

Informática Gráfica.

Sesión 1: Introducción.

Carlos Ureña, Sept 2025.
Dept. Lenguajes y Sistemas Informáticos.
Universidad de Granada.

Índice

Datos de la asignatura.....	3
Aplicaciones gráficas interactivas y visualización	26
APIs y motores gráficos.....	61

Sección 1.

Datos de la asignatura.

1. La materia.
2. Objetivos, programa, temario.
3. Horarios, datos de contacto, tutorías.
4. Evaluación.
5. Bibliografía y recursos *online*.

Subsección 1.1.

La materia.

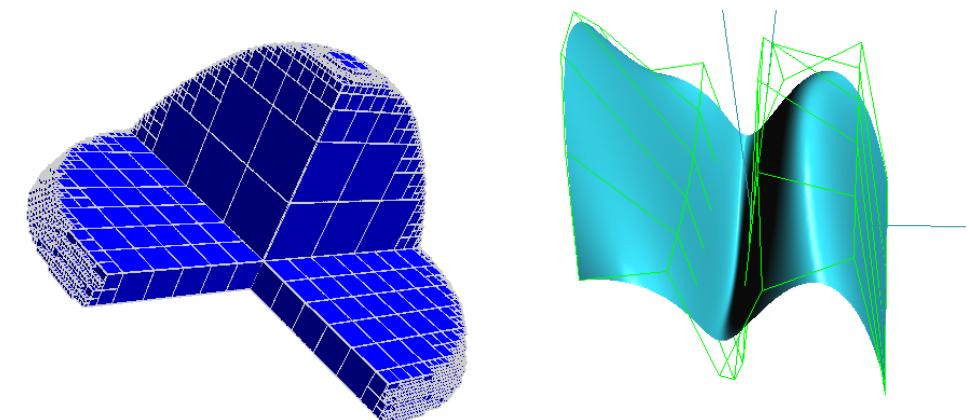
Informática Gráfica

La Informática Gráfica es la parte de la Informática que se ocupa del procesamiento de información geométrica y visual. Algunos de los campos más relevantes son:

- La representación de información: **modelos geométricos**.
- La generación de imágenes: **visualización (rendering)**.
- La entrada de información: **interacción y adquisición** de modelos.
- La computación geométrica: **operaciones y cálculos** sobre los modelos.

Modelos geométricos.

Diseño de modelos abstractos de objetos reales, y de las estructuras de datos que se usan para representarlos en la memoria de un ordenador. Creación de los modelos.



Aplicaciones: Videojuegos, realidad virtual, simuladores

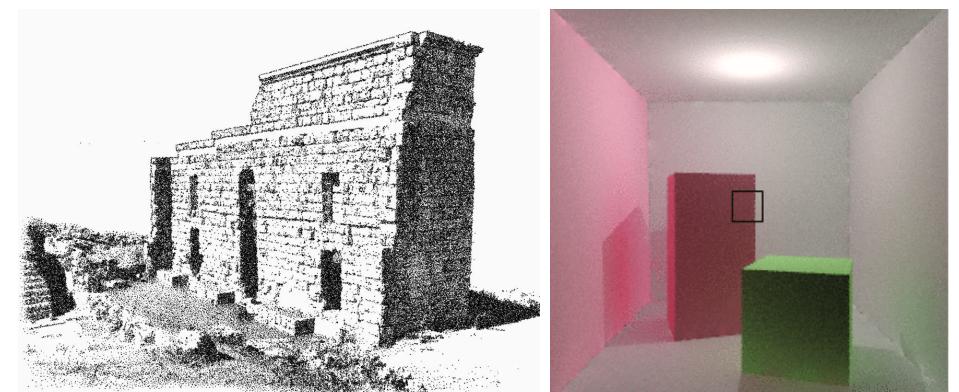


Simulador de Conducción con Editor de Entornos

(proyecto fin de carrera de Valerio M. Sevilla, tutor: Carlos Ureña)

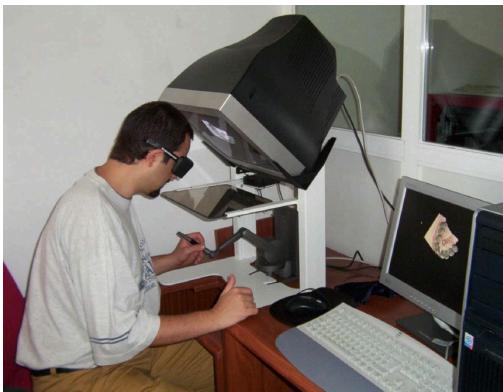
Aplicaciones: visualización (rendering)

Producción de imágenes a partir de modelos geométricos en memoria, no necesariamente de forma interactiva (para cine, anuncios y efectos especiales en general)



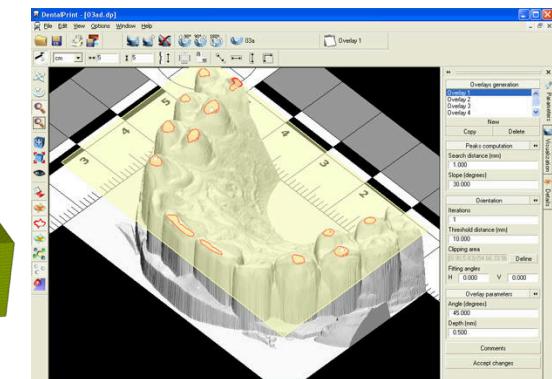
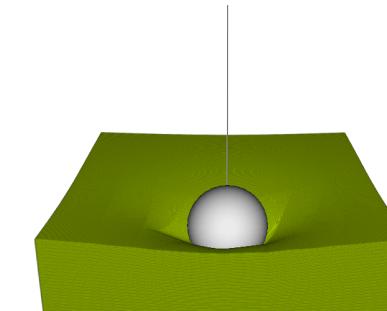
Aplicaciones: interacción y captura de modelos.

Adquisición de nuevos modelos a partir de objetos reales.



Aplicaciones: computación geométrica

Algoritmos y metodologías para procesamiento y edición de los modelos.



Objetivos de la asignatura

- Conocer los fundamentos del modelado geométrico
- Saber diseñar y utilizar las estructuras de datos más adecuadas para representar un modelo geométrico
- Saber diseñar modelos jerárquicos
- Saber utilizar y representar transformaciones geométricas utilizando coordenadas homogéneas
- Conocer los fundamentos de la visualización 2D y 3D
- Conocer los fundamentos de los modelos de iluminación
- Entender y poder configurar los parámetros de materiales y luces
- Conocer la funcionalidad básica del engine Godot
- Saber diseñar un programa interactivo con eventos, garantizando la accesibilidad y la usabilidad.
- Saber diseñar e implementar programas gráficos interactivos usando Godot
- Conocer los fundamentos de la animación por ordenador

Programa de teoría

El programa de teoría para el curso 2025-26 es el siguiente:

- 1. Introducción:** Introducción a la informática Gráfica. Godot.
- 2. Modelado de objetos:** Fundamentos de modelado. Representación de mallas poligonales. Transformaciones. Instanciación. Modelos jerárquicos. Cálculo de normales. Formatos.
- 3. Visualización:** Cámara y su configuración en Godot. Modelo de iluminación local. Iluminación en Godot. Sombreado para Z-buffer. Visualización de texturas. Texturas en Godot.
- 4. Animación e Interacción:** Sistemas interactivos. Gestión de eventos. Posicionamiento. Selección. Animación. Colisiones.
- 5. Aspectos avanzados de visualización:** Ray-tracing.

