Objetos y tuberías

Eduardo Graells-Garrido

1 de abril de 2026

1 De la definición de un sistema a su visualización

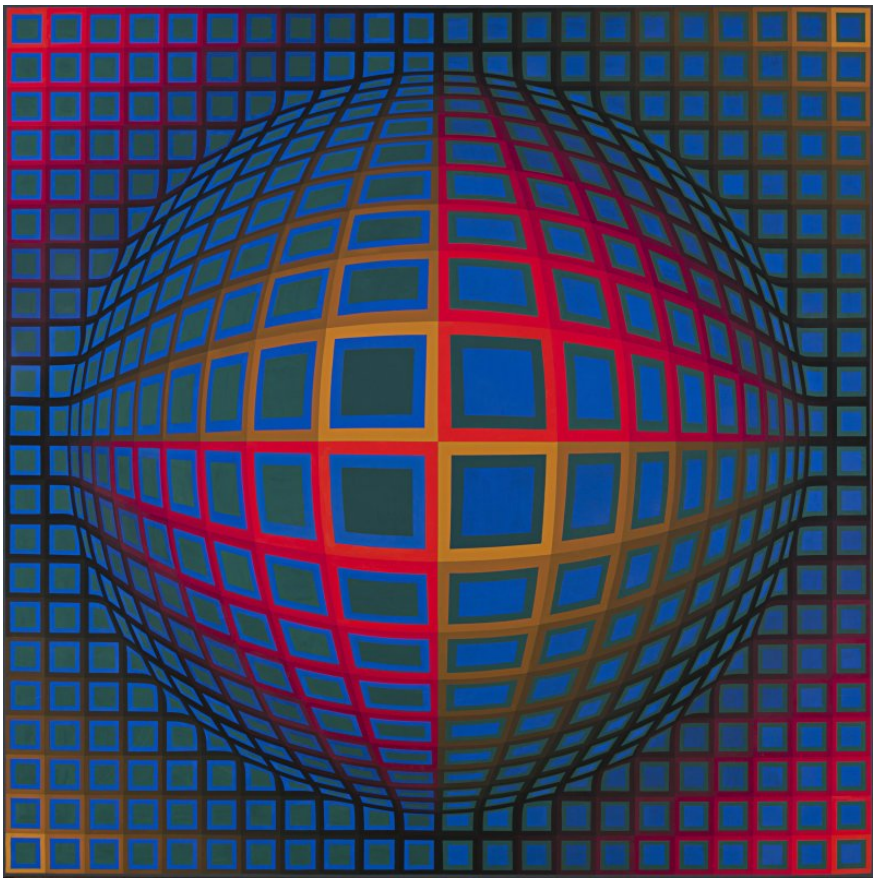

Figura 1. *Vega-nor* (1969), de Victor Vasarely, el padre del *op art* o arte óptico.

En la Fig. 1 vemos patrones geométricos que crean una fuerte ilusión de profundidad y volumen a través de deformaciones sistemáticas de cuadrados y círculos: formas planas pueden crear la ilusión de tridimensionalidad, de modo similar a cómo proyectamos objetos 3D en pantallas 2D. En las unidades anteriores definimos mundos virtuales con agentes y reglas de comportamiento, reglas físicas y funciones de actualización. Con esas herramientas podemos **modelar** sistemas y **simularlos**. El siguiente paso es visualizarlos para que podamos comprenderlos (como en la visualización científica) o incluso participar en ellos (como en los videojuegos) en un entorno 2D o 3D.

Hasta ahora, solo hemos trabajado con escenas básicas: un rectángulo que era convertido en píxeles, un conjunto de puntos (partículas), un conjunto de triángulos (boids). Y nos encargábamos de pintar cada píxel con un *fragment program*. Ahora hablaremos de los modelos 3D que constituyen escenas más complejas: ¿Qué contienen? ¿Cómo se cargan? Y comenzaremos a responder: ¿Cómo se procesan en la *rendering pipeline*?

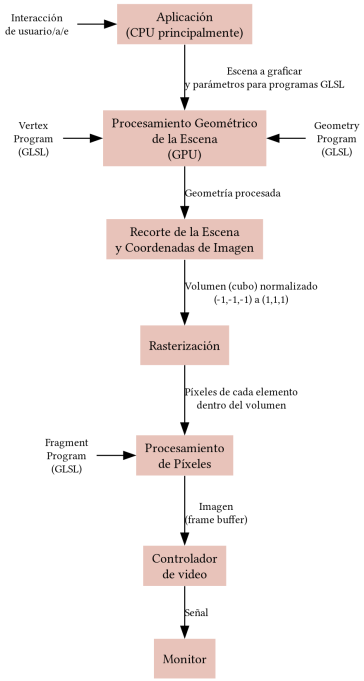

Figura 2. Versión básica de la *Rendering Pipeline*.

Recordemos el diagrama de la *pipeline* que vimos en la unidad de GPU y OpenGL (Fig. 2). Esta *pipeline* es como una línea de ensamblaje con múltiples etapas: la *Aplicación* entrega una escena y las etapas siguientes la procesan hasta obtener una imagen. Los modelos no existen aislados. Se posicionan y orientan en un espacio, como actores en un escenario. Y siguiendo esa metáfora, una cámara virtual los observa desde un punto específico, como si fuera el ojo del espectador. De hecho, la imagen final que vemos es la escena tal como se ve desde la cámara. Pero antes de llegar a esa imagen, debemos procesar los objetos y convertirlos en primitivas gráficas: puntos, líneas y triángulos. Estos elementos son los bloques con los que construimos todo lo que vemos en pantalla. Esta unidad realiza una exploración inicial de algunas de estas etapas.

Objetos y escenas 3D

Un **objeto** 3D contiene geometría y otros datos que describen sus propiedades. Uno clásico es el "Conejo de Stanford" (Stanford Bunny), que representa un conejo de cerámica creado en 1994. Podemos encontrarlo con diferentes resoluciones en la red. La original tiene 69.451 triángulos (Fig. 3).

