Texturas y atractores

Eduardo Graells-Garrido

23 de marzo de 2026

1 De color a datos en la GPU

En esta unidad exploraremos las texturas como estructuras de datos programables en la GPU y su aplicación en la visualización de sistemas dinámicos. Estudiaremos cómo representar y visualizar sistemas que evolucionan con el tiempo según reglas deterministas, centrándonos en la implementación de atractores caóticos.

En la unidad anterior aprendimos que una imagen se expresa en un espacio de color, como RGB (Fig. 1). Se puede almacenar en disco, utilizando algún formato (como BMP, PNG o JPEG) o en memoria.

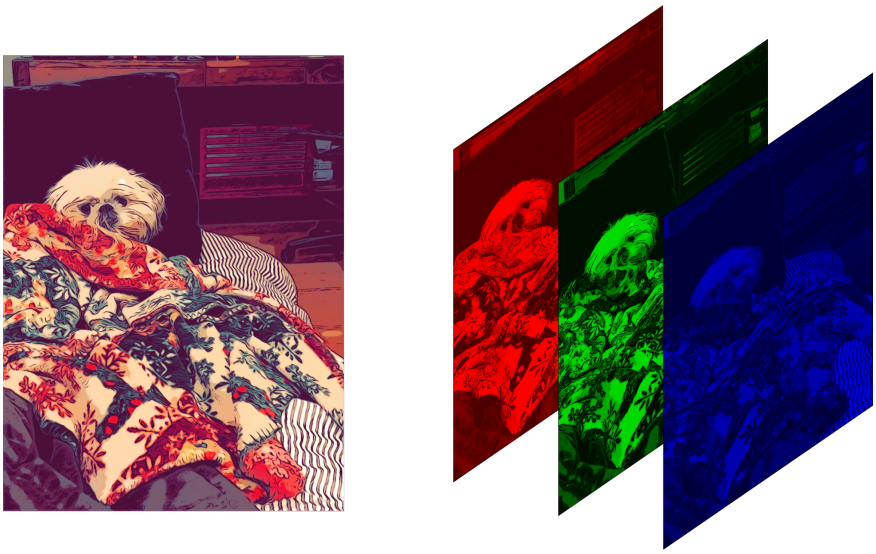

Figura 1. Una imagen en RGB es la suma de tres imágenes, una por cada componente. Fuente: Brandon Rohrer.

Independiente de cómo o dónde se almacene una imagen, para una computadora no es más que un *array* de números (Fig. 2). Esto es así por definición en el modelo *raster*. La manera en que se organiza ese *array*, en cantidad de filas y columnas, determina la resolución de la imagen: la cantidad de píxeles que contiene.

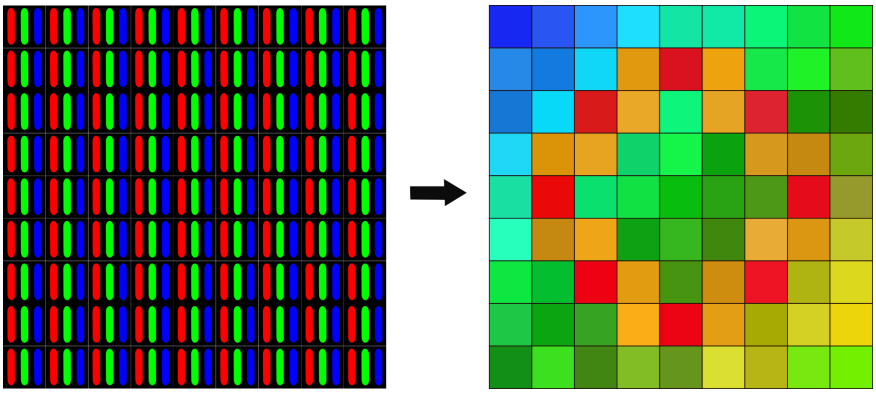

Figura 2. Un imagen es un *array* de colores, y cada color se representa por tres componentes. Fuente: Brandon y Diane Rohrer.

Cada píxel requiere al menos tres bytes: uno para el componente rojo (R), otro para el verde (G) y otro para el azul (B). Para acceder al color de un píxel en la posición (x, y) de una imagen de ancho w, basta calcular el índice en el *array*:

$$\text{index} = (y \cdot w + x) \times 3$$

y luego leer los tres bytes a partir de esa posición:

```
R = pixels[index]
G = pixels[index + 1]
B = pixels[index + 2]
```

Podemos verificar esto cargando una imagen y leyendo sus bytes:

```
pic = pyglet.image.load(filename)
raw_image = pic.get_image_data()
pixels = raw_image.get_bytes(fmt="RGB", pitch=pic.width * len("RGB"))
```

La variable `pixels` es ahora una secuencia de bytes que contiene todos los píxeles de la imagen, fila por fila. Con esto podemos, por ejemplo, inspeccionar el color bajo el cursor del *mouse*:

```
@win.event
def on_mouse_motion(x, y, dx, dy):
    if 0 ⩽ x < pic.width and 0 ⩽ y < pic.height:
        index = int(y * pic.width + x) * 3
        r = int(pixels[index])
        g = int(pixels[index + 1])
        b = int(pixels[index + 2])
        big_pixel.color = (r, g, b)
```

El ejemplo completo muestra la imagen en una ventana y un cuadrado grande al costado que refleja el color del píxel bajo el puntero (Fig. 3). Puedes ejecutarlo con:

```
uv run python caja_de_juguetes.py image_pixel <archivo_de_imagen>
```

Figura 3. Ejemplo `image_pixel` con una imagen de la carrera de caballos más épica que existe. Fuente: Hirohiko Araki & LUCKY LAND COMMUNICATIONS/SHUEISHA, JOJO The Animation Project.

