

# Shaders de Juguete (Shadertoy)

Usando trucos matemáticos para dibujar en tu navegador web

Dr. Jorge Antonio García Galicia

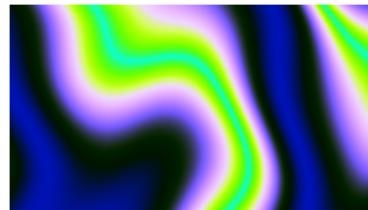
[nemediano.github.io](https://nemediano.github.io)

Congreso de Inteligencia Artificial, Abril 2025

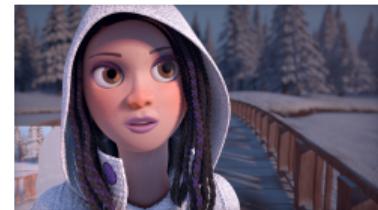


# ¿Qué esperar del taller?

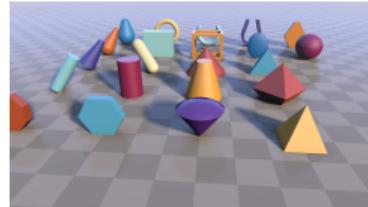
- Aprender el paradigma de [Rendering Worlds With Two Triangles](#)
- Parte del [creative coding](#): intersección entre el arte y la ciencia.
- Un poco de [GLSL](#), y las matemáticas básicas para hacer un [fragment shader](#) en el paradigma.



Swirly Whirly



Selfie Girl



Raymarch Primitives



Centric Mosaic Tiles

Propiedad de los respectivos autores

# ¿Qué *no* es éste taller?

No es un curso *completo* de “Graficación por Computadora”. Mas aun, no vamos a enseñar a hacer gráficas por computadora de manera correcta.

- Usar este paradigma **es un juego**. No se usa en producción.
- Los shaders que vamos a escribir son ineficientes.
- El código puede ser ofuscado.

Por razones de tiempo, tampoco enseñaremos todo lo que hay en el truco

- Este es un taller **básico** de shaders de juguete (Shadertoy).
- Revisaremos un conjunto de técnicas simples
- Recuerda: hacer shaders complejos, requiere de experiencia, creatividad y tiempo

# ¿Qué puedo aprovechar de éste taller?

- Aprender una manera de expresión artística usando programación
- Un excelente ejercicio para fomentar creatividad
- Una forma de practicar, aplicar y mejorar tus conocimientos matemáticos y de computación
- Un medio de compartir con una comunidad (Shadertoy tiene elementos de red social)
- Se convierte en un hobby productivo

## Graficación por Computadora

- Una probadita para saber si estas interesado en tomar un curso formal después.
- Si ya tomaste el curso, es un *hack* muy elegante que seguramente no se te había ocurrido
- Muchos profesionales del área lo practican como hobby.

# Acerca de mí

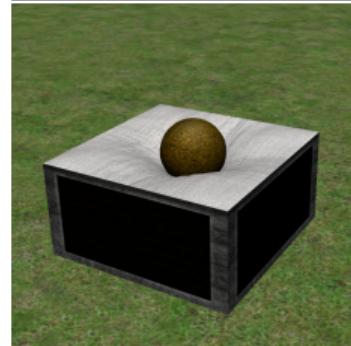
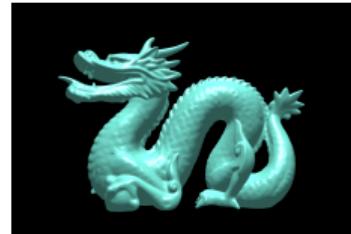
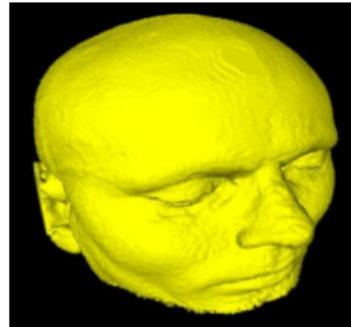


Dr. Jorge Antonio García Galicia

- Licenciatura y maestría en la UNAM
- Doctorado en Purdue University
- Hice una pasantía en Adobe y trabajé en Nvidia
- He dado clases en Acatlán y fui ayudante Ciencias y en Purdue
- Actualmente soy SWE en Google
- Tengo mas de 15 años de experiencia en Tecnología

# Mi experiencia con gráficos

- Tres tesis que tienen que ver con Graficación por Computadora
- Publicado en Leonardo, SIGGRAPH y en Eurographics
- Trabaje en el driver de OpenGL y en Stadia
- Actualmente trabajo en AndroidXR
- Provengo de una familia de artistas



# El GPU

## Graphics processing unit (GPU)

Un procesador programable con alta capacidad de computo en paralelo, que generalmente opera con imágenes digitales.

- Originalmente estaba dedicado a hacer gráficos.
- Ahora hace computo general, de hecho es el **AI acelerator** mas usado actualmente.
- No confundir con una tarjeta de video: Todas la tarjetas de video (actuales) tiene un GPU. Pero no todos los GPU están en tarjetas de video.
- Cuando un GPU esta físicamente separado del chip que tiene el CPU se le llama un GPU discreto.

# ¿Qué es un shader?

## Shader

Un programa que es ejecutado de manera paralela en el GPU como parte de un pipeline.

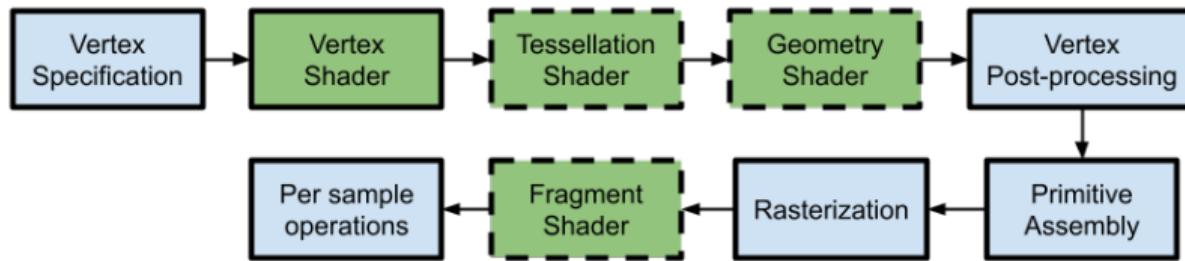
Acerca del nombre:

- El termino fue acuñado por Pixar en 1988 para Renderman.
- Originalmente (2001), los shaders solo hacian calculos de iluminacion: intensidad de la luz, color, sombras y brillos.
- De ahí su nombre, que significa *sombreado*

# La pieza básica

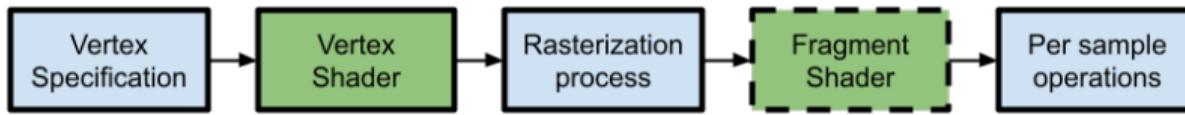
## Pipeline gráfico

Es una abstracción de SW, que describe el proceso que debe seguir un programa para trasformar una escena tridimensional en una imagen.



