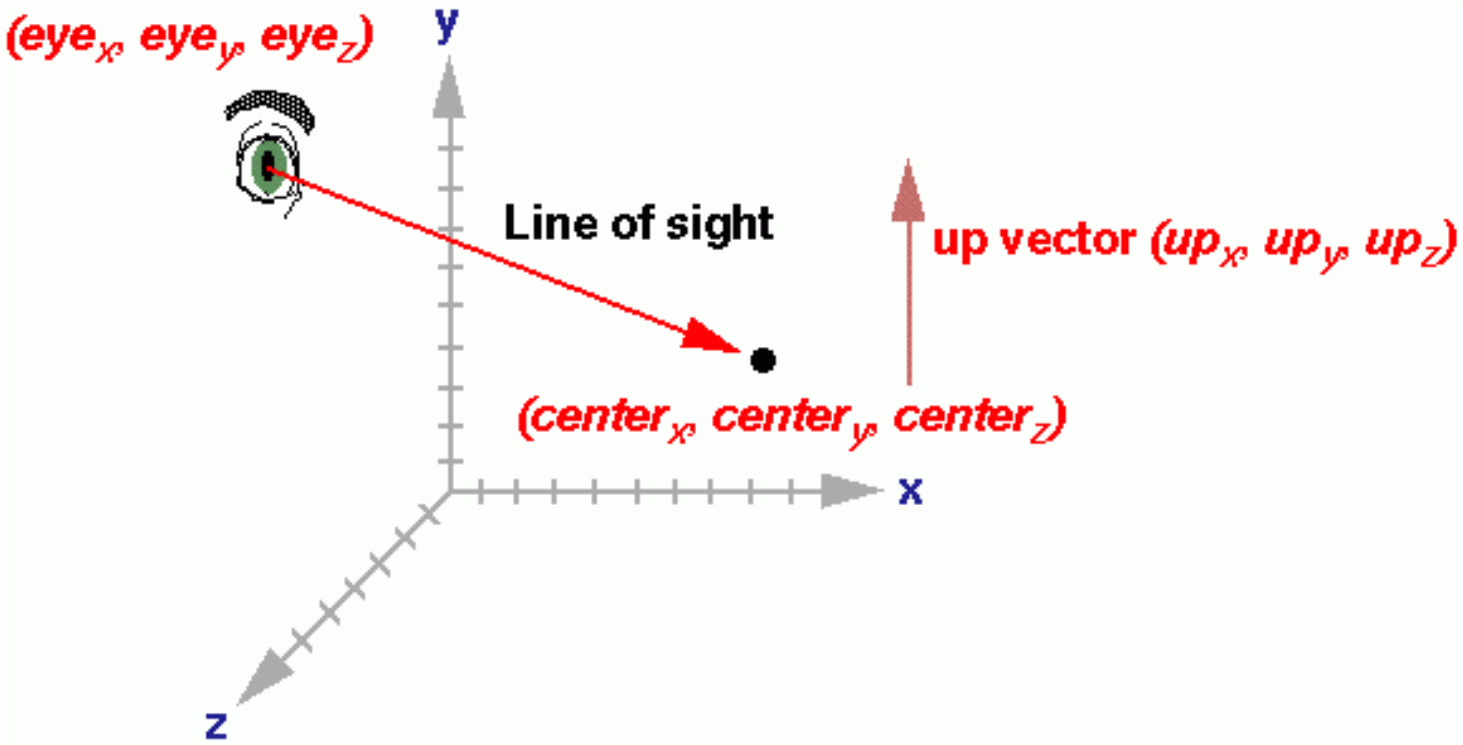
- Se simplifier la vie avec **gluLookAt**() :

  void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);



OpenGL caméra

```
1 GLvoid gluLookAt( GLdouble eyex, GLdouble eyey,
2                   GLdouble eyez, GLdouble centerx, GLdouble
3                   centery, GLdouble centerz, GLdouble upx,
4                   GLdouble upy, GLdouble upz )
```



Transformations projectives

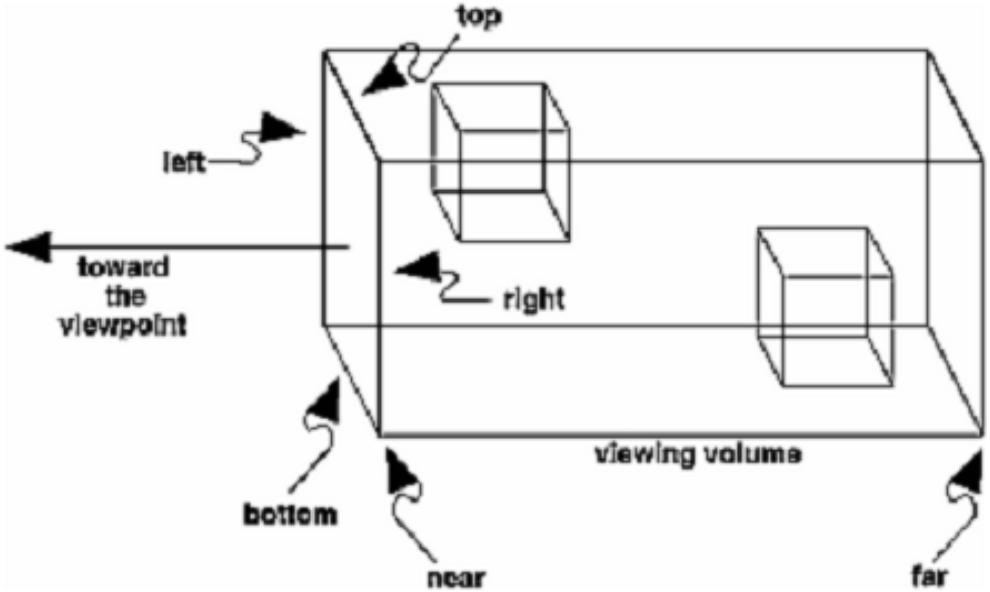
- 2 types de transformation projectives
  1. Projection perspective

     Transformation classique des univers 3D permettant de gérer la profondeur de vision.
  2. Projection orthographique

     Permet généralement de fixer le champ et la profondeur, typique des jeux 2D, programmes de CAD etc…

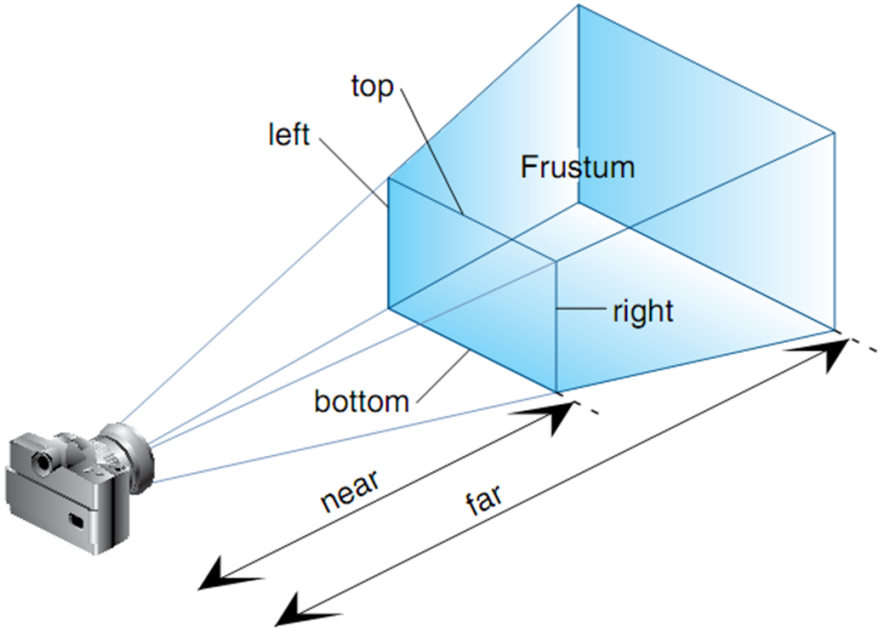
Transformation projective

- Orthographic projection avec glOrtho() etgluOrtho2D().
  - glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
  - gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

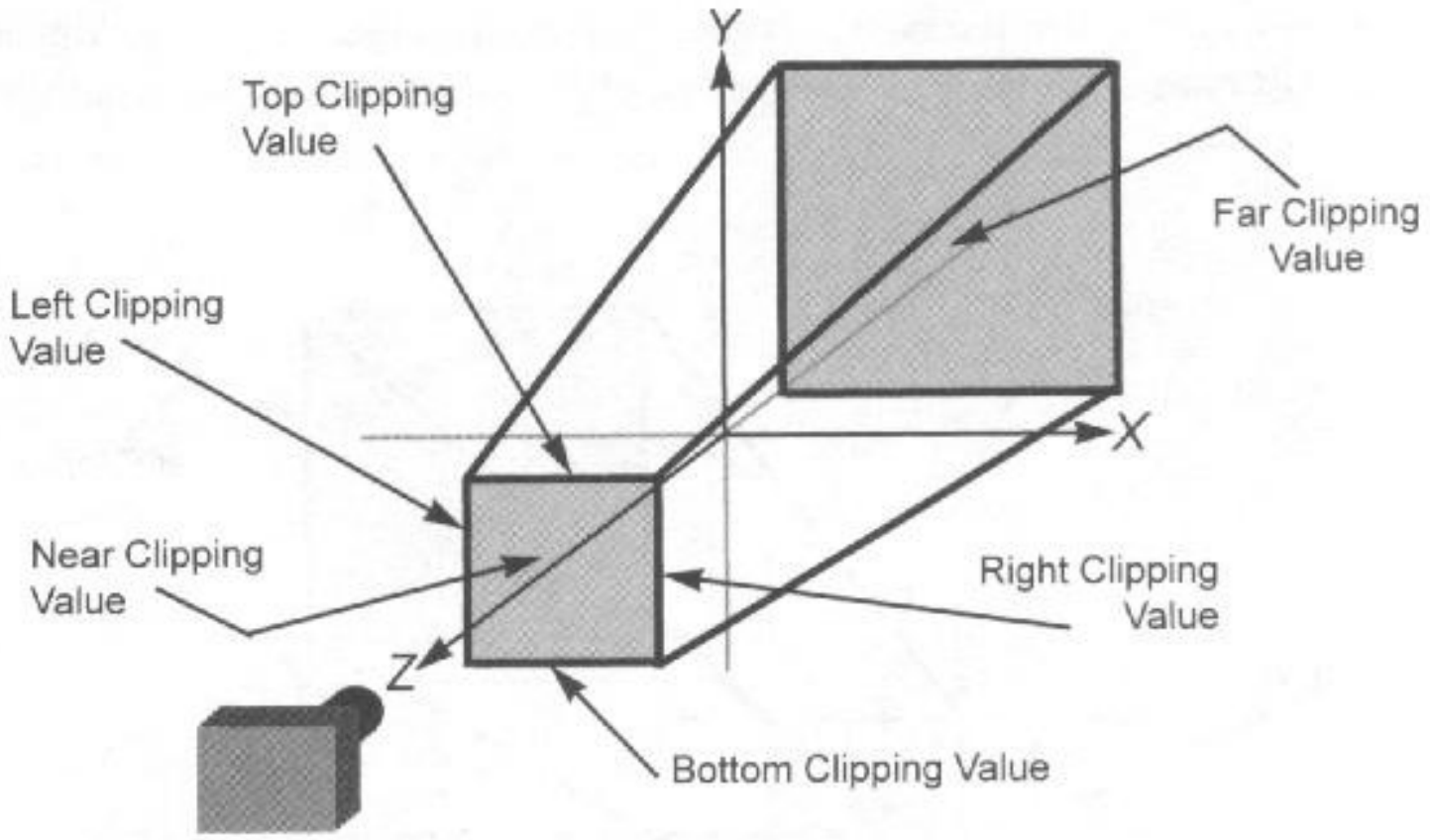


Transformation projective

- Projection avec **glFrustrum**() et **gluPerspective**().
  - void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

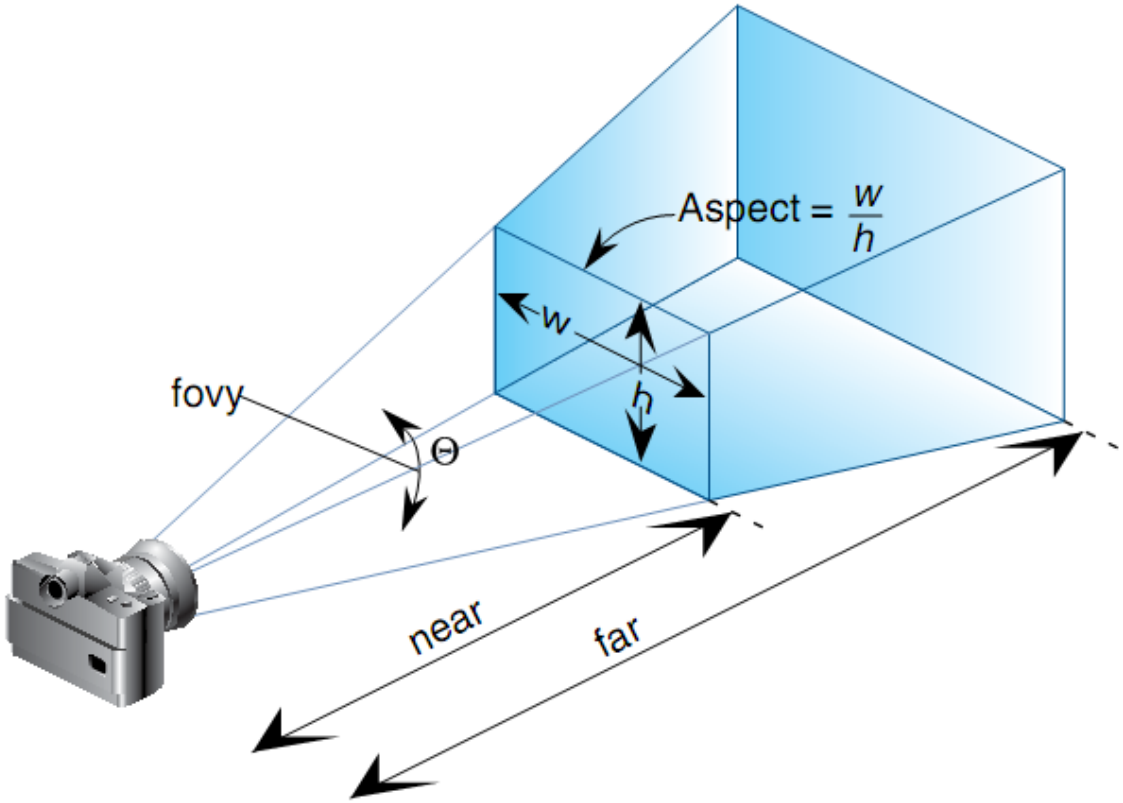


OpenGL


Y
Top Clipping Value
Far Clipping Value
Left Clipping Value
X
Near Clipping Value
Right Clipping Value
Z
Bottom Clipping Value

