

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.

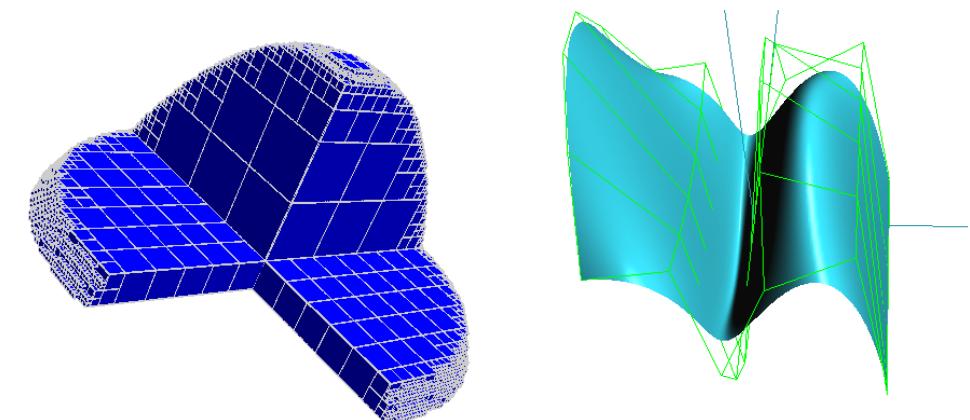
Informática Gráfica

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

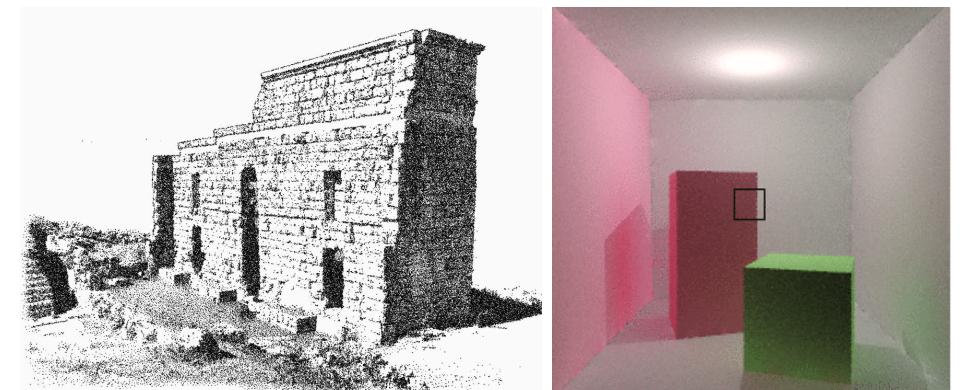


Simulador de Conducción con Editor de Entornos

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

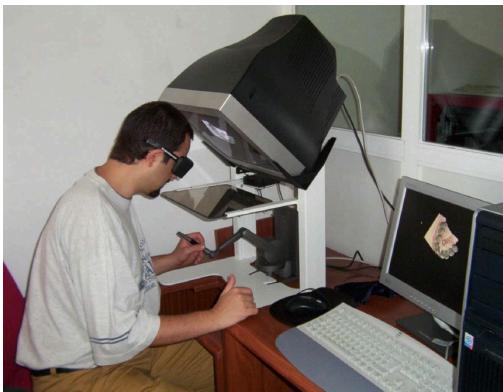
Aplicaciones: visualización (rendering)

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



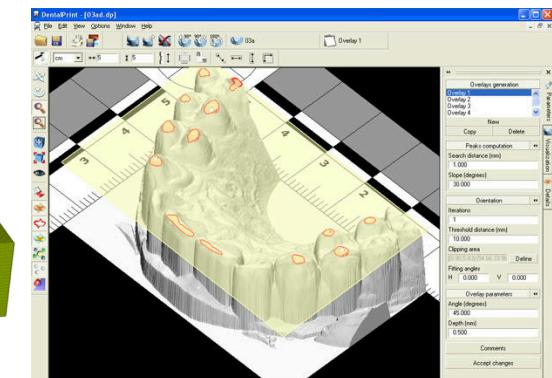
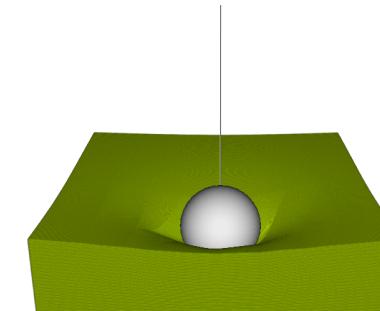
Aplicaciones: interacción y captura de modelos.

Adquisición de nuevos modelos a partir de objetos reales.



Aplicaciones: computación geométrica

Algoritmos y metodologías para procesamiento y edición de los modelos.



Subsección 1.2.

Objetivos, programa, temario.

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.

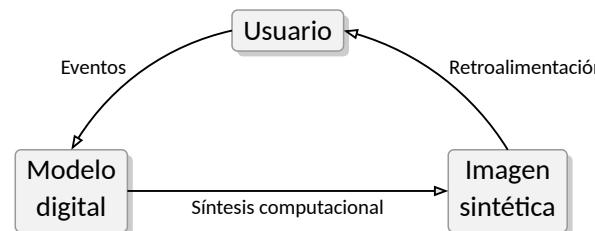
Este esquema es el que se usa típicamente en aplicaciones de simuladores, diseño asistido por computador, videojuegos, realidad virtual y realidad aumentada.

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...

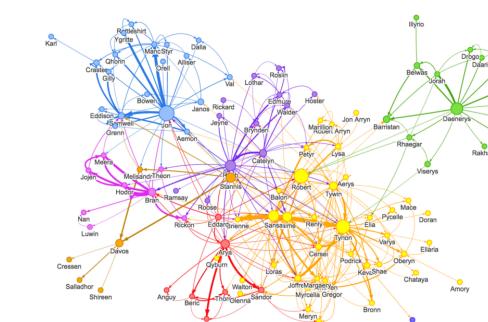


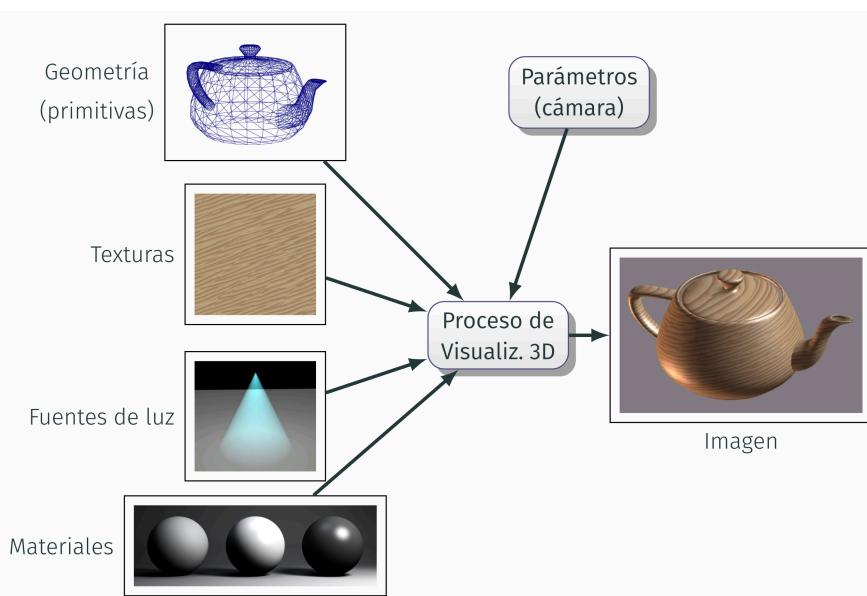
Imagen de *Hands on graph data visualization* (Relaciones entre personajes de Game of Thrones)
<https://neo4j.com/developer-blog/hands-on-graph-data-visualization/>

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.

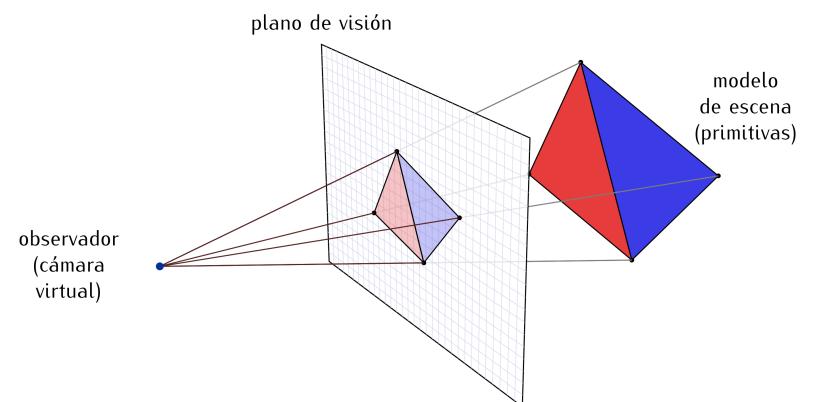
2. Sombreado:

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.

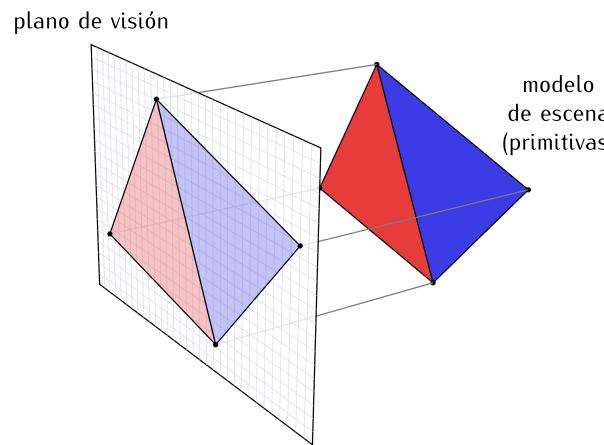
Transformación y proyección

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:



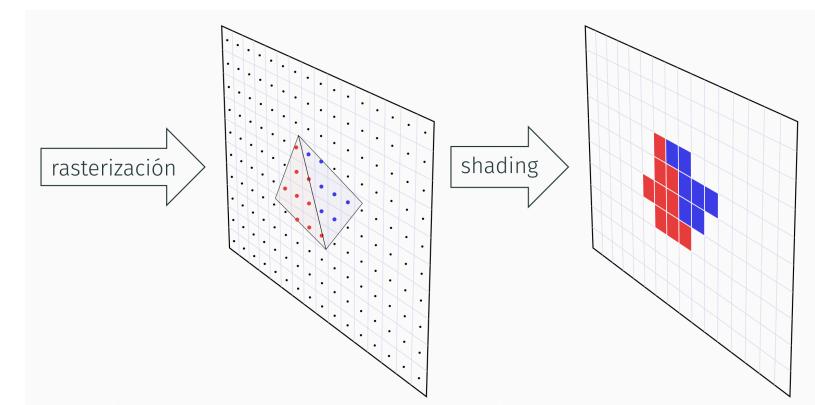
Sombreado: básico versus avanzado

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).

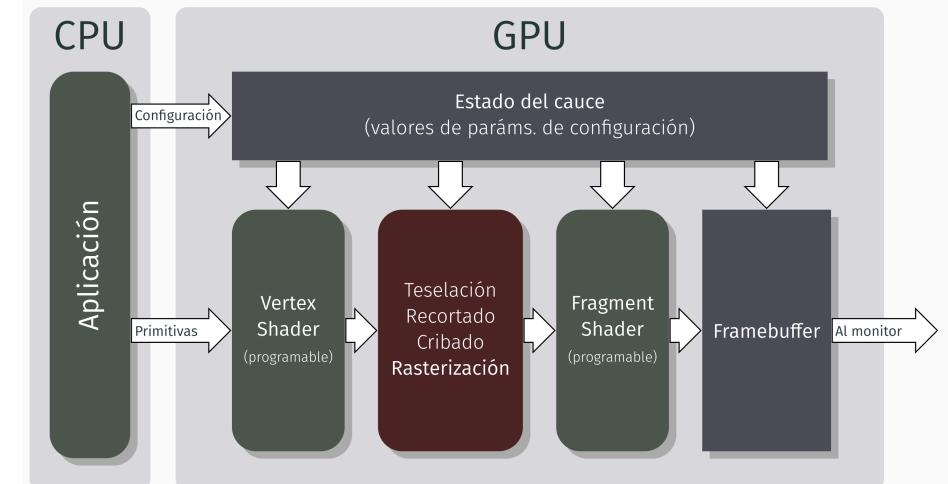
Etapas del cauce gráfico (2/2)

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).

