Transformaciones en Computación Gráfica

Eduardo Graells-Garrido

13 de abril de 2026

1 ¿Por qué transformaciones?

> Lo que transforma este mundo es el conocimiento. ¿Entiendes lo que quiero decir? Nada más puede cambiar nada en este mundo. Solo el conocimiento es capaz de transformar el mundo, dejándolo exactamente como está. Cuando miras el mundo con conocimiento, te das cuenta de que las cosas son inmutables y, al mismo tiempo, se transforman constantemente.
>
> *El Pabellón de Oro* (1956)
> Yukio Mishima

En computación gráfica podemos transformar un cubo constantemente y, sin embargo, sigue siendo el mismo cubo inmutable en la GPU. El objeto se ve transformado visualmente para nosotros, pero sus datos fundamentales siguen siendo los mismos. Utilizando un solo cubo podríamos incluso replicar una obra de M. C. Escher (Figura 1).

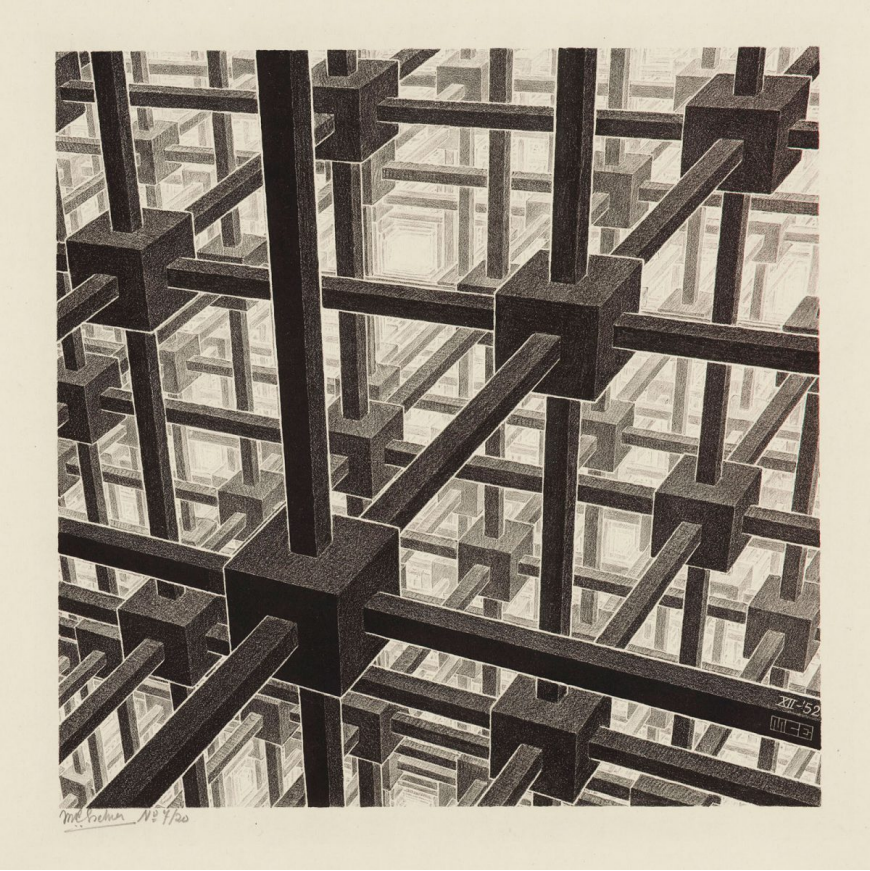

Figura 1. *Cubic Space Division* (1952), M. C. Escher.

En las unidades anteriores, construimos mundos virtuales cuyos elementos variaban sus posiciones, velocidades y orientaciones de manera dinámica. Pero hasta ahora, esos objetos eran recalculados en la CPU y luego copiados a la GPU constantemente con sus atributos actualizados. Por ejemplo, en cada imagen o cuadro de animación (*frame*), un boid se calculaba en la CPU, se creaba un *array* con sus coordenadas nuevas, y luego se entregaba a la GPU.

La manera ideal de hacerlo es copiar el objeto una única vez a la GPU en una posición inicial o neutral, y luego *transformarlo*. Las transformaciones son la herramienta que nos permite, por ejemplo, desplazar y rotar el modelo 3D de una nave espacial en *StarFox* o deformar a Kirby en su interacción con el mundo. En ambos casos, el modelo 3D no cambia, lo que cambia es la transformación que se aplica a él.

Este enfoque es más eficiente: En lugar de transferir miles o millones de vértices actualizados en cada *frame*, enviamos una matriz de 4×4 (16 números de punto flotante) que describe cómo transformar el modelo completo.

2 ¿Qué es una transformación?

Una transformación T es una función que mapea un conjunto de puntos a otro conjunto de puntos:

$$T : \mathbb{R}^n \to \mathbb{R}^m.$$

Las transformaciones que aplicaremos en un principio mantienen la dimensionalidad: $T : \mathbb{R}^2 \to \mathbb{R}^2$ para gráficos 2D o $T : \mathbb{R}^3 \to \mathbb{R}^3$ para mundos tridimensionales.

Las transformaciones nos permiten cambiar entre diferentes "espacios de coordenadas" a medida que procesamos nuestros objetos en la *rendering pipeline* (Figura 2):

- **Espacio del modelo**: Donde se define el objeto, típicamente con centro en el origen.
- **Espacio del mundo**: Donde se posiciona el objeto en la escena completa.
- **Espacio de la cámara**: Como ve la escena un observador virtual o cámara.
- **Espacio de proyección** y **espacio de pantalla**: Para convertir 3D en la imagen 2D final.

Cada transición requiere una transformación específica, como ilustra la Figura 3, y podemos encadenar todas estas operaciones mediante multiplicación de matrices.

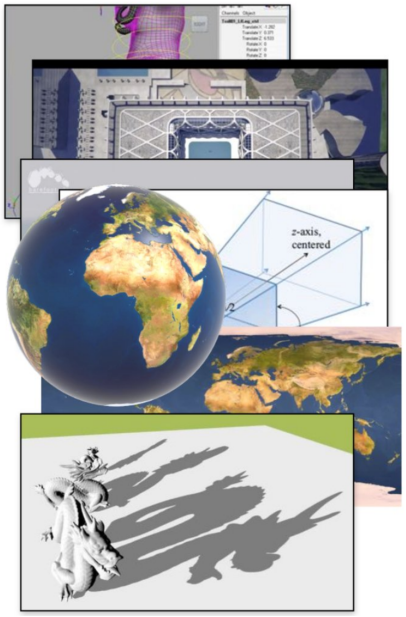

Figura 2. Las transformaciones sirven para muchas cosas: manipular objetos en el espacio, posicionar la cámara, animar objetos, proyectar superficies, entre otras aplicaciones.

