3D graphics with OpenGL using Python

Lorenzo Mancini <lmancini@develer.com>

9th May 2009
PyCon Italia 3 – Scoprire Python

develer

. software . hardware . innovation

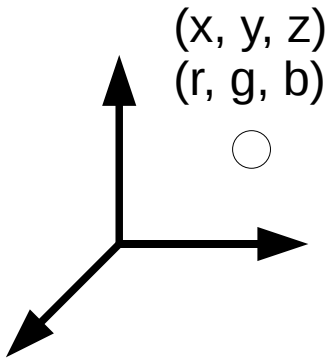
Target

- Python devs looking into OpenGL
- OpenGL C seasoned devs

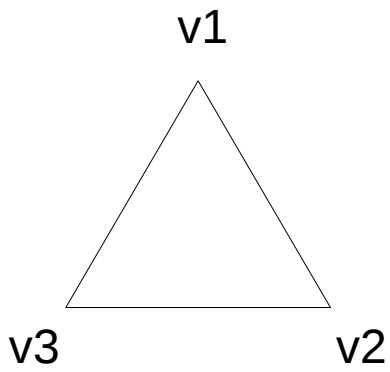
Talk overview

- 3D graphics crash course
- Introduction to OpenGL
- Choosing a Python OpenGL wrapper
- Drawing with PyOpenGL
- Extensions and how to use them
- Rewrite bottlenecks in C
- Measuring wrapper overhead
- A glimpse of the future

