Introducción a la GPU y al *rendering* con OpenGL

Eduardo Graells-Garrido

18 de marzo de 2026

1 La evolución gráfica

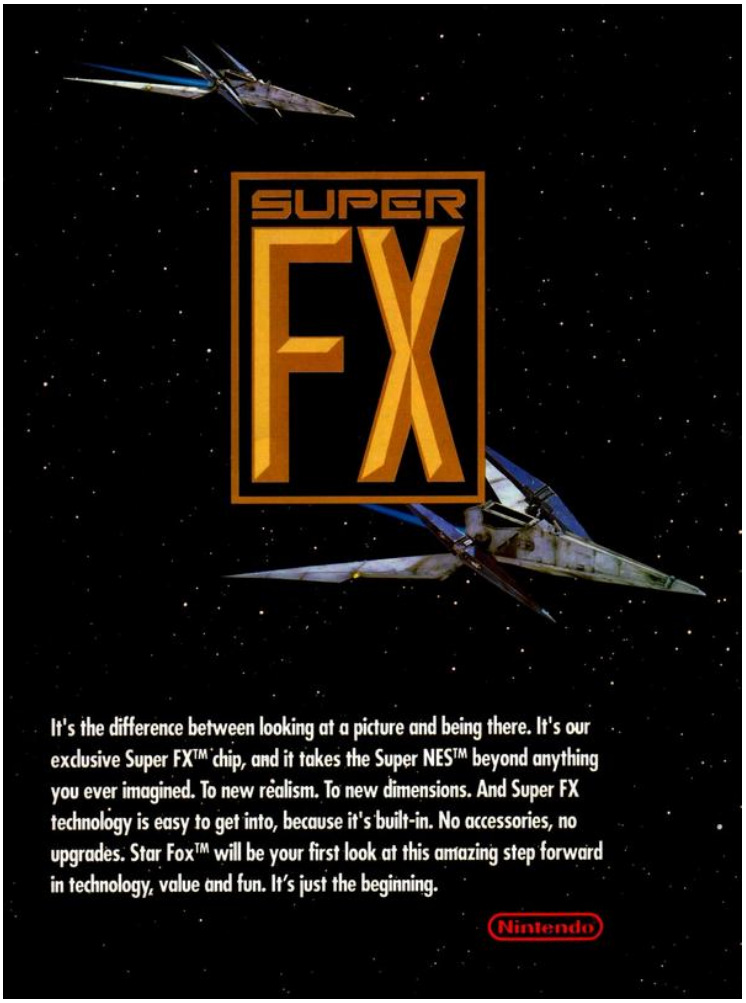


Figura 1. Aviso publicitario del chip Super FX que se añadió a algunos *cartridges* de juegos de Super Nintendo: "Es la diferencia entre ver una imagen y estar ahí".

Viajemos al pasado, a una época donde los computadores generaban gráficos a un ritmo muy distinto al actual. En aquellos tiempos, la capacidad de generar gráficos 3D en tiempo real (es decir, calcular la geometría, iluminación y color de una escena tridimensional lo suficientemente rápido como para desplegarla de forma interactiva) representaba una innovación tecnológica relevante. Los primeros computadores y consolas de videojuegos ni siquiera contaban con esta capacidad (hoy) básica, y cuando la obtuvieron, la velocidad de procesamiento gráfico se convirtió en el siguiente desafío a superar (Fig. 1).

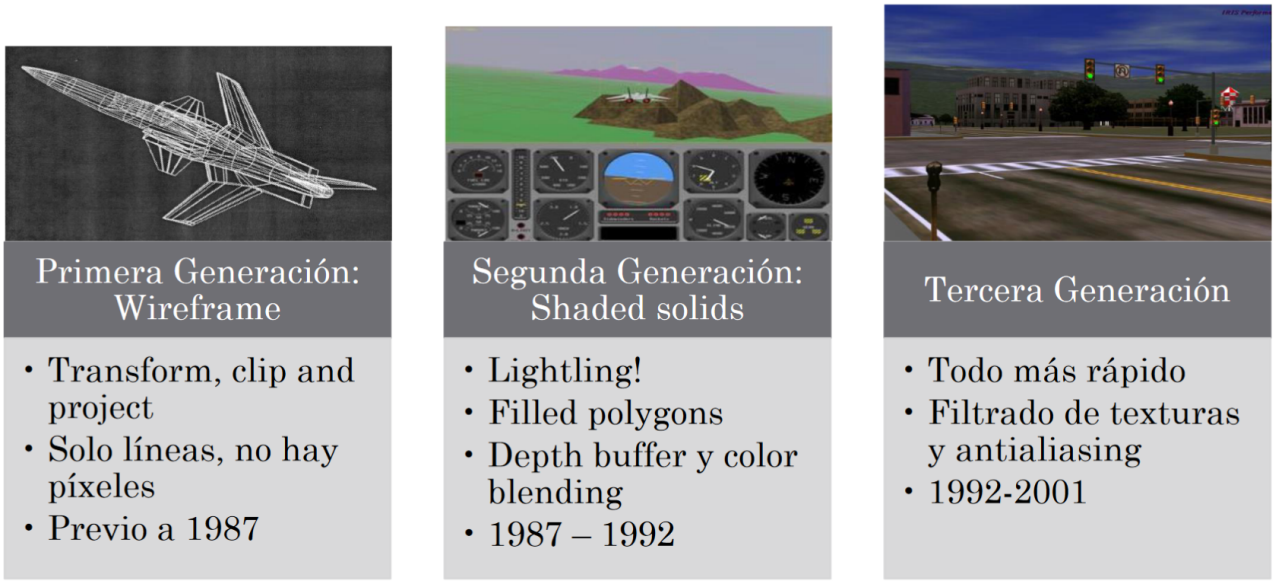


Figura 2. Evolución de los gráficos en tres generaciones: a) *Wireframe* (solo bordes de polígonos); b) *Shaded solids* (polígonos rellenados con color); c) Aplicación de texturas y diferentes filtros para mejorar la calidad de imagen. Fuente: The Evolution of GPUs for General Purpose Computing.

Hace algunas décadas, las arquitecturas GPU (Unidad de Procesamiento Gráfico o *Graphics Processing Unit*) no existían. Las tarjetas gráficas de entonces cumplían funciones más limitadas: establecer comunicación con el monitor, gestionar la imagen en memoria o, como mucho, procesar gráficos 2D básicos. Con el avance tecnológico (la Fig. 2 muestra las diferentes "generaciones" de este avance) surgieron diversas tarjetas con chips especializados como PowerVR[1], 3dfx Voodoo y otras. Cada fabricante desarrollaba su propia biblioteca de *rendering* y controladores específicos, ya que sus tecnologías funcionaban de manera independiente y seguían enfoques propios.