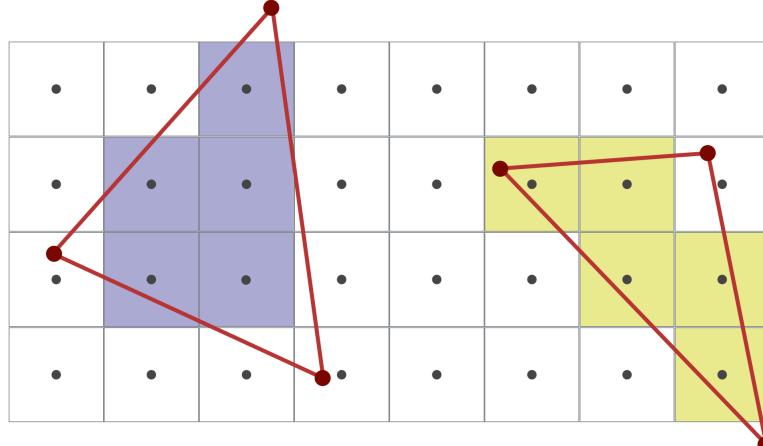
Esquema del cauce gráfico en una GPU

Este diagrama de flujo de datos refleja las etapas:



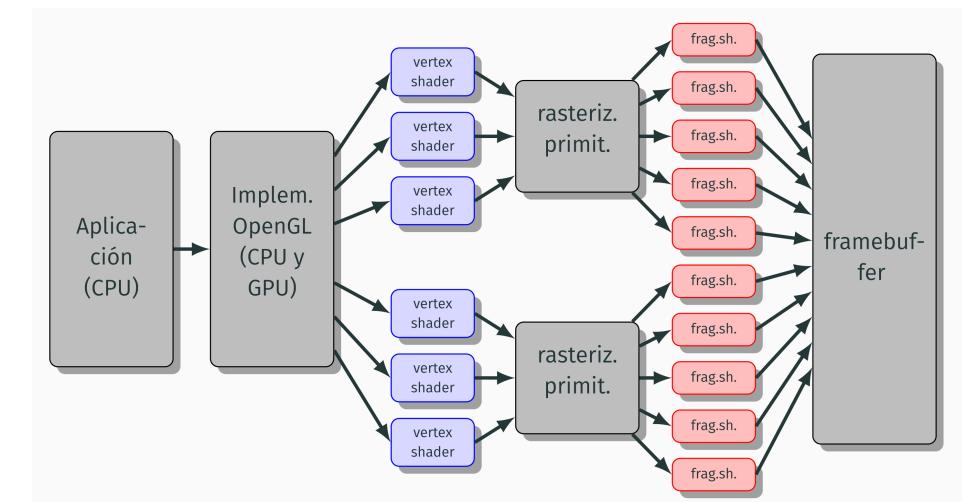
Ejemplo: rasterización de dos triángulos

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.

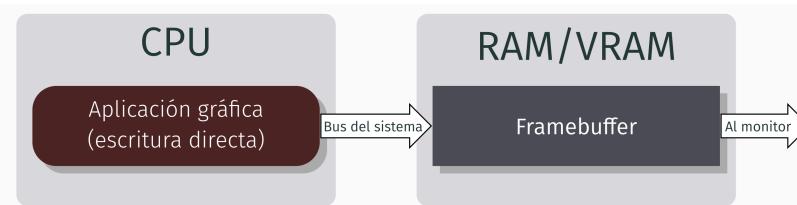
Evolución de la funcionalidad en rasterización

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)

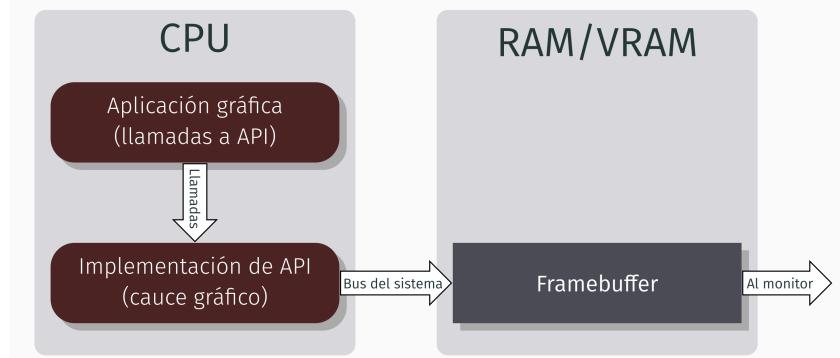


Desventajas

- 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 gráficas

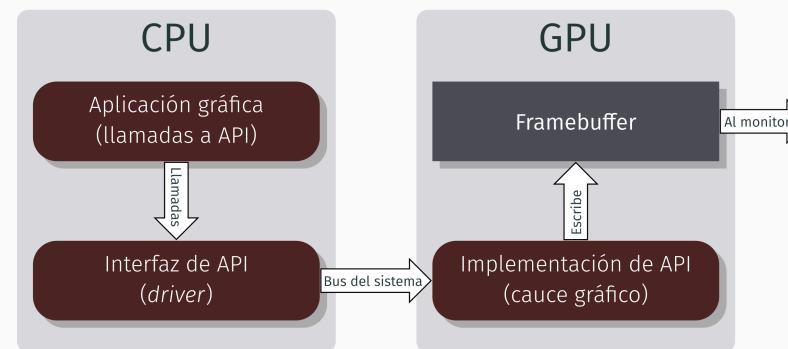
El uso de implementaciones de APIs gráficas portables (y gestores de ventanas) proporciona **portabilidad y acceso simultáneo**



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

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).



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

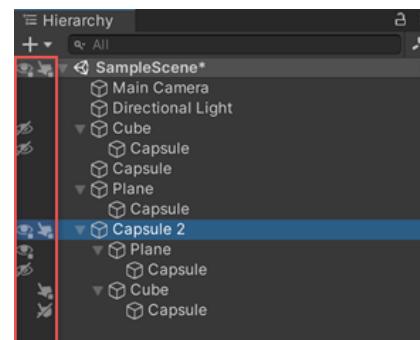
Motores gráficos (game engines)

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

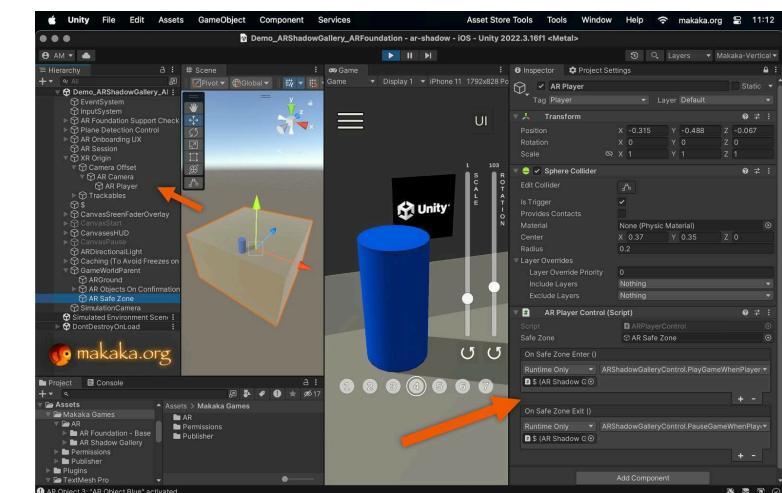
En un motor, el **grafo de escena** (o la **jerarquía**) es un estructura de datos (un árbol o un grafo dirigido acíclico) que modela las relaciones jerárquicas entre los objetos de la aplicación (escenas, cámaras, luces, objetos geométricos, entre otros)



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**:

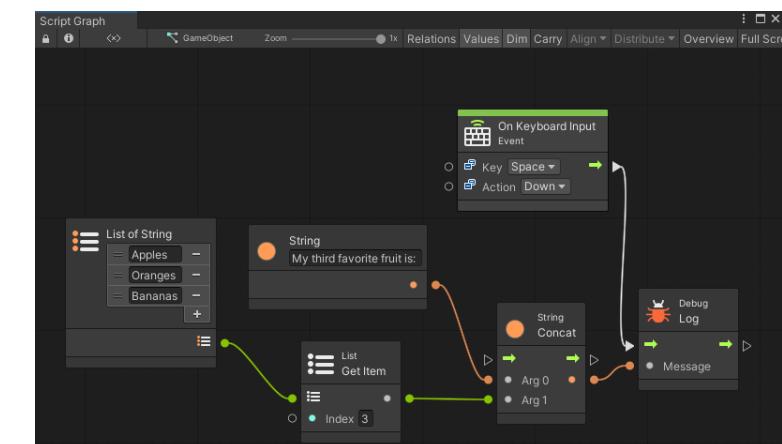


Imagen de un *script graph* de Unity: docs.unity3d.com/Packages/com.unity.visualscripting

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.

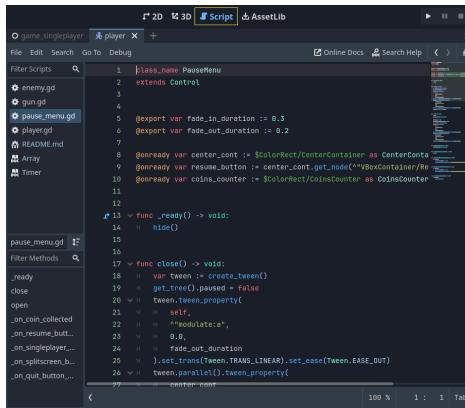


Imagen del editor de código fuente GDScript de Godot, tomada de:
docs.godotengine.org/en/latest/tutorials/editor/script_editor.html

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...

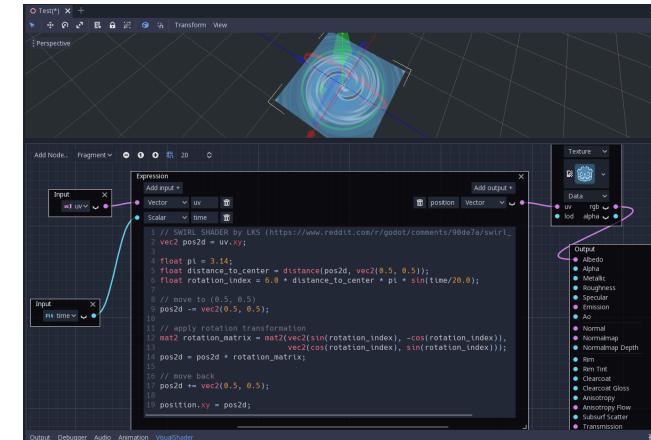


Imagen del editor visual de shaders de Godot, tomada de:
docs.godotengine.org/en/latest/tutorials/shaders/visual_shaders.html

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)

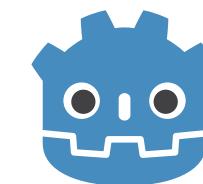
Unity



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.

Informática Gráfica.

Sesión 2: El *engine* Godot. Mallas..

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

Índice

Introducción a Godot	3
Mallas en Godot.	26
Problemas	89

1. Introducción a Godot.

Funcionalidad y características de Godot

Godot es un IDE (entorno integrado de desarrollo, *integrated development environment*) que permite:

- **Desarrollar aplicaciones gráficas** 2D y 3D interactivas, como videojuegos, simulaciones, visualizaciones, etc.
- **Ejecutar y depurar** esas aplicaciones desde Godot.
- **Generar archivos ejecutables** independientes con la aplicación creada, archivos que pueden ser ejecutados en diversas plataformas.

Sus características principales son:

- **Código abierto**
- **IDE Multiplataforma:** el IDE Godot se puede ejecutar en Linux, Windows, macOS, incluso en Web (aunque en Web con limitaciones)
- **Genera aplicaciones multiplataforma:** genera aplicaciones nativas independientes para Windows, macOS, Linux, Android, iOS, y aplicaciones Web.

-
- Sección 1.
- ### Introducción a Godot
1. El lenguaje de programación GDScript
 2. La jerarquía de clases de Godot.
 3. El bucle principal de Godot.

Elementos de Godot

Editor: aplicación tipo IDE con herramientas para crear y organizar los recursos del proyecto, diseñar escenas, programar scripts, etc... También permite **ejecutar** y **depurar** las aplicación en desarrollo.

Proyecto: conjunto de escenas, nodos, scripts y recursos asociados a una aplicación en desarrollo.

Escenas: una escena es una estructura jerárquica (un **árbol** llamado **árbol de escena**) de **nodos** que representan objetos y elementos de la aplicación en desarrollo.

Nodos: elementos de las escenas, cada uno es de una **clase** de Godot.

Scripts: código que define el comportamiento definido por el programador para los nodos y escenas. Se pueden usar los lenguajes orientados a objetos **GDScript** (similar a Python) o **C#**, y también su propio **lenguaje visual** (*visual script*).

Recursos: archivos de imágenes, audios, videos, modelos 3D, etc... que se usan para desarrollar una aplicación.

Características de GDScript

GDScript es un lenguaje de programación interpretado, de alto nivel, orientado a objetos, diseñado específicamente para Godot.

Sus características principales son:

- Sintaxis similar a Python: se usa **indentación** para definir bloques, sin punto y coma al final de las líneas (excepto si se quieren unir dos sentencias en una línea).
- Las variables pueden tener asociado un tipo conocido o no.
- Es **orientado a objetos**: incluye clases, herencia, y polimorfismo.
- Integración con el editor de Godot (permite crear y manipular nodos y escenas fácilmente)
- Gestión automática de memoria mediante conteo de referencias

Subsección 1.1.

