Plan

- Partie I: Introduction
  - De l'art graphique à la visualisation scientifique
  - De la donnée à l'image…
- Partie II: Fondements de l'imagerie 3D
  - Architecture des cartes
  - Principes de base d'OpenGl
- Partie III: Visualisation scientifique
  - Champs scalaires
  - Champs vectoriels

Visualisation

Visualisation 3D

La visualisation 3D :

- consiste à **calculer** une image numérique RVB à partir d'un modèle **géométrique 3D** et d'un modèle **d'éclairage** en utilisant un algorithme informatique.
- se caractérise par
  - **son style graphique (style photo-réaliste, cartoon, peinture…)**;
  - **sa vitesse de calcul (temps-réel, interactif ou batch)**;

Styles graphiques

On distingue en gros trois catégories de styles:

- Style « OpenGl » (Gouraud/Phong-shading)
- Photo-réaliste.
- Non-photo-réaliste (tout le reste)

Vitesse d'affichage

La vitesse d'affichage (appelée *frame-rate*) peut être:

- **Temps réel:** c'est la *carte graphique* qui fait l'essentiel;
- **Interactive**: les calculs sont partagés entre la carte graphique et le processeur central;
- **Off-line / batch**: c'est le (ou les) *processeur* qui fait (font) l'essentiel

La vitesse de calcul d'une image dépend du style graphique utilisé et de la complexité géométrique des scènes.

Architecture matérielle

Structure fonctionnelle

Vidéo
GPU — RAM
Carte graphique
Bus 16/32/64bits
CPU
RAM
Carte mère
Périphériques:
Disque dur, USB, carte son, etc.

- **CPU (Central Process Unit):** unité centrale (2-3 Ghz);
- **RAM (Random Access Memory)**: mémoire centrale (512Mo-2Go) ou dédiée au graphique (32Mo-256Mo);
- **GPU (Graphics Process Unit)**: processeur dédié aux calculs graphiques;

Evolution matérielle

- Avant 1980: **PC dispose juste d’une carte vidéo** (convertisseur analogique) – le processeur central remplit les pixels de la mémoire vidéo.
- Pendant 1980: **Rasterisation par la carte** (tracé de lignes, remplissage de polygones 2D);
- 1990-95: **cartes 3D pour stations graphiques** (SGI O2 contre Indigo 2) on introduit ici la fonctionnalité: **Transform & Lighting (T & L)**;
- 1995: **nouvelles fonctions** – textures, α-blending (Infinite Reality);
- 1998: **traitements géométriques pour PC** (T & L);
- 2000: **Performances PC** atteignent les cartes « high-end » (la 3D devient standard sur PC);
- 2001: **PC introduisent des fonctionnalités nouvelles** (multi-textures – combinaison de registres, **pixel et vertex programs**);
- 2002: **Langage de programmation des cartes** (OpenGl 2.0 – GLSL , NVIDIA Cg, DX9);

GPU >100 Mio. Transistors, 8 pipes, 16 unités de textures.

Carte graphique

Ce qui caractérise une carte graphique :

- **La fréquence d’affichage:** la résolution maximale des images en mode « true-color » à une fréquence donnée (60Hz p.ex.);
- **La profondeur des couleurs:** nombre de bits par composante R,V et B (8 ou 12 bits).
- **La vitesse de rasterization:** nombre de points, lignes ou triangles que l’on peut afficher par seconde (>1Million);
- **Le nombre de fonctionnalités 3D prises en charge**: programmabilité, nombre d’unités de textures, fonctionnalités supportées (ombres, convolution, compression, etc.);
- **Le nombre de fonctionnalités vidéo**: MPEG2, MPEG4, sortie stéréo, etc.
- **La taille de sa mémoire dédiée** (256Mo);

Composants carte graphique

graphics
CPU
network
performance
time

NVIDIA NV30 (90-120M)
ATI R300 (107M)
NVIDIA GeForce4 (63M)
NVIDIA GeForce3 (57M)
ATI Radeon 8500 (60M)
Riva 128 (3M)
transistors (Mio)
9/97 3/98 9/98 3/99 9/99 3/00 9/00 3/01 9/01 3/02
time (month/year)

Fonctionnalités

scène – transformations objet et perspective – Lumière, clipping, projection – Shaders, textures, occlusions – image

traitement géométriques **rasterisation**

CPU → Driver → GPU Vidéo

Partie II: Fondements de l'imagerie 3D

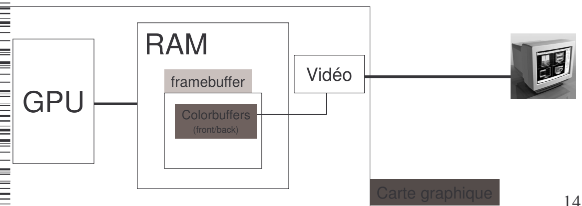
Rasterization

13

Partie II: Fondements de l'imagerie 3D

Frame-buffer

- La mémoire de la carte graphique contient un espace appelé le **frame-buffer**, qui contient lui-même une zone appelée **color-buffer**.
- Le color buffer est une matrice de pixels RVB.
- Le color-buffer est lu par un composant vidéo de la carte pour convertir la matrice de pixels RVB en signal vidéo.
- Pour définir une image il suffit donc de modifier les valeurs du color-buffer.

