

Shaders de Juguete (Shadertoy)

Usando trucos matemáticos para dibujar en tu navegador web

Dr. Jorge Antonio García Galicia

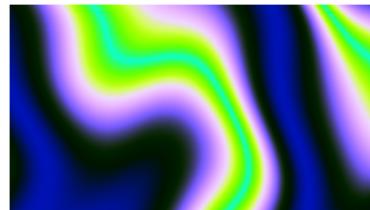
nemediano.github.io

Congreso de Inteligencia Artificial, Abril 2025

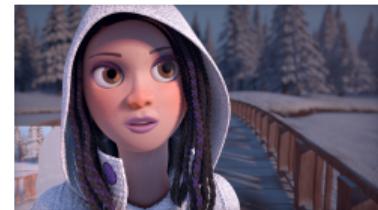


¿Qué esperar del taller?

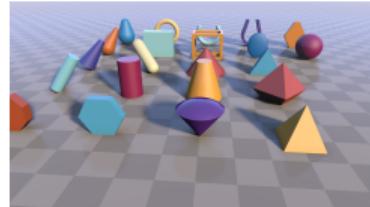
- Aprender el paradigma de [Rendering Worlds With Two Triangles](#)
- Parte del [creative coding](#): intersección entre el arte y la ciencia.
- Un poco de [GLSL](#), y las matemáticas básicas para hacer un [fragment shader](#) en el paradigma.



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

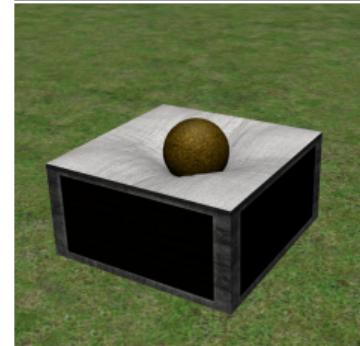
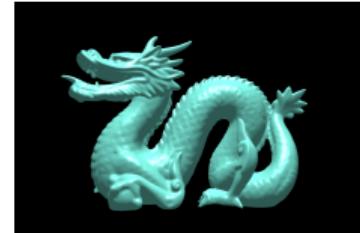
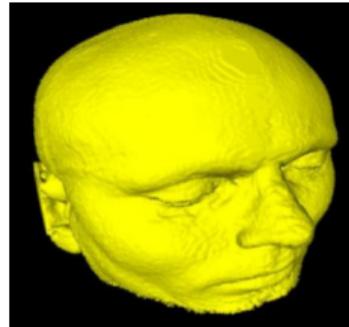


Dr. Jorge Antonio García Galicia

- 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.

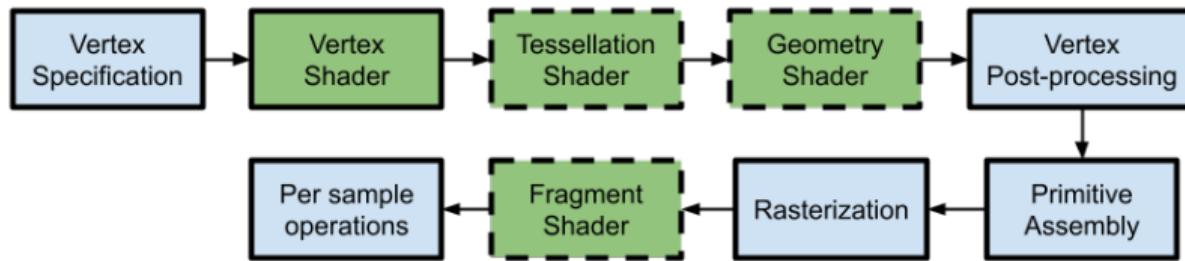
Acerca del nombre:

- 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

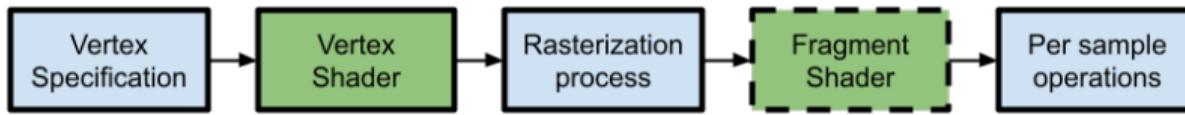
Pipeline gráfico

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



Simplificando...