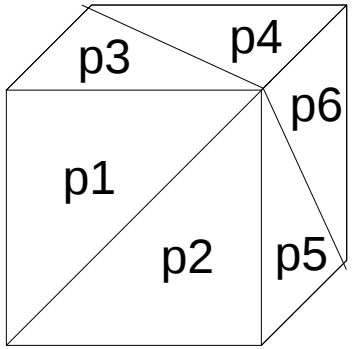
The building blocks



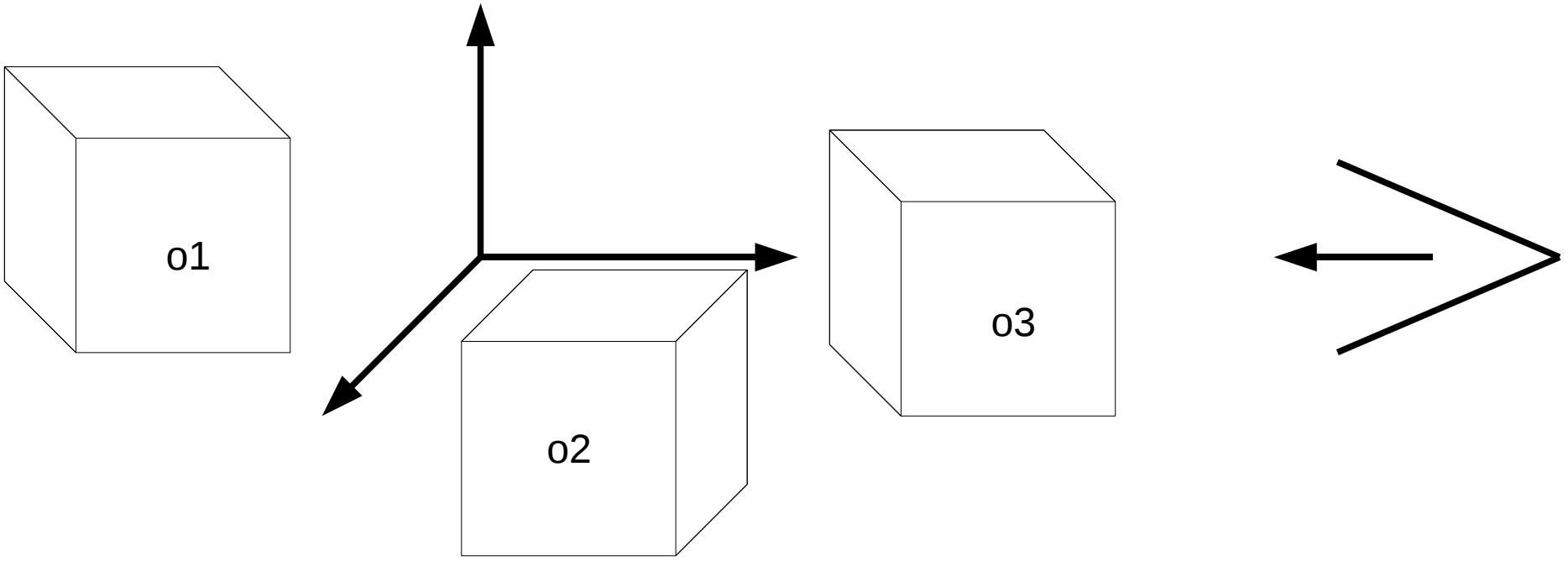
Vertex



Polygon



Object

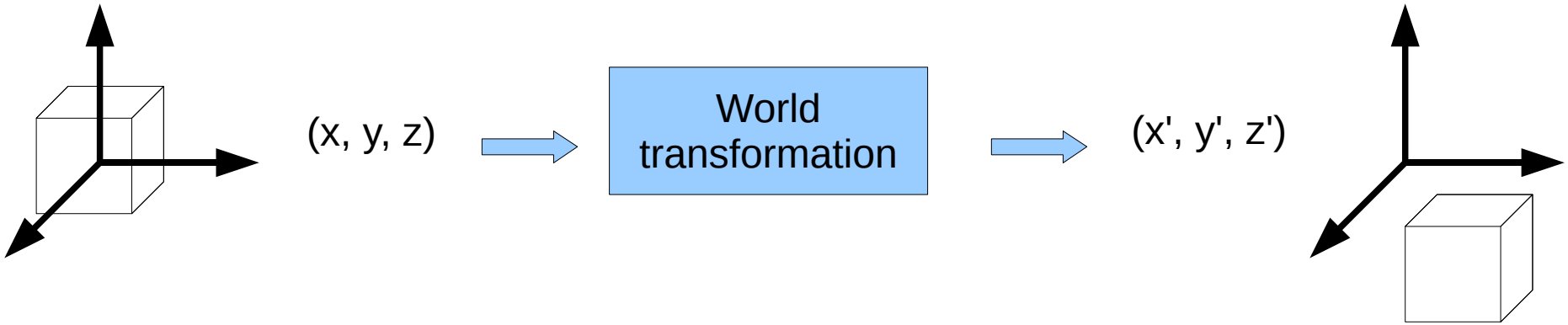


World

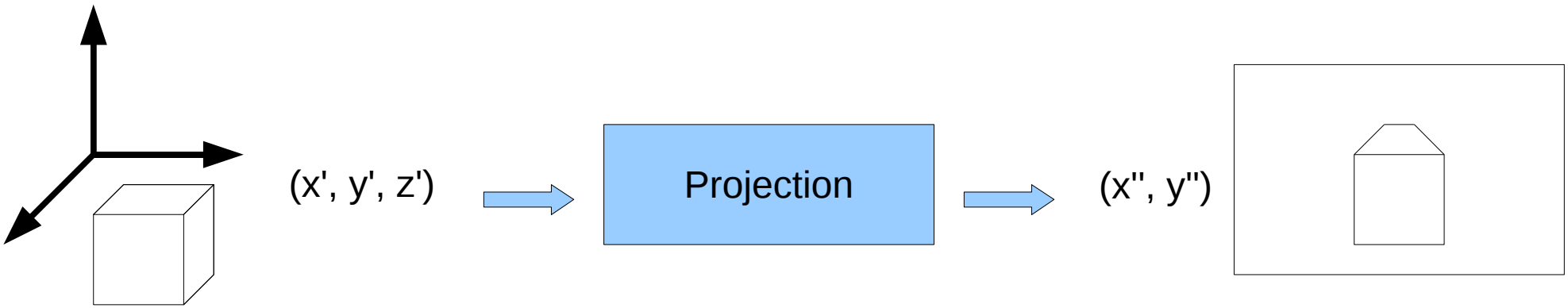
Camera

3D graphics crash course

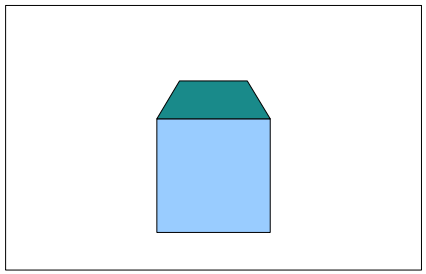
- Place objects in the world



- Project on the screen



- Fill polygons



Sharing the work

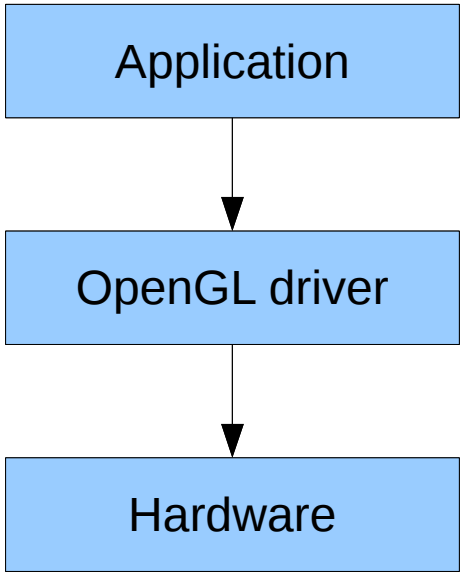
- Early 3D days: 100% CPU
- GPUs began evolving
  - Polygon fill (since 1996)
  - Transformation + lighting (since 2000)

Talk overview

- 3D graphics crash course
- **Introduction to OpenGL**
- Choosing a Python OpenGL wrapper
- Drawing with PyOpenGL
- Extensions and how to use them
- Rewrite bottlenecks in C
- Measuring wrapper overhead
- A glimpse of the future

OpenGL

- Standard API to describe 3D scenes
- Specs approved by ARB
- Hardware acceleration (nV, ATI, Intel)
- Software implementations (Mesa, TinyGL)
- ICD (Installable Client Driver)



First impressions

```
glClear(GL_COLOR_BUFFER_BIT)
glLoadIdentity()
glTranslatef(-1.5, 0.0, -6.0)

glBegin(GL_TRIANGLES)
    glColor3f(1.0, 0.0, 0.0)
    glVertex3f(0.0, 1.0, 0.0)
    glColor3f(0.0, 1.0, 0.0)
    glVertex3f(1.0, 1.0, 1.0)
    # ...
glEnd()
```

- Simple API
- Struct-less
- Global state (context)

GL context

- Creating context: graphic system dependent issue
- No specific OpenGL API:
  - wglCreateContext (Windows)
  - glXCreateContext (Xorg)
  - ...or use pygame/PyQt:

Using pygame with OpenGL

```
import pygame
from OpenGL.GL import *

def initGL():