El lenguaje de programación GDScript

Variables: declaración y tipo

GDScript permite tipos estáticos y dinámicos. Las variables pueden:

- Declararse con un **tipo explícito**: la variable no puede cambiar de tipo en su tiempo de vida.


```
var x : float = 10.0 # tipo 'float' (explícito)
```
- Declararse con un tipo **implícito (inferido)**: el tipo se calcula (se infiere) a partir de la expresión del valor inicial. Tampoco pueden cambiar de tipo después.


```
var y := 20 # tipo 'int' (es el tipo de la expresión '20').
```
- Declararse **sin tipo**: en este caso la variable puede cambiar de tipo en su tiempo de vida. Es de tipo **Variant**.


```
var z = 30.0 # sin tipo (inicialmente 'float', pero puede cambiar)
```

Ejemplo de archivo fuente GDScript

Tienen la extensión `.gd` y siempre definen una clase (`MiNodo` en este caso, aunque podría ser anónima) que siempre hereda de otra (`Node3D` en este caso):

```
extends Node3D # obligatorio: indica la clase base
class_name MiNodo # opcional: si no está es una clase anónima

var velocidad : float = 100.0 # variable de instancia (tipo opc.)

func v_cuadrado() -> float :
    return velocidad * velocidad # devuelve la velocidad al cuadrado

func _init():
    pass # constructor, puede tener parámetros
        # 'pass' indica que está vacío

func _ready():
    print("Nodo listo") # imprime en la consola o terminal

func _process( delta: float ):# método de proceso por frame
    position.x += velocidad * delta # usa 'position' de Node2D
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 9 / 100.

Tipos contenedores: arrays

GDScript incluye estos tipos arrays **dinámicos** (pueden crecerse o reducirse en tiempo de ejecución):

- **Array**: contiene elementos **Variant**, es decir, de cualquier tipo.
- **Array[T]**: todos sus valores son de tipo *T* (arrays homogéneos).
- **Arrays empaquetados (packed arrays)**: arrays homogéneos con elementos contiguos en memoria, se usan para enviar datos a la GPU. Los elementos pueden ser:
 - ▶ Bytes o enteros: `PackedByteArray`, `PackedInt32Array`, `PackedInt64Array`.
 - ▶ Flotantes de simple y doble precisión: `PackedFloat32Array`, `PackedFloat64Array`.
 - ▶ Vectores de 2, 3 o 4 componentes: `PackedVector2Array`, `PackedVector3Array`, `PackedVector4Array`.
 - ▶ Colores: `PackedColorArray`.

Tipos predefinidos en GDScript y Godot

Tipos básicos predefinidos en GDScript:

- **bool**: valores lógicos o booleanos (`true` o `false`).
- **int**: enteros (números sin parte decimal), de 64 bits.
- **float**: números reales (con parte decimal), de doble precisión (64 bits).
- **String**: cadenas de caracteres codificadas en *Unicode*.

Tipos para vectores predefinidos en GDScript:

- **Vector2**, **Vector3**, **Vector4**: tuplas con 2, 3 o 4 elementos flotantes de simple precisión (32 bits).
- **Vector2i**, **Vector3i**, **Vector4i**: tuplas con 2, 3 o 4 elementos enteros.
- **Transform2D**, **Transform3D**: matrices de transformación en 2D o 3D.

Tipos predefinidos en Godot:

- **Color**: colores en formato RGBA (rojo, verde, azul, alfa o transparencia).

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 10 / 100.

Ventajas del tipado estático

El uso de variables con tipo (tipado estático) es **aconsejable siempre**, ya que:

- Contribuye a **detectar más errores** en tiempo de desarrollo (en el editor) en lugar de en tiempo de ejecución, lo cual acorta el tiempo de desarrollo y disminuye la probabilidad de que el usuario final sufra errores.
- Permite **ejecutar la aplicación mucho más rápido**, ya que disminuye la sobrecarga del intérprete: no es necesario comprobar el tipo de una variable antes de realizar cualquier operación con ella.
- **Facilita la lectura y comprensión** del código: el programador puede añadir los tipos para que el lector (incluido él mismo en el futuro) entienda mejor el código.
- El uso de tipo implícito (inferido) permite **mayor expresividad** (aunque a veces disminuye la legibilidad).

Clase **Object** y derivadas

Las clases de Godot se organizan en una **jerarquía de herencia**.

La clase **Object** es clase raíz de la jerarquía de herencia (todas las demás clases heredan directa o indirectamente de ella). Destacamos estas clases derivadas directamente de **Object**:

Node: clase base para todos los nodos de las escenas. Incluye nodos para visualización 2D o 3D, elementos del interfaz de usuario (*controles*), cámaras, fuentes de luz, materiales, escenas, y otros muchos.

Viewport: representa una zona rectangular de una ventana (o una ventana completa) donde se renderiza una escena 2D o 3D. Cada proyecto tiene un **Viewport** por defecto que no aparece explícitamente en el grafo de escena.

MainLoop: clase abstracta con definiciones de métodos para implementar el bucle principal de la aplicación. Godot tiene una clase derivada que implementa por defecto los métodos, llamada **SceneTree**.

Subsección 1.2.

La jerarquía de clases de Godot.

- 1. Introducción a Godot.
- 1.2. La jerarquía de clases de Godot..

Otras clases derivadas de **Object**

RefCounted: clase base para objetos en memoria dinámica gestionada automáticamente mediante la técnica de *cuenta de referencias*. Hay múltiples clases derivadas, destacamos:

Resource: clase base para objetos que contienen recursos de Godot. También tiene muchas clases derivadas, destacamos:

Mesh: clase base para mallas 2D o 3D (estructuras de datos que codifican primitivas geométricas).

Material: clase base para materiales que definen la apariencia visual de objetos 2D o 3D.

Image, Texture: clases base para imágenes en 2D o 3D, y para texturas que se aplican a objetos 3D o superficies de objetos 3D.

Shader: clase base para shaders (programas que se ejecutan en la GPU).

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 14 / 100.

Tipos de nodos

Godot incorpora muchísimos tipos de nodos. Destacamos estas clases que se derivan directamente de la clase **Node**:

CanvasItem: clase base para elementos que se visualizan en 2D (es decir, que se definen en el plano). Tiene dos clases derivadas:

Control: clase base para elementos del interfaz de usuario (botones, menús, barras de progreso, etc...).

Node2D: clase base para nodos que representan objetos 2D en una escena 2D (imágenes, formas geométricas, textos, etc...).

Node3D: clase base para nodos que representan objetos 3D en una escena 3D. Su principal clase derivada es:

VisualInstance3D: objetos visuales en 3D: mallas y fuentes de luz.

Camera3D: nodo que representa una cámara en una escena 3D, permite visualizar la escena desde diferentes ángulos y posiciones.

Clases para mallas

Una **malla** es una estructura de datos que codifica una o varias primitivas geométricas (puntos, líneas, triángulos) y constituye la base para representar objetos 2D o 3D en gráficos por ordenador.

La clase **Mesh** es la clase base para mallas, un objeto de este tipo puede ser referenciado desde múltiples nodos (es derivada de **RefCounted**). Tiene estas clases derivadas:

ArrayMesh: malla definida por programador a partir de arrays de vértices y atributos (normales, colores, coordenadas de textura, etc...), se envía una vez a la GPU y permanece ahí para múltiples *frames*

ImmediateMesh: similar a la anterior, pero se envía a la GPU en cada *frame*.

PrimitiveMesh: clase base para diversas primitivas geométricas predefinidas, entre otras están: **BoxMesh**, **CylinderMesh**, **PlaneMesh**, **PointMesh**, **PrismMesh**, **SphereMesh**, **TextMesh**, **TorusMesh**.

Nodos 3D

Entre las clases derivadas de **VisualInstance3D** destacan las dedicadas a representar instancias de mallas y luces en 3D:

GeometryInstance3D: tiene varias subclases, entre ellas:

MeshInstance3D: nodo que instancia una malla (clase **Mesh**) en una escena 3D, asignándole una posición, orientación o escala. Un mismo objeto **Mesh** puede ser instanciado múltiples veces en una escena o en escenas diferentes.

MultimeshInstance3D: similar al anterior, pero permite múltiples instancias.

Light3D: clase base para diversos tipos de fuentes de luz, a saber:

DirectionalLight3D: luz lejana (en una dirección)

OmniLight3D: luz puntual (ilumina en todas las direcciones igual)

SpotLight3D: luz puntual que ilumina en direcciones en un cono.

Nodos 2D

Entre las clases derivadas de **Node2D** destacan:

MeshInstance2D: nodo que instancia una malla (clase **Mesh**) en una escena 2D, asignándole una posición, orientación o escala, y opcionalmente una textura. Un mismo objeto **Mesh** puede ser instanciado múltiples veces en una escena o en escenas diferentes.

MultiMeshInstance2D: similar al anterior, pero permite múltiples instancias.

Camera2D: nodo que representa una *cámara* en 2D, es decir, determina qué parte del plano 2D se visualiza en el viewport.

Sprite2D: un rectángulo con una textura o una parte de una textura (clase **Texture**) visible en su interior.

AnimatedSprite2D: similar al anterior, pero contiene más de una textura, de forma que se pueden reproducir como una animación.

El bucle principal: métodos

La clase abstracta `MainLoop` define los métodos que permiten configurar el comportamiento de cualquier aplicación creada con Godot. Los métodos son `_initialize`, `_process` y `_finalize`.

Cuando se ejecuta la aplicación en desarrollo, se dan estos pasos:

1. Se crea una instancia de un clase derivada de `MainLoop` (típicamente `SceneTree`, pero pueden definirse otras).
2. Se invoca el método `_initialize` para inicializar esa instancia.
3. Mientras no se termine la aplicación:
 1. Se invoca el método `_process` en dicha instancia para actualizar el estado de los objetos de la aplicación y renderizar la escena.
 2. Si es necesario, se hace una espera hasta que sea el momento del siguiente frame.
4. Se invoca el método `_finalize` para liberar recursos y finalizar la aplicación.

Subsección 1.3.

El bucle principal de Godot.

La clase `SceneTree`

La clase `SceneTree` es una implementación concreta de `MainLoop` que se caracteriza por incluir un árbol de nodos (una escena) y gestionar:

- La creación del árbol al inicio, según el diseño creado por el usuario en el editor.
- La adición y eliminación de nodos al árbol, de forma dinámica, en tiempo de ejecución.
- La actualización y renderizado de la escena en cada frame.
- Los cálculos asociados a la simulaciones físicas.
- Las entrada de usuario (teclado, ratón, etc...).
- La ejecución de scripts asociados a nodos y escenas.
- Terminación y liberación de recursos al finalizar la aplicación.

En esta asignatura únicamente se usará `SceneTree` para el bucle principal (no se intentan crear bucles personalizados).

Creación de nodos

En Godot el usuario (programador) puede diseñar el árbol de escena, creando los nodos que lo componen, y configurándolos por programas de dos formas:

Antes de ejecutar, en el editor

- Creando un nodo de una clase de Godot y asignándole después a ese nodo un `script` (archivo de código) que redefine uno o varios métodos de la clase `Node`,
- Creando un nodo de clase definida por el usuario, que herede directa o indirectamente de `Node`, y asignándole al nodo esa clase en el editor. Esa clase puede tener sus propios métodos específicos

En tiempo de ejecución, mediante un `script`

- Creando un nodo `h` con el método `new` de su clase y añadiéndolo al árbol como un hijo de un nodo `p` existente, mediante: `p.add_child(h)` (se usa el método `add_child` del nodo padre)

Redefinición de métodos

El comportamiento de un nodo puede adaptarse redefiniendo métodos de `Node` u `Object` que se invocan por `SceneTree` en determinados momentos durante la ejecución:

- `_init` : es el constructor del nodo (método de `Object`), se invoca al crear un nodo para inicializar sus variables de instancia. Puede tener parámetros.
- `_enter_tree` : se invoca al añadir un nodo al árbol, cuando su padre se ha añadido, pero antes de añadir sus hijos.
- `_ready` : se invoca al añadir el nodo, después de `_enter_tree`, cuando ya se han añadido los nodos hijos.
- `_process(delta)` : se invoca antes de cada frame para actualizar el estado del nodo. El parámetro `delta` indica el tiempo (en segundos) transcurrido desde el último frame.
- `_input(event)` : invocado cuando se produce un evento de entrada (teclado, ratón, etc...). El parámetro `event` lleva información del evento.

Introducción

En esta sección se introducen los arrays de vértices como una forma de representar conjuntos de primitivas gráficas en general, y se indica como se pueden usar para crear objetos gráficos 2D y 3D en Godot.

- La idea esencial es que una secuencia de vértices (puntos del espacio representados, como mínimo, por sus coordenadas), puede codificar polilíneas, segmentos, triángulos, polígonos, etc...
- En Godot a los objetos gráficos (2D y 3D) definidos de esta forma se les denomina de forma genérica **mallas** (*meshes*).
- Una clase particular de arrays de vértices (de mallas) son las **mallas indexadas de triángulos**, que son el tipo más común de representar las superficies de los objetos en 3D.

Sección 2.
Mallas en Godot.

1. Tipos de primitivas
2. Atributos de vértices
3. Modos de envío de datos a la GPU
4. Representación de mallas en Godot
5. Mallas en 2D.
6. Mallas en 3D.
7. Mallas indexadas de triángulos en 3D

Subsección 2.1.
Tipos de primitivas

Especificación de primitivas. Tipos de primitivas.

En Godot (y resto de engines y APIs de rasterización), cada primitiva o conjunto de primitivas se especifica mediante una secuencia ordenada de coordenadas de vértices:

- Un vértice es un punto de un espacio afín 3D.
- Se representa en memoria mediante una tupla de coordenadas en algún marco de coordenadas de dicho espacio afín.
- Puede tener asociados otros valores, llamados **atributos** (p.ej. un color).

Existen tres clases de primitivas: **puntos**, **segmentos** y **triángulos**:

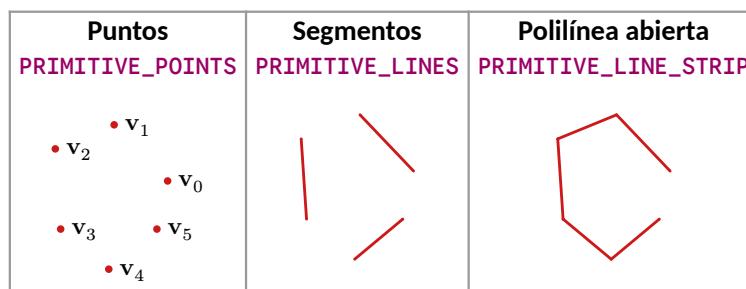
- Por tanto, además de la secuencia de vértices, es necesario tener información acerca de que tipo de primitiva representa dicha secuencia.

A cada forma de codificar primitivas en una secuencia se le llama un **tipo de primitivas**, en GDScript se definen diversas constantes para eso, del tipo enumerado **PrimitiveType**, en la clase **Mesh**.

Primitivas de tipo puntos y segmentos.

Una secuencia de coordenadas (v_0, v_1, \dots, v_{n-1}) pueden formar: puntos, o segmentos o polilíneas abiertas.

Aquí se ilustran los posibles tipos de primitivas (con puntos o segmentos) para una secuencia con $n = 6$:



Tipos de primitivas: puntos y segmentos

Una lista de n coordenadas de vértices (con $n \neq 1$) puede usarse para codificar puntos o segmentos. Más en concreto, puede codificar:

Tipo de primitiva	Descripción
Mesh.PRIMITIVE_POINTS	n puntos aislados (n arbitrario)
