

Applications 3D Interactives

Tamy Boubekour

INTRODUCTION

Applications 3D

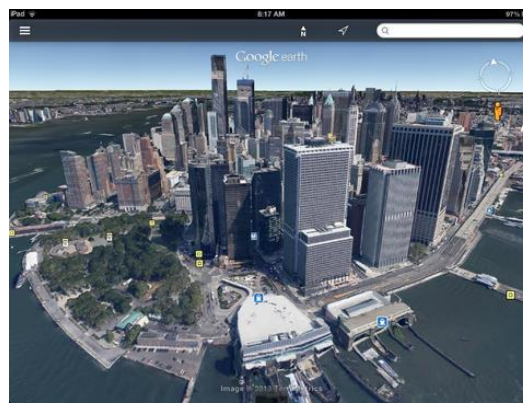
- Manipule des données en 3 dimensions
 - Surfaces
 - Volumes
- Potentiellement animées
- Capturées dans le monde réel ou créées virtuellement

Interaction 3D

- Navigation 3D
 - Manipulation de caméra
 - Définition de trajectoire
- Interaction avec le contenu 3D
 - Sélection
 - Edition
 - Transformation
 - Composition
 - Recherche
- Via des périphériques standards ou spécifiques
 - Souris, clavier écran tactile
 - Pinceau à retour de force, interacteurs pour réalité virtuelle



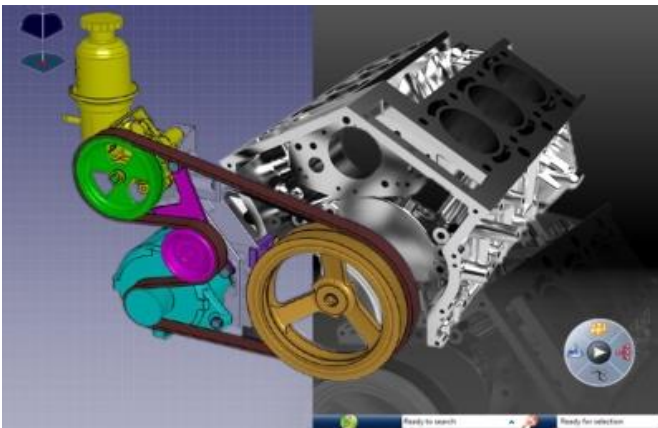
Jeux vidéo



Géomatique



Réalité Augmentée



CAO



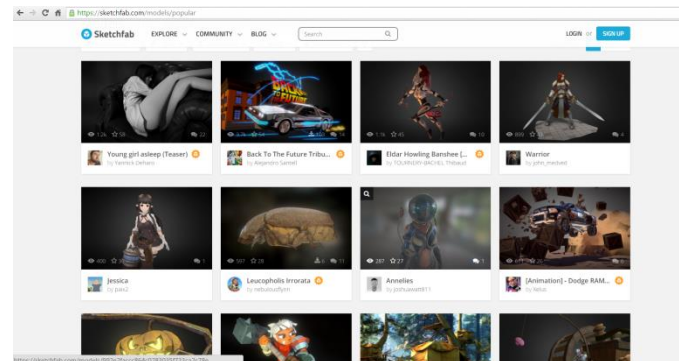
Réalité virtuelle



3D Mobile



Création de contenu



Web 3D

Etc...

Structure d'une App 3D

Fonction

Exemples

Données, logique

Application

Blender, Chrome,

Fenêtrage,
gestion des évènements

Environnement
Graphique

GLUT, Qt

Gestion et mise à
disposition des ressources
matériel

Système d'exploitation

Windows, Linux,,, iOS

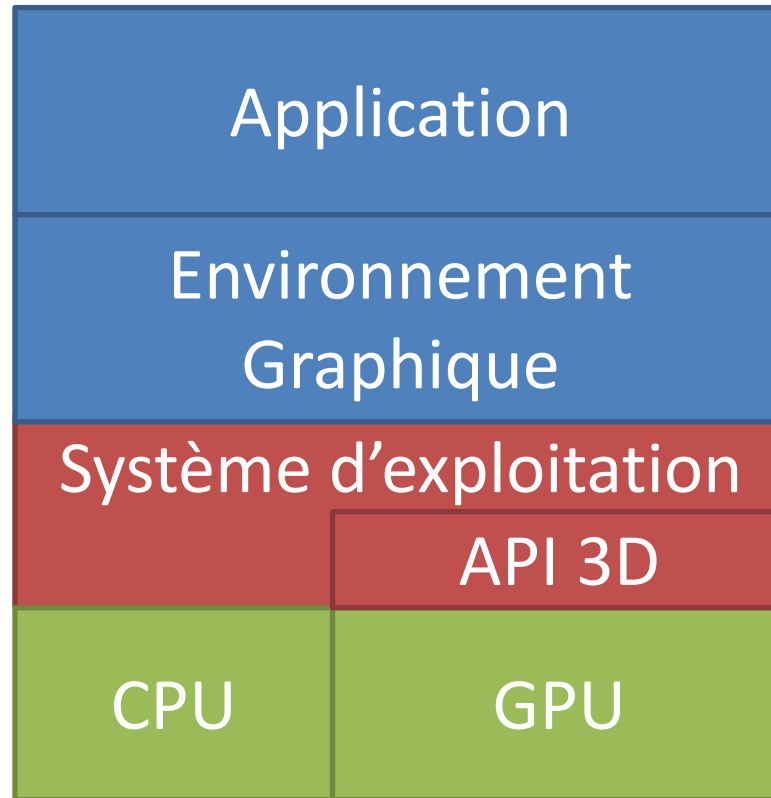
API 3D

OpenGL, DirectX

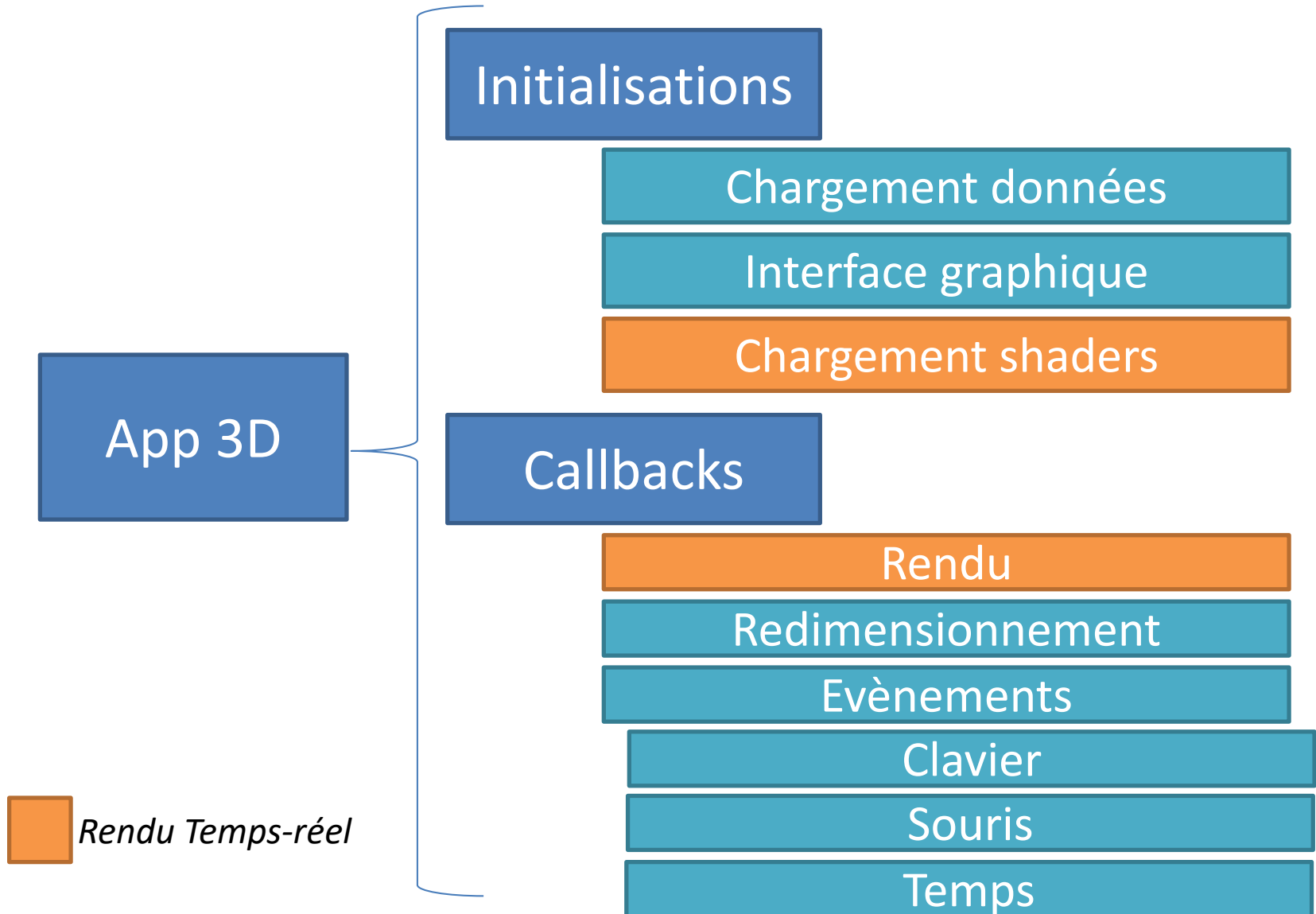
Exécution

CPU

GPU



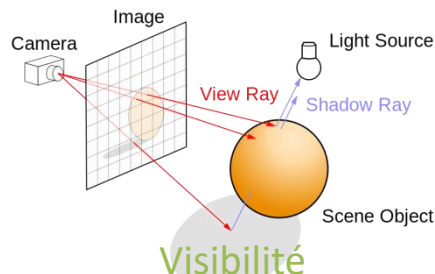
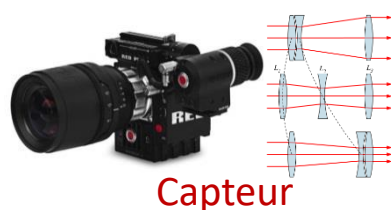
Squelette d'une App 3D