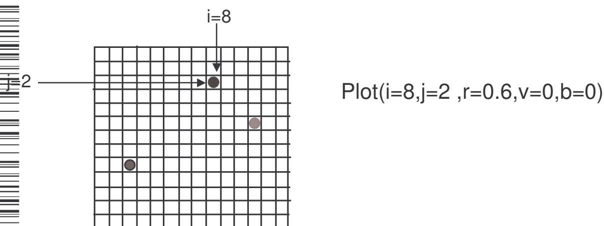


14

Partie II: Fondements de l'imagerie 3D

Plot

- La procédure la plus simple pour faire un dessin consiste à utiliser **PLOT**.
- Cette fonction permet de définir la couleur RVB d'un pixel de coordonnées (i,j).



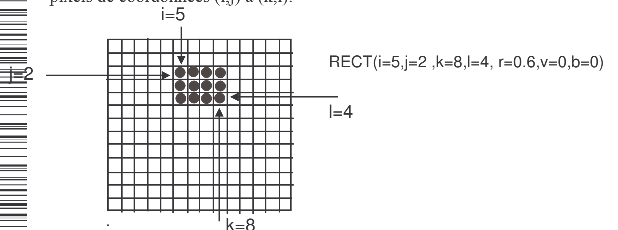
La CPU envoie la commande PLOT au driver qui traduit à la carte, qui modifie le pixel correspondant du color-buffer

15

Partie II: Fondements de l'imagerie 3D

Rectangle

- Une autre fonction très simple consiste à dessiner des rectangles.
- Une fonction RECT permet de définir la couleur RVB d'un rectangle de pixels de coordonnées (i,j) à (k,l).



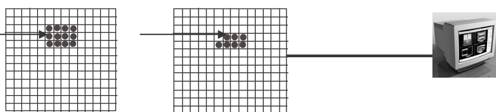
La CPU envoie la commande RECT au driver qui traduit à la carte, qui modifie les pixels ligne par ligne.

16

Partie II: Fondements de l'imagerie 3D

Double-buffering

- Le convertisseur vidéo et la GPU accèdent simultanément au frame-buffer
- De ce fait, il se peut que les images **scintillent** à l'écran. Pour éviter cet effet on introduit la technique du **double-buffering**.



Position du convertisseur

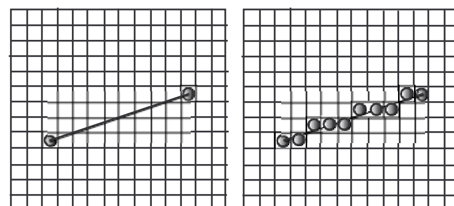
La GPU dessine dans le color-buffer BACK alors que le convertisseur lit dans le color-buffer FRONT. Une fois les dessins finis les buffers sont échangés.

17

Partie II: Fondements de l'imagerie 3D

Tracer des segments

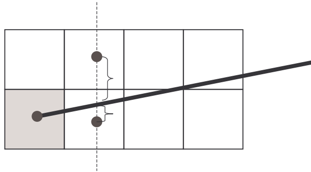
- Un des tous premiers algorithmes graphique intégré dans les cartes a été celui du **tracer de segment**:
- Comment joindre deux pixels pour donner l'illusion d'une droite?
- Le commande envoyée par la CPU est LIGNE.



18

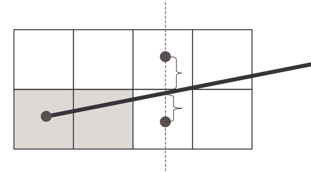
Algorithme de Bresenham

- L'idée consiste à minimiser l'erreur avec une ligne continue qui joindrait les centres des deux pixels.
- Quand on se limite au **premier octant**, on constate qu'il suffit de parcourir horizontalement pixel par pixel et de décider si l'on monte d'un pixel ou non.