Mesh.PRIMITIVE_LINES	$n/2$ segmentos independientes (n debe ser par)
Mesh.PRIMITIVE_LINE_STRIP	$n - 1$ segmentos formando una polilínea abierta (n debe ser mayor o igual a 2)
Mesh.PRIMITIVE_TRIANGLES	$n/3$ triángulos (n debe ser múltiplo de 3)
Mesh.PRIMITIVE_TRIANGLE_STRIP	$n - 2$ triángulos compartiendo aristas (tira de triángulos), (n debe ser mayor o igual que 3)

Triángulos delanteros y traseros. Cribado.

Cada primitiva de tipo triángulo (también llamada **cara**, **face**) es clasificada por Godot como de **orientación delantera** o **orientación trasera**:

- Será **delantera** si sus vértices se visualizan en pantalla en el sentido de las agujas del reloj.
- Será **trasera** si sus vértices se visualizan en pantalla en el sentido contrario de las agujas del reloj

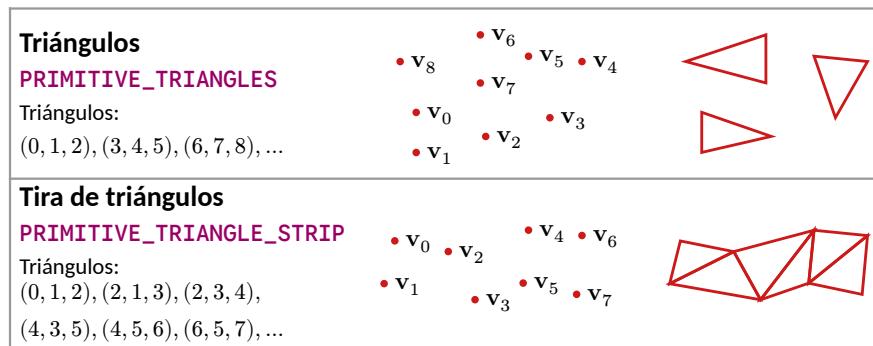
Godot usa la orientación para decidir si una cara se visualiza o no:

- Por defecto, Godot visualiza solo las delanteras (se llama hacer **cribado de caras**, **face culling**)
- Puede ser configurado para visualizar solo las delanteras, solo las traseras o todas. Se hace cambiando los parámetros de los *spatial shaders*.

Esta clasificación tiene utilidad especialmente en visualización 3D.

Primitivas de tipos triángulos (rellenos)

En estos tipos de primitivas la secuencia codifica uno o más triángulos, todos ellos rellenos. Aquí vemos los puntos en las coordenadas y a la derecha un esquema de las aristas de los triángulos que se formarán.



Secuencias indexadas

Para solucionar el problema, las APIs y engines permiten especificar una secuencia de vértices (con repeticiones) a partir de una secuencia de vértices únicos:

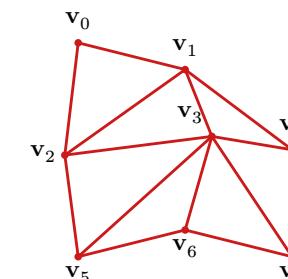
- Se parte de una secuencia V_n de n coordenadas arbitrarias de vértices $V_n = \{v_0, v_1, \dots, v_{n-1}\}$.
- Se usa una secuencia I_m de m **índices** $I_m = \{i_0, i_1, \dots, i_{m-1}\}$ donde cada valor i_j es un entero entre 0 y $n - 1$ (ambos incluidos). Puede haber valores repetidos.
- La secuencia de vértices V_n y la de índices determinan otra secuencia S_m de m vértices:

$$S_m = \{v_{i_0}, v_{i_1}, \dots, v_{i_{m-1}}\}$$

que tiene las mismas coordenadas de vértices de V_n pero en el orden especificado por los índices en I_m .

Problema de vértices replicados

A veces necesitamos repetir coordenadas de un vértice, p.ej. si queremos visualizar estos 7 triángulos (en modo aristas):



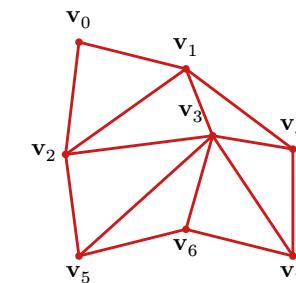
Usando **GL_TRIANGLES**, necesitamos esta secuencia de vértices:

$v_0, v_2, v_1, v_1, v_2, v_3, v_1, v_3, v_4, v_2, v_5, v_3, v_3, v_5, v_6, v_3, v_6, v_7, v_3, v_7, v_4$

Supone **emplear más memoria y/o tiempo para visualizar del necesario**. La secuencia tiene 21 coordenadas de vértices, pero solo hay 8 distintos (p.ej., v_2 aparece 4 veces y v_3 aparece 6 veces).

Ejemplo de secuencia indexada

En este ejemplo que hemos visto antes



usaríamos una lista de índices (cada tres forman un triángulo):

$$I_8 = \{0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 7, 4\}$$

$$I_{21} = \{0, 2, 1, 1, 2, 3, 1, 3, 4, 2, 5, 3, 3, 5, 6, 3, 6, 7, 3, 7, 4\}$$

Atributos de vértices

Las coordenadas de su posición se considera un **atributo** de los vértices, es un atributo imprescindible, pero en rasterización se pueden opcionalmente usar otros atributos, por ejemplo:

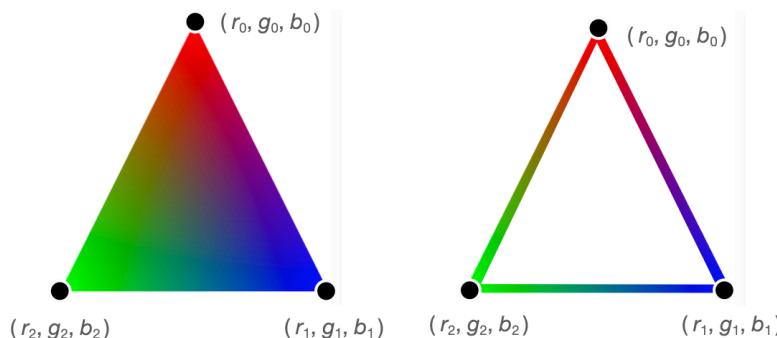
- El **color** del vértice (una terna RGB con valores entre 0 y 1).
- La **normal**: una vector unitario con tres coordenadas reales, determina la orientación de la superficie de un objeto en el punto donde está el vértice. Se usa para iluminación.
- Las **coordenadas de textura**: típicamente un par de valores reales, que se usan para determinar que punto de la textura se fija al vértice (lo veremos)

En Godot se pueden definir estos atributos y algunos más. En último término, el significado de un atributo (excepto el de la posición), está fijado en los *shaders* del cauce en uso. Godot permite atributos arbitrarios.

Subsección 2.2.
Atributos de vértices

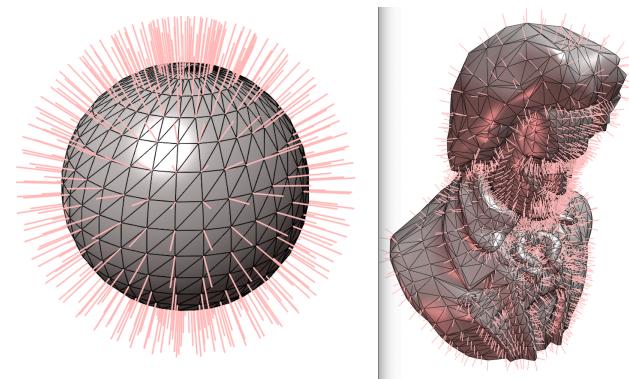
Atributos: colores de vértices

Es posible asignar un color a cada vértice, es una terna RGB con tres reales (r, g, b), (con valores entre 0 y 1) o bien una cuádrupla RGBA (RGB+transparencia). En el interior (o en las aristas) del polígono se usa interpolación para calcular el color de cada pixel.



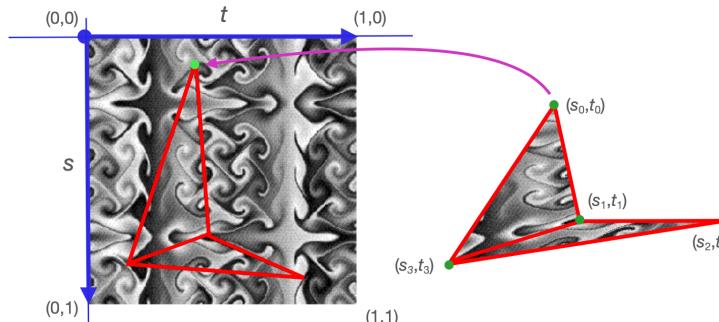
Atributos: normales

En visualización 3D, a cada vértice se le puede asociar un vector de 3 componentes (x, y, z) (su vector **normal**) que determina la orientación de la superficie en ese vértice y sirve para hacer el sombreado y la iluminación:



Atributos: coordenadas de textura

Para usar imágenes (texturas) en lugar de colores, podemos asociar a cada vértice un par de reales (s, t) (sus **coordenadas de textura**), típicamente en $[0, 1]^2$. Esto determina como se aplica la imagen (a la izquierda) a las primitivas (a la derecha):



Definición de valores de atributos

En Godot a cada vértice **siempre** se le asocia una tupla por cada atributo.

- Es decir, todo vértice tiene siempre asociado una posición, un color, una normal y unas coordenadas de textura (u otros atributos definidos por la aplicación).
- Según la configuración del cauce, algunos atributos serán usados o no. P.ej., si un objeto no tiene textura, no se usarán sus coordenadas de textura, o si no está activada la iluminación, no se usará la normal.
- Podemos definir único valor de un atributo para todos los vértices de una primitiva, o bien especificar un valor para cada vértice.

El valor de cada atributo está definido en cada pixel donde se proyecta la primitiva. Estos valores se calculan durante la rasterización usando **interpolación**.

Envío de datos de la CPU a la GPU

Las GPUs modernas están diseñadas para visualizar secuencias de vértices y atributos **almacenados en la memoria de la GPU**:

- Esto se debe a que desde la GPU el acceso a su propia memoria es muchísimo más rápido que el acceso a la memoria del sistema.
- Sin embargo, el origen de los datos siempre estará en la memoria de la aplicación (la memoria del sistema, es decir, la accesible por la CPU). Esto se debe a que los datos pueden leerse de un archivo o generarse, pero siempre quedan inicialmente en la CPU.
- Por tanto, es necesario realizar un **envío de datos** desde la CPU a la GPU, previo a la visualización.

Hay diversas formas de realizar ese envío, en base a la decisión sobre cuando se hace.

Modos de envío de datos a la GPU

Las dos formas de enviar las secuencias de vértices y sus atributos son:

- Envío en **modo inmediato**:

- ▶ Cada vez que queremos visualizar un frame, se envían los atributos e índices a la GPU por el bus del sistema.
- ▶ Este modo de visualización es muy ineficiente en tiempo (requiere transferir muchos datos por cada cuadro o frame), ya que el ancho de banda del bus del sistema es limitado (menor que los accesos a memoria en la GPU).

- Envío en **modo diferido**:

- ▶ Los datos de la secuencia de vértices se envían a la GPU una sola vez, usualmente como parte de la inicialización de la aplicación.
- ▶ Emplea mucho menos tiempo por cuadro que el anterior, ya que la transferencia de datos se hace menos veces, usualmente una sola vez.

Modos de envío en Godot

El **engine Godot**:

- Usa normalmente el envío en **modo diferido**, para los nodos que contienen mallas (secuencia de vértices) en 2D y 3D.
- Se puede usar el envío en **modo inmediato**, usando nodos de tipos específicos para ello.
- Incluye una API de visualización 2D en modo inmediato (funciones para dibujar explícitamente líneas, rectángulos, polígonos, círculos, texto, etc...). Al usar estas funciones, todas las coordenadas y atributos se envían en cada llamada (la cual se hace en cada frame).

Uso de los modos de envío

Por defecto, en gráficos se usa casi siempre el envío en **modo diferido** dada su mayor eficiencia. Sin embargo, a veces es conveniente usar el modo inmediato, únicamente cuando se dan cada una de estas dos condiciones:

- La secuencia de vértices y atributos es actualizada (cambia) por la aplicación (desde la CPU) con mucha frecuencia (p.ej. cada frame).
- La secuencia es pequeña (p.ej. menos de unos pocos miles de vértices).

En estos casos el envío previo a cada frame de los datos actualizados es realizable ya que no penaliza mucho el tiempo debido a que la secuencia de vértices no ocupa mucha memoria.

En mallas grandes se usa el envío en **modo diferido**:

- En cada frame pueden estar instanciadas en una posición, orientación o escala distinta (lo veremos más adelante).
- Si algunas coordenadas u otros atributos cambian, se pueden usar *vertex* o *geometry shaders* para programar esos cambios directamente en la GPU.

Subsección 2.4.

Representación de mallas en Godot

Almacenamiento de vértices y atributos: AOS y SOA (1/2)

Las APIs y engines suelen ofrecer dos formas de almacenar arrays de posiciones de vértices y sus atributos:

- **Array de estructuras (Array Of Structures, AOS):** se usa un array o vector, donde cada entrada contiene las coordenadas de un vértice y todos sus atributos.
- **Estructura de arrays (Structure Of Arrays, SOA):** se usa una estructura con varios (punteros a) arrays de número de elementos. Uno de ellos contiene las coordenadas y los otros contienen cada uno una tabla de atributos (colores, normales, coordenadas de textura).

Los **índices** (si hay) siempre están contiguos en su propio array.

En Godot se usa la opción SOA.

Tuplas de reales para posiciones y otros atributos

En GDScript, se contemplan diversos tipos de datos para guardar tuplas con valores reales, a saber:

- **Vector2:** tupla de dos reales (p.ej., para posiciones 2D o coordenadas de textura)
- **Vector3:** tupla de tres reales (p.ej., para posiciones 3D o normales)
- **Color:** tupla de cuatro reales (p.ej., para colores RGB o RGBA)

Características:

- Los elementos de las tuplas son valores reales (números de coma flotante de precisión simple, 32 bits, según la norma IEEE 754).
- Estos tipos son clases, incluyen métodos para hacer operaciones con ellos (suma, producto por un real, etc...).

Almacenamiento de vértices y atributos: AOS y SOA (2/2)

En AOS hay una única secuencia de valores reales:

$$\text{verts.} \equiv \left\{ \underbrace{\overbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{r_0, g_0, b_0}_{\text{color 0}}, \underbrace{n_{x0}, n_{y0}, n_{z0}}_{\text{normal 0}}, \underbrace{s_0, t_0}_{\text{cc.t. 0}}}^{\text{vértice 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \underbrace{r_1, g_1, b_1}_{\text{color. 1}}, \dots, \underbrace{s_{n-1}, t_{n-1}}_{\text{cc.t. } n-1} \right\}$$

En SOA hay una secuencia de reales por cada atributo (posibl. vacía):

$$\text{posiciones} \equiv \left\{ \underbrace{\overbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \underbrace{x_2, y_2, z_2}_{\text{posición 2}}, \dots, \underbrace{x_{n-1}, y_{n-1}, z_{n-1}}_{\text{posición } n-1}} \right\}$$

$$\text{colores} \equiv \left\{ \underbrace{\overbrace{r_0, g_0, b_0}_{\text{color 0}}, \underbrace{r_1, g_1, b_1}_{\text{color 1}}, \underbrace{r_2, g_2, b_2}_{\text{color 2}}, \dots, \underbrace{r_{n-1}, g_{n-1}, b_{n-1}}_{\text{color } n-1}} \right\}$$

$$\text{normales} \equiv \left\{ \underbrace{\overbrace{n_{x,0}, n_{y,0}, n_{z,0}}_{\text{normal 0}}, \underbrace{n_{x,1}, n_{y,1}, n_{z,1}}_{\text{normal 1}}, \underbrace{n_{x,2}, n_{y,2}, n_{z,2}}_{\text{normal 2}}, \dots, \underbrace{n_{x,n-1}, n_{y,n-1}, n_{z,n-1}}_{\text{normal } n-1}} \right\}$$

$$\text{cc.text.} \equiv \left\{ \underbrace{\overbrace{s_0, t_0}_{\text{cc.t. 0}}, \underbrace{s_1, t_1}_{\text{cc.t. 1}}, \underbrace{s_2, t_2}_{\text{cc.t. 2}}, \dots, \underbrace{s_{n-1}, t_{n-1}}_{\text{cc.t. } n-1}} \right\}$$

Identificación de atributos y tablas asociadas

Se suele asociar un valor entero a cada atributo de vértice. En GDScript se definen el tipo enumerado **ArrayType** en la clase **Mesh**. Los posibles valores son:

Identificador	Valor	Significado
Mesh.ARRAY_VERTEX	0	Coordenadas de posición (obligatorias)
Mesh.ARRAY_NORMAL	1	Vector normal
Mesh.ARRAY_TANGENT	2	Vector tangente
Mesh.ARRAY_COLOR	3	Color RGB
Mesh.ARRAY_TEX_UV	4	Coordenadas de textura
Mesh.ARRAY_TEX_UV2	5	Segundas coordenadas de textura
Mesh.ARRAY_CUSTOM0-3	6 – 9	Atributos definidos por el usuario

Existen otros posibles atributos relacionados con *skinning* y *rigging*, pero no los veremos aquí. Hay un total de **Mesh.ARRAY_MAX** (13 en la actualidad) posibles atributos.

Almacenamiento en arrays empaquetados de GDScript

En GDScript las tablas de atributos de vértices se pueden codificar en arrays **empaquetados (packed)**, son un tipo (una clase) con un arrays de elementos que garantizan que todos los valores están adyacentes en memoria.

Atributo	Clase array	Tipo elem.
Posiciones 2D	PackedVector2Array	Vector2
Posiciones 3D	PackedVector3Array	Vector3
Colores	PackedColorArray	Color
Normales	PackedVector3Array	Vector3
Coords. textura	PackedVector2Array	Vector2

Los índices (si hay) se almacenan en un array empaquetado de enteros, de tipo **PackedInt32Array** con enteros valiendo hasta 2^{31} , lo cual es suficiente para prácticamente cualquier malla (aunque se podría usar **PackedInt64Array** si hiciera falta).

Clases para mallas en Godot

Para definir objetos gráficos mediante mallas (arrays de vértices) en Godot, se usan diversas clases:

- **ArrayMesh**: derivada de **Mesh**, contiene una malla que se visualizará en **modo diferido**.
- **ImmediateMesh**: derivada de **Mesh**, contiene una malla que se visualizará en **modo inmediato**.
- **SurfaceTool**: derivada de **RefCounted**, es una clase que facilita la creación de mallas (de tipo **ArrayMesh** o **ImmediateMesh**) a partir de llamadas a métodos que van añadiendo vértices y atributos.
- **MeshInstance2D**, derivada de **Node2D**, permite *instanciar* una malla (**Mesh**) en una posición, orientación y escalas dadas en el plano 2D.
- **MeshInstance3D**, derivada de **Node3D**, permite *instanciar* una malla (**Mesh**) en una posición, orientación y escalas dadas en el espacio 3D.

Creación de mallas en GDScript

En las siguientes transparencias vemos diversos ejemplos de creación de mallas con GDScript:

- En primer ejemplos en 2D y después en 3D.
- Usando mallas no indexadas y mallas indexadas.
- Los envíos son en modo diferido o inmediato.
- En algunos ejemplos se usa una tabla de colores por vértice (se interpola el color).
- En 3D se supone que podemos calcular las normales (no se muestra cómo) y asignar material a las superficies.

Malla no indexada 2D, color plano (1/3)

Ejemplo de inicialización de un `MeshInstance2D` para visualizar dos triángulos no adyacentes (6 vértices) con un color plano (sin atributos de color por vértice).