El panorama cambió en 1999 cuando NVIDIA lanzó la tarjeta GeForce 256, presentándola como "la primera GPU"[2]. Si bien no era la primera en términos conceptuales (sus capacidades ya existían), su relevancia radica en que, en lugar de ofrecer una arquitectura *ad hoc*, NVIDIA logró sistematizarla y hacerla accesible para los consumidores (especialmente *gamers*). Así se popularizó el término GPU que hoy utilizamos de manera ubicua, a pesar de que la tecnología y la calidad gráfica ha seguido evolucionando (Fig. 3).

[1] Puedes ver esta nota en el programa *Cybernet* de los 90.

[2] Puedes ver un documental en YouTube.

Figura 3. La evolución en el personaje Malcom de *Unreal Tournament* en las primeras iteraciones de *Unreal Engine* nos permite apreciar la evolución de las capacidades gráficas.

NVIDIA definió la GPU como "un procesador de chip único que integra transformaciones, iluminación, configuración y recorte de triángulos, y un motor de graficación (*rendering*) capaz de procesar un mínimo de 10 millones de polígonos por segundo". Ahora bien, la GPU es el procesador en sí, no la tarjeta completa. De hecho, una GPU puede presentarse de diferentes formas (Fig. 4), cada una con sus propias características y ventajas. La **placa discreta** es una tarjeta dedicada al procesamiento gráfico y contiene múltiples *cores* (núcleos) distribuidos en diversos chips, junto con memoria especializada (VRAM) para almacenar texturas, modelos y otros datos gráficos. Un ejemplo contemporáneo sería la tarjeta GeForce RTX 4070. En cambio, un **procesador integrado** se trata de un chip de procesamiento gráfico con múltiples *cores* que forma parte de la CPU. En lugar de contar con memoria dedicada, tiene acceso a la RAM del sistema. Un ejemplo actual es el procesador Apple M4.

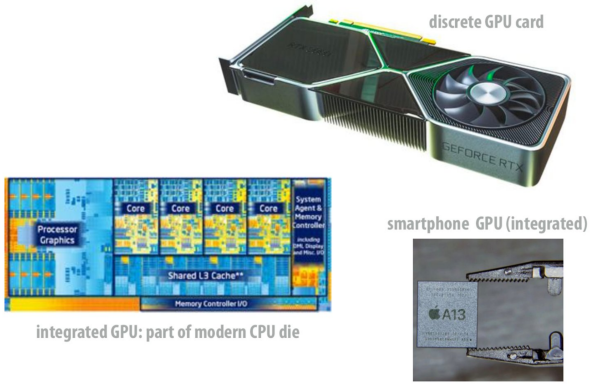


Figura 4. Distintas presentaciones de una GPU. La tarjeta o placa contiene la GPU pero también el *video controller*, la memoria de video (VRAM), otros procesadores auxiliares, y ventiladores y otros sistemas de control de temperatura.

A diferencia de sus predecesoras, las GPU modernas son procesadores programables. Las primeras generaciones operaban con una *pipeline*[3] fija de funcionalidad limitada (*fixed pipeline*) que, en el mejor de los casos, permitía configuración de algunas de sus capacidades. Aunque las GPU actuales todavía incorporan co-procesadores para operaciones gráficas específicas (como el *rastering*, que sigue siendo automático y no programable), la mayor parte de sus operaciones son programables.

Para entender la diferencia entre CPU y GPU, es útil examinar las componentes internas de cada procesador (Fig. 5). Ambos comparten los mismos tipos de componentes, pero en proporciones distintas:

- ***Core* (núcleo)**: es la unidad que ejecuta instrucciones y realiza cálculos aritméticos. Un *core* complejo puede ejecutar operaciones variadas y sofisticadas; uno simple está optimizado para realizar operaciones matemáticas básicas con mucha rapidez.
- **Unidad de control**: coordina la ejecución de instrucciones dentro de un *core*. Decide qué operación ejecutar, en qué orden, y gestiona el flujo del programa (por ejemplo, saltos condicionales). A mayor complejidad de las instrucciones soportadas, mayor proporción del *core* se dedica a control.
- **Caché**: memoria pequeña y extremadamente rápida que almacena datos de acceso frecuente. Se organiza en niveles jerárquicos (L1, L2, L3), donde L1 es la más rápida (y más pequeña) y L3 la más lenta (y más grande). Más caché permite reutilizar datos sin tener que buscarlos en la memoria principal.
- **Memoria (DRAM)**: almacenamiento principal de datos. En una CPU corresponde a la RAM del sistema; en una GPU discreta corresponde a la VRAM, memoria dedicada que reside en la propia tarjeta gráfica.

[3] Su significado literal es "tubería". En la práctica, es una secuencia de operaciones que se aplica a los datos que entran por un lado, y, en este caso, producen la imagen que sale por el otro lado.

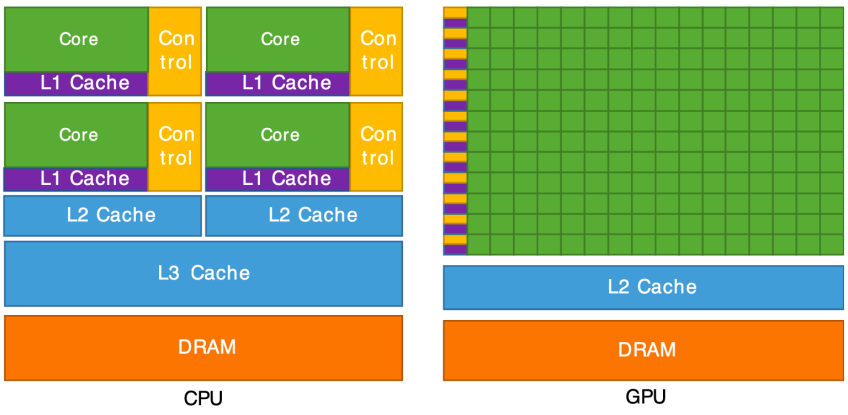


Figura 5. Diferencia conceptual en los tipos de transistores que tienen CPU y GPU. Fuente: *CUDA C Programming Guide* https://docs.nvidia.com/cuda/archive/11.2.0/pdf/CUDA_C_Programming_Guide.pdf. Puedes ver una explicación didáctica por los *Mythbusters* en este video: https://www.youtube.com/watch?v=ZrJeYFxpUyQ.

La Fig. 5 ilustra cómo se distribuyen estos componentes en cada procesador. La siguiente tabla resume las diferencias:

Componente	CPU	GPU
Cores	Pocos (4-24), grandes y complejos	Miles, pequeños y simples
Unidad de control	Grande, dedicada a cada *core*	Mínima, compartida entre muchos *cores*
Caché	Jerárquica en tres niveles (L1, L2, L3)	Reducida (solo L2)
Memoria	RAM del sistema	VRAM dedicada (o RAM compartida si es integrada)
Modelo de ejecución	Tareas diversas en paralelo (MIMD)	Misma operación sobre muchos datos en paralelo (SIMD)

Una CPU tiene pocos *cores*, pero cada uno es capaz de ejecutar instrucciones variadas y complejas de forma independiente. Esto se conoce como MIMD (*Multiple Instruction Multiple Data*): cada *core* puede estar haciendo algo distinto. Las grandes unidades de control y la jerarquía de caché (L1, L2, L3) permiten manejar eficientemente programas con muchas ramificaciones, accesos a datos impredecibles y coordinación entre tareas. Es un procesador generalista.

