# Python para Computación Gráfica

Eduardo Graells-Garrido

11 de marzo de 2026

# 1 ¿Por qué Python y no C++ o C?

Tradicionalmente, las aplicaciones gráficas se implementan en lenguajes de bajo nivel como C++ o C, incluyendo motores de videojuegos, las bibliotecas de *rendering* y la visualización médica, entre otras. Los estándares de la industria como OpenGL, Vulkan y DirectX también están especificados en esos lenguajes. Esto se debe, entre otras cosas, a que permiten acceso directo al *hardware* y así se puede obtener el máximo rendimiento (¡si es que eres *hacker* en tu implementación!). Sin embargo, dada la complejidad de nuestras aplicaciones, Python no es ineficiente y su flexibilidad de ejecución nos permite desarrollar y aprender de manera profunda y didáctica: vamos a interactuar principalmente con los conceptos, no con los detalles de implementación y compilación de nuestros programas.

## Lenguajes compilados e interpretados

C++ es un lenguaje **compilado**: el código fuente se traduce a instrucciones de máquina (*machine code*) antes de ejecutarse mediante un compilador como `g++`. El resultado es un ejecutable binario que corre directamente sobre el procesador. Una desventaja de esto es que el ciclo de desarrollo (editar, compilar, enlazar, ejecutar) puede volverse tedioso, y los errores de memoria (punteros colgantes, desbordamientos de *buffer*) son como ninjas: silenciosos, escurridizos y mortales (como *segmentation fault* o `NullPointerException`).

Python es un lenguaje **interpretado**: un programa intérprete como `python` lee el código fuente y lo ejecuta línea a línea en tiempo de ejecución. No hay compilación explícita, lo que hace más ágil el desarrollo. Una desventaja es la velocidad: Python puede ser decenas de veces más lento que el equivalente en C++. Sin embargo, las partes que requieren *performance* (álgebra lineal, operaciones sobre arreglos, comunicación con la GPU) suelen estar implementadas en C o C++ y expuestas a Python mediante una interfaz (*binding*). Cuando se llama a `numpy` para multiplicar matrices o a `pyglet` para dibujar en pantalla, el trabajo pesado lo hace una biblioteca implementada en un lenguaje eficiente; Python solo orquesta la ejecución. Para las aplicaciones de este curso eso es suficiente.

Para hacer tangible la diferencia, aquí está el mismo cálculo en ambos lenguajes: sumar los primeros 500.000 números de Fibonacci en un ciclo `for`.

| C++ | Python |
|---|---|
| `#include <iostream>`<br><br>`int main() {`<br>`  long long a = 0, b = 1, suma = 0;`<br>`  for (int i = 0; i < 500000; i++) {`<br>`  long long c = a + b;`<br>`  a = b; b = c;`<br>`  suma += a;`<br>`  }`<br>`  std::cout << suma << std::endl;`<br>`}` | `a, b = 0, 1`<br>`suma = 0`<br>`for _ in range(500_000):`<br>`  a, b = b, a + b`<br>`  suma += a`<br>`print(suma)` |
| `g++ -O2 fib.cpp -o fib`<br>`./fib` | `python fib.py` |

Ambos programas imprimen el mismo resultado final. Sin embargo, el de C++ termina en pocos milisegundos; el de Python tarda varios cientos. La razón es directa: el `for` de C++ compila a un puñado de instrucciones que la CPU ejecuta directamente. El `for` de Python pasa por el intérprete en cada iteración: busca el nombre de la variable, verifica su tipo, ejecuta la operación y repite. Con 500.000 pasos, esa diferencia se acumula.

El código también ilustra la diferencia de **tipado**. En C++, cada variable se declara con su tipo explícito (`long long`[1], `int`): el compilador sabe de antemano cuánta memoria reservar y qué instrucciones usar. En Python no hay declaraciones de tipo: la instrucción `a, b = 0, 1` crea dos variables y el intérprete deduce en tiempo de ejecución que son enteros. Esa flexibilidad tiene un costo, porque antes de cada operación Python debe verificar qué tipo tiene el objeto, algo que C++ resuelve en compilación y nunca vuelve a preguntar.

Los conceptos que veremos en el curso (transformaciones, *shaders*, grafos de escena, iluminación, simulación física) son independientes del lenguaje. Una vez que los domines en Python, trasladarlos a C++ es una cuestión de sintaxis y herramientas, no de comprensión.

