HMIN 317 – Moteur de Jeux

API RENDU OPENGL

Université Montpellier 2

Rémi Ronfard

remi.ronfard@inria.fr

https://team.inria.fr/imagine/remi-ronfard/

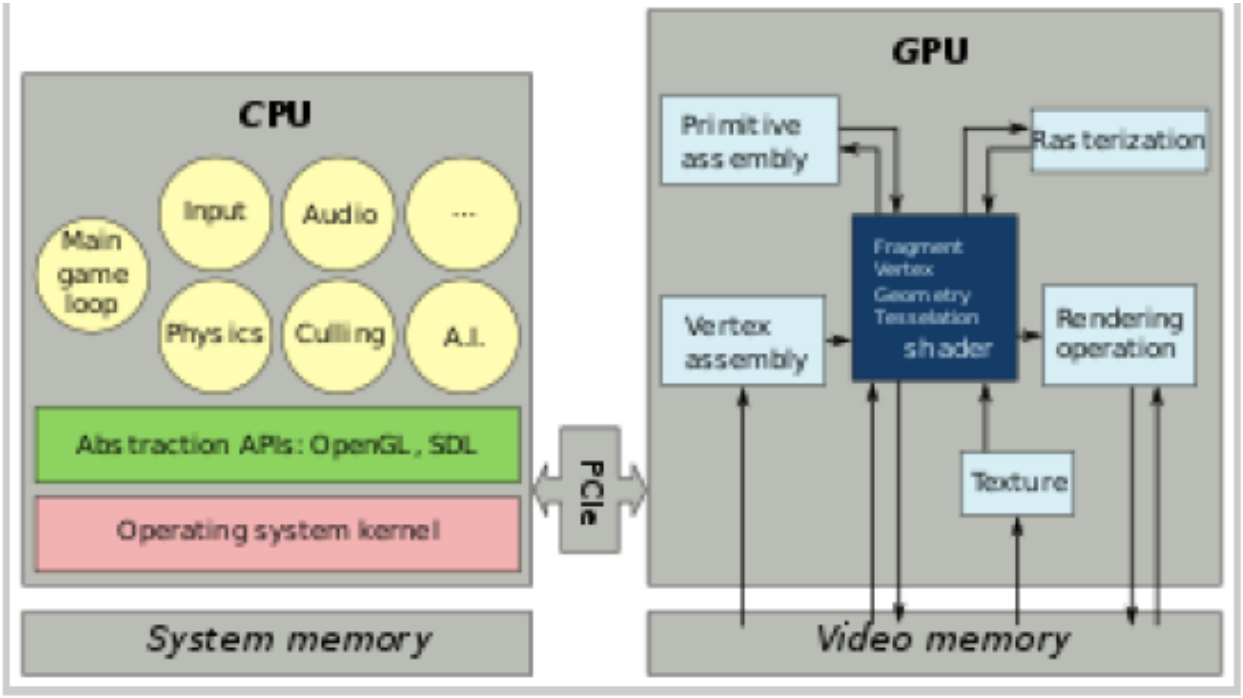
Ce cours est très largement inspiré des cours de ***Marc Moulis*** *et* ***Benoit Lange***.

Le but de cette présentation est de fournir une vue globale du fonctionnement des A.P.I. de rendu temps réel standards, ainsi que donner des points d'entrée concrets dans la documentation (OpenGL).

Qu'est ce qu'une A.P.I. ?

A.P.I. : Application Programming Interface (interface de programmation d'application). Ensemble des structures et fonctions permettant au programmeur le développement de tout ou partie d'applications logicielles.

Qu'est ce qu'une A.P.I. de rendu ?



C'est la couche logicielle qui permet de contrôler le matériel de rendu graphique: ensemble de procédures et fonctions qui permettent au programmeur de spécifier les opérations et objets nécessaires à la génération d'images, représentant en général des objets tridimensionnels.

- Quelle est la différence avec un moteur de rendu ?

- Le moteur de rendu est situé à une couche logicielle supérieure, et permet généralement la gestion d'une scène tridimensionnelle complexe avec tous ses acteurs. Le moteur de rendu utilise une API de rendu comme partie du processus de génération d'une image.

- Il ne faut pas confondre API de rendu graphique et API applications.
  - API d'application couche haut niveau

Sur consoles

Avant, des API propriétaires du fabricant, fournies avec les kits de développement. Elles offrent un accès complet aux capacités du matériel.

Maintenant, des API standards (OpenGL, Directx)

Sur PC

Direct3D : API de Microsoft, partie de DirectX (qui contient également des A.P.I. de gestion d'I/O, de son, de réseau). Disponible également sur XBox. Actuellement en version 11.

OpenGL : Standard multiplateformes. Les spécifications du standard sont publiques et gratuites. Actuellement en version 4.3 (Aout 2012). Historiquement développé par Silicon Graphics (IrisGL), puis repris en 2006 par le consortium Khronos, qui réunit tous les grands acteurs du marché.

Mantle: un nouvel API de RENDU bas niveau, développé par AMD

Vulkan

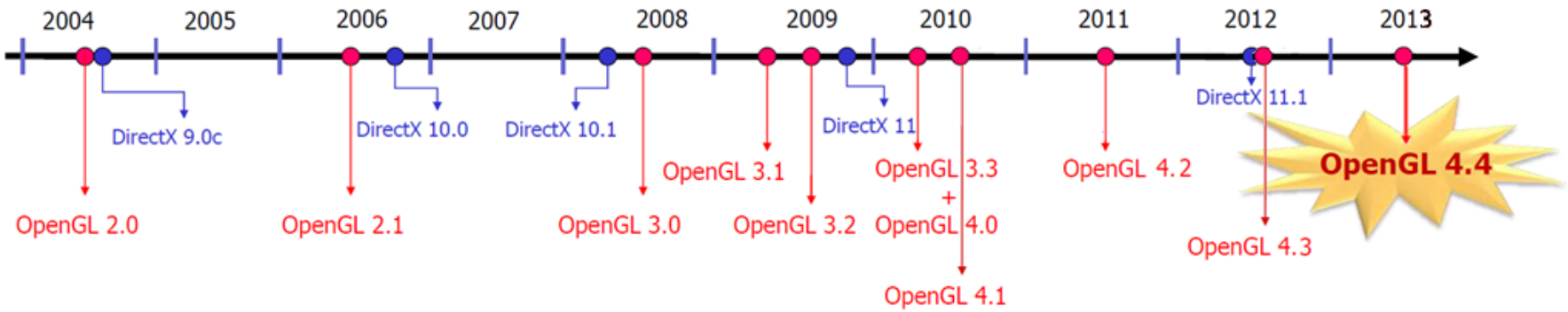
Vulkan: nouvelle version de Mantle, cross platform et nouvelle génération d'openGL.

OpenGL	Vulkan[24]
one single global state machine	object-based with no global state
state is tied to a single context	all state concepts are localized to a command buffer
GPU memory and synchronization are usually hidden	explicit control over memory management and synchronization
extensive error checking	Vulkan drivers do no error checking

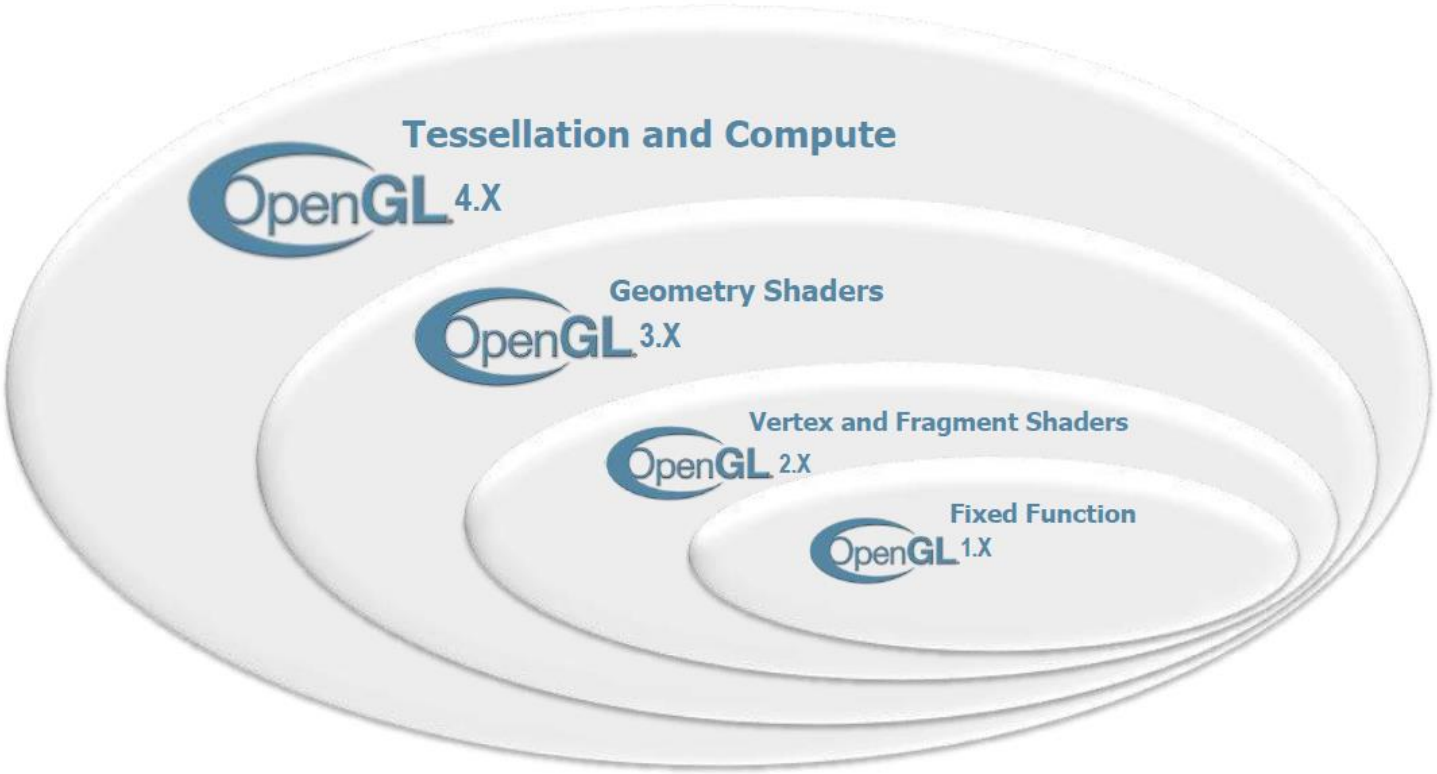
Introduction à OpenGL

- Une A.P.I. de rendu est un pipeline de traitement des données, dont certains étages sont programmables, et d'autres sont contrôlés par des variables d'état.
-
-
-
- La configuration du pipeline permet de contrôler les opérations nécessaires à la génération d'une image.
-
-
-
- Au fil des versions, OpenGL et Direct3D sont devenus de plus en plus proches du matériel, pour améliorer flexibilité, performances, et coller au mieux aux évolutions rapides du matériel.
-
-

- Evolution des versions de Direct3D et OpenGL. Les versions sont de plus en plus fréquentes pour coller au mieux aux évolutions du matériel



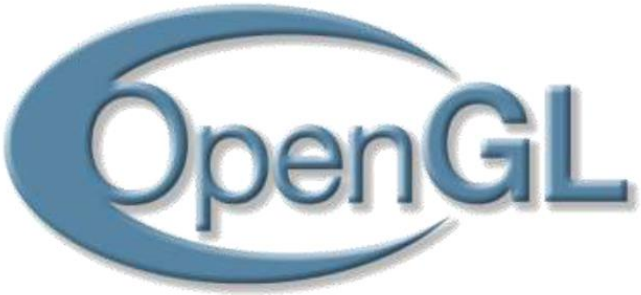
- Evolution des versions d'OpenGL. On retrouve l'apparition des shaders, puis des fonctionnalités GPGPU (OpenCL).