# Simplificando...

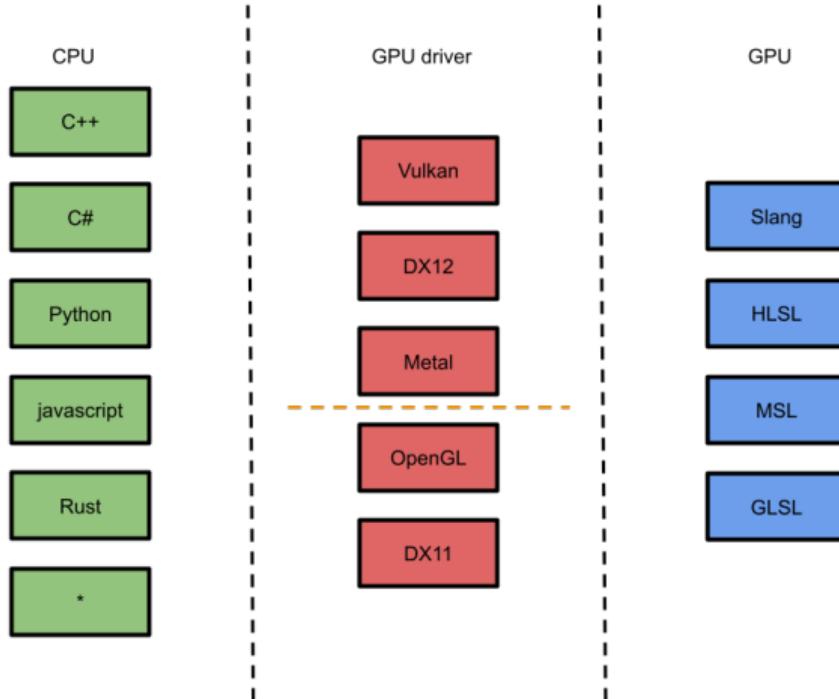
- En la práctica, los tessellation shaders y los geometry shaders solo se usan para cosas muy específicas.
- Mas aún, casi nunca se configuran los estados entre el vertex shader y la rasterización
- Para propósitos de esta explicación, podemos pensarlo de manera mas simple:



# ¿Cómo se programa una aplicación gráfica? I

- En general tienes que usar al menos tres cosas:
  - ① Un lenguaje de programación del lado del CPU para crear una aplicación.
  - ② Un API gráfico, para construir, configurar y conectar el pipeline.
  - ③ Un lenguaje para escribir shaders.
- En este taller solo escribiremos código de shaders usando [GLSL](#).
- Pero sin darnos cuenta el navegador de hecho usa: [javascript](#) y [WebGL](#), que es un subconjunto de [OpenGL](#).

# ¿Cómo se programa una aplicación gráfica? II

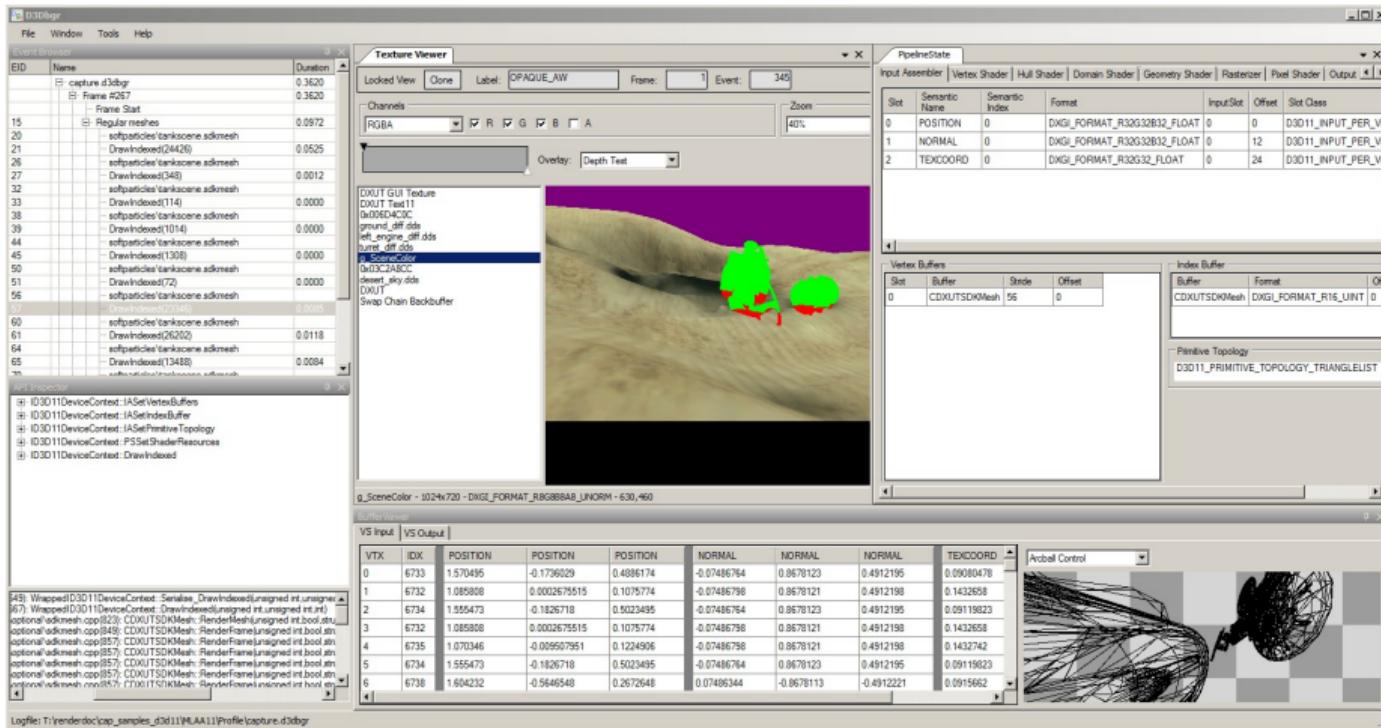


# Anatomía de una aplicación I

Actualmente, una aplicación típica:

- Esta escrita en un engine gráfico (Como [Unity](#) o [Unreal](#))
- Para producir un frame, se usan cientos de pipelines.
  - Varios pipelines, producen resultados en texturas que luego otros pipelines consumen
  - Cada material de los objetos de la escena tiene su propio pipeline
  - Hay pipelines especializados en ciertas tareas (Sombras, reflexiones, iluminación, etc.)
  - Algunos son directos, otros diferidos, otros por mosaicos
  - Sin contar que muchos efectos se llevan a cabo en compute shaders

# Anatomía de una aplicación II



# ¿Qué nos depara el futuro?

Hay una tendencia importante hacia:

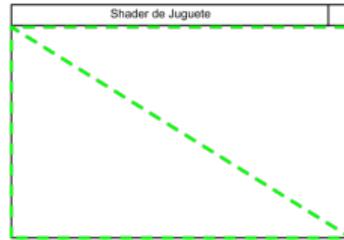
- Unificar en el modelo de iluminación PBR
- Las técnicas basadas en rayos: Raytracer, Raymarching, Pathtracers
  - Generar parte de los fragmentos de un frame, usando técnicas de AI en vez de calcularlos
- Definir un nuevo pipeline: Task (Amplification) shaders, mesh shaders
- Descargar la mayor parte de la tareas en el compute shader



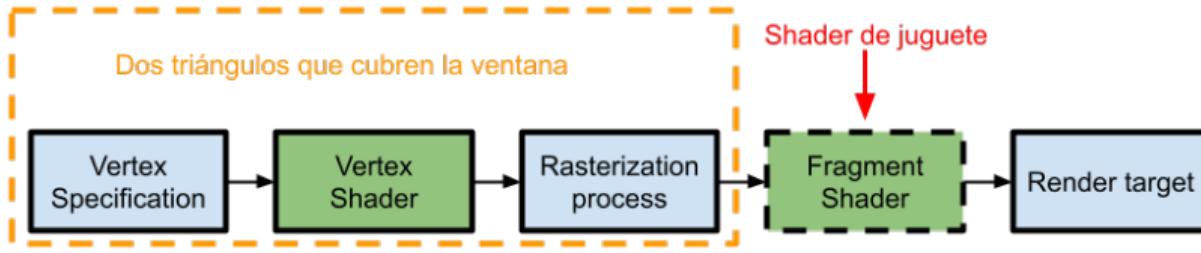
# Una idea ingeniosa

Todo empezó con una plática: “Rendering Worlds With Two Triangles” presentada en la conferencia NVScene el 22 Aug 2008 por Iñigo Quilez

- ① Si dibujamos un cuadrado (formado por 2 triángulos), que cubre toda la ventana. Esto provocará la ejecución del fragment (pixel) shader en todos los píxeles de la pantalla.
- ② Luego, usamos el fragment shader en donde cada fragment (pixel) sabe su respectiva posición en el render target, unas constantes (uniforms) y un montón de matemáticas, para dibujar la escena.

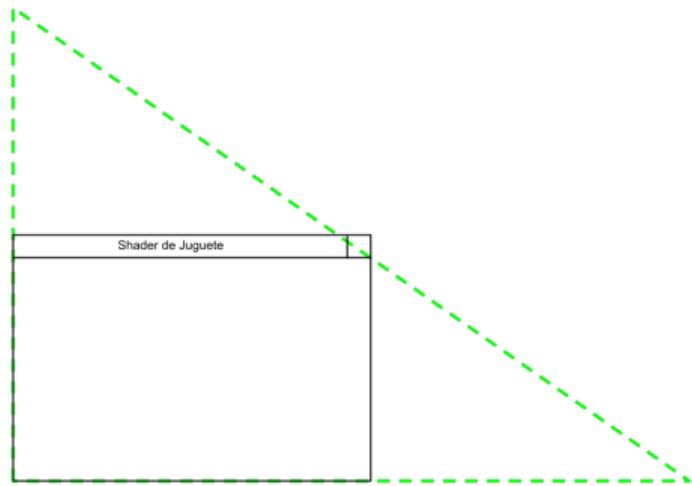


# Regresar al principio



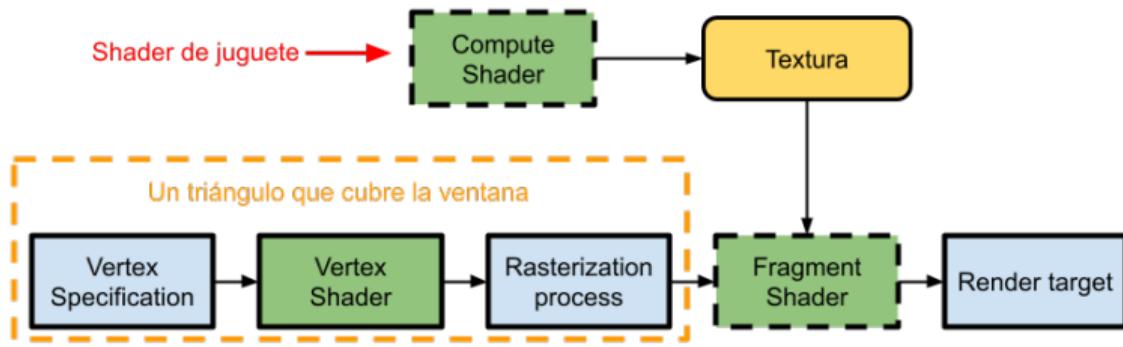
- Es similar a como se hacían los gráficos antes de que tuviéramos tarjetas de video.
- Solo que ahora aprovechamos el **inmenso poder paralelo** del GPU
- Y usamos GLSL

## Nota curiosa



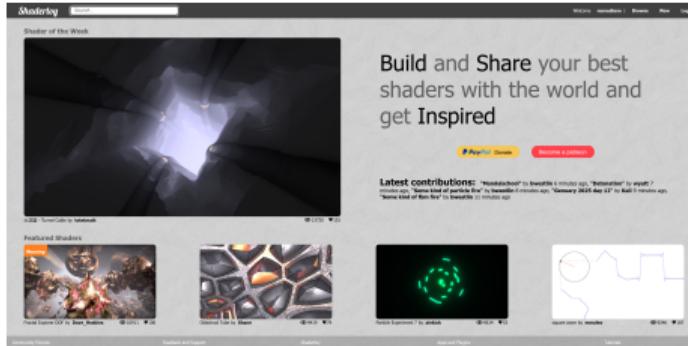
De hecho podemos hacerlo con un solo triángulo y luego hacemos clipping.