Transformation projective

- void **gluPerspective**(GLdouble fov, GLdouble aspect, GLdouble near, GLdouble far);



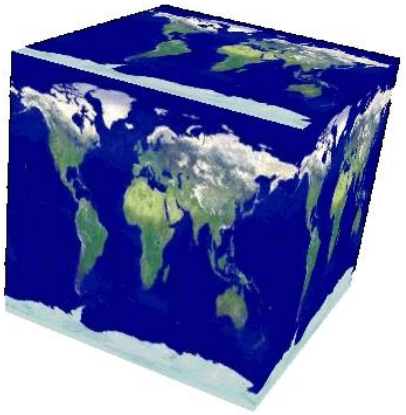
OpenGL

- Pixels and Bitmaps
- Displaying a bitmap
- Mixing Bitmaps and Geometry
- Colors and Masks
- Drawing Modes
- Reading and Writing Pixels
- Selecting Buffers
- Luminance
- Pixel Zoom

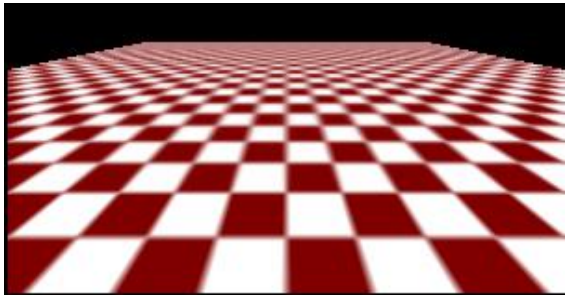
Texture mapping

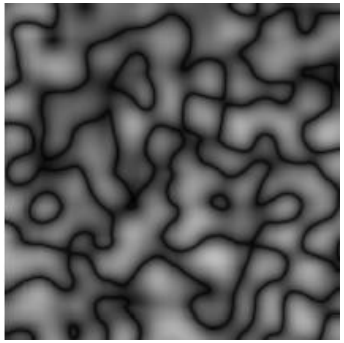
- Texture Coordinates
- Texture Parameters
- Applying textures to surface
- Minimaps

Textures



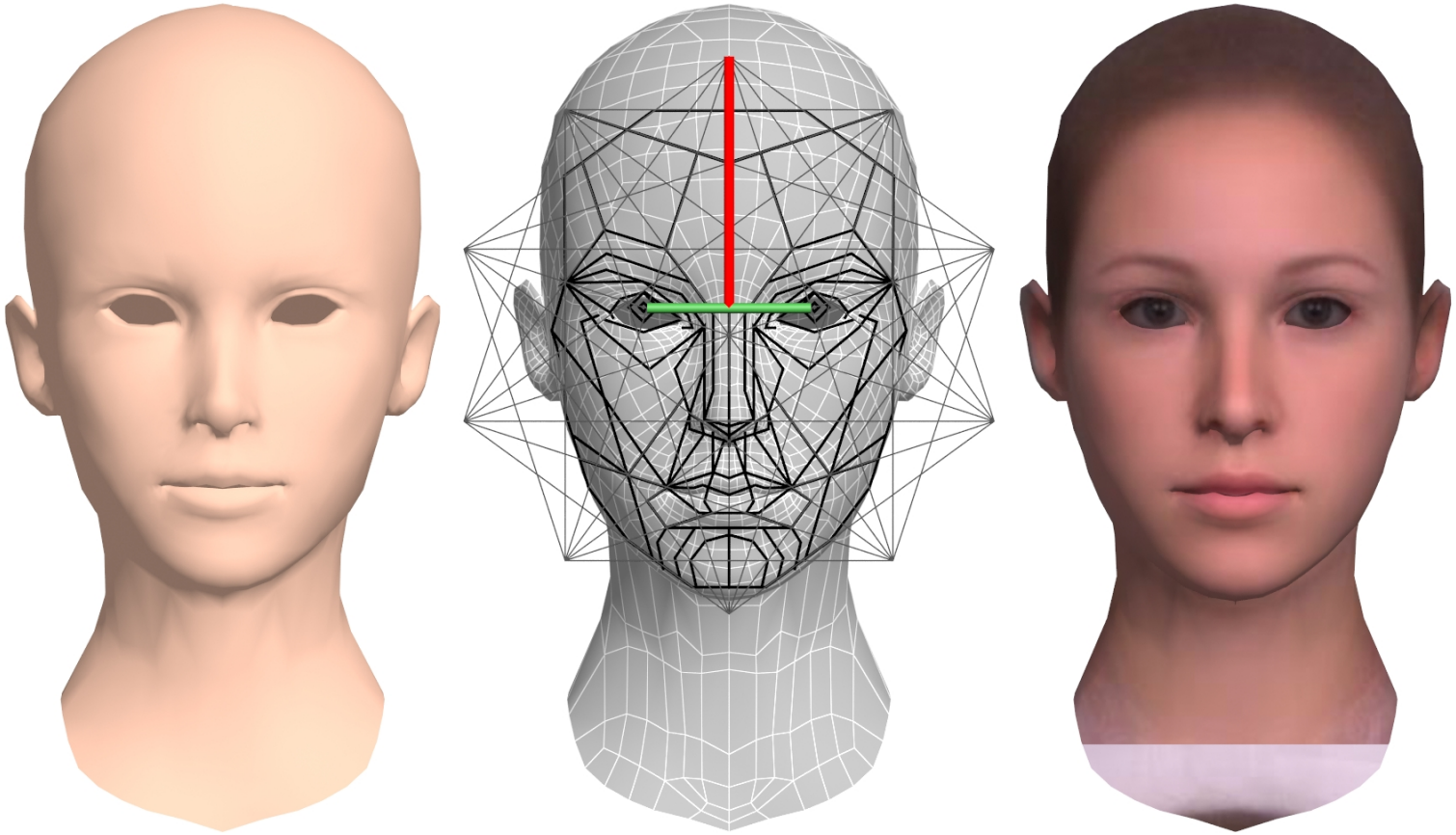
- Permet l'ajout de détails de surface
- Deux manières d'arriver à ce but:
  - ❖ Polygones supplémentaires pour modéliser les détails ?
    - ❖ Ajoute de la complexité à la scène
    - ❖ Ralentis considérablement le rendu graphique
    - ❖ Les détails sont très durs à modéliser
  - ✓ Mapper une texture à une surface





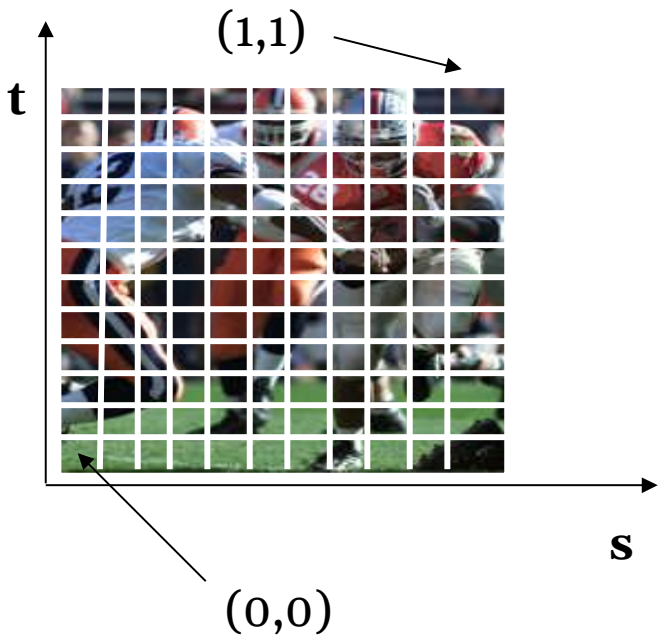
Complexité de l'image ne change rien à la complexité du calcul géométrique (transformation, clipping…)

OpenGL textures



Représentation des textures

- ✓ Bitmap (pixel map) textures (supporté par OpenGL)
- Procedural textures (utilisé en rendu avancé)



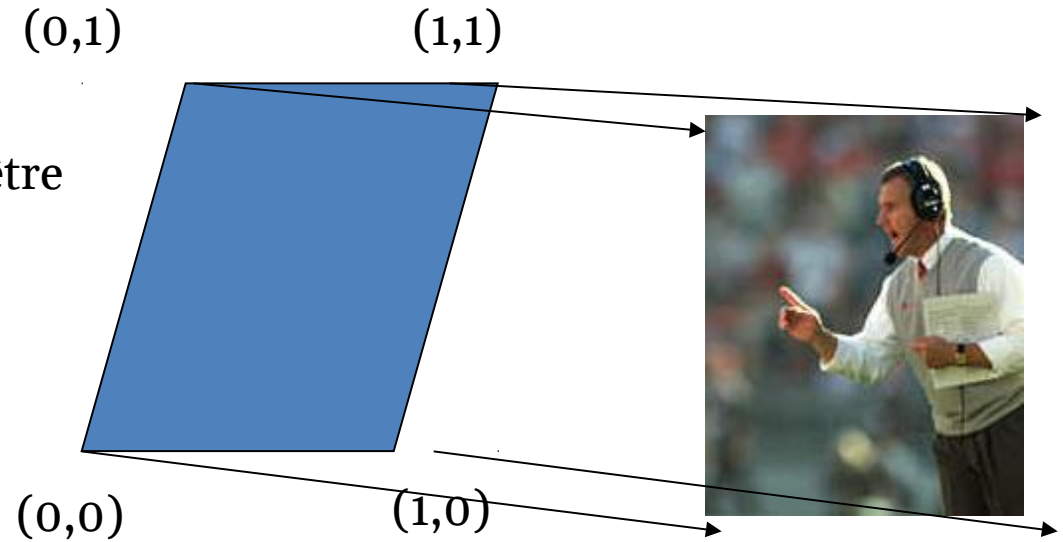
Bitmap texture:

- Une image 2D - representé par un 2D array texture[height][width]
- Chaque pixel (appellé **texel**) par une paire unique de coordonnées (s, t)
- Les s and t sont généralement normalisés dans l'intervalle [0,1].
- Pour tout (s,t) dans l'intervalle normalisé, il existe une unique valeur d'image (i.e., unique [red, green, blue])

Texture mapping

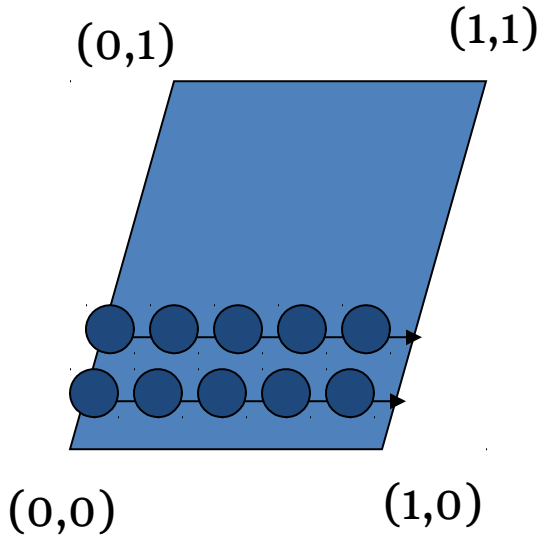
- Etablir un mapping depuis les textures vers les surfaces (polygones):
  - Le programme doit spécifier les coordonnées de texture pour chaque coin du polygone.

Le polygone peut être de taille arbitraire



Texture mapping

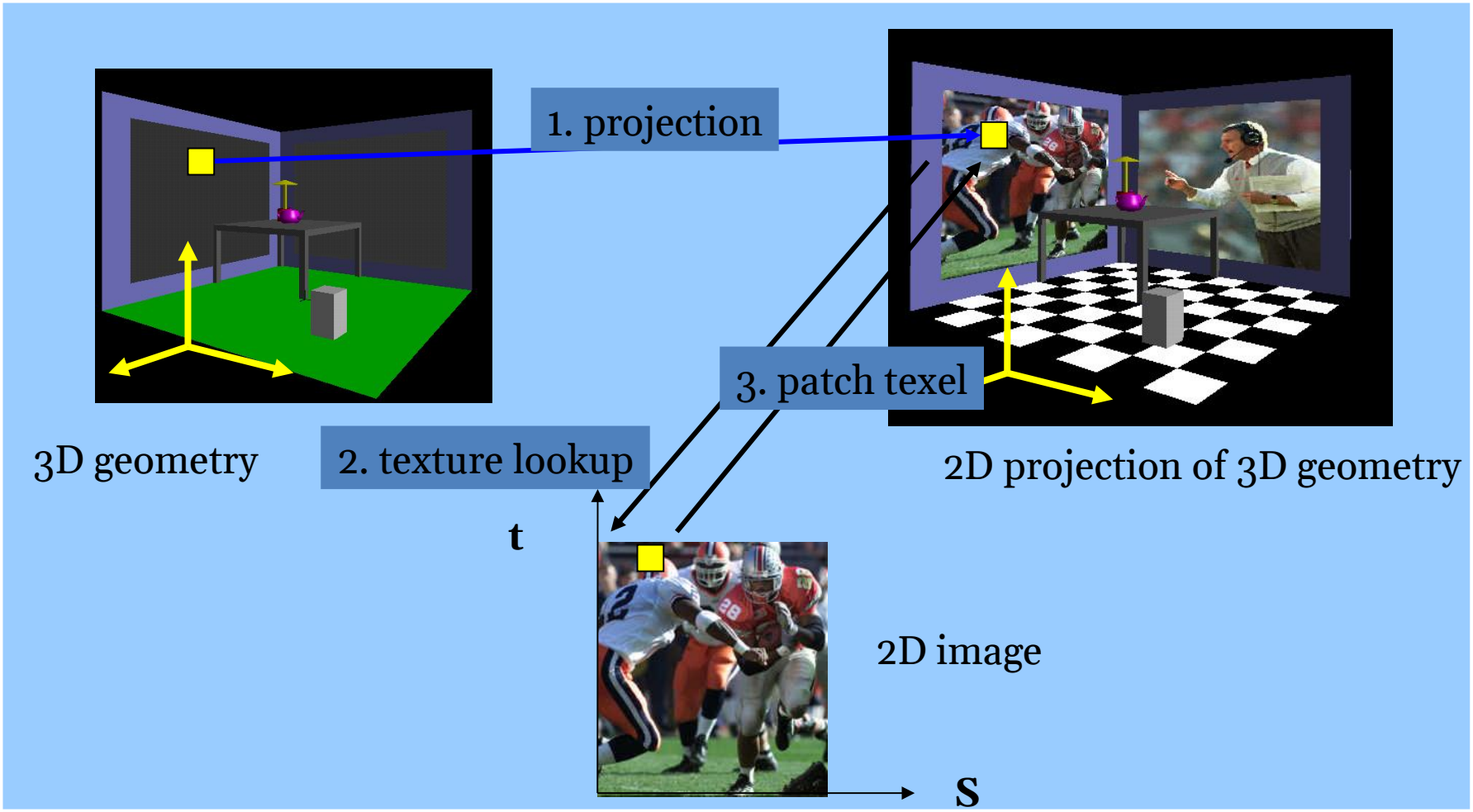
- Le *texture mapping* est effectué par *rasterization* (backward mapping)



- Pour chaque pixel à dessiner, ses coordonnées de texture (s, t) sont déterminées (interpolées) basés juste sur les angles principaux des coordonnées de texture (par rapport au polygone.

