Une interface expérimentale pour lier OpenGL à Python

Fabrice Salvaire

PyConFr 2014

26 Octobre 2014

fabrice.salvaire@orange.fr
https://github.com/FabriceSalvaire/pyconfr-2014

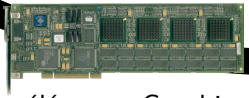
Évolution des Processeurs Graphiques

Evan & Sutherland
1970

Convergence

- du hardware professionnel et grand-publique
- des processeurs graphiques et de calculs
- des plateformes : de l'embarqué au super-calculateur

SGI Workstation
1990-2004



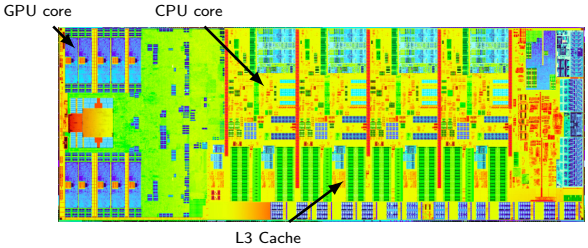
Accélérateur Graphique

Nvidia GeForce 256
1er GPU
2000

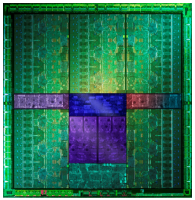
General Purpose GPU
2006

Qeulques mots sur les GPUs

Intel Haswell Die

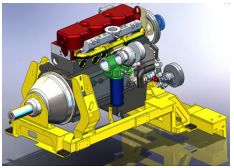


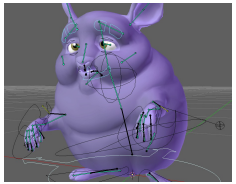
Nvidia Kepler Die



Architecture de calcul $(\times, +)$ hautement parallèle
des interpolateurs de lignes et de triangles
couplée à une mémoire rapide (GDDR5)

Usage des GPUs



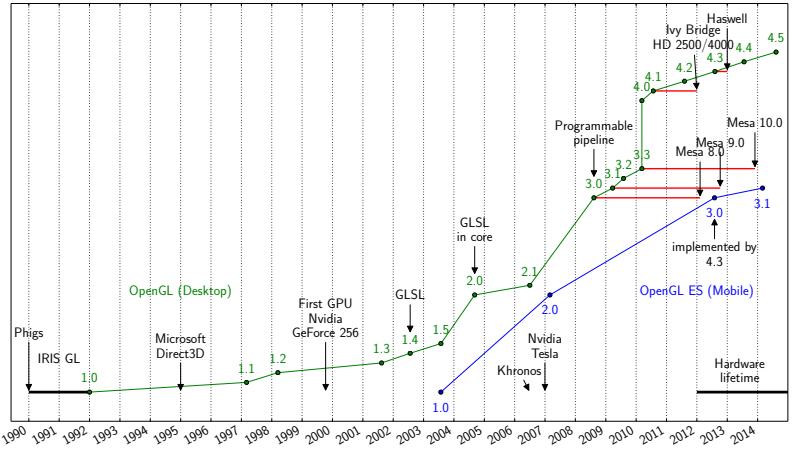


- Moteur de jeu
- Modeleur **3D**
- CAO
- **Visualisation** scientifique
- **Interface graphique**
  e.g. Qt QML, KDE Plasma
- **2D** avec accélération matériel
- **Calcul** : CUDA, OpenCL

Présentation de l'API OpenGL

- Un standard **ouvert** gouverné par le groupe Khronos
- La seule API **cross-platforme** et **cross-vendor**
- L'API de facto de GNU/Linux et d'Android
- Une API orientée desktop (OpenGL) et embarqué (OpenGL ES)
- et aussi orientée web (WebGL)
- Mais pas un idéale cf. Siggraph 2014 Khronos talk

Évolution de l'API OpenGL : 20 ans d'existence



Déphasage entre

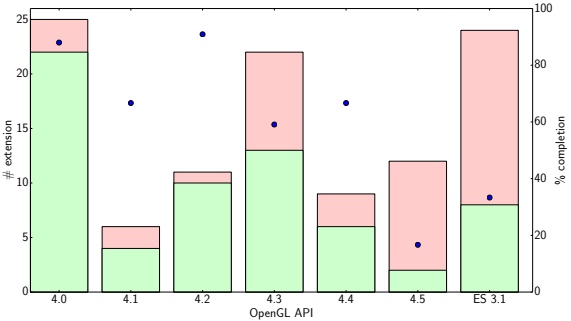
- OpenGL
- Hardware
- machine de l'utilisateur
- Mesa
- programmeur

Quelle API choisir ?

Toujours actif, mais concurrencé : DirectX 12, AMD Mantle, Apple Metal ...