Actualmente, es mejor hacer:



# Shadertoy

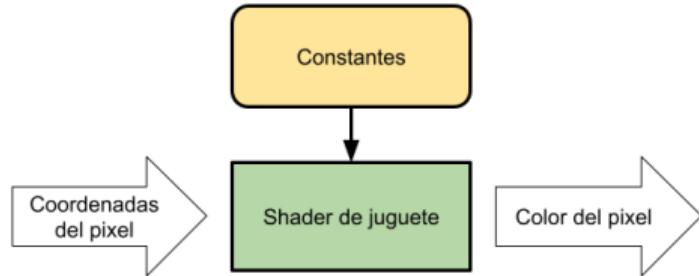
- Shadertoy es un sitio web: <https://www.shadertoy.com/> que nos da la infraestructura para usar el truco
- Y ciertas herramientas sociales
- Fue creado por [Iñigo Quilez](#) y [Pol Jeremias](#) en el 2013



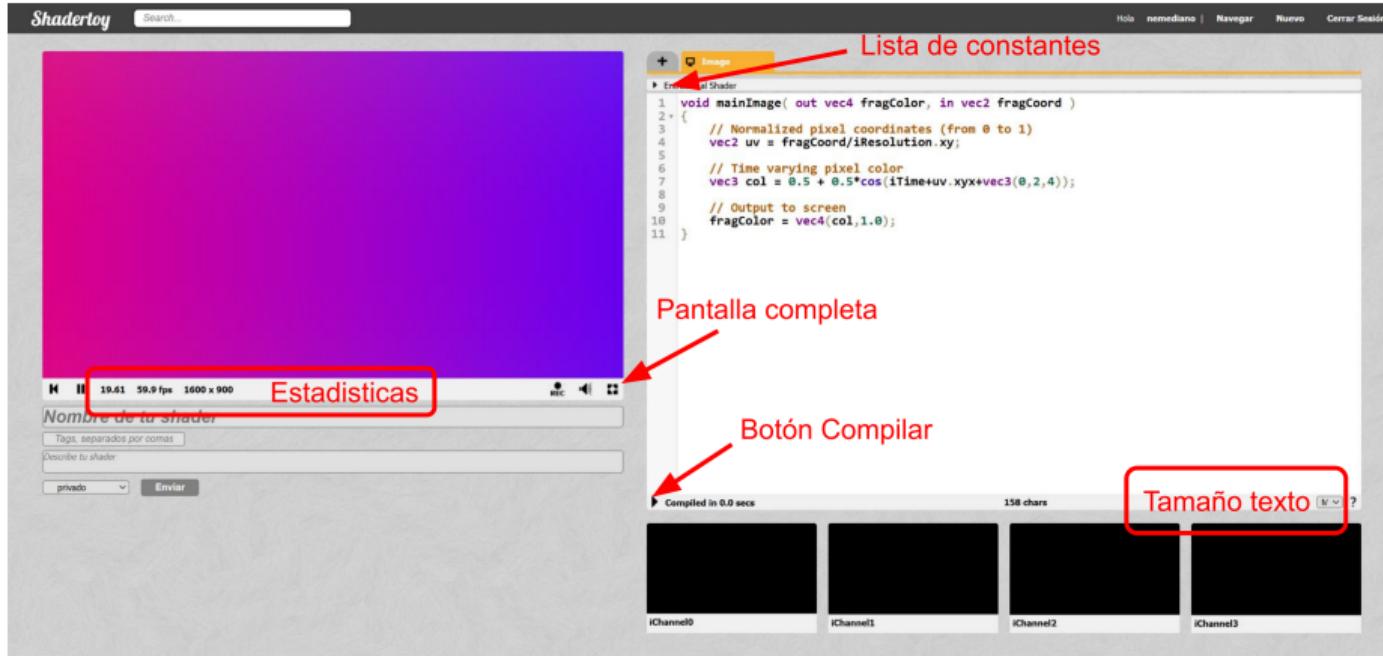
# ¿Cómo se usa?

Escribir un programa en [GLSL](#) que se ejecuta en paralelo por cada pixel de la salida.

- **Entradas:** recibes la coordenada del pixel
- **Salida:** debes regresar el color del pixel
- Recibe algunas constantes extra: el tiempo, el tamaño total del render target, etc.



Así se ve



# Portabilidad

Shadertoy:

- Crea el pipeline por nosotros
- Inyecta código en el fragment shader

Pero ten por seguro que:

Cualquier shader que funciona en shadertoy, puede ser **portado** y funcionar en cualquier otro entorno que pueda desplegar gráficos.

Adicionalmente:

- Hay muchos entornos de programación, que emulan Shadertoy
- Siempre puedes escribir tu propio programa que ejecute shaders de shadertoy

No es tan difícil, cualquier alumno que haya tomado una clase de CG puede hacerlo fácilmente.

# Tipos primitivos

Esta **no es** una lista exhaustiva, ante la duda consulta la [referencia](#).

- Tipos de escalares: `float`, `int`, `bool`.
- Vectores de  $n \in \{2, 3, 4\}$  dimensiones: `vec3`, `ivec2`, `bvec4`.
  - Nótese que cuando el tipo subyacente es `float` no se requiere el prefijo.
- Los operadores aritméticos entre vectores se aplican por componente.
  - Requieren que los operandos sean del mismo tamaño
- Matrices de  $n \times m$  dimensiones donde  $n, m \in \{2, 3, 4\}$  dimensiones: `mat2x3`, `mat3x4`.
  - Todas las matrices son de tipo `float`
  - Las matrices cuadradas se pueden abbreviar: `mat2`
  - Las operaciones entre matrices, son las esperadas del álgebra lineal
- Las multiplicaciones: “matriz por vector”, “escalar por vector” y “escalar por matriz” son las definida en álgebra lineal.

# Swizzling

- Los tipos vectoriales tienen accesores y mutadores a sus componentes individuales.
- Los accesores tienen tres sintaxis equivalentes:  $(x, y, z, w)$ ,  $(r, g, b, a)$ ,  $(s, t, p, q)$ .