Programa de prácticas

El programa de prácticas para el curso 2025-26 es el siguiente:

1. Introducción. Modelado y visualización de objetos 3D sencillos
2. Modelos poligonales: carga de PLYs y generación por revolución.
3. Modelos jerárquicos: creación de un objeto jerárquico.
4. Modelo de aspecto: materiales, fuentes de luz y texturas.
5. Interacción: gestión de cámara y selección.

El código de las prácticas se desarrollará de forma incremental, cada práctica se hace sobre las anteriores.

Horarios, datos de contacto, tutorías

Clases	Teoría: Martes 9:30-11:30 (aula 1.4) Prácticas: Lunes o Martes 11:30-13-30 (aula 2.1)
Profesor	Carlos Ureña Almagro
Despacho	ETSIIT, planta 3, pasillo izquierdo, desp. 34.
Teléfono	(+34) 958 240 577
E-mail	curena@ugr.es / curena@go.ugr.es
Web	https://www.ugr.es/~curena
Tutorías	Lunes 9:30 - 11:30 — Miércoles 9:30 - 13:30
Info y guía UGR	GIM , GIADE .

Evaluación: pruebas de evaluación.

Subsección 1.4. Evaluación.

En la convocatoria ordinaria, en evaluación continua se harán las siguientes pruebas:

(E1) Examen de teoría: escrito, en la fecha establecida por el centro, con un peso de 50// en la nota final.

(E2) Examen de prácticas: se establecerán una o varias fechas para la entrega y examen de prácticas, el examen consistirá en resolver problemas de programación usando el código entregado. Suponen el 50// en la nota final (10// por práctica).

En **convocatoria extraordinaria** y en **evaluación única final** se seguirán los mismos criterios excepto que las pruebas de prácticas (E2) se realizarán en una fecha única dentro del período de exámenes correspondiente.

Evaluación: calificación.

- Se podrá sumar hasta un 10% de la nota máxima por trabajos adicionales, re-realización de ejercicios o presentaciones, mejora de las prácticas, etc., siempre que se haga de forma previamente acordada con el profesor y siempre que se supere la asignatura con el resto de items evaluables (E1 y E2).
- Para aprobar la asignatura hay que obtener igual o más del 50% del total, e igual o más del 40% en cada parte (E1 y E2).
- Si un alumno no supera la asignatura en la convocatoria ordinaria del curso 25-26, pero tiene una nota igual o superior al 50% en algunas de las partes (E1 o E2), entonces podrá conservar esa nota para la convocatoria extraordinaria de 25-26.

Subsección 1.5.

Bibliografía y recursos online.

Bibliografía: Conceptos de Informática Gráfica (1/2)

J.F. Hughes, A.van Dam, M. McGuire, D.F. Sklar, J.D. Foley, S.K. Feiner, K. Akeley.
Computer Graphics: Principles and Practice (3ed ed.)
Ed. Pearson, 2014. ISBN: 978-0-321-39952-6.
Web autores: <http://cgpp.net>
Web editorial: <https://www.pearson.com/.....>

Steve Marschner, Peter Shirley.
Fundamentals of computer graphics. (5th ed.)
Ed. A.K. Peters / CRC Press, 2021. ISBN: 978-1032122861
Web autores: <https://www.cs.cornell.edu/~srm/fcg5/>
Web editorial (DOI): <https://doi.org/10.1201/9781003050339>

Bibliografía: Matemáticas para gráficos

Eric Lengyel.
Mathematics for 3D Game Programming and Computer Graphics (3rd ed.).
Ed. Cengage Learning, 2011. ISBN: 978-1-4354-5886-4
Web autores: <http://mathfor3dgameprogramming.com/>

Michael E. Mortenson.
Mathematics for Computer Graphics Applications (2nd ed.).
Ed. Industrial Press, 1999. ISBN: 0-8311-3111-X.
Web editorial:
<https://books.industrialpress.com/9780831131111/mathematics-for-computer-graphics-applications/>

Bibliografía: Conceptos de Informática Gráfica (2/2)

Steven J. Gortler.
Foundations of 3D Computer Graphics (1st ed.).
Ed. The MIT Press, 2012. ISBN: 9780262017350.
Web autores: <http://www.3dgraphicsfoundations.com/>
Web editorial: <https://mitpress.mit.edu/.....>

Tomas Akenine-Möller, Eric Haines, Naty Hoffman.
Real-Time Rendering (4th ed.)
Ed. CRC Press, 2018. ISBN: 978-1138627000
Web autores (recursos): <https://www.realtimerendering.com>

Bibliografía: Godot / GDScript.

Ivan Korotkin.
Godot 4: Introduction to GDScript. Autoeditado (2023). ISBN: 979-8394581557
Amazon:
https://www.amazon.com/Godot-Introduction-GDScript-practical-tutorials/dp/B0C5241244#detailBullets_feature_div

Sander Vanhove.
Learning GDScript by Developing a Game with Godot 4.
Ed. Packt Publishing 2024. ISBN: 9781801812498.
Web editorial:
<https://www.packtpub.com/en-us/product/learning-gdscript-by-developing-a-game-with-godot-4-9781801812498>

Recursos online: Godot / GDScript

Enlaces a las versiones en español, traducidas (algunas parcialmente) a partir de los originales en inglés:

Web con la documentación oficial de Godot 4:

<https://docs.godotengine.org/es/4.x/>

Introducción a Godot:

https://docs.godotengine.org/es/stable/getting_started/introduction/index.html

Guía para crear un juego 3D con Godot:

https://docs.godotengine.org/es/4.3/getting_started/first_3d_game/index.html

Referencia de las clases de Godot:

<https://docs.godotengine.org/es/4.x/classes/index.html>

Resumen de la sección

En esta sección se incluye:

- Una introducción a las aplicaciones gráficas interactivas,
- Los dos métodos básicos de visualización (Rasterización y por Ray-tracing).
- Se da una visión muy general sobre el procesamiento que se hace en el cauce gráfico.

Sección 2.

Aplicaciones gráficas interactivas y visualización

1. Aplicaciones gráficas interactivas
2. El proceso de visualización 2D y 3D
3. *Rasterización versus ray-tracing.*
4. El cauce gráfico en rasterización

Subsección 2.1.

Aplicaciones gráficas interactivas

Aplicaciones gráficas

Un **programa gráfico** es un programa (o parte de un programa o sistema) que

- Almacena una estructura de datos que constituye un **modelo** computacional de determinada información.
- Produce una salida constituida (principalmente) por una o varias imágenes.
- Las imágenes típicas son **imágenes raster**, constituidas por un array de pixels discretos, cada uno con un color RGB.
- Existen otros tipos de salidas gráficas, la más frecuentes son las **imágenes vectoriales** (p.ej.: archivos **.svg**).
- Los programas gráficos pueden ser: **interactivos** o **no interactivos**

Aplicaciones gráficas interactivas (1/2)

Un programa gráfico **interactivo** es un programa que:

- **Visualiza** en una ventana gráfica una imagen que constituye una representación visual del modelo.
- Procesa acciones del usuario (llamadas **eventos**), que se traducen en modificaciones del modelo.
- Cada vez que el modelo es modificado, se vuelve a visualizar, de forma **interactiva**, lo que significa que desde que el usuario produce el evento hasta que puede observar la imagen actualizada pasan tiempos del orden de decenas de milisegundos como mucho.