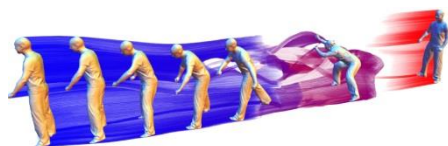
RENDU TEMPS-RÉEL

Synthèse d'Image

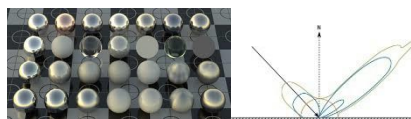
Modèles



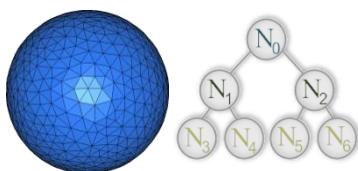
Algorithmes



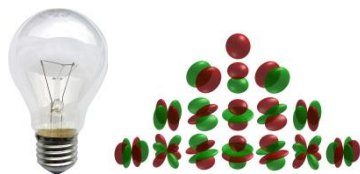
Mouvement



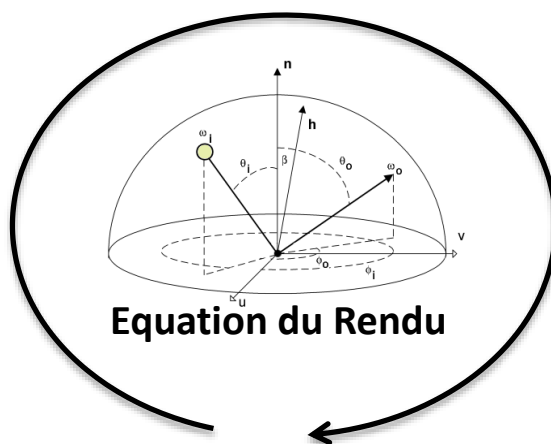
Apparence



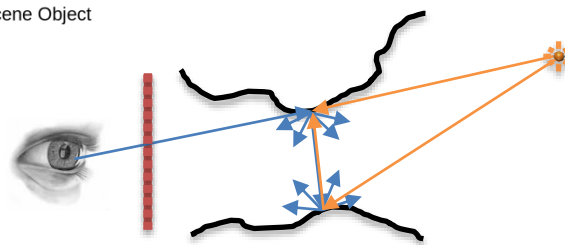
Forme



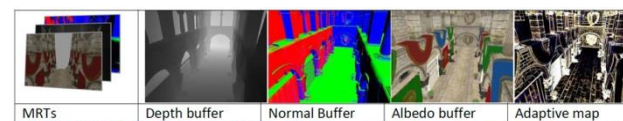
Lumière



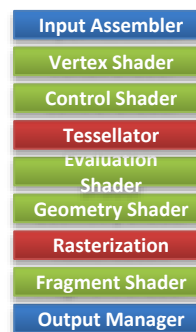
Equation du Rendu



Eclairage



Post-processing



Pipeline GPU



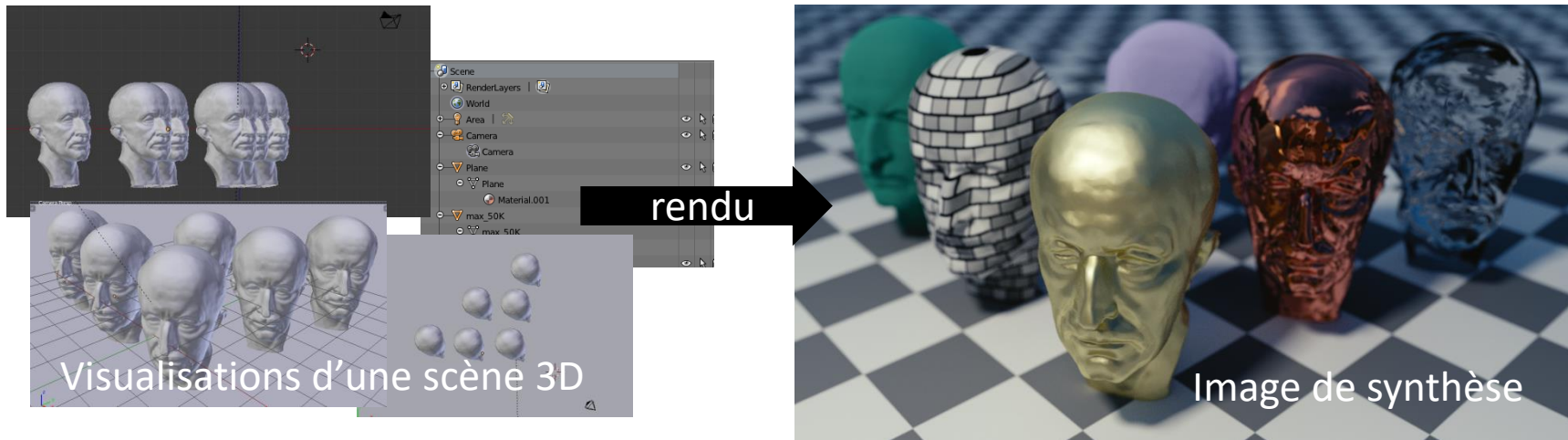
Rendu Temps-Réel

Implémentations

Synthèse d'Image

- Informatique
 - Physique
 - Mathématiques Appliquées
 - Traitement du Signal
-
- Une discipline de l'Informatique Graphique

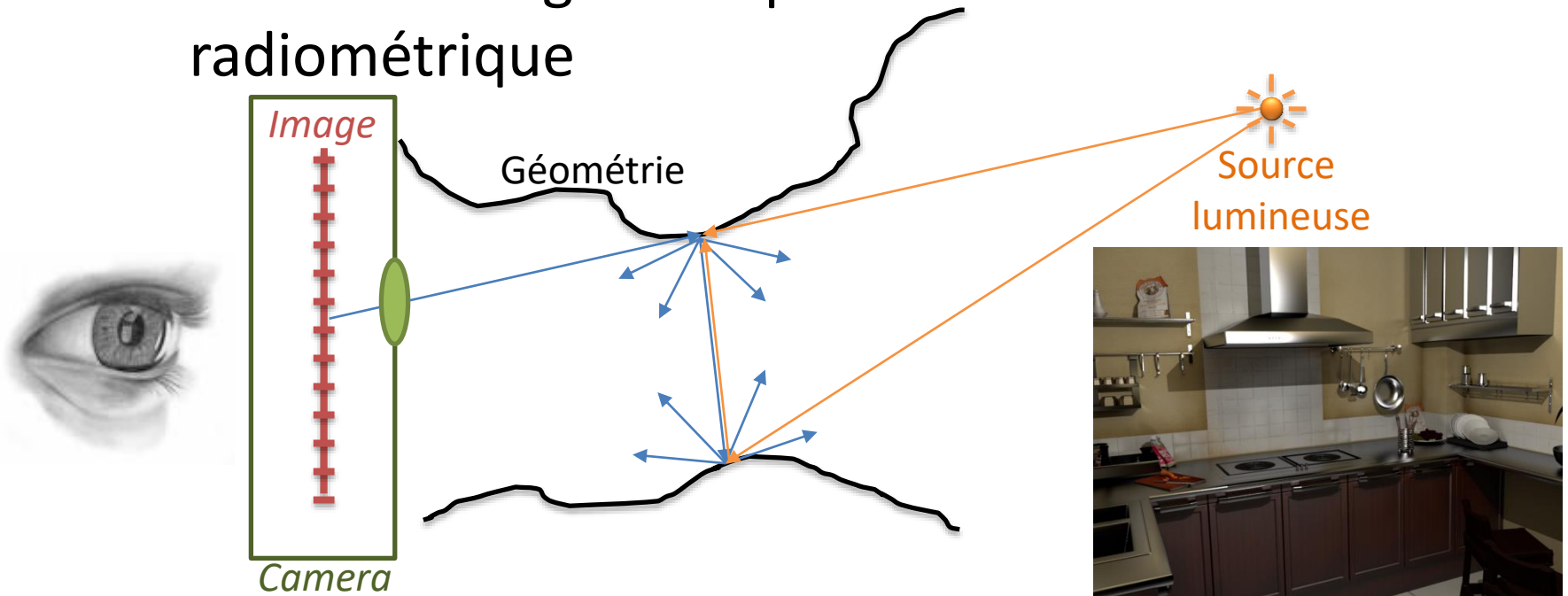
Un processus de simulation



- Rendu = Synthèse d'Image
- Génération d'une image numérique à partir d'une scène 3D
- Réaliste (physiquement plausible) ou expressif

Rendu basé physique

- Simulation du transport de la lumière
 - De la source au capteur
 - Dans une scène virtuelle
 - Pixel de l'image du capteur = mesure radiométrique



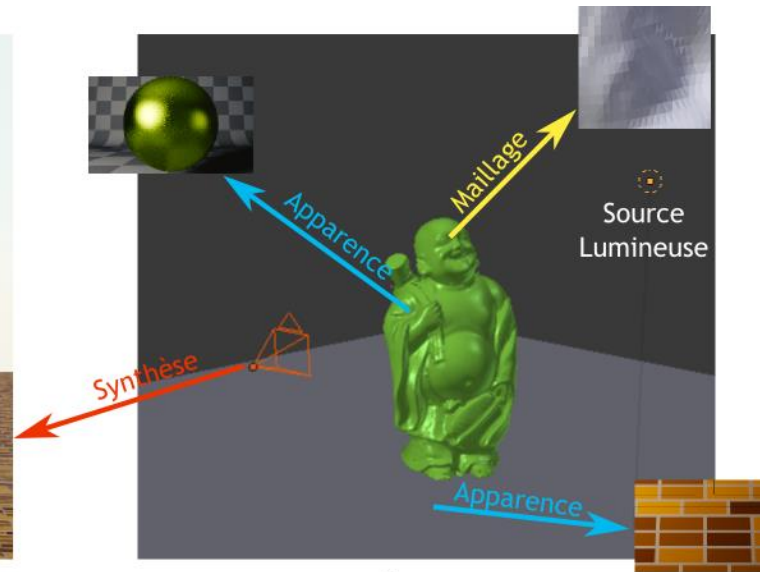
Rendu temps-réel

- Pour les systèmes interactifs
- Approximation géométrique et radiométrique de la scène
- Rendu par projection (rasterization)
- Calcul parallèle (GPU)