Esto define el llamado Swizzling:

---

```
vec2 someVec;
vec4 otherVec = someVec.xyxx;
vec3 thirdVec = otherVec.zyy;
vec4 fourthVec;
// Tambien funciona en l-values:
fourthVec.wzyx = vec4(1.0, 2.0, 3.0, 4.0); // Reverses the order.
fourthVec.zx = vec2(3.0, 5.0); // Sets the 3rd component to 3 and the 1st component to 5
```

---

# Constructores de vectores

Se pueden construir a partir de:

- Vectores de mayor dimensión (los componentes extra son ignorados).
- Una combinación de escalares y de vectores de menor dimensión.
- De manera abreviada, especificando un solo escalar que se repite.

---

```
vec4(vec2(10.0, 11.0), 1.0, 3.5) == vec4(10.0, vec2(11.0, 1.0), 3.5);
vec3(vec4(1.0, 2.0, 3.0, 4.0)) == vec3(1.0, 2.0, 3.0);
vec4(vec3(1.0, 2.0, 3.0)); // error. Not enough components.
vec2(vec3(1.0, 2.0, 3.0)); // OK
vec3(1.0) // Abbreviation to say: vec3(1.0, 1.0, 1.0);
```

---

# Constructores de matrices

- Se construyen por columnas.
- Se pueden construir a partir de matrices de menor o igual dimensión.
- O de manera abreviada especificando un solo escalar que llena la diagonal.

---

```
mat2(  
    float, float,    // first column  
    float, float); // second column  
  
mat4(  
    vec4,           // first column  
    vec4,           // second column  
    vec4,           // third column  
    vec4);          // fourth column  
  
mat3 diagMatrix = mat3(5.0); // Diagonal matrix with 5.0 on the diagonal.  
mat4 otherMatrix = mat4(diagMatrix); // The last element on the diagonal is 1.0
```

---

# Built-in functions

Además de las funciones habituales esperadas, algunas funciones interesantes:

<code>vec3 mix(vec3 x, vec3 y, float a)</code>	Interpolación lineal
<code>vec3 step(float edge, vec3 x)</code>	Función escalón
<code>vec3 smoothstep(float e0, float e1, vec3 x)</code>	Interpolación de Hermite
<code>float dot(vec3 x, vec3 y)</code>	Producto punto
<code>vec3 cross(vec3 x, vec3 y)</code>	Producto cruz
<code>vec3 reflect(vec3 i, vec3 n)</code>	Reflejar <code>i</code> a partir de <code>n</code>
<code>vec3 clamp(vec3 x, vec3 min, vec3 max)</code>	Limitar entre dos valores
<code>float length(vec3 x)</code>	Norma de un vector
<code>vec3 normalize(vec3 x)</code>	Vector normalizado
<code>float distance(vec3 x, vec3 y)</code>	Distancia entre dos vectores

Todas las funciones tienen las sobrecargas vectoriales, cuando corresponde.

# Repositorio

Todo lo necesario para este taller esta en este repositorio:

[https://github.com/nemediano/tallerShadertoy.](https://github.com/nemediano/tallerShadertoy)

- Ésta presentación, también esta en el mismo repositorio. Así puedes seguir los links.
- Para cada ejercicio, hay dos versiones de código fuente.
  - ① El código mínimo para empezar el ejercicio
  - ② Una posible solución al ejercicio

# Últimos detalles

En shadertoy, la ejecución inicia y termina con la función:

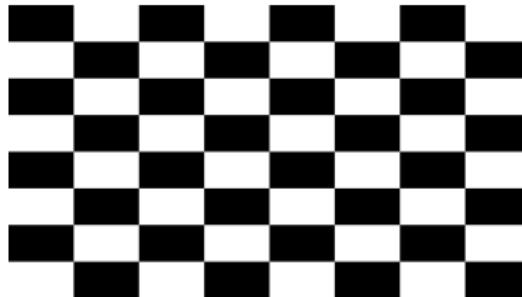
```
void mainImage(out vec4 fragColor, in vec2 fragCoord).
```

- Por cada fragment recibes de parámetro: `fragCoord`, con la posición del fragment en el *render target*.
- La salida: `fragColor`, es un output parameter. Un vector de dimensión 4 que debe contener el color final del fragment.
  - La salida  $\mathbf{x} \in \mathbb{R}^4$  debe tener  $x_i \in [0, 1]$ .
  - El último componente  $x_4$  (ó bien  $w$ ), representa el componente alpha, que en Shadertoy debe ser 1.

# Ejercicio: Bandera a cuadros

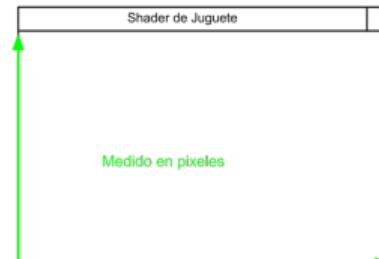
<https://github.com/nemediano/tallerShadertoy/tree/main/codigo/Ejercicio1>

- Tratar de escribir un shader que genere una bandera a cuadros (o tablero de ajedrez si lo prefieres)
- Puedes empezar con el código de default de shader toy.
- Recuerda las funciones trigonométricas



# Sistema de coordenadas

- Al principio, las coordenadas del fragmento están en el espacio del imagen del render target. Miden pixeles.
- Cuando dividimos entre la resolución del render target, están en coordenadas de textura (*uv-mapping*).  $u, v \in [0, 1]$



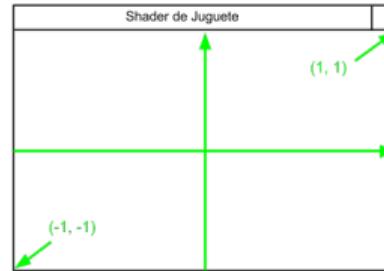
Espacio de imagen



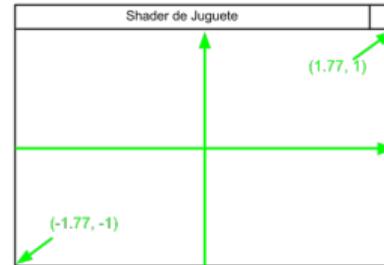
uv space

## Sistema de coordenadas II

- Cuando restamos 0.5 y multiplicamos por dos. Están en coordenadas normalizadas.  $x, y \in [-1, 1]$ .
- Cuando corregimos con el *aspect ratio* de la pantalla, están en coordenadas de la escena. El origen esta en el centro, el eje mas restrictivo esta en  $[-1, 1]$  y el otro es proporcional.