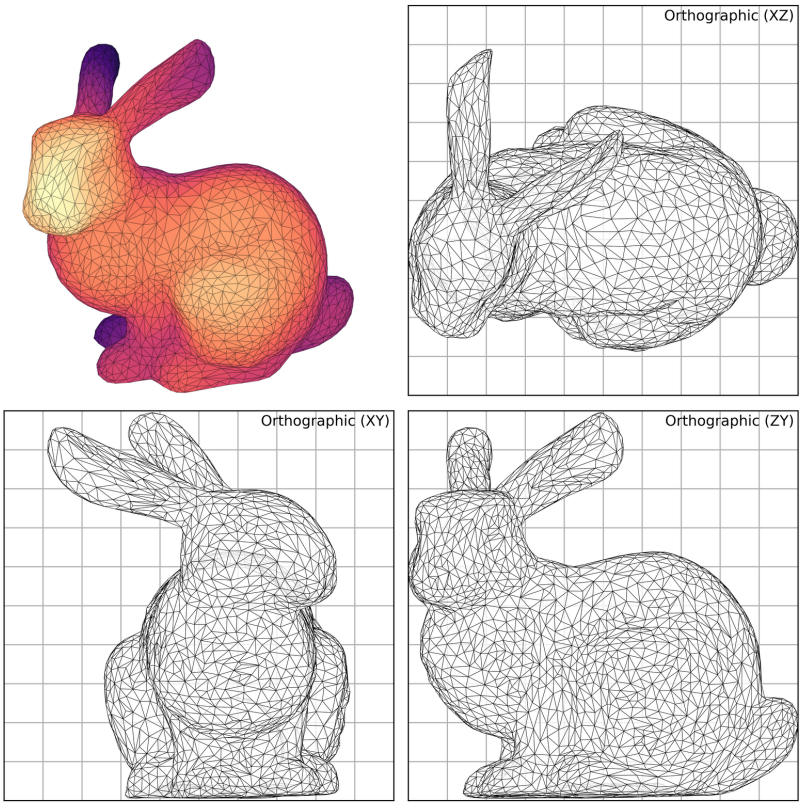

Figura 3. Múltiples vistas del conejo de Stanford. Fuente: Nicolas Rougier.

Las **escenas** son conjuntos de objetos. Por ejemplo, la "Caja de Cornell" es una escena clásica (Fig. 4): contiene varios objetos, una fuente de luz, y los elementos tienen distintas propiedades (colores diferentes, materiales con comportamientos variados frente a la luz). Al igual que el Stanford Bunny, está basada en un conjunto de objetos reales. En este caso, la disposición de los elementos, sus propiedades reflectantes y las características de la luz de la escena son conocidas, por lo que era común ver *renders* de ella en la aplicación de modelos y algoritmos. El *benchmark* correspondía en parecerse a una fotografía de la Caja de Cornell real[1], que fue creada en 1984.

[1]Puedes leer más sobre su historia en este artículo.

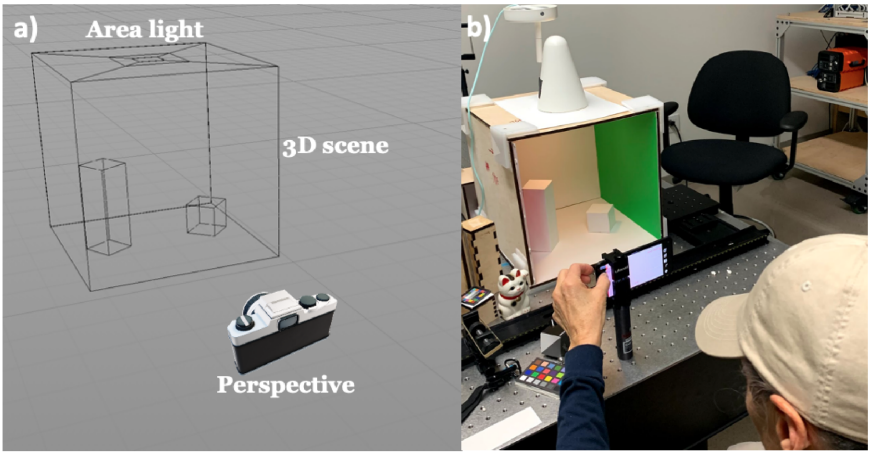

Figura 4. Creación de una Caja de Cornell. Fuente: https://arxiv.org/abs/2105.04106.

Tu *Aplicación* (primera etapa de la *pipeline*) le entrega a la GPU la escena en forma de una lista de primitivas gráficas. En la mayoría de las ocasiones, esta lista está conformada por triángulos. Los triángulos tienen propiedades matemáticas que los hacen ideales:

- **Son planares**: tres puntos definen un plano único[2].
- **Tienen una normal bien definida**: perpendicular al plano que forman.
- **Permiten interpolación**: con coordenadas baricéntricas podemos interpolar en el interior cualquier atributo especificado en sus vértices.
- **Son una primitiva**: cualquier polígono puede descomponerse en triángulos[3]. Incluso cuando dibujamos puntos o líneas, estos se convierten en triángulos[4].

La industria ha invertido décadas en optimizar el procesamiento de triángulos. Las GPU modernas pueden procesar miles de millones por segundo. Es algo que el *hardware* hace por diseño.

[2]A menos que sean colineales, pero entonces no forman un triángulo válido.

[3]Es común que se hagan modelos 3D con cuadriláteros en aplicaciones como *Blender*, pero no se puede garantizar que un cuadrilátero sea planar. A la hora de graficarlo, incluso con la opción `GL_QUADS`, se convierte en dos triángulos.

[4]¿Cómo? Un punto puede ser un triángulo muy pequeño, una línea puede ser dos triángulos que forman un rectángulo delgado.

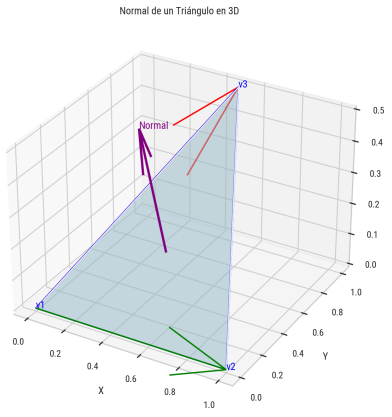

Figura 5. La normal de un triángulo se puede calcular a partir de sus tres vértices.

Atributos: colores, normales y texturas

Los triángulos pueden tener múltiples atributos, como colores en formato `RGBA`, vectores normales y texturas. Las normales son vectores perpendiculares a la superficie (Fig. 5). Aunque cada triángulo tiene una única normal:

$$\vec{n} = (\vec{v_2} - \vec{v_1}) \times (\vec{v_3} - \vec{v_1}),$$

a la GPU se le deben entregar tres: una por cada vértice (`in vec3`). Esto nos permite crear la ilusión de superficies suaves: al interpolar las normales entre vértices, podemos hacer que una esfera construida con triángulos parezca redonda cuando calculemos su iluminación (Fig. 6). En ocasiones, la superficie se define con una función y la malla de triángulos es una discretización. En estos casos, lo esperado es que la normal del vértice sea diferente a la normal del triángulo. En ambas situaciones se usa el mismo esquema para transferir la escena a la GPU. Más adelante usaremos las normales para múltiples propósitos: determinar la visibilidad de los triángulos, calcular la intensidad de la luz en una superficie y definir cómo los objetos interactúan entre sí, entre otras aplicaciones.