Una GPU, en cambio, implementa una arquitectura SIMD (*Single Instruction Multiple Data*): aplica las mismas instrucciones a múltiples datos de manera simultánea. Por eso tiene miles de *cores* simples en lugar de pocos complejos. La unidad de control es mínima (todos los *cores* ejecutan lo mismo) y la caché es reducida, porque cada elemento se procesa una vez y se avanza al siguiente, sin necesidad de revisitar datos previos. Esta arquitectura permite procesar los millones de polígonos por segundo necesarios en aplicaciones 3D.

La diferencia en potencia computacional bruta es notable (Fig. 6). Esta capacidad ha permitido que las GPU trasciendan su propósito original como *hardware* gráfico, encontrando aplicación en otros campos. En la actualidad, el *deep learning* representa su uso alternativo más prominente, mientras que hace algunos años, la "minería" de criptomonedas provocó una escasez en el mercado y un incremento significativo en los precios de estas tarjetas.

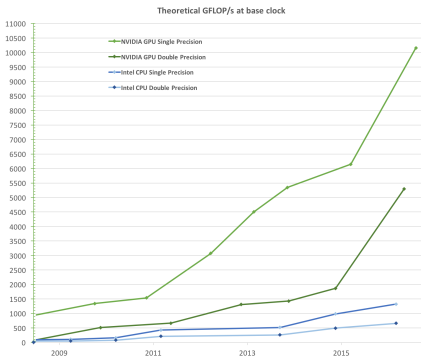
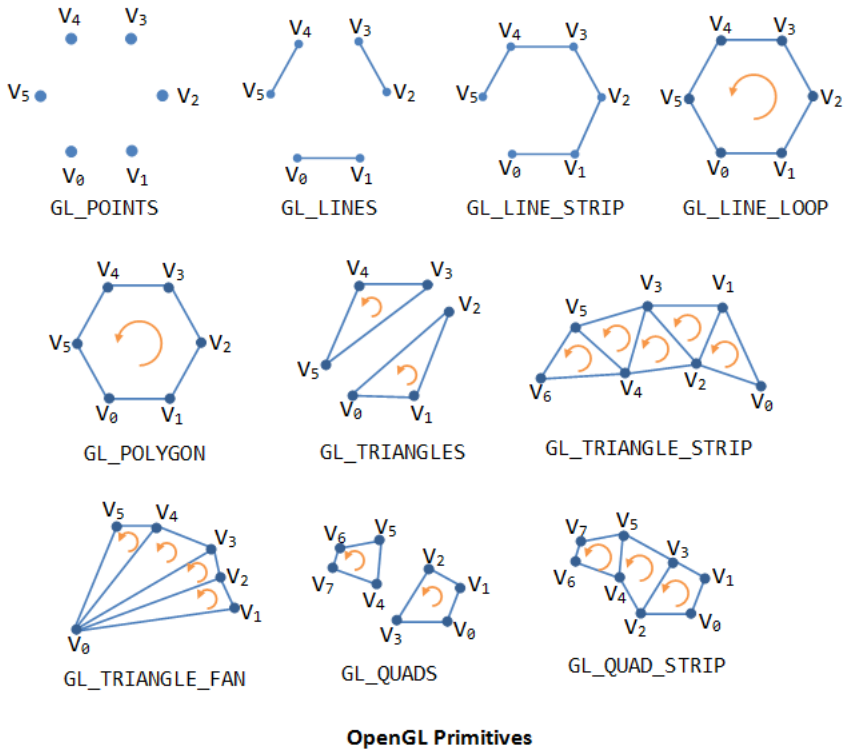


Figura 6. Comparación en potencia entre GPU y CPU. Fuente: CUDA C Programming Guide (v.9.1).

2 GPU y *rendering*

La renderización (*rendering*) es parte de un proceso llamado *rendering pipeline* que transforma la descripción de una escena en una imagen 2D (Fig. 7). Su naturaleza secuencial explica por qué se le denomina *pipeline*, donde cada componente procesa la información y la transmite al siguiente. A grandes rasgos, las etapas que trabajaremos en el curso son: 1) Aplicación, 2) Procesamiento geométrico, 3) Recorte, 4) Rasterización y 5) Procesamiento de píxeles. Hay otras etapas de la *pipeline* como el procesamiento del Controlador de video y el procesamiento de señal en el monitor (o el dispositivo que despliegue la imagen).

El punto de partida es una escena. Un ejemplo clásico es la Caja de Cornell[4], diseñada en 1984 (Fig. 8). Esta escena contiene múltiples objetos y una fuente de luz, y sus elementos poseen propiedades como colores y capacidad reflectante. Es una escena de referencia porque se conocen sus propiedades físicas y geométricas exactas, entonces es posible comparar una fotografía real de alta calidad con la imagen generada por una *rendering pipeline*.



La etapa de **Aplicación** se encarga de construir, actualizar y mantener el mundo virtual que se renderizará. Este componente realiza simulaciones, captura entradas del sistema, monitorea el estado de la red y ejecuta otras tareas similares. Una vez

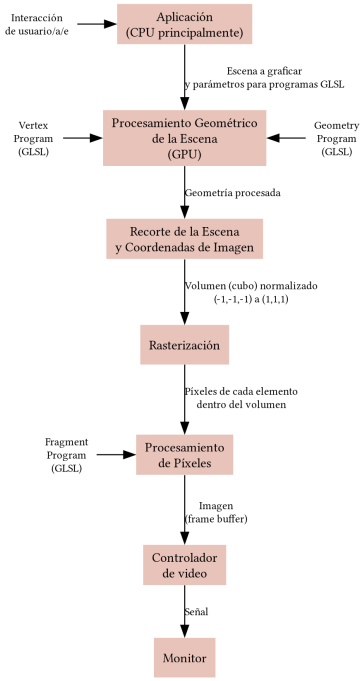


Figura 7. Versión básica de la *Rendering Pipeline*.

[4] https://www.graphics.cornell.edu/online/box/data.html

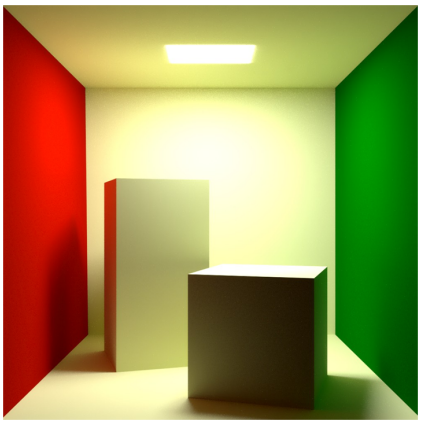


Figura 8. Una imagen de la Caja de Cornell.