- Les descriptions techniques qui vont suivre sont basées sur l'A.P.I. OpenGL pour les raisons suivantes:

- OpenGL dans ses versions < 3.x permet un prototypage plus rapide que DirectX
- OpenGL est un standard multiplateformes, que l'on retrouve sur de nombreuses machines différentes (PC, téléphone, etc)

- Néanmoins, la méthodologie est très similaire à Direct3D, et il reste relativement aisé d'utiliser l'une ou l'autre A.P.I. à partir du moment ou les principes de fonctionnement sont assimilés.



- Le programmeur contrôle OpenGL par le biais d'un contexte. Ce **contexte** est créé à la demande du programmeur par le driver matériel, et représente la machine à états OpenGL.

- Un **contexte** doit en premier lieu être associé à une « **vue** » du système graphique de la plateforme (en général une fenêtre). On utilise pour cela des fonctions système dédiées (WGL sous Windows, GLX sous X11), qui font la passerelle entre le système graphique de l'OS et l'implémentation OpenGL.

- Le programmeur définit une zone dans la vue, le **viewport**, qui représente la zone de sortie de l'image (en général la vue entière).

- Le programmeur **paramètre** les états de rendu, et génère des **primitives géométriques**.

- **L'image** générée est mise à disposition au travers du contexte, ou directement affichée dans le viewport.

- Que peut-on faire avec OpenGL:
  - Dessiner des primitives (point, ligne, polygon)
  - Appliquer des opérations matricielles
  - Cacher des surface (Z-buffer)
  - Réaliser des illuminations, ombrages
  - Afficher des textures
  - Opérations sur les pixels
  - Du calcul sur GPU

Attention au choix des axes !

- Different game engines define coordinate systems differently

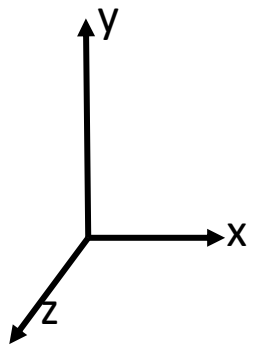
  Most of you will probably use the OpenGL coordinate system

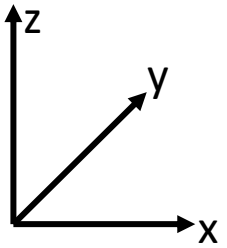
- TAs will strive to be coordinate-system independent
- “Horizontal plane”

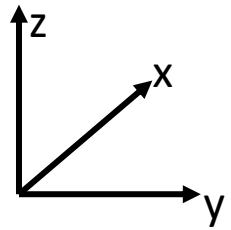
  Plane parallel to the ground (in OpenGL, the xz-plane)

- “Up-axis”

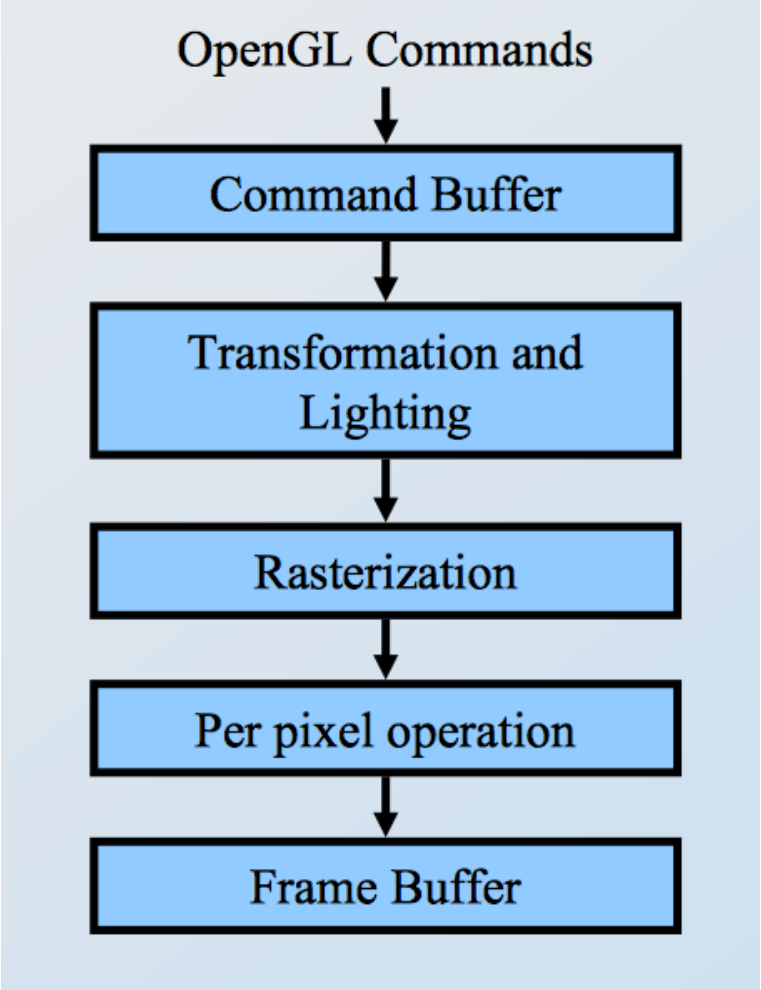
  Axis perpendicular to horizontal plane (in OpenGL, the y-axis)







- Et qu'est-ce que OpenGL ne fournit pas ?
  - Un système de gestion d'évènements
  - IO utilisateurs
  - La gestion de scène
  - Des fonction hauts niveau pour la gestion des modèles 3D

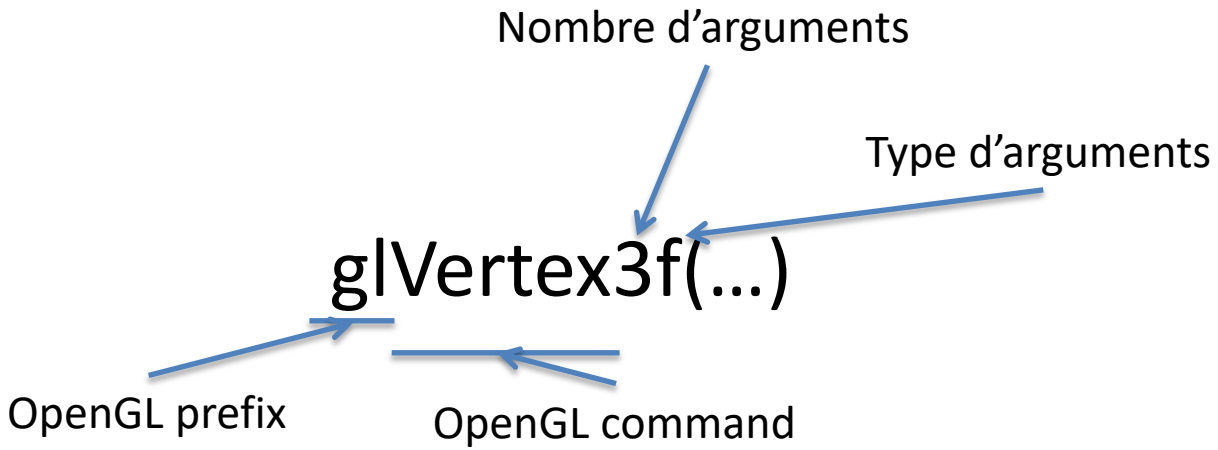

OpenGL Commands
Command Buffer
Transformation and Lighting
Rasterization
Per pixel operation
Frame Buffer

- OpenGL est une machine à état
  - Les changements de valeurs sont assignés jusqu'à un prochain changement de valeur
  - Chaque variable d'état ou mode a une valeur par défaut

Les primitives OpenGL

- Les commandes OpenGL utilisent le préfixe ***gl.***
- La première lettre suivante est en Majuscule.

- Les constantes globales OpenGL commencent par : ***GL_***



- Types de données

OpenGL Data Type	Internal Representation	Defined as C Type	C Literal Suffix
GLbyte	8-bit integer	Signed char	b
GLshort	16-bit integer	Short	s
GLint, GLsizei	32-bit integer	Long	l
GLfloat, GLclampf	32-bit floating point	Float	f
GLdouble, GLclampd	64-bit floating point	Double	d
GLubyte, GLboolean	8-bit unsigned integer	Unsigned char	ub
GLushort	16-bit unsigned integer	Unsigned short	us
GLuint, GLenum, GLbitfield	32-bit unsigned integer	Unsigned long	ui

- De manière courante, les cartes graphiques ne sont capables d'afficher que des triangles 2D. Ceux-ci sont issus de la projection des polygones représentant la surface des objets de notre scène 3D.

- Chacun des triangles envoyés à la carte graphique est porteur de toute une série d'attributs (appelés méta-données) associés aux sommets, qui vont permettre de caractériser le triangle:
  - Position projetée à l'écran et profondeur des sommets
  - Normales aux sommets
  - Couleurs aux sommets
  - Coordonnées de textures aux sommets

- Ces méta-données sont fournies à l'API au moment de la description des sommets (vertex, pluriel vertices) qui composent les triangles. Pour des raisons de performances, les triangles sont regroupés en « paquets », appelés primitives. Il existe plusieurs types de primitives géométriques.

- NB: La spécification de primitives OpenGL « à la volée » par le biais des A.P.I. glBegin()/glEnd() n'est plus supportée à partir de la version 3.0.