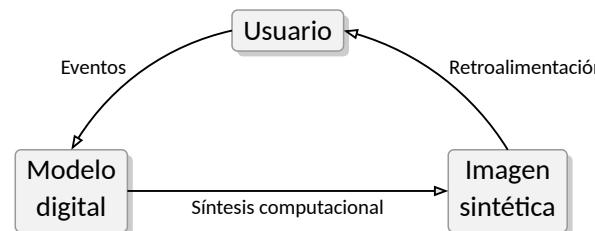
Este esquema es el que se usa típicamente en aplicaciones de simuladores, diseño asistido por computador, videojuegos, realidad virtual y realidad aumentada.

Aplicaciones gráficas interactivas (2/2)

Los elementos esenciales de una aplicación gráfica interactiva son:

- **Modelos digitales** de objetos reales, ficticios o de datos
- **Imágenes o videos digitales** que se usan para visualizar dichos objetos.

En las aplicaciones interactivas, los usuarios modifican los modelos (con eventos) y reciben retroalimentación inmediata:



Eventos de entrada

Los **eventos de entrada** son acciones del usuario que permiten enviar información a la aplicación, información que se usa para modificar el modelo o los parámetros de visualización. Ejemplos:

- Pulsar o levantar una tecla del teclado.
- Pulsar o levantar un botón del ratón.
- Mover el ratón.
- Mover la rueda del ratón.
- Cambiar el tamaño de una ventana de la aplicación.
- Cerrar una ventana de la aplicación.

También se pueden considerar eventos otros tipos de sucesos, no directamente iniciados por el usuario, como la llegada de datos por la red, la finalización de una tarea de cálculo, o el transcurso de un período de tiempo (para animaciones).

El bucle principal de gestión de eventos

Las aplicaciones gráficas interactivas ejecutan (explícitamente o implícitamente) un bucle, el llamado **bucle principal de gestión de eventos**. Es un bucle que se ejecuta continuamente, y que en cada iteración da estos pasos:

1. Espera a que ocurra un evento y entonces recuperar datos del mismo.
2. Procesar el evento: típicamente supone actualizar:
 - el modelo y/o
 - los parámetros de visualización.
3. Visualizar el modelo actualizado con los nuevos parámetros.

Subsección 2.2.

El proceso de visualización 2D y 3D

Visualización 2D y 3D (1/2)

En general, las aplicaciones gráficas pueden, en general, dividirse en dos tipos:

- **Aplicaciones 2D:** usan modelos de objetos visibles o dibujables que se definen en un plano (espacio bidimensional), o en varios planos (unos con más *prioridad* que otros, es decir, por delante de otros).
 - ▶ Pueden incluir también algunos elementos 3D, como sombras.
 - ▶ Ejemplos: aplicaciones de visualización de datos, edición de imágenes, diseño gráfico 2D.

Visualización 2D y 3D (2/2)

En general, las aplicaciones gráficas pueden dividirse en dos tipos:

- **Aplicaciones 3D:** se usan modelos de objetos visibles o dibujables que se definen en el espacio tridimensional. Dichos modelos incluyen información de aspecto como texturas, materiales, fuentes de luz, etc...que son propias de la visualización 3D.
 - ▶ Pueden incluir también elementos 2D, como texto, iconos, etc...
 - ▶ Ejemplos: videojuegos, simuladores, realidad virtual, realidad aumentada.

Visualización 2D. Entradas.

El proceso de visualización 2D produce una imagen a partir de un **modelo** y unos **parámetros**:

- **Modelo:** estructura de datos en memoria que representa los elementos que se van a visualizar en la imagen, suele incluir puntos, líneas, polígonos, textos, imágenes, gradaciones, etc...Estos elementos se construyen a partir de los datos de entrada en función del estilo de visualización que se desee.
- **Parámetros:** diversos valores que afectan a la visualización: la resolución de la imagen y su posición en pantalla (viewport), la zona de coordenadas de mundo que se quiere visualizar (un rectángulo del espacio de coordenadas de mundo), el rango de valores en Z, etc....

El proceso de visualización 3D: entradas (1/2)

El proceso de visualización 3D produce una imagen a partir de un **modelo de escena** y unos **parámetros**:

- **Modelo de escena:** estructura de datos en memoria que representa lo que se quiere ver, esta formado por varias partes:
 - ▶ **Modelo geométrico:** conjunto de **primitivas** (típicamente polígonos planos), que definen la forma de los objetos a visualizar
 - ▶ **Modelo de aspecto:** conjunto de parámetros que definen el aspecto de los objetos: tipo de material, color, texturas, fuentes de luz

Visualización 2D. Visualización de datos.

Las imágenes ayudan a entender la información en tablas numéricas, árboles, DAGs, grafos arbitrarios, relaciones, etc...

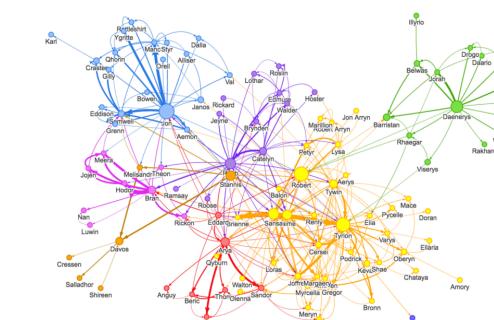


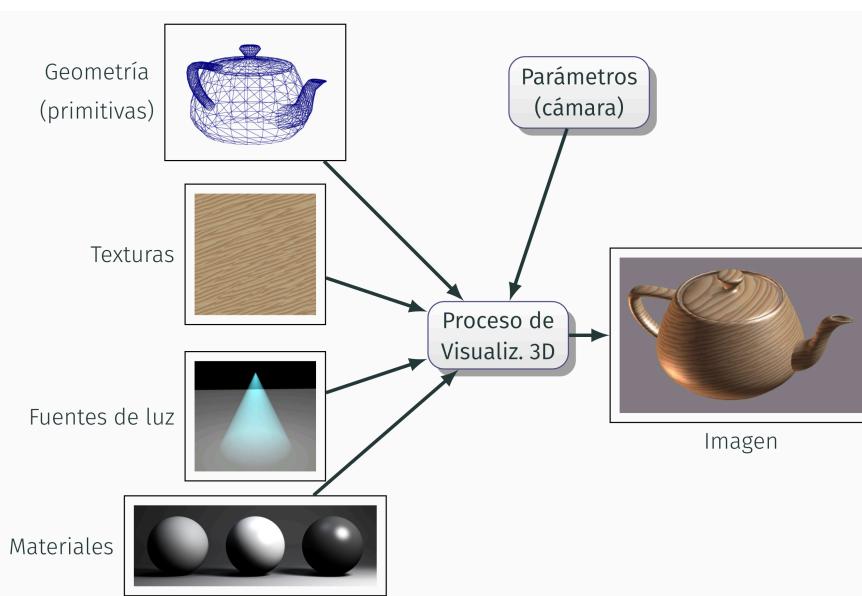
Imagen de *Hands on graph data visualization* (Relaciones entre personajes de Game of Thrones)
<https://neo4j.com/developer-blog/hands-on-graph-data-visualization/>

El proceso de visualización 3D: entradas (2/2)

El proceso de visualización produce una imagen a partir de un **modelo de escena** y unos **parámetros**:

- **Parámetros de visualización:** es un conjunto amplio de valores que determinan como se visualiza la escena en la imagen, algunos elementos esenciales son:
 - ▶ **Cámara virtual:** posición, orientación y ángulo de visión del observador ficticio que vería la escena como aparece en la imagen
 - ▶ **Viewport:** Resolución de la imagen, y, si procede, posición de la misma en la ventana.