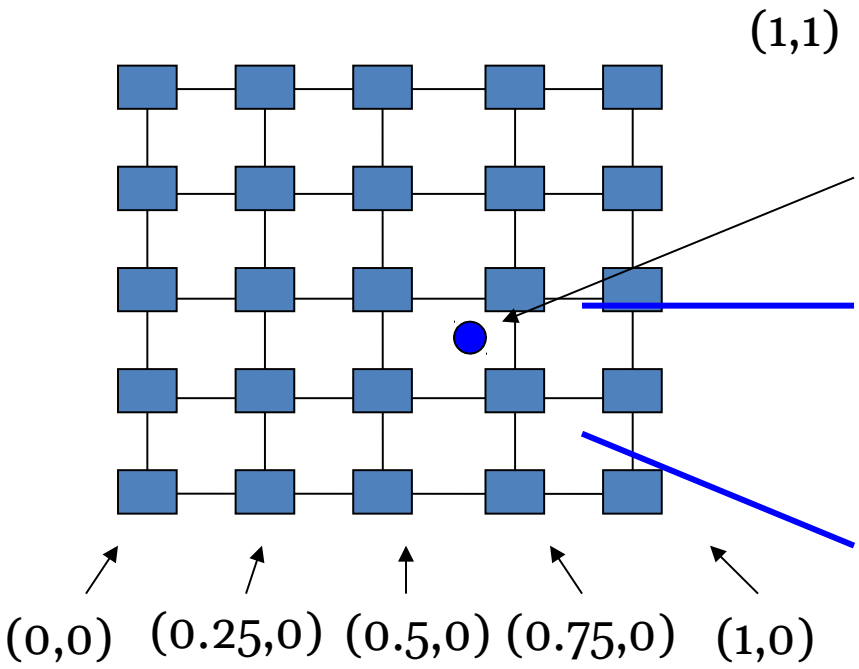
- Les coordonnées de texture interpolées sont ensuite utilisées pour effectuer un *texture lookup*.

Texture mapping

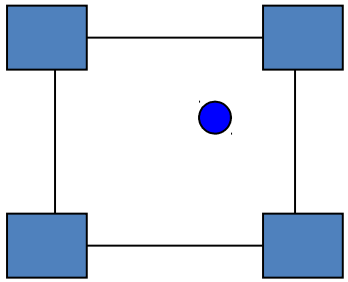


Texture value lookup

- Pour une coordonnée de texture donnée (s,t), on peut trouver une valeur unique dans l'image texture



Comment gérer les points n'étant pas exactement sur une coordonnée?



A) Nearest neighbor
B) Linear Interpolation
C) Other filters

Texture mapping

- Etapes du programme
  1) Spécifier la texture
     - Lire ou générer une image
     - Assigner à une texture
  2) Specifier les paramètres de texture mapping
     - Wrapping, filtering, etc.
  3) Activer le texture mapping (GL_TEXTURE_2D)
  4) Assigner les coordonnnées de texture aux points
  5) Dessiner les objets
  6) Désactiver le texture mapping (une fois que tous les objets texturés sont rendus)

Spécifier les textures

- Charger la texture depuis le disque dur (vers la mémoire des textures).

glTexImage2D(Glenum target, Glint level, Glint iformat, int width, int height, int border, Glenum format, Glenum type, Glvoid* img)

- Example:
  - glTeximage2D(GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0, GL_RGB, GL_UNSIGNED_BYTE, myImage);

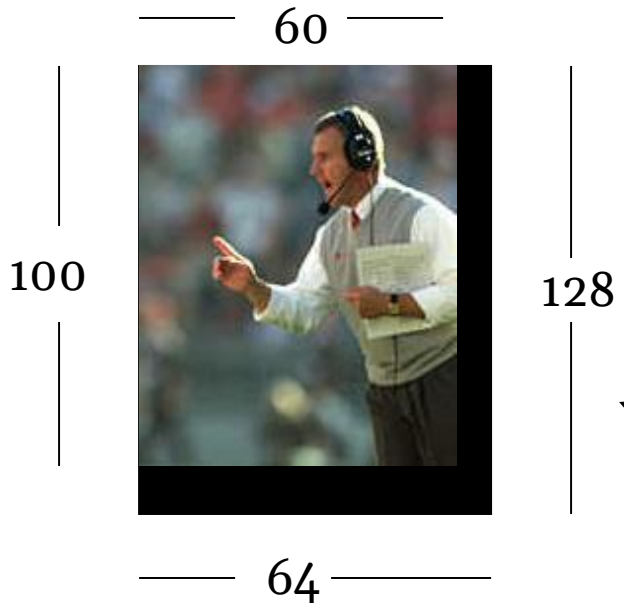
    (myImage est un 2D array: GLuByte myImage[64][64][3];)

- Les dimensions des images de texture **doivent être des puissances de 2**

OpenGL

- Si les dimensions de la texture ne sont pas des puissances de 2 …

1) Pad zeros 2) utiliser gluScaleImage()



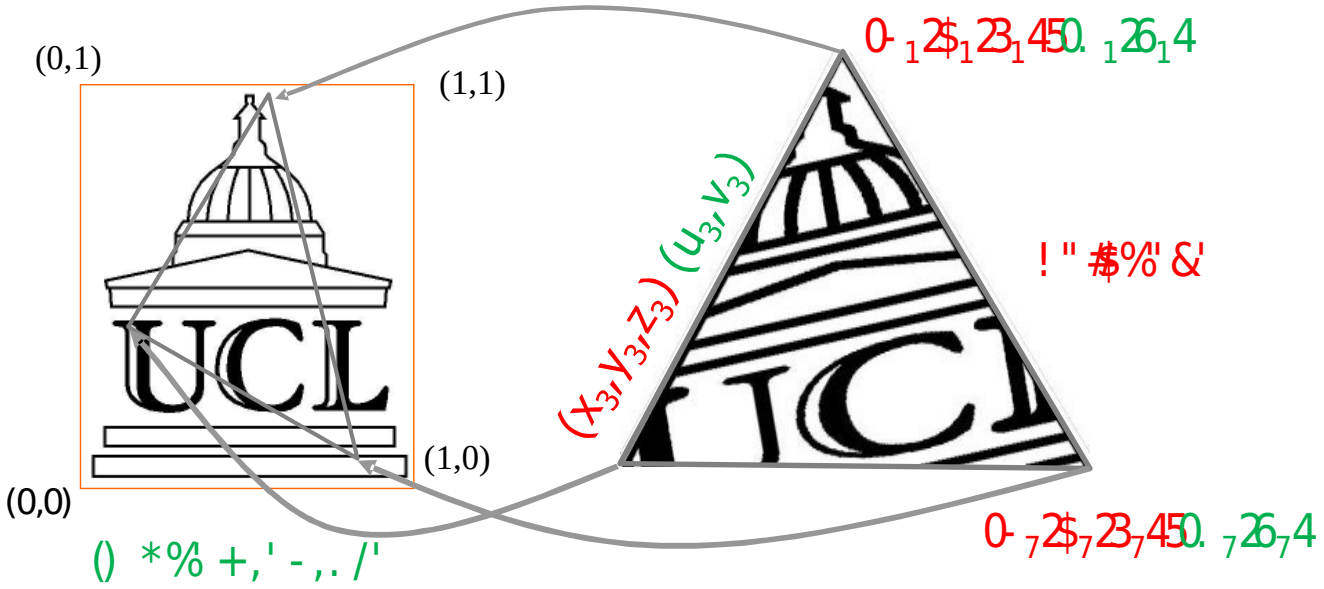
Demande à OpenGL de filtrer l'image … Mais induit un surcoût et dur à utiliser de manière simpliste

Il suffit par la suite d'affecter les coordonnées correctes pour pouvoir se débarasser des bandes noires

OpenGL: **textures**

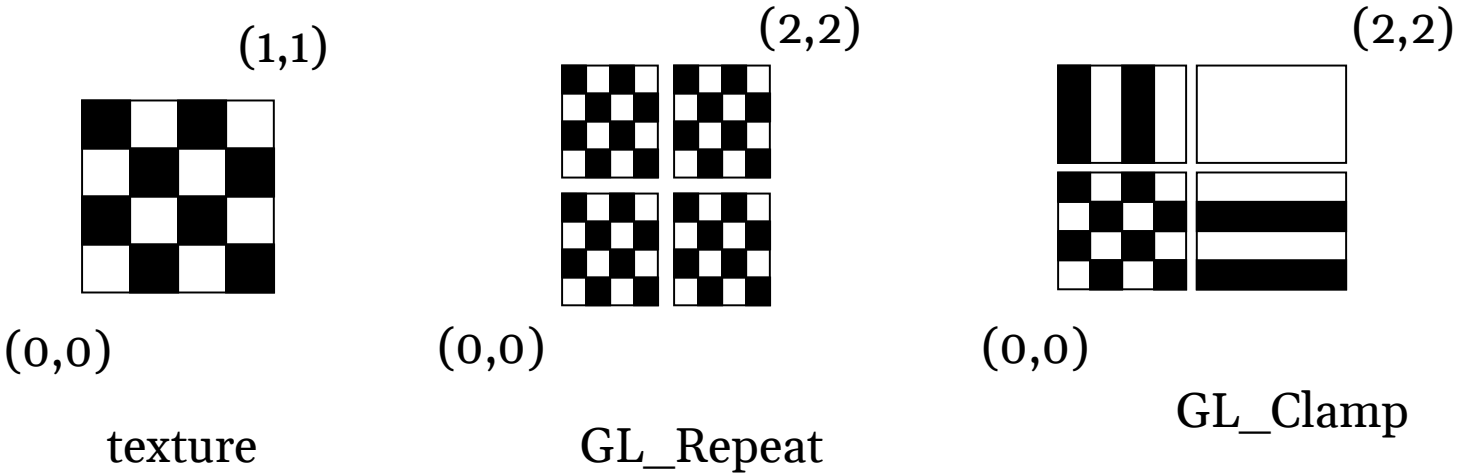
Pour des rendus plus agréables et pour le terrain, il faut gérer les images en plus de la 3D :

Plaquage de la texture



Texture mapping

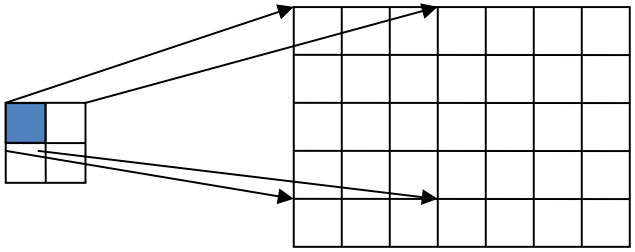
- Que se passe-t'il si les coordonnées de texture (s,t) sont en dehors de l'intervalle [0,1]?



- Example: glTexParameteri(GL_TEXTAURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)

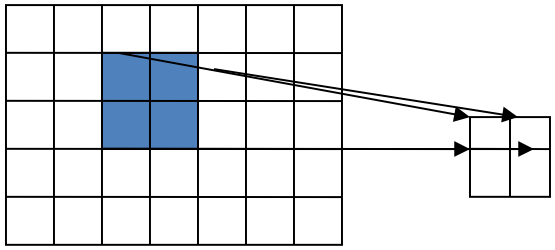
Texture mapping

- Comme un polygone peut être transformé vers un écran de taille arbitraire, les texels d'un texture map peuvent être magnifié ou minimisé (agrandissement vs. réduction).



texture polygon projection

Magnification



texture polygon projection

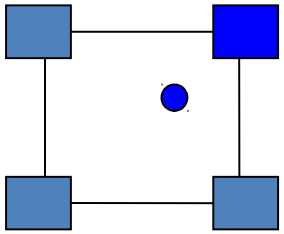
Minification

- Filtrage: interpolation de la valeur d'un texel value depuis ses voisins ou combinaisons de multiples texels en un seul.

Texture mapping

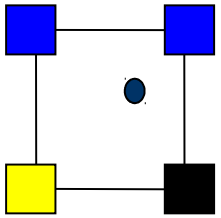
- OpenGL texture filtering:

1) Plus proche voisin
(qualité inférieure)



glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

2) Interpolation linéaire des voisins
(meilleure qualité, plus lent)



glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)

Or GL_TEXTURE_MAX_FILTER

Texture color blending

- Determiner comment combiner les couleurs des texel et celles de l'objet.
  - GL_MODULATE – multiplier les couleurs texture et objet
  - GL_BLEND – addition linéaire de couleurs texture et objet
  - GL_REPLACE – remplace la couleur objet par la texture

Example: glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

Activer (désactiver) les textures

- Activer les textures – glEnable(GL_TEXTURE_2D)
- Désactiver les textures – glDisable(GL_TEXTURE_2D)

Toujours désactiver les mappings de texture pour le dessin de polygones sans texture

Spécifier les coordonnées

- Spécifier les coordonnées de texture **avant** de définir chaque vertex.