```
extends MeshInstance2D

func _ready():

## declarar y dimensionar un array para las tablas
var tablas : Array = []
tablas.resize( Mesh.ARRAY_MAX )

## añadir la tabla con las coordenadas de posición de 6 vértices
## .... (siguiente transparencia) ....

## cambiar el color de la malla (atributo 'modulate' del nodo)
modulate = Color( 1.0, 0.5, 0.5 )

## inicializar el atributo 'mesh' de este nodo
mesh = ArrayMesh.new() ## crea malla en modo diferido, vacía
mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )
```

Sesión 2: El engine Godot. Mallas.

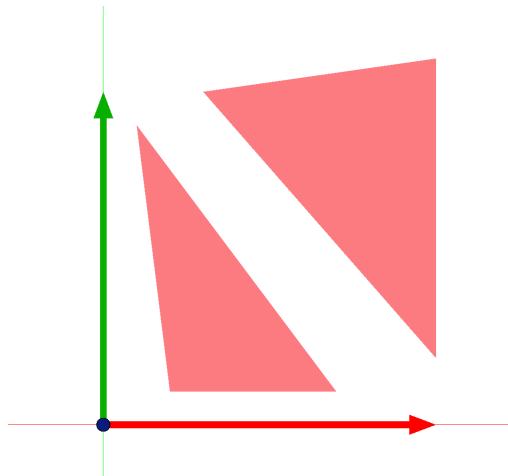
Created 2025-12-01

Page 57 / 100.

2. Mallas en Godot..
2.5. Mallas en 2D..

Malla no indexada 2D, color plano (3/3)

Aquí vemos los dos triángulos en el plano 2D (se han añadido los ejes de coordenadas para tener una referencia de la escala y posición):



Malla no indexada 2D, color plano (2/3)

La creación de la tabla de posiciones (3 vértices) se puede hacer directamente creando un `PackedVector2Array` con las coordenadas de los vértices y situandolo en la entrada correspondiente del array `tablas`:

```
## añadir la tabla con las coordenadas de posición de 6 vértices:
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([
    Vector2( 0.2, 0.1 ), Vector2( 0.7, 0.1 ), Vector2( 0.1, 0.9 ),
    Vector2( 0.3, 1.0 ), Vector2( 1.0, 0.2 ), Vector2( 1.0, 1.1 ),
])
```

Otra posibilidad es usar `push_back` (permitirá algoritmos complejos):

```
## añadir la tabla con las coordenadas de posición de 6 vértices:
var posv := PackedVector2Array() # crear array posic. verts. vacío

posv.push_back(Vector2(0.2,0.1)); posv.push_back(Vector2(0.7,0.1))
posv.push_back(Vector2(0.1,0.9)); posv.push_back(Vector2(0.3,1.0))
posv.push_back(Vector2(1.0,0.2)); posv.push_back(Vector2(1.0,1.1))

tablas[ Mesh.ARRAY_VERTEX ] = posv # asignar la tabla de posiciones
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 58 / 100.

2. Mallas en Godot..
2.5. Mallas en 2D..

Mallas con colores de vértices en 2D (1/3)

En este ejemplo se usa una tabla de colores de vértices, lo cual hace que esos colores se interpolen en el interior de los triángulos. No es necesario actualizar `modulate`

```
extends MeshInstance2D

func _ready():

## declarar y dimensionar un array para las tablas
var tablas : Array = []
tablas.resize( Mesh.ARRAY_MAX )

## añadir las tablas con posiciones y colores de 6 vértices
## .... (siguiente transparencia) ....

## inicializar el atributo 'mesh' de este nodo
mesh = ArrayMesh.new() ## crea malla en modo diferido, vacía
mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )
```

Mallas con colores de vértices en 2D (2/3)

La tabla de colores se debe proporcionar como un `PackedColorArray`, con un color por cada vértice. Por ejemplo:

```
## añadir las tablas con posición y colores de 6 vértices
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([ ... ]) # igual
tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
    Color( 1.0,0.0,0.0 ), Color( 0.0,1.0,0.0 ), Color( 0.0,0.0,1.0 ),
    Color( 1.0,1.0,0.0 ), Color( 0.0,1.0,1.0 ), Color( 1.0,0.0,1.0 ),
])
```

También es posible inicializar la tabla de colores con `push_back`:

```
var posv := PackedVector2Array() # crear array posic. verts. vacío
var colv := PackedColorArray() # crear array colores vértices vacío

posv.push_back(Vector2(0.2,0.1)); colv.push_back(Color(1.0,0.0,0.0))
# .... posiciones y colores del resto de vértices ...

tablas[ Mesh.ARRAY_COLOR ] = colv # asignar la tabla de colores
tablas[ Mesh.ARRAY_VERTEX ] = posv # asignar la tabla de posiciones
```

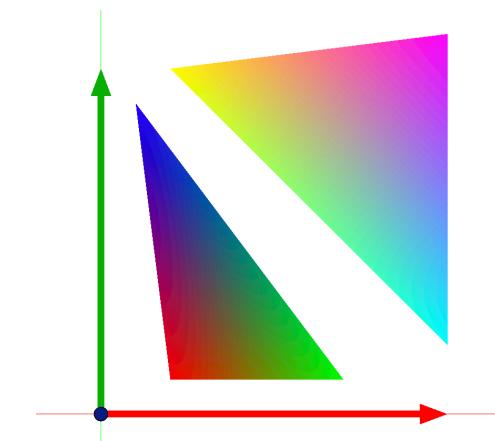
Mallas indexadas en 2D (1/2)

Malla de 4 vértices, que forman dos triángulos adyacentes (3×2 índices).

```
extends MeshInstance2D
func _ready():
    var tablas : Array = [] ## array con tablas de atributos
    tablas.resize( Mesh.ARRAY_MAX ) ## redimensionar el array
    ## añadir las tablas: posiciones y colores
    tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([
        Vector2( 0.2, 0.1 ), Vector2( 0.7, 0.2 ),
        Vector2( 0.3, 1.1 ), Vector2( 1.2, 1.1 ),
    ])
    tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
        Color( 1.0, 0.0, 0.0 ), Color( 0.0, 1.0, 0.0 ),
        Color( 0.0, 0.0, 1.0 ), Color( 1.0, 1.0, 0.0 )
    ])
    ## definir la tabla de índices (6 índices)
    tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([ 0,1,2, 2,1,3 ])
    ## crear un 'Mesh' en modo diferido y añadirle las tablas
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )
```

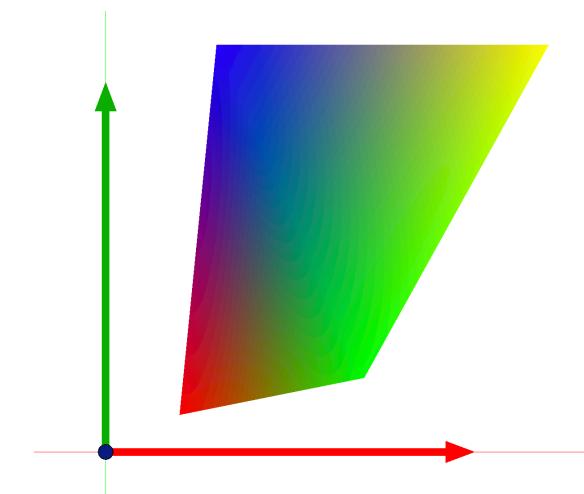
Malla no indexada 2D, color plano (3/3)

Aquí vemos los dos triángulos con las gradaciones de colores. En el interior de los triángulos se realiza una interpolación lineal cuyos valores extremos son los colores de los vértices:



Mallas indexadas en 2D (2/2)

Aquí vemos los dos triángulos adyacentes, con interpolación de colores entre los 4 colores:



Mallas 2D indexada con texturas (1/2)

Creamos las coordenadas de textura y asignamos el atributo `texture`

```
extends MeshInstance2D

func _ready():
    ## definir tablas para 4 vértices formando 2 triángulos:
    var tablas : Array = [] ; tablas.resize( Mesh.ARRAY_MAX )
    tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([ ... ]) # igual
    tablas[ Mesh.ARRAY_TEX_UV ] = PackedVector2Array([
        Vector2( 0.0,0.0 ), Vector2( 1.0,0.0 ),
        Vector2( 0.0,1.0 ), Vector2( 1.0,1.0 )
    ])
    tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([ 0,1,2, 2,1,3 ])

    ## crear el objeto 'mesh'
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

    ## cargar la textura y asignarla a este objeto
    texture = CargarTextura( "imgs/madera2.jpg" )
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 65 / 100.

Carga de texturas en GDScript

La función `CargarTextura` carga una imagen almacenada en un archivo y crea un objeto `ImageTexture` a partir de ella:

```
func CargarTextura( arch : String ) -> ImageTexture :

    ## crear un objeto 'Image' con la imgen
    var imagen := Image.new()
    assert( imagen.load(arch) == OK, "Error cargando '"+arch+"'." )

    ## crear un objeto 'ImageTexture' a partir del objeto 'Image'
    var textura := ImageTexture.create_from_image( imagen )
    print("Textura cargada desde archivo: '",arch,"'.")