    glClearColor(0.0, 0.0, 0.0, 0.0)

def resizeGL((w, h)):
    glViewport(0, 0, w, h)

def paintGL():
    glClear(GL_COLOR_BUFFER_BIT)
    glBegin(GL_TRIANGLES)
    # vertices here

pygame.init()
pygame.display.set_mode((640,480), pygame.locals.OPENGL)
initGL()
resizeGL((640,480))
paintGL()
```

Using PyQt with OpenGL

```
from PyQt4.Qt import *
from OpenGL.GL import *

class GLWidget(QGLWidget):
    def initializeGL(self):
        glClearColor(0.0, 0.0, 0.0, 0.0)

    def resizeGL(self, w, h):
        glViewport(0, 0, w, h)

    def paintGL(self):
        glClear(GL_COLOR_BUFFER_BIT)
        glBegin(GL_TRIANGLES)
        # vertices here

app = QApplication([])
glw = GLWidget()
glw.show()
app.exec_()
```

Talk overview

- 3D graphics crash course
- Introduction to OpenGL
- **Choosing a Python OpenGL wrapper**
- Drawing with PyOpenGL
- Extensions and how to use them
- Rewrite bottlenecks in C
- Measuring wrapper overhead
- A glimpse of the future

OpenGL wrappers

- **PyOpenGL – 3.0.0 released Apr 2009**
  - Auto-generated
  - array support (numpy, ctypes...)
  - Convenience classes
- pyglet
  - less friendly interface
  - multimedia functions

OpenGL and PyOpenGL

- Calling functions:
  - `glGetFloatv(GL_CURRENT_COLOR, &color);`
  - `color = glGetFloatv(GL_CURRENT_COLOR)`
- Error handling:
  - `glGetError() /* after GL calls */`
  - `# glGetError implicitly called, raise exc`
- Array handling
  - `glLoadMatrixf(const GLfloat *matrix);`
  - `glLoadMatrixf([1.0, ...]) # or numpy/ctypes`
- OpenGL tracing:
  - `/* external tool */`
  - `OpenGL.FULL_LOGGING = True`