Esta idea (que un píxel es solo una tupla de números) es el punto de partida de todo lo que veremos en esta unidad. Si un píxel no es más que tres valores numéricos, no hay nada que nos obligue a que representen un color: pueden ser coordenadas, velocidades, o el conteo de visitas de un atractor.

Copiar una imagen a la GPU

Para que la GPU pueda trabajar con una imagen, debemos copiar su *array* a la memoria de video (VRAM). En OpenGL, esto se hace con `glTexImage2D`:

```
texture_id = GL.glGenTextures(1)
GL.glBindTexture(GL.GL_TEXTURE_2D, texture_id)
GL.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, GL.GL_NEAREST)
GL.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, GL.GL_NEAREST)

GL.glTexImage2D(
    GL.GL_TEXTURE_2D, 0, GL.GL_RGB,
    width, height, 0,
    GL.GL_RGB, GL.GL_UNSIGNED_BYTE,
    texture_data.tobytes()
)
```

La función `glTexImage2D` recibe los parámetros de tamaño (`width` y `height`), la cantidad de canales (`GL_RGB`: tres; podrían ser cuatro en el caso de RGBA) y el tipo de dato (`GL_UNSIGNED_BYTE`: un byte por canal). Estos determinan el espacio necesario en

VRAM. Más adelante veremos el significado de los demás parámetros; por ahora basta saber que esta llamada copia el *array* de bytes desde la CPU a la GPU, creando una **textura** cuyo identificador quedó almacenado en la variable `texture_id`.

Dibujar en una textura: Framebuffer Objects (FBO)

Hasta aquí, el flujo va en una sola dirección: la CPU genera datos y los sube a una textura en la GPU para mostrarlos. Pero también es posible que la GPU **dibuje en una textura** en vez de en la pantalla.

Cuando la GPU renderiza, escribe el resultado en el *framebuffer* por omisión. Recordemos que el *framebuffer* es la imagen que luego es entregada al *video controller* para ser convertida en una señal que puede interpretar un monitor o proyector. OpenGL permite crear **framebuffer objects** (FBO), que son *framebuffers* alternativos que no se envían al *video controller*, sino que quedan disponibles como una textura a utilizar. Esto se conoce como *render-to-texture*: la GPU dibuja en la textura, y luego podemos leer esa textura, procesarla, o volver a dibujar encima (un ejemplo popular en el DCC es lo mostrado en la Fig. 4).