[1] Los números de Fibonacci crecen rápido: el valor en el paso 50 ya supera los 12.000 millones, más de lo que cabe en un entero de 32 bits. `long long` garantiza 64 bits en cualquier plataforma; `long` a secas tiene tamaño variable (32 bits en Windows, 64 en Linux y macOS), así que no es confiable para números grandes.

## 2 CLI

La CLI (*Command Line Interface*, también llamada terminal, consola o *shell*) es una interfaz textual para interactuar con el sistema operativo. En este curso la usaremos constantemente: para clonar el repositorio, instalar dependencias y ejecutar los ejemplos y las tareas. Ofrece control preciso sobre qué se ejecuta, con qué argumentos, en qué directorio y con qué entorno,

todo lo cual importa cuando el resultado es una ventana gráfica o una imagen renderizada.

### Comandos esenciales

Los comandos que usarás con más frecuencia son:

```
cd nombre-de-carpeta   # entrar a una carpeta
cd ..                  # subir un nivel
ls                     # listar archivos (Linux/macOS)
dir                    # listar archivos (Windows)
pwd                    # mostrar la carpeta actual
```

Para ejecutar un script de Python:

```
python mi_script.py
```

O, como veremos pronto, a través de `uv`:

```
uv run python mi_script.py
```

### Leer un mensaje de error

Cuando algo falla, Python imprime un *traceback*: una traza de las llamadas que llevaron al error. Se lee **de abajo hacia arriba**. La última línea indica el tipo de error y el mensaje; las líneas anteriores muestran dónde ocurrió y desde dónde se llamó. La línea relevante suele ser la segunda desde el fondo.

```
Traceback (most recent call last):
  File "mi_script.py", line 12, in <module>
    render(escena)
  File "mi_script.py", line 7, in render
    pipeline.use()
AttributeError: 'NoneType' object has no attribute 'use'
```

En este ejemplo, el error real es que `pipeline` es `None`, probablemente porque no fue inicializado antes de usarse.

### Rutas de archivos

Un programa gráfico carga muchos tipos de archivos: *shaders* (`.glsl`), texturas (`.png`, `.jpg`), modelos 3D (`.obj`, `.ply`, `.glb`), entre otros. Para cargarlos, Python necesita saber dónde están; y para eso, es necesario entender cómo funcionan las rutas.

Una **ruta absoluta** describe la ubicación completa desde la raíz del sistema de archivos:

```
/home/usuario/curso/shaders/vertex.glsl       # Linux / macOS
C:\Users\usuario\curso\shaders\vertex.glsl     # Windows
```

Una **ruta relativa** describe la ubicación a partir de un punto de referencia, que en Python es el **directorio de trabajo** (*working directory*): la carpeta desde la que se ejecuta el *script*. Es lo que muestra `pwd` en la terminal.

```
# Si el working directory es /home/usuario/curso/,
# esta ruta relativa apunta a /home/usuario/curso/shaders/vertex.glsl
open("shaders/vertex.glsl")
```

El error más común con rutas es ejecutar el *script* desde la carpeta equivocada:

```
FileNotFoundError: [Errno 2] No such file or directory: 'shaders/vertex.glsl'
```

El archivo existe, pero Python lo busca relativo al directorio de trabajo actual, no relativo al *script*. Por eso importa dónde ejecutas el comando en la terminal.

La solución robusta es anclar las rutas a la ubicación del propio *script* usando `Path(__file__)`, que contiene la ruta del archivo Python en ejecución:

```
from pathlib import Path

base = Path(__file__).parent          # carpeta donde está el script

vertex_src  = base / "shaders" / "vertex.glsl"
textura     = base / "assets"  / "ladrillo.png"
modelo      = base / "modelos" / "conejo.obj"

with open(vertex_src) as f:
    codigo = f.read()
```

El operador `/` en `Path` concatena segmentos de ruta de forma portátil: funciona igual en Linux, macOS y Windows, sin preocuparse por las diferencias entre `/` y `\`.

## 3 Git y el repositorio del curso

Git es un sistema de control de versiones: registra la historia de cambios de un proyecto y permite recuperar versiones anteriores, trabajar en ramas paralelas, y colaborar con otras personas. En este curso lo usarás principalmente para una sola cosa: obtener (y mantener actualizado) el código del curso.