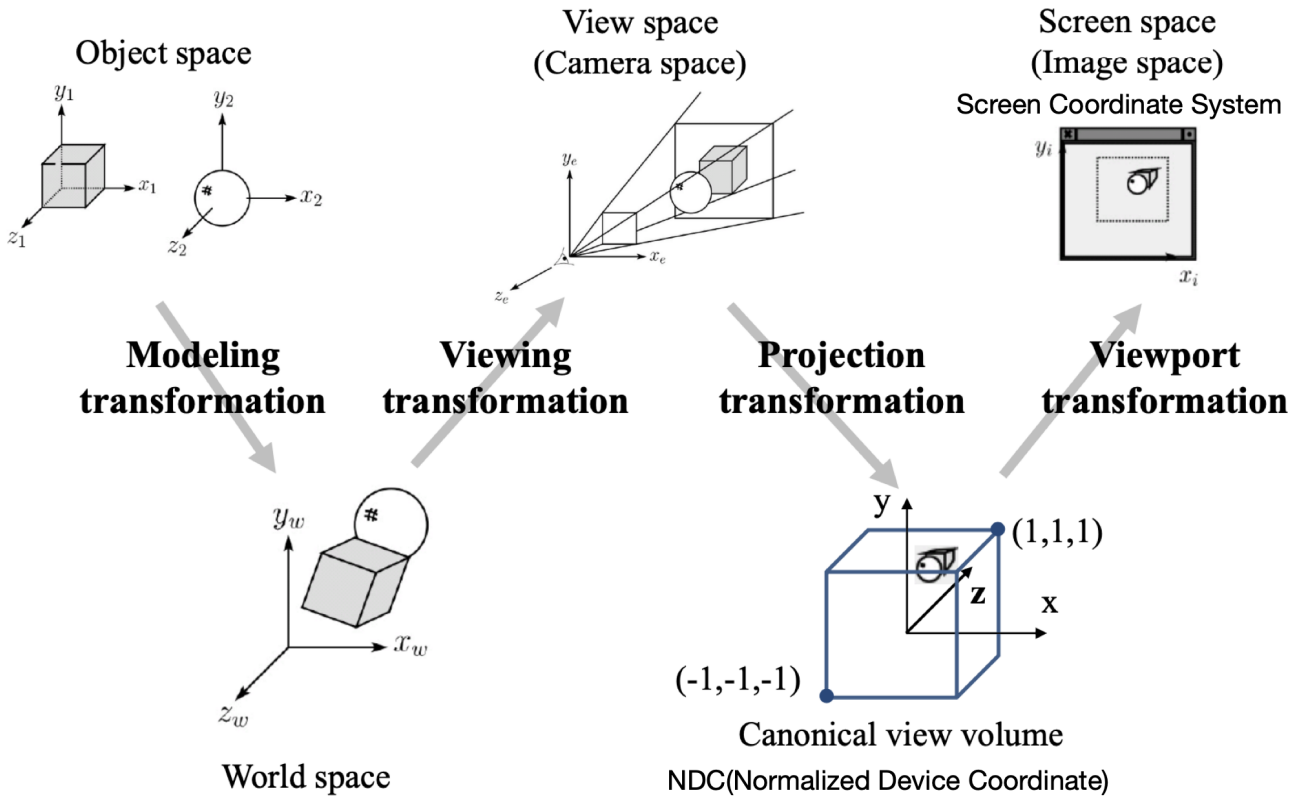

Figura 3. Las transformaciones nos permiten expresar nuestra escena en distintos espacios de coordenadas. Fuente: CMU.

Tipos de transformaciones

Consideraremos dos tipos: lineales y afines. Las **transformaciones lineales** satisfacen dos propiedades:

- Se preserva la suma: $L(u+v) = L(u) + L(v)$
- Se preserva el escalamiento escalar: $L(cu) = cL(u)$

Estas propiedades implican que siempre preservan el origen ($L(\vec{0}) = \vec{0}$), mantienen líneas rectas, preservan paralelismo y conservan proporciones de distancias. Las transformaciones lineales se representan como matrices. En 2D:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Y en 3D:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Esta representación es compacta y permite componer transformaciones mediante multiplicación de matrices.

En cambio, las **transformaciones afines** se definen como:

$$f(x) = Ax + b,$$

donde A es una matriz y b es un vector. No preservan necesariamente el origen, pero sí mantienen líneas rectas y paralelismo.