    ## devolver la textura
    return textura
```

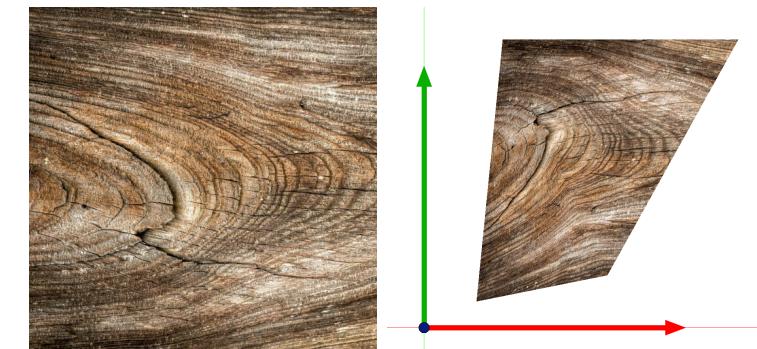
Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 67 / 100.

Mallas 2D indexada con texturas (2/2)

Aquí vemos la textura y resultado de aplicarla a los dos triángulos:



Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 66 / 100.

Envío en modo inmediato en 2D (1/2)

En cada frame se define de nuevo la malla no indexada con 1 triángulo. Ahora usamos el método `_process`, que se ejecuta en cada frame. En este ejemplo, se calcula un vector "d" que varía con el tiempo y luego se redefine la malla.

```
extends MeshInstance2D

var t : float = 0 # tiempo total desde inicio en segundos.

func _ready():
    mesh = ImmediateMesh.new() # crear 'Mesh' vacío al inicio.

func _process( delta: float ):

    # actualiza tiempo transcurrido
    t = t+delta

    # calcular un vector 'd' que va cambiando con el tiempo
    var d := 0.2*Vector2( cos(4.0*t), sin(4.0*t) )

    ## volver a definir las tablas
    ## ... (ver siguiente transparencia) ....
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 68 / 100.

Envío en modo inmediato en 2D (2/2)

Para redefinir en cada frame la malla, se usan los métodos de `ImmediateMesh` que permiten ir añadiendo cada vértice y sus atributos a la malla. Para cada vértice se especifica su posición después de haber fijado sus otros atributos:

```
## volver a definir las tablas
mesh.clear_surfaces(). ## limpia el mesh
mesh.surface_begin( Mesh.PRIMITIVE_TRIANGLES ) # dar tipo de primitiva

## definir vértice 0
mesh.surface_set_color( Color(1.0,0.0,0.0) ). 
mesh.surface_add_vertex_2d( Vector2(0.2,0.2) + d ) ## usa 'd' animado
## definir vértice 1
mesh.surface_set_color( Color(0.0,1.0,0.0) ) ## opcional: si no conserva
mesh.surface_add_vertex_2d( Vector2(1.0,0.5) )
## definir vértice 2
mesh.surface_set_color( Color(0.0,0.0,1.0) ) ## opcional: si no conserva
mesh.surface_add_vertex_2d( Vector2(0.5,1.0) )

mesh.surface_end() # fin de los vértices
```

Malla indexada 3D

En 3D es necesario definir un *material* para la malla. Un *material* es un objeto que determina como los triángulos reflejan la luz proveniente de las fuentes de luz.

```
extends MeshInstance3D

func _ready():
    ## declarar y dimensionar un array para las tablas (igual que en 2D)
    var tablas : Array = []
    tablas.resize( Mesh.ARRAY_MAX )

    ## añadir las tablas posiciones, colores e índices de un triángulo
    ## .... (ver siguiente transparencia) ....

    ## inicializar el atributo 'mesh' de este nodo (igual que en 2D)
    mesh = ArrayMesh.new() ## crea malla en modo diferido, vacía
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

    ## asignarle un material sin iluminación ni sombreado
    ## ... (ver siguientes transparencias) ....
```

Subsección 2.6. Mallas en 3D.

Malla indexada 3D: tablas

La creación de la tabla de posiciones (3 vértices) se puede hacer directamente creando un `PackedVector3Array`. Los colores y los índices (si están) se crean igual que en 2D.

Aquí vemos la creación de una malla con un único triángulo:

```
## añadir las tablas posiciones, colores e índices de un triángulo
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector3Array([
    Vector3(0.6,0.6,0.1), Vector3(0.1,0.8,0.1), Vector3(0.2,0.1,0.1)
])
tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
    Color( 1, 0, 0 ), Color( 0, 1, 0 ), Color( 0, 0, 1 )
])
tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([
    0, 1, 2
])
```

Al igual que en 2D, cualquiera de estas tablas **se puede crear usando `push_back`** en lugar de inicializarla directamente.

Malla indexada 3D: material con colores de vértices (1/2)

En este ejemplo se añade un material sin iluminación ni sombreado, quiere decir que los colores de los triángulos no se ven afectados por las fuentes de luz presentes en la escena.

```
# asignarle un material sin iluminación ni sombreado
var mat := StandardMaterial3D.new() # crear objeto 'StandardMaterial3D'

## configurar el objeto con el material ('mat')
mat.vertex_color_use_as_albedo = true # usar colores de vértices
mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED # sin sombr.
mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado

## redefinir atributo 'material_override' del nodo 'MeshInstance3D'
material_override = mat
```

Se desactiva el cribado de caras para que se vean todos los triángulos independientemente de en qué orden se especifiquen sus tres vértices.

Malla indexada 3D: material con color plano (1/2)

Si no hay colores de vértices y queremos un color plano, el material se debe configurar definiendo el atributo `albedo_color` del material (y poniendo a `false` el atributo `vertex_color_use_as_albedo`):

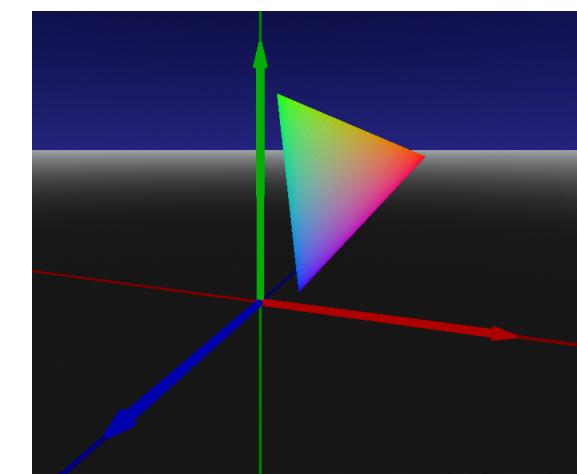
```
# asignarle un material sin iluminación ni sombreado
var mat := StandardMaterial3D.new() # crear objeto 'StandardMaterial3D'

## configurar el objeto con el material ('mat')
mat.vertex_color_use_as_albedo = false # no usar colores de vértices
mat.albedo_color = Color( 1.0, 0.4, 0.4 ) # color plano de la malla
mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED # sin sombr.
mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado
mat.lighting = false # no usar la luces

## redefinir atributo 'material_override' del nodo 'MeshInstance3D'
material_override = mat
```

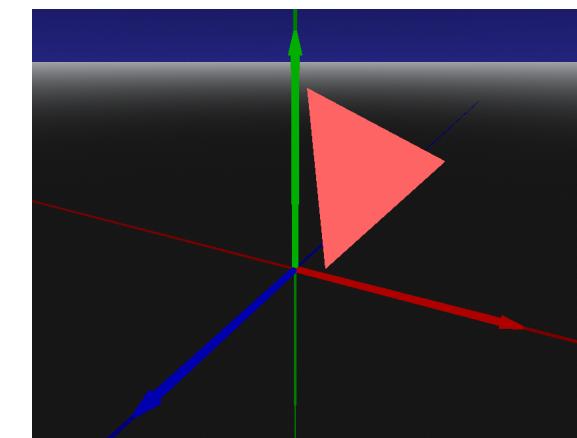
Mallas indexada 3D: : material con colores de vértices (2/2)

Aquí vemos el triángulo junto con unos ejes en 3D:



Mallas indexada 3D: : material con color plano (2/2)

Aquí vemos el triángulo con el color plano:



Sombreado con iluminación

En los ejemplos 3D anteriores, el material no tiene texturas ni responde a la iluminación

- En las muchas aplicaciones 3D se usan materiales que responden a la iluminación y además muchas veces tienen asociadas texturas.
- En Godot se pueden definir materiales complejos, con texturas y que responden a la iluminación, usando la clase `StandardMaterial3D` (o creando materiales personalizados con *shaders*).
- También se pueden usar texturas.
- Las texturas requieren definir una *tabla de coordenadas de textura*, que es un `Vector2` por cada vértice.
- La iluminación requiere que cada vértice tenga asociada un vector normal que determina la orientación de la superficie en ese vértice.

Especificación de un material con iluminación

La creación de un material con iluminación se puede hacer como se indica aquí:

```
var mat := StandardMaterial3D.new()

mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado
mat.albedo_color = Color( 1.0, 0.5, 1.0 ) # color base de la malla
mat.metallic = 0.25
mat.roughness = 0.15
material_override = mat
```

En caso de que se quiera usar una textura para el color base (en lugar de un color plano), se puede asignar el atributo `albedo_texture` en lugar de `albedo_color`:

```
mat.albedo_texture = CargarTextura( "imgs/grid1.png" )
```

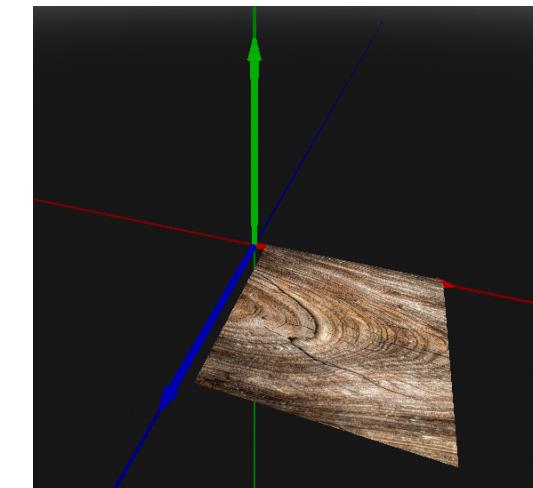
Especificación de normales y coordenadas de textura

En este ejemplo 3D se define una malla rectangular con dos triángulos, todos ellos con la misma normal (dirección Y+). Las coordenadas de textura se crean de forma que la imagen de textura se vea completa en el rectángulo:

```
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector3Array([
    Vector3( 0.1, 0.1, 0.1 ), Vector3( 0.9, 0.1, 0.1 ),
    Vector3( 0.9, 0.1, 0.9 ), Vector3( 0.1, 0.1, 0.9 )
])
tablas[ Mesh.ARRAY_NORMAL ] = PackedVector3Array([
    Vector3( 0.0, 1.0, 0.0 ), Vector3( 0.0, 1.0, 0.0 ),
    Vector3( 0.0, 1.0, 0.0 ), Vector3( 0.0, 1.0, 0.0 ),
])
tablas[ Mesh.ARRAY_TEX_UV ] = PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
    Vector2( 1.0, 1.0 ), Vector2( 0.0, 1.0 )
])
tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([
    0, 1, 2, 0, 2, 3
])
```

Material con iluminación y textura

Aquí vemos la malla de dos triángulos con iluminación y textura:



Uso de *SurfaceTool* para generar mallas en 3D

Los objetos de tipo **ArrayMesh** pueden, adicionalmente, crearse usando un objeto de tipo **SurfaceTool**.

- Un objeto **SurfaceTool** guarda las posiciones y otros atributos de los vértices de una malla.
- Se puede inicializar especificando el tipo de primitiva, y luego la posición y atributos de cada vértice, uno a uno.
- Permite crear arrays indexados o no indexados
- La ventaja frente a los otros métodos vistos reside en que **SurfaceTool** incluye métodos para:
 - ▶ Calcular automáticamente las normales (y las tangentes) de los vértices
 - ▶ Obtener la caja englobante de la malla.