Price tag

- Function call performance penalty
- Tune PyOpenGL 3.x performance:
  - `OpenGL.ERROR_CHECKING = False`
  - `OpenGL.ERROR_ON_COPY = True`
- Still want more? Raw ctypes wrapper:
  - `from OpenGL.raw.GL import *`
  - `v = (GLfloat * len(matrix))(*matrix)`
  - `glLoadMatrixf(v)`
- You **might** benefit from these

Talk overview

- 3D graphics crash course
- Introduction to OpenGL
- Choosing a Python OpenGL wrapper
- **Drawing with PyOpenGL**
- Extensions and how to use them
- Rewrite bottlenecks in C
- Measuring wrapper overhead
- A glimpse of the future

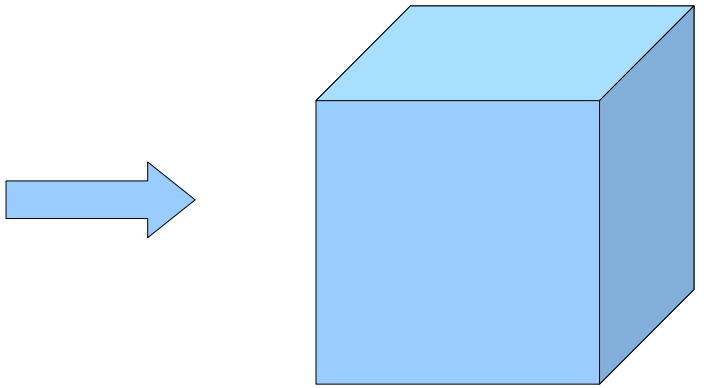
Drawing modes

- Command-based
  - Immediate mode
  - Display lists
- Array-based
  - Vertex array
  - Vertex buffer objects

Immediate mode

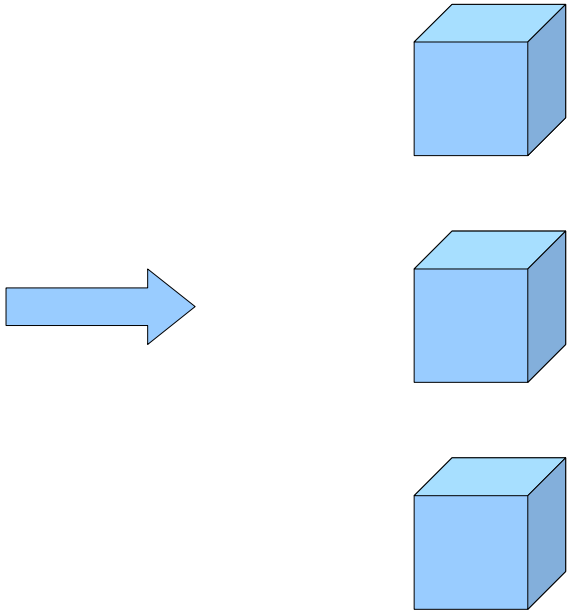
- Explicit GL commands:

```
glBegin(GL_TRIANGLES)
glColor3f(0, 1, 1)
glVertex3f(0, 0, 0)
#...
glEnd()
```



- Draw multiple istances of object:

```
for y in (10, 20, 30):
    glLoadIdentity()
    glTranslate3f(0, y, 0)
    glBegin(GL_TRIANGLES)
    glColor3f(0, 1, 1)
    glVertex3f(0, 0, 0)
    #...
    glEnd()
```



Immediate mode (2)

- Very slow; for **each** function call:
  - argument marshalling
  - C function call
  - glGetError handling
- ...millions of vertices in a scene...
- Resend even if unchanged

Display lists

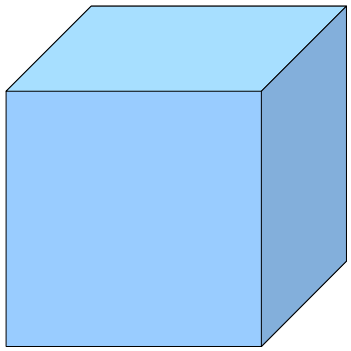
- Record GL commands off-line:

```
list = glGenLists(1)
glNewList(list, GL_COMPILE)
glBegin(GL_TRIANGLES)
#...
glEnd()
glEndList()
```

(...nothing...)