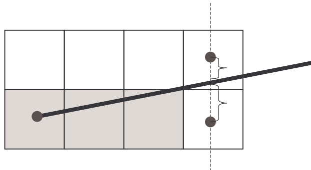
Algorithme de Bresenham

- L'idée consiste à minimiser l'erreur avec une ligne continue qui joindrait les centres des deux pixels.
- Quand on se limite au **premier octant**, on constate qu'il suffit de parcourir horizontalement pixel par pixel et de décider si l'on monte d'un pixel ou non.



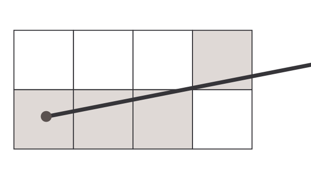
Algorithme de Bresenham

- L'idée consiste à minimiser l'erreur avec une ligne continue qui joindrait les centres des deux pixels.
- Quand on se limite au **premier octant**, on constate qu'il suffit de parcourir horizontalement pixel par pixel et de décider si l'on monte d'un pixel ou non.



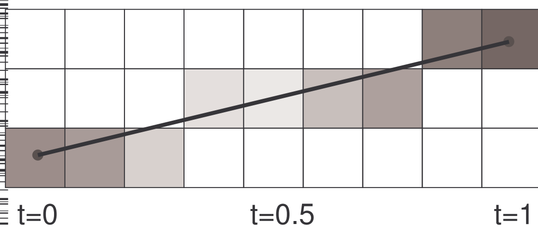
Algorithme de Bresenham

- L'idée consiste à minimiser l'erreur avec une ligne continue qui joindrait les centres des deux pixels.
- Quand on se limite au **premier octant**, on constate qu'il suffit de parcourir horizontalement pixel par pixel et de décider si l'on monte d'un pixel ou non.



Interpolation de couleurs

- La couleur n'est pas obligatoirement constante le long de la droite.
- On peut définir deux couleurs différentes aux extrémités et interpoler linéairement ces couleurs: $\mathbf{C=tC_2+(1-t)C_1}$

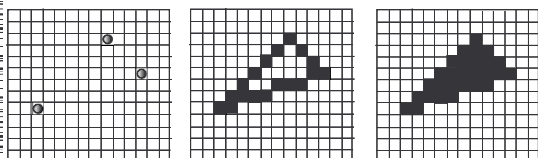


Remplissage

Le second algorithme graphique intégré aux cartes a consisté à remplir les pixels à l'intérieur d'un **polygone**.

On sait déjà tracer les arêtes d'un polygone en utilisant l'algorithme précédent.

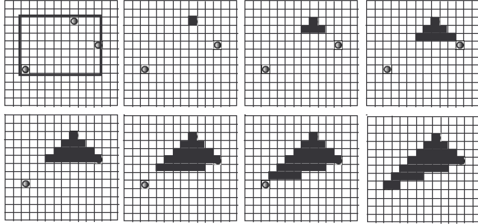
Mais comment déterminer quels sont les pixels intérieurs?



Scan-line

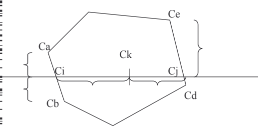
Idée: utiliser la propriété des polygones, qui dit que toute demi-droite issue d'un point intérieur au polygone a un nombre impair d'intersections.

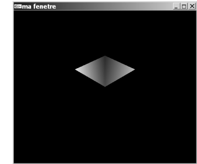
On place le polygone dans une boite rectangulaire, puis on parcourt ligne par ligne cette boite



Interpolation des couleurs

On interpole les couleurs **bi-linéairement** (comme pour une droite)**:**





Transform

Tracés 3D

Jusqu' aux années 80 les tracés (PLOT, RECT et LIGNE) sont purement **2D**.

Avec le développement de la CAO, des simulateurs, etc. une demande croissante de la 3D a vu le jour.

Il a fallu introduire de nouvelles fonctionnalités dans les cartes :

- **Transformation perspective:** l'œil humain est un capteur optique de type lentille qui déforme l'image selon une transformation **perspective**. Deux lignes parallèles se rencontrent en un point que l'on nomme le **point de fuite**.
- **Clipping:** une fois les points et les lignes 3D projetés sur le plan de l'écran, il faut déterminer si ces points sont à l'extérieur de l'écran ou à l'intérieur de l'écran.

Projection perspective

La projection se fait en définissant une matrice de transformation perspective en coordonnées homogènes 4x4:

$$(x,y,z,1)\cdot\begin{pmatrix} m_{11} m_{21} m_{31} m_{41} \\ m_{12} m_{22} m_{32} m_{42} \\ m_{13} m_{23} m_{33} m_{43} \\ m_{14} m_{24} m_{34} m_{44} \end{pmatrix} = \begin{pmatrix} u \\ v \\ r \\ s \end{pmatrix}$$

Ce calcul revient à faire 4 fois un produit scalaire.