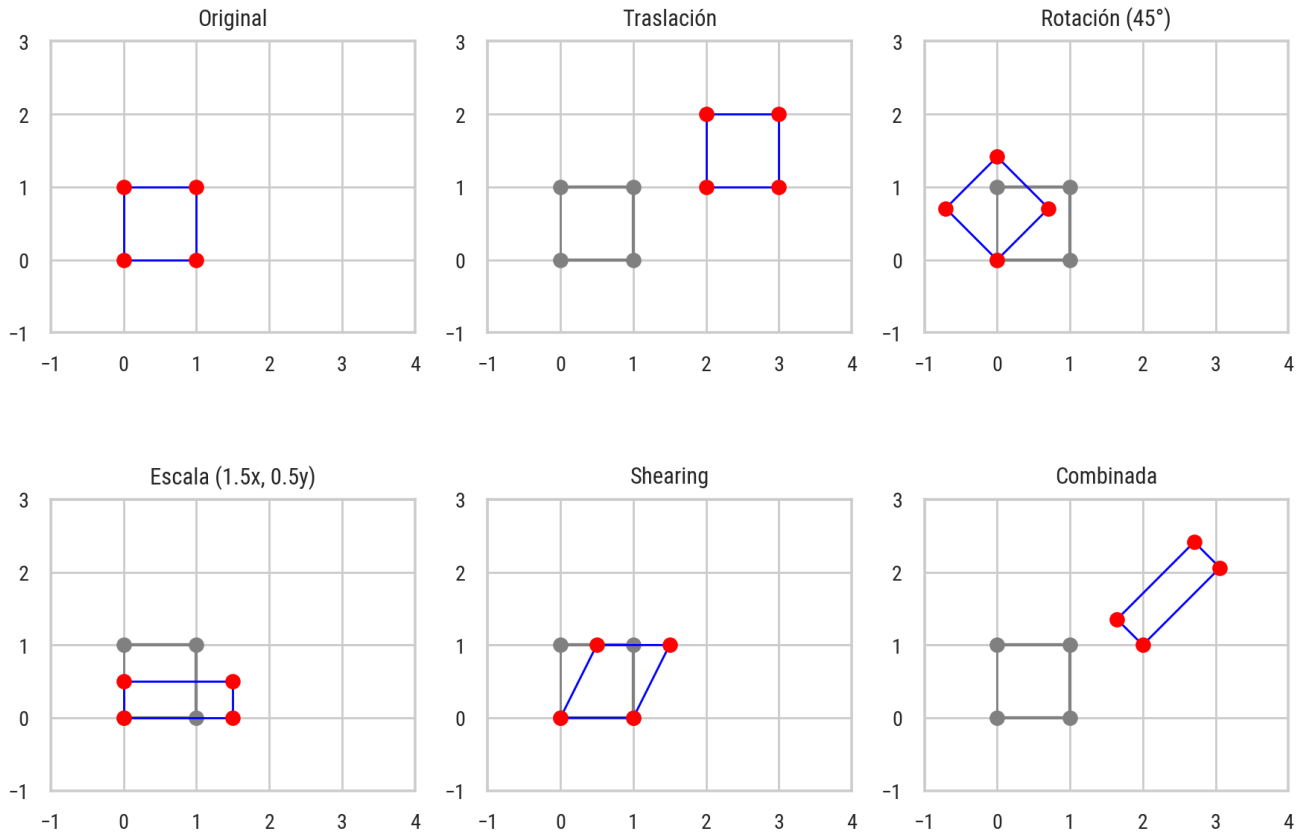

Figura 4. Transformaciones en 2D aplicadas a un cuadrado.

La Figura 4 muestra el efecto de distintas transformaciones 2D sobre un cuadrado. Las exploramos a continuación.

Rotación 2D

La rotación es una transformación ortogonal que preserva distancias y ángulos:

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

La Figura 5 ilustra la geometría de la rotación. Una propiedad importante es que la inversa de una rotación es su transpuesta: $R^{-1} = R^T$. Además, las rotaciones en 2D son conmutativas: $R(\alpha)R(\beta) = R(\alpha + \beta) = R(\beta)R(\alpha)$.

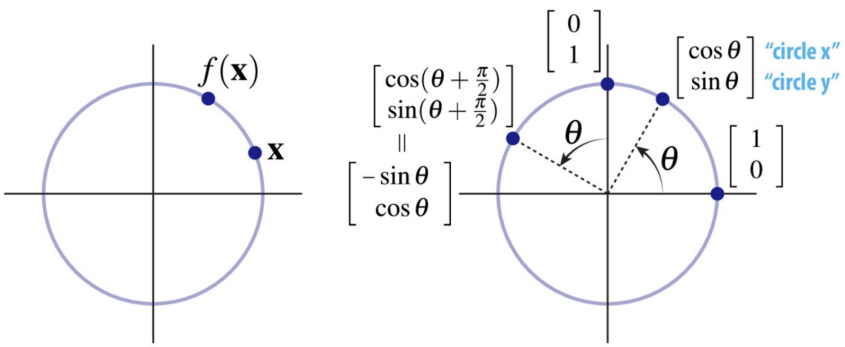

Figura 5. Una rotación convierte un punto $\mathbf{x}$ en un punto $\mathbf{x}'$ que se sitúa dentro de la circunferencia de radio $|\mathbf{x}|$. Fuente: CMU.

Adicionalmente, un punto (x, y) puede representarse como el número complejo $z = x + yi$. Rotar por ángulo θ equivale a multiplicar por $e^{i\theta} = \cos\theta + i\sin\theta$. Esta representación es más compacta y facilita la composición: $e^{i\alpha} \cdot e^{i\beta} = e^{i(\alpha+\beta)}$.

La inversa de una rotación es una rotación en sentido opuesto: $R^{-1} = R^T$ (la inversa es igual a la transpuesta). En 2D:

$$R(\theta)^{-1} = R(-\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

Escalamiento 2D

Esta transformación modifica el tamaño de los objetos:

$$S(s) = \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix},$$

donde s es la escala. También puede definirse de manera no uniforme:

$$S(s_x, s_y) = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}.$$

Shearing 2D (cizallamiento)

Esta transformación desplaza puntos en proporción a su distancia de un eje. Su definición general es:

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}.$$

En forma matricial:

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = (I + \mathbf{u}\mathbf{v}^\top)\mathbf{x}.$$

Casos específicos:

- *Shearing* horizontal (desplaza en X según Y), con $\mathbf{u} = (1, 0)^\top$ y $\mathbf{v} = (0, s)^\top$:

$$H_x(s) = \begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix}$$

- *Shearing* vertical (desplaza en Y según X), con $\mathbf{u} = (0,1)^\top$ y $\mathbf{v} = (s,0)^\top$:

$$H_y(s) = \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix}$$

Reflexión 2D

Esta transformación crea imágenes especulares o reflejadas. Sobre el eje X:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Sobre el eje Y:

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Sobre el origen:

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Traslación 2D

La traslación se define como:

$$T(t_x, t_y) : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x + t_x \\ y + t_y \end{pmatrix}$$

No es una transformación lineal (no preserva el origen):

$$\begin{aligned} T_{\mathbf{x}+\mathbf{y}}(\mathbf{u}) &= \mathbf{x} + \mathbf{y} + \mathbf{u} \\ T_{\mathbf{x}}(\mathbf{u}) + T_{\mathbf{y}}(\mathbf{u}) &= \mathbf{x} + \mathbf{y} + 2\mathbf{u} \\ T_{a\mathbf{x}}(\mathbf{u}) &= a\mathbf{x} + \mathbf{u} \\ aT_{\mathbf{x}}(\mathbf{u}) &= a\mathbf{x} + a\mathbf{u} \end{aligned}$$

Esto significa que no podemos representarla con una matriz 2×2 estándar, y por tanto no podemos combinarla fácilmente con otras transformaciones usando multiplicación de matrices. Es una **transformación afín**: una función de la forma $f(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$, donde A es una matriz y $\mathbf{b}$ es un vector no nulo. En la traslación, $A = I$ y $\mathbf{b} = (t_x, t_y)^\top$. Las transformaciones afines preservan líneas rectas y paralelismo, pero no el origen. Como función, sí tiene inversa: $T(t_x, t_y)^{-1} = T(-t_x, -t_y)$.