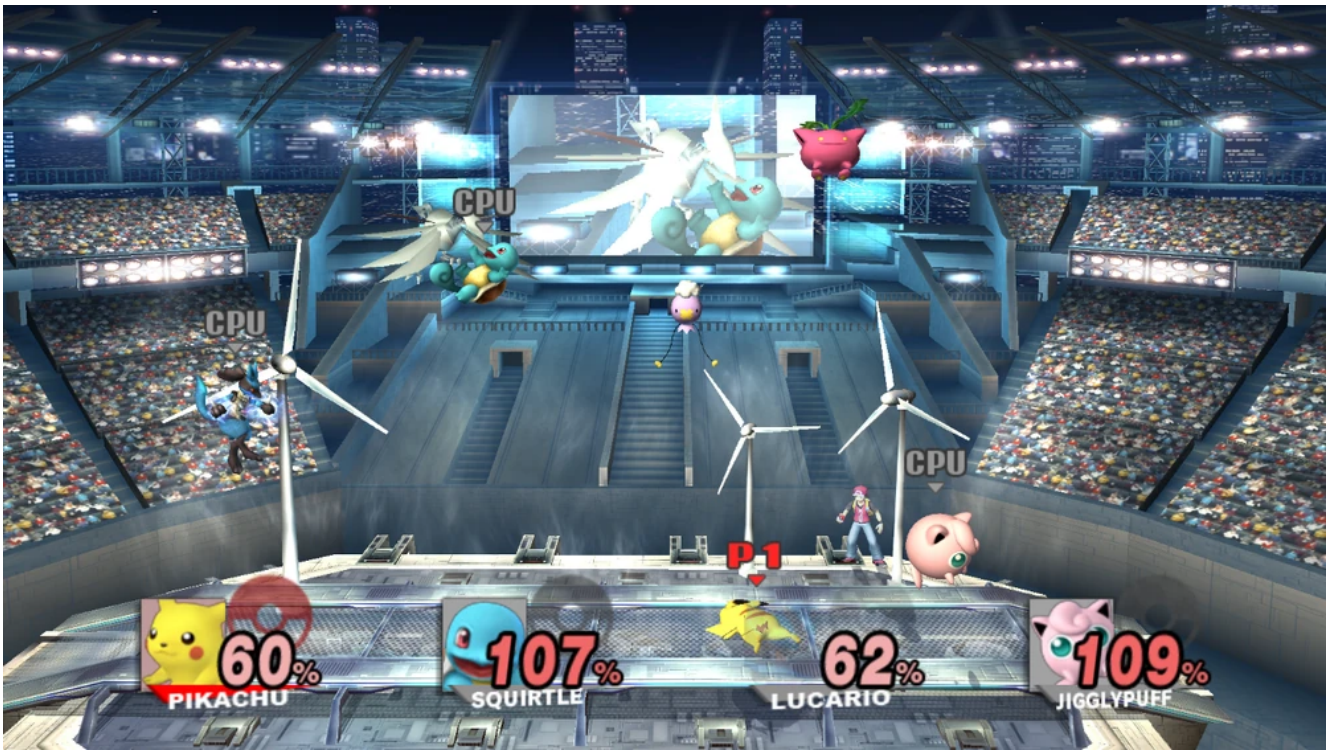

Figura 4. El efecto de pantalla que reproduce lo que está sucediendo en la escena, como en el Estadio Pokemon de *Smash*, se logra con FBO. Fuente: Nintendo.

De hecho, para utilizar un FBO (`fbo` en el siguiente código), es necesario crear una textura primero (`fbo_tex`) que se le entrega como parámetro:

```
# Crear una textura vacía en la GPU
fbo_tex = GL.glGenTextures(1)
GL.glBindTexture(GL.GL_TEXTURE_2D, fbo_tex)
GL.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGB,
                width, height, 0, GL.GL_RGB, GL.GL_UNSIGNED_BYTE, None)


# Crear un FBO que escriba en esa textura
fbo = GL.glGenFramebuffers(1)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo)
GL.glFramebufferTexture2D(GL.GL_FRAMEBUFFER, GL.GL_COLOR_ATTACHMENT0,
                          GL.GL_TEXTURE_2D, fbo_tex, 0)
```

Con `glBindFramebuffer` activamos el FBO; a partir de ese momento, todo lo que la GPU dibuje va a la textura en vez de a la pantalla. Para volver a dibujar en la pantalla, desactivamos el FBO con `glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)`. Usaremos esta técnica más adelante para acumular las visitas de un atractor en la GPU.

Una textura es, entonces, una matriz bidimensional que reside en la memoria de la GPU. Aunque suelen asociarse con imágenes (pues una textura *era* una imagen), hoy podemos utilizarlas para almacenar cualquier tipo de información que requiera procesamiento paralelo[1].

Las texturas están conformadas por téxeles. Un téxel es la unidad mínima de una textura, y si la textura es una imagen 2D, es equivalente a un píxel[2]. Ahora bien, los canales RGB de cada téxel pueden interpretarse como contenedores de datos arbitrarios. Por ejemplo, podemos usar `(r, g, b)` para almacenar coordenadas `(x, y, z)` y el canal `alpha` para información auxiliar como velocidad o tiempo[3].

Más adelante en el curso trabajaremos con texturas desde la perspectiva visual. En esta unidad las trataremos como *buffers de acumulación*, donde cada téxel almacena el estado de la simulación de un sistema dinámico.

[1] Este es uno de los conceptos clave en lo que se conoce como GPGPU: *General Purpose GPU Computing*. Las texturas pueden almacenar datos arbitrarios para cálculos que no sean necesariamente gráficos. Debido a la arquitectura de las GPU, es un desafío implementar algoritmos de esta manera.

[2] Esto no siempre es así, pero en el caso de esta unidad, lo son porque la textura es del mismo tamaño que la imagen a graficar.

[3] Recuerda que en GLSL, si tienes un `vec3` puedes acceder a sus atributos, por ej., `.xy` o `.rg` de manera equivalente.

2 Sistemas dinámicos

Figura 5. Jackson Pollock, *Number 23* (1948). Pollock es conocido por su arte de apariencia aleatoria. Sin embargo, estudios científicos han demostrado que sus patrones son fractales, muy similares a los encontrados en la naturaleza. Los fractales son objetos geométricos que exhiben patrones similares a diferentes escalas de observación.

¿Qué tienen en común una textura y una pintura de Pollock (Fig. 5)? Ambas son matrices de valores; la diferencia es qué proceso las genera. En el caso de Jackson Pollock, el proceso es físico: el artista lanzaba pintura sobre un lienzo en el suelo, y el goteo y salpique acumulados sobre él también son parte de la obra. Nosotros generaremos nuestras texturas con procesos matemáticos que producen patrones complejos: los **sistemas dinámicos**.

Un sistema dinámico está definido por un estado X y una regla f que determina el estado siguiente:

$$X_{t+1} = f(X_t).$$

El estado puede ser un número, un vector o una función. La secuencia $\{X_0, X_1, X_2, \ldots\}$ que produce se llama **trayectoria**, y

queda determinada por la condición inicial X_0 y la regla f. Cuando f es no lineal (contiene productos, valores absolutos u otras operaciones de este tipo sobre X), el sistema puede exhibir **caos determinista**: trayectorias que parecen aleatorias pero están determinadas por sus condiciones iniciales, donde dos puntos iniciales muy cercanos pueden divergir de manera exponencial con el tiempo.

Para visualizar estos sistemas usaremos dos conceptos. El **espacio de fase** es el espacio de todos los estados posibles (por ejemplo, el plano xy para un sistema bidimensional). Un **atractor** es la región del espacio de fase hacia la que convergen las trayectorias. Los hay de tres tipos:

- **Punto fijo**: el sistema se estabiliza en un único estado.

$$\vec{X}(t) \to \vec{X}^* \quad \text{cuando} \quad t \to \infty.$$

- **Ciclo límite**: el sistema oscila.

$$\vec{X}(t+T) = \vec{X}(t) \quad \text{para algún período } T.$$

- **Atractor extraño**: el sistema tiene estructura fractal y comportamiento caótico.