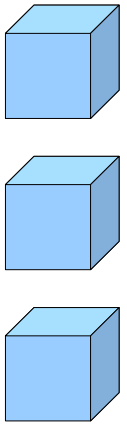
- Playback when needed:

```
glCallList(list)
```



- Multiple instances:

```
for y in (10, 20, 30):
    glLoadIdentity()
    glTranslate3f(0, y, 0)
    glCallList(list)
```

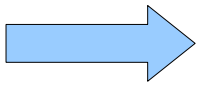


Display lists (2)

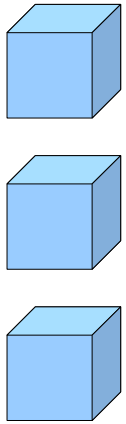
- …or use a nested display list

```
nested_list = glGenLists(1)
glNewList(nested_list, GL_COMPILE)
for y in (10, 20, 30):
    glLoadIdentity()
    glTranslate3f(0, y, 0)
    glCallList(list)
glEndList()

glCallList(nested_list)
```



(...nothing...)



- Works on current context
- No Python-side vertices iteration
- HW impls can store data on GPU
- Not inspectable nor mutable

Vertex array

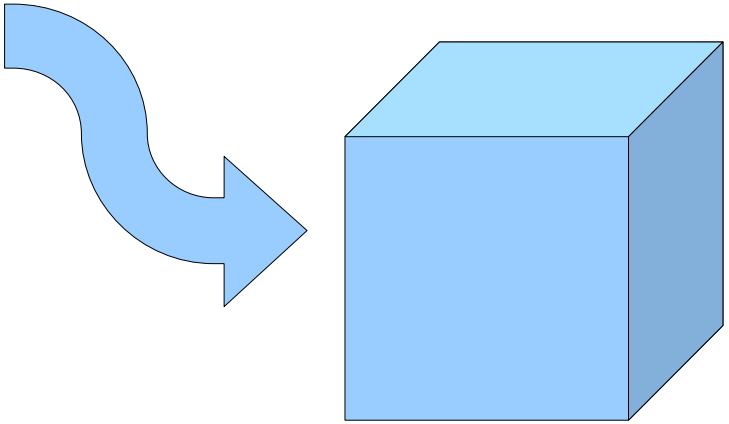
- Array of vertices

```
vertices = [(1.0, 1.0, 0.0), (0.0, 0.0, 1.0), ...]
varray = numpy.array(vertices, 'f'))
```

- Need for semantics

```
glVertexPointer(3, GL_FLOAT, 12, varray)
glDrawArrays(GL_TRIANGLES, 0, len(varray))
```

- Multiple objects:
  - multiple glDrawArrays
  - bigger array



- Vertices stored application side

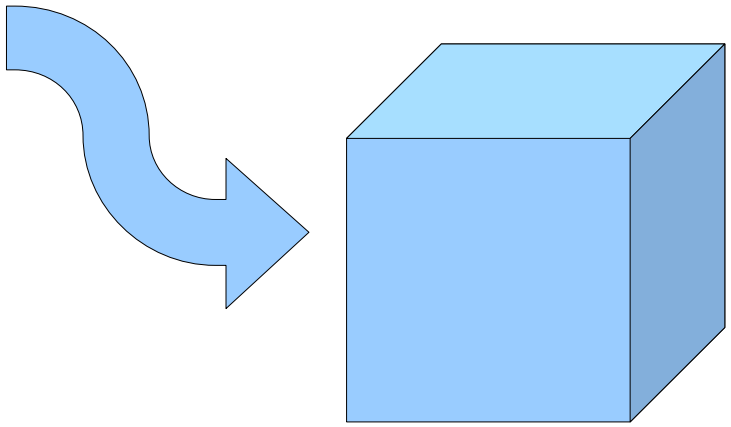
Vertex buffer objects

- Closer to the metal
- HW impls store vertex array on GPU

```
vertices = [(1.0, 1.0, 0.0), (0.0, 0.0, 1.0), ...]
vb = vbo.VBO(numpy.array(vertices, 'f'))
```

- Draw on command (bind-to-use)

```
vb.bind()
try:
    glVertexPointer(3, GL_FLOAT, 12, 0)
    glDrawArrays(GL_TRIANGLES, 0, len(vb))
finally:
    vb.unbind()
```