3 Transformaciones en 3D

La Figura 6 muestra las transformaciones vistas previamente, esta vez en 3D. La interpretación de cada transformación se mantiene, sin embargo, algunas de sus propiedades son diferentes. Antes de analizar dichas propiedades, describamos los sistemas de coordenadas en los que trabajaremos.

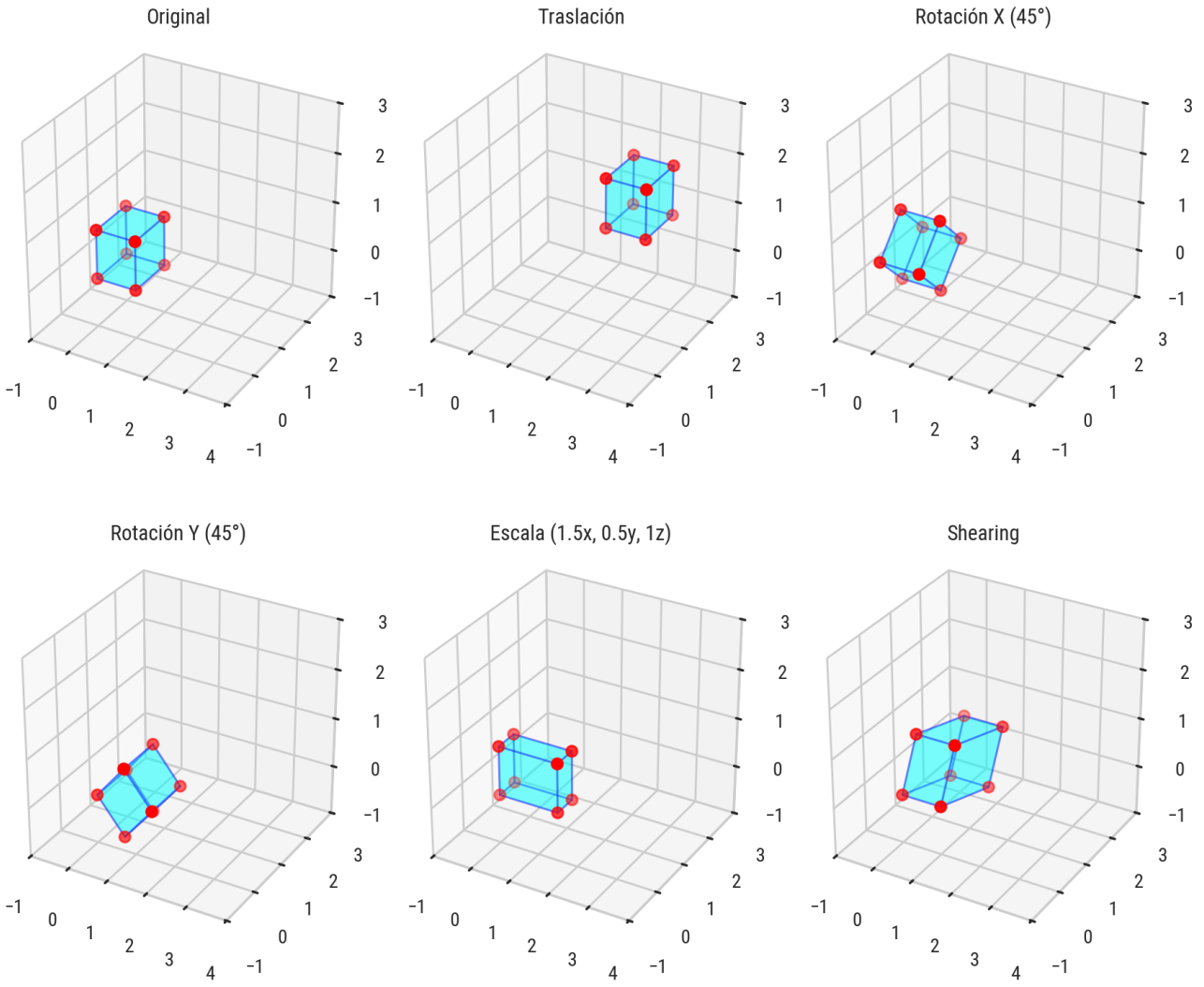

Figura 6. En 3D las transformaciones suelen funcionar de manera análoga a 2D. Las principales diferencias es que se trabajan matrices de 3×3 y vectores de tres dimensiones en el caso base. Además, en ocasiones es necesario especificar un eje de referencia, como en las rotaciones.

Sistemas de coordenadas

OpenGL y Vulkan utilizan la **regla de la mano derecha** (Figura 7): tu pulgar apunta al eje X positivo, tu índice al eje Y positivo, y tu dedo cordial hacia Z positivo. Además, los ejes XYZ se suelen representar con los colores RGB en el mismo orden.

Esta convención además determina la dirección positiva de rotación: los dedos curveados en dirección de rotación y el pulgar apunta en la dirección del eje de rotación. La rotación positiva es en sentido antihorario vista desde el extremo positivo del eje.

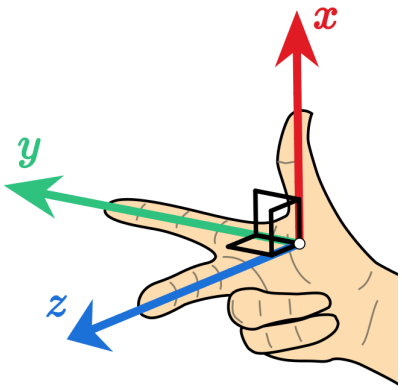

Figura 7. Los ejes cartesianos son ortogonales y estamos acostumbrados a trabajar con X e Y en la pantalla. Entonces, ¿hacia dónde apunta el eje Z? ¿Afuera o adentro de la pantalla? La respuesta es otorgada por la regla de la mano derecha: alínea tu pulgar con el eje X, y sabrás cuáles son los ejes Y y Z.