Los atractores extraños son los más interesantes visualmente. Tienen **autosimilaridad**: al ampliar cualquier región, aparecen patrones similares al conjunto completo. Su dimensión fractal es un número no entero (por ejemplo, 1.3: más que una línea, menos que un plano). Sin importar cuánto ampliemos la visualización, siempre encontraremos estructura.

En el contexto del curso, los atractores nos permiten aprender a gestionar el estado de un sistema y a usar las texturas como *buffers* de acumulación. En aplicaciones gráficas, son útiles para generar movimiento orgánico que no se repite: el vaivén de una llama de fuego o el recorrido errático de una luciérnaga pueden modelarse con trayectorias sobre un atractor.

Mapas discretos

Un mapa discreto aplica de manera repetida una función al estado actual. Algunos ejemplos clásicos:

- **Mapa logístico** (unidimensional):

$$x_{n+1} = r\, x_n (1 - x_n).$$

- **Mapa de Henon** (bidimensional):

$$\begin{aligned} x_{n+1} &= 1 - a\, x_n^2 + y_n, \\ y_{n+1} &= b\, x_n. \end{aligned}$$

[4] Lit. "hombre galleta de jengibre".

- **Mapa de Gingerbreadman**[4] (bidimensional):

$$x_{n+1} = 1 - y_n + |x_n|,$$
$$y_{n+1} = x_n.$$

Estos mapas generan comportamientos complejos. Los implementaremos más adelante en esta unidad.

Atractores extraños célebres

El **atractor de De Jong** es un sistema discreto bidimensional:

$$x_{n+1} = \sin(a \cdot y_n) - \cos(b \cdot x_n),$$
$$y_{n+1} = \sin(c \cdot x_n) - \cos(d \cdot y_n).$$

Los parámetros a, b, c y d determinan la forma del atractor. Algunos valores típicos:

- Conjunto 1: $a = 1.4$, $b = -2.3$, $c = 2.4$, $d = -2.1$
- Conjunto 2: $a = -2.7$, $b = -0.09$, $c = -0.86$, $d = -2.2$

Los siguientes dos atractores son **sistemas continuos**: sus ecuaciones están formuladas como derivadas, es decir, describen cómo cambia el estado de manera continua en el tiempo. Sin embargo, una computadora no puede evaluar derivadas de forma exacta; debe integrarlas con pasos discretos. La forma más simple de hacerlo es el **método de Euler**: si la derivada dice $dx/dt = f(x)$, avanzamos un paso pequeño Δt haciendo $x_{n+1} = x_n + f(x_n) \cdot \Delta t$. Esto convierte cualquier sistema continuo en un mapa discreto que podemos iterar igual que los anteriores. Los retomaremos en la unidad siguiente.

- El **atractor de Rossler** es un sistema tridimensional:

$$\frac{dx}{dt} = -y - z,$$
$$\frac{dy}{dt} = x + ay,$$
$$\frac{dz}{dt} = b + z(x - c).$$

  Es caótico para ciertos valores de sus parámetros (un conjunto típico: $a = 0.2$, $b = 0.2$, $c = 5.7$). Para visualizarlo en 2D basta con proyectar sobre el plano xy.

- El **atractor de Lorenz** es quizás el más famoso:

$$\frac{dx}{dt} = \sigma(y - x),$$
$$\frac{dy}{dt} = x(\rho - z) - y,$$
$$\frac{dz}{dt} = xy - \beta z.$$

Con su característica forma de “alas de mariposa” (Fig. 6), este sistema ha llegado a simbolizar el caos determinista. Parámetros típicos: $\sigma = 10$, $\rho = 28$, $\beta = 8/3$.