El proceso de visualización 3D: esquema



Visualización basada en rasterización (1/2)

En este curso nos centramos en los algoritmos relacionados con la visualización basada en **rasterización** (*rasterization*). En ese algoritmo, se produce como salida una imagen I a partir de un conjunto de *primitivas* de entrada E . En pseudocódigo:

```
1: Inicializar el color de todos los pixels al color de fondo.
2: for cada primitiva  $P$  del conjunto  $E$  do
3:    $S \leftarrow$  conjunto de pixels de la imagen  $I$  cubiertos por  $P$ 
4:   for cada pixel  $q$  de  $S$  do
5:      $c \leftarrow$  color de la primitiva  $P$  en el pixel  $q$ 
6:     Asignar el color  $c$  al pixel  $q$  en  $I$ 
7:   end
8: end
```

Subsección 2.3. Rasterización versus ray-tracing.

Visualización basada en rasterización (2/2)

En el código anterior:

- Las **primitivas** son los elementos más pequeños que pueden ser visualizados (típicamente triángulos en 3D, aunque también pueden ser otros: polígonos, puntos, segmentos de recta, círculos, etc...)
- El número de pixels en S es proporcional al número p de pixels en I
- El tiempo de cálculo total es proporcional al número de iteraciones del bucle interno, ese número es a su vez proporcional al número de primitivas (n) por el número de pixels (p).
- Es decir, decimos que el tiempo de cálculo está en el orden de $p \cdot n$, o lo que es lo mismo: el tiempo esta en $O(p \cdot n)$.

Si se considera p una constante, el tiempo está en $O(n)$.

Visualización basada en ray-tracing (1/2)

En la técnica de **Ray-Tracing**, los dos bucles de antes se intercambian:

```
1: Inicializar el color de todos los pixels
2: for cada pixel  $q$  de la imagen  $I$  a producir do
3:    $T \leftarrow$  subconjunto de primitivas de  $E$  que cubren  $q$ 
4:   for cada primitiva  $P$  del conjunto  $T$  do
5:      $c \leftarrow$  color de la primitiva  $P$  en el pixel  $q$ 
6:     Asignar color  $c$  al pixel  $q$  en  $I$ 
7:   end
8: end
```

- Cuando se trata de visualización 3D, la implementación de esta esquema se conoce como **algoritmo de Ray-tracing**.
- Al igual que en rasterización, el tiempo de cálculo total es proporcional al número de de primitivas (n) por el número de pixels (p), es decir, la complejidad en tiempo está también en $O(n \cdot p)$.

Rasterización

Respecto a la técnica de *rasterización*:

- Las **unidades de procesamiento gráfico** (GPUs) son un hardware diseñado originalmente para ejecutar la rasterización de forma eficiente en tiempo.
- El método de rasterización es preferible para **aplicaciones interactivas**, y es el que se usa principalmente en la actualidad para **videojuegos, realidad virtual y simulación**, asistido por GPUs.
- La totalidad de las aplicaciones gráficas en dispositivos móviles y tabletas usan rasterización.

En este curso nos centraremos en los pasos de cálculo necesarios para la rasterización 2D y 3D, y veremos ejemplos prácticos en el contexto del *game engine* Godot.

Visualización basada en ray-tracing (2/2)

El algoritmo de Ray-tracing se puede optimizar para lograr que el cálculo de T en la línea 3 sea muy eficiente:

- Esto requiere el uso de **indexación espacial**: las primitivas se organizan en una estructura de datos que permite encontrar rápidamente las primitivas que cubren un pixel dado.
- Al usar esta optimización, la complejidad en tiempo será proporcional al logaritmo natural del número de primitivas ($\log n$), en lugar de a n , es decir, el tiempo de cálculo estará ahora en el orden de $O(p \cdot \log n)$.

Si se considera p una constante el tiempo está en $O(\log n)$

Ray-tracing

Respecto de la técnica de *Ray-tracing*:

- El método de Ray-tracing y sus variantes suele ser **más lento**, pero consigue resultados **más realistas** cuando se pretende reproducir ciertos efectos visuales.
- Las variantes y extensiones de Ray-tracing son preferibles para síntesis de imágenes *off-line* (no interactivas), y es el que se usa principalmente en la actualidad para **producción de animaciones y efectos especiales** en películas o anuncios.
- En los últimos años (2018 en adelante) han aparecido arquitecturas de GPUs con aceleración por hardware para Ray-Tracing, lo que está llevando a **implementar algunos videojuegos** usando Ray-Tracing (pero requieren GPUs muy potentes).

En este curso veremos algo de Ray-tracing.

El cauce gráfico en rasterización: entradas y salidas

El **cauce gráfico** es el conjunto de etapas de cálculo que permiten la síntesis de imágenes por rasterización:

- Las entradas al cauce gráfico se denominan **primitivas**, una primitiva es una forma visible que no se puede descomponer en otros más simples, en rasterización típicamente las primitivas son: triángulos, segmentos de líneas o puntos (en 2D o en 3D)
- Un **vértice** es un punto del espacio 2D o 3D, extremo de una arista de un triángulo, o de un segmento de recta, o donde se dibuja un punto. Una o varias primitivas se especifican mediante una **lista de coordenadas de vértices**, más alguna información adicional.
- El cauce escribe en el **framebuffer**, que es una zona de memoria donde se guardan uno o varios arrays con los colores RGB de los pixels de la imagen (y alguna información adicional). Está conectado al monitor.

Subsección 2.4.

El cauce gráfico en rasterización

Transformación y sombreado

Hay (entre otros) dos pasos importantes del cauce gráfico que se ejecutan en la GPU:

1. Transformación:

En esta etapa se parte de las coordenadas de un vértice que se especifican en la aplicación, y se calculan las coordenadas (normalizadas) de su proyección en la ventana.

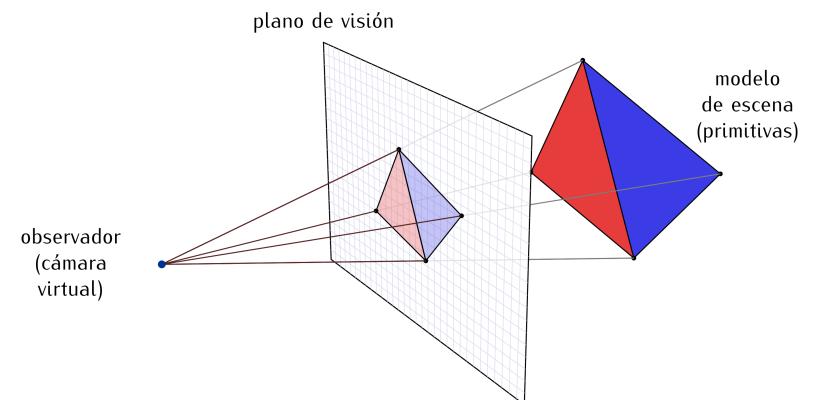
2. Sombreado:

El cálculo del color de un pixel (una vez que se ha determinado que una primitiva se proyecta en dicho pixel). Por lo visto hasta ahora, esto se hace simplemente asignando un color prefijado al pixel (pero es usualmente más complicado).