Les cartes graphiques optimisent ce calcul par le biais du parallélisme et de la gestion de pipelines.

Tracé de polyèdres

Pour dessiner un polyèdre en *mode filaire*, on trace tous les polygones du polyèdre en mode filaire.

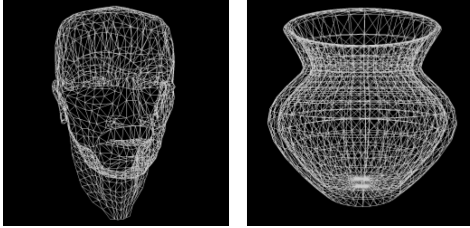
Un polygone est tracé en mode filaire en dessinant toutes ses arêtes rectilignes.

Pour faire un tracé de polyèdre, la carte propose deux commandes:

- une commande PERSPECTIVE: elle permet de définir la perspective (la matrice en fonction des paramètres de projection)
- Une commande POLYGONE: elle permet de tracer un polygone en donnant les coordonnées 3D des sommets.

Partie II: Fondements de l'imagerie 3D

Tracé de polyèdres



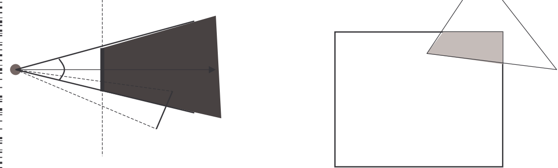
La CPU envoie les commandes PERSPECTIVE et POLYGONE au driver qui traduit à la carte, qui applique la transformation aux sommets puis dessine les lignes concernés.

31

Partie II: Fondements de l'imagerie 3D

Clipping

Selon la taille de l'écran certains polygones peuvent être partiellement ou complètement à l'extérieur du champ de vision (*view frustum*):



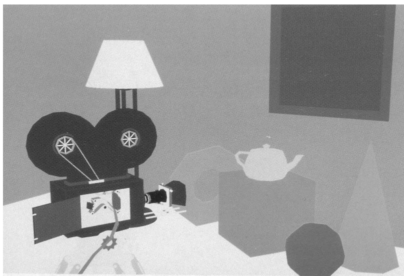
- La carte graphique élimine dans le polygone les parties à l'extérieur du rectangle de l'écran: c'est le **clipping**.
- La carte remplace un polygone par un autre (ici un triangle par un quadrilatère)

32

Partie II: Fondements de l'imagerie 3D

Remplissage

Au lieu de tracer uniquement les arêtes du polyèdre, on peut aussi remplir les polygones en utilisant le principe du scan-line:



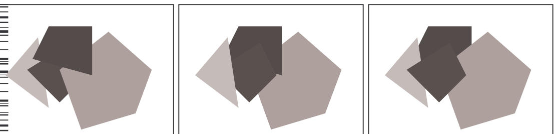
33

Partie II: Fondements de l'imagerie 3D

Elimination des parties cachées

En affichant tous les polygones d'un ou plusieurs polyèdres et en les remplissant, certains risquent d'en recouvrir d'autres.

Le résultat de l'image ne sera donc pas le même selon l'ordre avec lequel on aura affiché ces polygones:



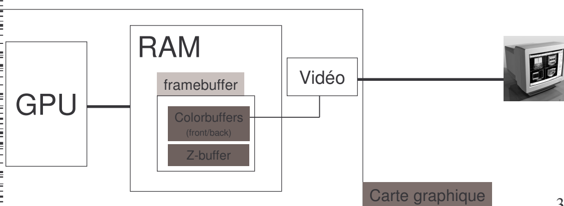
34

Partie II: Fondements de l'imagerie 3D

Z-Buffer

Principe: ajouter dans le frame-buffer, à coté du color-buffer, un autre buffer qui s'appelle le **Z-buffer (tampon de profondeur)**.

Le Z-buffer contient à tout moment et pour chaque pixel la profondeur Z la plus petite (donc le point le plus proche).



35

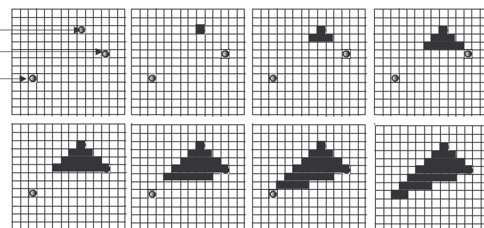
Partie II: Fondements de l'imagerie 3D

Algorithme du Z-Buffer

Initialement le Z-buffer contient l'infini.