Nuestro repositorio está en GitHub, un servidor de Git administrado por Microsoft. Para clonarlo en tu computador ejecuta lo siguiente en la terminal:

```
git clone https://github.com/PLUMAS-research/cc3501-computer-graphics
cd cc3501-computer-graphics
```

Esto crea una carpeta local con todo el código llamada igual que el repositorio (`cc3501-computer-graphics`) y, una vez creada, entra en ella. Cuando el repositorio se actualice con nuevos ejemplos, podrás sincronizarla con:

```
git pull
```

No es necesario que sepas crear *commits*, trabajar con ramas, ni resolver conflictos para este curso. Pero si modificas los archivos de ejemplo y luego haces `git pull`, Git puede reportar conflictos. La forma más simple de evitarlo: trabaja en copias de los archivos, no en los originales.

# 4 Editor de código

Para escribir código necesitas un editor. Recomendamos Zed, un editor moderno, rápido y con soporte nativo para Python. Tiene resaltado de sintaxis, autocompletado, integración con Git y una CLI que permite abrirlo desde la terminal con `zed .` (el punto se refiere a la carpeta en la que ejecutas el comando).

Si ya usas otro editor (VS Code, Neovim, PyCharm), puedes seguir con él sin problemas. Lo importante es que el editor entienda Python y que puedas ejecutar *scripts* desde la terminal integrada o desde una terminal externa.

Editar archivos y ejecutarlos en la terminal no es la única manera de trabajar. Zed y VS Code también permiten ejecutar el código de forma **interactiva**: el archivo se divide en celdas separadas por el marcador `# %%`, y cada celda se ejecuta de manera independiente con un atajo de teclado. El resultado aparece de inmediato, sin relanzar el programa completo. Este flujo es muy útil para el curso y se explica en detalle más adelante.

# 5 Python en este curso

## Módulos, paquetes y librerías

En Python, un **módulo** es simplemente un archivo `.py`. Un **paquete** es una carpeta que contiene módulos (y un archivo especial de nombre `__init__.py`, que puede estar vacío). Una **bi-**

**blioteca** (a la que se le dice *library* en inglés, y por eso a veces se le llama de manera incorrecta como "librería") es una colección de paquetes y módulos reutilizables. En el caso de una biblioteca pública, se instala con un gestor como `pip` o `uv`.

Los paquetes (incluyendo una biblioteca que en sí misma es un paquete) se importan así:

```
import numpy as np          # importa el módulo completo con alias np
from pathlib import Path    # importa solo lo que necesitas
```

Las librerías principales que usaremos en el curso son:

- `pyglet`: gestión de ventanas, E/S (teclado, *mouse*) y acceso a OpenGL desde Python.
- `numpy`: vectores, matrices y operaciones algebraicas sobre ellos.
- `trimesh`: carga y manipulación de mallas geométricas 3D.
- `pymunk`: simulación de física de cuerpos rígidos en 2D.
- `mesa`: simulación basada en agentes (*Agent-Based Modeling*).
- `grafica`: módulo propio del curso, incluido en el repositorio.

### El módulo `grafica`

El repositorio del curso contiene un paquete llamado `grafica` que complementa a `pyglet` con herramientas que desarrollaremos a lo largo del semestre. No es una biblioteca externa porque vive directamente en el repositorio, así que no requiere instalación adicional.

Sus componentes principales son:

- `grafica.transformations`: matrices de transformación (traslación, rotación, escala) listas para usar con OpenGL.
- `grafica.scenegraph`: implementación de grafos de escena basada en `networkx`.
- `grafica.textures`: utilidades para cargar y gestionar texturas.
- `grafica.particle`: estructuras para simulación de partículas.
- `grafica.math`: funciones matemáticas auxiliares.
- `grafica.intersections`: pruebas de intersección geométrica.
- `grafica.arcball`: control de cámara interactivo mediante *arcball*.

Se importa igual que cualquier otro módulo Python, siempre que el *script* se ejecute desde la raíz del repositorio:

```
import grafica.transformations as tr
from grafica.scenegraph import Scenegraph
```

## Estructuras del lenguaje que usaremos

### Listas, tuplas y *arrays*

Python tiene tres formas principales de agrupar valores numéricos, y en este curso las tres aparecen con roles distintos.