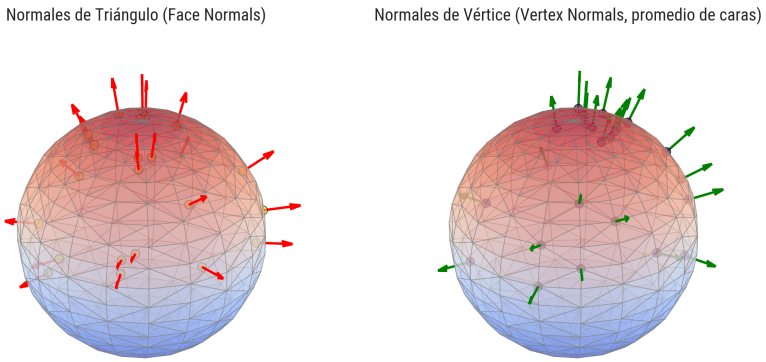

Figura 6. En un modelo 3D podemos calcular sus normales tanto a nivel de triángulos (izquierda) como de vértices (derecha). En ocasiones podemos promediar las normales de las caras, en otras podemos utilizar una fórmula.

También puede haber texturas. En las unidades anteriores vimos que las texturas son imágenes que se almacenan en la memoria de la GPU. Su uso original (y probablemente su mayor uso actual) es otorgar detalles visuales a una superficie sin incrementar su cantidad de polígonos. Imaginemos un personaje de videojuego (Fig. 7): sin texturas, necesitaríamos millones de triángulos para representar detalles como las arrugas de la ropa o los rasgos faciales. Con texturas, podemos usar un modelo simple y "pintarlo" con una imagen que aporte esos detalles (Fig. 8).

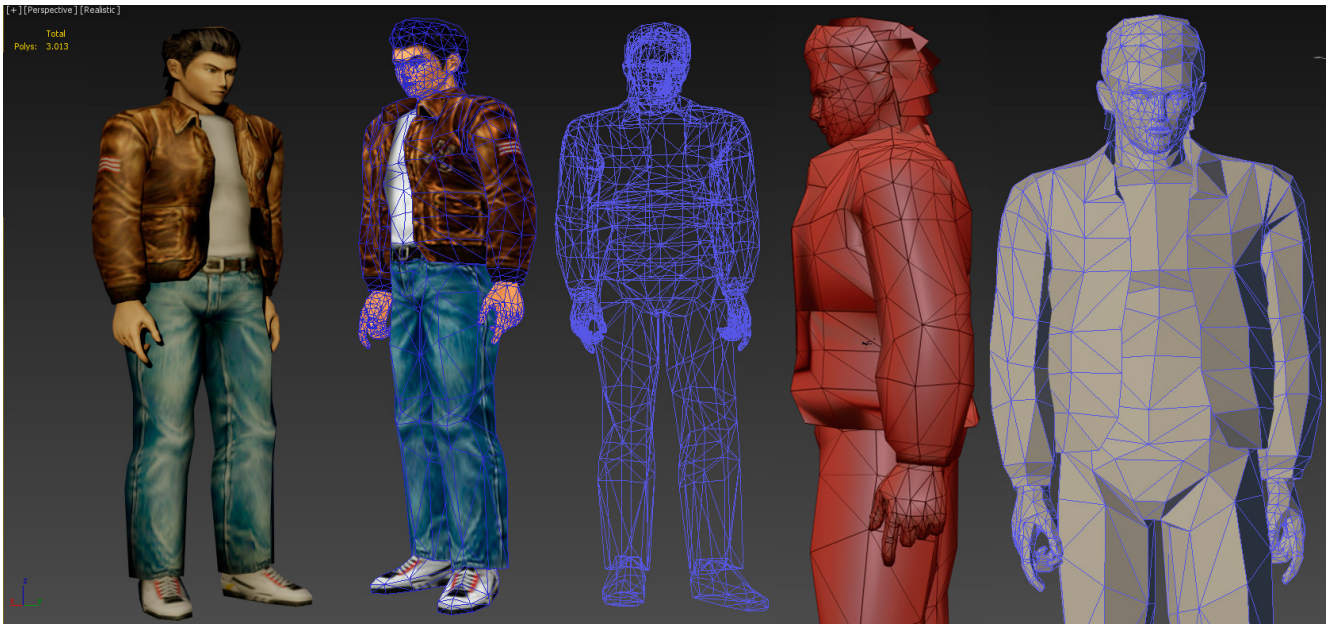

Figura 7. Ryo Hazuki de *Shenmue*. El modelo que se muestra aquí tiene poco más de tres mil polígonos.

Figura 8. Un cuadro del cortometraje *Luxo Jr.* (1986). Los objetos y personajes tienen apariencia realista debido al uso de texturas de madera, metal y goma, además de los efectos de iluminación. Fuente: Pixar.

Para aplicar una textura a un modelo, necesitamos especificar sus **coordenadas de textura** en cada vértice: posiciones 2D (`in vec2`) en la imagen que definen cómo se mapea la textura sobre cada triángulo. Imagina que esas *texcoords* definen un *sticker* con forma de triángulo en la imagen, y lo que hace la GPU es aplicar ese *sticker* a la superficie del triángulo, de modo que sus píxeles finales tengan los colores del *sticker* (Fig. 9).

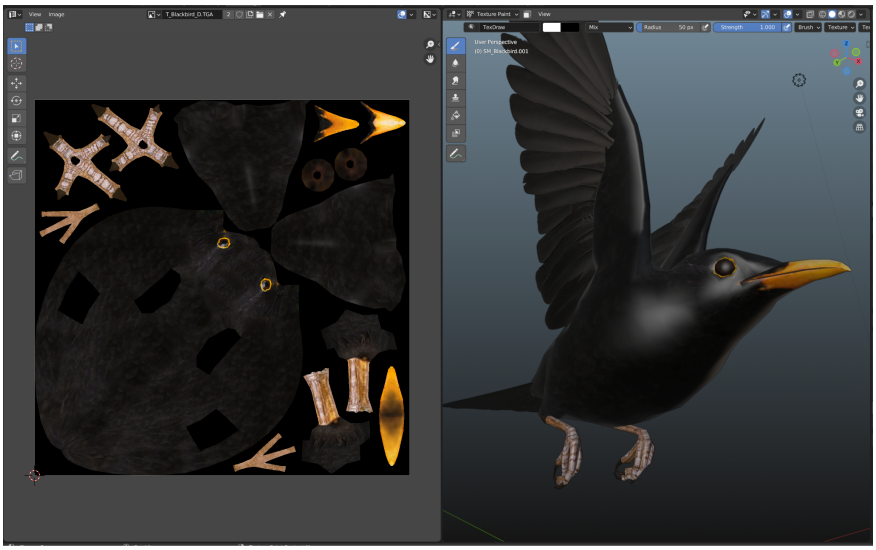

Figura 9. Una textura aplicada a un zorzal negro.