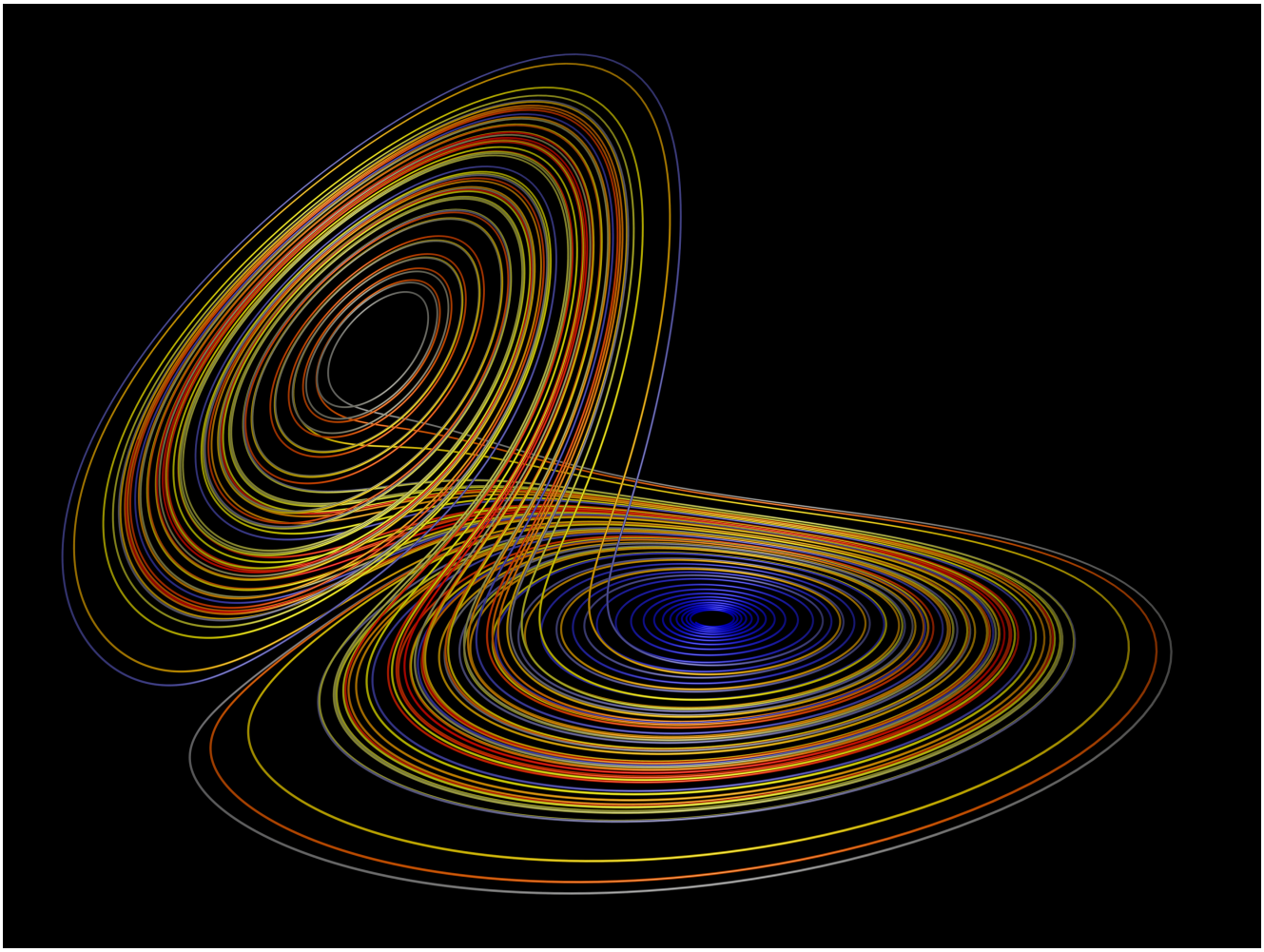

Figura 6. Atractor de Lorenz, gráfico por Paul Bourke. Fuente: https://paulbourke.net/fractals/lorenz/.

3 Implementación: el Sr. Jengibre

Implementaremos el mapa de Gingerbreadman, que llamaremos *Sr. Jengibre*, porque es simple pero produce resultados atractivos. Haremos dos versiones: primero una que se ejecuta en la CPU (con NumPy), y luego una versión semi-GPU que mantiene la iteración en CPU pero mueve la acumulación a la GPU usando un *framebuffer* con *blending* aditivo.

Versión CPU

En esta versión, todo el cálculo ocurre en Python. El flujo tiene tres pasos: iterar el mapa y acumular en un *array*, convertir el *array* a una imagen RGB y transferirla a la GPU como textura para mostrarla en pantalla (Fig. 7). Puedes ejecutar el ejemplo con el comando:

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

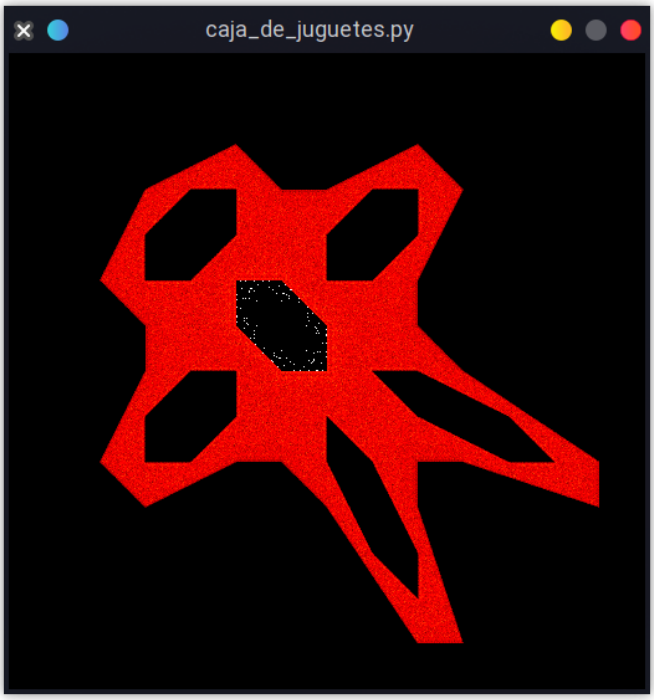

Figura 7. *Sr. Jengibre (numpy).*

Paso 1: Implementar el mapa en Python

El Sr. Jengibre se define mediante las ecuaciones:

$$x_{n+1} = 1 - y_n + |x_n|,$$

$$y_{n+1} = x_n.$$

Su implementación en Python es directa:

```python
def gingerbread_step(x, y):
    """Un paso del mapa de Gingerbreadman"""
    x_new = 1 - y + np.abs(x)
    y_new = x
    return x_new, y_new
```