Una **lista** es una secuencia mutable de longitud variable. Es lo que más se usa en cursos introductorios de Python:

```
colores = ["rojo", "verde", "azul"]
colores.append("amarillo")  # se puede modificar
```

Una **tupla** es una secuencia inmutable. Se usa para representar valores que conceptualmente son una unidad y no deberían cambiar: una coordenada, un color, un tamaño:

```
posicion  = (0.5, -1.0, 0.0)   # x, y, z
color_rojo = (1.0, 0.0, 0.0)   # r, g, b
resolucion = (1280, 720)        # ancho, alto
```

Un **array de NumPy** (`np.ndarray`) es una colección de valores del mismo tipo numérico, almacenados de manera compacta en memoria[2]:

```
vertices = np.array([
    -1.0, -1.0, 0.0,
     1.0, -1.0, 0.0,
     1.0,  1.0, 0.0,
], dtype=np.float32)
```

[2]Esto es algo que se puede hacer en bajo nivel. Es importante porque si los números están separados en memoria, para hacer cálculos hay que realizar múltiples operaciones de lectura; es decir, es más caro.

Además de asegurar que sus elementos son contiguos, un *array* tiene tipo (`dtype`). En GLSL, el tipo `float` es de 32 bits: `vec3`, `mat4` y todos los tipos que usaremos en los *shaders* operan en esa precisión. NumPy usa `float64` por omisión. Aunque `pyglet` puede encargarse de la conversión implícitamente al copiar los datos a la GPU, es buena práctica especificar el tipo explícitamente, como en el ejemplo: comunica la intención, evita conversiones innecesarias, y es consistente con lo que el *shader* declara.

La diferencia práctica entre ambas precisiones es la cantidad de decimales significativos: `float32` representa aproximadamente 7 dígitos; `float64`, unos 15. Para *rendering* en tiempo

real, 7 son más que suficientes: la posición de un vértice en pantalla no necesita más precisión que la que distingue un píxel del siguiente. Donde `float64` importa es en simulaciones científicas con acumulación de errores numéricos a lo largo de muchos pasos. La otra diferencia es el tamaño en memoria: `float64` ocupa el doble que `float32`, lo que en un modelo con cientos de miles de vértices se nota.

### Ciclos `for` versus operaciones vectorizadas

La forma natural de procesar una colección es mediante un ciclo `for`:

```
# escalar todos los vértices por 2.0: forma lenta
vertices_escalados = []
for i in range(0, len(vertices), 3):  # grupos de x, y, z
    vertices_escalados.append(vertices[i]     * 2.0)
    vertices_escalados.append(vertices[i + 1] * 2.0)
    vertices_escalados.append(vertices[i + 2] * 2.0)
```

En NumPy, la misma operación se expresa sobre el *array* completo:

```
# escalar todos los vértices por 2.0: forma rápida
vertices_escalados = vertices * 2.0
```

Es más corto, pero además, NumPy ejecuta la operación en C sobre todos los elementos a la vez, sin el *overhead* de Python en cada iteración. Con miles de vértices (lo normal en un modelo 3D), la diferencia es apreciable.

El mismo principio aplica a operaciones más complejas, como calcular la longitud de cada vector en una lista:

```
# Con for:
longitudes = []
for i in range(0, len(vertices), 3):
    x, y, z = vertices[i], vertices[i+1], vertices[i+2]
    longitudes.append((x**2 + y**2 + z**2) ** 0.5)

# Con NumPy, reinterpretando el array como matriz de filas (x, y, z):
v = vertices.reshape(-1, 3)             # convierte a matriz de Nx3
longitudes = np.linalg.norm(v, axis=1)  # norma de cada fila
```

Para multiplicar matrices, NumPy ofrece el operador `@`:

```
# Con listas anidadas y for:
resultado = [[sum(A[i][k] * B[k][j] for k in range(n))
              for j in range(m)] for i in range(p)]

# Con NumPy:
resultado = A @ B
```

Además de ser más eficiente en términos de *performance*, este código es más legible y menos propenso a errores. Por eso, como regla, si ves un `for` que recorre números para hacer aritmética, entonces probablemente hay una operación (o combinación de operaciones) de NumPy que lo reemplaza.