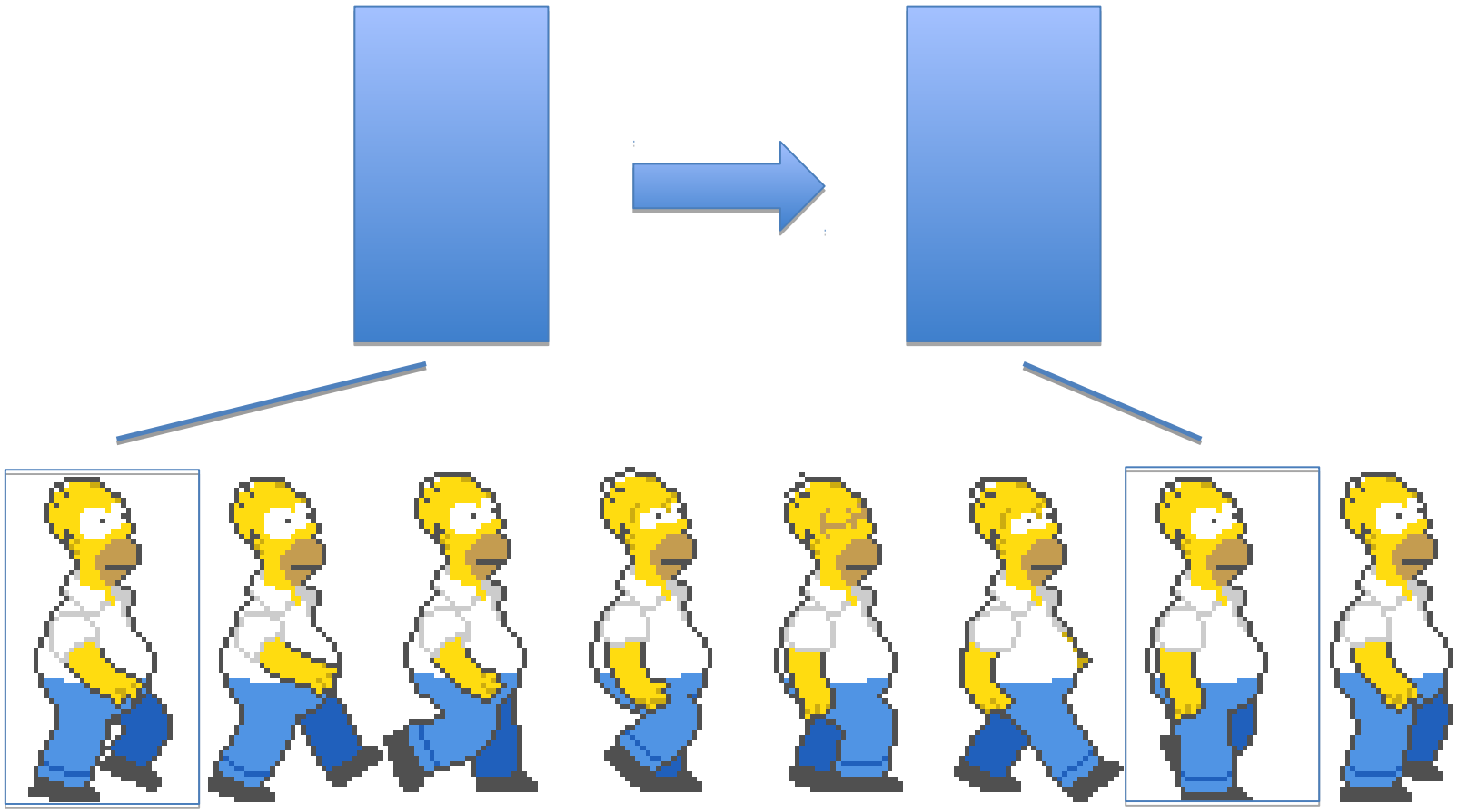
```
glBegin(GL_QUADS);
   glTexCoord2D(0,0);
  glVertex3f(-0.5, 0, 0.5);
  ...
 glEnd();
```

Tout mettre ensemble

```
…
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
…
glEnable(GL_TEXTURE_2D);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0, GL_RGB,
    GL_UNSIGNED_BYTE, mytexture);

Draw_picture1(); // define texture coordinates and vertices in the function
….
```

Sprite animation



OpenGL textures

- La classe possède un attribut Texture
- Une texture est stockée dans la mémoire graphique et possède un identifiant
- Texture = contrainte : souvent des images carrées, toujours en puissance de 2

```
1  public void drawGL()  {
2 //  chargement lors du premier appel
3    if(texture == null){
4     BufferedImage im = ToolsTerrain.imageFromCircuit(track); //  c
5     try {
6      ImageIO.write(im, "PNG", new File("terrain.png"));
7      texture = TextureLoader.getTexture("PNG", ResourceLoader.get
8     } catch (IOException e) {
9      e.printStackTrace();
10    }
11    width = texture.getTextureWidth();
12    height = texture.getTextureHeight();
13   }
14   ...
```

OpenGL textures

```
1  ...
2    GL11.glBindTexture(GL11.GL_TEXTURE_2D, texture.getTextureID());
3    GL11.glPixelStorei( GL11.GL_UNPACK_ALIGNMENT, 1 );
4
5    // draw a quad textured to match the sprite
6    GL11.glBegin(GL11.GL_QUADS);
7    {
8     GL11.glTexCoord2f(0, 0); GL11.glVertex3f(0, 0, 0);
9     GL11.glTexCoord2f(0, 1); GL11.glVertex3f(0, height, 0);
10    GL11.glTexCoord2f(1, 1); GL11.glVertex3f(width,height, 0);
11    GL11.glTexCoord2f(1, 0); GL11.glVertex3f(width,0, 0);
12   }
13   GL11.glEnd();
14   GL11.glBindTexture(GL11.GL_TEXTURE_2D, 0);
15
16  }
```

Courbes et surfaces

- Bezier Curves and surfaces
- One dimensional OpenGL evaluators for Bezier Curves
- Two dimensional evaluators to evaluate Bernstein polynomials and to form Bezier surfaces.

OpenGL

- RAZ du buffer
- Définition d'une scène

```
1  private void glsettings(){
2    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
3    glViewport(0, 0, getWidth(), getHeight());// 100% de la fenêtre
4    glClearColor(0.f, 0.f, 0.f, 1.0f);
5    glMatrixMode(GL_PROJECTION);
6
7    glLoadIdentity(); // RAZ transformations
8
9    // definition de la taille de la fenetre 3D où l'on regarde
10
11   glOrtho((float) -getWidth()/zoom, (float) getWidth()/zoom,
12         (float) -getHeight()/zoom/0.5, (float) getHeight()/zoom/1
13
14   glMatrixMode(GL_MODELVIEW);
15 ...
```

Système de lumières

- Activer/Desactiver OpenGL lighting
- Spécifier une source lumineuse
- Contrôler le calcul de lumière
- Smooth Shading
- Transparence

OpenGL

- Qu'en est-il de la lumière ?
- Il est possible de spécifier des lumières (état OpenGL)
- On règle alors position, couleur, intensité, etc…
- D'abord définir une lumière
  - `float lightColor[] = {1.0f, 1.0f, 1.0f};`
  - `gl.glLightfv(gl.GL_LIGHT0, gl.GL_COLOR, lightColor ); (similar for position, etc)`
- Ensuite l'activer
  - `glEnable(gl.GL_LIGHTING);`
  - `glEnable(gl.GL_LIGHT0);`

OpenGL

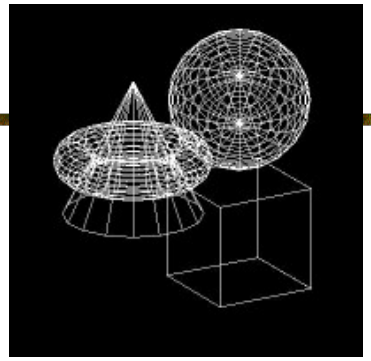
- Les matériaux sont également des états
  - `float materialColor[] = {1.0f, 1.0f, 0.0f};`
  - `gl.glMaterialfv(gl.GL_FRONT, gl.GL_DIFFUSE, materialColor);`
- Généralement les lumières et matériaux basiques sont moins utilisé dans les moteurs de jeu

OpenGL

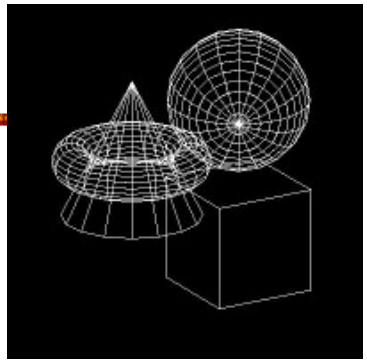
- Définition des lumières/matériaux

```
1  private void glsettings(){
2   [...]
3   initLightArrays(); // code fourni sur le site web
4
5   glShadeModel(GL_SMOOTH); // mode de rendu
6
7   glMaterial(GL_FRONT, GL_SPECULAR, matSpecular);
8   glMaterialf(GL_FRONT, GL_SHININESS, 50.0f);
9
10  glLight(GL_LIGHT0, GL_POSITION, lightPosition);
11  glLight(GL_LIGHT0, GL_SPECULAR, whiteLight);
12  glLight(GL_LIGHT0, GL_DIFFUSE, whiteLight);
13  glLightModel(GL_LIGHT_MODEL_AMBIENT, lModelAmbient);
14
15  glEnable(GL_LIGHTING); // activation
16  glEnable(GL_LIGHT0);
17  glEnable(GL_COLOR_MATERIAL);
18  glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
19
20  glLoadIdentity();
```

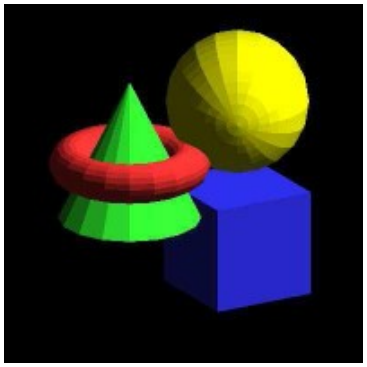
OpenGL



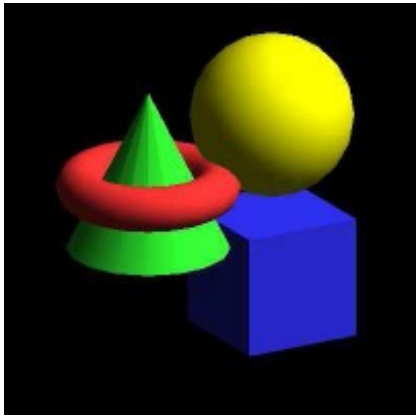
Fil de fer



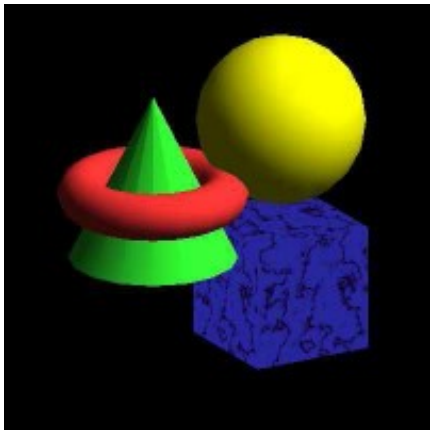
Faces cachées (objet)



Rendu Gouraud



Rendu Phong



Texture

Interaction and animation

- Reshape and Idle Callback
- Double buffering
- Using keyboard and mouse
- Mouse motion
- Subwindows and Multiple Windows

Vertex Arrays

- Le mode immédiat est parfait pour le dessin géométrique simple.
- Cependant les modèles plus complexes (personnages, …) requiert une grande quantité de vertex.
- Les vertex arrays permettent d'augmenter le mode immédiat par une gestion directe.
  - Dessin successif de triangles.
  - Pas encore accès aux accélérations GPU.

Vertex Arrays

- Pour travailler avec des vertex arrays :
  1. Activer l'utilisation des vertex arrays
  2. Définir le format et contenu des données
  3. Utiliser le rendu de vertex array.