Lors du remplissage, on interpole le Z (coordonnée *r*) et on compare:

- Si le Z du buffer est plus petit on ne fait rien
- Si le Z du buffer est plus grand, on place la couleur dans le color-buffer et on remplace le Z par la coordonnée *r*.

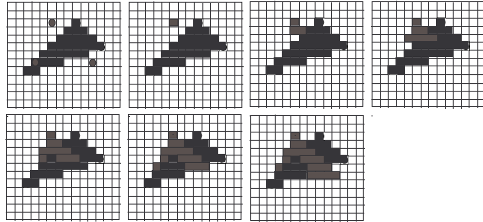


36

Partie II: Fondements de l'imagerie 3D

Algorithme du Z-Buffer

Initialement le Z-buffer contient l'infini.
Lors du remplissage, on interpole le Z (coordonnée *r*) et on compare:
- Si le Z du buffer est plus petit on ne fait rien
- Si le Z du buffer est plus grand, on place la couleur dans le color-buffer et on remplace le Z par la coordonnée *r*.



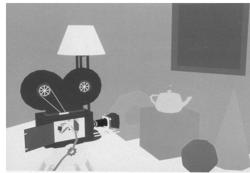
Partie II: Fondements de l'imagerie 3D

Lighting

Partie II: Fondements de l'imagerie 3D

Tracés 3D « en couleur »

Avec un remplissage de couleur uniforme, l'affichage paraît « plat ».
Il faut introduire une notion d'éclairage pour faire varier l'intensité sur les polygones.

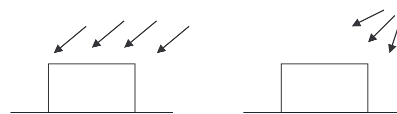


Ceci implique:
- **Définir des sources de lumière:** les sources permettent de calculer l'intensité lumineuse qui atteint les polygones.
- **Définir une réflectance:** la réflectance permet de caractériser la manière dont les polygones restituent la lumière reçue.

Partie II: Fondements de l'imagerie 3D

Types de sources

On peut définir essentiellement deux types de sources lumineuses:
- **Source directionnelle:** la source est infiniment loin et la lumière arrive dans une direction donnée. Cette direction est définie par un **vecteur**.
- **Source ponctuelle:** la source est une petite ampoule qui éclaire dans toutes les directions possibles. La source est définie par un **point**.

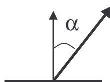


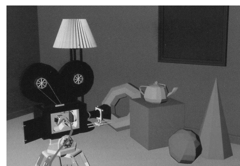
On peut également définir un **spot** qui est une combinaison des deux.

Partie II: Fondements de l'imagerie 3D

Calcul de la couleur

La couleur du polygone se calcule de la façon suivante:
- Calculer la normale du polygone
- Faire le produit scalaire avec la direction de la lumière (valeur entre 0 et 1)
- Multiplier ce résultat par la couleur de l'objet.





Plus le polygone est « face » à la source, plus il reçoit de lumière (le produit scalaire vaut 1).
Plus le polygone est rasant par rapport à la source, moins il reçoit de lumière (le produit scalaire vaut 0).

Partie II: Fondements de l'imagerie 3D

GPU / CPU

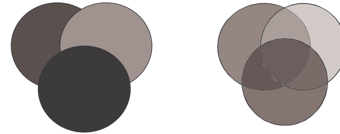
Pour faire ce type de tracés 3D, le processeur central (CPU) fournit les informations suivantes à la GPU :
- Les sommets des polygones et en chaque sommet une normale + une couleur;
- La position / direction de la source de lumière, ainsi que sa couleur (intensité);

La GPU effectue les opérations suivantes :
- Transformation perspective des sommets et clipping;
- Calcule de la couleur à afficher en chaque sommet en fonction de la source de lumière (par produit scalaire);
- Remplissage du polygone avec interpolation bilinéaire de la couleur;
- Élimination des parties cachées lors du remplissage.

Paramétrisation des affichages

Transparence

- Avec le Z-buffer les objets sont toujours supposés **opaques**.
- Or pour certaines applications, il peut être intéressant de rendre des surfaces données transparentes.
- Problème: un objet transparent ne cache pas les autres objets, mais combine sa couleur avec celle de l’objet juste derrière:

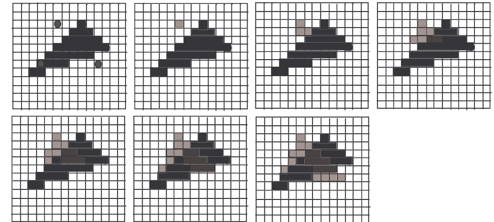


α-blending

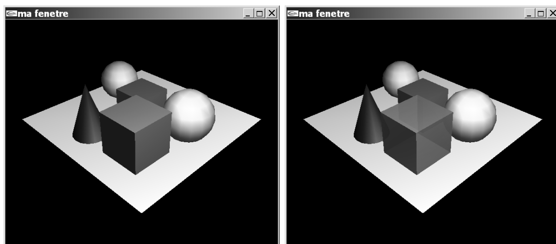
- On ajoute une 4ème composante de couleur à chaque sommet du polygone: **la composante α**, qui représente un coefficient d’opacité.
- La couleur devient un quadruplet RVBA.
- On affiche d’abord les polygones qui sont complètement opaques (non-transparents).
- Puis, on affiche les polygones transparents, mais pas en remplaçant la couleur du frame-buffer, en la **combinant**:

$$C_f = tC_b+(1-t)C_s$$

α-blending

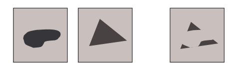


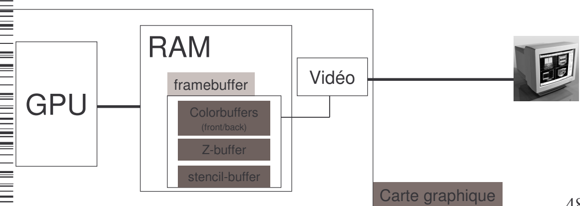
α-blending



Affichage conditionnel

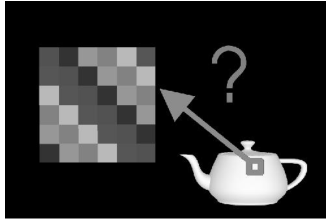
- **Principe:** ajouter dans le frame-buffer, un buffer qui s’appelle le **stencil-buffer**.
- Ce buffer permet de moduler l’affichage.





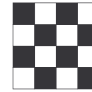
Associer des paramètres aux surfaces

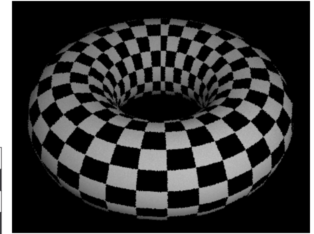
- Associer à chaque point (x,y,z) d'une surface 3D un point (i,j) du carré unité $[0,1]^2$.
- Ceci revient à définir une **application** de $\Re^3$ vers $[0,1]^2$ (x,y,z) → (i,j)
- Cette application s'appelle le **placage (ou mapping)**
- Le mapping permet d'associer **une fonction F(i,j)** aux surfaces.
- Cette fonction discrète se nomme une **texture**.



Modulation de la couleur

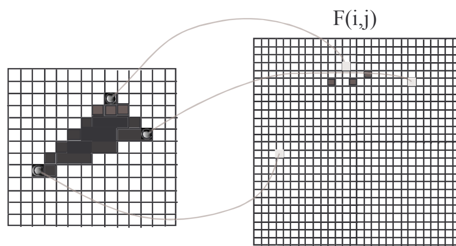
- **F(i,j)** est définie de manière discrète, donc sous la forme d'une matrice (ou image).
- Initialement, le mapping servait à **moduler la couleur** de la surface, sans introduire de nouvelles primitives géométriques.





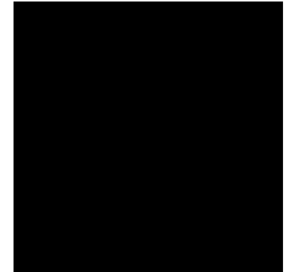
Rasterization revisitée

- Pour pouvoir utiliser une texture, il faut associer à chaque sommet du polyèdre des coordonnées de texture (u,v).



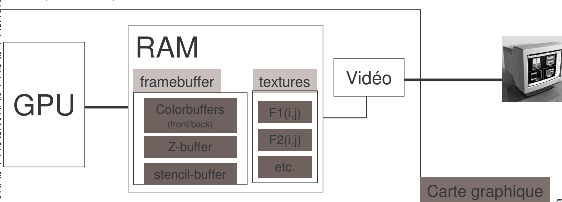
Modulation de la normale

- Tout paramètre peut être modulé par le biais de F(i,j), y compris la normale, par exemple, pour donner l'illusion de détails géométriques et ainsi accélérer les affichages.



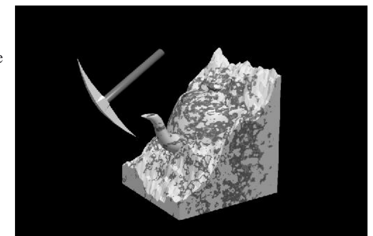
Mémoire de texture

- F(i,j) doit être accessible à la carte graphique.
- Pour augmenter la vitesse d'accès, F(i,j) est donc directement chargée en mémoire dédiée.
- Le nombre d'unités de textures (de fonctions $F_k(i,j)$) accessibles est limité par la carte (16 unités).