Modèles de Scène 3D



Image Numérique

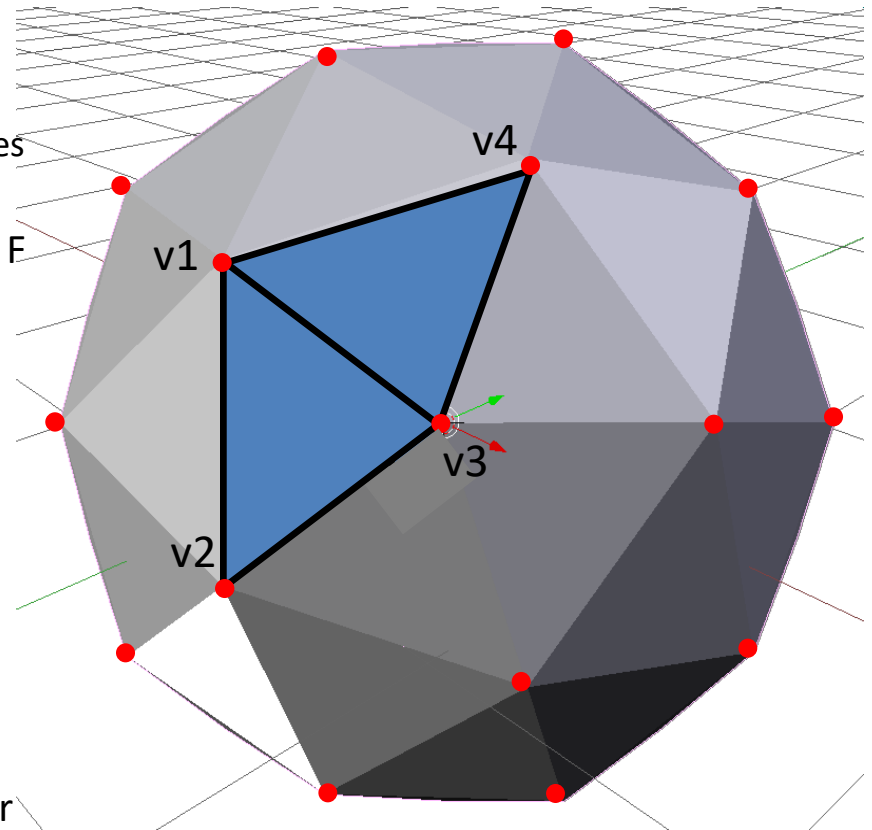


Scène 3D

- Une collection de modèles :
 - Capteur (caméra)
 - Géométries
 - Maillages, particules, iso-surfaces, etc
 - Apparence
 - Matériaux, textures
 - Lumières
 - Animation
 - Évolution temporelles des paramètres
- des autres modèles
 - Physique solide, fluides, corps déformables
 - Interactivité et actuators
- Une structure entre ces modèles
 - Appartenance et hiérarchie
 - Données et instances

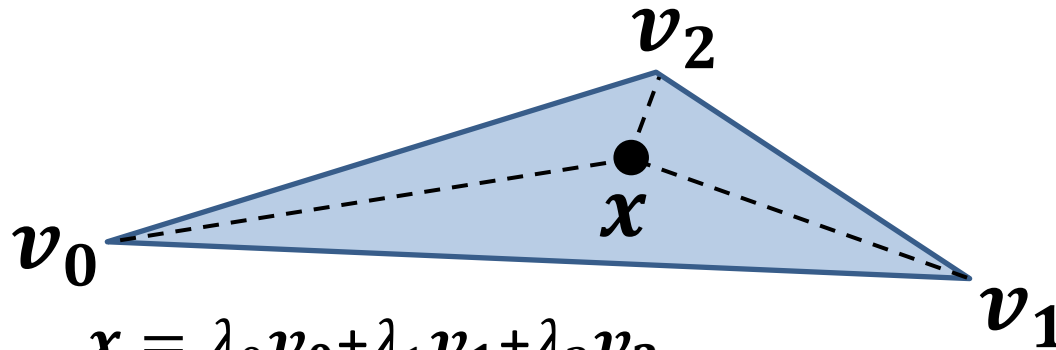
Surface Maillée

- Maillage:
 - modèle géométrique dominant en rendu
 - Génération possible à partir des autres modèles
 - Cf cours de Modélisation Géométrique
- Définition: un ensemble de faces polygonales F indexant un ensemble de sommets V .
- V : Ensemble de sommets (géométrie)
 - $v1(x, y, z)$
 - $v2(x, y, z)$
 - $v3(x, y, z)$
 - $v4(x, y, z)$
- F : Ensemble de faces (topologie)
 - $(v1, v2, v3)$
 - $(v1, v3, v4)$
- Outre la position, chaque sommet peut porter d'autres attributs:
 - vecteur *normales*, critiques en rendu.
 - *Couleur par sommet*
 - *Coordonnées de textures (UV)*



Coordonnées barycentrique

- Coordonnée d'un point dans l'espace d'un polygone
- Simple pour un triangle
- Permet d'interpoler linéairement tout attribut de sommet

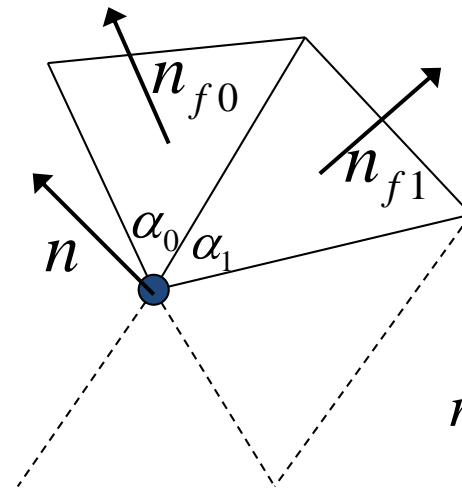
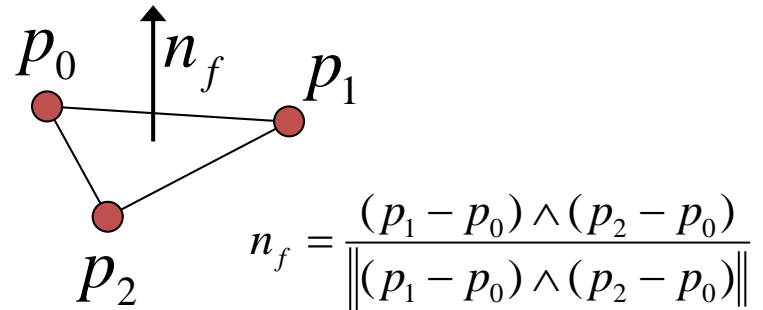


Coordonnée
barycentriques de x
dans le triangle

$$\begin{cases} x = \lambda_0 v_0 + \lambda_1 v_1 + \lambda_2 v_2 \\ \lambda_0 = \frac{[(v_2 - v_1) \otimes (x - v_1)] \cdot a}{(v_1 - v_0) \otimes (v_2 - v_0)} \\ \lambda_1 = \frac{[(v_0 - v_2) \otimes (x - v_2)] \cdot a}{(v_1 - v_0) \otimes (v_2 - v_0)} \\ \lambda_2 = \frac{[(v_1 - v_0) \otimes (x - v_0)] \cdot a}{(v_1 - v_0) \otimes (v_2 - v_0)} \end{cases}$$
$$a = \frac{(v_1 - v_0) \otimes (v_2 - v_0)}{\|(v_1 - v_0) \otimes (v_2 - v_0)\|^2}$$

Normales

- **Essentielles pour le rendu**
 - Alignement de la BRDF
- Stockage aux sommets ou par cartes (normal maps)
- Calculs possibles:
 - Moyennes des normales des faces incidentes
 - Moyennes pondérée par les angles des arêtes incidentes
 - Plus robustes pour les distributions de triangles non uniformes
 - Moyenne pondérée par l'aire de l'intersection du triangle et de la cellule de voronoi du sommet.



$$\dot{n} = \frac{\sum_i \alpha_i n_i}{\sum_i \alpha_i}$$
$$n = \frac{\dot{n}}{\|\dot{n}\|}$$

Interpolation de Normales

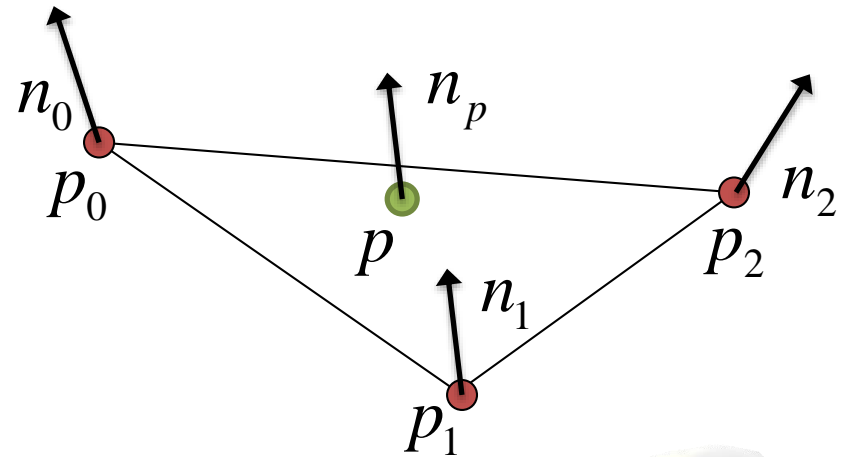
- Une normale en chaque point d'un maillage à partir des normales de ses sommets [Phong 75].
- Soit un point p sur un triangle t tel que :

$$p = \lambda_0 p_0 + \lambda_1 p_1 + \lambda_2 p_2$$

Avec $(\lambda_0, \lambda_1, \lambda_2)$ les **coordonnées barycentrique** de p dans t .