Vertex Arrays

1. Activation et désactivation des vertex arrays
   - glEnableClientState(GLenum cap);
   - glDisableClientState(GLenum cap);
- Les paramètres des fonctions glEnableClientState() dan glDisableClientState() dépendent d'un ensemble de **array type flags**

Vertex Arrays

- Différent array type flags :
  - **GL_COLOR_ARRAY** Enables an array containing primary color information for each vertex
  - **GL_EDGE_FLAG_ARRAY** Enables an array containing edge flags for each vertex
  - **GL_INDEX_ARRAY** Enables an array containing indices to a color palette for each vertex
  - **GL_NORMAL_ARRAY** Enables an array containing the vertex normal for each vertex
  - **GL_TEXTURE_COORD_ARRAY** Enables an array containing the texture coordinate for each vertex
  - **GL_VERTEX_ARRAY** Enables an array containing the position of each vertex
  - **GL_SECONDARY_COLOR_ARRAY** Enables an array containing secondary colors

Vertex Arrays

2. La définition des données passe par les pointeurs GL
   - gl*Pointer().

a. Définition des coordonnées vertex :
   - void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *array);
     - size : 2 (x,y), 3 (x,y,z), 4(x,y,z,w)
     - type : GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
     - array : tableau contenant les données
     - stride : taille de padding (en bytes)
   - Exemple

```
glVertexPointer(3, GL_FLOAT, 0, &myArray[0]);
```

Vertex Arrays

b. Pour l'utilisation des couleurs

- void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *array);
  - size : 3 (r,g,b), 4(r,g,b,a)
  - type : GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE.
  - array : tableau contenant les données
  - stride : taille de padding (en bytes)
- Exemple :

```
glColorPointer(3, GL_FLOAT, 0, &myArray[0]);
```

Vertex Arrays

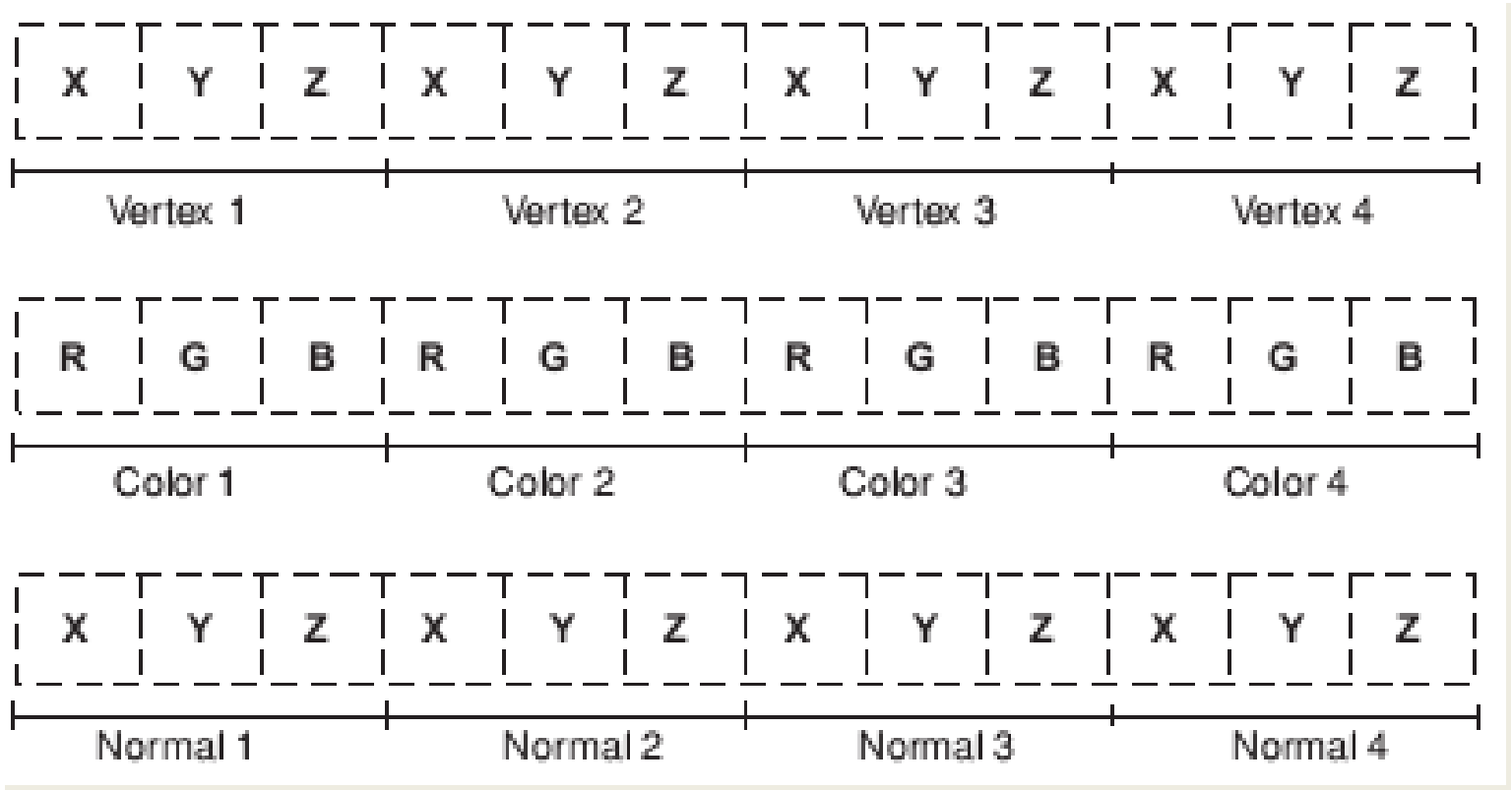
c. Et de même pour les textures

- void glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const Glvoid *array);
  - size : fonction des coordonnées par vertex (1,2,3,4)
  - type : GL_SHORT,GL_INT, GL_FLOAT, GL_DOUBLE.
  - array : tableau contenant les données
  - stride : taille de padding (en bytes)
- Exemple :

```
glTexCoordPointer(2, GL_FLOAT, 0,&myArray[0]);
```

Vertex Arrays

- Illustration de vertex arrays



Vertex Arrays

3. Rendu des vertex array :
   - glDrawArrays().
   - glDrawElements().

Vertex Arrays

- glDrawArrays() : rendu fait sur la base de **primitives**
  - void glDrawArrays(GLenum mode, GLint first, GLsizei count); :
    - mode : type primitive
    - first : index du premier élément à utiliser
    - count : taille de chaque élément
  - Exemple (dessin d'un triangles à 3 vertex) :

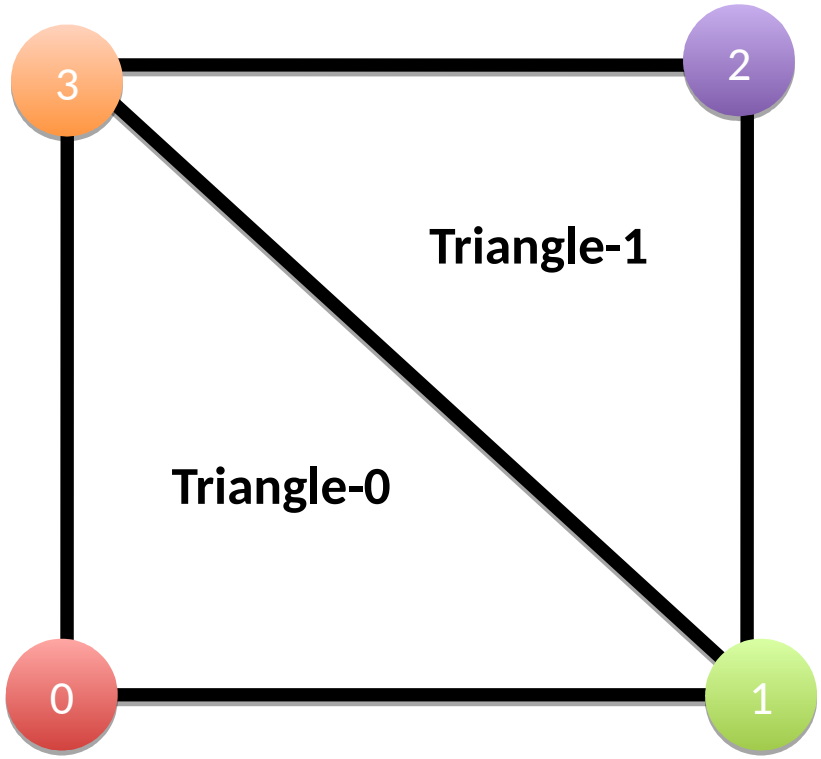

```
//Enable the vertex array
glEnableClientState(GL_VERTEX_ARRAY);
//Tell OpenGL where the vertices are
glVertexPointer(3, GL_FLOAT, 0, &m_vertices[0]);
//Draw the triangle, starting from vertex index zero
glDrawArrays(GL_TRIANGLES, 0, 3);
//Finally disable the vertex array
glDisableClientState(GL_VERTEX_ARRAY);
```

Vertex Arrays

```
//Enable the vertex array
glEnableClientState(GL_VERTEX_ARRAY);
//Tell OpenGL where the vertices are
glVertexPointer(3, GL_FLOAT, 0, &m_vertices[0]);
//Draw the triangles, we pass in the number of
indices, the data type of
//the index array (GL_UNSIGNED_INT) and then the
//pointer to the start of the array
glDrawElements(GL_TRIANGLES, m_indices.size(),
GL_UNSIGNED_INT, &m_indices[0]);
//Finally disable the vertex array
glDisableClientState(GL_VERTEX_ARRAY);
```

Vertex Arrays

- Ilustration et avantages :



Vertex Buffer Objects

- Les vertex array permettent d'éviter la lourdeur du mode immédiat …
- Mais sont tout aussi gourmants en RAM
- Pour pouvoir utiliser les accélérations GPU
- Utilisation des VBO (Vertex Buffer Objects)

Vertex Buffer Objects

- Les grandes étapes d'un VBO :
  1. Generation (allocation) d'un buffer.
  2. Activation (bind) du buffer.
  3. Remplissage du buffer avec les données.
  4. Utilisation du buffer pour le rendu des données.
  5. Libération (destroy) du buffer.

Vertex Buffer Objects

1. La génération et destruction de buffer nommés avec glGenBuffers() et glDeleteBuffers().
   - void glGenBuffers(GLsizei n, GLuint *buffers);
   - void glDeleteBuffers(GLsizei n, const GLuint *buffers);
     - buffers : pointer d'identification du buffer
     - n : taille de buffers requis

```
GLuint bufferID;
//Generate the name and store it in bufferID
glGenBuffers(1, &bufferID);
// Do some initialization and rendering with the buffer
// Release the name
glDeleteBuffers(1, &bufferID);
```

Vertex Buffer Objects

2. Activation (bind) de buffer avec glBindBuffer().
   - void glBindBuffer(GLenum target, GLuint buffer);
     - target : GL_ARRAY_BUFFER : per-vertex data (positions, colors, normals, etc.), GL_ELEMENT_ARRAY_BUFFER : index des vertex, GL_PIXEL_PACK_BUFFER etGL_PIXEL_UNPACK_BUFFER : données des pixel.
     - buffer : nom et ID du buffer.

```
glBindBuffer(GL_ARRAY_BUFFER, bufferID);
```

Vertex Buffer Objects

3. Remplissage du buffer avec glBufferData().
   - void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, Glenum usage);
     - target : GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER
     - size : taille du vertex array en bytes
     - data : pointer vers les données
     - usage : type du buffer, GL_{Buffer Frequency Values}_{Buffer Access Values} : GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ,or GL_DYNAMIC_COPY

Vertex Buffer Objects

- Buffer frequency values
  - **STREAM** The data will be modified only once, and accessed only a few times.
  - **STATIC** The data will be altered once and accessed multiple times (this hint is good for static geometry).
  - **DYNAMIC** The buffer will be modified a lot and accessed many times (this is suitable for animated models).

Vertex Buffer Objects

- Buffer access values
  - **DRAW** The contents of the buffer will be altered by the application and will be used for rendering using OpenGL.
  - **READ** The contents will be filled by OpenGL and then subsequently read by the application.
  - **COPY** The contents will be modified by OpenGL and then later used as the source for rendering.

```
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 3,
&vertex[0], GL_STATIC_DRAW);
```