### Diccionarios

Un diccionario (`dict`) asocia llaves a valores. Es útil para agrupar propiedades relacionadas bajo un nombre, evitando variables sueltas:

```
# Sin diccionario
material_color     = (0.8, 0.2, 0.1)
material_shininess = 32.0
material_texture   = None

# Con diccionario
material = {
    "color":     (0.8, 0.2, 0.1),
    "shininess": 32.0,
    "texture":   None,
}
```

En el ejemplo, `material` es un diccionario con tres llaves. Se accede a sus valores mediante ellas:

```
color = material["color"]

if material["texture"] is not None:
    bind_texture(material["texture"])
```

En el curso los usarás, por ejemplo, para mapear nombres a recursos ya cargados (*shaders*, texturas) y para representar propiedades de objetos de la escena antes de enviarlos a la GPU. Un patrón común es construir el diccionario dinámicamente:

```
texturas = {}
for nombre in ["ladrillo", "madera", "metal"]:
```

```
    texturas[nombre] = cargar_textura(f"assets/{nombre}.png")

# luego:
textura_activa = texturas["ladrillo"]
```

### Celdas y REPL

El flujo más directo para explorar código Python es el *REPL* (*Read-Eval-Print Loop*): una sesión interactiva en la terminal donde cada expresión se ejecuta de inmediato y el resultado aparece en pantalla. Para probar una línea o verificar un cálculo es muy cómodo, pero se vuelve incómodo cuando hay que repetir varias instrucciones en secuencia o retomar el estado de una sesión anterior.

Una alternativa que combina lo mejor de los archivos y del *REPL* es dividir el código en **celdas**. Una celda es un bloque delimitado por el marcador `# %%`:

```
# %% Celda 1: Importaciones
import numpy as np
import grafica.transformations as tr

# %% Celda 2: Construir una matriz de rotación
R = tr.rotationZ(np.pi / 4)
print(R)

# %% Celda 3: Verificar que preserva la norma del vector
v = np.array([1.0, 0.0, 0.0, 1.0], dtype=np.float32)
v_rot = R @ v
print(f"norma: {np.linalg.norm(v_rot[:3]):.4f}")  # debe ser ~1.0
```

Cada celda se ejecuta de manera independiente: puedes correr solo la que necesitas, modificarla y volver a ejecutarla sin tocar las demás. Zed y VS Code detectan los marcadores `# %%` y permiten ejecutar cada celda con un atajo de teclado (`Ctrl+Shift+Enter` en Zed; `Shift+Enter` en VS Code). El resultado aparece de inmediato en un panel lateral o en la consola.

Cuando estés construyendo una escena, calculando una transformación o *debuggeando* por qué un vértice aparece en la posición equivocada, no necesitas ejecutar todo de nuevo: puedes aislar el cálculo en una celda, ejecutarlo, verificar los valores y ajustar el código si es necesario. Una vez que el fragmento funciona como esperas, no necesitas hacer nada más: ya está integrado en el archivo `.py`.

## Cómo difiere este curso de otros usos de Python

Si has usado Python antes (en un curso de programación introductoria, de análisis de datos o de *machine learning*), probablemente tienes un modelo mental de cómo funciona un programa Python:

```
importar → inicializar → procesar → mostrar resultado → terminar
```

En Computación Gráfica ese modelo **no aplica**. Un programa gráfico no termina cuando terminó de calcular algo: vive en un bucle continuo mientras la ventana esté abierta.

### El *game loop*

La estructura de una aplicación gráfica incorpora un bucle de ejecución potencialmente infinito llamado *game loop*[3]. Cada iteración comprende al menos tres operaciones:

- **Entrada**: leer teclado, *mouse*, controladores, u otras fuentes de interacción.
- **Actualización**: modificar el estado del mundo virtual según el tiempo transcurrido y las interacciones.
- **Renderizado**: graficar el nuevo estado usando *shaders* en la GPU.

[3]El nombre viene de los videojuegos, que fueron los primeros en necesitar esta estructura de manera sistemática.

En `pyglet`, este ciclo se gestiona internamente a través de la **ventana**, un rectángulo en la pantalla que está bajo tu control. Se crea con el siguiente código:

```
win = pyglet.window.Window(width, height)
```

La ventana `win` es el objeto que posee el *game loop*: tiene su propio mecanismo de eventos, sabe cuándo hay que redibujar, cuándo se presionó una tecla, cuándo se movió el *mouse*. Es, en cierto sentido, un programa paralelo al tuyo que `pyglet` gestiona por debajo: tú no ves ese bucle en tu código, pero está corriendo.