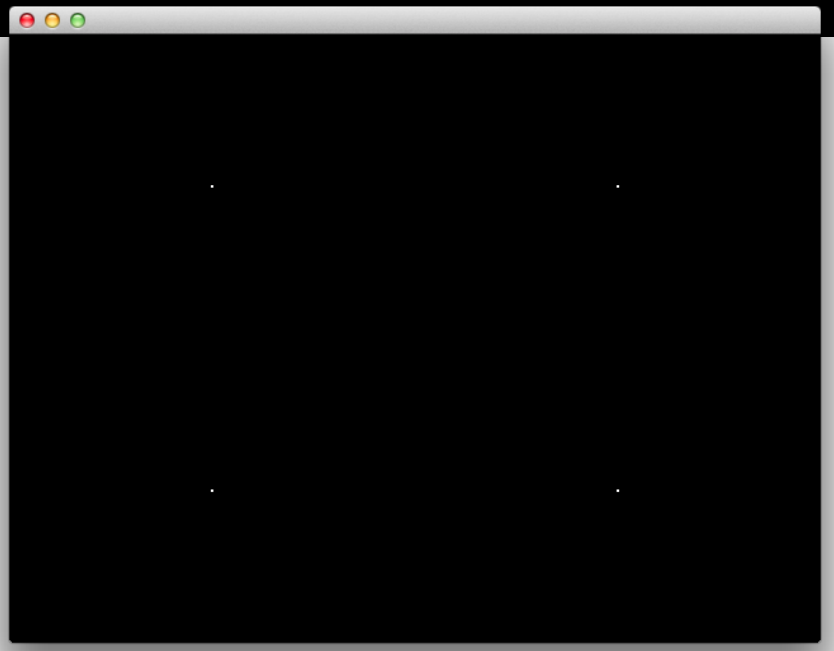
glVertex3f(), glColor4f(), glNormal3f(), glTexCoord2f()

Liste de points

- Primitive la plus simple. Chaque vertex spécifié représente un point unique.
- Certaines options de rendu permettent de spécifier la taille des points affichés à l'écran, voire même d'afficher un quadrilatère 2D texturé à la position du point projeté (billboards ou sprites).
- Rendement : N points = N vertices

glBegin(GL_POINTS)

- Un exemple:
  - Dans la boucle de rendu :

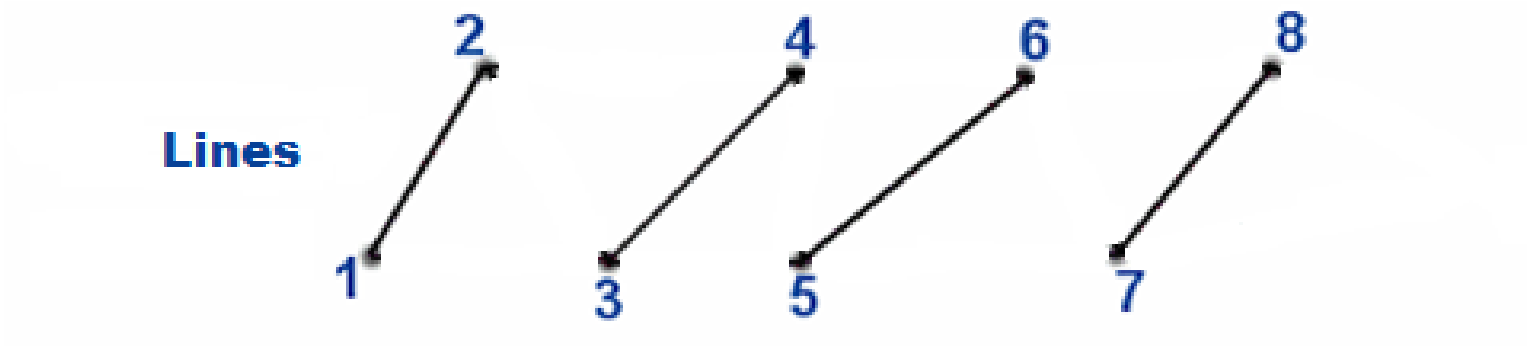


```
glBegin(GL_POINTS);
glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    glVertex3f(0.0f,1.0f, 0.0f);
glEnd();
```

Liste de segments

Chaque paire de vertices représente un segment de droite.

- Certaines options de rendu permettent de spécifier l'épaisseur des lignes tracées.
- Rendement : N segments = 2N vertices



glBegin(GL_LINES)

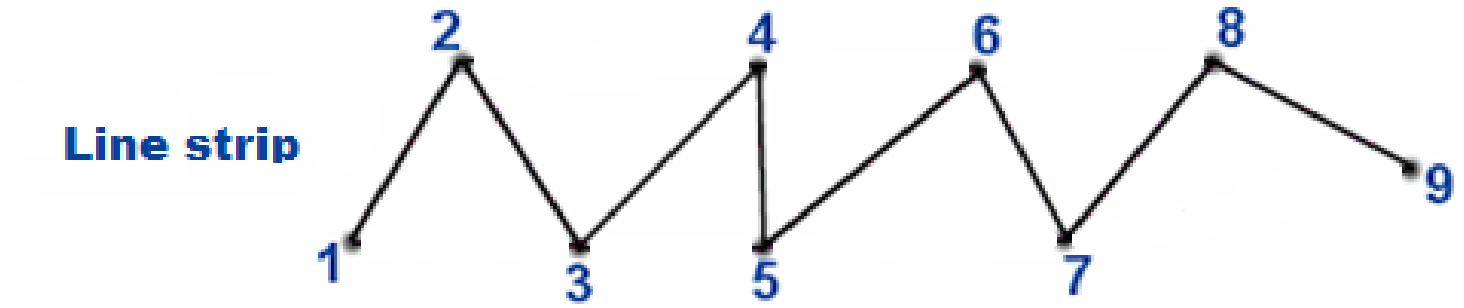
- Un exemple:
  - Dans la boucle de rendu :

```
glBegin(GL_LINES);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    glVertex3f(0.0f,1.0f, 0.0f);
glEnd();
```

Ligne brisée (line strip)

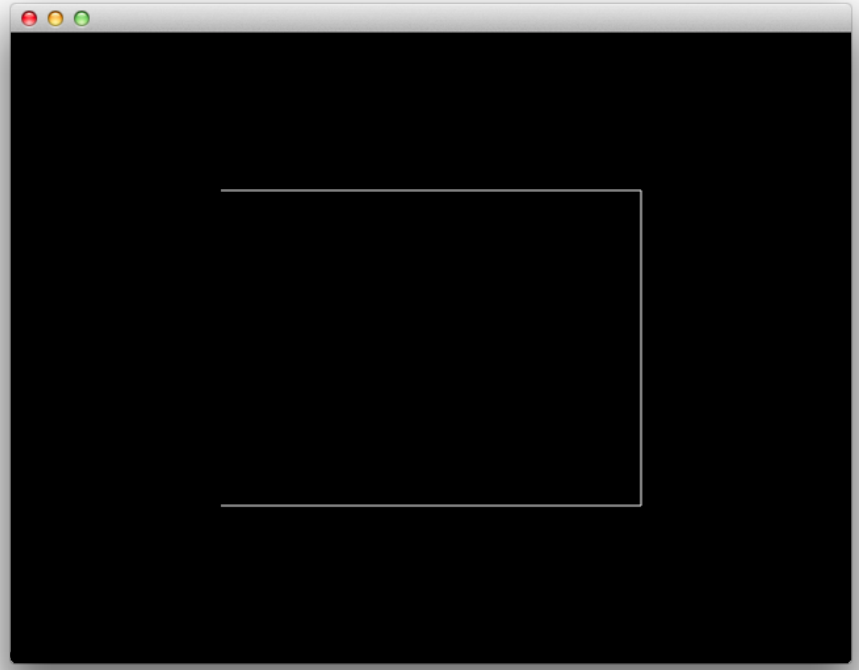
Les vertices sont reliés deux à deux pour former une ligne brisée entre le premier et le dernier vertex.

- Certaines options de rendu permettent de spécifier l'épaisseur des lignes tracées.
- Rendement : N segments = N-1 vertices



glBegin(GL_LINE_STRIP)

- Un exemple:
  - Dans la boucle de rendu :

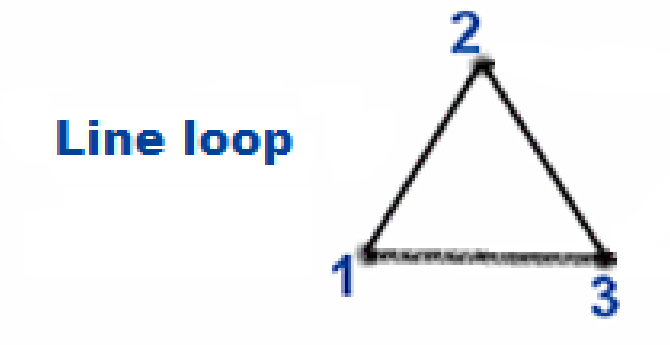


```
glBegin(GL_LINE_STRIP);
glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    glVertex3f(0.0f,1.0f, 0.0f);
glEnd();
```

Ligne brisée fermée (line loop)

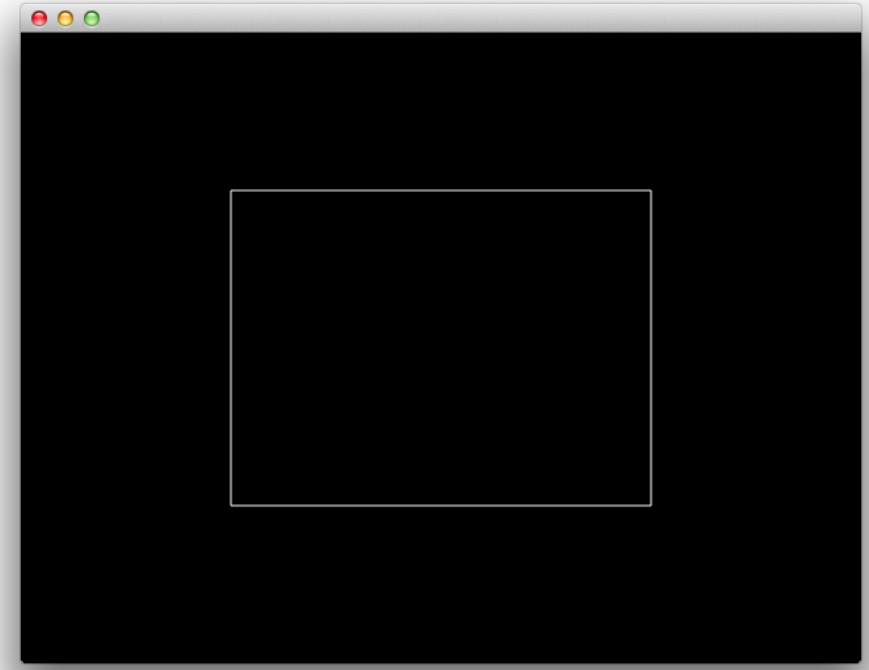
Les vertices sont reliés deux à deux pour former une ligne brisée entre le premier et le dernier vertex. Le dernier vertex est automatiquement relié au premier vertex.

- Certaines options de rendu permettent de spécifier l'épaisseur des lignes tracées.
- Rendement : N segments = N vertices



glBegin(GL_LINE_LOOP)

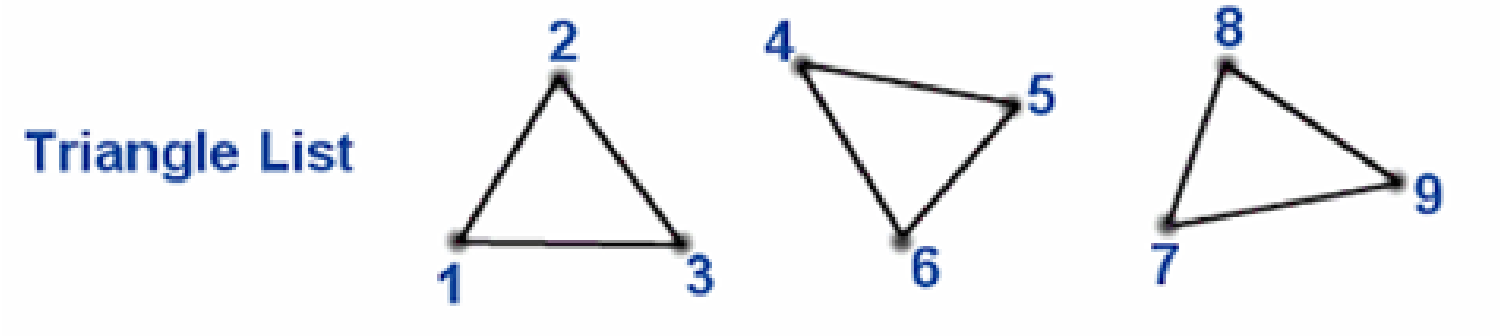
- Un exemple:
  - Dans la boucle de rendu :



```
glBegin(GL_LINE_LOOP);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 1.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
glEnd();
```