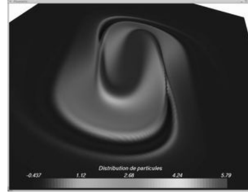
Textures 3D

- Avec l'augmentation de la mémoire RAM des cartes, il est devenu possible d'associer une fonction à 3 dimensions à chaque sommet (x,y,z).
- F(i,j,k) est définie par un ensemble discret de voxels 3D, c'est donc un tableau à trois dimensions (un empilement d'images).
- Le problème qui se pose est un problème de mémoire: une fonction F(i,j,k) de résolution 1024^3 requiert 1Go de mémoire!



Utiliser la texture pour visualiser des données scalaires

- Les fonctions F(i,j) ou F(i,j,k) permettent de visualiser des données scalaires sur les surfaces, sans introduire un nombre trop important de primitives.
- Pour cela on utilise une **fonction de transfert** F_t associant à un scalaire un triplet RVB de couleur:
- F_t : [0,1] → {RVB}
- La fonction de transfert est elle même une texture 1D (un tableau).
- La carte graphique doit pouvoir faire deux accès de texture par pixel rasterizé.



GPU / CPU

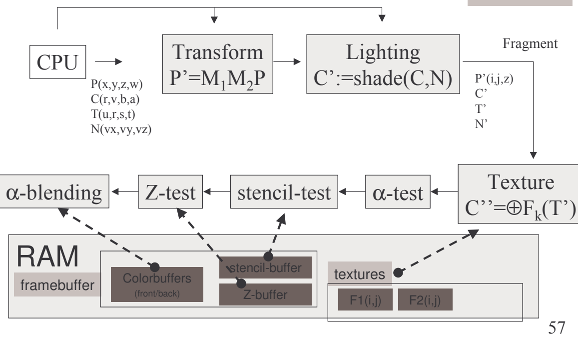
Pour faire des tracés 3D, le processeur central (CPU) fournit les informations suivantes à la GPU :

- La matrice de projection perspective et la matrice de positionnement des sommets.
- La position / direction des sources de lumière (jusqu'à 8), ainsi que leur couleur (intensité);
- Les textures 1D, 2D et 3D;
- Le contenu du stencil buffer pour moduler les affichages;
- Finalement, les sommets des polygones et en chaque sommet : une normale + une couleur + des coordonnées de textures;

La GPU effectue les opérations suivantes :

- Transformation perspective et positionnement des sommets, clipping;
- Calcule de la couleur à afficher en chaque sommet en fonction de la source de lumière (par produit scalaire);
- Remplissage du polygone avec accès aux textures pour moduler la couleur, la normale, etc.;
- Test du stencil buffer;
- Élimination des parties cachées lors du remplissage.

Chaîne graphique



OpenGl

API 3D

OpenGl (Open Graphics Library) est une API 3D (application programming interface);

OpenGl regroupe un ensemble de fonctionnalités pour faire des affichages 2D et 3D.

Un programme OpenGl est structuré sous la forme d'une suite de procédures appelées successivement (en utilisant une notation de type grammaire):

$$\text{Prog} = (\Im^{+} \Lambda^{*} \wp^{*})^{+}$$

$\Im$: ensemble de procédure d'initialisation de données globales
Λ : ensemble de procédures de transfert de données
$\wp$: ensemble de tracés de primitives graphiques

Initialisation

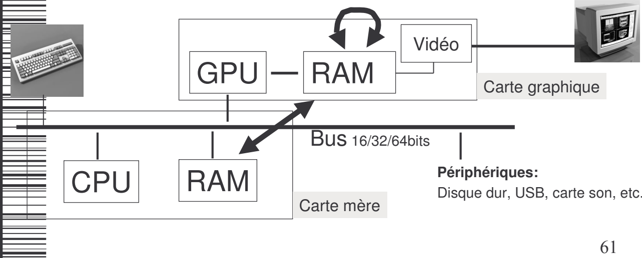
L'initialisation consiste à définir certains paramètres d'affichage globaux:

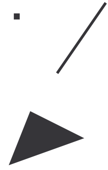
- Les matrices de transformation:
  - void glLoadMatrixf(float m[16]) ;
- Les sources de lumière:
  - glLightfv(GL_LIGHTi, GL_DIFFUSE, float *ptr);
- Les tests (alpha, Z, stencil, etc.):
  - glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);

Transferts

Les procédures de transferts permettent de déplacer en blocs des données depuis la RAM CPU vers la RAM GPU (et vis versa) ainsi qu'au sein même des buffers GPU ;

Pendant le transfert les données peuvent être **filtrées, modulées ou transcodées**.