Alors on définit la normale interpolée de Phong en p comme :

$$n_p = \frac{\lambda_0 n_0 + \lambda_1 n_1 + \lambda_2 n_2}{\|\lambda_0 n_0 + \lambda_1 n_1 + \lambda_2 n_2\|}$$



Normale de face



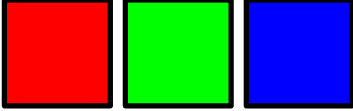
Normale de Phong

IMAGERIE

Image Numérique

- Grille de pixels couleurs
- Pixel:
 - Photographie numérique : correspond à un point une petite région sur le capteur photosensible
 - Image de synthèse :
 - correspond à un point le capteur virtuel
 - correspond à un ensemble de chemins lumineux
 - correspond à un point 3D de la scène
 - correspond à une collection d'attributs interpolés

Couleur en image de synthèse

- Espace RGB 
- Précision variable
 - 8bits/composante : standard des APIs, des GPU, du rendu et des écrans
 - 16/32bits par composante : imagerie à haute dynamique (HDR)
 - Nécessite une conversion avant affichage (Tone Mapping)
- **RGBA** : transparence *alpha* par pixel
 - Pose un problème d'ordonnancement
 - Pas directement compatible avec les algorithmes de rendu par projection (Z-Buffer)

INTRODUCTION À C++

C++

- Langage de programmation dominant en informatique graphique
- Abstraction de haut niveau
 - Structure élégante des programmes
- Accès bas niveau
 - Programmes efficaces

Du C au C++

- **Objets** et héritage multiple
- Patrons de classes (templates)
- Surcharge et **redéfinition** des opérateurs
- Gestion des exceptions

- STL : bibliothèque standard (vector, list, IO, etc)
- Langage très (trop) permissif
 - Se donner des règles et les respecter
 - Dans ce cours :
 - pas besoin d'héritage, ni de template complexes, ni d'itérateurs, etc
 - se limiter à des classes simples
 - ne pas hésiter à implémenter des fonctions isolées type C pour le traitement
 - exploiter au maximum la STL

Class et opérateur

```
#include <iostream>
```

```
class Vec2 {
```

```
public:
```

```
    Vec2 (float x = 0.0, float y = 0.0) : _p[0] (x), _p[1] (y) {} // Constructeur
```

```
    ~Vec2 () {} // Destructeur
```

```
    inline float length () const { return sqrt (_p[0]*_p[0]+_p[1]*_p[1]); } // Méthode
```

```
    inline float & operator[] (int i) { return _p[i]; } // Redéfinition de l'opérateur crochet
```

```
    inline const float & operator[] (int i) const { return _p[i]; } // Surchage pour instances constantes
```

```
private:
```

```
    float _p[2]; // Attribut
```

```
};
```

```
int main (int argc, const char * argv[]) {
```

```
    Vec2 x (2.0, 3.0); // Construction
```

```
    std::cout << x.length () << std::endl; // Affiche 3.6055...
```

```
    return 0; // Retour sans erreur
```

```
}
```

Mémoire et pointeurs

```
#include <iostream>
```

```
[...]
```

```
int main (int argc, const char * argv[]) {  
    Vec2 * x = new Vec2 (2.0, 3.0); // Appelle du constructeur  
    std::cout << x->length () << std::endl;  
    delete x; // Détruit la valeur pointée à l'aide du destructeur du type  
    return 0;  
}
```

L'allocation dynamique est couteuse en temps : à réserver au « gros » objets.

Références

```
#include <iostream>
```

```
void doNothing (Vec2 v) {
```

```
    v[0] = 5.0;
```

```
}
```

```
void doSomething (Vec2 & v) {
```

```
    v[0] = 5.0;
```

```
}
```

```
int main (int argc, const char * argv[]) {
```

```
    Vec2 x (2.0, 3.0);
```

```
    doNothing (x);
```

```
    std::cout << x[0] << std::endl; // Affiche 2.0
```

```
    doSomething (x);
```

```
    std::cout << x[0] << std::endl; // Affiche 5.0
```

```
    return 0;
```

```
}
```

STL

```
#include <iostream>
```

```
#include <vector>
```

```
int main (int argc, const char * argv[]) {
```

```
    std::vector<Vec2> polygon;
```

```
    polygon.push_back (Vec2 (0.0, 0.0));
```

```
    polygon.push_back (Vec2 (1.0, 0.0));
```

```
    polygon.push_back (Vec2 (0.0, 1.0));
```

```
    if (polygon.size () == 3)
```

```
        std::cout << « C'est un triangle » << std::endl;
```

```
    return 0;
```

```
}
```

Une classe complète

- La class Vec3
- Base des TPs de ce cours

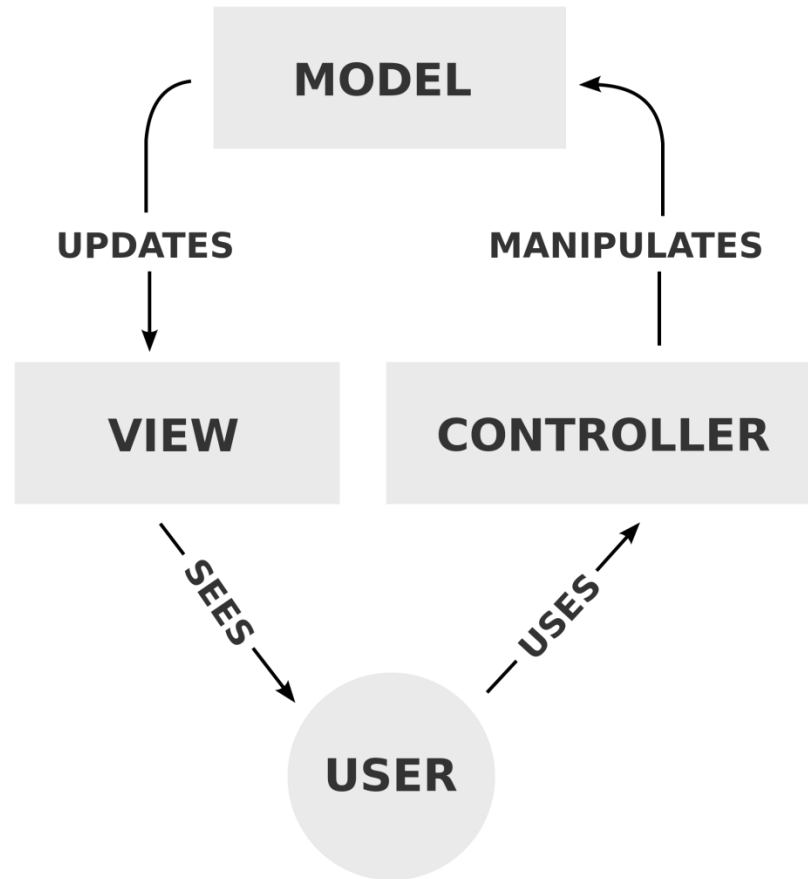
[Listing code]

Références sur C++

- The C++ Programming Language, Special Edition, Bjarne Stroustrup, Addison-Wesley, 2009
- www.cplusplus.com
- <https://www.sgi.com/tech/stl/>

ELEMENTS D'ARCHITECTURE

Model-View Controller



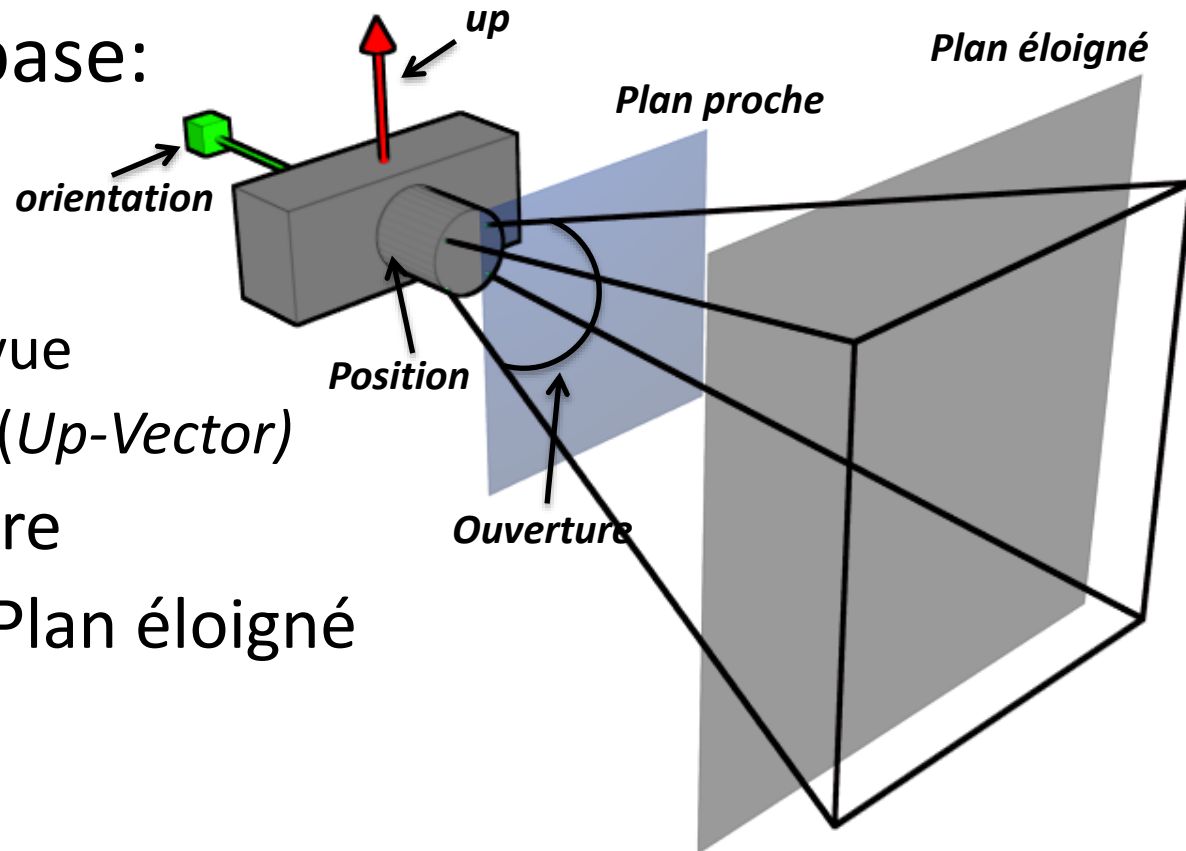
Design Patterns Utiles

- Singleton
- Décorateur
- Composition
- Fabrique (abstraite)
- Façade
- Observateur
- Commande