Rotaciones 3D

En 2D el eje de rotación era implícito. En 3D, necesitamos tres matrices básicas, una alrededor de cada eje:

- Rotación alrededor del eje X: $R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$.
- Rotación alrededor del eje Y: $R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$.
- Rotación alrededor del eje Z: $R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Sin embargo, las rotaciones pocas veces requieren un único eje de rotación. Para rotar un ángulo θ alrededor de un eje unitario $\mathbf{k} = (k_x, k_y, k_z)$, la matriz de rotación es:

$$R(\mathbf{k}, \theta) = \begin{pmatrix} k_x^2(1-\cos\theta) + \cos\theta & k_x k_y(1-\cos\theta) - k_z\sin\theta & k_x k_z(1-\cos\theta) + k_y\sin\theta \\ k_x k_y(1-\cos\theta) + k_z\sin\theta & k_y^2(1-\cos\theta) + \cos\theta & k_y k_z(1-\cos\theta) - k_x\sin\theta \\ k_x k_z(1-\cos\theta) - k_y\sin\theta & k_y k_z(1-\cos\theta) + k_x\sin\theta & k_z^2(1-\cos\theta) + \cos\theta \end{pmatrix}$$

Esta fórmula se conoce como la **fórmula de Rodrigues**[1]. Es útil cuando necesitamos rotar alrededor de un eje arbitrario, independiente de los ejes cartesianos, pero es menos intuitiva de aplicar.

[1] https://es.wikipedia.org/wiki/Rotaciones_en_el_espacio_eucl%C3%ADdeo_4-dimensional#F%C3%B3rmula_de_Euler-Rodrigues_para_rotaciones_3D.

Una alternativa distinta, y más común en la práctica, es descomponer la rotación en tres rotaciones sucesivas alrededor de los ejes cartesianos. Los **ángulos de Euler** formalizan esta idea: representan una rotación como tres rotaciones sucesivas en los ejes X, Y y Z, una para cada uno. Una convención común es ZYX (*yaw-pitch-roll*).

A diferencia de 2D, las rotaciones en 3D no conmutan, por ejemplo, $R_x(\theta)R_y(\phi) \neq R_y(\phi)R_x(\theta)$. ¡El orden importa, porque

una rotación puede cambiar los ejes! De hecho, el problema del **gimbal lock** ocurre cuando dos ejes de rotación se alinean, perdiendo un grado de libertad, como muestra la Figura 8. Este fenómeno es crítico en aplicaciones reales: los simuladores de vuelo del programa Apollo tuvieron que implementar cuaterniones después de que el *gimbal lock* casi causara problemas en la misión Apollo 11. En videojuegos, puede hacer que la cámara "salte" repentinamente cuando el jugador mira hacia arriba o abajo.

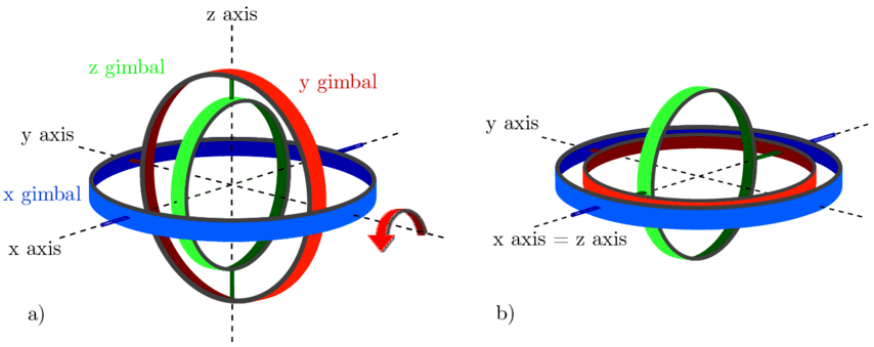

Figura 8. Al usar ángulos de Euler se puede producir el *gimbal lock*, una situación en la que dos ejes de rotación distintos quedan alineados. En a), los ejes son perpendiculares entre sí. Pero como las rotaciones bajo este esquema se aplican de manera independiente, b) después de rotar 90° alrededor del eje y, los otros ejes quedan alineados. Esta situación impide determinar adecuadamente las rotaciones que siguen en los ejes restantes. Fuente: Julian Zeitlhöfler.

Un ejemplo de *gimbal lock* es el siguiente:

1. **Estado inicial**: Los tres ejes (Z, Y, X) son perpendiculares entre sí.
2. **Primera rotación** (*yaw*): Rotamos alrededor del eje Z. Los ejes Y y X cambian de orientación, pero siguen siendo perpendiculares a Z y entre sí.
3. **Segunda rotación** (*pitch*): Rotamos alrededor del nuevo eje Y. Si esta rotación es de ±90°, el eje Z queda **paralelo al eje X**.
4. **Tercera rotación** (*roll*): Ahora rotamos alrededor del eje X, pero como Z y X son paralelos, las rotaciones *yaw* y *roll* producen el mismo efecto.

En esta configuración, perdemos la capacidad de rotar en una dirección específica. Tenemos solo dos grados de libertad en lugar de tres, y ciertos movimientos se vuelven imposibles de representar.

Vale la pena aclarar que el *gimbal lock* no es un bloqueo permanente: alejarse del pitch $\pm 90°$ recupera los tres grados de libertad. Es una singularidad de configuración, no un atasco mecánico. El problema real es que, en esa configuración, dos controles independientes colapsan en un solo efecto y no es posible lograr ciertos movimientos sin primero salir de ella.

Cuaterniones (*quaternions*)

Los cuaterniones resuelven el problema del *gimbal lock*. Así como los números complejos nos permiten representar rotacio-

nes 2D, los cuaterniones hacen lo mismo para rotaciones 3D: con 4 dimensiones manejamos apropiadamente rotaciones 3D (sobre tres ejes imaginarios):

$$q = w + xi + yj + zk,$$

donde $i^2 = j^2 = k^2 = ijk = -1$. Para representar una rotación de ángulo θ alrededor de un eje unitario $\hat{\mathbf{n}} = (n_x, n_y, n_z)$, se usa un cuaternión unitario ($w^2 + x^2 + y^2 + z^2 = 1$):

$$q = \cos(\theta/2) + \sin(\theta/2)(n_x i + n_y j + n_z k).$$

Para rotar un vector v, se calcula $v' = qvq^{-1}$ (donde v se trata como un cuaternión con parte real cero). La conversión a matriz de rotación, en términos de los componentes w, x, y, z del cuaternión, es:

$$R = \begin{pmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) \end{pmatrix}.$$

Se necesitan solo cuatro valores versus los nueve de una matriz para expresar una rotación.