Más adelante tendremos una unidad dedicada a explorar el uso de texturas, tanto para pintar objetos como para crear efectos gráficos avanzados. De hecho, es posible combinar estos dos tipos de atributos en efectos como *normal mapping*, en el cual se almacenan vectores normales en las celdas de una textura[5].

[5]Queda propuesto ver en qué consiste esta técnica.

¿Cómo especificamos un mundo 3D?

Existen diferentes formatos para almacenar modelos 3D. El formato `OBJ` es uno de los más simples y accesibles. Fue definido a mediados de los 80 por Wavefront Technologies. Un archivo `OBJ` es de texto plano y cada línea contiene un elemento cuyo tipo es identificado con los primeros caracteres de la línea:

- coordenadas de vértices (carácter `v`)
- coordenadas de texturas (caracteres `vt`)
- vectores normales (caracteres `vn`)
- triángulos o caras (carácter `f`)
- comentarios (carácter `#`)
- grupos (carácter `g`, se utiliza en caso de tener objetos compuestos por "mini objetos").

Esos elementos pueden referenciar a otros. Por ejemplo, los triángulos se definen con índices de vértices. Los índices se asignan según el orden en el que aparecen en el archivo.

Veamos un ejemplo de archivo para representar un cubo:

```
# cube.obj
# Import into Blender with Y-forward, Z-up
#
# Vertices:                         Faces:
#      f-------g                           +--------+
```

```
#     /.      /|                        /.  5   /|  3 back
#    / .     / |                       / .     / |
#   e-------h  |                   2   +-------+ 1|
#   |  b . .|. c      z          right |  . . .|. +
#   | .     | /       | /y              | . 4   | /
#   |.      |/        |/                |.      |/
#   a-------d         +---- x           +-------+
#                                           6
#                                        bottom

g cube

# Vertices
v 0.0 0.0 0.0  # 1 a
v 0.0 1.0 0.0  # 2 b
v 1.0 1.0 0.0  # 3 c
v 1.0 0.0 0.0  # 4 d
v 0.0 0.0 1.0  # 5 e
v 0.0 1.0 1.0  # 6 f
v 1.0 1.0 1.0  # 7 g
v 1.0 0.0 1.0  # 8 h

# Normal vectors
# One for each face. Shared by all vertices in that face.
vn  1.0  0.0  0.0  # 1 cghd
vn -1.0  0.0  0.0  # 2 aefb
vn  0.0  1.0  0.0  # 3 gcbf
vn  0.0 -1.0  0.0  # 4 dhea
vn  0.0  0.0  1.0  # 5 hgfe
vn  0.0  0.0 -1.0  # 6 cdab

# Faces v/vt/vn
#   3-------2
#   | -     |
#   |   #   |  Each face = 2 triangles (ccw)
#   |     - |            = 1-2-3 + 1-3-4
#   4-------1

# Face 1: cghd = cgh + chd
f 3//1 7//1 8//1
f 3//1 8//1 4//1

# Face 2: aefb = aef + afb
f 1//2 5//2 6//2
f 1//2 6//2 2//2

# Face 3: gcbf = gcb + gbf
```

```
f 7//3 3//3 2//3
f 7//3 2//3 6//3

# Face 4: dhea = dhe + dea
f 4//4 8//4 5//4
f 4//4 5//4 1//4

# Face 5: hgfe = hgf + hfe
f 8//5 7//5 6//5
f 8//5 6//5 5//5

# Face 6: cdab = cda + cab
f 3//6 4//6 1//6
f 3//6 1//6 2//6
```

El formato `OBJ` es uno de muchos formatos para representar modelos 3D. Existen otros formatos con diferentes características. Entre los más básicos se encuentran:

- `STL`: Enfocado en la geometría (mallas de triángulos), sin soporte para texturas, normales o materiales. Es muy usado en impresión 3D por su simplicidad.
- `OFF` (*Object File Format*): Almacena solo la geometría de mallas poligonales: vértices, caras y, de forma opcional, colores.

Otros formatos más avanzados y modernos son:

- `FBX`: formato propietario de Autodesk que soporta animaciones, esqueletos, materiales complejos y luces. Se utiliza en la industria de animación y videojuegos.
- `glTF` (*GL Transmission Format*): es un formato moderno diseñado para web y aplicaciones en tiempo real, con soporte para PBR (*Physically Based Rendering*), animaciones y compresión eficiente.
- `blend` (*Blender*): contiene toda la escena que puede trabajarse en el *software Blender*. Esto incluye modelos 3D, pero también parámetros de escena y otros tipos de elementos.

Cada formato tiene ventajas según el caso de uso: `STL` para impresión 3D, `OBJ` para intercambio básico de modelos con materiales y texturas, `FBX` para animación en motores de juegos y películas y `glTF` para aplicaciones web y en tiempo real.

Todo esto se determina en la etapa de *Aplicación,* donde tú tienes el control y decides cuál formato utilizar, cómo cargarlo, y qué entregarle a la GPU. Puedes explorar un ejemplo práctico con el siguiente comando:

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

Este ejemplo usa la biblioteca `trimesh` para cargar el conejo de Stanford. La biblioteca se encarga de preparar los datos en el formato adecuado para OpenGL, y `pyglet` de gestionar y graficar dichos datos. `trimesh` soporta muchos formatos de modelos 3D, aunque `FBX` y `gLTF` son un desafío todavía debido a su complejidad. La biblioteca también permite convertir entre formatos.

El ejemplo también ilustra dos conceptos de la *pipeline* que veremos más adelante. Primero, el volumen normalizado: el cuadro amarillo marca los bordes del cubo $[-1, 1]^3$ donde OpenGL puede ver. Ese cuadro no coincide con el borde de la ventana porque la escena fue escalada hacia adentro; en la práctica, ese cubo ocupa exactamente la ventana. Segundo, el *clipping* (recorte): cuando el conejo excede los bordes del cubo, el *fragment shader* descarta los fragmentos que quedan fuera, simulando lo que la GPU hace en su etapa de *Clipping*.