Vertex Buffer Objects

4. Utilisation du buffer pour le rendu

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,m_indexBuffer);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glDrawElements(
  GL_TRIANGLES,      // mode
  m_indices_copy.size(),  // count
  GL_UNSIGNED_INT,   // type
  BUFFER_OFFSET(0)   // element array buffer offset
  );
glDisableClientState(GL_VERTEX_ARRAY);
```

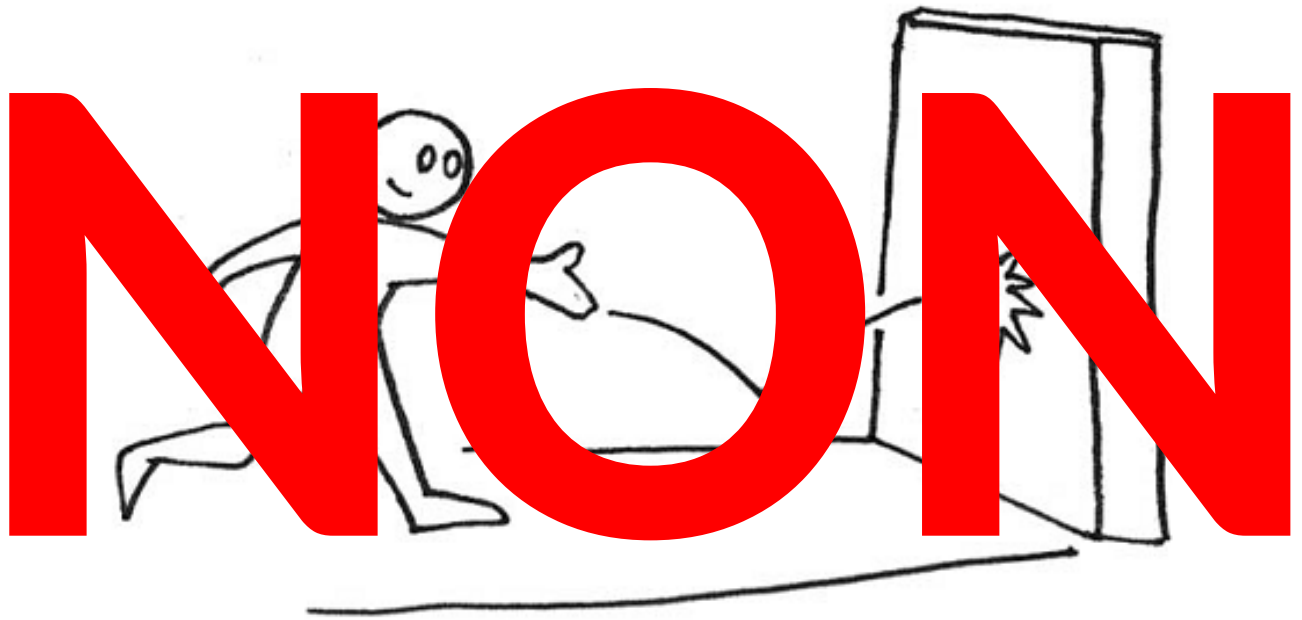
Jeu vidéo

Jeu vidéo

Est-ce un jeu vidéo ?

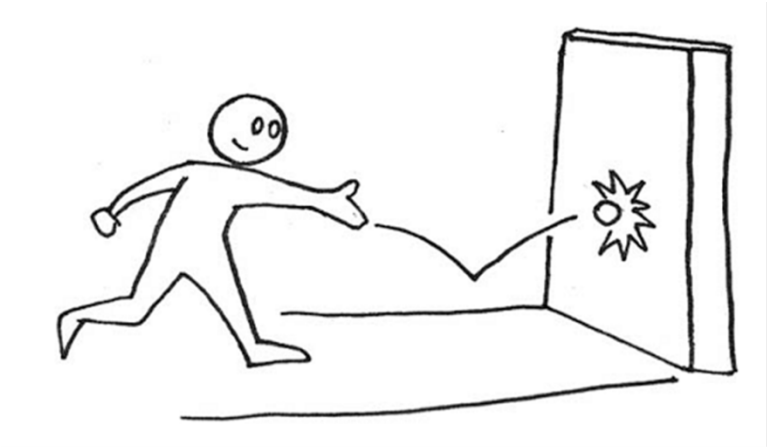
Jeu vidéo

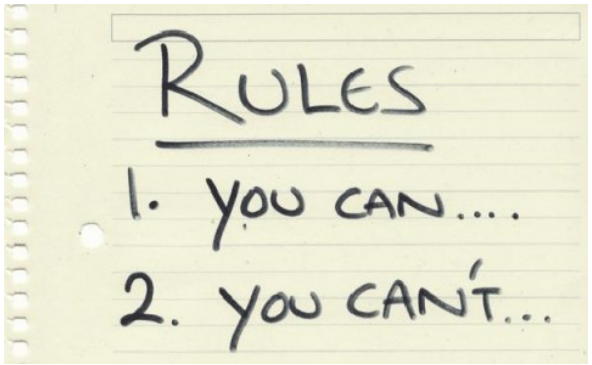
Est-ce un jeu vidéo ?



OpenGL

Et maintenant ?

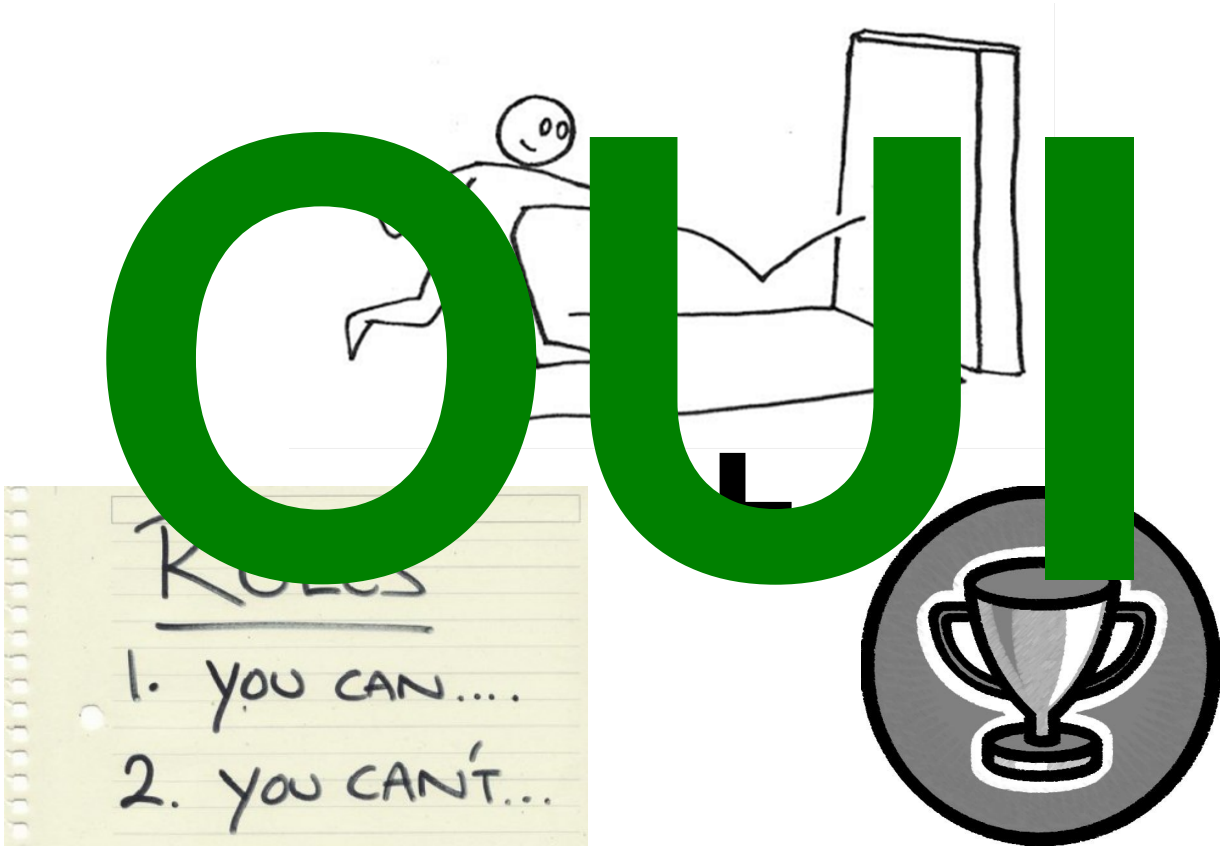




+

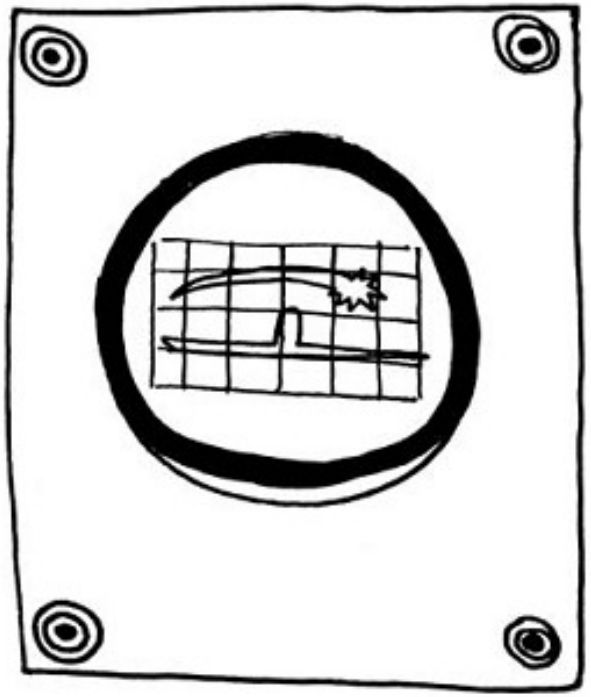
OpenGL

Et maintenant ?



Qu'est-ce qu'un jeu?

- Un vrai jeu
  - Nécessite au moins un joueur (ou plus)
  - Contient des règles
  - A des conditions de victoire et défaite



Tennis for Two

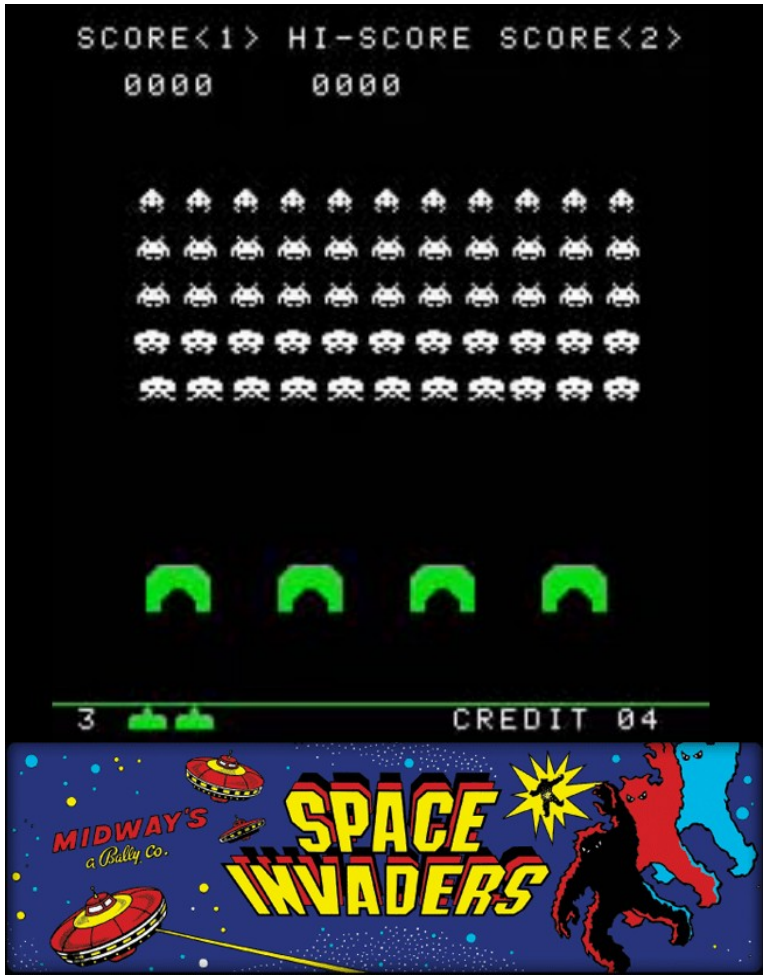
Faire un bon jeu …

- Très dur !
- Et long !
- Besoin de budget
- (Nouveaux jeux ont des budgets équivalents aux films hollywoodiens)

Simplified Sample Budget of 24-Month Project

Item	Cost x Number	Total Cost
Project		