Normalize coordinates



World Coordinates

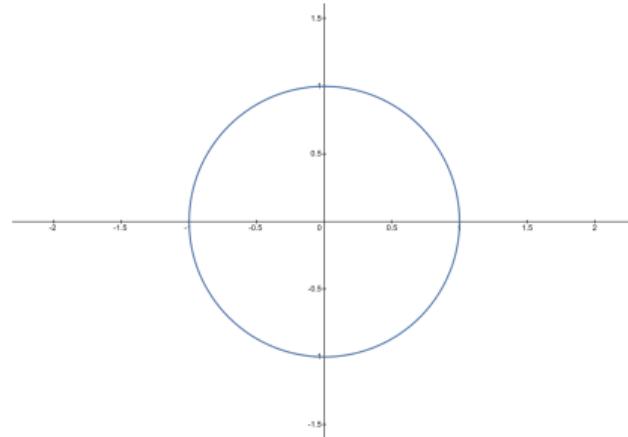
# Función de transformación

```
vec3 getWorldCoordinates(vec2 fragCoord, vec3 iResolution) {  
  
    float aspectRatio = iResolution.x / iResolution.y;  
  
    vec2 scaleFactor =  
        iResolution.x > iResolution.y ? vec2(aspectRatio, 1.0) : vec2(1.0, 1.0 / aspectRatio);  
  
    vec2 world = scaleFactor * (2.0 * (fragCoord/iResolution.xy) - vec2(1.0));  
  
    return vec3(world, 1.0);  
}
```

Después veremos por que esta función, de hecho regresa un vector en  $\mathbb{R}^3$ , cuyo tercer componente es 1.

# Función de distancia con signo

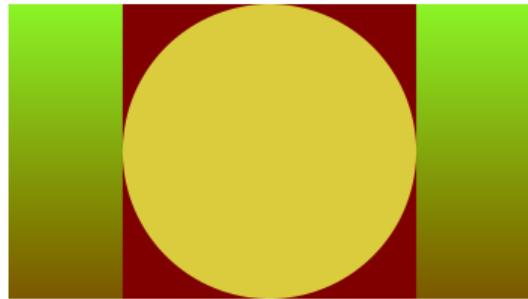
- En Inglés mejor conocida como: *signed distance field* (sdf).
- Hay una [definición formal](#). Pero intuitivamente:
  - Si tienes una curva cerrada  $c$  en  $\mathbb{R}^n$ , cuya frontera es  $\delta$ .
  - Entonces la  $sdf(c)$  es una función continua  $sdf: \mathbb{R}^n \rightarrow \mathbb{R}$ , tal que es positiva en el exterior de  $f$ , negativa en el interior de  $f$  y cero en  $\delta$ .
- Para el circulo  $x^2 + y^2 = 1$
- Una posible sdf es:  $x^2 + y^2 - 1$



# Ejercicio: Un cuadrado detrás de un circulo

<https://github.com/nemediano/tallerShadertoy/tree/main/codigo/Ejercicio2>

- Abstraer el código en funciones
- Tener una función que cambia las coordenadas
- Tener una función para el fondo
- Puedes utilizar la guía de colores



# Transformaciones Afines

Hay una definición formal de **transformación afín**. Pero para nuestros propósitos, podremos decir que: una transformación afín  $A : \mathbb{R}^n \rightarrow \mathbb{R}^n$  es una transformación lineal seguida de una traslación.

Si  $\mathbf{x} \in \mathbb{R}^n$ ,  $M$  es una matriz de  $n \times n$  y  $\mathbf{d} \in \mathbb{R}^n$  un vector. Entonces  $A(\mathbf{x}) = M\mathbf{x} + \mathbf{d}$  es una transformación afín.

- Todas las transformaciones lineales son transformaciones afines (con  $\mathbf{d} = \mathbf{0}$ )
- Todas las traslaciones son transformaciones afines (con  $M = I$ ).

# Coordenadas homogéneas

- Todas las transformaciones lineales pueden llevarse a cabo multiplicando matrices.
- Pero las traslaciones no se pueden llevar a cabo multiplicando matrices
- Para solucionar ese problema usamos las **coordenadas homogéneas**

Vamos a adoptar la convención de que tanto puntos, como vectores son representados en columna.

Las coordenadas homogéneas de un punto  $\mathbf{x} \in \mathbb{R}^n$ , son una tupla de  $n + 1$  componentes, formada por los  $n$  componentes de  $\mathbf{x}$ , seguidos por el escalar 1.

Ésta *no* es la definición general, pero hará las explicaciones más sencillas.

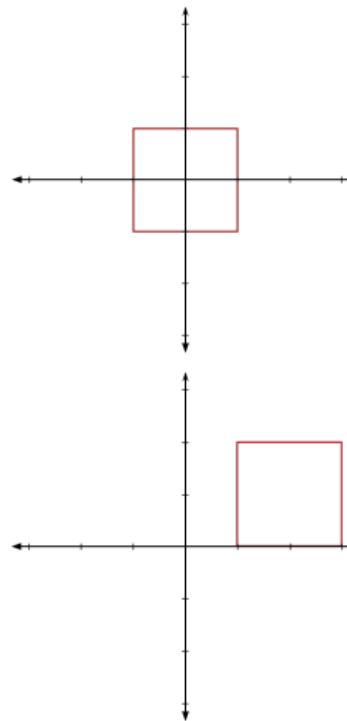
- Las coordenadas homogéneas representan al punto  $\mathbf{x} \in \mathbb{R}^n$  con un punto  $\mathbf{x}_h \in \mathbb{R}^{n+1}$ .
- Pero permiten expresar *las transformaciones afines como una multiplicación de matrices*.