2 Procesamiento geométrico

En la etapa de procesamiento geométrico, la GPU procesa las primitivas a nivel de vértice. Por ahora, nos basta saber que esta etapa se divide en varias subetapas:

- **Vertex Shading**: Aplica transformaciones a los vértices, como cambios de sistema de coordenadas (del espacio del modelo 3D al espacio del mundo y luego al espacio de la cámara[6]), y calcula diversos atributos por vértice como color, normal, etc.
- **Projection**: Convierte las coordenadas 3D de los vértices en coordenadas 2D mediante una matriz de proyección que puede ser perspectiva (objetos más lejanos se ven más pequeños) u ortográfica (sin efecto de perspectiva, puedes ver una comparación con perspectiva en Fig. 10), entre otras. Esta transformación mapea el *volumen de vista* (lo que ve la cámara) a un volumen normalizado, un cubo con coordenadas entre -1 y 1 en cada eje, lo que facilita las operaciones posteriores.
- **Clipping**: Opera sobre el volumen normalizado generado en la etapa de proyección. Descarta o recorta las primitivas que están total o parcialmente fuera de este volumen cúbico normalizado, y conserva solo las partes que serán visibles en la imagen final. Este proceso es más eficiente al trabajar con un volumen normalizado, independiente de la configuración original de la cámara.

[6] El nombre "espacio" viene del inglés, donde se dice *object space* y *view space*, pero quizás una mejor traducción es "el sistema de coordenadas del modelo 3D" y el "sistema de coordenadas de la cámara".

- **Screen Mapping**: Transforma las coordenadas de los vértices desde el volumen normalizado a las coordenadas de pantalla (píxeles), según la resolución y dimensiones de la ventana de visualización.

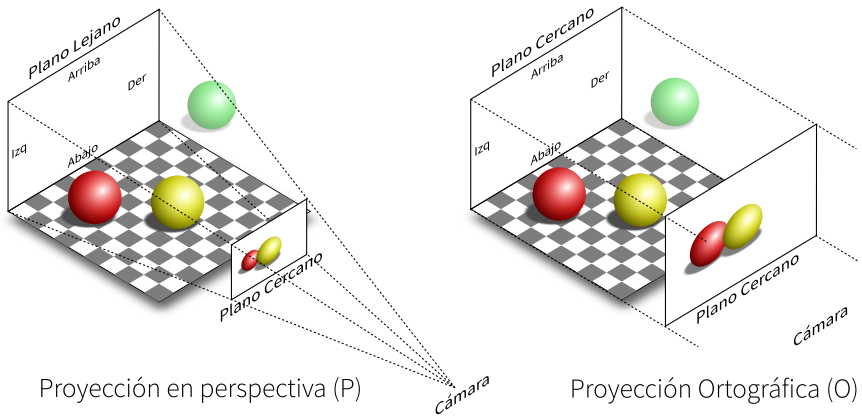

Figura 10. Una escena graficada mediante dos proyecciones diferentes: en perspectiva (izquierda) y ortográfica (derecha). Fuente: Nicolas Rougier.

En esta fase tenemos flexibilidad, pero nuestro control sobre lo que ocurre en la GPU no es absoluto. La arquitectura de la GPU está optimizada para el procesamiento paralelo de datos gráficos y ciertas operaciones están implementadas en *hardware* para maximizar el rendimiento. Por ejemplo, aunque podemos programar cómo se transforman los vértices, no tenemos control sobre cómo la GPU distribuye estos cálculos entre sus múltiples núcleos de procesamiento, ni sobre los detalles de implementación de operaciones como el *clipping* o el *screen mapping*.

3 Rasterización

Después del procesamiento geométrico viene la rasterización (*rastering* en inglés): el proceso de convertir primitivas (triángulos) en píxeles. Es similar a discretizar una función vectorial en una grilla de puntos *raster* que conformarán nuestra imagen final. Esta etapa se realiza sobre triángulos que ya han sido proyectados al espacio 2D de la pantalla; es decir, trabajamos con las primitivas después de que han pasado por todas las transformaciones y la proyección (ver Fig. 11). Respecto a las primitivas que conocemos, la rasterización de un punto es simple: determinar si el píxel cubre el punto. Para una línea, se realiza un barrido a lo largo de esta para determinar qué píxeles intersecta. Algoritmos como el de Bresenham optimizan este proceso.

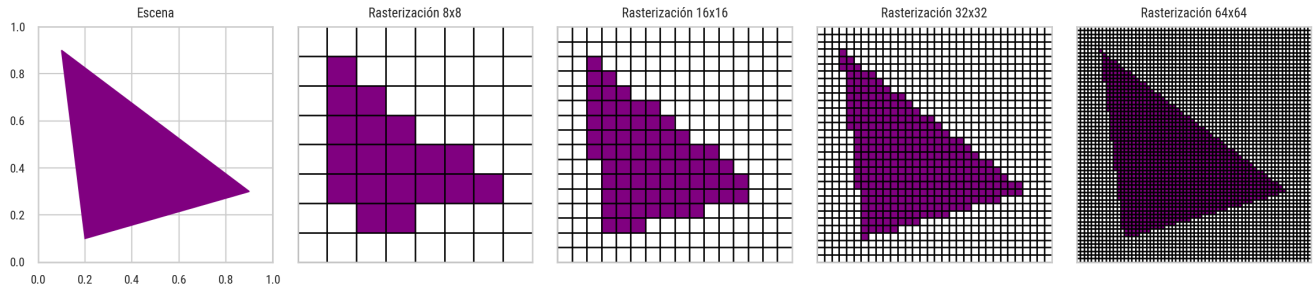

Figura 11. Rasterización de un triángulo a diferentes resoluciones.

Pero, ¿cómo determinamos si un píxel está cubierto por un triángulo? Una estrategia común es considerar el punto central del píxel y verificar si está contenido dentro del triángulo. Sin embargo, surgen casos ambiguos cuando el centro cae sobre una arista. En rigor, si hay dos triángulos, entonces ambos triángulos tienen un punto en el centro del píxel; si hay uno solo que contiene esa arista, podría ser una situación 50/50. Hay dos reglas específicas que permiten dirimir la situación (Fig. 12):

- Arista superior: es una arista horizontal ubicada en la parte superior del triángulo.
- Arista izquierda: no es horizontal y está en el lado izquierdo del triángulo.