Project Manager	U.S. $6,000 x 24	U.S. $144,000
Programming		
Lead Programmer	U.S. $4,000 x 24	U.S. $96,000
Game Programmer	U.S. $3,000 x 24	U.S. $72,000
Tool Programmer	U.S. $3,000 x 24	U.S. $72,000
Graphics and Artwork		
Level-Designer (Lead)	U.S. $2,500 x 24	U.S. $60,000
Level-Designer	U.S. $2,000 x 24	U.S. $48,000
Graphics Artist	U.S. $2,500 x 24	U.S. $60,000
Modeler	U.S. $2,000 x 24	U.S. $48,000
Licenses and Software		
Discreet "3D Studio Max 5"	U.S. $5,000 x 3	U.S. $15,000
id Software "Quake II Engine"	U.S. $10,000 x 1	U.S. $10,000
Adobe "Photoshop 7"	U.S. $1,200 x 1	U.S. $1,200
Rent and Equipment		
Office rent	U.S. $1.000 x 1	U.S. $24,000
Computer	U.S. $1.000 x 7	U.S. $7,000
Total		U.S. **$657,200**

De quoi a-t'on besoin ?



- Graphics
- Math
- Input handling
- Audio
- AI
- Physics
- Scripting
- Level Editor
- Network

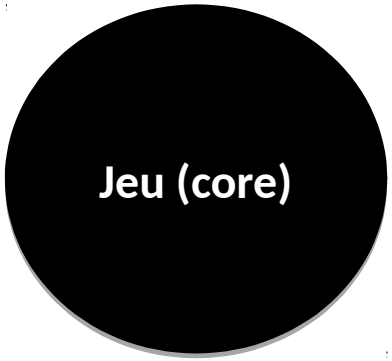
Moteur de jeu

- Un jeu peut impliquer la répétition de code.
- Il faut donc séparer les composants logiciels centraux de chaque tâche du moteur de jeu.
- Le but est de produire un logiciel extensible
- Pouvant être utilisé comme fondations de nombreux jeux sans modifications majeures.