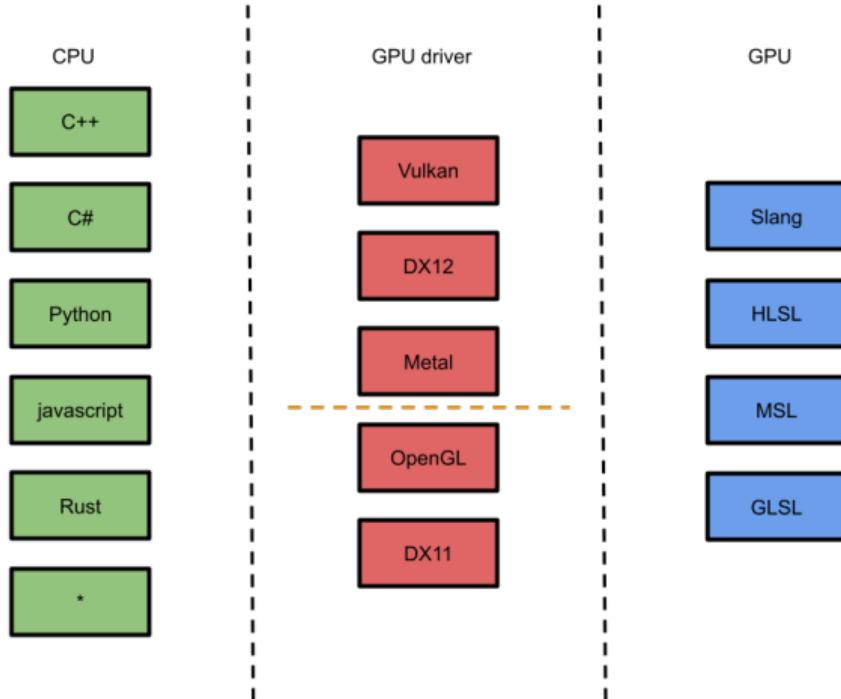
- En la práctica, los tessellation shaders y los geometry shaders solo se usan para cosas muy específicas.
- Mas aún, casi nunca se configuran los estados entre el vertex shader y la rasterización
- Para propósitos de esta explicación, podemos pensarlo de manera mas simple:



¿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

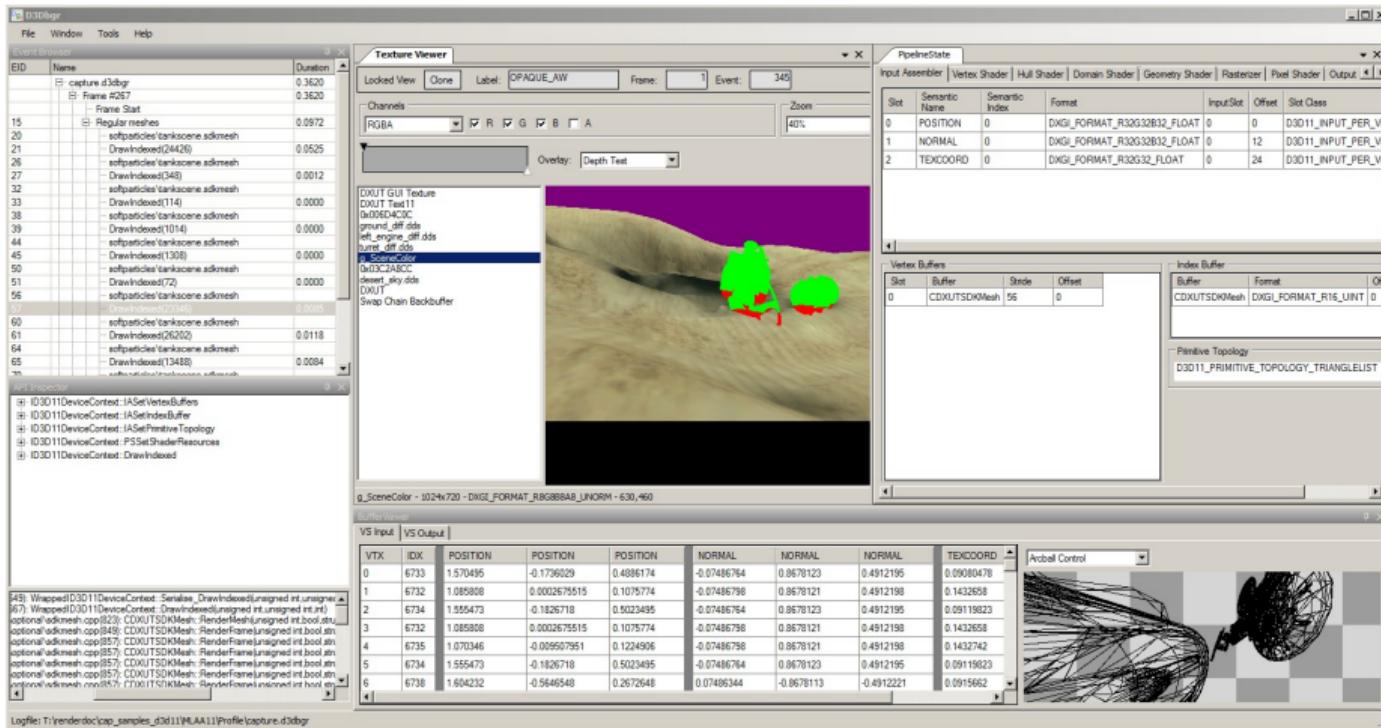


Anatomía de una aplicación I

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:

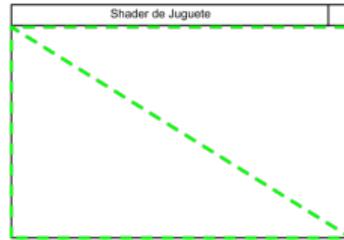
- Unificar en el modelo de iluminación PBR
- Las técnicas basadas en rayos: Raytracer, Raymarching, Pathtracers
 - Generar parte de los fragmentos de un frame, usando técnicas de AI en vez de calcularlos
- Definir un nuevo pipeline: Task (Amplification) shaders, mesh shaders
- Descargar la mayor parte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



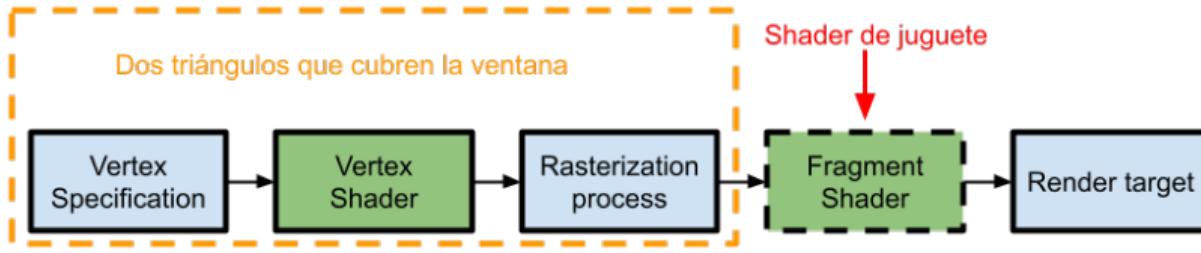
Una idea ingeniosa

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.