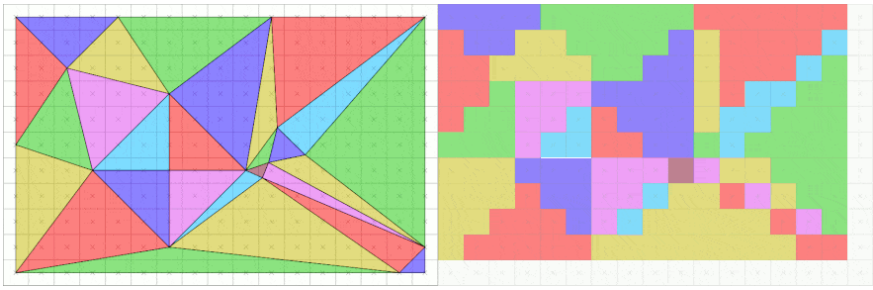

Figura 12. Ejemplo de rasterización de múltiples triángulos. Se aplica la regla de la arista superior. Puedes explorar más detalles en: https://en.wikipedia.org/wiki/Rasterisation.

Una vez que identificamos los píxeles pertenecientes a un triángulo, debemos determinar su color. Recordemos que solo conocemos los atributos de un objeto en sus vértices, pero, ¿cómo los conocemos en su interior? Necesitamos interpolar con **coordenadas baricéntricas**. Este tipo de coordenadas nos permite expresar cualquier punto dentro de un triángulo como una combinación ponderada de sus vértices:

$$\alpha \mathbf{v_1} + \beta \mathbf{v_2} + \gamma \mathbf{v_3} = \mathbf{P},$$
$$\alpha + \beta + \gamma = 1.$$

Para calcular las coordenadas baricéntricas de un punto P (la posición del píxel):

$$\alpha = \frac{\text{Area}(P\mathbf{v_2}\mathbf{v_3})}{\text{Area}(\mathbf{v_1}\mathbf{v_2}\mathbf{v_3})},$$
$$\beta = \frac{\text{Area}(\mathbf{v_1}P\mathbf{v_3})}{\text{Area}(\mathbf{v_1}\mathbf{v_2}\mathbf{v_3})},$$
$$\gamma = \frac{\text{Area}(\mathbf{v_1}\mathbf{v_2}P)}{\text{Area}(\mathbf{v_1}\mathbf{v_2}\mathbf{v_3})}.$$

Se puede expresar la operación como un sistema de ecuaciones:

$$\begin{aligned}\alpha x_1 + \beta x_2 + \gamma x_3 &= x,\\ \alpha y_1 + \beta y_2 + \gamma y_3 &= y,\\ \alpha z_1 + \beta z_2 + \gamma z_3 &= z,\\ \alpha + \beta + \gamma &= 1.\end{aligned}$$

Este sistema tiene cuatro ecuaciones y tres incógnitas. Para un punto en el plano del triángulo, una de las ecuaciones es redundante, por lo que podemos resolver el sistema con cualquier combinación de tres ecuaciones. En la práctica usamos las ecuaciones para x e y junto con la restricción $\alpha + \beta + \gamma = 1$.

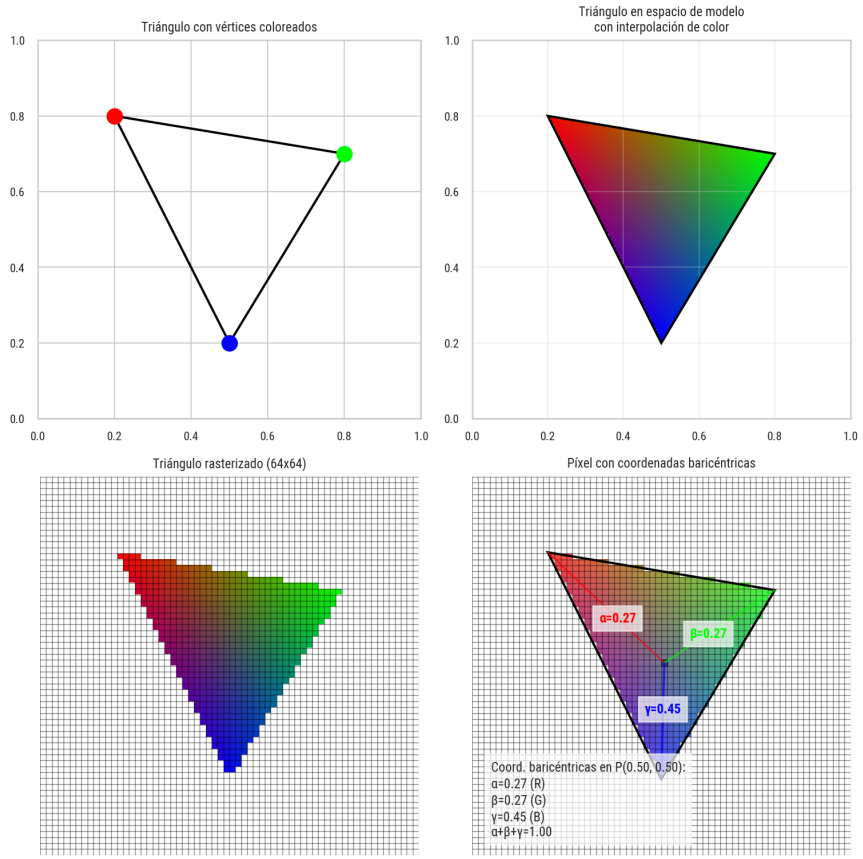

Figura 13. Ejemplo de interpolación de colores en el píxel interno de un triángulo. Se utilizan coordenadas baricéntricas.

Una vez calculados los coeficientes α, β y γ, podemos interpolar cualquier atributo asignado a los vértices: posición, colores, normales, coordenadas de textura, etc. (Fig. 13).

Los colores y otros atributos de los vértices se interpolan de forma automática a lo largo del triángulo mediante coordenadas baricéntricas: los atributos de salida del *vertex program* (aquellos declarados como `out`) llegan interpolados al *fragment program* en la *pipeline* a través de sus variables de entrada (`in`), para ser considerados en la etapa de *Procesamiento de píxeles*.