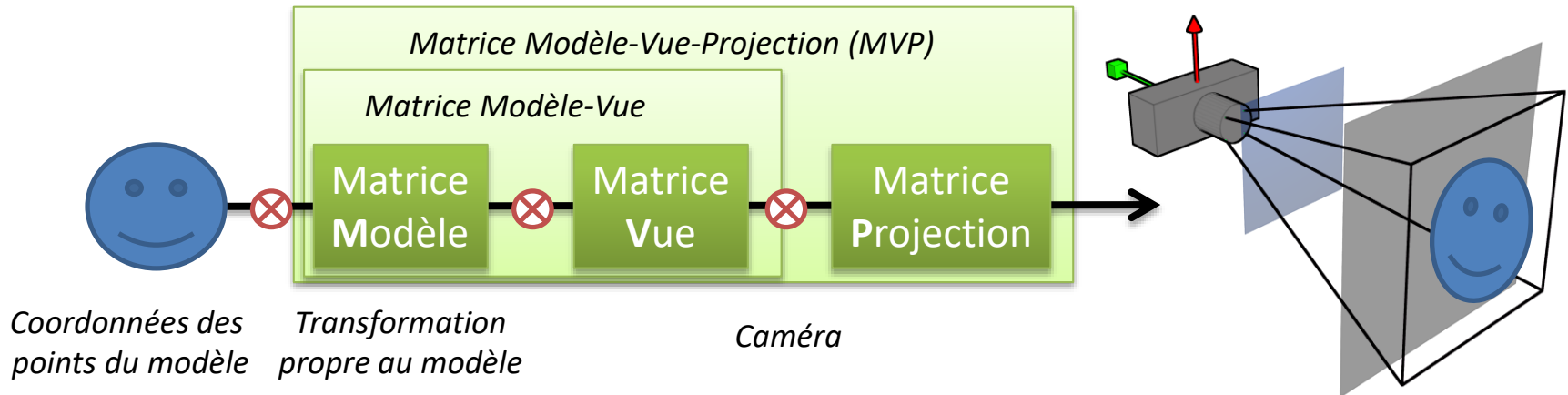
CAMÉRA ET TRANSFORMATIONS

Modèle de Caméra

- En général: *pinhole* camera
- Projection perspective
- Paramètre de base:
 - Position
 - Direction de vue
 - Vecteur haut (*Up-Vector*)
 - Angle Ouverture
 - Plane proche/Plan éloigné



Transformation et Projections

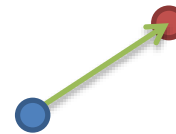


- Représentation par une matrice 4x4
- Transformation rigide
 - Translation
 - Rotation
 - Echelle
- Utilisation: changement de repère pour le placement des géométries dans le repère de la caméra et leur projection

Transformation Affine

Translation

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$



Rotation

$$R_X(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix} \quad R_Y(\theta) = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix} \quad R_Z(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Scaling

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transformations appliquées pour :

- *Déplacer les sommets des polygones en 3D*
- *Les placer dans le repère de la caméra*

Projection

- Projeter les sommets des polygones (transformés) dans le plan de l'image.
- 2 types de projections:



- Encore une fois exprimable à l'aide d'une matrice 4x4 : la **matrice de projection**

Géométrie Projective

- Reasonner dans l'espace des droites
- Projection en perspective

Point $xyz > xyzw$: **coordonnées homogènes**

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ w=1 \end{pmatrix} \rightarrow \dots \textit{transformations} \dots \rightarrow \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \rightarrow \begin{pmatrix} x' / w' \\ y' / w' \\ z' / w' \end{pmatrix}$$

***Projection
Homogène***

APP 3D AVEC OPENGL

App 3D OpenGL

- Manipule un ensemble de polygones, structuré par l'application
 - De la simple liste
 - Au graphe de scène complet, avec description sémantique
- Textures : images couleurs plaquées sur les polygones
- Rendu temps réel de la scène sous forme d'images couleurs affichées à l'écran
 - Boucle de rendu
- Interaction : événements utilisateurs (e.g., clavier, souris, *touch screen*)
 - Callbacks

Boucle de rendu

- Rendu temps-réel : en général effectué par le GPU
 - Effectue des appels à une API graphique
 - API dédiés OpenGL, DirectX, Metal, Optix, etc
- Données
 - Maillage polygonal échantillonnant la scène
 - Propriétés de surface : normal, coordonnées de textures,
 - Textures
 - Matériaux
 - Sources de lumières
 - Paramètres caméra

GPU : Données en entrée

- **Maillage polygonal** : approximation de la surface d'un objet à l'aide d'un ensemble de polygones
 - **Soupe de Polygones** : suites de n-uplets de coordonnées 3D correspondants aux polygones
 - **Maillages indexés** : graphe avec géométrie et topologie séparés
 - Une liste de sommets (V)
 - Une liste de relation topologique:
 - Arêtes (Edge, E)
 - Faces (F)
- En pratique, {V,F} (exemple : OpenGL)

Pipeline Graphique Moderne

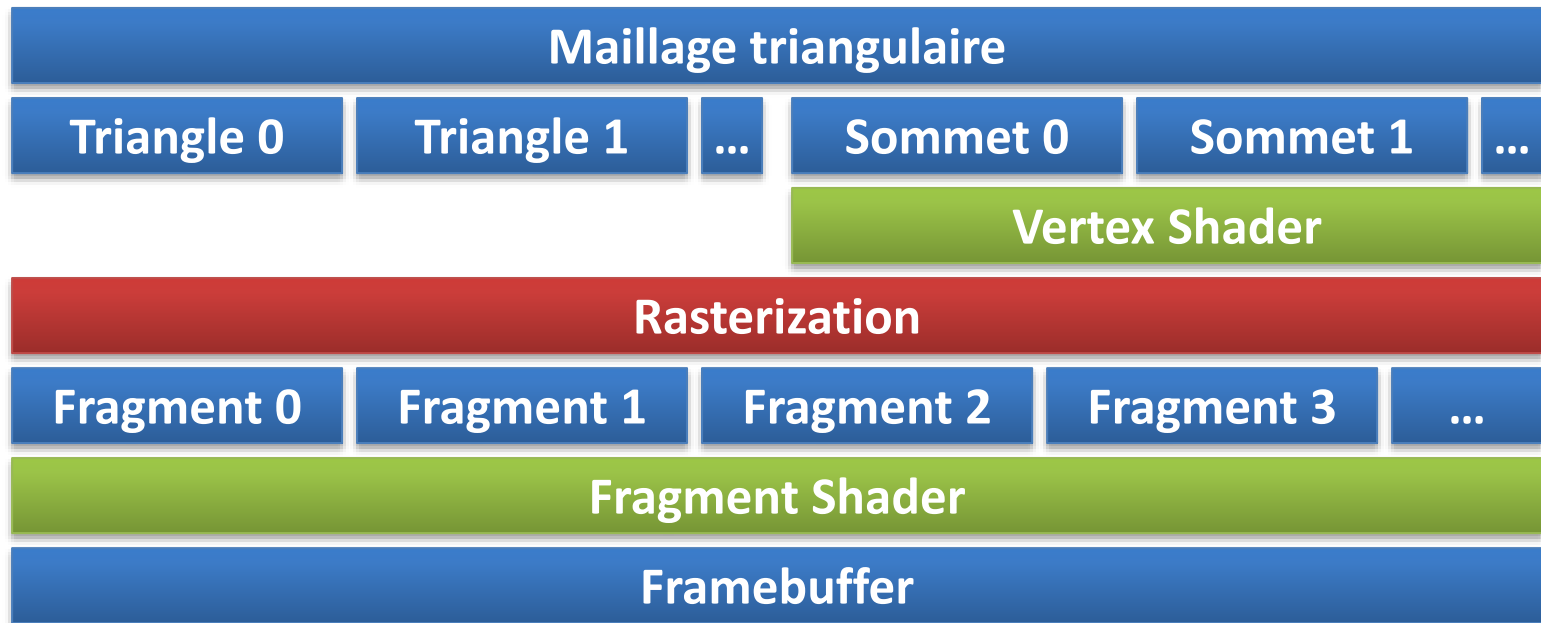
- Direct3D 11+ / OpenGL 4+
- Vue API
- Collaboration GPU computing possible (CUDA/OpenCL)
- Implémentation sur processeurs de flux génériques
- Compute Shaders : calcul non graphique de support (mini GPU Computing)