Además de estas etapas, hay otras como la rasterización y el recortado de polígonos.

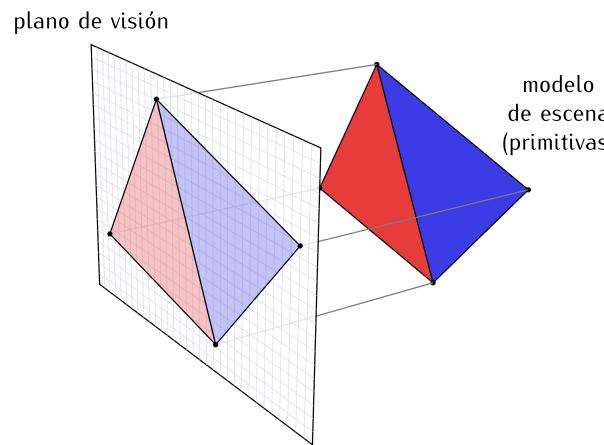
Transformación y proyección

Cada primitiva se sitúan en su lugar en el espacio, y se encuentra su proyección en un plano imaginario (**plano de visión**, *viewplane*) situado entre el **observador** y la escena (las primitivas):



Proyección paralela

La proyección puede ser **perspectiva** (como en la transparencia anterior), o **paralela**, como aparece aquí:



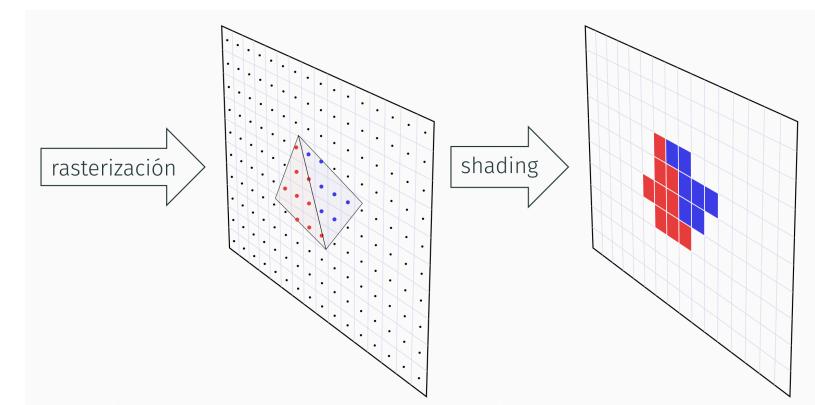
Sombreado: básico versus avanzado

El proceso de sombreado puede simplemente asignar un color *plano* a cada polígono (izquierda) o bien incluir cálculos avanzados con *iluminación* y *texturas* (derecha)



Rasterización y sombreado

- **Rasterización:** para cada primitiva, se calcula que pixels tienen su centro cubierto por ella.
- **Sombreado:** (*shading*) se usan los atributos de la primitiva para asignar color a cada pixel que cubre.



Etapas del cauce gráfico (1/2)

Con más detalle, el cauce gráfico tiene estas etapas:

1: Procesado de vértices: parte de una secuencia de vértices (puntos del espacio) y produce una secuencia de primitivas (puntos, segmentos o triángulos). Tiene estas sub-etapas:

1.1: Transformación: los vértices de cada **primitiva** son transformados en diversos pasos hasta encontrar su proyección en el plano de la imagen. Es realizado por un sub-programa llamado **Vertex Shader** (modificable por el programador, o *programable*).

1.2: Teselación y nivel de detalle: transformaciones adicionales avanzadas, realizadas por varios programas, entre ellos el **tesselation shader** y el **geometry shader** (programables).

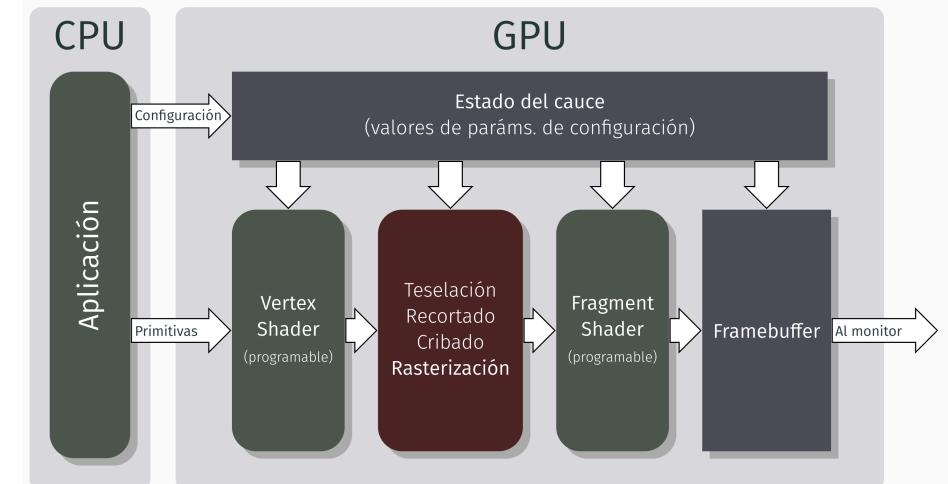
Etapas del cauce gráfico (2/2)

A continuación ocurren estas etapas:

2. **Post-procesado de vértices y montaje de primitivas** incluye varios cálculos como el *recortado* (*clipping*) y el *cribado de caras* (*face culling*), ninguno de ellos programable.
3. **Rasterización (rasterization)**: cada primitiva es *rasterizada* (discretizada), y se encuentran los pixels que cubre en la imagen de salida (no es programable).
4. **Sombreado (shading)**: en cada pixel cubierto se calcula el color que se le debe asignar. Se realiza por un programa llamado **fragment shader** o **pixel shader** (programable).

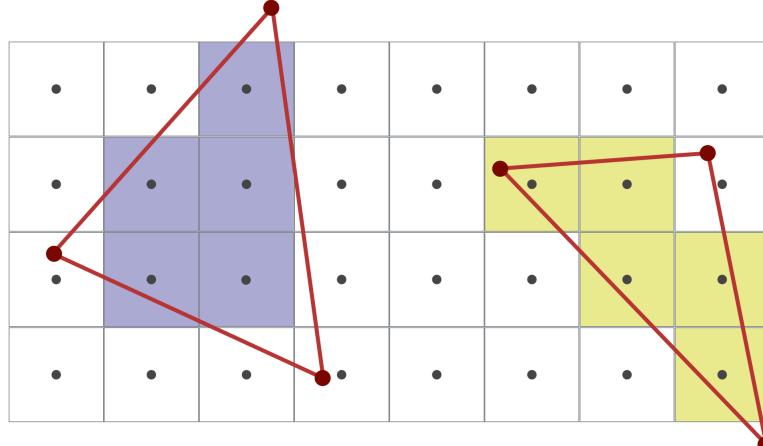
Esquema del cauce gráfico en una GPU

Este diagrama de flujo de datos refleja las etapas:



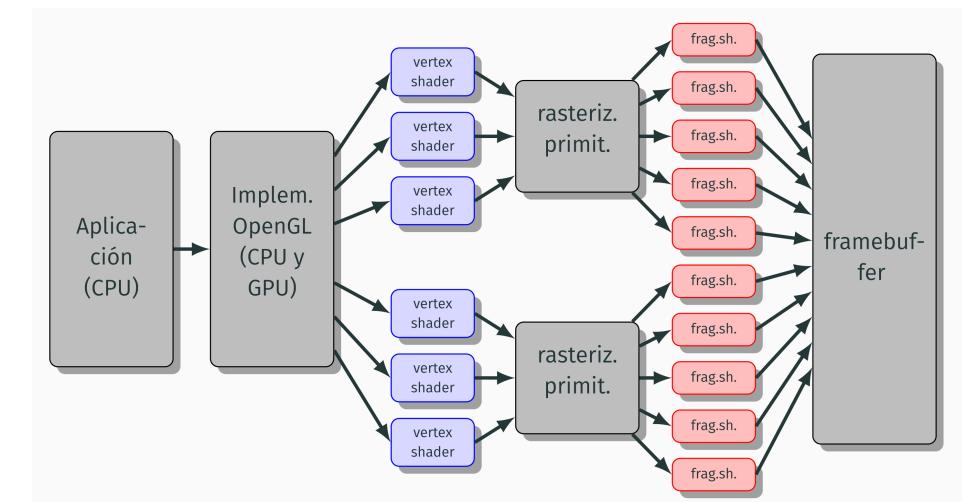
Ejemplo: rasterización de dos triángulos

La rasterización consiste en calcular los pixels cuyo centro está cubierto por la proyección de cada primitiva:



Paralelismo potencial en la GPU

En este grafo de dependencia vemos como se pueden ejecutar concurrentemente las ejecuciones de los distintos shaders para el ejemplo anterior:



Sección 3.

APIs y motores gráficos.

1. APIs para Rasterización, Ray-tracing y GPGPU
2. Motores gráficos

3. APIs y motores gráficos..

3.1. APIs para Rasterización, Ray-tracing y GPGPU.

APIs de rasterización

Una **API de rasterización** es la **especificación** de un conjunto de funciones y/o clases útiles para visualización 2D/3D basada en rasterización, es decir un documento con descripción de funciones (y sus parámetros), clases, interfaces, etc... junto con el comportamiento esperado de estas funciones y clases.

- Estas APIs son definidas por organizaciones sin ánimo de lucro (consorcios) o empresas privadas.
- Permiten la rasterización de primitivas de bajo nivel (polígonos, líneas segmentos), de forma eficiente y portable.
- La API OpenGL fue pionera en la rasterización en GPUs.
- Las APIs usan un *Shading Language*: un lenguaje de programación específico para los programas que se ejecutan en la GPU (*shaders*).

Subsección 3.1.

APIs para Rasterización, Ray-tracing y GPGPU

3. APIs y motores gráficos..

3.1. APIs para Rasterización, Ray-tracing y GPGPU.

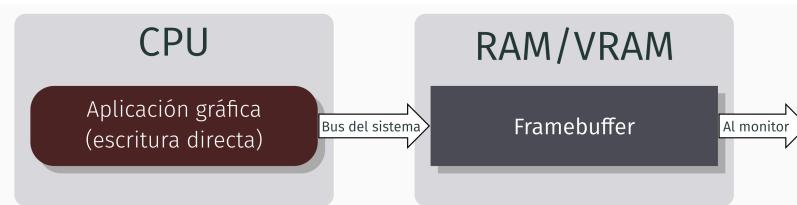
Evolución de la funcionalidad en rasterización

A lo largo de los años y hasta la actualidad, se incrementa la funcionalidad y programabilidad de las GPUs para rasterización, así como las plataformas soportadas:

- Se van incorporando lenguajes de *shading* de alto nivel estandarizados, como GLSL (Khronos), Cg (nVidia), HLSL (Microsoft), MSL (Apple).
- Se definen formatos de código intermedio para lenguajes de shading, como SPIR-V.
- Se pueden programar más etapas del cauce: *tesselation shaders*, *geometry shaders*, *mesh shaders*, etc...
- Partiendo de las *workstations* de finales de los 80, se incorporan GPUs programables en toda clase de dispositivos: ordenadores de sobremesa, ordenadores portátiles, tabletas, móviles y relojes.

Aplicaciones de escritura directa

Inicialmente (años 70-80), las aplicaciones gráficas escribían directamente en la memoria de vídeo (VRAM)

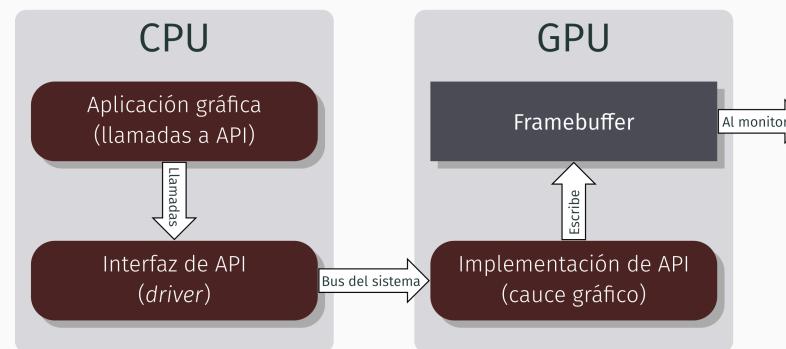


Desventajas

- La escritura en el framebuffer a través del bus del sistema es lenta, y se realiza pixel a pixel.
- Solución no portable entre arquitecturas hardware o software.
- Una aplicación gráfica no puede coexistir con otras

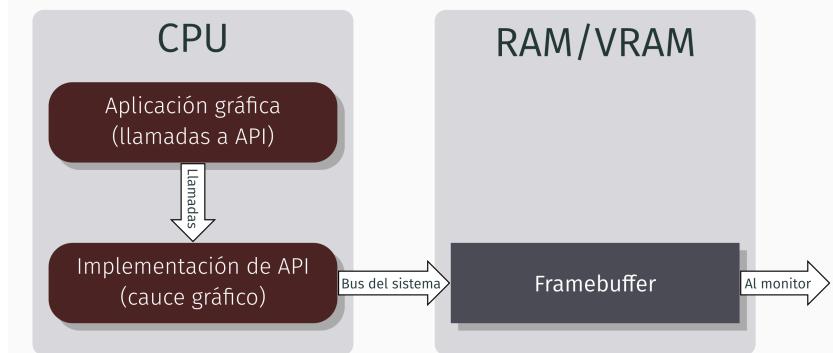
Uso de APIs y hardware gráfico (GPUs)

El uso de **GPUs aumenta la eficiencia** ya que ejecutan el cauce y reducen el tráfico a través del bus del sistema (se envía menos información de más alto nivel).



Uso de APIs gráficas

El uso de implementaciones de APIs gráficas portables (y gestores de ventanas) proporciona **portabilidad y acceso simultáneo**



- La escritura en el framebuffer a través del bus del sistema sigue siendo lenta.

Plataformas hard./soft.: OpenGL, OpenGL ES y Vulkan



- **OpenGL** (www.opengl.org) se puede usar en aplicaciones que se ejecutan en:
 - ▶ Linux y Windows, usando implementaciones (drivers) de los fabricantes de GPUs y macOS (nVidia, AMD, Intel)
 - ▶ macOS, con una implementación de Apple, que la considerada *obsoleta* (*deprecated*), pero funcional.
- **OpenGL ES** se puede usar en dispositivos Android, PS3 y PS4
- **Vulkan** (www.vulkan.org): se puede usar en aplicaciones que se ejecutan en:
 - ▶ Linux y Windows (con drivers de nVidia, AMD o Intel)
 - ▶ macOS, mediante MoltenVK de Khronos: traduce llamadas de Vulkan 1.2 a llamadas de Metal.
 - ▶ Dispositivos móviles y consolas: Android, Nintendo Switch.