Tu código se conecta a la ventana mediante **decoradores**[4]. Un decorador como `@win.event` registra tu función en la ventana, asociándola a un evento específico según como se llame tu función, así:

[4]Un decorador es una función que recibe otra función y modifica o extiende su comportamiento. Puedes leer más en: https://python-intermedio.readthedocs.io/es/latest/decorators.html.

```
@win.event
def on_draw():
    win.clear()
    pipeline.use()
    gpu_data.draw(GL.GL_TRIANGLES)
```

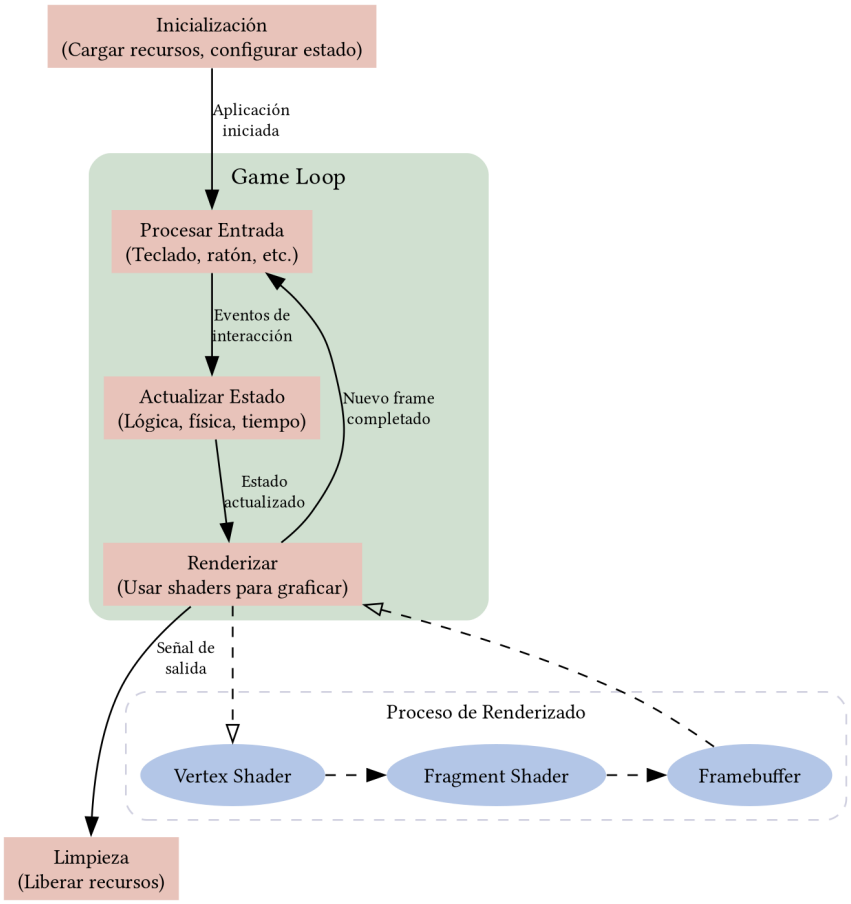


**Figura 1.** Diagrama esquemático de un *game loop* típico.

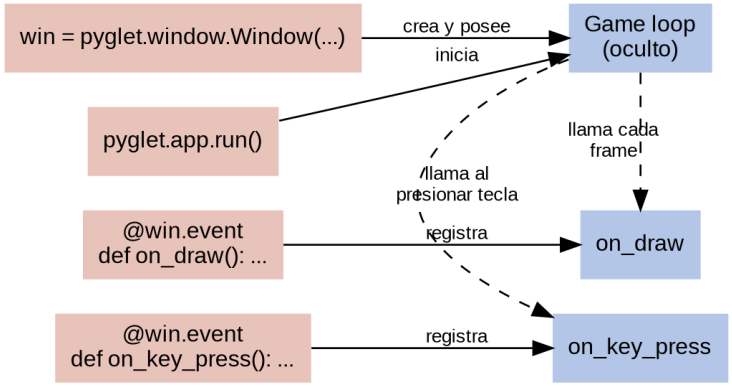


**Figura 2.** Relación entre tu código y el *game loop* interno de `pyglet` mediante decoradores.

En el ejemplo, tú no ejecutas `on_draw()`: la ventana `win` la llama por ti cada vez que necesita redibujar. Lo mismo ocurre con otros eventos (`on_key_press`, `on_mouse_drag`, `on_resize`) que puedes registrar del mismo modo si tu programa necesita reaccionar a ellos. Esta inversión de control (*el sistema llama tus funciones*, no al revés) es lo que caracteriza la programación orientada a eventos[5].