 Programmable  Configurable  Fixe

Etages majeurs

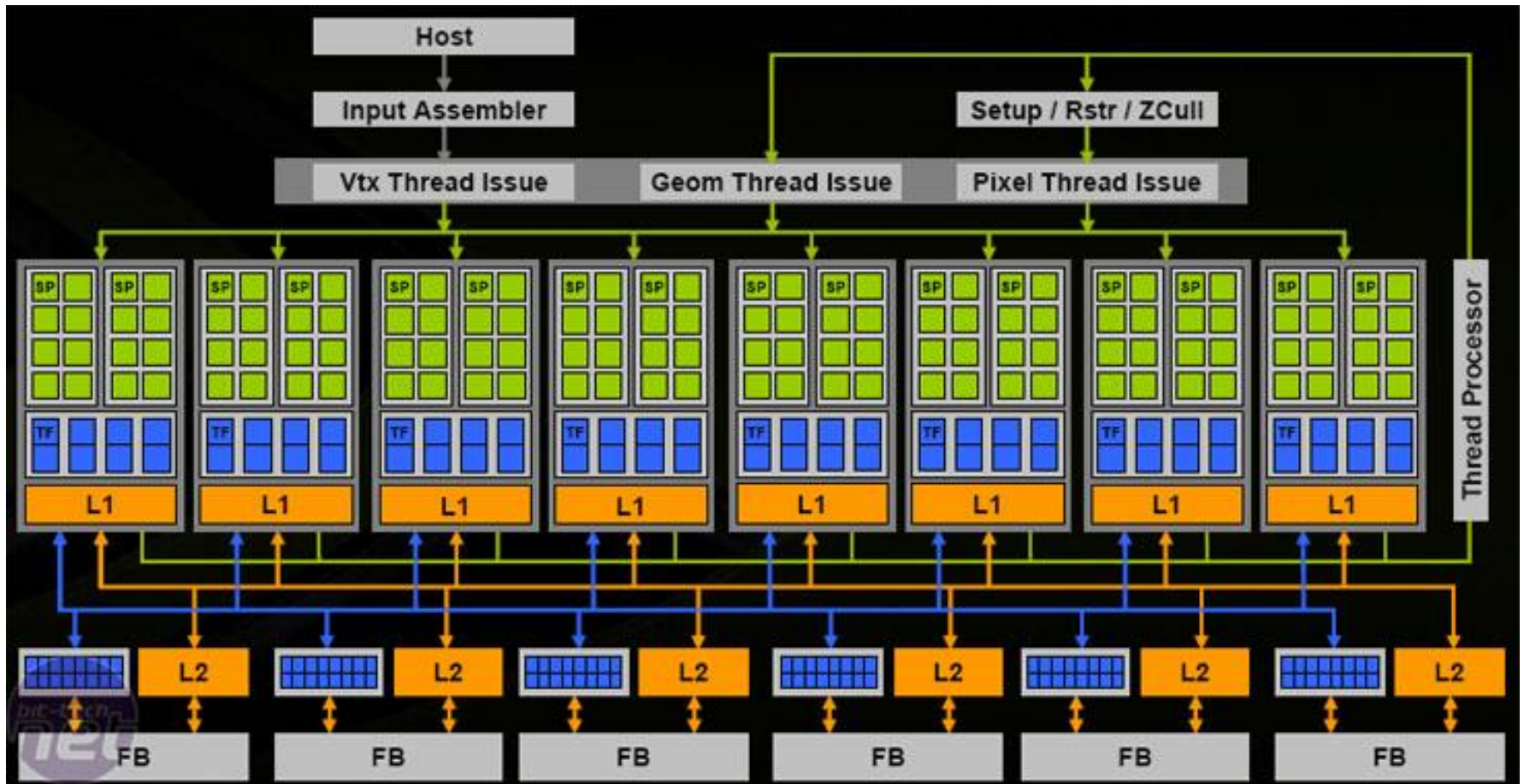
- Calcul en flux



- Intégralement parallèle
 - Chaque sommet traité indépendamment
 - Chaque fragment traité indépendamment

Architecture Vue Matérielle

(NVIDIA G80)

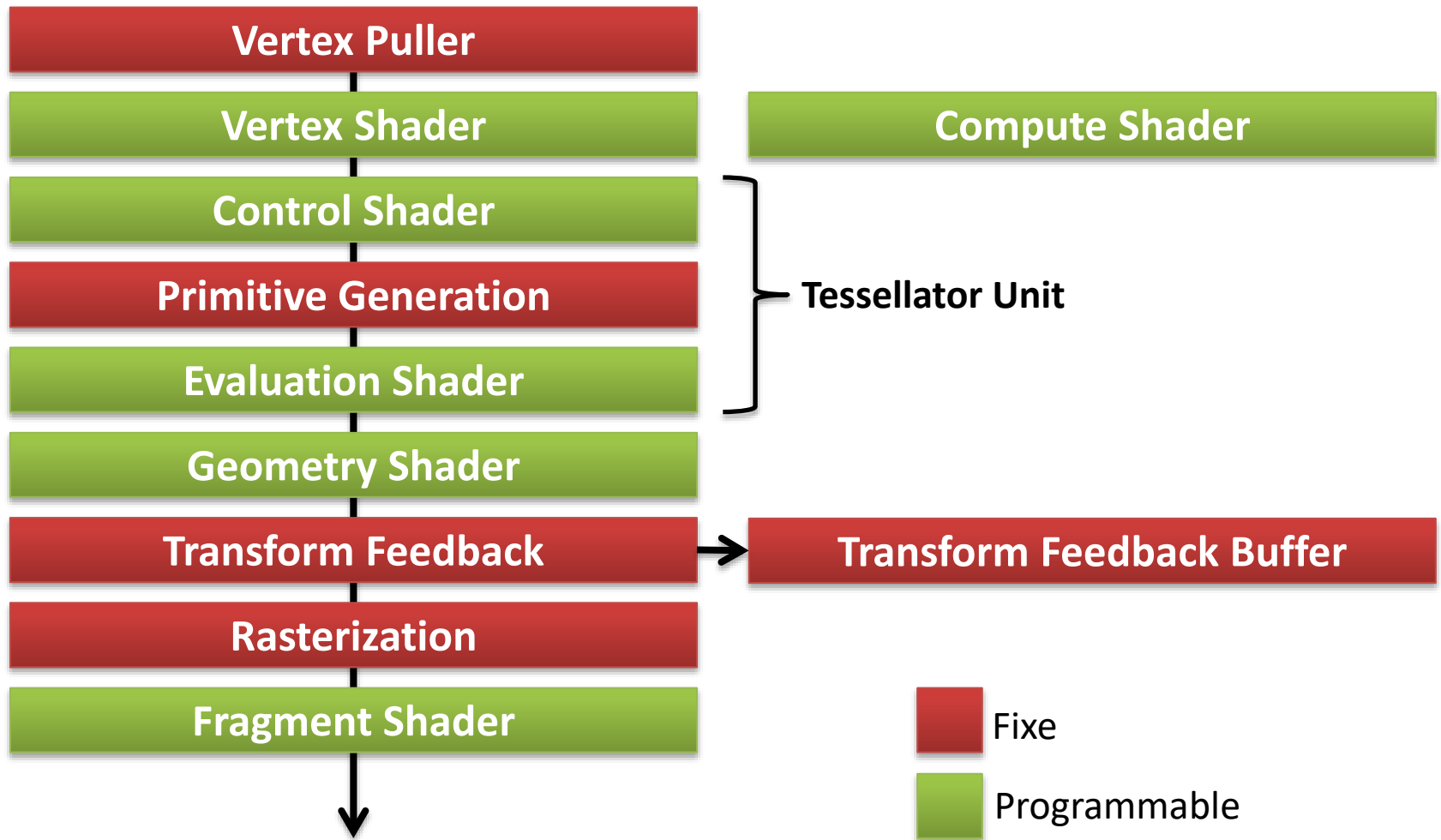


API 3D

- **OpenGL**
 - Maintenue par Khronos Group (consortium)
 - Multiplateforme
 - Robuste (CAD)
 - Evolution lente, *suiveur*
- **Direct 3D**
 - Développé par Microsoft
 - Standard sous MS Windows (PC, XBOX, PocketPC)
 - Evolue très vite
 - Dicte la marche à suivre aux constructeurs de GPUs
 - Drame du *Géométrie Shader*
 - Futurs drames possibles : Tessellator Unit, Raytracing

Pipeline Moderne

OpenGL 4.3



OpenGL

- OpenGL classique (v1)
 - Pas de programmation GPU, pipeline fixe
- OpenGL programmable (v2,v3)
 - Programmation shaders, entrées-sorties formatées
- OpenGL moderne (v4)
 - Shaders graphiques et shaders calcul, entrées sorties redéfinissables.

OpenGL

- API graphique générique
 - Mac/PC/Linux/iOS/Android/html5/etc
- Plusieurs versions
 - OpenGL Classique (v1.2)
 - Pas de programmation GPU
 - Pipeline fixe
 - OpenGL Programmable (v2.0)
 - Programmation GPU (shaders)
 - Entrées-sorties formatées
 - OpenGL Moderne (v3/v4)
 - Shaders graphiques et shaders calcul,
 - Entrées sorties redéfinissables.
 - Nouveaux étages : geometry, tessellation

Programmation avancée en OpenGL traitée en IGR202

Mode Immédiat (OpenGL v1.x)

- Structure glBegin () ... glEnd ();
- Programmation déclarative de la scène à dessiner
 - Mise à jour des états OpenGL
 - vecteur normal : glNormalf (nx, ny, nz)
 - glColor3f (r, g, b);
 - Emission de sommets
 - glVertex3f (x, y, z);
 - Composition de primitive à dessiner
 - glBegin (GL_TRIANGLES) : une triangle est dessiné après trois appel à glVertex3f
 - glBegin (GL_LINES) : un segment est dessinée après 2 appels à glVertex3f

Mode Immédiat (OpenGL v1.x)

- Boucle de rendu typique

```
void display () {  
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glBegin (GL_TRIANGLES);  
    for (unsigned int i = 0; i < .size (); i++)  
        for (unsigned int j = 0; j < 3; j++) {  
            glNormalf (triangles[i].vertex[j].normal [0],  
                      triangles[i].vertex[j].normal [1],  
                      triangles[i].vertex[j].normal [2]);  
            glColor3f (triangles[i].vertex[j].color.red,  
                      triangles[i].vertex[j].color.green,  
                      triangles[i].vertex[j].color.blue);  
            glVertex3f (triangles[i].vertex[j].position [0],  
                      triangles[i].vertex[j].position [1],  
                      triangles[i].vertex[j].position [2]);  
        }  
    glEnd ();  
}
```