Figura 9. Primitivas disponibles en OpenGL. Fuente: https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html.

construida la escena, se transmite a la GPU en forma de lista de primitivas gráficas (Fig. 9): puntos, líneas y triángulos. De estas, los triángulos son la primitiva fundamental[5]; más adelante veremos las distintas formas en que OpenGL permite organizarlos. Una escena básica como un simple cuadrilátero 2D se expresa de la siguiente manera[6]:

[5] Cualquier polígono puede descomponerse en triángulos. Incluso los puntos y líneas se representan mediante triángulos en la GPU.

[6] Normalmente, los modelos 3D no se escriben en el código, sino que se crean en programas como Blender. Existen numerosos formatos para modelos 3D y escenas, tema que abordaremos más adelante al trabajar con modelos tridimensionales.

```
vertices = np.array(
    [
        -1, -1, # esquina inf izq
         1, -1, # esquina inf der
         1,  1, # esquina sup der
        -1,  1, # esquina sup izq
    ], dtype=np.float32, # ¿por qué se especifica el tipo de número?
)

indices = np.array([0, 1, 2, 2, 3, 0], dtype=np.uint32)
```

En este caso, el cuadrilátero tiene vértices en 2D. Sus coordenadas están expresadas en números de punto flotante (`np.float32`). Esto es posible porque `numpy` (`np`) está implementada en C, un lenguaje con tipos. Además de sus vértices, se definen los índices que configuran los dos triángulos que conforman el cuadrilátero: primero los vértices `(0, 1, 2)` y luego `(2, 3, 0)`. Ambas tuplas de vértices aparecen concatenadas como una sola en la variable `indices`. Como los índices son valores enteros, se especifica que son de tipo `np.uint32` (`unsigned int`).

Dentro de la GPU ocurren diversos **procesamientos geométricos** programables. Esta fase realiza cálculos geométricos a nivel de vértice. La mayoría de estas operaciones son programables mediante un *vertex program* (que procesa vértices) o un *geometry program* (que crea, modifica o elimina vértices). Esta etapa comprende varias sub-etapas estándar: transformación de las posiciones de los vértices a un sistema de coordenadas estandarizado para visualización, cálculo de atributos de vértices (color, normal de superficie, etc.), proyección de vértices sobre el volumen definido por el campo visual de la cámara (conocido como *volumen de vista* o *volumen normalizado*, un cubo cuyos vértices están en $(-1, -1, -1)$ y $(1, 1, 1)$), y filtrado de elementos fuera del volumen de visión. Esta última operación, denominada *clipping* (recorte), optimiza recursos al procesar solo lo visible por la cámara, ya que la conversión de la escena a píxeles es computacionalmente intensiva. Para terminar, el proceso de *screen mapping* convierte las coordenadas de los vértices desde el volumen normalizado hasta las coordenadas específicas de la pantalla o ventana.

La siguiente etapa es la **rasterización** (*rastering*), que convierte los triángulos dentro del volumen de visión en píxeles (Fig. 10). Este método opera mediante procesadores específicos en la GPU y no es programable ni configurable en términos generales.

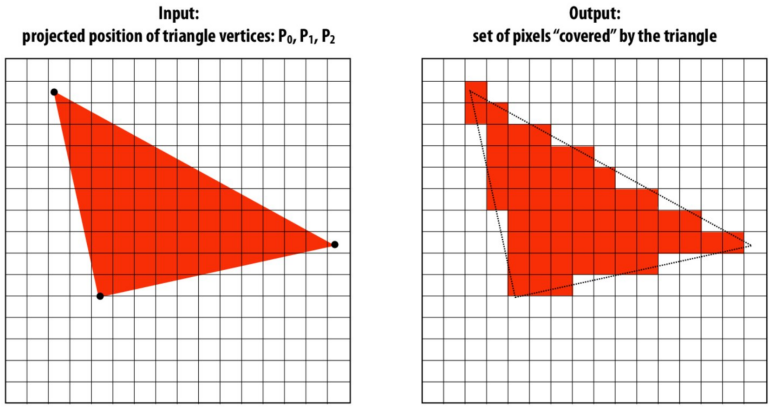


Figura 10. Ejemplo de rasterización de un triángulo. Fuente: CMU. Si quieres ver una demostración interactiva, puedes entrar a este sitio.

Tras la rasterización, disponemos de un conjunto de píxeles para cada elemento de nuestra escena. Sigue entonces el **procesamiento de píxeles**: determinar el color correspondiente a cada uno, considerando si será visible en la imagen final (no todos lo serán, puesto que un triángulo podría tapar a otro en la escena, total o parcialmente). El color se determina en función de múltiples atributos: el triángulo puede tener un color asignado, pero las condiciones de iluminación también influyen en su apariencia dentro de la escena. Este proceso se programa mediante un *fragment program* (donde *fragment* es equivalente a un píxel) cuya salida se escribe en el *frame buffer*.

Esta descripción ofrece una visión general simplificada. A lo largo del curso exploraremos cada etapa con mayor detalle en unidades específicas, experimentando con su implementación.

Figura 11. Logo de Vulkan, por Khronos Group.

Figura 12. Logo de OpenGL, también por Khronos Group.

3 API Gráficas

¿Cómo implementar *rendering*? Aquí entran en juego las API (*Application Programming Interface*) gráficas. Hoy, la API estándar es Vulkan (Fig. 11), desarrollada por Khronos Group. Vulkan ha tomado el relevo de su predecesora OpenGL (Fig. 12). El panorama incluye también otras alternativas como DirectX (desarrollada por Microsoft) y Metal (creada por Apple).

En el curso trabajaremos con OpenGL, a pesar de que su desarrollo se haya detenido[7] en la versión 4.6 (2017). Para nuestro aprendizaje, OpenGL constituye una excelente herramienta debido a la abundante documentación y recursos disponibles. Aunque OpenGL no siga evolucionando, el *hardware* actual mantiene su compatibilidad[8]. Vulkan ofrece mayor potencia, pero

[7] Puedes consultar su historia en: https://www.khronos.org/opengl/wiki/History_of_OpenGL.

[8] Un comentario que he recibido en las encuestas docentes sugiere que "nos enseñan una biblioteca anticuada que no sirve para el mundo laboral". Esto es un mito, pues OpenGL nos permite aprender los fundamentos esenciales. Además, OpenGL sigue utilizándose y su soporte está garantizado en las GPU por largo tiempo. De hecho, WebGL https://www.khronos.org/webgl/ continúa siendo el estándar para gráficos 3D en navegadores web, y está basado en OpenGL. Muchas aplicaciones de uso intensivo, no necesariamente 3D, requieren *performance* y programabilidad de la GPU. Algunos ejemplos: https://www.awwwards.com/the-rise-of-shaders-filters-and-effects-in-web-projects.html.

también implica mayor dificultad de uso y aprendizaje, requiriendo herramientas adicionales y conocimientos más avanzados de programación que los contemplados en este curso.