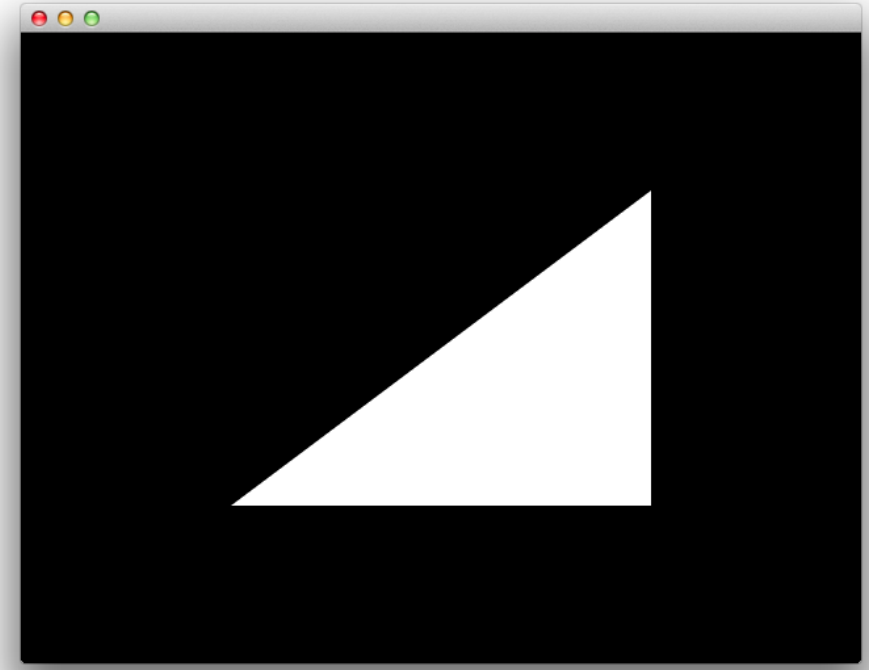
Liste de triangles

- Chaque triplet de vertices définit un triangle indépendant.
- Certaines options de rendu permettent de définir le mode de remplissage ou encore l'élimination automatique des faces arrières (backface culling).
- Rendement : N triangles = 3N vertices



glBegin(GL_TRIANGLES), glCullFace(), glPolygonMode()

- Un exemple:
  - Dans la boucle de rendu :

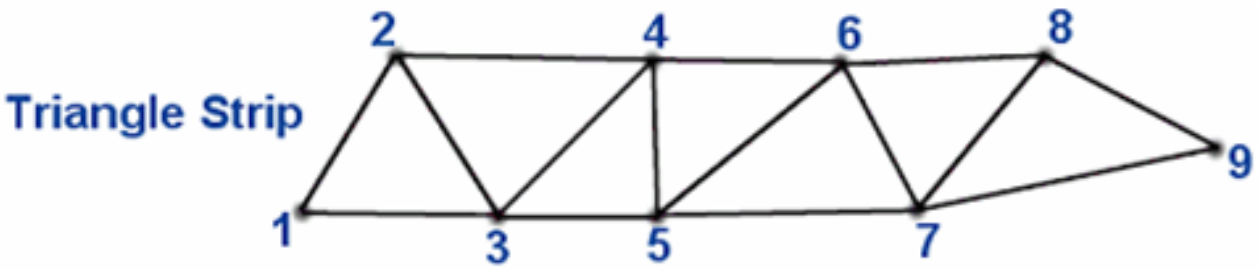


```
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    //glVertex3f(0.0f,1.0f, 0.0f);
glEnd();
```

Bande de triangles (triangle strip)

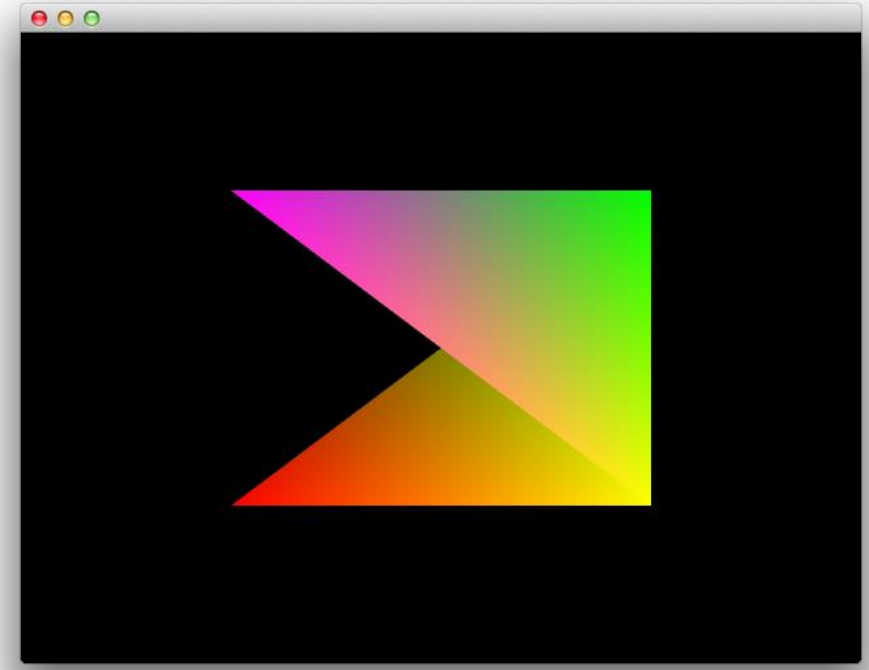
Définit un groupe de triangles connectés. Chaque vertex supplémentaire forme un triangle avec les 2 vertices précédents.

- Ce type de primitive est à privilégier dès que possible par rapport aux listes de triangles, pour des raisons de performances (partage des vertices).
- Il est possible de concaténer 2 bandes non jointives en une seule primitive, en dupliquant le dernier vertex de la première bande et le premier vertex de la seconde (génération de triangles dégénérés).
- Rendement : N triangles = N+2 vertices



glBegin(GL_TRIANGLE_STRIP), glCullFace(), glPolygonMode()

- Un exemple:
  - Dans la boucle de rendu :

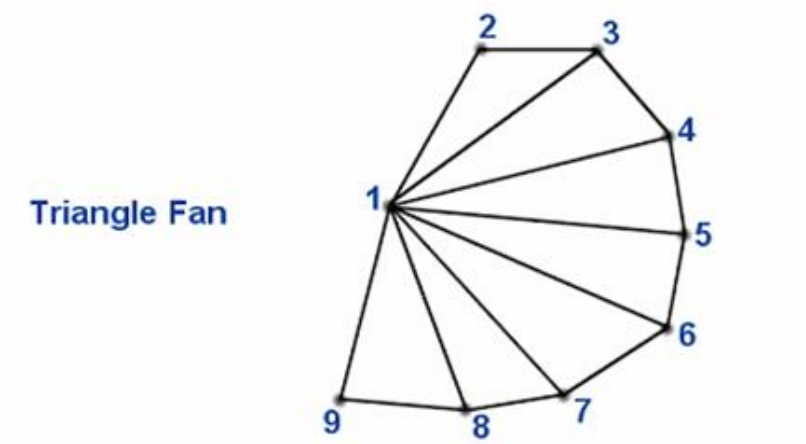


```
glTranslatef(-0.5f,-0.5f,0.0f);
glBegin(GL_TRIANGLE_STRIP);
      glColor3f(1.0f, 0.0f, 0.0f);
     glVertex3f(0.0f, 0.0f, 0.0f);
     glColor3f(1.0f, 1.0f, 0.0f);
     glVertex3f(1.0f, 0.0f, 0.0f);
     glColor3f(0.0f, 1.0f, 0.0f);
     glVertex3f(1.0f, 1.0f, 0.0f);
     glColor3f(1.0f, 0.0f, 1.0f);
     glVertex3f(0.0f,1.0f, 0.0f);
 glEnd();
```

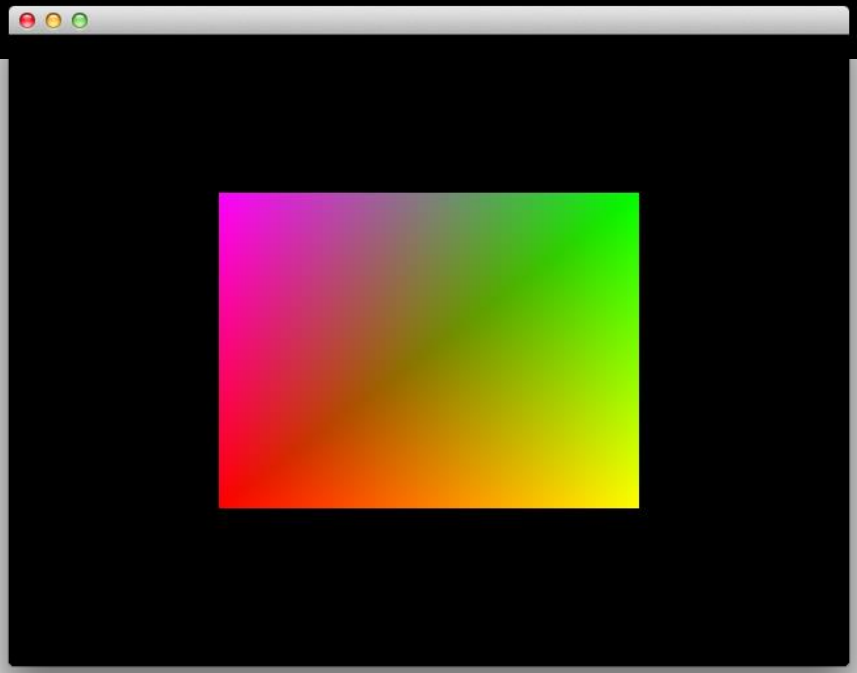
Triangles en éventail (triangle fan)

Définit un groupe de triangles connectés. Chaque vertex supplémentaire forme un triangle avec le tout premier vertex et le vertex précédent.

- Ce type de primitive est également à privilégier dès que possible par rapport aux listes de triangles, toujours pour des raisons de performances.
- Rendement : N triangles = N+2 vertices



glBegin(GL_TRIANGLE_FAN), glCullFace(), glPolygonMode()



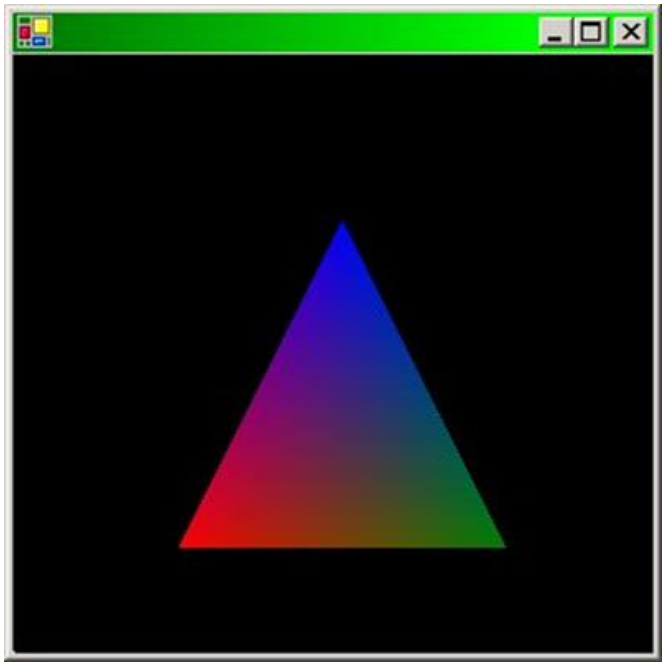
- Un exemple:
  - Dans la boucle de rendu :

```
glBegin(GL_TRIANGLE_FAN);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    glColor3f(1.0f, 0.0f, 1.0f);
    glVertex3f(0.0f,1.0f, 0.0f);
glEnd();
```

- Pour le rendu effectif des primitives (lignes, triangles), appelé rasterization, la carte interpole pour chaque pixel calculé les méta-données (couleur, coordonnées de texture, profondeur) entre chacun des vertex du triangle.