Ejemplo: rasterizador por *software*

Para comprender la rasterización y la interpolación baricéntrica, implementaremos un rasterizador que replica en la CPU lo que la GPU realiza en hardware. El ejemplo se ejecuta con:

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

La idea del ejemplo es mostrar triángulos rasterizados píxel a píxel, donde el color de cada píxel se calcula con coordenadas baricéntricas. La resolución del raster es baja (40 píxeles de ancho por omisión) para que se aprecien los píxeles individuales. El cálculo de coordenadas baricéntricas traduce las fórmulas de áreas a operaciones algebraicas. Dado un punto (p_x, p_y) y un triángulo con vértices (x_1, y_1), (x_2, y_2), (x_3, y_3):

```
def barycentric(px, py, v1, v2, v3):
    x1, y1 = v1
    x2, y2 = v2
    x3, y3 = v3
    denom = (y2 - y3) * (x1 - x3) + (x3 - x2) * (y1 - y3)
    if abs(denom) < 1e-10:
        return -1, -1, -1
    alpha = ((y2 - y3) * (px - x3) + (x3 - x2) * (py - y3)) / denom
    beta  = ((y3 - y1) * (px - x3) + (x1 - x3) * (py - y3)) / denom
    gamma = 1.0 - alpha - beta
    return alpha, beta, gamma
```

El denominador `denom` es proporcional al doble del área del triángulo (con signo). Si es cercano a cero, los vértices son colineales y no forman un triángulo válido. Los numeradores son proporcionales a las áreas de los subtriángulos formados por el punto P y cada par de vértices, tal como indican las fórmulas. Si alguna de las tres coordenadas es negativa, el punto está fuera del triángulo. Si las tres son positivas (o cero), el punto está dentro (o sobre una arista).

Para rasterizar un triángulo, recorremos los píxeles dentro de su *bounding box* (el rectángulo que lo contiene) y, para cada uno, calculamos las coordenadas baricéntricas del centro del píxel:

```
def rasterize_triangle(buffer, v1, v2, v3, c1, c2, c3, mode="color"):
    h, w = buffer.shape[:2]

    # bounding box del triangulo, restringido al buffer
    x_min = max(0, int(np.floor(min(v1[0], v2[0], v3[0]))))
    x_max = min(w - 1, int(np.ceil(max(v1[0], v2[0], v3[0]))))
    y_min = max(0, int(np.floor(min(v1[1], v2[1], v3[1]))))
    y_max = min(h - 1, int(np.ceil(max(v1[1], v2[1], v3[1]))))

    for py in range(y_min, y_max + 1):
        for px in range(x_min, x_max + 1):
            # centro del pixel
            cx, cy = px + 0.5, py + 0.5
            alpha, beta, gamma = barycentric(cx, cy, v1, v2, v3)

            if alpha < 0 or beta < 0 or gamma < 0:
                continue

            if mode == "barycentric":
                r, g, b = alpha, beta, gamma
            else:
                r = alpha * c1[0] + beta * c2[0] + gamma * c3[0]
                g = alpha * c1[1] + beta * c2[1] + gamma * c3[1]
                b = alpha * c1[2] + beta * c2[2] + gamma * c3[2]

            buffer[py, px] = (
                int(np.clip(r * 255, 0, 255)),
                int(np.clip(g * 255, 0, 255)),
                int(np.clip(b * 255, 0, 255)),
            )
```

Para cada píxel dentro del triángulo, las coordenadas baricéntricas (α, β, γ) se usan como pesos para interpolar los colores de los vértices. El modo `"barycentric"` permite visualizar las coordenadas como colores (α = rojo, β = verde, γ = azul), lo que ayuda a comprender cómo varían estas coordenadas en el interior del triángulo.

El *raster* generado en CPU se sube a la GPU como textura con filtrado `GL_NEAREST` (sin interpolación entre píxeles), de modo que cada píxel del *buffer* se muestra como un bloque cuadrado.

Si se usara `GL_LINEAR`, la GPU suavizaría los bordes entre píxeles y ocultaría la naturaleza discreta del raster.

Dificultades y casos especiales

¿Qué sucede si un píxel está cubierto de forma parcial por un triángulo? ¿Cómo manejamos la superposición de múltiples triángulos en un mismo píxel? En teoría, podríamos calcular el porcentaje exacto de cobertura para cada píxel y asignar su color en proporción a esa cobertura. Pero esto requeriría calcular intersecciones de polígonos para cada píxel asociado a cada primitiva, lo que sería costoso (Fig. 14).

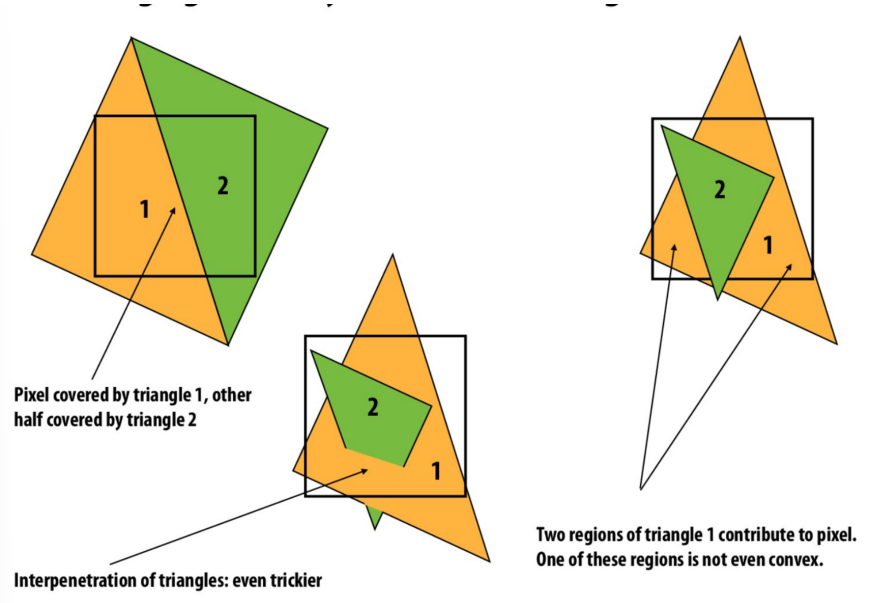

Figura 14. Casos difíciles en la rasterización: oclusión, transparencia y configuraciones complejas. Fuente: CMU.

Además, en escenas complejas es común que múltiples triángulos compitan por el mismo píxel o deban compartirlo. Como los triángulos son procesados en el orden de llegada a la GPU, pueden darse múltiples configuraciones difíciles:

- Oclusión: un píxel que ya fue asignado a un triángulo podría ser asignado a otro. Esto sucede cuando se grafica primero un triángulo que está más lejos de la cámara que otro.
- Transparencia: trabajamos con el modelo de color `RGBA`. Esto quiere decir que hay triángulos que permiten ver a través de ellos.
- Las reglas de selección de triángulo asumen coplanaridad. ¿Pero qué pasa con cortes o intersecciones no convexas?