El origen de OpenGL se remonta a Silicon Graphics a principios de los años 90, y las siglas GL derivan de *Graphics Library*. Es importante entender que OpenGL no es código abierto en sí misma; lo que se considera abierto es su especificación, es decir, la API. Existen diversas implementaciones de OpenGL. Una implementación de OpenGL se limita a la generación de gráficos, delegando otras funcionalidades (*input*, sonido, simulación física, gestión de ventanas, redes, etc.) a bibliotecas externas como `Qt`, `SDL` o `pyglet`. Esta última será la que utilizaremos en nuestro curso: https://pyglet.org/.

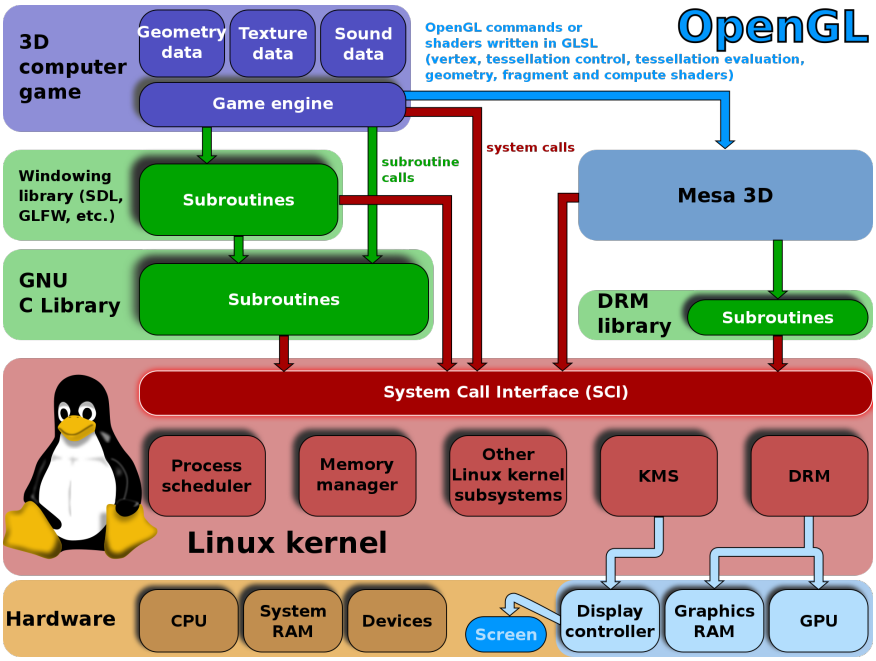


Figura 13. Estructura de un sistema Linux y el rol de Mesa 3D en la generación de gráficos.

Por ejemplo, en sistemas Linux encontramos Mesa 3D https://mesa3d.org/, que proporciona la API de OpenGL a las aplicaciones (Fig. 13). Sin embargo, Mesa no implementa OpenGL; esta tarea recae en el *driver* de la GPU o, en ausencia de ésta, en una versión de *software rendering* donde la CPU asume todo el procesamiento.

OpenGL (como otras API gráficas) opera con vértices e índices, similares a los que especificamos en nuestro ejemplo de escena. A bajo nivel, esta información se almacena en tres estructuras. Para entender su relación, retomemos el cuadrilátero de ejemplo:

- *Vertex Buffer Object* (**VBO**): alberga los datos asociados a cada vértice en un bloque contiguo de memoria. Es solo una

secuencia de números. Para nuestro cuadrilátero, el VBO contendría:

```
[-1, -1,   1, -1,   1, 1,   -1, 1]
  v0         v1       v2       v3
```

- *Element Buffer Object* (**EBO**): contiene los índices que indican cómo agrupar los vértices del VBO en primitivas gráficas (triángulos). Para nuestros dos triángulos:

  ```
  [0, 1, 2,   2, 3, 0]
   tri 1       tri 2
  ```

  El primer triángulo usa los vértices `v0`, `v1` y `v2`; el segundo usa `v2`, `v3` y `v0`. Los índices permiten reutilizar vértices sin duplicar sus datos en el VBO.

- *Vertex Array Object* (**VAO**): le indica a OpenGL cómo interpretar los números del VBO. Sin esta información, el VBO es solo una secuencia de `float` sin estructura. El VAO especifica, por ejemplo: "cada `2` `float` consecutivos corresponden a un atributo `vec2 position`". Si cada vértice tuviera además un color RGB, el VAO indicaría que los primeros `2` `float` son la posición y los siguientes `3` son el color.

Estas operaciones se suelen programar en C o C++. En Python, la biblioteca `pyglet` abstrae esta complejidad mediante mecanismos más *pythónicos*. En este curso no manipularemos directamente `VBO`, `EBO` ni `VAO`, pero sí trabajaremos con vértices e índices, ya que constituyen el fundamento para expresar los objetos de una escena.

Ahora que sabemos que los índices del EBO definen cómo agrupar vértices en triángulos, podemos ver que OpenGL ofrece distintos modos para interpretar esos índices. Como modo principal utilizaremos `GL_TRIANGLES`, que toma cada grupo de tres índices consecutivos como un triángulo independiente (así especificamos nuestro cuadrilátero de ejemplo). Para aplicaciones más avanzadas, existen modos como `GL_TRIANGLE_STRIP`, que reutilizan los dos últimos vértices del triángulo anterior para formar el siguiente, logrando una relación más eficiente entre triángulos y vértices.

4 Programas para la GPU

Una GPU moderna utiliza programas, antes denominados *shaders*, que permiten manipular vértices, geometría y píxeles pa-

ra generar diversos efectos gráficos, desde los más realistas hasta los más estilizados. El término *shader* deriva del verbo "sombrear" que, según la RAE, significa "poner sombra en una pintura o dibujo". Esta denominación hace referencia a la técnica de utilizar líneas o tonos para simular sombras, o más precisamente, para representar la intensidad de la iluminación. Por eso, el "sombreado" constituye el efecto gráfico que refleja la variación realista de luz sobre una superficie (y, consecuentemente, de su color, como se observa en Fig. 14).

Figura 14. Un personaje de videojuego con múltiples efectos programados en *shaders*.

Con la evolución de las capacidades de las GPU, la terminología se ha actualizado: lo que antes llamábamos *vertex shader* ahora se conoce como *vertex program* cuando trabaja con vértices, y el antiguo *pixel shader* se denomina *fragment program* cuando procesa píxeles. Adicionalmente, existe el *geometry program* que permite crear, modificar o descartar geometría en la GPU.

No es obligatorio utilizar *shaders* para renderizar una escena, ya que existe una *rendering pipeline* por defecto, la *fixed pipeline*. Es posible implementar un *vertex program* y dejar el resto a la *fixed pipeline*, pero para trabajar con un *fragment program* siempre resulta necesario tener también un *vertex program* asociado. Por ejemplo, en una escena que muestra el mar durante el atardecer, el *vertex program* se encarga de simular el movimiento de las olas y el *fragment program* de la apariencia del agua y las correspondientes reflexiones de luz (Fig. 15).