- Le rendu polygonal approxime uniquement la surface des objets représentés : les objets sont creux !

- Il existe des méthodes de visualisation de données volumiques (volume rendering), notamment à l'aide de textures 3D.



Mode immédiat

Permet de spécifier les primitives « à la volée » par le biais des A.P.I. de description.

- Globalement peu performant (nécessite un appel de fonction pour le passage de chaque nouvelle métadonnée), il reste néanmoins extrêmement pratique pour réaliser du prototypage.

- NB: Comme indiqué précédemment, ce mode est obsolète depuis OpenGL 3.0.

Geometry arrays

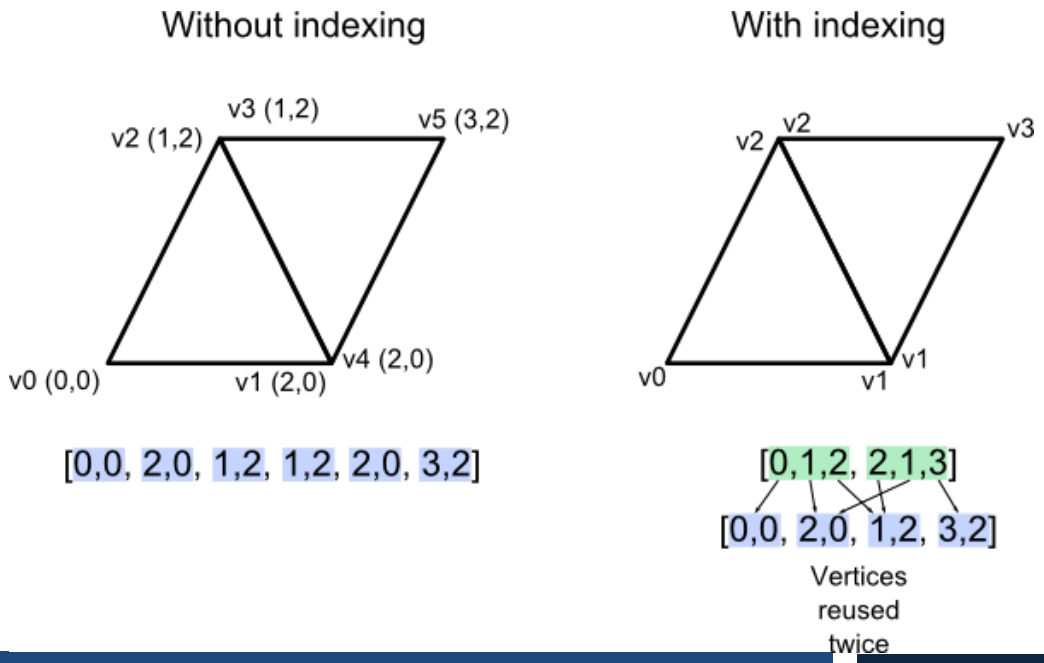
Permet de stocker les données des primitives dans des tableaux en mémoire centrale.

- Grande souplesse d'exécution (on ne spécifie que les paramètres nécessaires) et bonnes performances (un seul appel déclenche le rendu).

- Obsolète à partir d'OpenGL 3.0.

- Les geometry arrays introduisent la notion de primitive indexée: en plus des tables contenant la description des vertices (couleur, coordonnées, etc), une seconde table peut optionnellement être spécifiée, contenant des entiers qui sont des indices dans la table des vertices. Ce sont alors les indices qui décrivent la primitive géométrique, et plus l'ordre des vertices.

- Ce système permet de partager les vertices entre plusieurs primitives, donc d'augmenter la bande passante.



Geometry buffers

Les geometry arrays ont évolué en geometry buffers, directement embarqués en VRAM. Le mode de fonctionnement est calqué sur le fonctionnement du matériel.

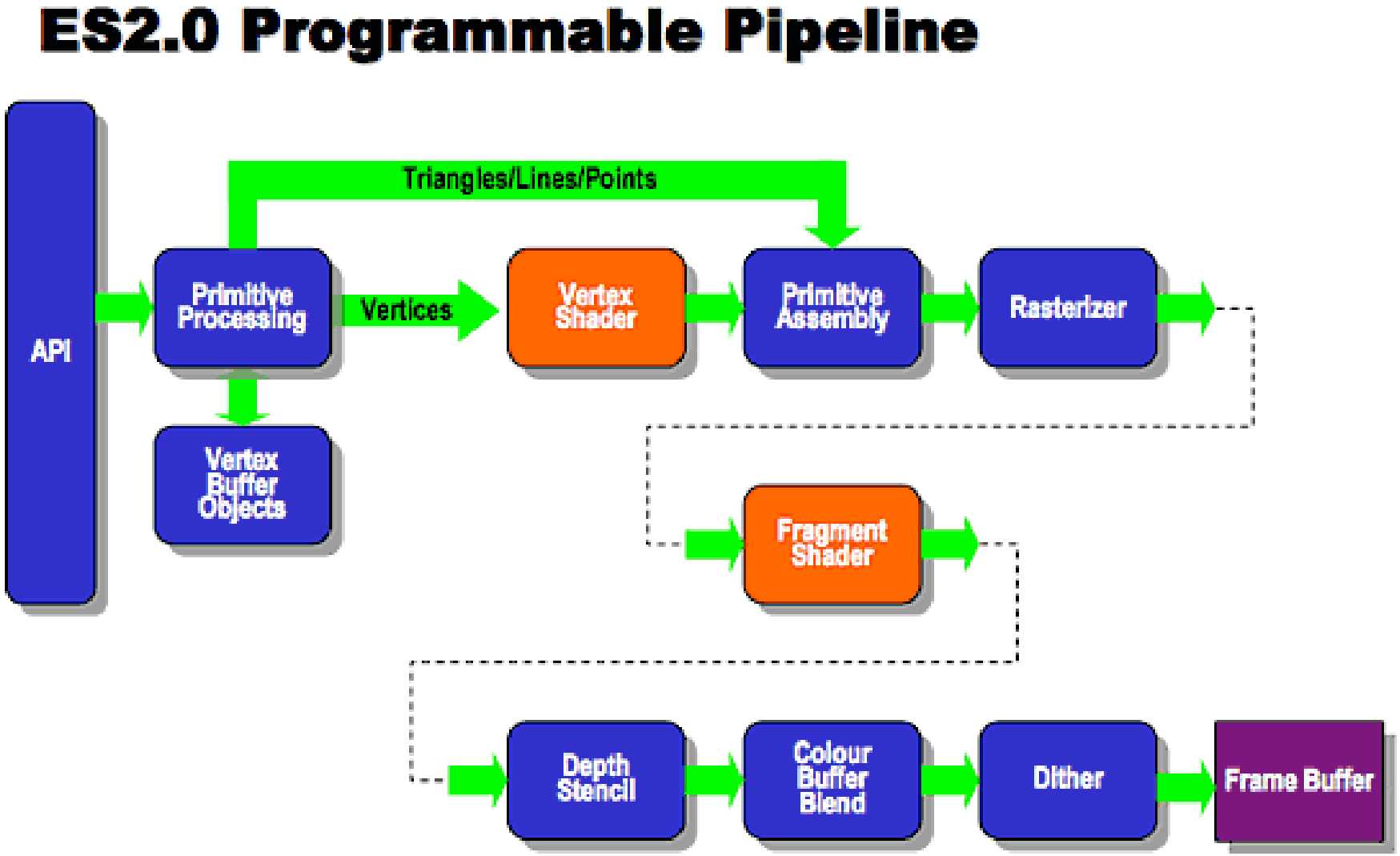
- Techniquement, on distingue les vertex buffers (VBO), qui contiennent les données des vertices, et les index buffers (IBO), qui contiennent l'indexation des vertices. Le programmeur spécifie précisément à l'A.P.I. quelles sont les méta-données présentes dans les buffers.

- Pour des performances optimales, on parallélise l'écriture des données dans le geometry buffer avec la lecture par la carte de ces données (pas de zones de chevauchement des primitives dans le buffer pendant son utilisation).

- Autres primitives utiles au rendu :
  - glFlush()
    - Forcer la commande précédente OpenGL à démarrer une exécution
  - glFinish()
    - Forcer toutes les commandes précédentes à terminer.

- Et bien d'autres …
  - Doc OpenGL

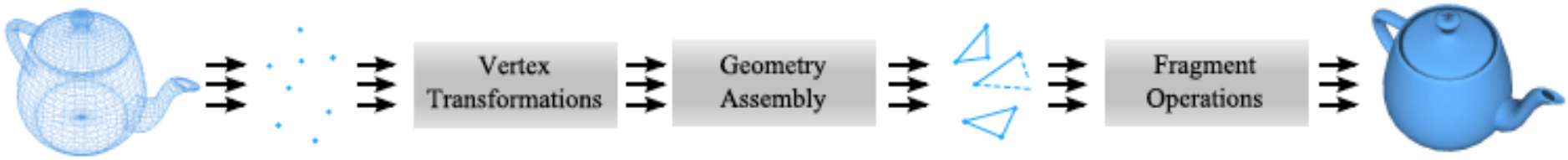
Pour les TP et les mini-projets, nous recommandons OPENGL ES 2.0



Exemple OPENGL ES 2.0 (TP1)

- glViewport(0, 0, width, height,scale);
- glClear(GL_COLOR_BUFFER_BIT);
- glVertexAttribPointer(m_posAttr, 2, GL_FLOAT, GL_FALSE, 0, vertices);
- glVertexAttribPointer(m_colAttr, 3, GL_FLOAT, GL_FALSE, 0, colors);
- glEnableVertexAttribArray(0);
- glEnableVertexAttribArray(1);
- glDrawArrays(GL_TRIANGLE_FAN, 0, 6);

OpenGL ES 2.0 and GLSL Shaders



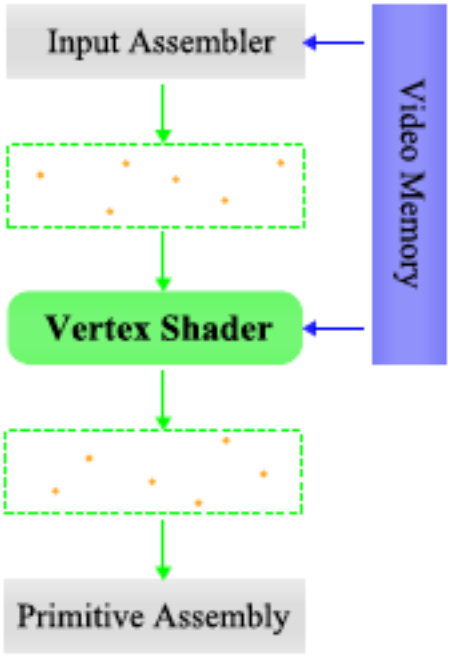


Figure 4. Pipeline Vertex Shader.