Estos desafíos se resuelven a nivel de *hardware*. No podemos programar estos aspectos de forma directa, pero entender cómo funcionan permite sacar partido de las capacidades de la GPU. Más adelante profundizaremos en oclusión y transparencia.

Aliasing

El *aliasing* es un fenómeno visual donde se pierde información al convertir una señal de alta frecuencia en una de baja frecuencia. En gráficos rasterizados, se manifiesta como bordes dentados en los triángulos (conocidos como *jaggies*) o como patrones de muaré en texturas (Fig. 15).

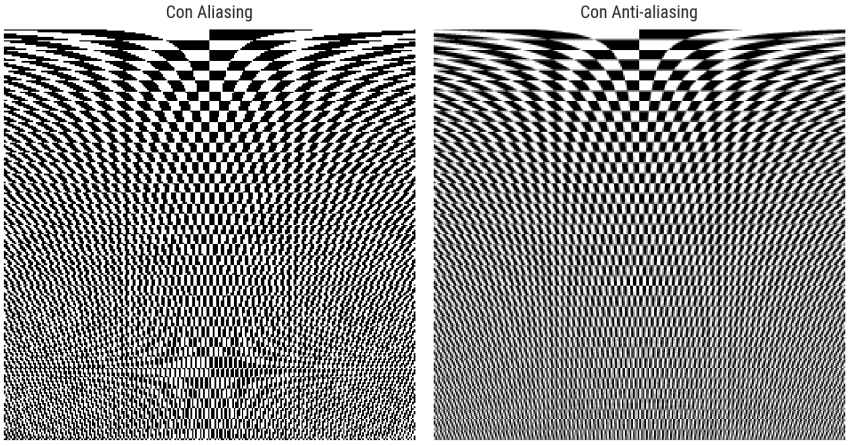

Figura 15. El patrón de la textura se distorsiona por la perspectiva. Aumentar la resolución mueve la distorsión hacia el horizonte, pero no la resuelve. Este tipo de efecto visual se llama *patrón de muaré* https://es.wikipedia.org/wiki/Patr%C3%B3n_de_muar%C3%A9.

Aumentar la resolución puede mitigar el problema pero no lo elimina. La solución radica en el *antialiasing*. Una técnica común es el supermuestreo (*supersampling*) (Fig. 16): en lugar de calcular un solo color por píxel, calculamos múltiples muestras dentro del área del píxel y promediamos los resultados. Realizar este proceso es costoso puesto que se cuadruplica la cantidad de cálculos por píxel.

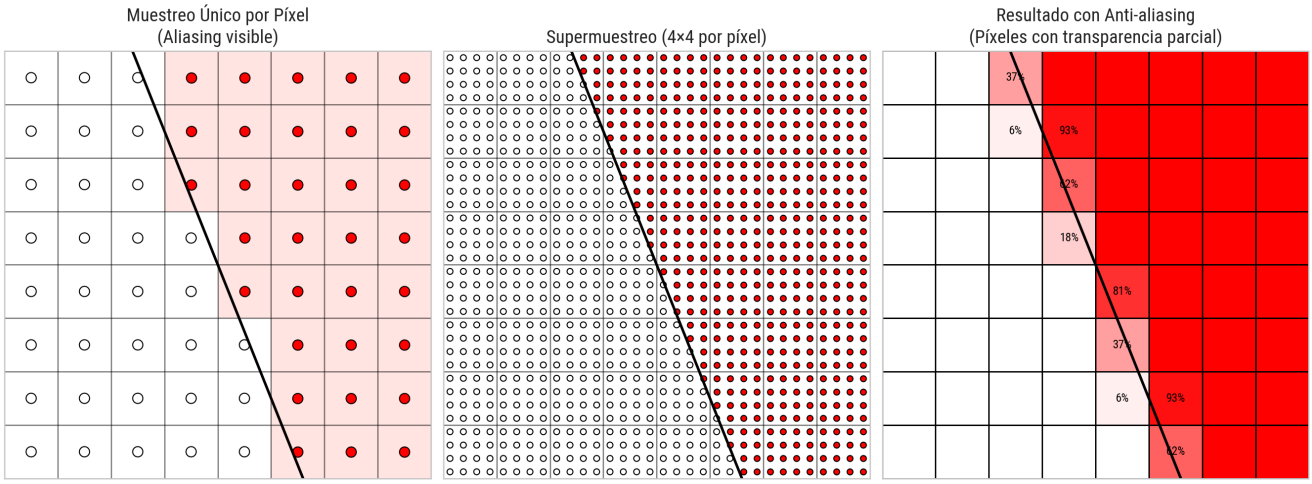

Figura 16. Ejemplo de *antialiasing* utilizando supermuestreo (*supersampling*) de 4×4.

Puedes observar el *aliasing* en el ejemplo `rasterizer`: al mover los vértices, los bordes dentados del triángulo cambian de

forma abrupta. Con resolución baja (por ejemplo, 20 píxeles de ancho), los bordes son burdos. Al subir la resolución, los bordes se suavizan pero el *aliasing* persiste; se necesitaría una resolución infinita para eliminarlo por completo, de ahí la importancia del *antialiasing*.

4 *Pixel Processing*

En la etapa de *procesamiento de píxeles* determinamos el color definitivo de cada píxel en nuestra imagen a partir de la información que tenemos. Para cada píxel identificado durante la rasterización, aplicamos un *fragment program* que calcula su color final basándose en:

- Los atributos interpolados de los vértices (color, normal, coordenadas de textura).
- Información adicional como las coordenadas de textura y la textura correspondiente.
- Propiedades de los materiales y condiciones de iluminación (veremos esto más adelante, en la unidad de modelos de iluminación).

Las técnicas de manipulación de color, filtros, efectos visuales y otros procesamientos de imagen se suelen implementar en *fragment programs*. Un ejemplo básico es el mismo que vimos en la primera unidad:

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

Esta aplicación muestra un rectángulo con colores asignados a cada vértice, interpolados con coordenadas baricéntricas a lo largo de toda la superficie de cada triángulo.

A lo largo del curso exploraremos cada etapa de la *rendering pipeline* en detalle. Las veremos de forma individual y experimentaremos con su implementación práctica. Hoy existen herramientas que implementan por completo esta *pipeline*, como los motores gráficos (por ej., *Unreal Engine*) donde, como desarrolladores(as), nuestra responsabilidad principal es configurar el mundo virtual, definir sus reglas y apariencia. Sin embargo, como ingenieros e ingenieras, nuestra labor es mayor: debemos comprender las bases para así crear los motores del futuro. En este curso las aprenderemos.