Ejemplo de uso de *SurfaceTool* para malla no indexada

En este ejemplo se crea una malla no indexada con dos triángulos, y a todos los vértices se les asigna el mismo color, aunque se podría cambiar el color antes de cada **add_vertex** (solo muestro la creación de **mesh**):

```
var st = SurfaceTool.new()

st.begin( Mesh.PRIMITIVE_TRIANGLES ) ## iniciar vertices
st.set_color( Color( 1.0, 0.5, 0.5 ) ) ## color de siguientes vértices

st.add_vertex( Vector3( 0.1, 0.0, 0.1 ) ) ## añade vert. 0
st.add_vertex( Vector3( 0.9, 0.0, 0.1 ) ) ## añade vert. 1
st.add_vertex( Vector3( 0.1, 0.0, 0.9 ) ) ## añade vert. 2

st.add_vertex( Vector3( 1.0, 0.0, 0.2 ) ) ## añade vert. 3
st.add_vertex( Vector3( 0.2, 0.0, 1.0 ) ) ## añade vert. 4
st.add_vertex( Vector3( 1.1, 0.0, 1.1 ) ) ## añade vert. 5

mesh = st.commit() ## crea objeto 'ArrayMesh' y lo asigna
```

Ejemplo de uso de *SurfaceTool* para malla indexada

Ahora se añade una tabla de índices y se crea un malla indexada (2 triángulos adyacentes, 4 vértices):

```
var st = SurfaceTool.new()

st.begin( Mesh.PRIMITIVE_TRIANGLES ) ## iniciar vertices
st.set_color( Color( 0.0, 1.0, 1.0 ) ) ## color de siguientes vértices

## añadir los 4 vértices
st.add_vertex( Vector3( 0.1, 0.0, 0.1 ) ) ## añade vert. 0
st.add_vertex( Vector3( 0.9, 0.0, 0.1 ) ) ## añade vert. 1
st.add_vertex( Vector3( 0.9, 0.0, 0.9 ) ) ## añade vert. 2
st.add_vertex( Vector3( 0.1, 0.0, 0.9 ) ) ## añade vert. 3

## añadir los 6 índices (2 triángulos)
st.add_index(0) ; st.add_index(1) ; st.add_index(2) # 1er tri.
st.add_index(0) ; st.add_index(2) ; st.add_index(3) # 2º tri.

mesh = st.commit() ## crea objeto 'ArrayMesh' y lo asigna
```

Subsección 2.7.

Mallas indexadas de triángulos en 3D

Mallas Indexadas.

En gráficos (y sobre todo en visualización 3D), es común que una secuencia de vértices codifique una malla de triángulos que comparten vértices. A esas secuencias las llamamos **Mallas indexadas** (de triángulos). En la aplicación se suelen representar usando dos tablas:

- **Tabla de vértices:** tiene una entrada por cada vértice, incluye sus coordenadas
- **Tabla de triángulos:** tiene una entrada por triángulo, incluye los índices de sus tres vértices en la tabla anterior. Se puede considerar como una secuencia de valores enteros consecutivos en memoria.
- **Tablas de atributos:** por cada atributo puede haber una tabla de valores flotantes (colores, normales, coordenadas de textura, etc...). Tiene un número de entradas igual al número de vértices. Cada entrada puede ser una tupla de 2, 3 o 4 flotantes.

Estructura de datos

La tabla de índices en realidad se llama *tabla de triángulos* y (para n triángulos) tiene $3n$ índices de vértices (enteros sin signo), y la de vértices $3m$ valores reales:

Tabla Triángulos (n tri.)			Tabla Vértices (m verts.)			
$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	0	x_0	y_0	z_0
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$	1	x_1	y_1	z_1
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$	2	x_2	y_2	z_2
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$	3	x_3	y_3	z_3
:	:	:	4	x_4	y_4	z_4
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$	\vdots	\vdots	\vdots	\vdots
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$	$m-2$	x_{m-2}	y_{m-2}	z_{m-2}
			$m-1$	x_{m-1}	y_{m-1}	z_{m-1}

$0 \leq i_{jk} < m$

Generación y manipulación procedural de mallas indexadas en Godot

En Godot, las mallas indexadas se pueden crear y manipular en GDScript de diversas formas:

- Creando en cada frame la malla como **ImmediateMesh**, vértice a vértice.
- Usando al inicio **SurfaceTool** para crear un **ArrayMesh** vértice a vértice.
- Creando inicio arrays Godot (empaquetados o no) con atributos de vértices e índices, y luego creando con esas tablas objetos **ArrayMesh**.
- Usando la clase **MeshDataTool** para manipular arrays de vértices e índices a partir de un **ArrayMesh** ya existente. La clase **MeshDataTool** incorpora diversos algoritmos de manipulación de mallas y estructuras de datos auxiliares (por ejemplo, la tabla de aristas).

Ejemplo de algoritmos

Diversos ejemplos de algoritmos para mallas indexadas en 3D:

- Generación de mallas como superficies parámetricas (*B-splines*, *NURBS*, etc...)
- Generación de mallas de revolución (ver práctica 2)
- Cálculo de normales de superficies suaves (ver práctica 2).
- Cálculo de tangentes.
- Calculo de aristas para visualización, o bien como entrada de otros algoritmos.
- Subdivisión de caras para mejorar la calidad.
- Asignación procedural de coordenadas de texura.

Sección 3. Problemas

1. Problemas de creación de mallas 2D
2. Problemas de creación de mallas 3D

3. Problemas.
- 3.1. Problemas de creación de mallas 2D.

Introducción

Vemos problemas de mallas 2D en Godot. Para hacer los problemas, debes de:

1. Crear un proyecto nuevo en Godot.
2. En la escena principal, añadirle un nodo raíz de tipo **Node2D**
3. Descargar el archivo **raiz_problemas_2D.gd** de los materiales de teoría y situarlo en la carpeta del proyecto.
4. Adjuntar el script **raiz_problemas_2D.gd** al nodo raíz de la escena. Se encarga de dibujar los ejes, poner el fondo a blanco, y permitir interacción con el ratón.
5. Para cada problema 2D, crear un nodo hijo del nodo raíz de tipo **Node2D** y adjuntarle el script que resuelve el problema.
6. Ejecutar. Mediante el uso del ratón, se puede mover y hacer zoom en la vista 2D. Inicialmente esa vista está centrada en el origen y muestra un cuadrado de lado 2 (de -1 a 1 en ambos ejes). Para ver únicamente el nodo de un problema, en el árbol de escena haz todos los nodos no visibles excepto ese nodo.

Subsección 3.1. Problemas de creación de mallas 2D

3. Problemas.
- 3.1. Problemas de creación de mallas 2D.

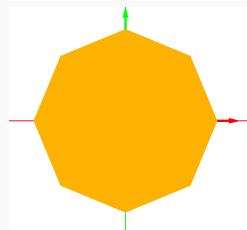
Polígono regular relleno de color plano

Problema 2.1:

Implementa un nodo de tipo **MeshInstance** con una malla (**no indexada**) para un polígono regular de **n** lados relleno de color naranja plano (RGB (1.0, 0.7, 0.0)), con radio **r** y centro en el origen (ver figura).

El polígono estará formado por **n** triángulos, cada uno con un vértice en el centro y los otros dos en el contorno. Los valores de **n** y **r** se declaran como dos constantes de GDScript (**const**), como se indica aquí:

```
const n : int = 8  
const r : float = 0.8
```



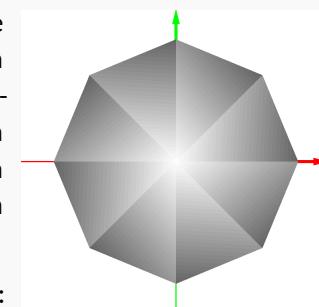
Los valores de estas constantes se podrán cambiar sin tocar nada del resto del script.

Polígono regular relleno con gradaciones

Problema 2.2:

Crea otro **Node2D**, y asígnale un script para visualizar el mismo polígono regular que antes (también con una malla **no indexada**), solo que ahora debes asignar colores a los vértices para que los triángulos aparezcan con una graduación en tonos de gris como en la figura. Cada triángulo que forma el polígono regular será blanco en el vértice del centro, gris claro en otro y gris oscuro en el tercero.

Responde razonadamente a esta cuestión:
 ¿ cuantos vértices debe tener la tabla de vértices ?



Polígono regular hecho de líneas

Problema 2.3:

Repite los dos problemas anteriores, con los mismos requerimientos, pero ahora usando **mallas indexadas**, de forma que el número de vértices e índices sea mínimo.

Responde razonadamente a estas cuestiones:

- ¿ cuantos vértices debe tener ahora la tabla de vértices en cada caso ?
- ¿ y cuantos índices debe haber ?

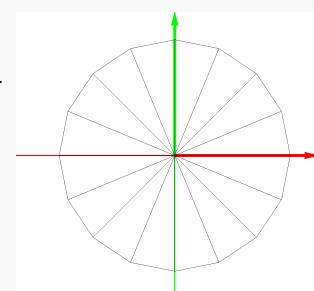
Aristas del polígono regular

Problema 2.4:

Crea un nuevo nodo **MeshInstance2D** de forma que ahora veamos simplemente las aristas del polígono regular descrito en los anteriores problemas. En la figura se ve para **n** a 16 y el mismo radio.

Considera dos casos:

- Usando una malla no indexada.
- Usando una malla indexadas.



Generación de malla con segmentos de normales

Problema 2.5:

Usando el código de la práctica 2, crea un script global (*autoload*) con una función que genere un objeto de tipo `MeshInstance3D` con una malla no indexada con los segmentos en las normales de una malla dada. La función tendrá la siguiente declaración:

```
func genSegNormales( verts, norms : PackedVector3Array,
    lon : float, color : Color ) -> MeshInstance3D :
```

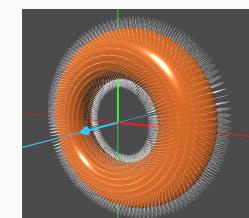
donde `verts` es la tabla de vértices de la malla, `norms` la tabla de normales, `lon` la longitud de los segmentos y `color` el color de los segmentos. Usa el tipo de primitiva *lineas*, y asegúrate de que a los segmentos no les afecta la iluminación.

(continua...)

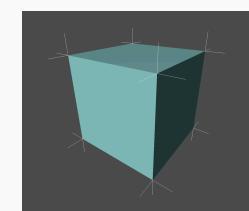
Generación de malla con segmentos de normales

Problema 2.5 (continuación):

Una vez tengas la función disponible, úsala en la función `_ready` de alguna malla (por ejemplo, el *Donut* o los cubos de la práctica 2), para añadir al objeto un nodo hijo con la malla de segmentos creada por la función.



Puedes capturar el evento de pulsación de la tecla N del objeto para activar y desactivar la visualización de las normales en ese objeto. Para ello, usa un valor lógico y el atributo de visibilidad de la malla de segmentos.



Generación de normales con ponderación por área

Problema 2.6:

Las normales de una malla indexada de triángulos se pueden calcular promediando en cada vértice las normales de los triángulos adyacentes al vértice (el código está en el guión de la práctica 2).

Sin embargo, una variante de este algoritmo consiste en modificar ese promedio, de forma que el peso de cada normal en el promedio sea proporcional al área del triángulo correspondiente a dicha normal, lo cual se supone que puede aproximar mejor la orientación real de la superficie en el vértice.

Usando el código de la práctica 2, escribe y prueba una variante del algoritmo de cálculo de normales que implemente este método. Puedes partir de una copia del código del guion de prácticas, haciendo las mínimas modificaciones o añadidos necesarios.

Fin de transparencias.

Informática Gráfica.

Sesión 3: Espacios y transformaciones afines.

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

Índice

Espacios afines y marcos de referencia	3
Transformaciones afines y matrices de transformación.	41
Transformaciones usuales en Informática Gráfica.	66
Transformaciones en Godot.	97

Sección 1.

Espacios afines y marcos de referencia

-
1. Las estructuras de espacio afín y espacio vectorial.
 2. Marcos afines y coordenadas homogéneas
 3. Producto escalar y vectorial de vectores.

Subsección 1.1.

Las estructuras de espacio afín y espacio vectorial.

Espacios afines

La noción abstracta de **Espacio Afín** es esencial en Informática Gráfica, ya que permite:

- Razonar sobre los puntos en el espacio,
- Diseñar e implementar representaciones y de esos puntos y estructuras de datos relacionadas en la memoria de los ordenadores, y
- Diseñar e implementar algoritmos que operan sobre esas representaciones y estructuras de datos.

Un espacio afín esencialmente es un modelo de los puntos de un espacio geométrico (de 1,2,3 o más dimensiones) y de las operaciones que podemos hacer con ellos. Está muy relacionado con el concepto de **Espacio Vectorial**

Axiomas del espacio vectorial

El espacio vectorial V_n cumple las siguientes propiedades, para cualquier $\vec{u}, \vec{v}, \vec{w} \in V_n$ y $a, b \in \mathbb{R}$:

1. Asociatividad de la suma: $(\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$
2. Comutatividad de la suma: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$
3. Elemento identidad de la suma: existe un elemento $\vec{0} \in V_n$ (llamado vector nulo) tal que $\vec{u} + \vec{0} = \vec{u}$.
4. Elemento opuesto de la suma: para cada $\vec{u} \in V_n$, existe su vector opuesto $-\vec{u} \in V_n$ tal que $\vec{u} + (-\vec{u}) = 0$
5. Elemento identidad del producto: $1\vec{u} = \vec{u}$
6. Asociatividad del producto: $a(b\vec{u}) = (ab)\vec{u}$
7. Distributividad de la suma de vectores y producto: $a(\vec{u} + \vec{v}) = a\vec{u} + a\vec{v}$
8. Distributividad de la suma de reales y producto: $(a + b)\vec{u} = a\vec{u} + b\vec{u}$

(identificar vectores con desplazamientos permite entender estos axiomas).