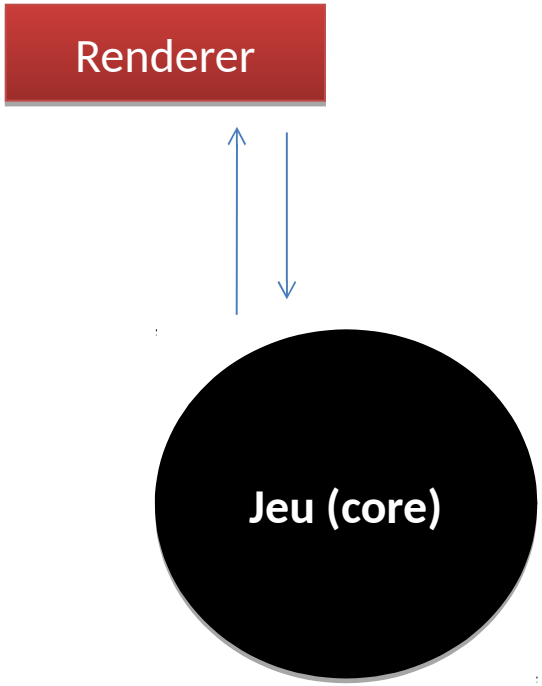
Sous-moteurs de jeu

- Un moteur de jeu contient généralement tout un ensemble de sous-moteurs
  - Graphics (2D/3D) engine
  - Input engine
  - Audio engine
  - AI engine
  - Physics engine
  - Network engine

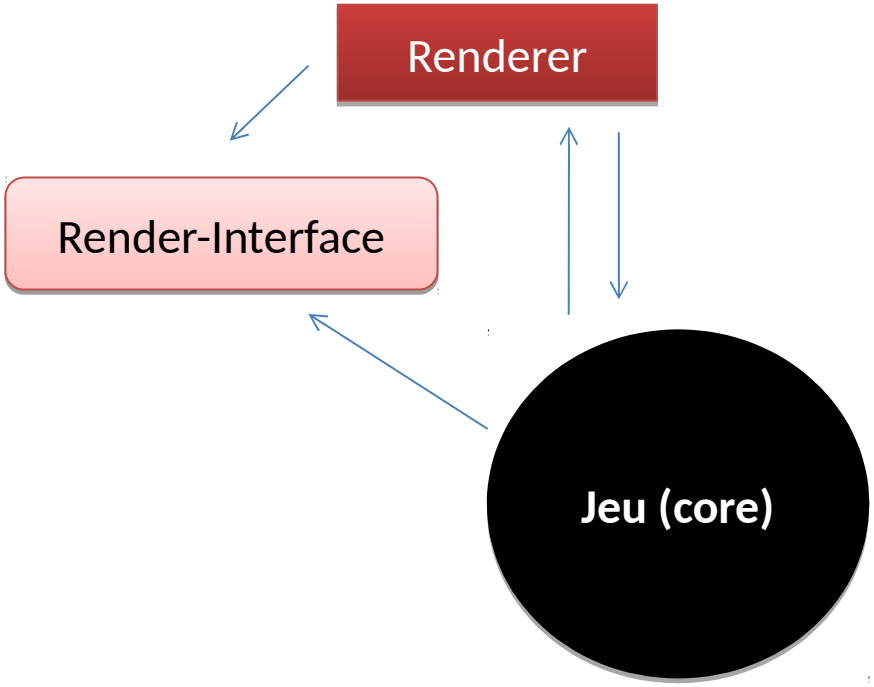
Structure d’un moteur de jeu



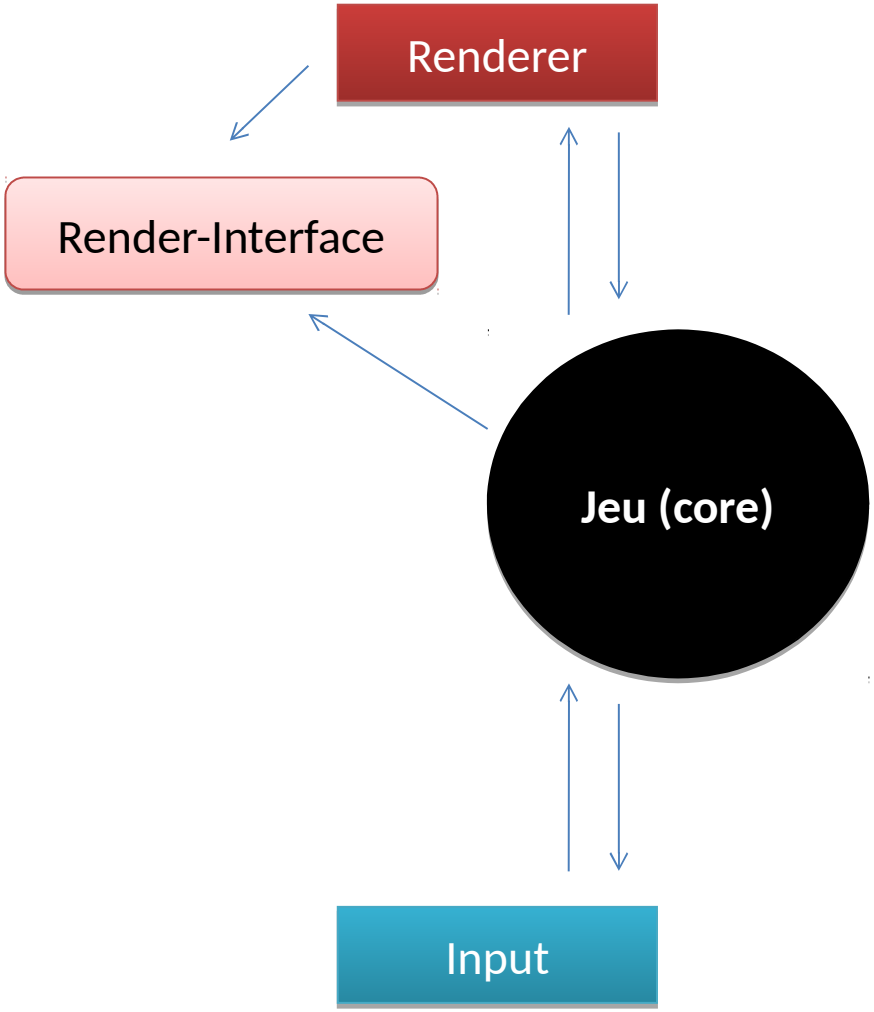
Structure d'un moteur de jeu



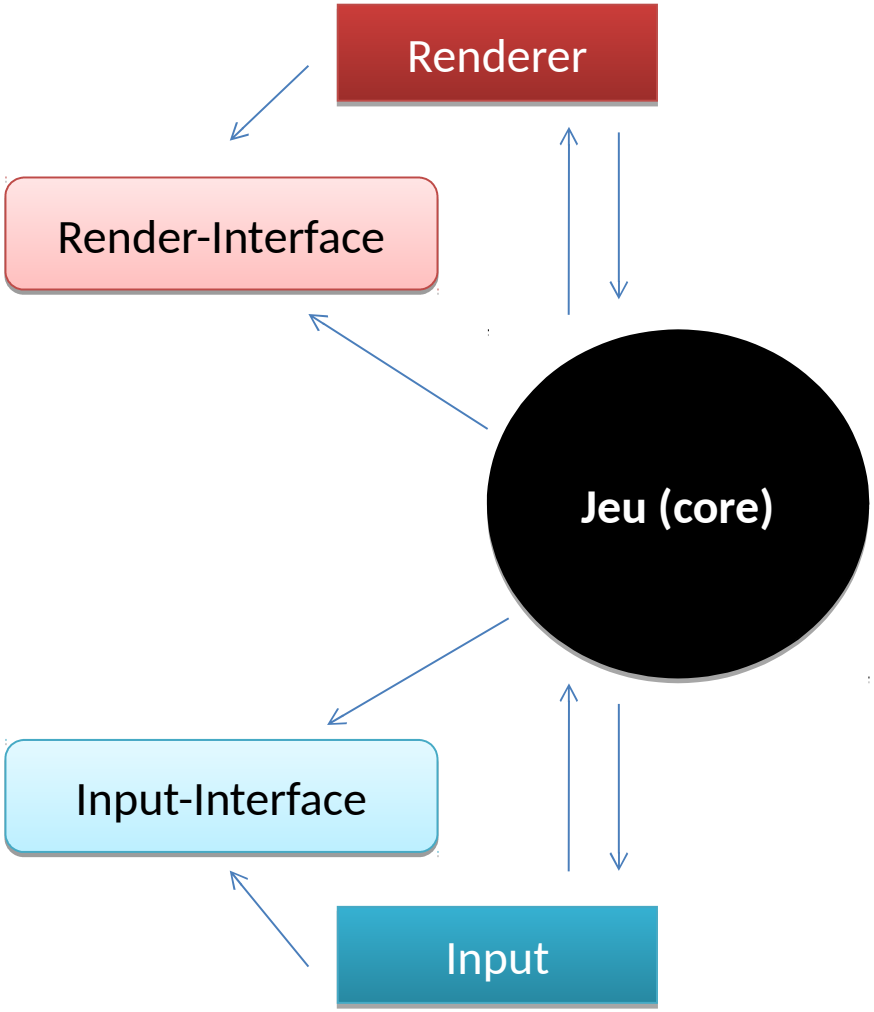
Structure d'un moteur de jeu



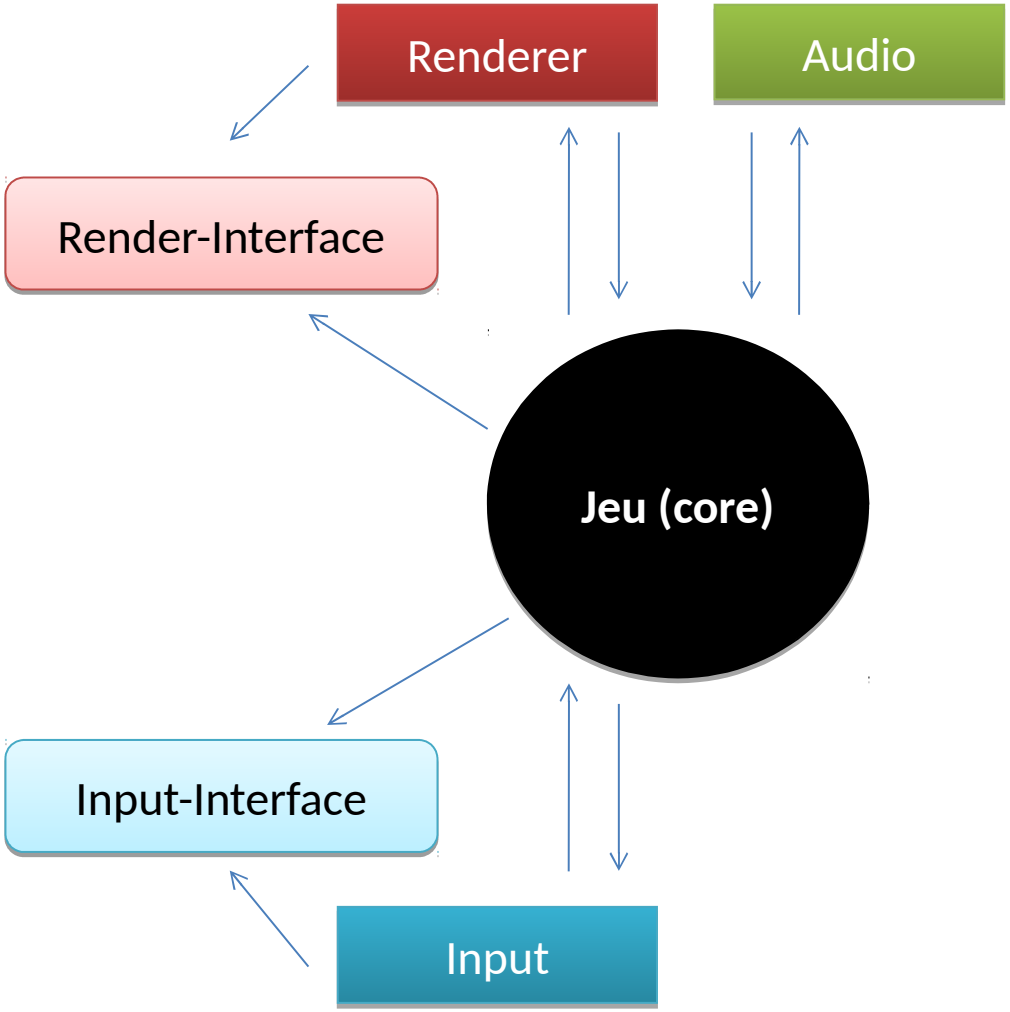
Structure d'un moteur de jeu



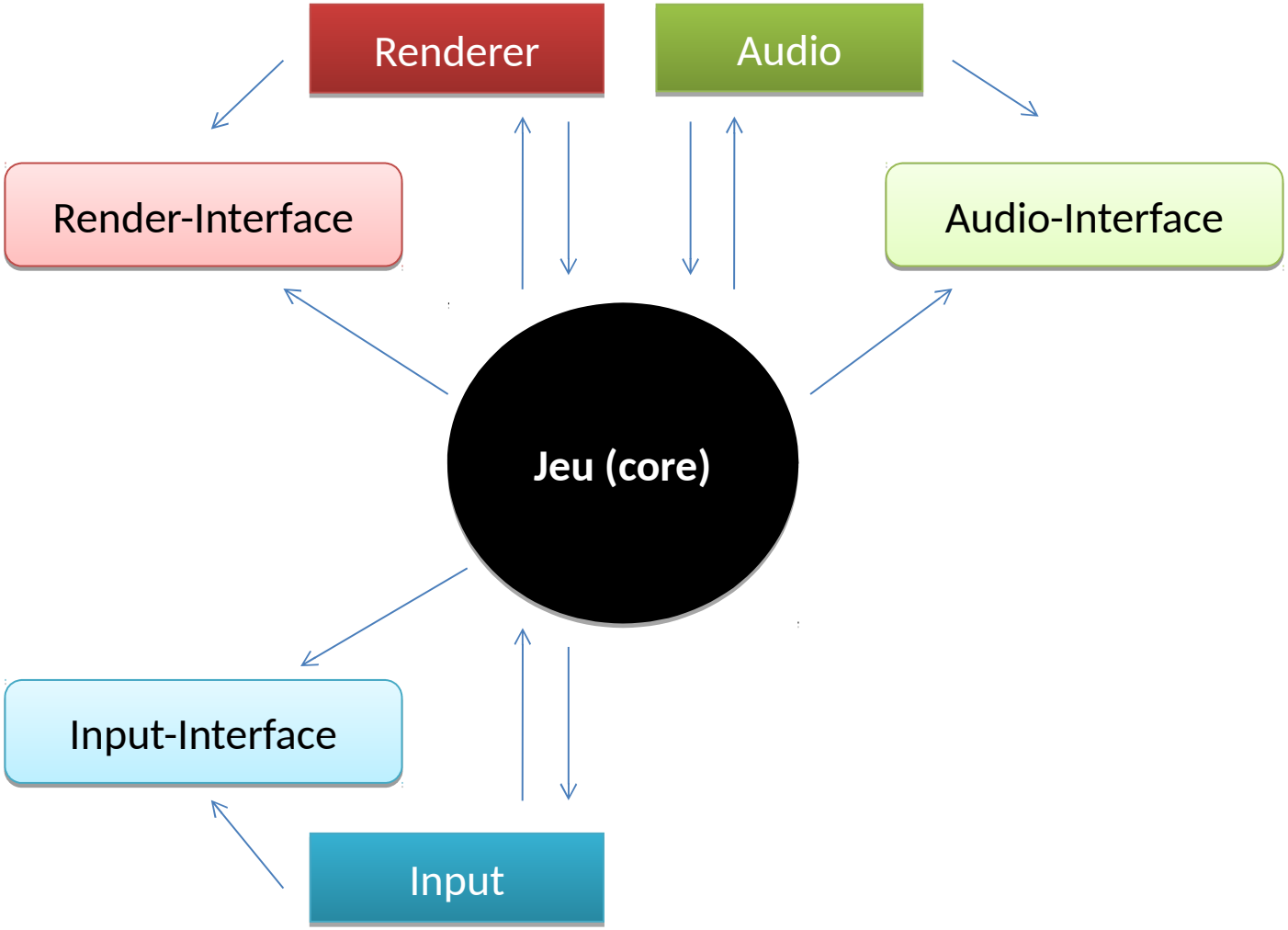
Structure d'un moteur de jeu



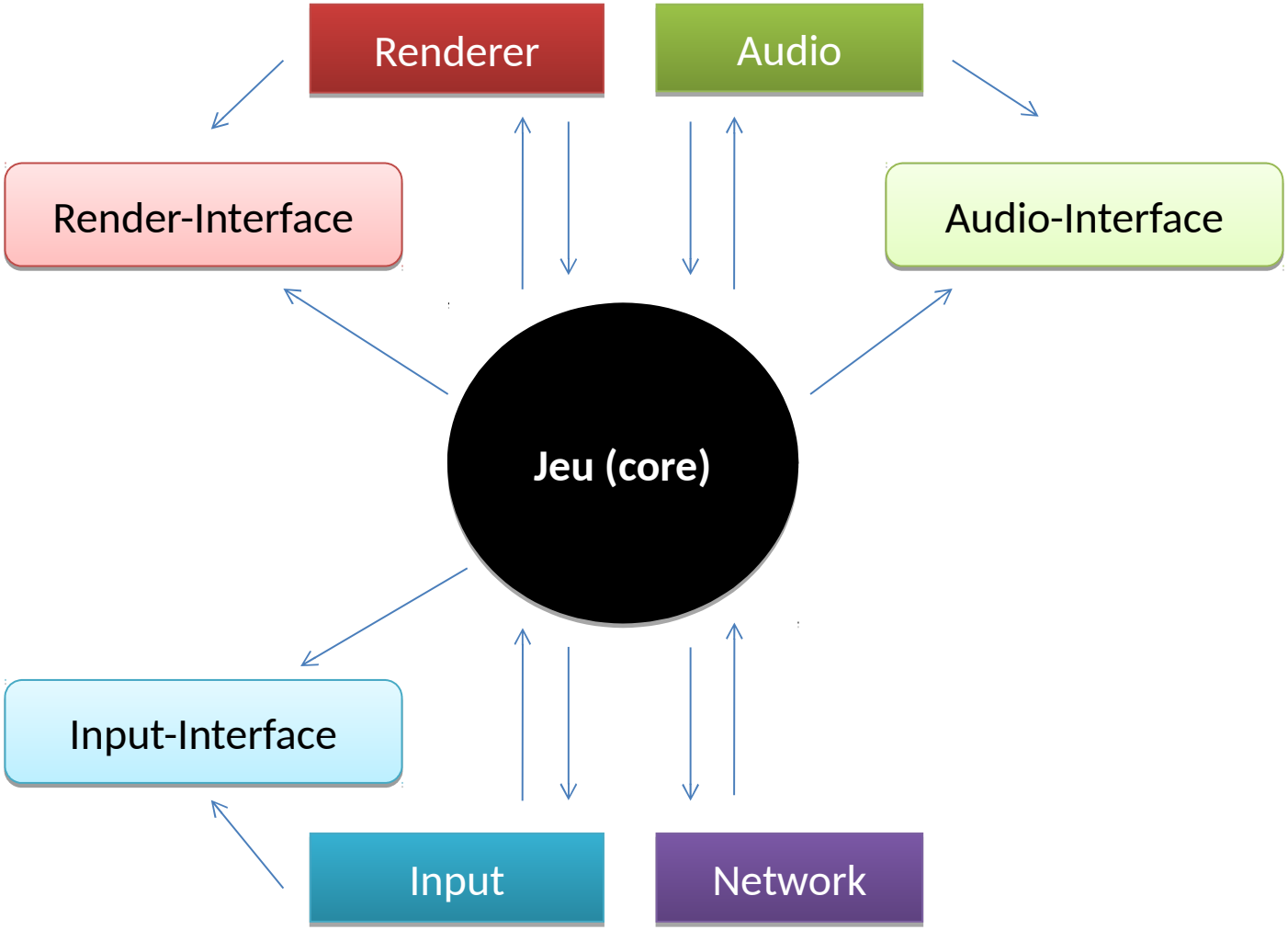
Structure d'un moteur de jeu



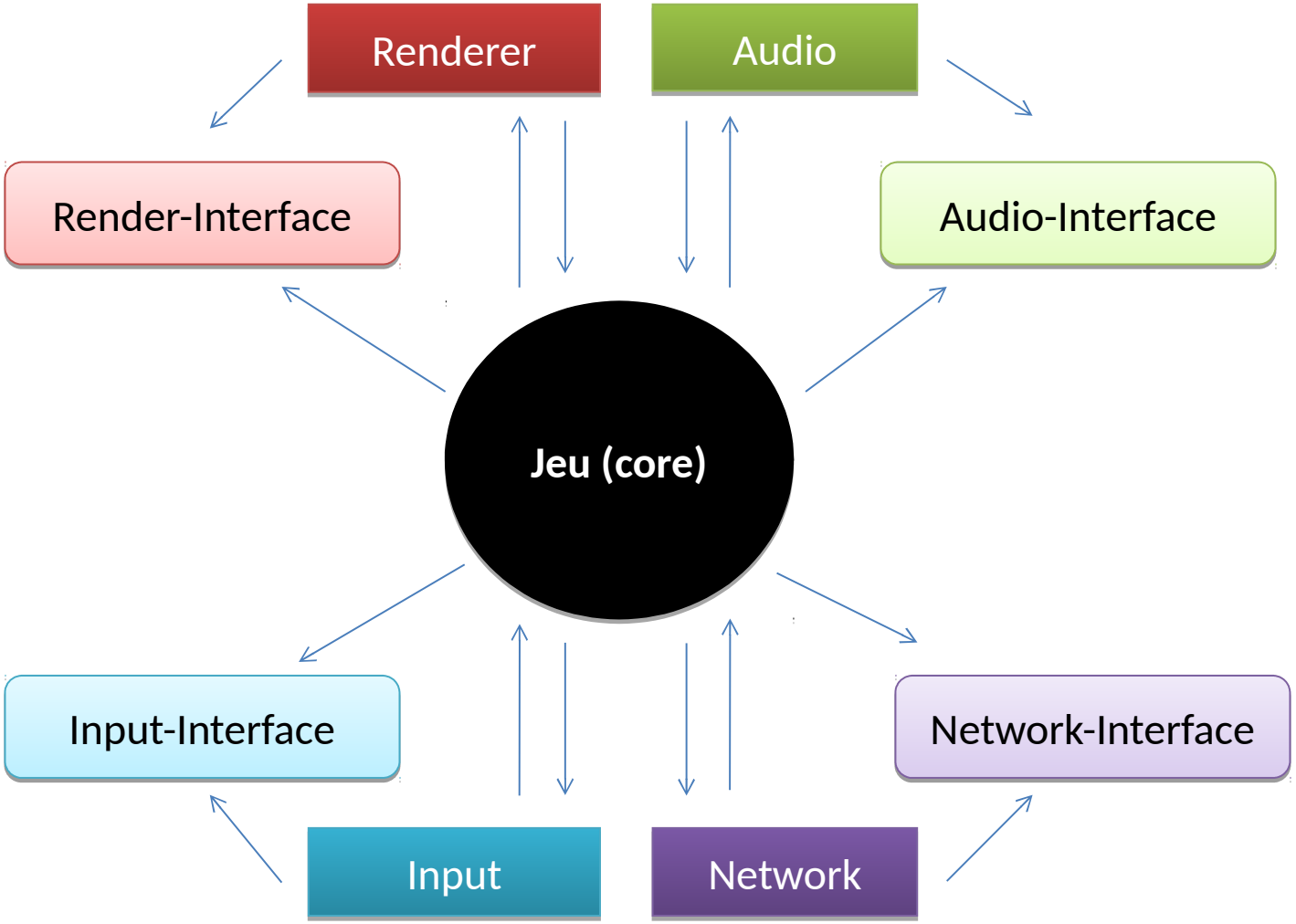
Structure d'un moteur de jeu



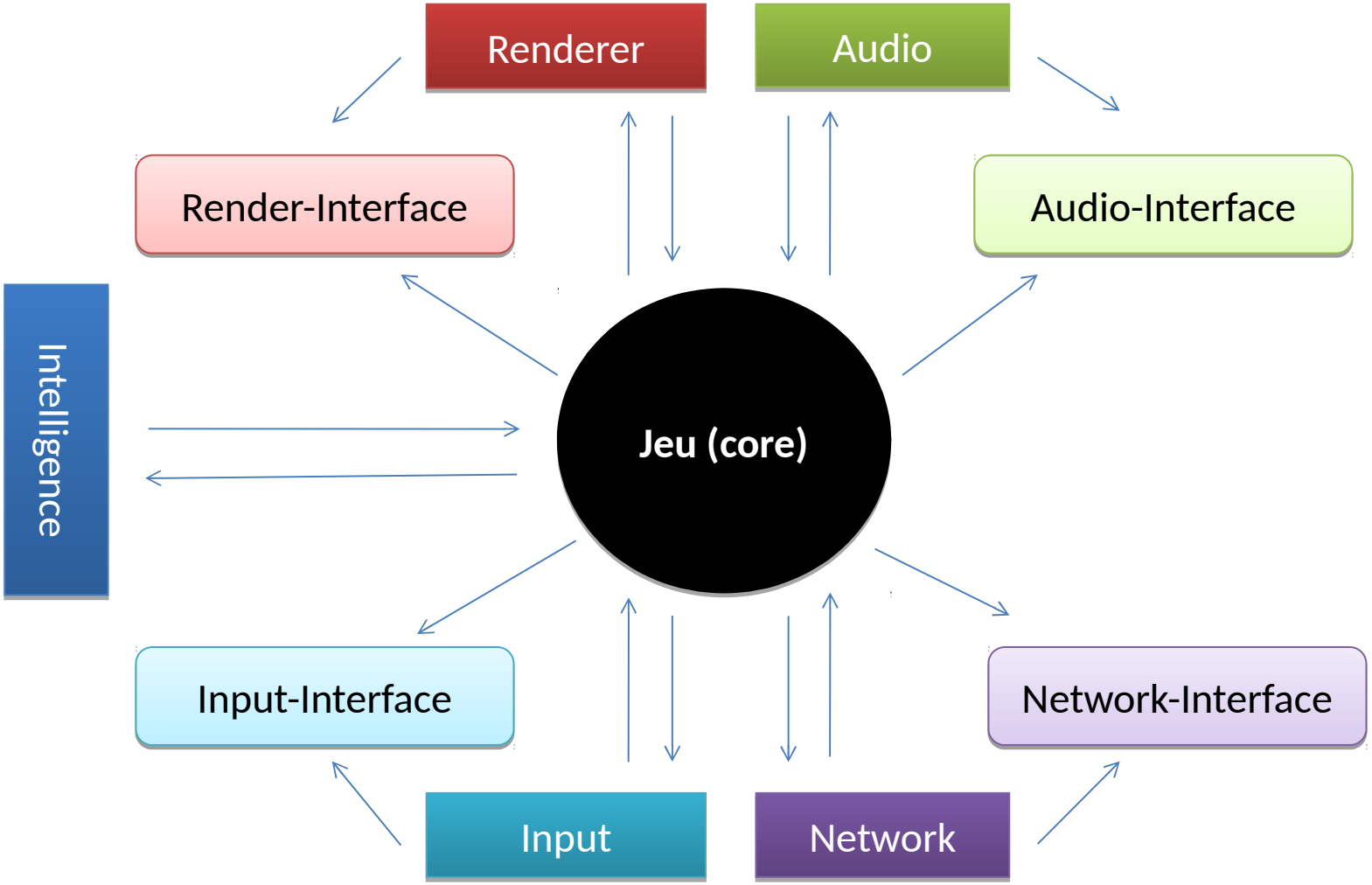
Structure d'un moteur de jeu



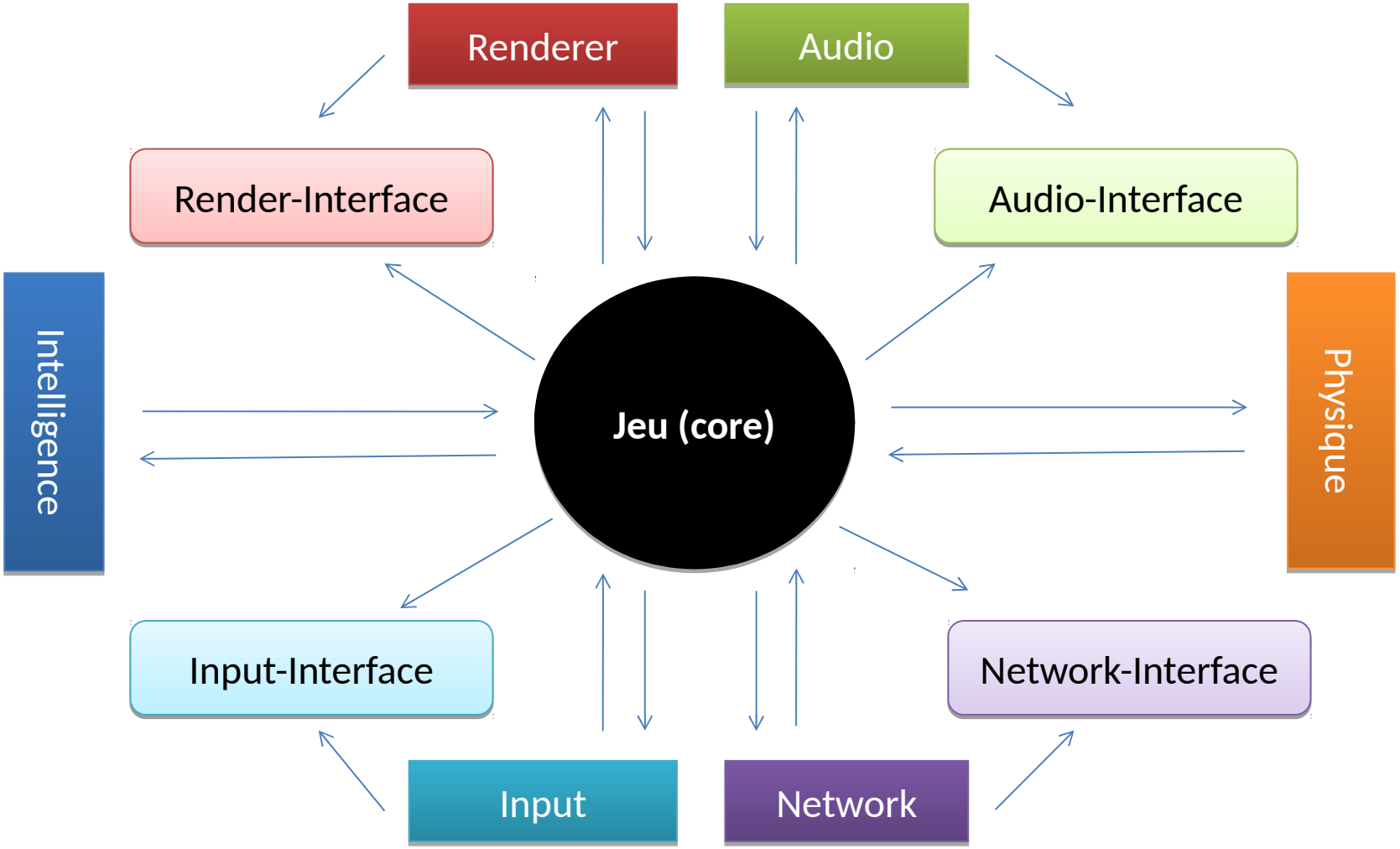
Structure d'un moteur de jeu



Structure d'un moteur de jeu



Structure d'un moteur de jeu



Moteurs de jeu spécifiques

- FPS game engine
- Platformers & third person game engine
- Fighting game engine
- Racing game engine
- RTS game engine
- MMO game engine

Une boucle de jeu