Plataformas hardware y software: DirectX y Metal



- DirectX se ha implementado por Microsoft para diversas plataformas:
 - ▶ Windows para ordenadores.
 - ▶ Consolas de videojuegos XBox.
- Metal se ha implementado por Apple para sus distintos sistemas operativos sobre distintos tipos de dispositivos:
 - ▶ macOS en todos los ordenadores de Apple.
 - ▶ iOS en móviles iPhone.
 - ▶ iPadOS en dispositivos iPad.
 - ▶ tvOS en dispositivos Apple TV.
 - ▶ visionOS para dispositivos Vision PRO.

APIs modernas frente a tradicionales

Llamamos **APIs tradicionales** a: OpenGL, OpenGL ES, Direct X (hasta versión 11 incluida) y WebGL, y **APIs modernas** a Vulkan, Metal, DirectX (versión 12) y WebGPU.

Las APIs modernas son más eficientes:

- **Bajo nivel:** permiten un control más fino sobre el hardware
- **Multihebra:** aprovechan múltiples núcleos de CPU a la vez.
- **Menor sobrecarga:** reducen la sobrecarga de la CPU y el consumo de memoria.

Aunque también presentan desventajas:

- **Mayor complejidad:** requieren más código (y por tanto mayor tiempo de desarrollo) en comparación con las tradicionales.
- **Menor portabilidad:** requieren más esfuerzo para portar aplicaciones a distintas plataformas, y en algunos casos no se podrá mantener la eficiencia o la funcionalidad sin un esfuerzo importante de desarrollo.

APIs para programación del cauce en la Web



- **WebGL** (www.khronos.org/webgl/): soportado por todos los navegadores, diseñado por Mozilla foundation. Se han publicado dos versiones:
 - ▶ **WebGL 1:** lanzado en 2011, basado en OpenGL ES 2.0. Disponible en la inmensa mayoría de ordenadores y dispositivos móviles.
 - ▶ **WebGL 2:** lanzado en 2017, basado en OpenGL ES 3.0. Disponible en ordenadores y dispositivos móviles modernos (algunos dispositivos móviles de gama baja o antiguos podrían no soportarlo).
- **WebGPU** (www.w3.org/TR/webgpu/): basado en Vulkan, diseñado por W3C. Soportado (desde mediados de 2023) exclusivamente en las versiones para ordenadores de los navegadores.

Histórico de versiones de APIs (tradicionales).

Año	OpenGL	DirectX	OpenGL ES	WebGL
1992	1.0			
1995	1.5	1.0		
1996-2000		2.0 al 8.0		
2003		9.0	1.0	
2004	2.0		1.1	
2006	2.1	10.0		
2007			2.0	
2008	3.0			
2009	3.1 - 3.2	11.0		
2010	3.3 - 4.0 - 4.1			
2011	4.2			1.0
2012	4.3	11.1	3.0	
2013	4.4	11.2		
2014	4.5		3.1	
2015			3.2	
2017	4.6			2.0

Histórico de versiones de APIs (modernas).

A partir de 2017 no hay nuevas versiones de OpenGL ni OpenGL ES (se sustituye por Vulkan) ni WebGL (se sustituye por WebGPU).

Las APIs modernas se iniciaron con Metal en 2014, este es el historial de versiones:

Año	Metal	DirectX 12	Vulkan	WebGPU
2014	1 (iOS)			
2014	1 (macOS)			
2015		12.0		
2016			1.0	
2017	2			
2018		12.1	1.1	
2020		12.2	1.2	
2022	3		1.3	
2023				1.0

Evolución de la funcionalidad: soporte para Ray-Tracing

Además del cauce gráfico en rasterización, las GPUs se usan en la actualidad para acelerar otros tipos de algoritmos, tanto dentro como fuera del ámbito de los gráficos:

- En gráficos: se incorpora soporte hardware y software para acelerar Ray-Tracing (desde 2018 en adelante), a través de nuevas APIs o de extensiones de las APIs existentes:
 - ▶ **OptiX** (nVidia, 2009),
 - ▶ **Vulkan Ray-Tracing** (Khronos, 2018)
 - ▶ **Direct X Ray-Tracing** (Microsoft, desde 2018),
 - ▶ **Metal Ray-Tracing** (Apple, desde 2020).

Evolución de la funcionalidad: soporte para GPGPU

El uso de las GPUs se ha extendido a otros ámbitos, fuera del campo de la informática gráfica, en lo que se conoce como **cálculo de propósito general en GPUs** (*General Purpose Computing on GPUs*) abreviado como **GPGPU**:

- Los campos de aplicación son principalmente:
 - ▶ Entrenamiento de modelos de IA.
 - ▶ Ejecución de modelos de IA.
 - ▶ Simulación numérica (meteorología, dinámica de fluidos, resistencia de materiales, cálculo de estructuras, etc...)
 - ▶ Minado de criptomonedas.
- Se definen APIs o herramientas para este tipo de cómputo:
 - ▶ **CUDA** (nVidia, desde 2006).
 - ▶ **OpenCL** (consorcio Khronos, desde 2009),

Subsección 3.2.
Motores gráficos

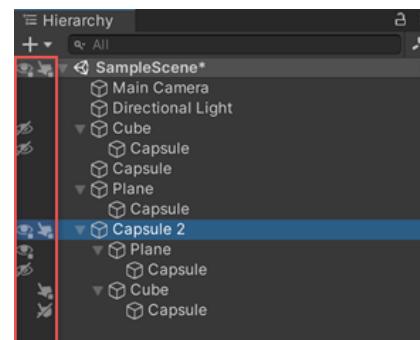
Motores gráficos (game engines)

Un **motor gráfico (game engine)** es un conjunto de herramientas software que facilita la creación de aplicaciones gráficas interactivas, principalmente videojuegos, aunque también se usan en simulación, visualización científica, realidad virtual, etc...

- Incluyen un motor de renderizado (basado en rasterización y/o ray-tracing), un editor visual, herramientas para animación, físicas, sonido, programación tradicional o visual, entre otras.
- Permiten crear aplicaciones gráficas portables y complejas sin necesidad de considerar detalles de bajo nivel.
- Algunos motores gráficos son de código abierto y gratuitos (Godot), otros son comerciales (Unreal Engine, Unity).
- Los motores gráficos usan APIs gráficas como OpenGL, Vulkan, DirectX o Metal para la rasterización y otras tareas gráficas.