# Traslación

- Traslada una curva en el espacio
- Tiene parámetro el vector de traslación  $t$
- Se puede expresar con la siguiente matriz:

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

- Si  $t = \mathbf{0}$ , hay una operación nula
- La operación inversa es trasladar por  $-t$
- La figura muestra una traslación por  $t = (2, 1)^t$

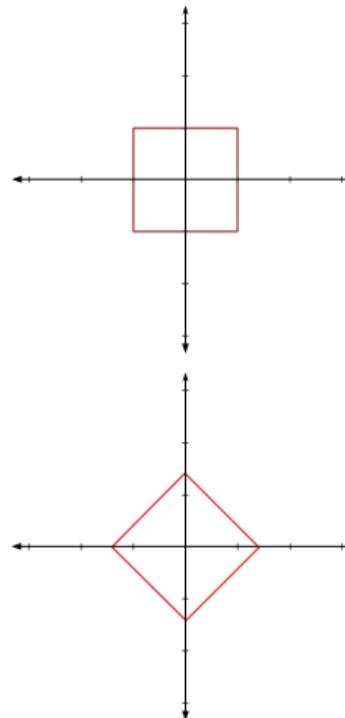


# Rotación

- Rota una curva *alrededor del origen*
- Tiene parámetro el angulo de rotación  $\theta$
- Se puede expresar con la siguiente matriz:

$$\begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \\ 1 \end{bmatrix}$$

- Si  $\theta = 0$ , hay una operación nula
- La operación inversa es rotar por  $-\theta$
- La figura muestra una rotación de  $\theta = \frac{\pi}{4}$

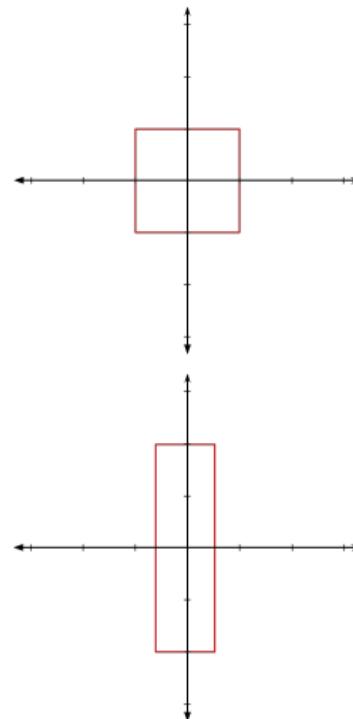


# Escalamiento

- Escala una curva *con respecto al origen*
- Tiene parámetro el factor de escala  $s$
- Se puede expresar con la siguiente matriz:

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x \cdot x \\ s_y \cdot y \\ 1 \end{bmatrix}$$

- Si  $s = (1, 1)^t$ , hay una operación nula
- La operación inversa es escalar por  $s = (\frac{1}{s_x}, \frac{1}{s_y})^t$
- La figura muestra un escalamiento por  $s = (\frac{1}{2}, 2)^t$



# Composición de transformaciones afines

Las transformaciones afines, se pueden aplicar una después de otra.

- Esto se llama **composición de transformaciones**
- Y es particularmente útil que se haga con matrices
- La multiplicación de matrices es asociativa

$$T_1 T_2 T_3 \mathbf{x} = T_1(T_2(T_3 \mathbf{x})) = (T_1 T_2 T_3) \mathbf{x}$$

- La multiplicación de matrices no es conmutativa:

$$T_1 T_2 \mathbf{x} \neq T_2 T_1 \mathbf{x}$$

# Ejemplo

Si  $R$  fuera una rotación de  $\frac{5\pi}{4}$  y  $S$  escalamiento por  $(\frac{1}{2}, 2)^t$

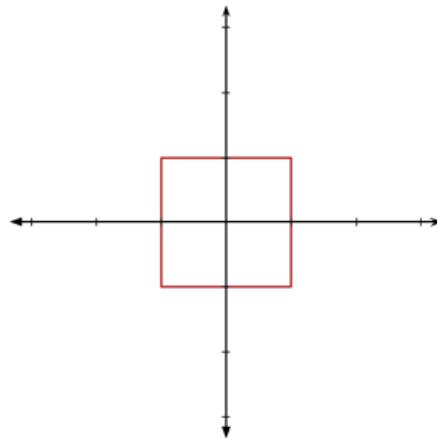
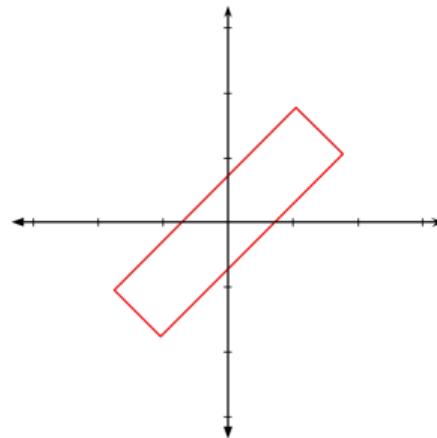
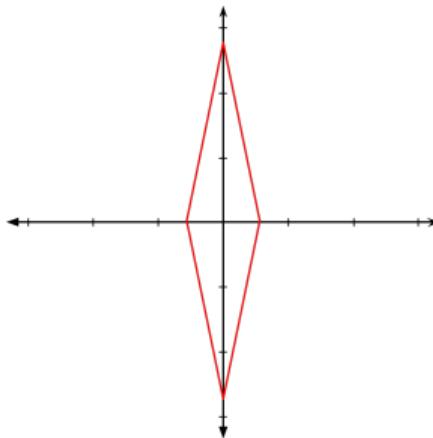


Figura original



$RSx = (R(Sx))$



$SRx = (S(Rx))$

# Transformaciones afines en Shadertoy

- La explicación anterior es como se hacen las gráficas tradicionalmente.
- Es decir, transformamos la geométrica para acomodarla en el espacio.

## En Shadertoy en 2D hacemos lo contrario

- En shadertoy en 2D, transformamos las coordenadas del fragmento hacia la figura.
- Esto es equivalente a decir que transformamos el espacio hacia la figura.
- Por lo tanto, para situar figuras en 2D usamos la matriz inversa de la matriz de transformación.

# Ejemplo en Shadertoy 2D

---

```
mat3 M = translate(vec2(0.75, 0.0)) * scale(vec2(0.25));  
  
float disCircle = sdfCircle(inverse(M) * coord);  
  
color = mix(Red, color, step(0.0, disCircle));
```

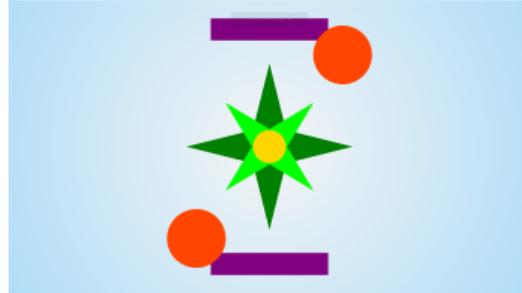
---

- En este ejemplo se escala y luego se traslada una figura (circulo)
- Nótese la inversión de la matriz, antes de evaluar las sdf.
- Y el uso de la función step dentro de la función mix, para cambiar el color del fragment si estaba dentro de la figura.