Périphériques:
Disque dur, USB, carte son, etc.

Primitives graphiques

OpenGl ne connaît en fait que trois types de primitives graphiques:

- point, segment et triangle

Ces trois primitives utilisent toujours des sommets 3D:

```
for (i=0.0; i<2.0*PI; i+=da)
    {
    glBegin(GL_TRIANGLES);
        glColor3f(1.0,1.0,1.0);
        glTexCoord2f((i)/(2.0*PI), 1.0);
        glNormal3f(cos(i)*cos(beta),sin(i)*cos(beta), sin(beta));
        glVertex3f(x+r*cos(i),y+r*sin(i), z);
        glColor3f(1.0,1.0,1.0);
        glTexCoord2f((i+da)/(2.0*PI), 1.0);
        glNormal3f(cos(i+da)*cos(beta),sin(i+da)*cos(beta), sin(beta));
        glVertex3f(x+r*cos(i+da),y+r*sin(i+da), z);
        glColor3f(1.0,1.0,1.0);
        glNormal3f(0.0, 0.0, 1.0);
        glTexCoord2f(0.5, 0.0);
        glVertex3f(x, y, z+d);
    glEnd();
    }
```


Traitements prédéfinis

Points
Segments
Triangles

Traitement par vertex

transformation perspective éclairage par vertex, clipping, Interpolation linéaire

Vertex
Normal
Couleurs
Coord. textures

Traitement par fragment

Accès aux textures, Z-test, alpha-test, alpha blending, stencil-test

Vertex
Normal
Couleurs
Coord. textures

image

Traitements semi-programmables

Points
Segments
Triangles

Traitement par vertex

transformation perspective éclairage par vertex, clipping, Interpolation linéaire

Vertex
Normal
Couleurs
Coord. textures

Traitement par fragment

Accès aux textures et combinaison des valeurs, Z-test, alpha-test, alpha blending, stencil-test

Vertex
Normal
Couleurs
Coord. textures

image

Programmabilité « totale » des cartes

scène

Traitement par vertex

Programme Utilisateur (vertex programm)

Traitement par fragment

Programme utilisateur (fragment programm)

image

Un langage C spécifique a été développé (Cg de Nvidia ou GL Shading Language pour OpenGl)

GL Shading Language

Ce langage utilise :

- Des variables de type : float, bool, int, vec2, vec3, vec4, mat2, mat3, mat4, sampler1D, sampler2D, sampler3D ;
- selon trois qualificatifs: attribute (normal, vertex, etc.), uniform (LightPosition) ou varying.
- Des expressions arithmétiques (il existe également des opérations de base)
- Des structures de contrôle: if , for, while

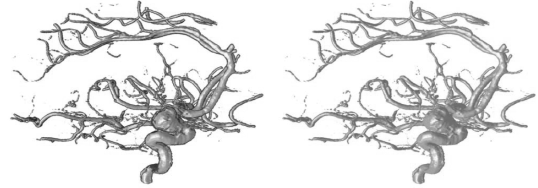
- Un certain nombre de variables sont pré-définies: gl_Color, gl_Vertex, etc.

Exemple de programmes

```
varying vec3 vNormal;
varying vec3 vVertex;


void main(void)
{
 vVertex = gl_Vertex.xyz;
 vNormal = gl_Normal;

 gl_Position = gl_ModelViewProjectionMatrix *
      gl_Vertex;
}
```

```
varying vec3 vNormal;
varying vec3 vVertex;
uniform vec4 color0;
uniform vec4 color1;
uniform vec4 color2;
#define shininess 20.0
void main (void)
{
vec3 eyePos = vec3(0.0,0.0,5.0);
vec3 lightPos = vec3(0.0,5.0,5.0);

vec3 EyeVert = normalize(eyePos - vVertex);
vec3 LightVert = normalize(lightPos - vVertex);
vec3 EyeLight = normalize(LightVert+EyeVert);
vec3 Normal = normalize(gl_NormalMatrix * vNormal);

float sil = max(dot(Normal,EyeVert), 0.0);
if (sil < 0.3) gl_FragColor = color1;
else
{
 gl_FragColor = color0 ; //* sil;
```


Exemple d'application



Conclusion Partie II

Les cartes graphiques ont subi une évolution rapide et intègrent de plus en plus de fonctionnalités (poussées par l'industrie du jeu vidéo)

OpenGl +GLSL est un langage de programmation bas niveau permettant de faire des affichages graphiques 3D à partir de primitives simples.