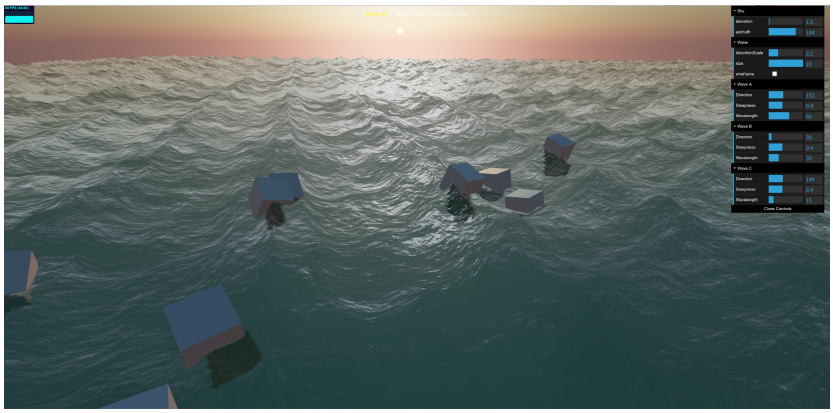


Figura 15. Una escena graficada con *shaders* para simular olas trocoidales https://en.wikipedia.org/wiki/Trochoidal_wave. Ejemplo en WebGL: https://raw.githack.com/Sean-Bradley/three.js/gerstner-waves/examples/webgl_shaders_ocean_gerstner.html.

Estos programas se implementan en un lenguaje denominado GLSL (OpenGL *Shading Language*). GLSL presenta una sintaxis similar a C, y por lo mismo sigue un paradigma imperativo. A diferencia de Python, que utiliza indentación para definir bloques de código, GLSL emplea llaves ({ y }) para delimitar niveles de profundidad y separa instrucciones con punto y coma (;) en lugar de saltos de línea (`\n`).

Este lenguaje se compila en instrucciones para la GPU y no está diseñado como un lenguaje de propósito general. Una característica fundamental de GLSL es que se trata de un **lenguaje fuertemente tipado**. Esto significa que sus variables requieren tipos explícitos (como `vec2` o `mat4`) y no siempre permite flexibilidad en la asignación u operación entre diferentes tipos de datos.

Para ilustrar esta diferencia, comparemos cómo se calcula el producto punto entre dos vectores tridimensionales. En Python:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
np.dot(a, b)
```

En GLSL:

```
vec3 a = vec3(1, 2, 3);
vec3 b = vec3(4, 5, 6);
dot(a, b)
```

¿Cuándo determina Python que `a` y `b` son vectores tridimensionales? No lo hace hasta la ejecución del código. En cambio, en GLSL, los **tipos** se declaran de manera explícita, proporcionando claridad y permitiendo optimizaciones durante la compilación. Es decir, antes de ejecutarse, se garantiza que `a` y `b` son vectores 3D y que se puede realizar la operación `dot` con ellos.

La siguiente tabla resume los tipos de GLSL que utilizaremos a lo largo del curso:

Categoría	Tipo	Descripción	Ejemplo
Escalares	`float`	Número de punto flotante	`float x = 3.14;`
	`int`	Número entero con signo	`int i = 7;`
	`bool`	Valor lógico	`bool visible = true;`
Vectores	`vec2`, `vec3`, `vec4`	Vector de 2, 3 o 4 `float`	`vec3 color = vec3(1.0, 0.0, 0.0);`
	`ivec2`, `ivec3`, `ivec4`	Vector de 2, 3 o 4 `int`	`ivec2 pixel = ivec2(800, 600);`
Matrices	`mat2`, `mat3`, `mat4`	Matriz cuadrada de `float`	`mat4 transform = mat4(1.0);`
Texturas	`sampler2D`	Referencia a una textura 2D	`uniform sampler2D tex;`

Los componentes de un vector se pueden acceder de varias formas. Por posición: `.x`, `.y`, `.z`, `.w` (o `.r`, `.g`, `.b`, `.a` cuando representan colores). También se puede hacer *swizzling*, que consiste en reordenar o seleccionar componentes: por ejemplo, `color.rgb`, `position.xy` o incluso `vec.yx`. Las matrices se acceden por columna e índice: `mat[0]` entrega la primera columna como vector, y `mat[1][2]` entrega un `float` específico.

Además de estos tipos básicos, GLSL soporta *arrays* de tamaño fijo. A diferencia de Python, el tamaño debe conocerse en tiempo de compilación: `float valores[4];` declara un arreglo de 4 `float`. Los *arrays* se acceden por índice de la misma forma que en Python: `valores[0]`, `valores[1]`, etc.

El sistema de tipos también se aplica a las funciones: toda función debe declarar el tipo que retorna. Por ejemplo, la función `hsv_to_rgb` que veremos más adelante declara `vec3` como tipo de retorno, indicando que recibe un `vec3` y produce otro `vec3`:

```
vec3 hsv_to_rgb(vec3 hsv_color) {
    // ...
```

```
    return clamp(rgb + m, 0.0, 1.0);
}
```

Si una función no retorna ningún valor (como `main`), su tipo de retorno es `void`.

GLSL incluye una biblioteca de funciones matemáticas incorporadas, optimizadas para ejecutarse en la GPU. Las más comunes se resumen en la siguiente tabla:

Función	Descripción	Ejemplo
`sin(x)`, `cos(x)`, `tan(x)`	Funciones trigonométricas (en radianes)	`float y = sin(time);`
`atan(y, x)`	Arcotangente de dos argumentos	`float angle = atan(dy, dx);`
`abs(x)`	Valor absoluto	`float d = abs(a - b);`
`mod(x, y)`	Resto de la división `x / y`	`float r = mod(h, 360.0);`
`min(x, y)`, `max(x, y)`	Mínimo y máximo entre dos valores	`float tope = min(x, 1.0);`
`clamp(x, lo, hi)`	Restringe `x` al rango `[lo, hi]`	`vec3 c = clamp(rgb, 0.0, 1.0);`
`mix(a, b, t)`	Interpolación lineal: `a*(1-t) + b*t`	`vec3 c = mix(rojo, azul, 0.5);`
`step(edge, x)`	Retorna `0.0` si `x < edge`, `1.0` si no	`float mask = step(0.5, uv.x);`
`smoothstep(a, b, x)`	Interpolación suave entre `a` y `b`	`float f = smoothstep(0.0, 1.0, x);`
`length(v)`	Largo (norma) de un vector	`float d = length(to_center);`
`normalize(v)`	Vector unitario en la misma dirección	`vec3 dir = normalize(v);`
`dot(a, b)`	Producto punto entre dos vectores	`float d = dot(normal, luz);`
`cross(a, b)`	Producto cruz (solo `vec3`)	`vec3 n = cross(u, v);`

Estas funciones operan tanto sobre escalares como sobre vectores (excepto `cross`, que requiere `vec3`). Cuando reciben un vector, aplican la operación componente a componente: `sin(vec3(1.0, 2.0, 3.0))` retorna un `vec3` con el seno de cada componente.

5 Ejemplo de *vertex* y *fragment program*.

Analizaremos un ejemplo mediante el comando: `python caja_de_juguetes.py color_wheel`.

Vertex program (VP)

Un VP se ejecuta en la fase inicial de la *pipeline*, calculando la posición y atributos de cada vértice, como color, coordenadas de textura, vector normal, entre otros. El siguiente es un programa básico escrito para procesar el cuadrilátero de ejemplo:

```
#version 330

in vec2 position;

void main() {
    gl_Position = vec4(position, 0.0, 1.0f);
}
```

En este ejemplo, existe un único parámetro llamado `position` de tipo `vec2` (vector bidimensional), especificado como entrada (`in`) para cada vértice.

La variable global `gl_Position` (de allí su nombre con prefijo `gl_`) representa la posición final del vértice en coordenadas homogéneas (4D): `vec4`. Por ahora, asumiremos que si no hay una componente z, su valor es 0; y si no hay una cuarta componente, su valor es 1.0. Este programa transmite la posición original del vértice al siguiente programa. De cierto modo, es un programa *identidad*. En unidades posteriores exploraremos distintas funcionalidades posibles.

Fragment program (FP)

Este programa se ejecuta para cada píxel de cada objeto en la escena, después de la fase de rasterización. Recibe como entrada los atributos de los vértices, interpolados en el píxel mediante coordenadas baricéntricas (concepto que podemos observar en el ejemplo `hello_world` del repositorio). El resultado puede ser el color definitivo del píxel o incluso la decisión de descartarlo.

Un FP define su variable de salida mediante la palabra calificadora `out`. En nuestro ejemplo, declaramos `out vec3 outColor`, indicando que el resultado será un vector tridimensional en formato RGB (si fuese RGBA, declararíamos `vec4`). También definimos variables calificadas como `uniform`, que son parámetros de

entrada establecidos desde la aplicación y que mantienen el mismo valor para todas las instancias del programa en ejecución. Aquí utilizamos dos: `time`, de tipo `float`, que registra el tiempo transcurrido desde el inicio de la ejecución, y `resolution`, que informa al FP sobre la resolución de la ventana donde se renderiza la escena, información que no está disponible por omisión.

GLSL permite definir funciones dentro de un *shader*. En este caso, implementamos la función `hsv_to_rgb` que aplica la fórmula de conversión del espacio de color HSV a RGB que estudiamos en la unidad anterior. La firma de la función especifica tanto el tipo de dato de entrada (`vec3 hsv_color`) como el que calcula (también un `vec3`, en este caso representando un color RGB):

```
#version 330
#define TWO_PI 6.28318530718

uniform float time;
uniform vec2 resolution;

out vec3 outColor;

vec3 hsv_to_rgb(vec3 hsv_color) {
    float h = mod(hsv_color.x, 360.0);
    float s = hsv_color.y;
    float v = hsv_color.z;

    float c = v * s;
    float h_prima = h / 60.0;
    float x = c * (1.0 - abs(mod(h_prima, 2.0) - 1));
    float m = v - c;

    vec3 rgb;

    if(h_prima < 1.0) {
        rgb = vec3(c, x, 0.0);
    } else if(h_prima ⩽ 2.0) {
        rgb = vec3(x, c, 0.0);
    } else if(h_prima ⩽ 3.0) {
        rgb = vec3(0.0, c, x);
    } else if(h_prima < 4.0) {
        rgb = vec3(0.0, x, c);
    } else if(h_prima < 5.0) {
        rgb = vec3(x, 0.0, c);
    } else {
        rgb = vec3(c, 0.0, x);
    }
```

```
    return clamp(rgb + m, 0.0, 1.0);
}

void main() {
    vec2 st = gl_FragCoord.xy / resolution;
    vec2 to_center = vec2(0.5) - st;
    float angle = atan(to_center.y, to_center.x) / TWO_PI * 360.0;
    float radius = length(to_center) * 2.0;
    float value = abs(sin(time));
    outColor = hsv_to_rgb(vec3(angle, radius, value));
}
```

No todos los programas necesitan definir funciones propias, pero la función `main` es obligatoria en todo *shader*. Esta es la función principal de nuestro ejemplo `color_wheel` (llamado así porque implementa una representación visual de la rueda de color HSV).

¿Qué operaciones realiza este código? Primero, calcula la posición normalizada del píxel (`vec2 st`). La variable global de OpenGL `gl_FragCoord.xy` proporciona la coordenada en la ventana, pero necesitamos la resolución (proporcionada como parámetro `uniform`) para normalizarla. Luego, calculamos la distancia y el ángulo del píxel respecto al centro de la ventana. También utilizamos el tiempo transcurrido desde el inicio de la aplicación para variar el componente *valor* (en el contexto HSV).

El programa genera una rueda de color donde el matiz varía circularmente, la saturación aumenta desde el centro hacia los bordes, y el brillo oscila con el tiempo.

Un VP que no es identidad: `bad_tv`

En el ejemplo `color_wheel`, el VP es una función identidad: recibe la posición y la transmite sin modificarla. Toda la complejidad reside en el FP. Pero el VP puede hacer mucho más que eso. Para ilustrarlo, analizaremos un segundo ejemplo que invierte los roles: el VP realiza el trabajo interesante y el FP es simple. Este ejemplo se ejecuta con `python caja_de_juguetes.py bad_tv`.

El ejemplo está inspirado en los viejos televisores CRT y la señal análoga. Cuando esta selak se perdía, la imagen se distorsionaba con ondas que recorren la pantalla de arriba a abajo (Fig. 16). Ese efecto se puede simular moviendo los vértices de la geometría en el VP.

Figura 16. Distorsión horizontal de señal, mostrando las clásicas barras de color SMPTE. Este tipo de interferencia, común en la era de la televisión análoga y dispositivos CRT, se produce por la pérdida de sincronización horizontal.

Si nuestra geometría fuese un simple cuadrilátero de cuatro vértices, el VP solo podría mover esas cuatro esquinas. El resultado sería un paralelogramo o un trapezoide, pero no una onda. Para que la deformación sinusoidal sea visible, necesitamos una **grilla de vértices**: muchas filas de puntos a lo largo del eje vertical, de modo que cada fila pueda desplazarse horizontalmente de forma independiente. Esto ilustra que el VP opera **por vértice**, y la rasterización interpola entre ellos.

El VP de este ejemplo recibe la posición de cada vértice y la deforma antes de enviarla a la siguiente etapa:

```
#version 330

in vec2 position;

uniform float time;

out vec2 frag_uv;

void main() {
    frag_uv = (position + 1.0) / 2.0;

    float wave = sin(position.y * 10.0 + time * 4.0) * 0.06;
    wave += sin(position.y * 3.5 - time * 1.5) * 0.04;

    wave *= (0.5 + 0.5 * sin(time * 0.8));

    gl_Position = vec4(position.x + wave, position.y, 0.0, 1.0);
}
```

Examinemos las diferencias respecto al VP de `color_wheel`:

- **Variable `uniform`**: al igual que en el FP de `color_wheel`, aquí el VP también recibe un parámetro `uniform float time`. Esto demuestra que los `uniform` no son exclusivos del FP; cualquier programa en la *pipeline* puede recibirlos.
- **Variable de salida `out vec2 frag_uv`**: el VP calcula las coordenadas UV del vértice original (mapeadas del rango $[-1, 1]$ al rango $[0, 1]$) **antes** de aplicar la deformación, y las envía al FP. La palabra `out` indica que esta variable será interpolada durante la rasterización y llegará al FP como entrada.
- **Deformación de `gl_Position`**: en lugar de asignar la posición original, el VP suma un desplazamiento horizontal (`wave`) calculado con funciones sinusoidales. La combinación de dos ondas con frecuencias y velocidades distintas produce un patrón orgánico. Además, la amplitud global varía con el tiempo: a veces la TV "casi funciona" y a veces se distorsiona.

El FP, por su parte, dibuja las clásicas barras de color SMPTE (la "carta de ajuste" que se veía en los televisores cuando no había programación):