Una ventaja central de los cuaterniones sobre los ángulos de Euler es la interpolación. Para interpolar entre dos orientaciones $\mathbf{q_1}$ y $\mathbf{q_2}$, se utiliza SLERP (*Spherical Linear Interpolation*):

$$\text{SLERP}(\mathbf{q_1}, \mathbf{q_2}, t) = \mathbf{q_1} \frac{\sin((1-t)\omega)}{\sin \omega} + \mathbf{q_2} \frac{\sin(t\omega)}{\sin \omega},$$

donde $\cos \omega = \mathbf{q_1} \cdot \mathbf{q_2}$ y $t \in [0, 1]$. Esta operación es numéricamente estable, algo que no ocurre al interpolar matrices de rotación entre sí.

Para convertir ángulos de Euler (ZYX) a cuaternión:

$$q = q_{\text{yaw}} \cdot q_{\text{pitch}} \cdot q_{\text{roll}},$$

donde cada factor es una rotación alrededor de un solo eje:

$$\begin{aligned} q_{\text{roll}} &= (\cos(\text{roll}/2),\ \sin(\text{roll}/2),\ 0,\ 0) \\ q_{\text{pitch}} &= (\cos(\text{pitch}/2),\ 0,\ \sin(\text{pitch}/2),\ 0) \\ q_{\text{yaw}} &= (\cos(\text{yaw}/2),\ 0,\ 0,\ \sin(\text{yaw}/2)) \end{aligned}$$

De cuaternión $q = (w, x, y, z)$ a ángulos de Euler (ZYX):

$$\begin{aligned} \text{roll} &= \text{atan2}(2(wx + yz),\ 1 - 2(x^2 + y^2)) \\ \text{pitch} &= \text{asin}(2(wy - zx)) \\ \text{yaw} &= \text{atan2}(2(wz + xy),\ 1 - 2(y^2 + z^2)) \end{aligned}$$

La única desventaja es que "nadie entiende cuaterniones"[2]. La entrada es concreta: un eje $\hat{\mathbf{n}}$ y un ángulo θ. La salida también: una rotación sin singularidades, compacta e interpolable. Lo que resulta difícil es la intuición sobre el álgebra interna, pero eso no impide usarlos con confianza.

[2] https://ewpratten.com/blog/quaternions.

Escalamiento

La extensión a 3D es directa desde 2D:

- Escalamiento uniforme: $S(s) = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{pmatrix}$.
- Escalamiento no uniforme: $S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$.

Traslación

La traslación también se extiende de manera directa:

$$T(t_x, t_y, t_z) : \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \end{pmatrix}.$$

Al igual que en 2D, no es una transformación lineal (no preserva el origen), sino afín.

4 Composición de transformaciones

Si tenemos transformaciones lineales representadas por matrices A y B, entonces aplicar B primero y A después equivale al producto $AB \cdot v$. El orden es importante: en general, $AB \neq BA$. Rotar y luego trasladar es diferente de trasladar y luego rotar. Por ejemplo, rotar un objeto alrededor de su propio centro requiere la secuencia $T(p) \cdot R(\theta) \cdot T(-p)$ versus rotar alrededor del origen.

El problema es que la traslación no es lineal y, por tanto, no tiene representación matricial. No puede componerse con otras transformaciones mediante multiplicación de matrices. Supongamos que queremos rotar un punto 2D 45° alrededor del origen y luego trasladarlo en $(3, 2)$.

La rotación se aplica como producto:

$$R(45°) = \begin{pmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{pmatrix}, \quad v' = R(45°) \cdot v.$$

La traslación debe aplicarse como suma por separado:

$$v'' = v' + \begin{pmatrix} 3 \\ 2 \end{pmatrix}.$$

Para el punto $v = (1, 0)^\top$:

1. $v' = R(45°) \cdot (1, 0)^\top = (0.707,\ 0.707)^\top$

2. $v'' = (0.707,\ 0.707)^\top + (3,\ 2)^\top = (3.707,\ 2.707)^\top$

El resultado es correcto. Este mismo procedimiento se puede hacer en 3D. Sin embargo, si encadenamos más transformaciones, la mezcla de productos matriciales y sumas vectoriales se vuelve engorrosa: una secuencia "escalar, rotar, trasladar, rotar de nuevo, trasladar" obliga a intercalar operaciones distintas y no puede reducirse a un único producto. Para resolverlo necesitamos representar traslaciones también como matrices. Ese mecanismo son las coordenadas homogéneas.

5 Coordenadas homogéneas

Las **coordenadas homogéneas**, introducidas por August Ferdinand Möbius (Figura 9) en 1837, resuelven el problema añadiendo una dimensión extra. Un punto 2D (x, y) se representa como una tripleta (x, y, w), y uno 3D (x, y, z) como una cuadrupleta (x, y, z, w). El punto original se recupera dividiendo por w:

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} \longleftrightarrow \begin{pmatrix} x/w \\ y/w \end{pmatrix}, \qquad \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \longleftrightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}, \quad w \neq 0.$$

Para puntos ordinarios se usa $w = 1$, de modo que $(x, y, 1)$ representa al punto (x, y). Cualquier múltiplo $(\alpha x, \alpha y, \alpha w)$ con $\alpha \neq 0$ representa el mismo punto, porque al dividir por αw se recupera el mismo resultado. Cuando $w = 0$, la división no está definida: los vectores de la forma $(x, y, 0)$ representan **direcciones** en el infinito, no posiciones. Por eso en un *vertex program* la variable `gl_Position` es de tipo `vec4`.

Figura 9. August Ferdinand Möbius, creador de las coordenadas baricéntricas y de las coordenadas homogéneas. Fuente: Wikipedia.