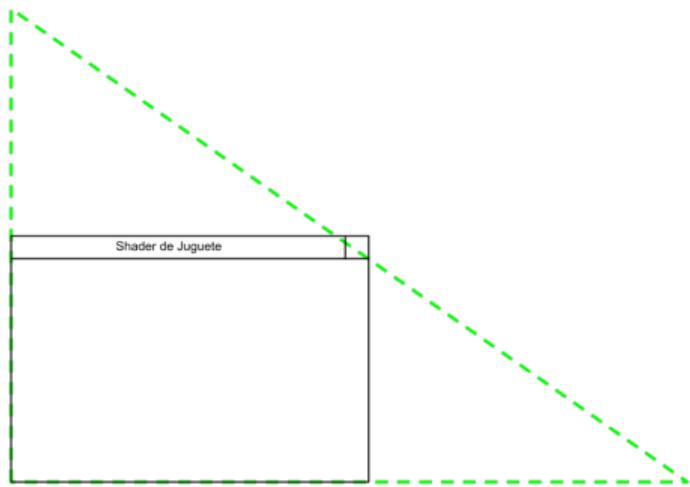


Regresar al principio



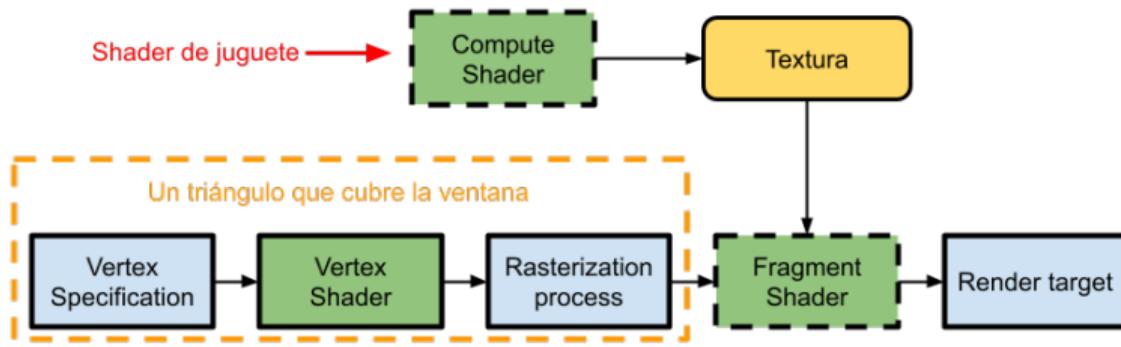
- Es similar a como se hacían los gráficos antes de que tuviéramos tarjetas de video.
- Solo que ahora aprovechamos el **inmenso poder paralelo** del GPU
- Y usamos GLSL

Nota curiosa



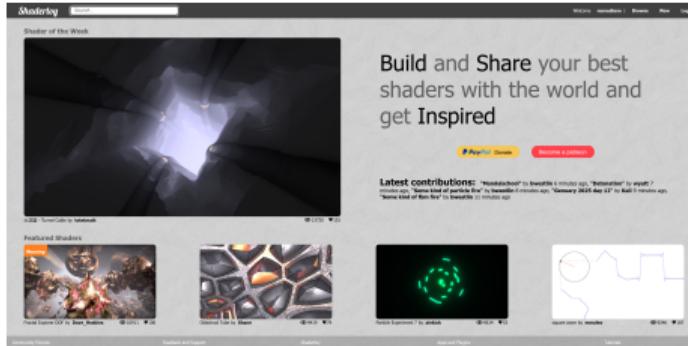
De hecho podemos hacerlo con un solo triángulo y luego hacemos clipping.