El bucle se inicia explícitamente al final del programa:

[5]Este es un paradigma de programación del que podría hacerse un curso completo. Sin embargo, para nuestro curso es suficiente con saber lo que contiene esta unidad.

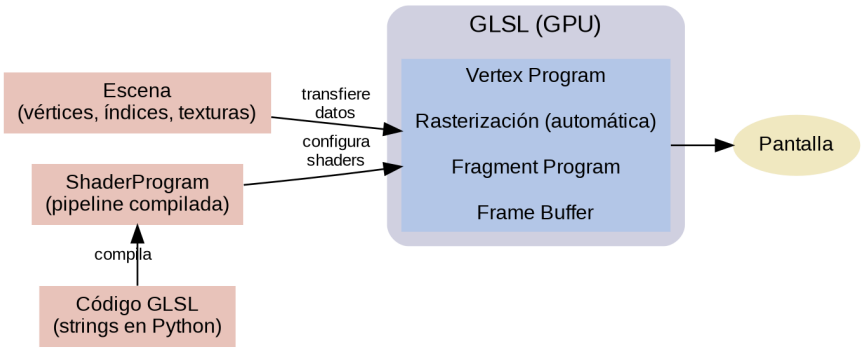


**Figura 3.** El programa "vive" en dos lugares: Python en orquesta la escena en la CPU; GLSL determina cómo la GPU ejecuta los cálculos de cada vértice y cada píxel.

```
# antes de esta línea va todo tu código
pyglet.app.run()
```

Esta instrucción transfiere el control a `pyglet`, que gestionará el ciclo hasta que la persona cierre la ventana. Todo lo que escribas después de esta línea no se ejecutará mientras la ventana esté abierta.

### GLSL: código que corre en la GPU

Los *shaders* (los programas que controlan cómo se ve cada vértice y cada píxel) no se escriben en Python. Se escriben en GLSL (*OpenGL Shading Language*), un lenguaje similar a C con tipos explícitos. Desde Python, un *shader* no es más que un string que se compila y se envía a la GPU en tiempo de ejecución. Esto introduce complejidades de implementación puesto que ya un programa gráfico tiene una dualidad entre tu código y el código de la ventana. Una dificultad concreta es interpretar mensajes de error: los errores de GLSL no se ven igual que los de Python, puesto que aparecen en la consola como mensajes del *driver* gráfico, sin tener contexto del código que le rodea ni de las condiciones en las que sucedió. Veremos ejemplos concretos en la unidad de GPU y OpenGL.

## 6 Entornos virtuales y uv

### El problema de las dependencias

Distintos proyectos pueden necesitar versiones distintas de una misma librería. Si instalas todo en el sistema operativo de manera global, más temprano que tarde dos proyectos van a entrar en conflicto. La solución es un **entorno virtual** (*virtual env*): una carpeta aislada que contiene su propia instalación de Python y sus propias bibliotecas, independiente del resto del sistema.

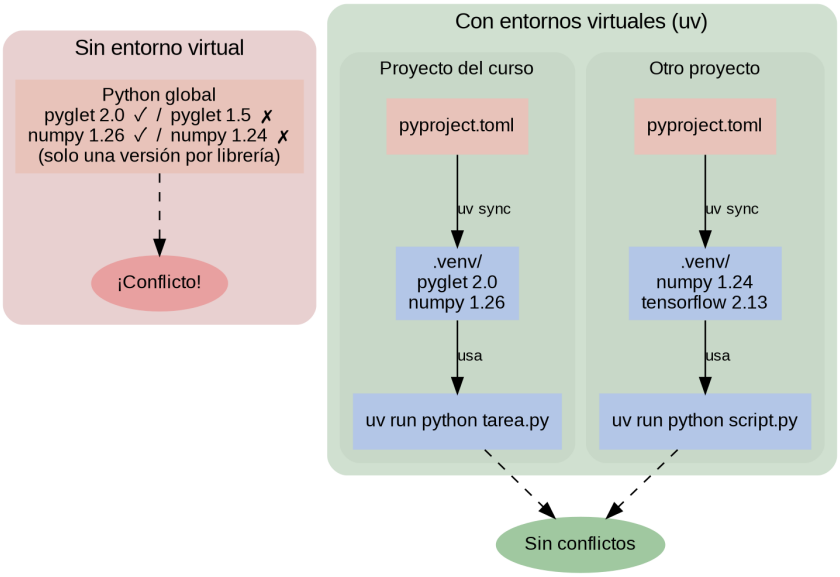


**Figura 4.** Sin entornos virtuales, todos los proyectos comparten las mismas versiones de las bibliotecas, lo que genera conflictos. Con `uv`, cada proyecto tiene su propio `.venv` aislado.