Implémentations

Status de Mesa pour l'Intel HD Graphics (*)



- Extensions :
  ensemble de nouvelles fonctions
  avant leurs intégrations dans l'API core
- Mesa : OpenGL sous Linux
  à 2 doigts d'OpenGL V4.0
- Nvidia implémente 99,9 % de l'API

(*) core i3/5/7

Programmable Pipeline

OpenGL Moderne = Calcul Générique + Graphic Stuffs

flux de vertex $\longrightarrow$ ligne / triangle $\longrightarrow$ pixels

- programme : shader
- constantes : uniform
  e.g. float, matrice
  interpolateur d'images 1D/2D/3D
- données : in
  sous formes de flux (array)
  vecteur 1D/2D/3D
- résultat : out

Numpy Array

Shader GLSL

```
uniform mat4 model_view_matrix;
uniform mat3 normal_matrix;
uniform usampler2D texture0;

in vec3 position;
in vec3 normal;

out vec4 fragment_colour;

void main() {
  ...
  fragment_colour = ...;
}
```

Outils pour interfacer du code C avec Python

	Avantages	Inconvénients	Pypy
API C Python	• au coeur de CPython	• implémentation manuelle • requiert un compilateur	Non
SWIG = API C automatisé	• **header parser** • **supporte C++** • multi-langages	• le mécanisme de macros (typemap) engendre du code bloat • pas optimisé pour un langage	Non
Ctypes	• **inclus dans CPython**	• pas d'header parser • pas d'API level	Non
Cffi	• **header parser** • **ABI et API level**	• ???	Oui

Cffi est la solution idéale pour interfacer du C

SWIG est une solution pour interfacer du C++

mais l'idéale serait la réflexion

⋆ libffi

mais aussi
sip Pyrex

PyOpenGL : la réference actuelle

3 générations :

- V1.x C extension basé sur l'API C Python (David Ascher et. al)
- V2.x généré par SWIG (Tarn Weisner Burton)
- V3.x basé sur ctypes et un header parser (Mike C. Fletcher)
  compatibilité ascendante avec la V2.x

et des alternatives plus spécifique :

- Pyglet a cross-platform windowing and multimedia library
- Vispy a high-performance interactive 2D/3D data visualization library

PyOpenGL : ce qui pêche

Exemples de difficultés rencontrées au cours du temps :

- absence de constante :

```
GL.GL_RG_INTEGER = OpenGL.constant.Constant('GL_RG_INTEGER', 0x8228)}
```

- `glGetObjectParameteriv` est ici :

```
import OpenGL.GL.ARB.shader_objects as GL_SO
```

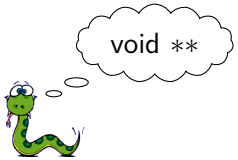
- `glGetActiveUniformBlockName` n'a pas de Pythonic wrapper :

```
name = ctypes.create_string_buffer(max_name_length)
name_length = OpenGL.arrays.GLsizeiArray.zeros((1,))
GL.glGetActiveUniformBlockName(program, index, max_name_length, name_length, name)
return name.value[:int(name_length[0])]
```

En résumé :

- **une longue histoire**
- un code source touffu : 2000 fichiers (*en comptant les extensions*)
- Python et OpenGL ont évolué
- pas d'API de haut niveau pour l'OpenGL moderne

L'API OpenGL du point vue du codeur C



API OpenGL V4.5 core :

- 1328 constantes i.e. 0x1234
- 653 fonctions

Paramètres :

- type de base via typedef : `(unsigned) char, short, int, float, double`
- pointeur : `void *(*), char *(*), int *, float *, ...`

Return :

- `unsigned char, (unsigned) int`
- `void *, const unsigned char *` (quelques cas)

Soit

- type de bases
- chaîne de caractères
- tableaux : Numpy array

XML API Registry : c'est cool !

Fichier XML définissant :

- les constantes
- les fonctions et leurs prototypes

```
void glClearBufferData (GLenum target, GLenum internalformat, GLenum format, GLenum type, const void * data)
```

```
<command>
    <proto>void <name>glClearBufferData</name></proto>
    ...
    <param><ptype>GLenum</ptype> <name>format</name></param>
    <param><ptype>GLenum</ptype> <name>type</name></param>
    <param len="COMPSIZE(format,type)">const void *<name>data</name></param>
</command>
```

la taille est indiquée

- les extensions
- les **versions** et leurs **profiles**

$\longrightarrow$ Apporte des informations essentielles par rapport au fichier d'en-tête

Tous les paramètres ne ressemblent pas . . .

simple
`GLenum target`
passé par copie

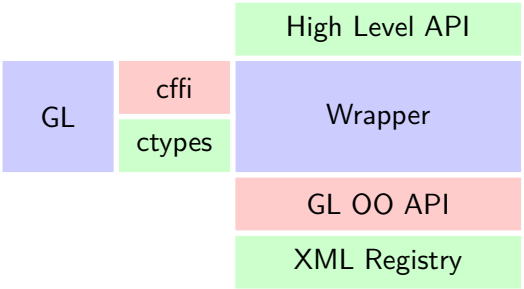
output par référence
`int *[1] length`
plus d'un paramètre retourné (on suppose)

input/output via pointeur
`GLsizeiptr size, void *[size] data`
la taille d'un tableau doit être indiqué en C

input via pointeur
`GLsizeiptr size, const void *[size] data`
`const` indique que le paramètre est en lecture seul

pointeur complexe
```
GLenum pname, GLint *[COMPSIZE(pname)] data
const void *[COMPSIZE(format,type,width)] pixels
```
la taille du tableau est calculé à partir des paramètres