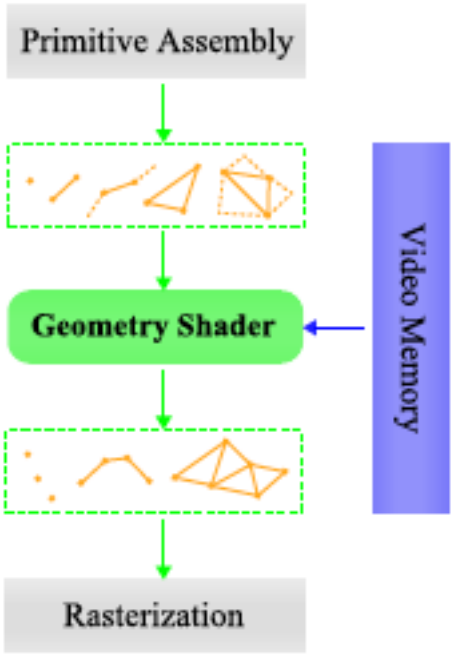


Figure 5. Pipeline Geometry Shader.

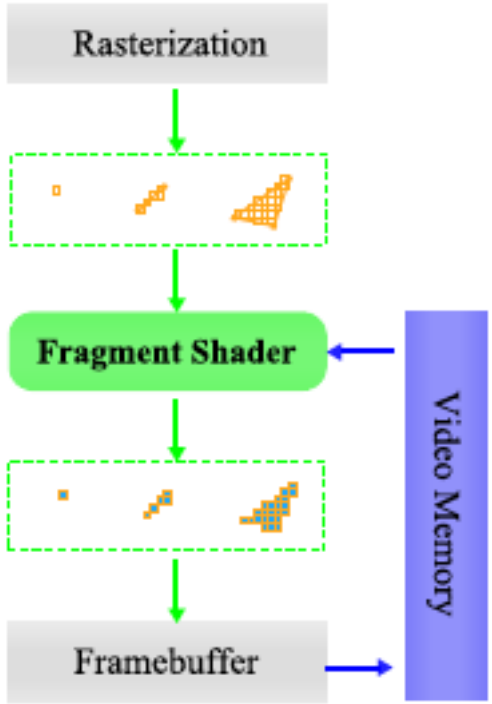
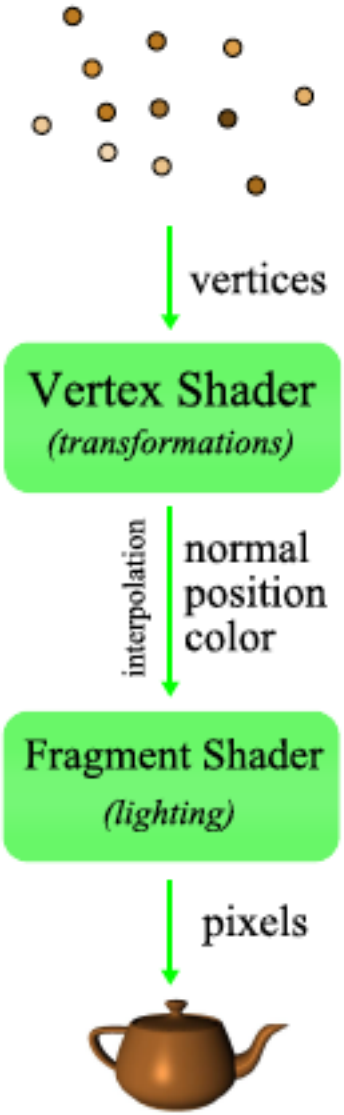


Figure 6. Pipeline Fragment Shader.

OpenGL ES 2.0 and GLSL Shaders



Vertex shader summary

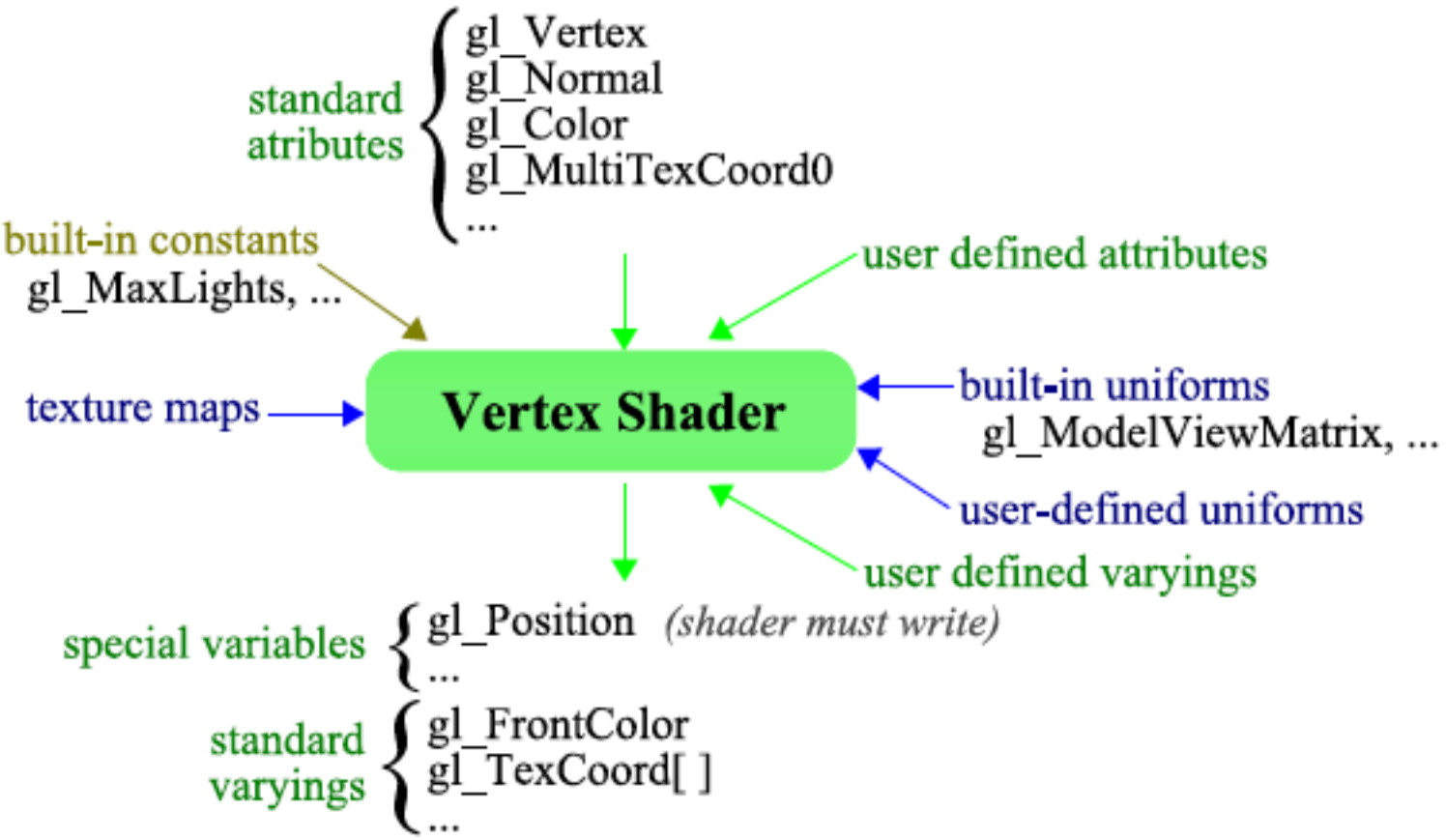


Figure 16. Input/Ouput summary of the vertex shader.

Geometry shader summary

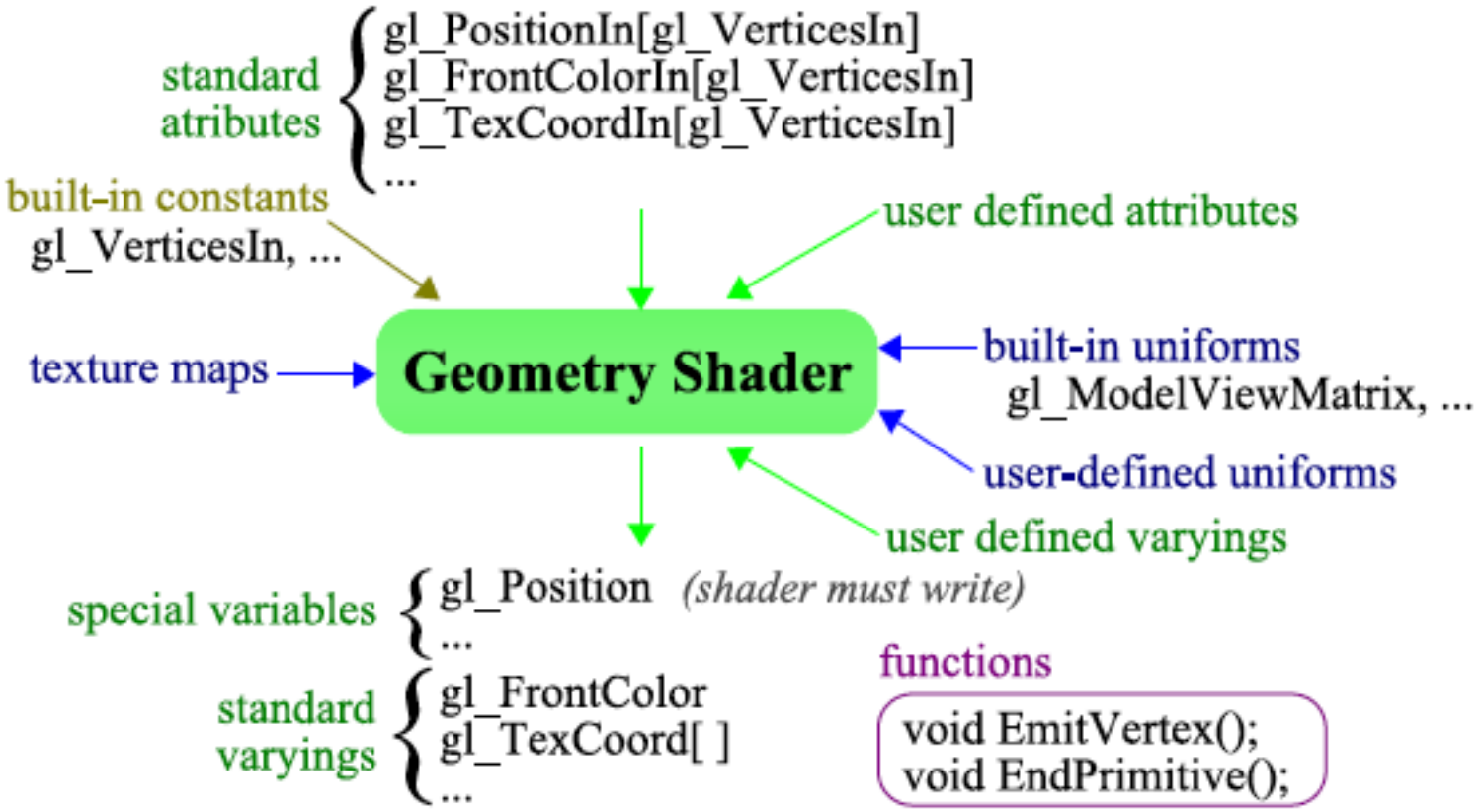


Figure 17. Input/Ouput summary of the geometry shader.