Utilizamos múltiples partículas con condiciones iniciales aleatorias y acumulamos sus trayectorias en un *buffer*:

```
num_particles = 10
particles_x = np.random.uniform(-0.5, 0.5, num_particles)
particles_y = np.random.uniform(-0.5, 0.5, num_particles)

accumulator = np.zeros((height, width), dtype=np.float32)

def world_to_pixel(x, y, width, height):
    """Convierte coordenadas del espacio de fase a píxeles"""
    x_min, x_max = -7.0, 7.0
    y_min, y_max = -7.0, 7.0

    px = int((x - x_min) / (x_max - x_min) * width)
    py = int((y - y_min) / (y_max - y_min) * height)

    if 0 ⩽ px < width and 0 ⩽ py < height:
        return px, py
    return None, None

# Actualizar y acumular
for i in range(num_particles):
    particles_x[i], particles_y[i] = gingerbread_step(particles_x[i], particles_y[i])

    px, py = world_to_pixel(particles_x[i], particles_y[i], width, height)
    if px is not None and py is not None:
        # Acumular visitas con saturación controlada
        if accumulator[py, px] < max_value_per_pixel:
            accumulator[py, px] += 0.5
```

La técnica de acumulación permite visualizar la densidad de la trayectoria: las regiones más visitadas aparecerán más brillantes, lo que revelará la estructura del atractor.

Paso 2: Traspasar la imagen a la GPU

El acumulador es un array de `float32` con valores que van desde 0 hasta `max_value_per_pixel`. Antes de enviarlo a la GPU con `glTexImage2D` (como vimos al inicio), debemos convertirlo a datos RGB de 8 bits. Esto requiere dos transformaciones: normalizar los valores al rango $[0, 1]$ y aplicar una paleta de color.

```
def create_texture_data():
    """Convierte el acumulador a datos de textura RGB"""
    if accumulator.max() > 0:
        # Normalización con raíz cuadrada para suavizar
```

```
        normalized = np.sqrt(accumulator / max_value_per_pixel)
        normalized = np.clip(normalized, 0, 1)
    else:
        normalized = accumulator

    # Crear imagen RGB con gradiente de color
    texture_data = np.zeros((height, width, 3), dtype=np.uint8)
    intensity = (normalized * 255).astype(np.uint8)

    # Paleta: azul oscuro → amarillo brillante
    texture_data[:, :, 0] = intensity  # rojo
    texture_data[:, :, 1] = (intensity * 0.8).astype(np.uint8)  # verde
    texture_data[:, :, 2] = ((1.0 - normalized) * 100).astype(np.uint8)  # azul

    # Voltear para OpenGL (origen abajo a la izquierda)
    return np.flipud(texture_data)
```

El resultado es un array de forma `(height, width, 3)` con valores `uint8`: el formato que espera `glTexImage2D` con `GL_RGB` y `GL_UNSIGNED_BYTE`. La normalización con raíz cuadrada comprime el rango dinámico: sin ella, los pocos píxeles muy visitados dominarían la imagen mientras que la mayoría sería casi invisible.

Paso 3: Pintar la textura

Para mostrar la textura en pantalla, creamos un cuadrilátero que cubra toda la ventana utilizando dos triángulos:

```
# Geometría: cuadrilátero que cubre la pantalla
vertices = np.array([-1, -1, 1, -1, 1, 1, -1, 1], dtype=np.float32)
uv = np.array([0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0], dtype=np.float32)
indices = np.array([0, 1, 2, 2, 3, 0], dtype=np.uint32)

# En el renderizado
GL.glActiveTexture(GL.GL_TEXTURE0)
GL.glBindTexture(GL.GL_TEXTURE_2D, texture_id)

# Configurar uniform del sampler
sampler_location = GL.glGetUniformLocation(pipeline.id, "sampler_tex")
GL.glUniform1i(sampler_location, 0)

gpu_data.draw(GL.GL_TRIANGLES)
```

El *vertex program* mapea el cuadrilátero a coordenadas de pantalla (algo que aprenderemos más adelante cuando veamos 3D,

por ahora solo asumimos que cubre toda la ventana), mientras que el *fragment program* muestra el atractor acumulado a lo largo del tiempo.

Versión semi-GPU

La versión anterior calcula todo en la CPU: la iteración del mapa, la acumulación en un array NumPy y la conversión a colores RGB. La GPU solo recibe la imagen final y la muestra. Podemos mover la acumulación a la GPU usando un FBO con *blending* aditivo[5]. Puedes ejecutar esta versión del programa con:

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

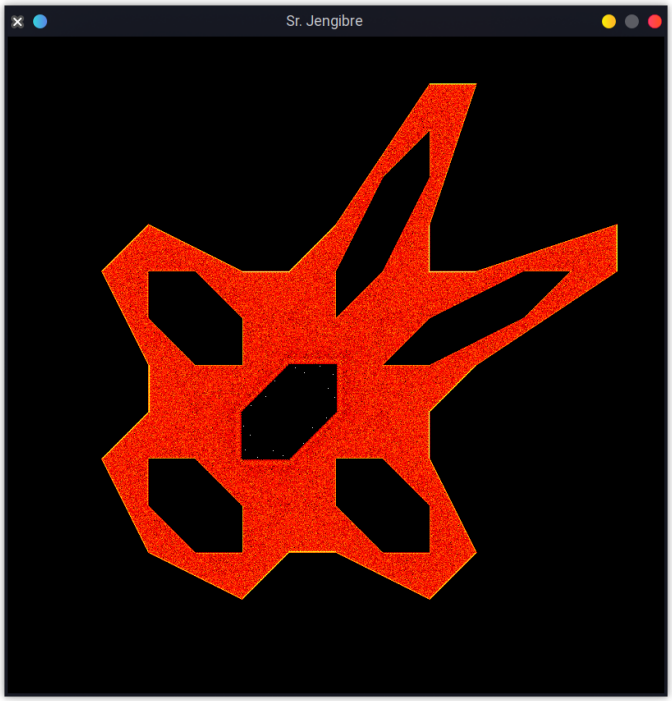

Figura 8. *Sr. Jengibre (semi GPU).*

En vez de mantener un *array* `accumulator` en NumPy, mantenemos una textura en la GPU a través de un FBO. En cada *frame*, la CPU itera las partículas y envía los puntos visitados como primitivas `GL_POINTS`. La GPU acumula estos puntos en la textura usando *blending* aditivo (`GL_ONE, GL_ONE`), donde cada punto suma un valor constante al píxel correspondiente. Un segundo *fragment program* de visualización lee la textura acumulada y aplica *tone mapping* (raíz cuadrada con exposición ajustable) y una paleta de color (Fig. 8).

[5] ¿Qué es el *blending*? Cuando la GPU va a escribir un color en un píxel que ya tiene un valor, este último es reemplazado. El *blending* (mezcla) permite combinar el color nuevo con el existente según una regla configurable. En nuestro caso usamos **blending aditivo**: el nuevo valor se **suma** al anterior en vez de reemplazarlo. Esto es lo que necesitamos para acumular visitas; cada punto que pasa por un píxel incrementa su valor, igual que hacía `np.add.at` en la versión CPU, pero ejecutado en la GPU. Veremos el *blending* con más detalle en la unidad de texturas para rendering, donde lo usaremos para transparencia y otros efectos visuales.

```
# Crear textura float para acumular sin saturar
accum_tex = GL.glGenTextures(1)
GL.glBindTexture(GL.GL_TEXTURE_2D, accum_tex)
GL.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_R16F,
                width, height, 0, GL.GL_RED, GL.GL_FLOAT, None)

# Crear FBO que escribe en esa textura en vez de en la pantalla
fbo = GL.glGenFramebuffers(1)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo)
GL.glFramebufferTexture2D(GL.GL_FRAMEBUFFER, GL.GL_COLOR_ATTACHMENT0,
                          GL.GL_TEXTURE_2D, accum_tex, 0)
```

La textura usa formato `GL_R16F` (un canal de 16 bits en punto flotante) en lugar de `GL_RGB` con `GL_UNSIGNED_BYTE`. Esto es crucial: con enteros de 8 bits, el valor máximo por píxel sería 255 y la imagen se saturaría. Con punto flotante, podemos acumular miles de visitas sin perder detalle; el *tone mapping* en el *shader* de visualización se encarga de comprimir el rango dinámico a valores visibles.

Cada frame, las coordenadas del espacio de fase se convierten a *Normalized Device Coordinates* (NDC), el sistema de coordenadas de OpenGL donde $(-1, -1)$ es la esquina inferior izquierda y $(1, 1)$ la superior derecha:

```
ndc_x = 2.0 * (x - x_min) / (x_max - x_min) - 1.0
ndc_y = 2.0 * (y - y_min) / (y_max - y_min) - 1.0
```

Estos puntos se suben a un *vertex buffer object* (VBO) y se dibujan con blending aditivo:

```
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo)
GL.glEnable(GL.GL_BLEND)
GL.glBlendFunc(GL.GL_ONE, GL.GL_ONE)  # aditivo: dst = dst + src

GL.glBindBuffer(GL.GL_ARRAY_BUFFER, vbo)
GL.glBufferData(GL.GL_ARRAY_BUFFER, points.nbytes, points, GL.GL_STREAM_DRAW)
GL.glDrawArrays(GL.GL_POINTS, 0, num_points)

GL.glDisable(GL.GL_BLEND)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)
```

Si ejecutas ambas versiones, notarás que la imagen aparece reflejada verticalmente. Esto no es un error, sino una conse-

cuencia directa de cómo NumPy y OpenGL organizan sus datos en memoria.

En la versión NumPy, el acumulador es un *array* 2D indexado como `accumulator[py, px]`, donde `py` se calcula así:

```
py = int((y - y_min) / (y_max - y_min) * height)
```

El valor mínimo de y (y_{min}) corresponde a `py = 0`, que es la **fila 0** del *array*. En NumPy, la fila 0 es la primera en memoria (la parte "superior" si visualizamos la matriz como tabla). Cuando `glTexImage2D` recibe estos datos, coloca la fila 0 en la **base** de la textura, porque OpenGL tiene su origen en la esquina inferior izquierda. Hasta aquí, todo calza: y_{min} queda abajo y el eje y crece hacia arriba, que es la convención matemática estándar.