Architecture



1. l'utilisateur requiert sont API

```
enums, commands = generate_api(api, api_number, profile)
```

```
enums, commands = generate_api('gl', '4.0', 'core')
```

2. on construit le wrapper à la volé

```
setattr(GL, enum, value)
```

```
setattr(GL, command, CommandWrapper())
```

La classe CommandWrapper (simplifié)

❶ **Python ⟶ C**

```
class ParameterWrapper():

    def __init__(self, parameter):
        ...

    def from_python(self, parameter, c_parameters):
        ...
        return to_python_converter()
```

❸ **C ⟶ Python**

```
class ToPythonConverter():

    def __init__(self, data):
        ...

    def __call__(self):
        ...
        return converted value
```

```
class CommandWrapper():

    def __init__(self, command):
        self._parameter_wrappers = [parameter_wrapper(parameter) for parameter in command.parameters
                                    if ...]
❷   def __call__(self, *args):
        c_parameters = [None, ...]

        for parameter_wrapper, parameter in zip(self._parameter_wrappers, args):
            to_python_converter = parameter_wrapper.from_python(parameter, c_parameters)

        output = self._function(c_parameters)

        return [output] + [to_python_converter() for to_python_converter in to_python_converters]
```

Traduction automatique des prototypes

simple `GLenum target`

output par référence `int *[1] length` / new type[1]

int foo(int p1 , int *r1, int *r2)

o0 , . . . , r1, r2 = foo(p1)

Pointeur Complexe ⚠ XML est incomplet : la formule n'est pas indiqué

If

1. type = `const char *` (e.g. shader source)
2. p = Numpy array :
   check type if type not void

Solution :

- Pythonic Wrapper qui se substitue à la fonction
- compléter le fichier XML

Traduction automatique des prototypes : input via pointeur

`GLsizeiptr size, const void *[size] data`

⚠ XML est incomplet : `size = len(p)` **ou** `size_of(p)`
logiquement `void` $\Longrightarrow$ `size_of` et `float` $\Longrightarrow$ `len`

If

1. type = `const char **` :
   `str(p)`
   ou `[str(x) for x in p]`
2. p = Numpy array :
   check type if type not void
   `size = p.size` ou `p.nbytes`
3. p = iterable
   `size = len(p)`

```
int foo( int p1 , int *s2, const float *p2 )
o0 = foo( p1 , p2 )
```

Traduction automatique des prototypes : Output/Input via Pointeur

If

1. type = `void *` :
   p = Numpy array
   `size = p.nbytes`

   `int foo( int p1 , int s2, float *p2 )`
   `o0 = foo( p1 , p2 )`

2. type = `char *` :
   p = size
   new string

   `int foo( int p1 , int s2, char *p2 )`
   `o0 , p2 = foo( p1 , s2 )`

3. p = Numpy array :
   check type
   `size = p.size`

   idem ❶

4. else :
   p = size
   new Numpy array or list

   idem ❷

Pour se simplifier la vie . . .

- query-opengl-api : utilitaire pour interroger l'API

- `command.manual()` dans une session IPython
- `command.call_counter` pour tracer son application

- Gestion des erreurs

```
with GL.error_checker():
    ...
    # at exit:
    #     call glGetError
    #     raise exception if GL error
```

High Level API

1. OpenGL moderne est une API bas niveau

Surcouche offrant des services de bases :

- viewport, projection, caméra
- interface Orienté objet : texture, VAO, VBO
- font

Shader

```
uniform mat4 model_view_matrix;

in vec3 position;

void main() {
  ...
}
```

2. Interface Orientée Objet :
   - Get/Set uniform
     `shader.model_view_matrix = array`
   - Accéder au slot d'un varying
     `shader.position`

Pour aller plus loin

Code source : https://github.com/FabriceSalvaire/PyOpenGLng

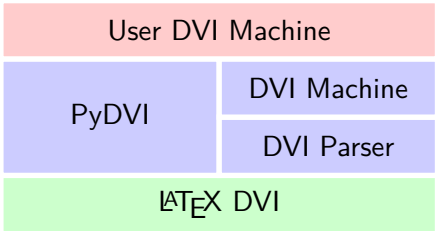
- Peaufiner la GlApi
- Conserver uniquement cffi ?
- Tester plus largement le wrapper
- Améliorer l'interface de haut niveau
- Benchmark
- Compléter le fichier XML

Un visualisateur de fichier DVI accéléré par GPU

PyDVI a Python library to read and process DVI files

- packed font, Type 1, virtual font
- TeX font metric, Adobe Font Metrics
- font map
- font encoding

- DVI $\longrightarrow$ PNG tool
- OpenGL Viewer



https://github.com/FabriceSalvaire/PyDVI