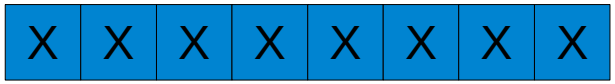


- Same API as VA

Vertex buffer objects (2)

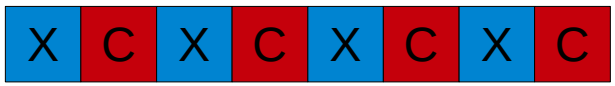
- Homogeneous VBO:

```
glVertexPointer(3, GL_FLOAT, 12, 0)
glDrawArrays(GL_TRIANGLES, 0, len(vb))
```



- Heterogeneous VBO:

```
glVertexPointer(3, GL_FLOAT, 24, 0)
glColorPointer(3, GL_FLOAT, 24, 12)
glDrawArrays(GL_TRIANGLES, 0, len(vb))
```



- Mutable (glBufferSubData / glMapBuffer)

A bigger picture

- IM : **DL** = VA : **VBO** (...stretched, but you get the idea)

- DL
  - vertices
  - transforms
  - nested lists
  - GL commands
  - ...
- VBO
  - array of vertices

- DLs mirror ancient hardware (emulated through VBO)
- DL deprecated since OpenGL 3.0
- VBO way to go with OpenGL 3.x
  - not available on older hardware (<2005?)

My 2c

- New application
  - aim for VBO
  - VA-based fallback codepath
- Existing application
  - DL emulation support in drivers
  - another option: layer DL on VA yourself
    - then plan a VBO transition (see above)

Talk overview

- 3D graphics crash course
- Introduction to OpenGL
- Choosing a Python OpenGL wrapper
- Drawing with PyOpenGL
- **Extensions and how to use them**
- Rewrite bottlenecks in C
- Measuring wrapper overhead
- A glimpse of the future

The whole story about extensions

- OpenGL implementations declare:
  - “Core” version supported (e.g. 2.1)
  - Supported extensions

- Extension: set of additional functions

```
glLoadIdentity()
glTranslate3f(10, 0, 0)
glDrawArraysInstancedARB(GL_TRIANGLES, 0,\
                         len(array), 10)
```

- Throughly documented on OpenGL website

Why should I care?

- 1) Newer GPUs - Bleeding edge factor:
  - ARB approval slow
  - Shiny new GPU features as extensions
- 2) Older GPUs – Lazy driver vendors:
  - Some drivers still at OpenGL 1.4
  - No whole 1.5, just relevant features as exts
  - notable example: VBO

Extensions in PyOpenGL

- Core functions
  - `from OpenGL.GL import *`
- Extension functions
  - `from OpenGL.GL.ARB.draw_instanced import *`
- Check for extension support
  - `bool(glDrawArraysInstancedARB)`

Talk overview

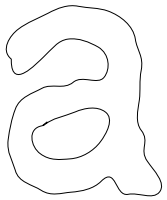
- 3D graphics crash course
- Introduction to OpenGL
- Choosing a Python OpenGL wrapper
- Drawing with PyOpenGL
- Extensions and how to use them
- **Rewrite bottlenecks in C**
- Measuring wrapper overhead
- A glimpse of the future

The need for speed

- **Long** GL command list
  - maybe iterating on a bunch of vertices
- What about DLs?
  - Setup time alone can be too long for realtime
- Rewrite just that part in C
- Expose the entry-point to Python

Real-life experience

- External library (Qt) producing font outlines
- How to draw outlines?
  - Calculate a polygon set that fills the outline
  - Remember there are holes
  - → Lots of Python function calls
- ...the “a” outline above has 44 vertices



Outline

Final appearance

Write, Wrap, W00t

- Compile in a shared library

```
void tesselate(const void *data, int n_points)
{
    /* GL/GLU commands */
}
```

- `gcc -shared -fPIC tessie.c -o tessie.so -lGL -lGLU`

- Wrap on-the-fly with ctypes:

```
tessie_lib = cdll.LoadLibrary("./tessie.so")
```

- tesselate draws **on current context**

```
tessie_lib.tesselate(poly.data(), poly.size())
```

Talk overview

- 3D graphics crash course
- Introduction to OpenGL
- Choosing a Python OpenGL wrapper
- Drawing with PyOpenGL
- Extensions and how to use them
- Rewrite bottlenecks in C
- **Measuring wrapper overhead**
- A glimpse of the future