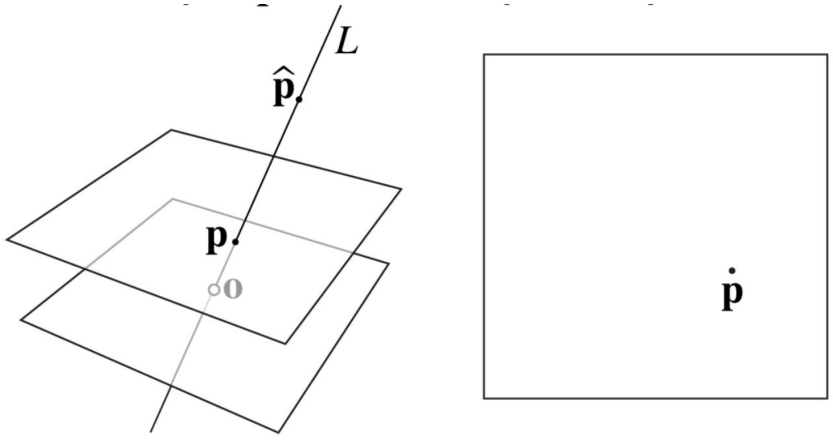

Figura 10. Considera cualquier plano que no contenga al origen. Cualquier línea L que pase a través del origen se intersecta con ese plano. Para representar a la línea podemos usar el punto **p** en que intersecta al plano. Cualquier punto que esté en L puede ser usado para representar a **p**.

El espacio de las coordenadas homogéneas es conocido como **espacio proyectivo** $\mathbb{P}^n(\mathbb{R})$ (Figura 10). Geométricamente, ca-

da punto proyectivo corresponde a una línea que pasa por el origen en $\mathbb{R}^{n+1}$: todos los escalares múltiplos $(\alpha x, \alpha y, \ldots, \alpha w)$ con $\alpha \neq 0$ pertenecen a esa línea y representan el mismo punto.

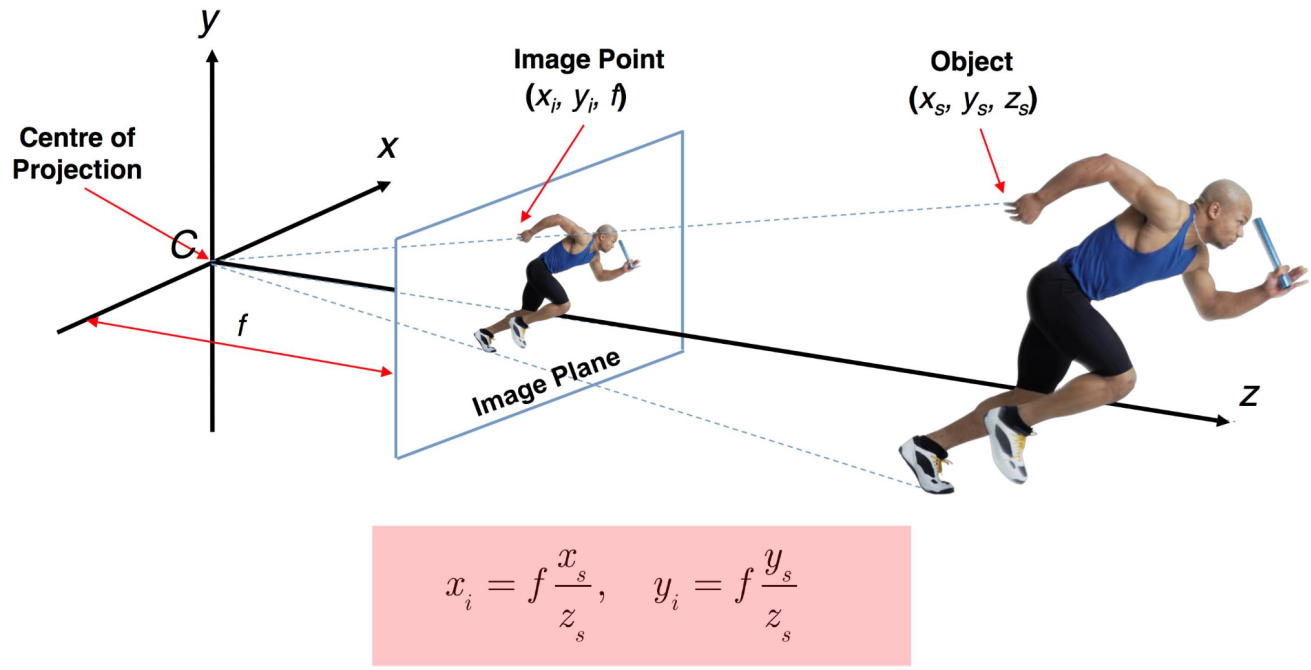

Figura 11. Las coordenadas homogéneas se utilizan en la proyección puesto que nos permiten expresar cualquier punto en la escena como un punto en un plano específico: aquel que utilizamos como imagen. Aprenderemos más de esto en la unidad de proyección.

Una consecuencia de esta geometría es que **dos líneas paralelas se intersectan en un punto del infinito** (Figura 11). En el espacio euclidiano, $y = 2x + 1$ e $y = 2x + 3$ nunca se intersectan porque son paralelas: comparten la misma dirección $(1, 2)$ pero tienen distinto término independiente. En coordenadas homogéneas, esa dirección compartida se representa como $(1, 2, 0)$ (con $w = 0$), que es precisamente el punto en el infinito donde ambas "convergen". Esto tiene una interpretación visual directa: las vías de tren paralelas parecen converger en el horizonte.

Traslación como *shearing*

La traslación no es lineal en $\mathbb{R}^n$, pero en coordenadas homogéneas se convierte en una transformación lineal en $\mathbb{R}^{n+1}$. Consideremos el caso 2D: un punto (p_1, p_2) se representa como $(p_1, p_2, 1)$. Aplicamos un *shearing* en el espacio homogéneo $\mathbb{R}^3$ con $\mathbf{u} = (u_1, u_2, 0)^\top$ y $\mathbf{v} = (0, 0, 1)^\top$ (el *shearing* actúa según la coordenada w):

$$I + \mathbf{u}\mathbf{v}^\top = \begin{pmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{pmatrix}$$

Aplicado al punto $(p_1, p_2, 1)$:

$$\begin{pmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 + u_1 \\ p_2 + u_2 \\ 1 \end{pmatrix}.$$

El resultado es el punto trasladado $(p_1 + u_1,\ p_2 + u_2)$, ya en coordenadas homogéneas con $w = 1$. La traslación en $\mathbb{R}^n$ es, en términos matemáticos, un *shearing* en $\mathbb{R}^{n+1}$. La explicación geométrica: el hiperplano $w = 1$ donde viven los puntos ordinarios se desplaza respecto al hiperplano $w = 0$ donde viven las direcciones.