Sin embargo, el código aplica `np.flipud(texture_data)` antes de transferir los datos. Esto invierte todas las filas: la fila 0 (que era y_{min}) pasa al final, y OpenGL la coloca ahora en la parte superior de la textura. El resultado es que el eje y crece hacia **abajo**, invirtiendo la orientación.

En la versión semi-GPU, la conversión a NDC mapea y_{min} a -1 (abajo) e y_{max} a $+1$ (arriba), por lo que el eje y crece hacia arriba sin ningún paso adicional.

Entonces, ¿por qué existe el `flipud`? Es un patrón común cuando se trabaja con **imágenes**: las imágenes almacenan la primera fila de píxeles en la parte superior (la primera *scanline*), y como OpenGL espera la primera fila abajo, hay que invertirlas. Pero cuando los datos son sintéticos (generados por nuestro código), la fila 0 del array ya corresponde a y_{min}, que queda abajo en OpenGL. El `flipud` en ese caso invierte la orientación en vez de corregirla.

El ejemplo incluye las opciones `--flip` y `--no-flip` para experimentar con esto:

```
# eje y hacia abajo
uv run python caja_de_juguetes.py sr_jengibre_numpy --flip
# eje y hacia arriba (= versión semi-GPU)
uv run python caja_de_juguetes.py sr_jengibre_numpy --no-flip
```

4 Ejemplo Buddhabrot

El conjunto de Mandelbrot puede entenderse como un mapa de "supervivencia" matemática que clasifica cada punto del plano complejo según su destino final. Al elegir un número como punto de partida e iterarlo con la fórmula $z_{n+1} = z_n^2 + c$ (con $z_0 = 0$), observamos si el resultado se mantiene bajo control o si crece hacia el infinito como una explosión demográfica. En la representación visual clásica, lo que vemos es el veredicto final para cada coordenada: el color negro señala a los puntos que se quedan atrapados en un ciclo infinito de estabilidad, mientras que los colores vibrantes de los bordes marcan a los que "escapan", graduados según la velocidad con la que perdieron el control. Es, en esencia, una fotografía estática que divide el plano en dos mundos: los que pertenecen al conjunto y los que no (Fig. 9).

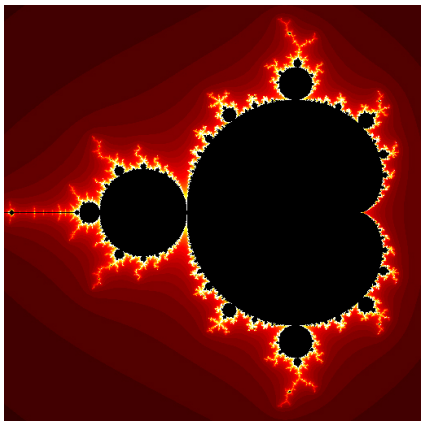

Figura 9. Visualización del [conjunto de Mandelbrot] utilizando el método de tiempo de escape. Fuente: Wikipedia

Para un punto c elegido al azar en el plano complejo, se itera la función de Mandelbrot y se almacena la trayectoria completa $\{z_0, z_1, z_2, \ldots, z_N\}$. Si la trayectoria escapa (es decir, $|z_n| > 2$ para algún $n \leq N_{\text{max}}$), se recorre la trayectoria guardada y se incrementa un contador en cada píxel por el que pasó. Si no escapa, se descarta: los puntos que pertenecen al conjunto de Mandelbrot no contribuyen a la imagen. La imagen resultante tiene una apariencia orgánica que recuerda a una figura sentada en meditación (Fig. 10). Puedes ejecutar el ejemplo con:

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

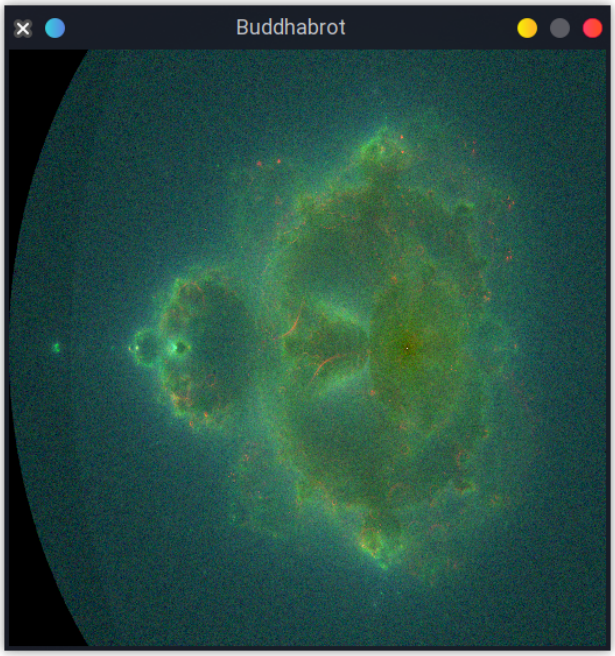

Figura 10. Buddah + Mandelbrot (debes mirar de lado). La implementación usa la variante llamada *Nebulabrot*, que asigna distintos límites de iteración a cada canal RGB, separando las estructuras de baja, media y alta frecuencia en colores distintos.

En los ejemplos vistos, migrar la iteración misma a la GPU requeriría *compute shaders,* una funcionalidad de OpenGL 4.3 que permite ejecutar programas de propósito general; eso está fuera del alcance de este curso, donde nos limitamos a *vertex* y *fragment programs*.