# Ejercicio: Figuras animadas

<https://github.com/nemediano/tallerShadertoy/tree/main/codigo/Ejercicio3>

- Usar transformaciones compuestas para crear una escena
- Puedes utilizar la [esta lista](#) para ver mas figuras
- Puedes comenzar con el código del ejercicio anterior



# Algoritmos basados en rayos

En realidad esta es una familia de algoritmos, los más importantes son:

- **Ray Casting:** El primero en ser desarrollado. Consiste en lanzar un rayo por cada pixel del viewport. Para ver donde interfecta la escena.
- **Ray Tracing:** Una generalización donde cada vez que se intersecta la escena, se calcula el ángulo de reflexión y se continua siguiendo el rayo, hasta cumplir algún criterio de paro.
- **Ray Marching:** Una forma de usar la sdf, para acelerar el camino de los rayos de manera recursiva. Este es el más usado en ShaderToy
- **Path Tracing:** En vez de lanzar un rayo, se lanzan varios rayos a ángulos muy similares usando el método de Monte Carlo.

# El rayo

## Rayo

Un rayo es el semisegmento de linea  $\mathbf{r}(s) = \mathbf{o} + s\mathbf{d}$ , donde  $s \in \mathbb{R}^+$  es un escalar positivo,  $\mathbf{o}$  es un punto en  $\mathbb{R}^3$  y  $|\mathbf{d}| = 1$  un vector unitario en  $\mathbb{R}^3$ .

- $\mathbf{o}$  es el origen del rayo.
- $\mathbf{d}$  es la dirección del rayo.
- $s$  es el parámetro.



# Ray Marching

Es una técnica para hacer un avance adaptativo de los rayos.

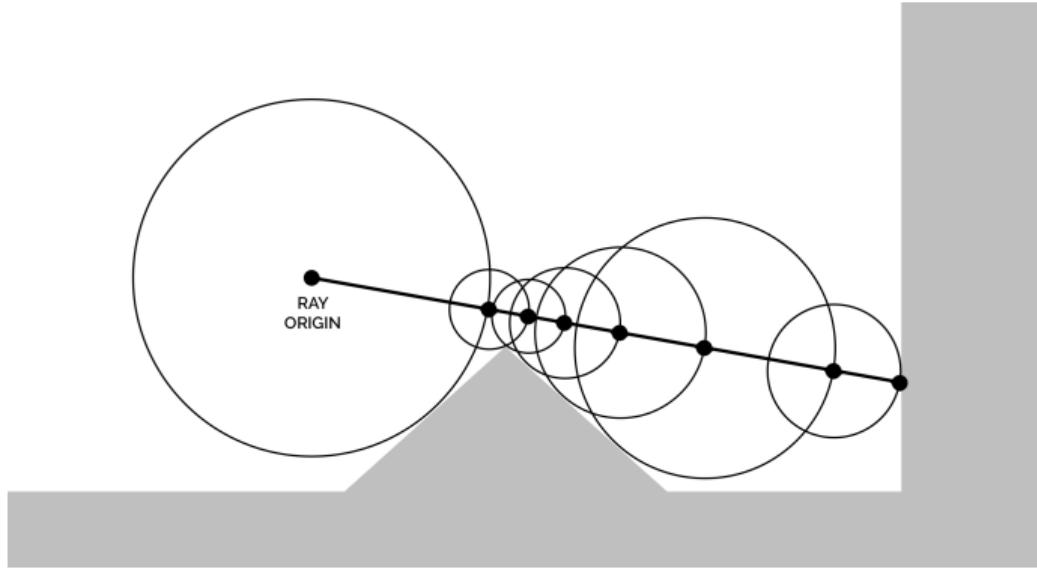
- ① En el punto actual del rayo calcula la distancia  $d$  del punto con respecto a la escena. La *distancia* de la escena, es la mínima de todas las sdfs a las figuras en la escena.
- ② Si  $d$  es menor que un cierto  $\epsilon$ , termina y reporta que hay una colisión con la escena.
- ③ Si  $d$  es mayor que un cierto valor frontera termina y di que no hay colisión.
- ④ Avanza el rayo en  $d$  unidades, y repite desde 1.

Esta técnica solo se puede usar si...

- Definimos la escena por medio de sdfs.
- El rayo tiene un vector de dirección unitario.
- Las sdf están bien definidas:
  - Continuas
  - La distancia está en la misma dimensión que la escena.

# Ray Marching: una imagen

- En la práctica se pone un número máximo de pasos como criterio extra de paro.



Hay un [ejemplo](#) para visualizar la técnica en ShaderToy mismo.

# Ray Marching: en código

---

```
float rayMarch(Ray ray, float start, float end) {
    float depth = start;

    for (int i = 0; i < MAX_MARCHING_STEPS; i++) {
        vec3 position = ray.origin + depth * ray.direction;
        float d = sdScene(position);
        depth += d;
        if (d < PRECISION || depth > end) break;
    }

    return depth;
}
```

---

## Visualizar en 3D

- Para no ver todo de un color plano en 3D, necesitamos hacer shading.
- Para hacer shading, necesitamos (entre otras cosas), la normal  $\mathbf{n}$  a la superficie en el punto de contacto.
- Podemos utilizar la misma sdf, para calcular el gradiente  $\nabla$  de la figura. El gradiente es una buena aproximación a la normal:  $\nabla \approx \mathbf{n}$ .

$$\nabla(f(\mathbf{x})) = \begin{pmatrix} f(x + \epsilon, y, z) - f(x - \epsilon, y, z) \\ f(x, y + \epsilon, z) - f(x, y - \epsilon, z) \\ f(x, y, z + \epsilon) - f(x, y, z - \epsilon) \end{pmatrix}$$

Donde  $f(\mathbf{x})$  es una sdf y  $\epsilon$  un numero positivo muy pequeño.

- Finalmente, si definimos una fuente de luz, con una posición  $\mathbf{l}$  en la escena.
- Una forma muy sencilla de shading puede ser:  $c = \max(\mathbf{k}_d(\mathbf{l} \cdot \mathbf{n}), 0)$ .
- Donde  $\mathbf{c}$  es el color resultante y  $\mathbf{k}_s$  es el color predominante de la superficie.

# Ejercicio: Esfera usando Ray Marching

<https://github.com/nemediano/tallerShadertoy/tree/main/codigo/Ejercicio4>

- De momento usa solo una esfera en la escena
- Trata de definir tu cámara, la esfera y una fuente de luz, en lugares sencillos respecto al origen.
- De momento usa la reflexión difusa, para que puedas visualizar al esfera
- Puedes usar el código de comienzo de la escena para empezar.