Pixel (fragment) shader summary

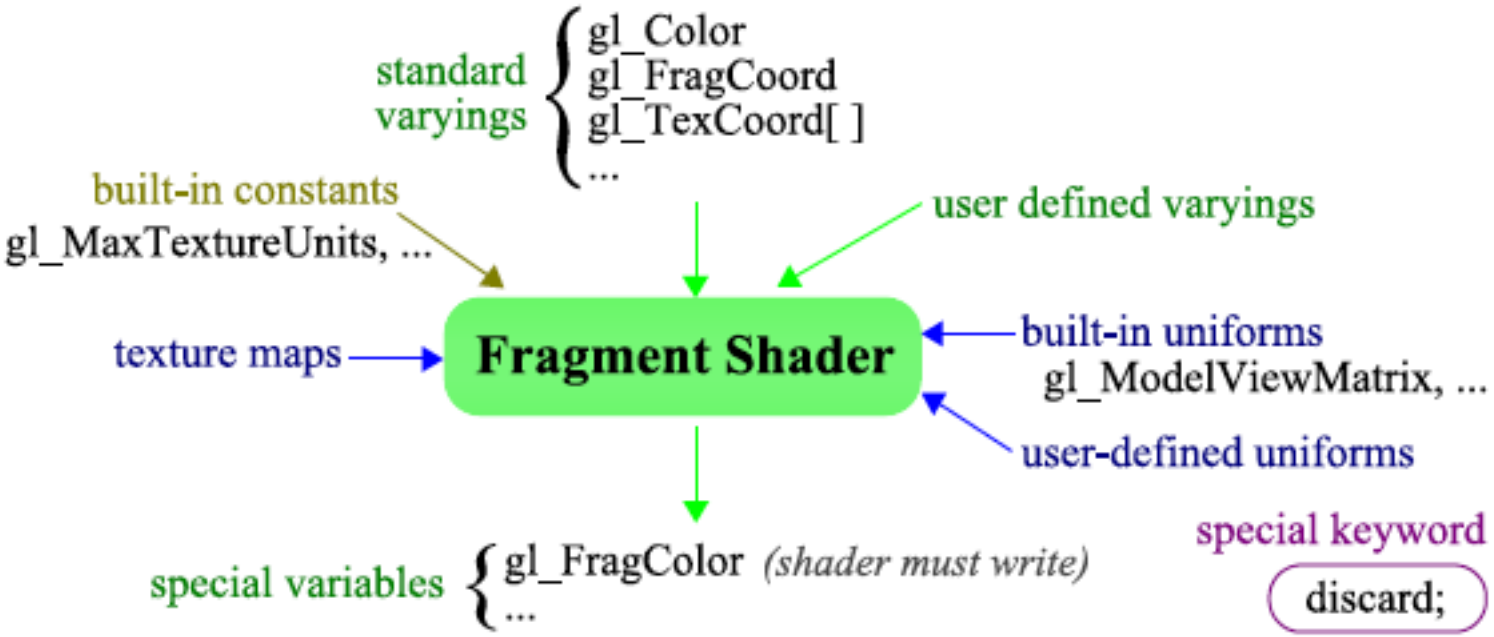


Figure 18. Input/Ouput summary of the fragment shader.

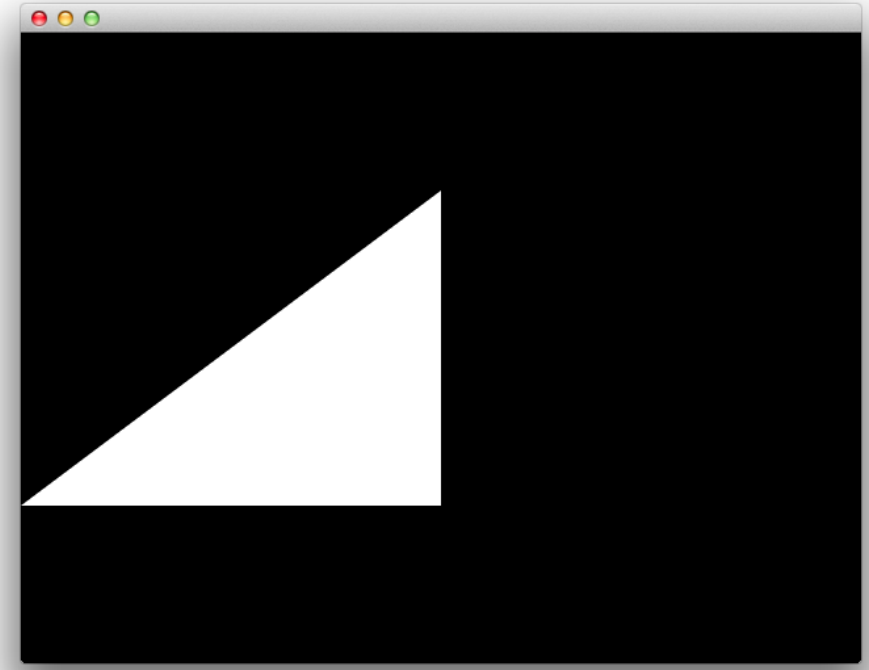
Transformations

- Une transformation (pour un modèle) est composée d'une position et une orientation.
  - Rotation, translation, mise à l'échelle et leurs combinaisons.

- En OpenGL, les modèles et la vue, sont combinés au travers d'un seul pipeline hardware appelé : *modelview matrix*
  - Attention, la première transformation (écrite dans le programme) sera appliquée en dernier.

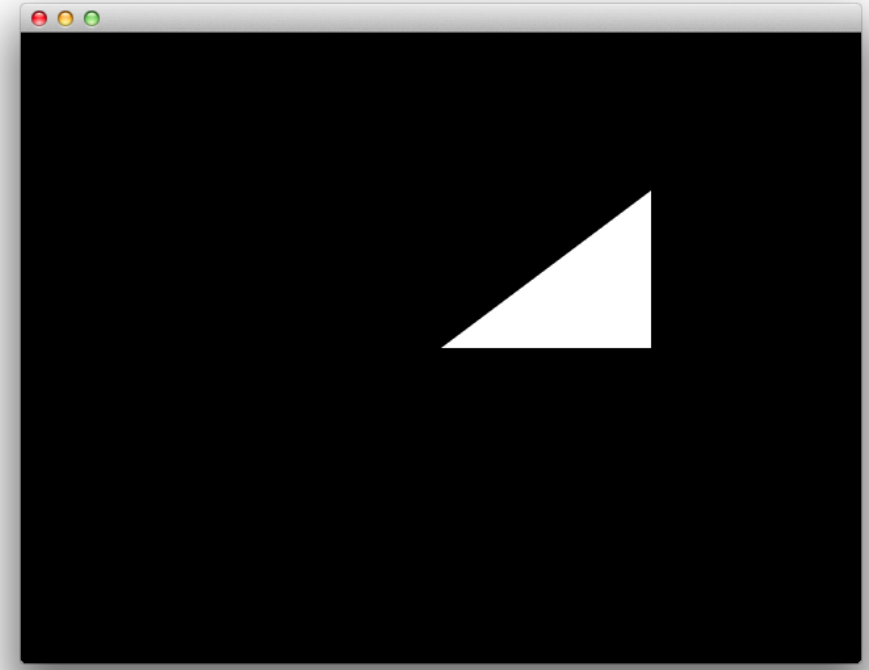
- glTranslate{fd}(Type x, Type y, Type z)
  - Multiplie la matrice courante par une matrice de *mouvement*
- glRotate{fd}(TypR angle, Type x, TypR x, TypR y, TypR z)
  - Multiplie la matrice courante par une matrice qui effectue une rotation du modèle courant
- glScale{fd}(Type x, Type y, Type z)
  - Multiplie la matrice courante avec une matrice de réduction

- Un exemple:
  - Dans la boucle de rendu :



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

- Un exemple:
  - Dans la boucle de rendu :



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

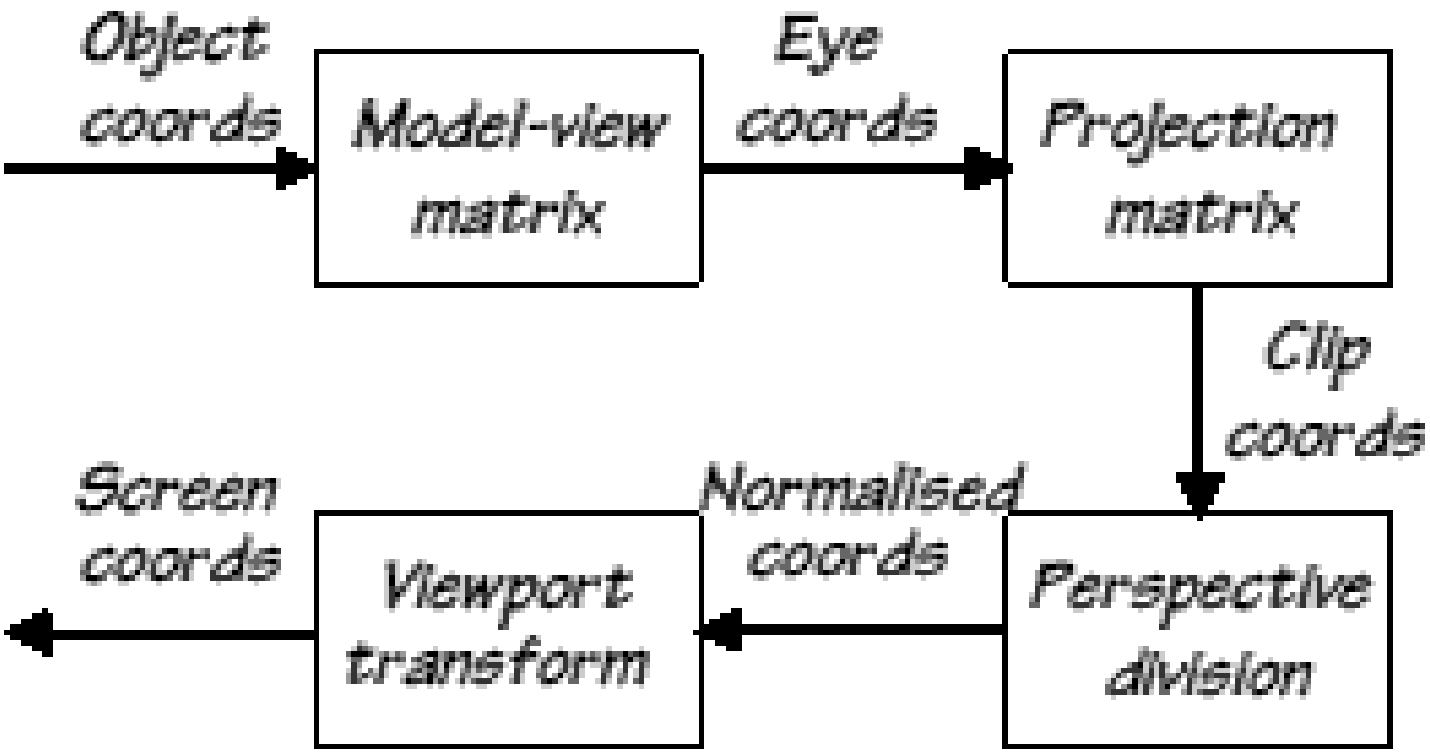
- Des opérations sur les matrices :
  - glLoadIdentidy()
    - Mettre la matrice courante en matrice identité 4x4

$$I_1 = (1),\ I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},\ \cdots,\ I_n = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