Grafo de escena: modelo de la escena y objetos

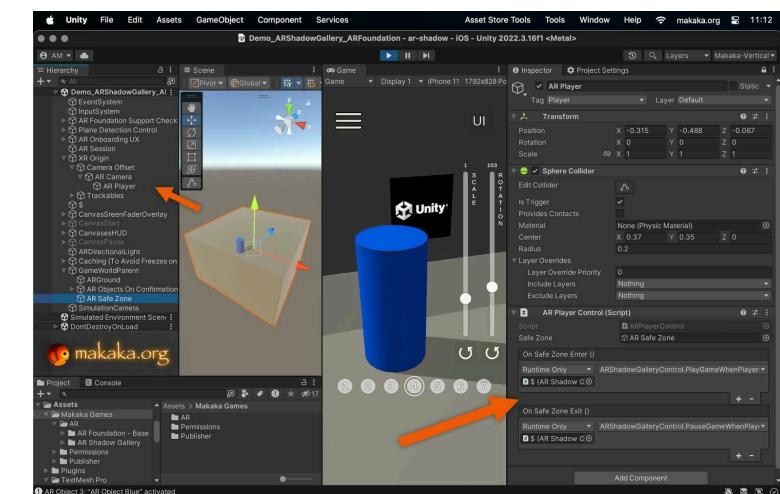
En un motor, el **grafo de escena** (o la **jerarquía**) es un estructura de datos (un árbol o un grafo dirigido acíclico) que modela las relaciones jerárquicas entre los objetos de la aplicación (escenas, cámaras, luces, objetos geométricos, entre otros)



Captura de una *hierarchy* en Unity, obtenida de:
docs.unity3d.com/6000.2/Documentation/Manual/Hierarchy.html

El editor de los motores gráficos

El **editor** de un motor es la aplicación que permite diseñar visual e interactivamente



Captura del editor de Unity, obtenida de: makaka.org/unity-tutorials/ar-testing

Programabilidad: visual scripting

El término **visual scripting** se refiere a la posibilidad de programar aspectos de la aplicación gráfica creando un grafo que codifica un **diagrama de flujo de datos**:

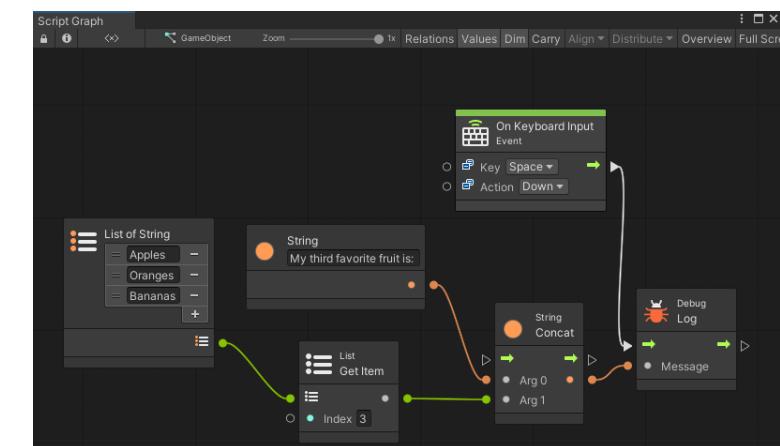


Imagen de un *script graph* de Unity: docs.unity3d.com/Packages/com.unity.visualscripting

Programabilidad: lenguajes de programación

Los motores gráficos también permiten programar partes de la aplicación usando lenguajes de programación tradicionales, como C++, C#, u otros específicos de un motor, como *GDScript* en Godot.

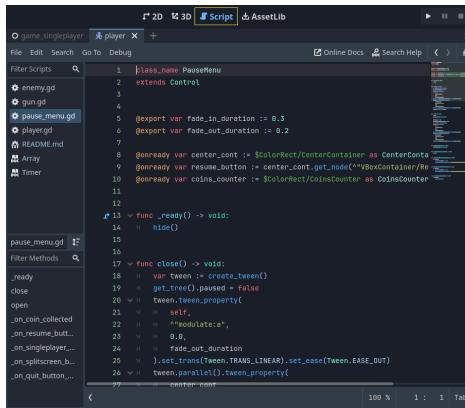


Imagen del editor de código fuente GDScript de Godot, tomada de:
docs.godotengine.org/en/latest/tutorials/editor/script_editor.html

Sesión 1: Introducción

Created 2025-12-01

Page 81 / 87.

Principales motores gráficos

Los motores gráficos más usados en la actualidad son:

- **Unreal Engine** (Epic Games): de fuentes *accesibles* (no de fuentes abiertas).
- **Unity** (Unity Technologies): privativo.
- **Godot** de fuentes abiertas.

Otros:

- **CryEngine** (Crytek)
- y muchísimos otros, ver: en.wikipedia.org/wiki/List_of_game_engines

Programabilidad: shaders

Se pueden programar los shaders, mediante *visual scripting* y/o lenguajes de *shading* bien propios, bien GLSL, HLSL, etc...

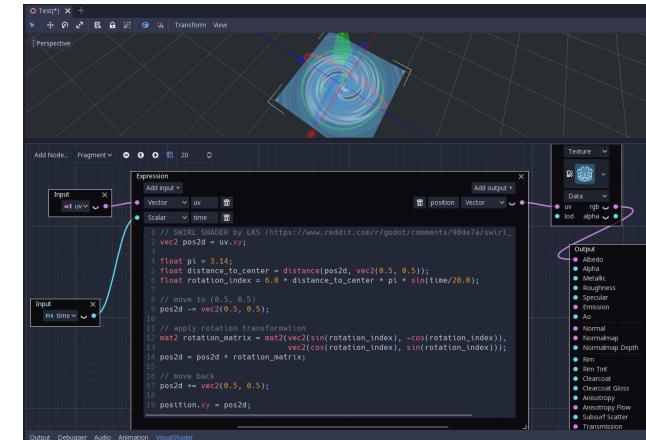


Imagen del editor visual de shaders de Godot, tomada de:
docs.godotengine.org/en/latest/tutorials/shaders/visual_shaders.html

Sesión 1: Introducción

Created 2025-12-01

Page 82 / 87.

Unreal Engine



**UNREAL
ENGINE**

Unreal Engine (www.unrealengine.com) es un motor gráfico de **Epic Games**:

- Desarrollo iniciado en 1995 para el videojuego *Unreal*.
- Programable con **C++** o bien usando *scripting visual* (**Blueprints**).
- Permite desarrollo en Windows, macOS y Linux.
- Fuentes accesibles gratuitamente, bajo registro en GitHub: github.com/EpicGames. Se pueden ver y modificar, pero no redistribuir cambios.
- La **licencia** permite uso **gratuito** educacional, o bien comercial si se factura menos de 1 millón de dólares, o bien si se comercializa en la [Epic Games Store](https://www.epicgames.com/store)

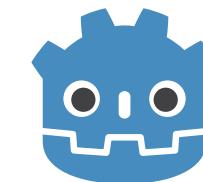
Unity



Unity (unity.com) es un motor gráfico desarrollado por **Unity Technologies**

- La primera versión se publicó en 2005.
- Programable usando el lenguaje **C#** o bien *scripting* visual.
- Permite desarrollo en Windows, macOS y Linux.
- Es software propietario.
- Se ofrece una versión gratuita limitada (sin acceso al código fuente del motor, sin soporte técnico *prioritario*, sin generación de ejecutables para consolas, entre otras limitaciones)

Godot



Godot (godotengine.org) es un motor gráfico *Open Source* desarrollado inicialmente por Juan Linietsky y Ariel Manzur y luego por una comunidad de desarrolladores:

- La primera versión se publicó en 2014.
- Programable usando **GDScript**, **C#**, **C++** y *scripting* visual.
- Permite desarrollo en Windows, macOS y Linux.
- Es de código abierto, los fuentes están disponibles en GitHub: github.com/godotengine/godot
- Es completamente gratuito y no requiere pago de licencias ni royalties.

Fin de transparencias.