```
#version 330

in vec2 frag_uv;

out vec3 outColor;

void main() {
    int bar = int(frag_uv.x * 7.0);

    vec3 color;

    if (bar == 0) color = vec3(1.0, 1.0, 1.0);      // blanco
    else if (bar == 1) color = vec3(1.0, 1.0, 0.0);  // amarillo
    else if (bar == 2) color = vec3(0.0, 1.0, 1.0);  // cian
    else if (bar == 3) color = vec3(0.0, 1.0, 0.0);  // verde
    else if (bar == 4) color = vec3(1.0, 0.0, 1.0);  // magenta
    else if (bar == 5) color = vec3(1.0, 0.0, 0.0);  // rojo
    else color = vec3(0.0, 0.0, 1.0);                // azul

    float scanline = 0.85 + 0.15 * sin(gl_FragCoord.y * 2.0);

    outColor = color * scanline;
}
```

La variable `in vec2 frag_uv` es la contraparte de la variable `out` declarada en el VP: el valor que el VP calculó por vértice lle-

ga aquí interpolado por píxel. El FP usa la componente horizontal (`frag_uv.x`) para determinar en cuál de las 7 barras de color se encuentra cada píxel. Además, aplica un efecto sutil de *scanlines* (líneas horizontales alternas ligeramente más oscuras) para simular la apariencia de un monitor CRT.

En la práctica, la mayoría de las aplicaciones gráficas distribuyen trabajo entre ambos programas. A lo largo del curso iremos combinando operaciones en VP y FP para lograr efectos cada vez más sofisticados.

6 Ejemplo `hello_world`

Veamos cómo se conectan los conceptos anteriores (vértices, índices, VBO, EBO, *pipeline*, VP, FP) en un programa concreto. El ejemplo `hello_world` del repositorio dibuja un cuadrilátero con un degradado de colores. Analizaremos su código paso a paso usando `pyglet`. Para más detalles sobre la estructura general de una aplicación gráfica y el *game loop*, consultar el Apéndice A.

El primer paso es crear una ventana:

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

A continuación, definimos los datos de nuestra escena. Cada vértice tiene una posición 3D y un color RGB; los índices especifican los dos triángulos que forman el cuadrilátero. Estos datos corresponden a lo que OpenGL almacenará internamente en un VBO (posiciones y colores) y un EBO (índices):

```
vertices = np.array(
    [-1, -1, 0.0,  # inf izq
      1, -1, 0.0,  # inf der
      1,  1, 0.0,  # sup der
     -1,  1, 0.0,  # sup izq
    ], dtype=np.float32,
)

vertex_colors = np.array(
    [1.0, 204 / 255.0, 1.0,  # inf izq
     1.0, 204 / 255.0, 1.0,  # inf der
     204 / 255.0, 1.0, 1.0,  # sup der
     204 / 255.0, 1.0, 1.0,  # sup izq
    ], dtype=np.float32,
)

indices = np.array([0, 1, 2, 2, 3, 0], dtype=np.uint32)
```

Cargamos los archivos GLSL y creamos la *pipeline* (VP + FP compilados):

```
with open(Path(os.path.dirname(__file__)) / "vertex_program.glsl") as f:
    vertex_source_code = f.read()

with open(Path(os.path.dirname(__file__)) / "fragment_program.glsl") as f:
    fragment_source_code = f.read()

vert_shader = pyglet.graphics.shader.Shader(vertex_source_code, "vertex")
frag_shader = pyglet.graphics.shader.Shader(fragment_source_code, "fragment")
pipeline = pyglet.graphics.shader.ShaderProgram(vert_shader, frag_shader)
```

Ahora transferimos los datos a la GPU. La llamada `vertex_list_indexed` crea las estructuras VBO, EBO y VAO. Le indicamos que tendremos 4 vértices organizados con `GL_TRIANGLES`, y le entregamos los índices:

```
gpu_data = pipeline.vertex_list_indexed(4, GL.GL_TRIANGLES, indices)
```

Luego enviamos los atributos de cada vértice. Las claves (`position`, `color`) corresponden a los nombres de las variables `in` declaradas en el VP; `pyglet` usa esta correspondencia para configurar el VAO automáticamente[9]:

```
gpu_data.position[:] = vertices
gpu_data.color[:] = vertex_colors
```

[9] El operador `[:]` permite decir "todos los elementos de una lista". `pyglet` lo usa para actualizar los valores internos antes de transferirlos a la GPU.

Finalmente, definimos qué hacer en cada cuadro. La función `on_draw` limpia el *frame buffer*, activa la *pipeline* y ordena a OpenGL que dibuje los triángulos almacenados en `gpu_data`:

```
@win.event
def on_draw():
    GL.glClearColor(0.5, 0.5, 0.5, 1.0)
    win.clear()
    pipeline.use()
    gpu_data.draw(GL.GL_TRIANGLES)
```

Con `pyglet.app.run()` se inicia el bucle principal que invocará `on_draw` en cada cuadro hasta que se cierre la ventana.

7 Ejercicios

Verdadero/Falso

1. (Control 1, Otoño 2025) La evolución de los gráficos por computadora siguió históricamente la secuencia *wireframe* → polígonos rellenos → texturas, reflejando tanto el avance tecnológico como nuestra comprensión de cómo representar visualmente objetos tridimensionales.
2. (Control 1, Otoño 2025) El término GPU fue acuñado por Nintendo para describir el chip SuperFX incluido en algunos cartuchos de Super Nintendo.
3. (Control 1, Otoño 2025) OpenGL utiliza exclusivamente tipos de datos específicos como `vec2`, `vec3` y `mat4` que son incompatibles con los tipos nativos de los lenguajes de programación host como C++ o Python.
4. (Control 1, Otoño 2025) En la arquitectura de una aplicación OpenGL con `pyglet`, el *game loop* se implementa explícitamente mediante un ciclo `while` controlado por quien hizo el programa.
5. (Control 1, Primavera 2025) El término GPU fue acuñado por Silicon Graphics en 1992 para describir sus estaciones de trabajo con aceleración gráfica.
6. (Examen, Primavera 2025) En un *vertex program*, las variables declaradas como `uniform` pueden modificarse individualmente para cada vértice, mientras que las `in` son constantes para todo el *draw call*.