  - glLoadMatrix{fd}(const Type*m)
    - Remplace la matrice actuelle
  - glMultMatrix{fd}(const Type*m)
    - Multiplie la matrice actuelle
  - glPushMatrix()
    - Push la matrice actuelle dans la pile de matrice
  - glPopMatrix()
    - Pop la matrice actuelle dans la pile de matrice

Cameras

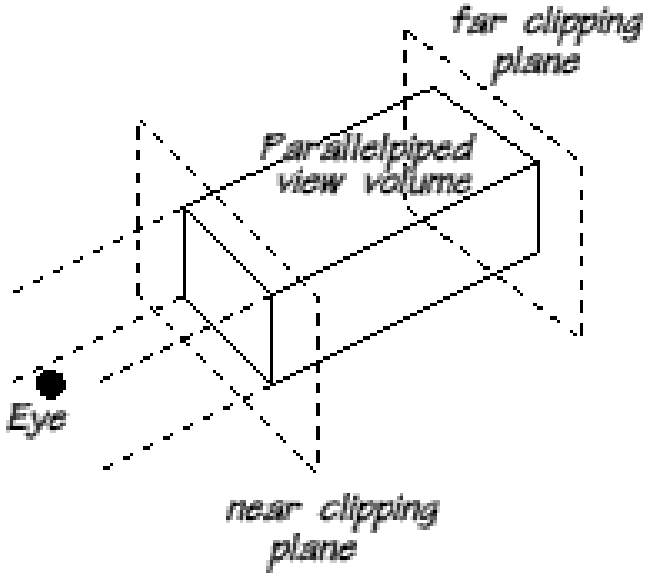
- La camera OpenGL peut être vue comme un appareil photo. Pour prendre une bonne photo il faut :
  - Placer la camera dans l'espace et la faire pointer vers la scène
  - Puis arranger la composition de la scène par rapport à la caméra
  - Définir l'objectif (lentille) de la camera et ajuster le zoom
  - Finaliser la photo en choisissant l'agrandissement désiré.


Object coords
Model-view matrix
Eye coords
Projection matrix
Clip coords
Perspective division
Normalised coords
Viewport transform
Screen coords

- La transformation de vue:
  - Est composée de translation et rotation
  - L'état initial de la caméra est composé de la position de ***l'œil,*** la direction dans laquelle regarde la camera.
  - Différentes caméra sont possible:
    - **Perspective**
    - ***Orthogonal***

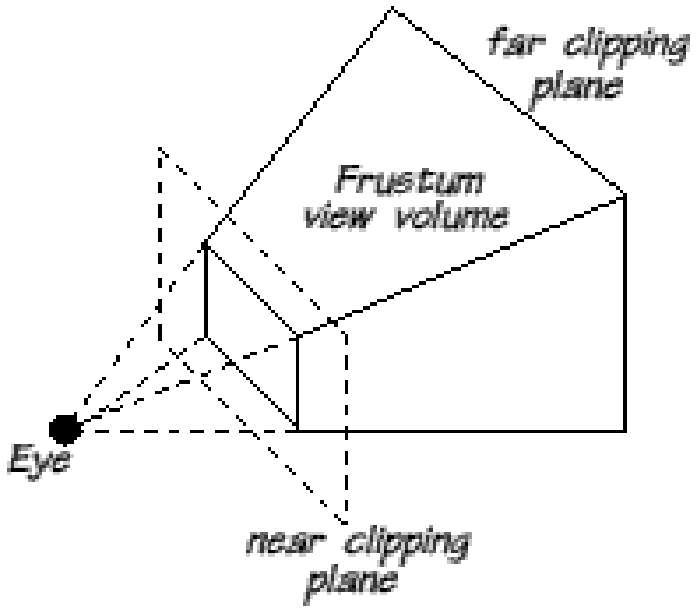
- glOrtho(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top, Gldouble near, Gldouble far)

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- glFrustrum(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top, Gldouble near, Gldouble far)

$$\begin{bmatrix} \frac{2 \times near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2 \times near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 \times far \times near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



- La transformation ***Viewport***
  - Transforme l'image finale dans une région de l'écran.
  - Le ***viewport*** est mesuré par le système de coordonné de la fenêtre.
  - glViewport(GLint x, GLint y, Glsizei width, Glsizei height)
- Il est possible de changer le mode d'une matrice:
  - glMatrixMode(Glenum mode)
    - Type de projection, texture, etc

- Les opérations de pile permettent la construction d'un système hiérarchique
  - Un exemple:

```
glPushMatrix();
    the car body transformation;
    Draw the car body;
    for(1 to 4)
        glPushMatrix();
        The wheel's local transformation;
        Draw the wheel;
    glPopMatrix();
glPopMatrix();
```

- Deux modes pour gérer les couleurs:
  - RGBA
    - 4 composantes
    - Normalisé entre 0.0 et 1.0
    - Ex:
      - glColor{34}[v] (type Colors)
  - Index de couleur

    utilise une carte de couleur (externe)

    Ex:

    -glIndex{sidf..}(type c)

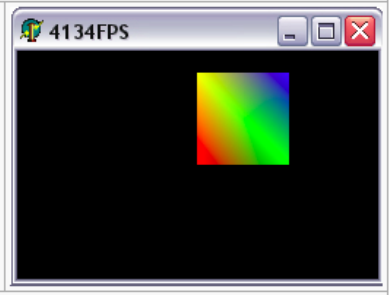
Lumières

- Clearing le tampon de couleur
  - glClearColor()
  - glClearIndex()

- Spécifié le model de shading
  - glShadeModel(Glenum mode)
    - GL_FLAT
    - GL_SMOOTH

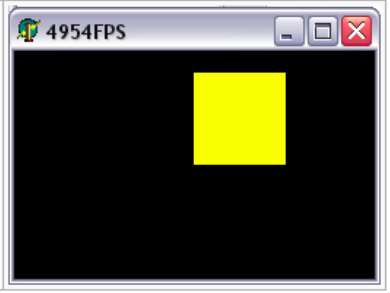
Smooth Shading (GL_SMOOTH)

```
glShadeModel(GL_SMOOTH);
glBegin(GL_QUADS);
  glColor3f(1, 0, 0); glVertex3f(0, 0, 0);
  glColor3f(0, 1, 0); glVertex3f(1, 0, 0);
  glColor3f(0, 0, 1); glVertex3f(1, 1, 0);
  glColor3f(1, 1, 0); glVertex3f(0, 1, 0);
glEnd;
```



Flat Shading (GL_FLAT)

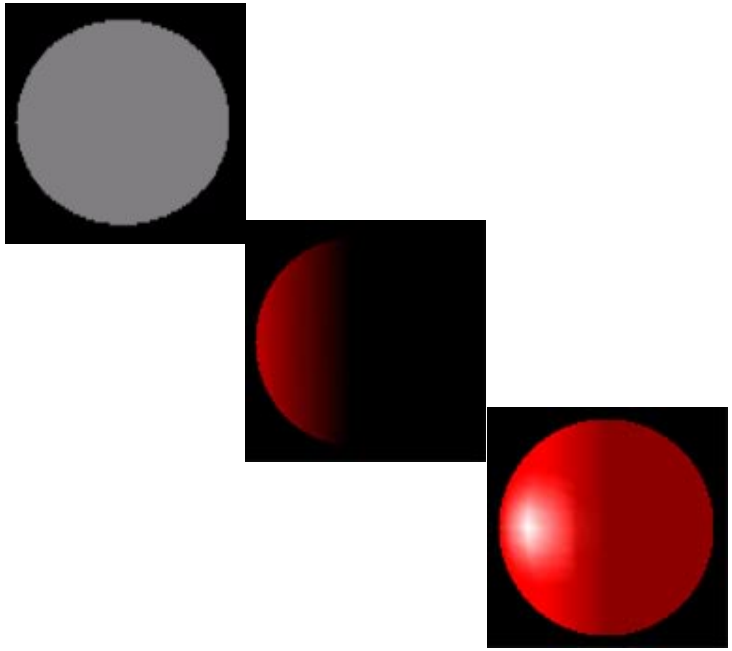
```
glShadeModel(GL_FLAT);
glBegin(GL_QUADS);
  glColor3f(1, 0, 0); glVertex3f(0, 0, 0);
  glColor3f(0, 1, 0); glVertex3f(1, 0, 0);
  glColor3f(0, 0, 1); glVertex3f(1, 1, 0);
  glColor3f(1, 1, 0); glVertex3f(0, 1, 0);
glEnd;
```



- Cacher les surfaces
  - glEnable(GL_DEPTH_TEST)

- Nettoyer le tampon de profondeur
  - glClear(GL_DEPTH_BUFFER_BIT)

- Différents type de lumières
  - Spéculaire
  - Diffuse
  - Ambiant



- Activer la lumière : glEnable(GL_LIGHTING)
- glLight{if}(Glenum light, Glenume pname, type param)

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(x, y, z) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

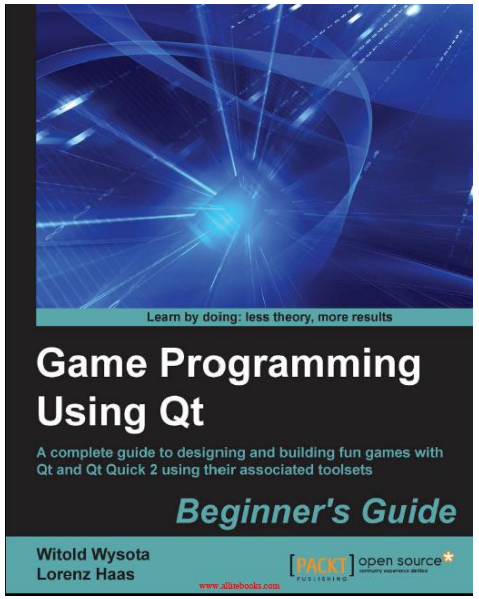
Effets spéciaux

- Plus d'informations sont disponible dans la documentation OpenGL.
  - Blending
    - glBlendFunc()
  - Brouillard
    - glFog()
  - AntiAliasing
    - glHint()

- Pour les effets spéciaux, nous utiliserons les shaders (cours 6)
  - Vertex Shaders
  - Fragment Shaders
- Exemples
  - Textures
  - Détails
  - Ombres portées
  - Profondeur de champ

- Merci de votre attention.

- Ressources:
  - OpenGL: www.opengl.org
  - Gamasutra: www.gamasutra.com
  - Rendu temps réel: www.realtimerendering.com
  - Game programming Using Qt



CONCLUSION

- Et maintenant … premier TP
- Créer un compte sur Github à votre nom
- Cloner le TP avec la commande « fork »

https://github.com/master-imagina/hmin317-tp1.git

- Lire le TP et répondre aux questions
- Faire des commits régulièrement
- Quand tout est fini, Insérer le compte-rendu
- Revenir dans Github et faire un « pull request »
- Envoyez moi aussi un message avec vos noms, prénom et identifiants GitHub