Wrapper overhead

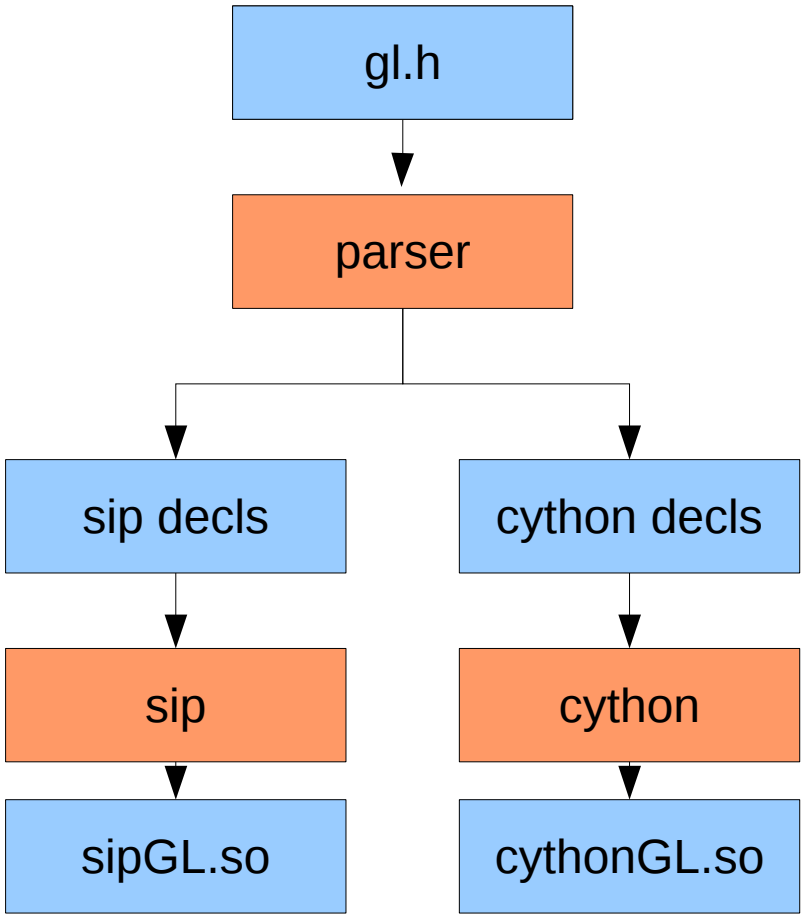
- ctypes in PyOpenGL 3
  - ease of maintenance and developing aids
  - considerable overhead **on function call**
- Out of my curiosity (and personal itch):
  - How much “considerable” is it?
  - Less overhead possible?

The experiment

- A wrapper with the least possible overhead
- Realistic prototype goals:
  - no error checking, logging, etc...
  - 100% C or C++
  - let's implement IM only
    - it's the best way to compare overhead
- I produced prototypes with sip and cython

Generating the wrapper

- C parser stol^Wborrowed from pyglet
- two prototypes, sip and cython based
- generate a sip/cython description file



Performance results

- Again: just measuring function call overhead
  - normally you never use IM

- PyOpenGL 3.x: ~12 fps
- PyOpenGL 3.x raw tuned: ~ 21 fps
- sipGL: ~50 fps
- cythonGL: ~50 fps

- Rough measure of overhead in PyOpenGL 3
- Needs a lot more work, will release when completed
- http://www.develer.com/~lmancini/pyopengl

Talk overview

- 3D graphics crash course
- Introduction to OpenGL
- Choosing a Python OpenGL wrapper
- Drawing with PyOpenGL
- Extensions and how to use them
- Rewrite bottlenecks in C
- Measuring wrapper overhead
- **A glimpse of the future**

What's coming?

- 1992 (1.0)
- 1997 (1.1)
- 1998 (1.2)
- 2001 (1.3)
- 2002 (1.4)
- 2003 (1.5)
- 2004 (2.0)
- 2006 (2.1)
- 2008 (3.0)
- 2009 (3.1)

- 16 years (1992-2008) of backward source compatibility
- OpenGL 3.0:
  - old API deprecated
- OpenGL 3.1:
  - old API moved in extension
- Why an API cleanup?
  - mirror recent hardware
  - lighter drivers