Actualmente, es mejor hacer:



Shadertoy

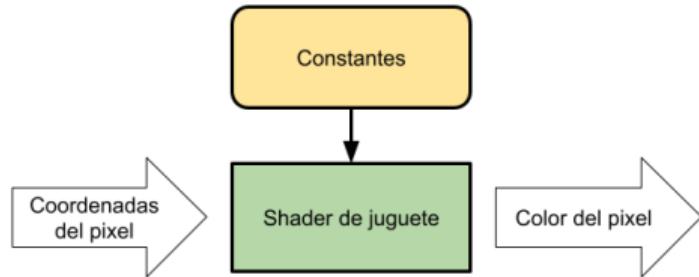
- Shadertoy es un sitio web: <https://www.shadertoy.com/> que nos da la infraestructura para usar el truco
- Y ciertas herramientas sociales
- Fue creado por [Iñigo Quilez](#) y [Pol Jeremias](#) en el 2013



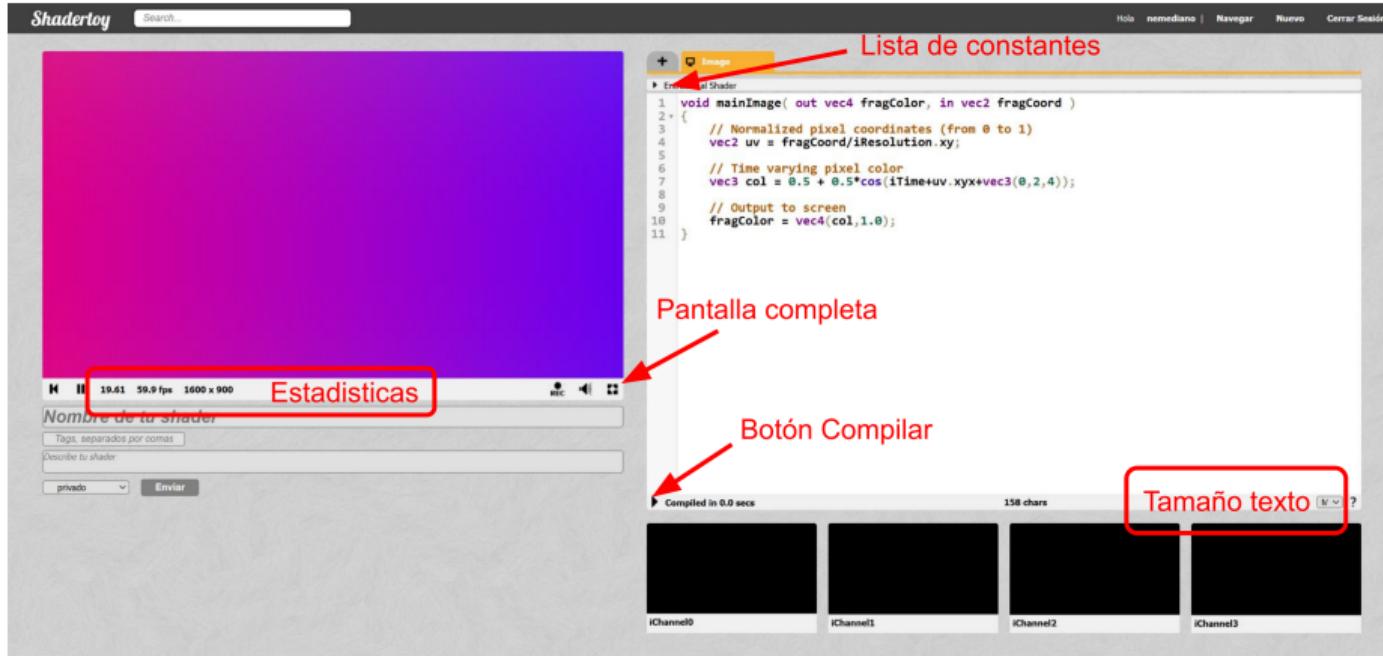
¿Cómo se usa?