HelloWorldGL.cpp (OpenGL v1.x)

```
#include <GL/glut.h> // Inclue GL.h automatiquement
// Pour compiler: g++ -o HelloWorldGL -lGL -lglut HelloWorldGL.cpp

void display () {
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Efface les composantes couleurs et profondeur (zbuffer) du framebuffer
    glBegin (GL_TRIANGLES); // Commence une liste de primitive de type triangle
    glColor3f (1.0, 0.0, 0.0); // Spécifie la couleur RGB de tous les sommets à partir de maintenant
    glVertex3f (0.0, 0.0, 0.0); // Emet un sommet de coordonnées x=0, y=0, et z=0
    glVertex3f (1.0, 0.0, 1.0);
    glVertex3f (0.0, 1.0, 0.0); // Trois sommets ont été émis > un triangle, rouge ici, a été dessiné
    glColor3f (0.0, 1.0, 0.0); // La couleur passe au vert désormais
    glVertex3f (1.0, 0.0, 0.0);
    glVertex3f (1.0, 1.0, 0.0);
    glVertex3f (0.0, 0.0, 1.0); // Un triangle vert est dessiné
    glEnd (); // Termine la liste de primitive à dessiner
    glutSwapBuffers ();

    // Tout a été dessiné dans un buffer arrière, pour éviter un dessin progressif à l'écran. On échange le buffer arrière avec le buffer avant
    // (celui dessiné à tout instant à l'écran) avec glutSwapBuffers.
}

int main (int argc, char ** argv) {
    glutInit (&argc, argv); // Initialise le gestionnaire de fenêtre glut
    glutInitDisplayMode (GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE); // Les buffers seront en RGBA. Un buffer arrière est présent.
    glutInitWindowSize (1280, 800); // La résolution de la fenêtre et de ses buffers
    glutCreateWindow («HelloWorldGL.cpp»); // Création de la fenêtre à l'écran
    glEnable (GL_DEPTH_TEST); // Active le zbuffer
    glutDisplayFunc (display); // Place le pointeur vers la fonction de dessin OpenGL
    glutMainLoop (); // Commence un boucle de dessin infinie.
    return 0;
}
```

Interfaçage au système d'exploitation

- GLUT
- Qt
- WxWidget
- Fournissent en général un mécanisme de callback pour
 - la mise à jour de l'image affichée
 - les évènements claviers
 - les évènements souris

Shaders

- **Vertex**
 - **Control**
 - **Evaluation**
 - **Geometry**
 - **Fragment**
 - ***Compute***
- **Pipeline**
 - Synchronisation I/O
 - Faiblement dynamique
 - Geometry Shader :
 - Très Flexible
 - Faible amplification
 - Tessellation :
 - Peu flexible
 - Grande amplification

Langages de Shaders

- Premiers langages développés pour la programmation GPU
- « Types de bases »
 - Sommets, vecteurs, matrices, textures, pixel (fragment), couleur, etc
- De plus en plus flexibles
 - Boucles, branchements conditionnels, macros, structures
- Exemples :
 - Cg (nVidia)
 - HLSL (Direct3D/Microsoft)
 - **GLSL** (OpenGL)
 - tous relativement équivalents
- *Premier succès populaire de la programmation parallèle*

GLSL I

- OpenGL Shading Language
- Complète OpenGL
 - Programmation des différents étages de rendu
 - 2009 : Vertex, Geometry, Fragment
 - 2011 : Control, Evaluation (Tessellator)
 - 2012 : Compute
- Structure similaire à C/C++

GLSL II

- Types spécifiques au calcul graphique
 - vec3, vec4, Texture Sampler
- Construction standard :
 - *Boucles*
 - *Branchements*
 - *Fonctions à retour de valeur*
- Variable globales spécifiques
 - Définit au niveau du code application (C/C++/Java/Python + OpenGL)
 - Position des sources de lumières, propriétés de matériaux, matrice de transformation « model-vue »
 - Accès mémoire : texture (lecture seulement)
- Limitations :
 - Pas de récursion
 - Pas d'allocation

GLSL III

OpenGL v2/3: Entrées-sortie formatées, spécialisées à l'étage

Etage	Entrée	Sortie
Vertex	<ul style="list-style-type: none">•Un sommet•Matrice de transformation•Propriétés par sommets : position (glVertex), normale (glNormal), etc	Un sommet transformé et colorié (e.g. éclairage)
Geometry	Une primitive (line , triangle , etc)	Plusieurs primitives (nombre borné et petit)
Fragment	<ul style="list-style-type: none">•Coordonnée image x,y•Couleur et coordonnées de texture par sommet interpolée au fragment•Option : toute variable de type « varying » spécifiée aux étages supérieurs	<ul style="list-style-type: none">•Un fragment RGBA•MRT : plusieurs fragment pour les même coordonnées

Vertex Shader (OpenGL v2.x)

Valeur sera
interpolée par
fragment

`varying vec4 P;`

`varying vec3 N;`

Vecteur 4D,
type de base du
langage

`void main (void)`

`{`

`P = gl_Vertex;`

`N = gl_Normal;`

Position du sommet courant,
correspond à la valeur passée à
glVertex3f coté application CPU

Position du sommet
transformé
Couleur du sommet
transformé

`gl_Position = ftransform ();`

`}`

*Sortie du
vertex shader*

Fragment Shader (OpenGL v2.x)

Evaluation de la BRDF de Phong par pixel

```
uniform float diffuseRef;  
uniform float specRef;  
uniform float shininess;
```

```
varying vec4 P;  
varying vec3 N;
```

```
void main (void) {  
    gl_FragColor = vec4 (0.0, 0.0, 0.0, 1);  
  
    for (int i = 1; i < 4; i++) {  
        vec3 p = vec3 (gl_ModelViewMatrix * P);  
        vec3 n = normalize (gl_NormalMatrix * N);  
        vec3 l = normalize  
        (gl_LightSource[i].position.xyz - p);  
        float diffuse = max (dot (l, n), 0.0);
```

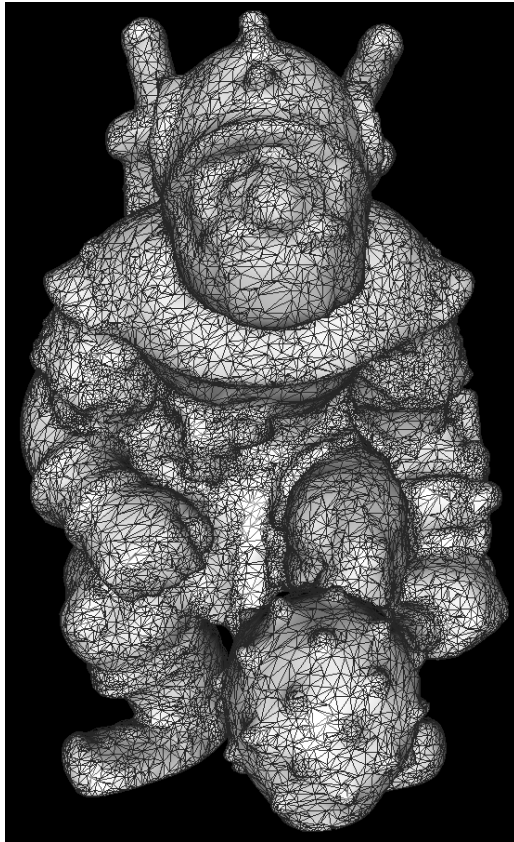
```
vec3 r = reflect (l, n);  
vec3 v = normalize (-p);
```

```
float spec = max(dot(r, v), 0.0);  
spec = pow (spec, shininess);  
spec = max (0.0, spec);
```

```
vec4 LightContribution =  
    diffuseRef * diffuse *  
gl_LightSource[i].diffuse +  
    specRef * spec * gl_LightSource[i].specular;  
gl_FragColor += vec4 (LightContribution.xyz, 1);  
}
```

Fragment Shader (OpenGL v2.x)

Rendu binaire style « Sin City »



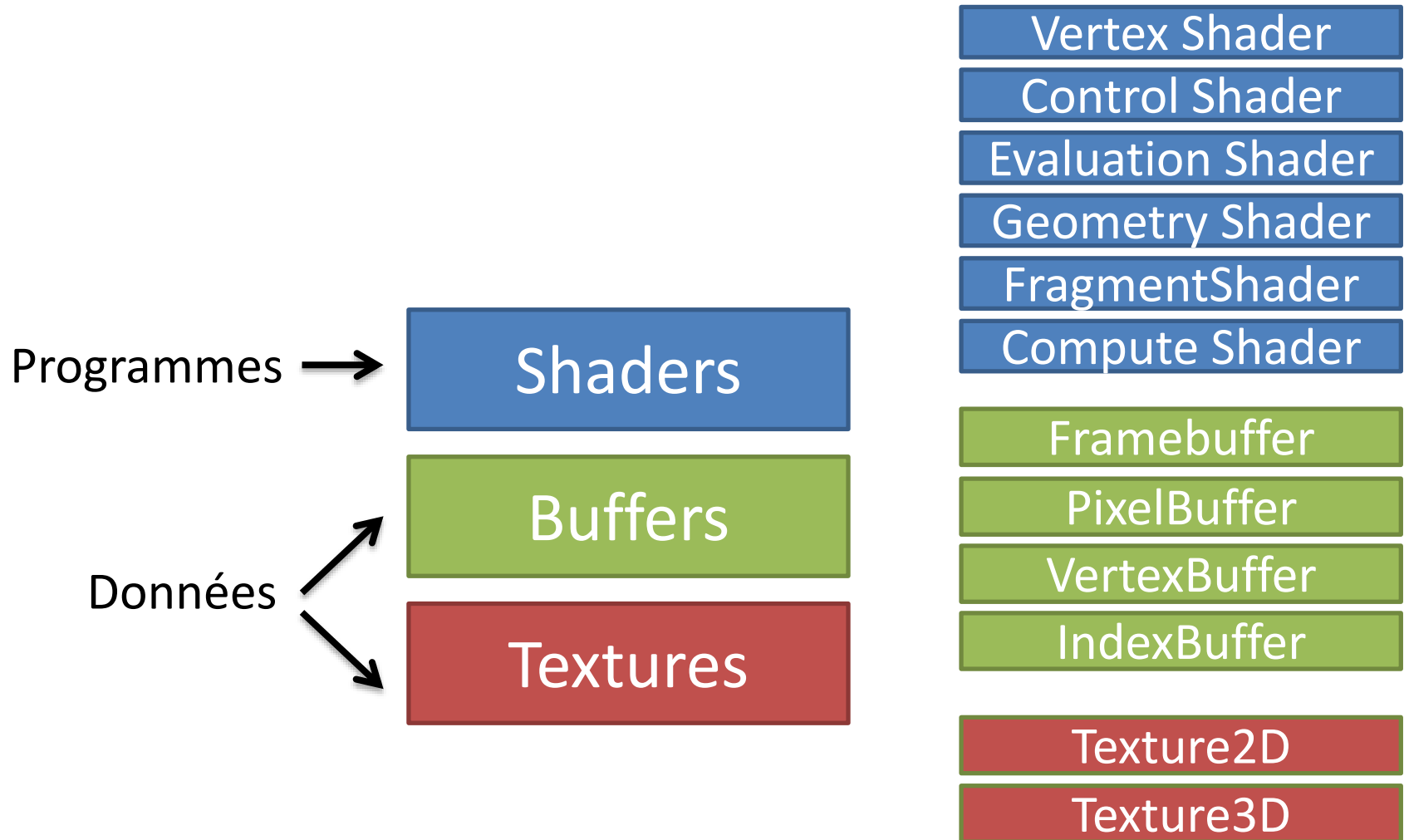
Maillage

```
//-----  
// GLSL Fragment Shader  
// for "Frank Miller's Shading"  
// Author: Tamy Boubekeur  
// Version: 0.01  
//-----  
  
varying vec4 P;  
varying vec3 N;  
  
void main (void) {  
    gl_FragColor = vec4 (0.0, 0.0, 0.0, 1);  
  
    int light = 0;  
  
    vec3 p = vec3 (gl_ModelViewMatrix * P);  
    vec3 n = normalize (gl_NormalMatrix * N);  
    vec3 c = gl_LightSource[light].position;  
    vec3 l = normalize (c - p);  
    vec3 r = reflect (-l, n);  
    vec3 v = normalize (-p);  
  
    float spec = max(dot(r, v), 0.0);  
    spec = pow (spec, 16.0);  
    spec = max (0.0, spec);  
  
    float sil = dot (v, n);  
  
    if (spec > 0.0 || sil < 0.3)  
        gl_FragColor = vec4 (1,1,1,1);  
}
```