En 3D, la misma construcción da la matriz de traslación $(\Delta x, \Delta y, \Delta z)$:

$$T(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Con esto, todas las transformaciones se expresan como matrices de 4×4. Una rotación o escalamiento $M_{3\times3}$ se extiende añadiendo una fila y columna de identidad:

$$\begin{pmatrix} M_{3\times3} & \mathbf{0} \\ \mathbf{0}^\top & 1 \end{pmatrix}.$$

Y cualquier composición de rotaciones, escalamientos y traslaciones queda como un único producto de matrices, sin mezclar sumas.

6 Implementación

Un ejemplo típico de transformación en un *vertex program* es el siguiente:

```
#version 330

in vec3 position;
uniform mat4 model;  // Transformación del modelo

void main() {
    gl_Position = model * vec4(position, 1.0);
}
```

Aquí aplicamos la transformación del modelo al objeto. La matriz `mat4 model` puede contener cualquier combinación de traslación, rotación y escalamiento, compuestas en la CPU mediante multiplicación de matrices. Es una variable `uniform` porque

tiene el mismo valor para todos los vértices del modelo. Y ahora podemos ver por qué `gl_Position` es de tipo `vec4`: porque trabaja en coordenadas homogéneas.

En Python, el módulo `grafica.transformations` provee las matrices de transformación como *arrays* de NumPy de tipo `float32` y dimensión 4×4. Sus funciones principales son:

- `tr.rotationX(θ)`, `tr.rotationY(θ)`, `tr.rotationZ(θ)`: rotación alrededor de cada eje.
- `tr.rotationA(θ, axis)`: rotación alrededor de un eje arbitrario (fórmula de Rodrigues).
- `tr.translate(tx, ty, tz)`: traslación.
- `tr.uniformScale(s)`, `tr.scale(sx, sy, sz)`: escalamiento.
- `tr.identity()`: identidad.

Las matrices se componen con el operador `@` de NumPy (o con `tr.matmul([m1, m2, ...])`). El orden es el mismo que en la notación matemática: la transformación más a la derecha se aplica primero. El ejemplo `spirograph` implementa un espirógrafo (Figura 12): un mecanismo en el que un círculo de radio r rueda dentro de un círculo fijo de radio R, con un lápiz a distancia d de su centro.

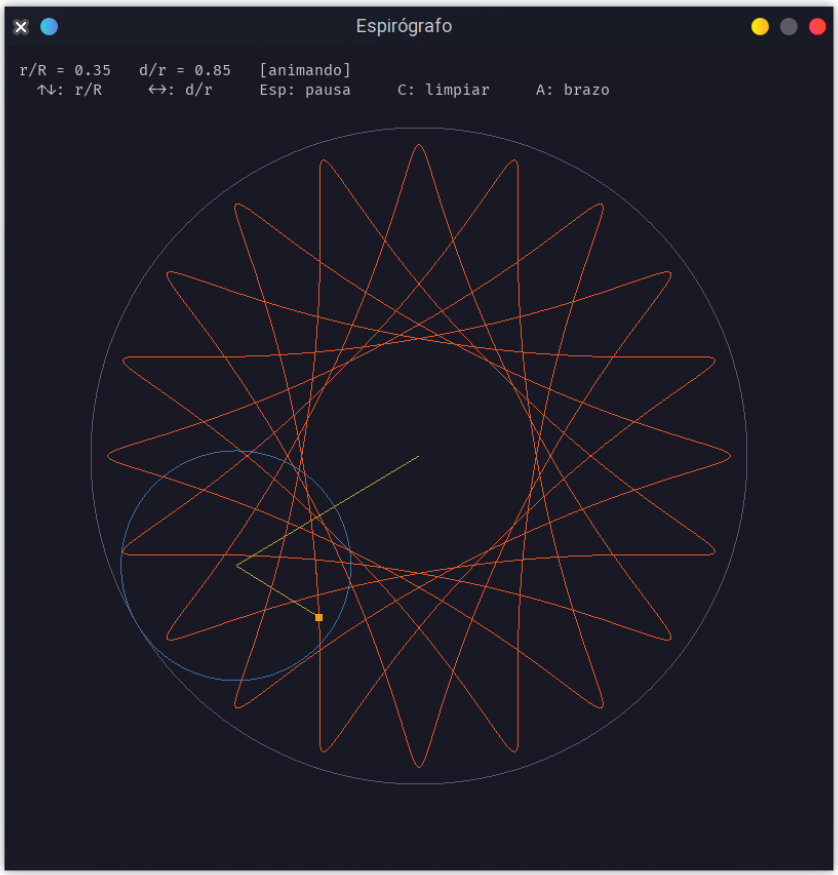

Figura 12. El espirógrafo del ejemplo `spirograph`. Según los valores de r/R y d/r, la curva toma formas distintas: pétalos, estrellas o espirales.

Las cuatro transformaciones que ubican el lápiz en cada instante se componen así:

```
import grafica.transformations as tr

outer_rotation  = tr.rotationZ(angle)
arm_translation = tr.translate(R - r, 0, 0)
inner_rotation  = tr.rotationZ(-angle * R / r)
pen_offset      = tr.translate(d, 0, 0)

pen_transform = outer_rotation @ arm_translation @ inner_rotation @ pen_offset
```

Para enviar la matriz al *vertex program* como `uniform mat4`, hay que pasarla en orden columna-mayor (*column-major*), que es el que usa OpenGL. En pyglet esto se hace con `.reshape(16, 1, order="F")`:

```
pipeline["model"] = pen_transform.reshape(16, 1, order="F")
```

Más adelante en el curso veremos cómo añadir transformaciones adicionales para cámaras y proyección:

```
gl_Position = projection * view * model * vec4(position, 1.0);
```

Por ahora, con las transformaciones básicas estudiadas podemos manipular objetos en la GPU enviando solo matrices de 4×4 (16 números) en lugar de recalcular y transferir geometría completa en cada *frame*.

El ejemplo `transformed_bunny` muestra una animación compuesta de rotación, balanceo y escala. Con la tecla `H` se puede activar una variación del componente w de `gl_Position`: en lugar de $w = 1$ (sin división perspectiva), el shader asigna $w = 1 + k \cdot y$, con lo que las partes altas del modelo se ven más pequeñas que las bajas. Es el mismo principio que usa la proyección perspectiva, que veremos en la siguiente unidad. El ejemplo `compositions` muestra el mismo modelo con cuatro animaciones distintas: órbita circular, trayectoria en figura de ocho, péndulo con aplastamiento proporcional a la velocidad, y doble rotación en dos ejes simultáneos.