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

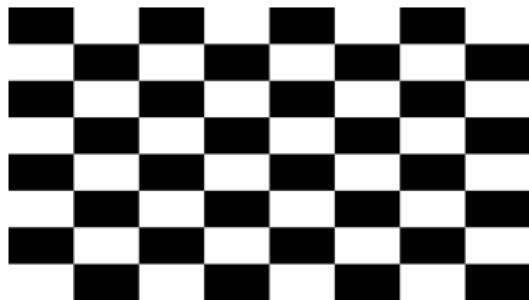
En shadertoy, la ejecución inicia y termina con la función:

```
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.

Bandera a cuadros

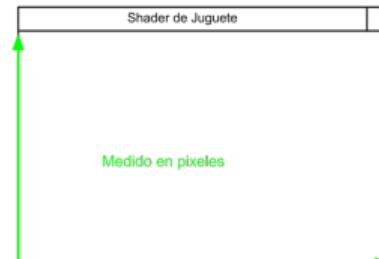
- Tratar de escribir un shader que genere una bandera a cuadros (o tablero de ajedrez si lo prefieres)
- Puedes empezar con el código de default de shader toy.
- Recuerda las funciones trigonométricas



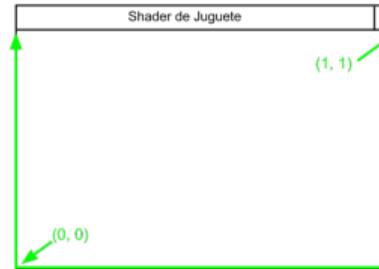
<https://github.com/nemediano>

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*). $u, v \in [0, 1]$



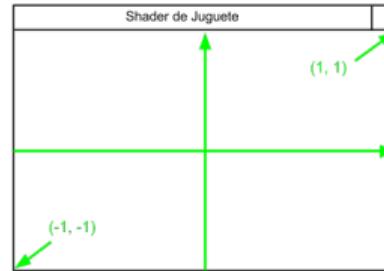
Espacio de imagen



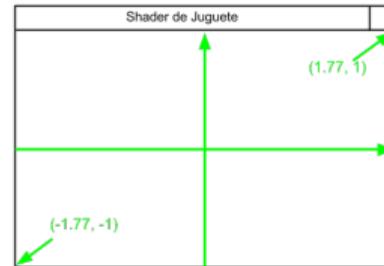
uv space

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

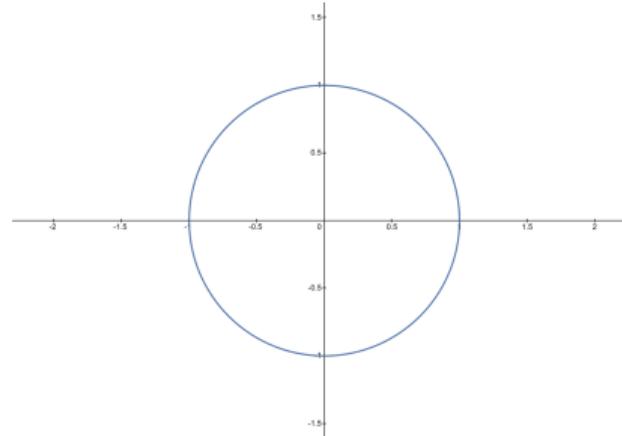
Función de transformación

```
vec3 getWorldCoordinates(vec2 fragCoord) {  
    // UV coordinates  
    vec2 uv = fragCoord/iResolution.xy;  
    // Normalized coordinates  
    vec2 norm = 2.0 * (uv - vec2(0.5));  
    // World coordinates  
    float aspectRatio = iResolution.x / iResolution.y;  
    vec2 world = norm * vec2(aspectRatio, 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 δ .
 - 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 δ .
- Para el circulo $x^2 + y^2 = 1$
- Una posible sdf es: $x^2 + y^2 - 1$



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

Rotación

Escalamiento

Composición de transformaciones

Operadores booleanos

Constructive solid geometry (CSG)

Shaders de juguete con formas complejas
Dibujar un Gato (u alguna otra figura)

Raymarching

Proyección en perspectiva

Raycaster

Modelos de iluminación

Phong

Cook-Torrance

PBR

¿Cómo seguir aprendiendo?

Página de Inigo quilez
Shader Book