Rendu

OpenGL Moderne (v4.x)



Vertex Shader (OpenGL v4.x)

```
Uniform mat4 u_matMVP;
```

```
in vec4 i_vPosition;
```

```
out vec4 v_vPosition;
```

```
void main () {
```

```
    // sans tessellation:
```

```
    gl_Position = u_matMVP * i_vPosition;
```

```
    // avec tessellation:
```

```
    o_vPosition = i_vPosition;
```

```
}
```


Tessellation Control Shader (OpenGL v4.x)

```
layout( vertices = 3 ) out;    // Sommets de contrôle
```

```
in vec4 v_vPosition[];
```

```
out vec4 tc_vPosition[];
```

```
uniform float u_fTessLevelInner;
```

```
uniform float u_fTessLevelOuter;
```

```
void main () {
```

```
    tc_vPosition[gl_InvocationID]=v_vPosition[gl_InvocationID];
```

```
    if (gl_InvocationID == 0 ) {
```

```
        gl_TessLevelInner[ 0 ] = u_fTessLevelInner;
```

```
        gl_TessLevelOuter[ 0 ] = u_fTessLevelOuter;
```

```
        gl_TessLevelOuter[ 1 ] = u_fTessLevelOuter;
```

```
        gl_TessLevelOuter[ 2 ] = u_fTessLevelOuter;
```

```
    }
```

```
}
```

Tessellation Evaluation Shader (OpenGL v4.x)

```
Layout (triangles, fractional_odd_spacing, ccw ) in;
```

```
in vec4 tc_vPosition[];
```

```
uniform mat4 u_matMVP;
```

```
void main () {
```

```
    vec3 vtePosition = vec3( 0.0 );
```

```
    // simple tessellation linéaire :
```

```
    vtePosition += gl_TessCoord.x * tc_vPosition[0].xyz;
```

```
    vtePosition += gl_TessCoord.y * tc_vPosition[1].xyz;
```

```
    vtePosition += gl_TessCoord.z * tc_vPosition[2].xyz;
```

```
    gl_Position = u_matMVP * vec4( vtePosition, 1 );
```

```
}
```

Geometry Shader (OpenGL v4.x)

```
#version 400
```

```
#extension GL_EXT_geometry_shader4 : enable
```

```
layout( triangles )           in;  
layout( triangle_strip, max_vertices = 3 ) out;
```

```
void main()  
{  
    for (int i = 0; i < 3; i++) {  
        gl_Position = gl_in[ i ].gl_Position;  
        EmitVertex();  
    }  
    EndPrimitive();  
}
```

Fragment Shader OpenGL v4.x)

```
#version 400
```

```
out vec4 FragColor;
```

```
void main()
```

```
{
```

```
    FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
```

```
}
```

Compute Shader (OpenGL v4.3)

CUDA	Compute Shader
<code>__syncthreads()</code>	<code>barrier()</code> <code>groupMemoryBarrier()</code>
<code>__shared__</code>	<code>shared</code>
<code>threadIdx</code>	<code>gl_GlobalInvocationID</code>
<code>blockIdx</code>	<code>gl_WorkGroupID</code>
<code>__threadfence()</code>	<code>memoryBarrierShared() (?)</code>

Compute Shader : simple copie

```
#version 430

#define TILE_WIDTH 16
#define TILE_HEIGHT 16

const ivec2 tileSize = vec2( TILE_WIDTH, TILE_HEIGHT );
layout( binding=0, rgba8 ) uniform image2D input_image;
layout( binding=1, rgba8 ) uniform image2D output_image;
layout( local_size_x = TILE_WIDTH,
        local_size_y = TILE_HEIGHT ) in;

void main()
{
    const ivec2 tile_xy = ivec2(gl_WorkGroupID);
    const ivec2 thread_xy = ivec2(gl_LocalInvocationID);
    const ivec2 pixel_xy = tile_xy*tileSize + thread_xy;
    vec4 pixel = imageLoad(input_image, pixel_xy);
    imageStore(output_image, pixel_xy, pixel);
}
```

Tampons GPU

- *Buffer Objects*
 - *Frame buffer*
 - *Pixel buffer (transactions CPU pour le calcul en flux)*
 - *Texture buffer (accès de haut niveau)*
 - *Vertex buffer*
 - *Géométrie*
 - *Topologie*

Vertex Buffer (VBO)

- Stock la géométrie des objets
- Liste d'attributs par sommet
- Topologie
 - Vue API : triangle, quads, lignes, etc
 - Sous le drivers : « triangle strips »
- Cache critique
 - Réordonnancement de maillages [Sanders 2006]
 - Compression

Vertex Buffer (VBO)

- Création à l'initialisation du programme
- Appel dans la boucle de rendu
- Entrelacé: [nxnynz0,rgba0, xyz0,nxnynz1, ...]
- Ou multi-buffer (un par attribut)

Framebuffer (FBO)

- Tableau/image 2D
- Sortie du pipeline GPU
- Contexte de la programmation GPU
 - FBOs non affichés
 - Rendu multi-passes et calcul multipass
 - Tampon de stockage intermédiaire
 - Calcul de chaque passe encapsulé dans les shaders
 - *1 pixel = 1 thread*

Framebuffer (FBO)

Concept OpenGL:

```
Texture          g_ActiveTextures[8];
```

```
struct FBO {  
    Texture          m_Stencil;  
    Texture          m_Depth;  
    Texture          m_RenderTargets[8];  
};
```

C++:

```
Texture tex;
```

```
/*... charger les données */
```

```
int iTextureSlot = 0;
```

```
glActiveTexture (GL_TEXTURE0 + iTextureSlot);
```

```
tex.Use ();
```

```
m_Program.SetUniform1i (iTexLocation, iTextureSlot);
```

```
iTextureSlot++;
```

Ressources OpenGL

- Une vaste communauté en ligne
- Une référence : www.opengl.org
- Quelques programme d'exemples sur mon site
 - Notamment gmini : gmini.googlecode.com

Environnement Graphique

- GLUT, SDL, Qt, etc
- La plupart sont simples et multiplateforme
 - GLUT : pour des applications minimales
 - AntTweakBar : interface graphique pour GLUT
 - Qt : besoin d'une interface graphique avancée
 - GLWidget : objet spécifique pour l'affichage OpenGL
 - LibQGLViewer : surcouche Qt pour la création rapide d'application 3D interactives

Middleware

- Environnement complet de création d'applications 3D interactives
- Au-delà du rendu : physique, son, interaction avancée, RV, réseau, etc
- Moteurs de jeux : Unreal Engine, CryEngine, Unity, Ogre
- Environnement généraliste : VTK,

Alternatives à OpenGL

- DirectX : plateformes Windows
- Optix : plateformes Nvidia, rendu par lancer de rayon (cf IGR202)
- Metal : plateformes Apple

WEBAPP 3D

WebGL

- **OpenGL en partie accessible en javascript pour les webapp 3D**
- S'insère naturelle dans une webapp modern
 - html5/css3/javascript
 - Compatible avec la majorité des navigateurs web, y compris les navigateurs mobile
- Fonctionnalités correspondant à un sous-ensemble d'OpenGL ES
- Pas de mode immédiat
 - Nécessité d'utiliser des *shader* type OpenGL 2.1+
 - Pas de `glBegin () ... glEnd ()`, utilisation de tampons géométriques à la place (*vertex buffers*)

Three.js

- Bibliothèque construite au dessus de WebGL, permettant de rapidement programmer des applications web 3D.
- Voir par exemple : <http://www.telecom-paristech.fr/~boubek/webapps/gmini/>
- Très simple d'utilisation, beaucoup d'exemples disponibles.

WebApp 3D

- Structure et squelette similaire à une WebApp classique
- Execution limitée par l'environnement (le navigateur)
- Accès aux webservices
- Cf. : SketchFab, TinkerCAD, SculptGL

Références

- OpenGL Red Book *Informatique Graphique:
(IGR202/M1)*
- GPU Gems Séries
- GPU Pro Series *Informatique Graphique Avancée
(IGR M2 / SI960 / Master IMA)*
- developer.nvidia.com
- developer.amd.com *Nuage de Points et Modélisation
3D*
- Modern OpenGL *(IGR M2 / SI954 / Master MVA)*
- Unity.com
- Three.js

Page du module

<http://www.telecom-paristech.fr/~boubek/ens/igr/app>