## `uv` como gestor de entornos

`uv` es un gestor moderno de entornos y dependencias para Python[6]. Reproduce el entorno necesario para un programa de manera confiable a partir de un archivo de configuración (`pyproject.toml`), garantizando que todas las personas (y hoy, *agentes* LLM) que lo ejecuten usen exactamente las mismas versiones de las bibliotecas requeridas.

[6] Sitio oficial: https://docs.astral.sh/uv/.

Para instalar `uv` puedes ejecutar un comando en tu consola:

```
# Linux / macOS
curl -LsSf https://astral.sh/uv/install.sh | sh

# Windows (PowerShell)
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Una vez dentro de la carpeta del repositorio, no es necesario instalar nada manualmente. `uv run` detecta la configuración en `pyproject.toml`, crea el entorno virtual (en `.venv/`, el punto al inicio del nombre de carpeta indica que está oculta) e instala todas las dependencias la primera vez que se ejecuta:

```
uv run python caja_de_juguetes.py hello_world
```

No hay que activar el entorno ni correr `uv sync` por separado; `uv run` hace todo eso automáticamente. Si en algún momento quieres preparar el entorno antes de ejecutar cualquier ejemplo (por ejemplo, para instalarlo sin conexión después), puedes hacerlo explícitamente:

```
uv sync
```

El resultado es el mismo: tu *virtual env* configurado. Esto también garantiza que quien corrija tu tarea use el mismo entorno que tú, evitando el clásico "en mi computador funciona".

# 7 Ejecutar e iterar

Todos los ejemplos del repositorio se ejecutan desde la raíz del proyecto con `uv run`:

```
uv run python programa.py
```

El flujo típico cuando experimentas con el código es:

1. Editar el archivo en tu editor.
2. Guardar los cambios.
3. Ejecutar en la consola(`uv run python ...`).
4. Observar el resultado en la ventana y los mensajes en la consola.
5. Repetir.

Si hay un error de sintaxis, aparecerá inmediatamente en la consola (y si prestas atención y usas un buen editor, probablemente ya estaba subrayado en el código). Si el error es lógico (la ventana abre pero no muestra lo esperado), hay que razonar e hipotetizar respecto al porqué del comportamiento distinto. Aquí es donde es valiosa la ejecución por celdas, porque puedes cerrar la ventana, modificar algunas celdas, y ejecutar de nuevo, sin necesidad de cargar todos los archivos necesarios de nuevo. Es más rápido, y más rapidez en el desarrollo permite que pienses y analices con más profundidad y dedicación.

## *Scripts* de ejemplo con `click`

Tanto los ejemplos del curso como las tareas usan `click`[7] para construir su interfaz de línea de comandos. `click` permite definir comandos con opciones y argumentos de manera declarativa, lo que produce scripts con `--help` informativo y comportamiento predecible.

[7] Documentación de `click`: https://click.palletsprojects.com/.

El punto de entrada de todos los ejemplos del curso es `caja_de_juguetes.py`, un *script* `click` que agrupa cada ejemplo como un subcomando. Para ver la lista de ejemplos disponibles, ejecuta:

```
uv run python caja_de_juguetes.py
```

Para ejecutar un ejemplo específico:

```
uv run python caja_de_juguetes.py hello_world
```

Algunos ejemplos reciben argumentos posicionales:

```
uv run python caja_de_juguetes.py image_texture assets/dice.jpg
```

Otros tienen opciones con valores por omisión:

```
uv run python caja_de_juguetes.py sr_jengibre --x0 10 --y0 -1
```

**Tareas**

Utilizaremos un esquema similar para las tareas. El código de cada tarea estará en una carpeta que tendrá tu RUT como nombre (por ejemplo, `12345678-9`). Se ejecutará como módulo Python:

```
uv run python -m 12345678-9
```

Esto es posible porque les entregaremos una carpeta base sobre la que deben programar sus tareas. La carpeta base incluye un *script* `click`. Así, la corrección se realizará **ejecutando el módulo**, no solo leyendo el código. Quien revise tu tarea correrá los comandos indicados en el enunciado y evaluará el resultado.