Estructura de Espacio Vectorial

Un **espacio vectorial** V_n (con n entero, > 0) es un conjunto de elementos llamados **vectores**, o **vectores libres**, y que notaremos con una flecha: $\vec{u}, \vec{v}, \vec{w}$, etc ..., los cuales interpretamos informalmente como flechas en el espacio, representando un **desplazamiento** desde el origen hasta el destino. El origen es indiferente, lo importante es la distancia y la dirección hasta el destino (por eso se llaman **vectores libres**). En V_n hay definidas estas **dos operaciones**:

Suma de vectores:

Para cualesquiera $\vec{u}, \vec{v} \in V_n$, existe un único $\vec{w} \in V_n$ tal que $\vec{w} = \vec{u} + \vec{v}$. El vector \vec{w} representa un desplazamiento equivalente a aplicar primero el desplazamiento \vec{u} y luego el desplazamiento \vec{v} (o al revés).

Producto por real:

Para cualquier $\vec{u} \in V_n$ y cualquier $a \in \mathbb{R}$, existe un $\vec{v} \in V_n$ tal que $\vec{v} = a\vec{u}$. El vector \vec{v} representa un desplazamiento en la misma dirección que \vec{u} , pero con una distancia multiplicada por a (si a es negativo, el sentido es el opuesto). Los vectores \vec{u} y $\vec{v} = a\vec{u}$ son **vectores paralelos**.

La estructura de Espacio Afín

Un **espacio afín** A_n (con n entero, > 0) sobre un espacio vectorial V_n es un conjunto de elementos llamados **puntos**, que notaremos con un punto sobre ellos $\dot{p}, \dot{q}, \dot{r}$, etc ..., los cuales interpretamos informalmente como posiciones en un espacio de n dimensiones. En A_n definimos esta operación:

Suma punto y vector:

Para cualquier punto $\dot{p} \in A_n$ y cualquier vector $\vec{v} \in V_n$, existe un punto $\dot{q} \in A_n$ tal que $\dot{q} = \dot{p} + \vec{v}$.

Intuitivamente, el punto \dot{q} representa la posición a la que se llega al desplazarse desde la posición \dot{p} siguiendo el desplazamiento representado por el vector \vec{v} .

Axiomas del espacio afín

El espacio afín A_n cumple las siguientes propiedades, para cualquier $\dot{p}, \dot{q}, \dot{r} \in A_n$ y cualquier $\vec{u}, \vec{v} \in V_n$:

1. Identidad de la suma: $\dot{p} + \vec{0} = \dot{p}$
2. Asociatividad: $(\dot{p} + \vec{v}) + \vec{w} = \dot{p} + (\vec{v} + \vec{w})$
3. Dados dos puntos \dot{p} y \dot{q} cualesquiera, existirá un único vector $\vec{v} \in V_n$ tal que $\dot{q} = \dot{p} + \vec{v}$.

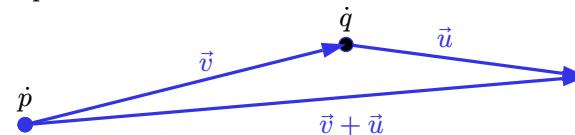
Los axiomas implican que podemos definir la operación:

Resta de puntos: para cualquiera dos puntos \dot{p} y \dot{q} se define el vector $\vec{v} = \dot{q} - \dot{p}$ como el único vector tal que $\dot{p} = \dot{q} + \vec{v}$.

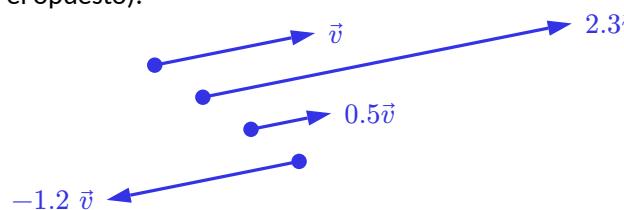
Lógicamente, se cumple $\dot{p} - \dot{p} = \vec{0}$.

Suma de vectores y multiplicación por escalar

Si \vec{v} traslada de \dot{p} a \dot{q} , y \vec{u} traslada de \dot{q} a \dot{r} , entonces el vector $\vec{v} + \vec{u}$ traslada directamente de \dot{p} a \dot{r} :

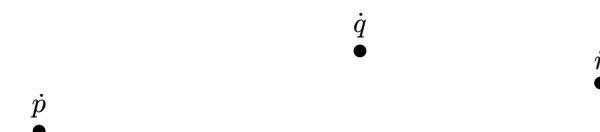


Dado un vector \vec{v} y un real a , el vector $a\vec{v}$ representa un desplazamiento en la misma dirección que \vec{v} , pero con una distancia multiplicada por a (si a es negativo, el sentido es el opuesto):

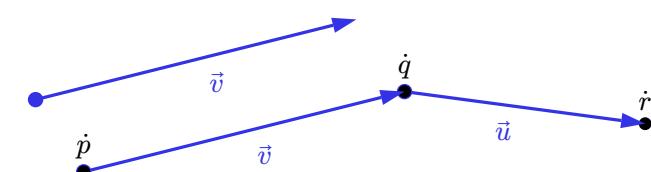


Ejemplos de puntos y vectores.

En este ejemplo vemos los puntos \dot{p}, \dot{q} y \dot{r} en el espacio afín 2D de la pantalla:



El vector libre $\vec{v} = \dot{q} - \dot{p}$, representa el desplazamiento desde \dot{p} hasta \dot{q} . Como representa un desplazamiento se puede situar con su extremo en cualquier sitio (lo dibujamos dos veces). Igualmente, el vector $\vec{u} = \dot{r} - \dot{q}$ es el desplazamiento desde \dot{q} hasta \dot{r} .



Rectas en el espacio afín.

Una **recta** que pasa por dos puntos distintos \dot{q} y \dot{r} se define como el conjunto de puntos \dot{p} tales que existe un $t \in \mathbb{R}$ tal que:

$$\dot{p} = \dot{q} + t(\dot{r} - \dot{q})$$

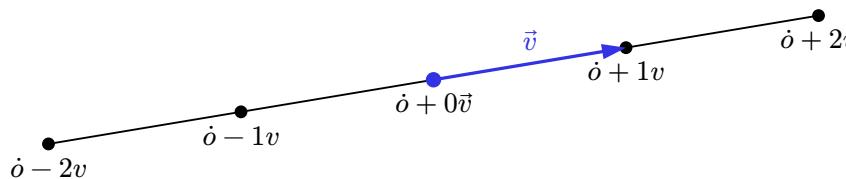
- A esta expresión se le denomina **ecuación paramétrica** de la recta, y el punto \dot{p} puede escribirse como $\dot{p}(t)$ para indicar su dependencia del parámetro t .
- Al vector $\vec{v} = \dot{q} - \dot{p}$ se le denomina **vector director** de la recta (se cumple $\|\vec{v}\| > 0$)
- La ecuación paramétrica permite identificar puntos en un recta con valores reales.
- El valor t puede interpretarse (informalmente) como la *distancia* entre \dot{r} y \dot{p} , medida en unidades de la longitud del vector $\dot{q} - \dot{p}$.
- Dos rectas **son paralelas** cuando sus vectores directores \vec{u} y \vec{v} son paralelos, es decir, si existe un real a tal que $\vec{u} = a\vec{v}$.

Ejemplo de recta en el espacio afín.

La ecuación paramétrica de una recta que pasa por \dot{o} y tiene como vector director \vec{v} es:

$$\dot{p}(t) = \dot{o} + t\vec{v}$$

Se pueden obtener todos los puntos de la recta dando valores a t , aquí vemos algunos puntos en posiciones correspondientes a t entero:



Si los valores de t están equiespaciados, los puntos $\dot{p}(t)$ también lo estarán.

Combinaciones afines de puntos

Los puntos **no pueden ser multiplicados por reales**, pero se puede definir la **combinación afín** $a\dot{p} + b\dot{q}$ de dos puntos \dot{p} y \dot{q} (con $a, b \in \mathbb{R}$ y $a + b$ vale 1 o 0):

- Cuando $a + b = 1$ la combinación afín se define como un **punto en la recta** que pasa por \dot{p} y \dot{q} , así:

$$a\dot{p} + b\dot{q} \equiv \dot{p} + b(\dot{q} - \dot{p})$$

- Cuando $a + b = 0$ la combinación se define como un **vector paralelo a la recta** que pasa por \dot{p} y \dot{q} , así:

$$a\dot{p} + b\dot{q} \equiv b(\dot{q} - \dot{p})$$

Puesto que \dot{p} y \dot{q} pueden ser el mismo punto, esto permite definir la **multiplicación de un punto \dot{p} por 1 o por 0**:

$$\begin{aligned} 1\dot{p} &\equiv 1\dot{p} + 0\dot{p} = \dot{p} + 0(\dot{p} - \dot{p}) = \dot{p} \\ 0\dot{p} &\equiv 0\dot{p} + 0\dot{p} = 0(\dot{p} - \dot{p}) = \vec{0} \end{aligned}$$

Bases de un espacio vectorial

Una **base** de un espacio vectorial V_n es un conjunto ordenado de n vectores $\{\vec{b}_0, \vec{b}_1, \dots, \vec{b}_{n-1}\}$, con $\vec{b}_i \in V_n$, que cumplen estas dos propiedades:

- Cualquier vector $\vec{v} \in V_n$ puede expresarse como **combinación lineal** de todos los vectores de la base, es decir, existe un única tupla $(a_0, a_1, \dots, a_{n-1}) \in \mathbb{R}^n$ tal que:

$$\vec{v} = a_0\vec{b}_0 + a_1\vec{b}_1 + \dots + a_{n-1}\vec{b}_{n-1}.$$

- Ningún vector de la base puede expresarse como combinación lineal del resto (son **linealmente independientes**).

Cualquier conjunto de n vectores linealmente independientes en V_n es una base del espacio. Se dice que n es la **dimensión** de V_n

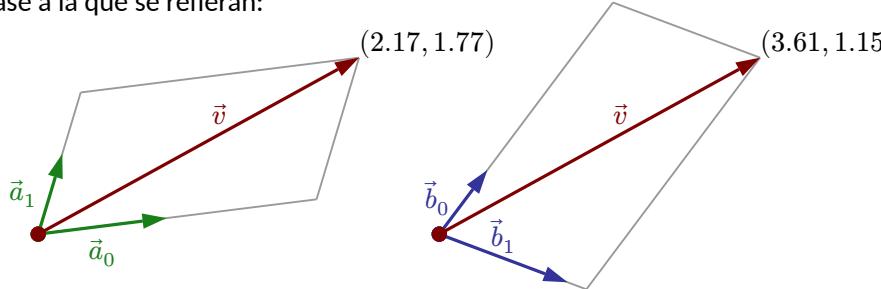
La elección de una base permite representar cualquier vector $\vec{v} \in V_n$ mediante sus **coordenadas** (a_1, a_2, \dots, a_n) , que siempre serán **relativas a dicha base**.

Ejemplos de bases y coordenadas en un espacio vectorial.

En 2D, vemos dos bases $A \equiv \{\vec{a}_0, \vec{a}_1\}$ (en verde) y $B \equiv \{\vec{b}_0, \vec{b}_1\}$ (en azul):



Un mismo vector \vec{v} (en rojo) tendrá dos pares de coordenadas distintas, según la base a la que se refieran:



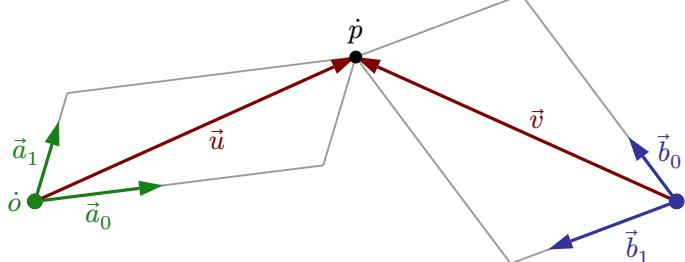
Ejemplos de marcos afines.

Vemos dos marcos afines $\mathcal{R} \equiv \{\vec{a}_0, \vec{a}_1, \dot{o}\}$ (en verde) y $B \equiv \{\vec{b}_0, \vec{b}_1, \dot{q}\}$ (en azul):



Un mismo punto \dot{p} tendrá dos pares de coordenadas distintas:

- $(2.25, 1.35)$ del vector $\vec{u} = \dot{p} - \dot{o}$ (relativas a $\{\vec{a}_0, \vec{a}_1\}$)
- $(3.22, 1.29)$ del vector $\vec{v} = \dot{p} - \dot{q}$ (relativas a $\{\vec{b}_0, \vec{b}_1\}$)



Marcos afines

Un **marco afín** (o *marco de referencia*, o *marco de coordenadas*, o *sistema de coordenadas*) \mathcal{R} en un espacio afín A_n es una tupla compuesta por los n vectores de una **base** de V_n y un punto \dot{o} del A_n (llamado **origen** del marco):

$$\mathcal{R} = (\vec{e}_0, \vec{e}_1, \dots, \vec{e}_{n-1}, \dot{o})$$

Fijado el marco afín \mathcal{R} , cualquier punto $\dot{q} \in A_n$ puede expresarse como:

$$\dot{q} = a_0 \vec{e}_0 + a_1 \vec{e}_1 + \dots + a_{n-1} \vec{e}_{n-1} + 1\dot{o}$$

e igualmente cualquier vector \vec{v} puede expresarse como:

$$\vec{v} = a_0 \vec{e}_0 + a_1 \vec{e}_1 + \dots + a_{n-1} \vec{e}_{n-1} + 0\dot{o}$$

donde a_0, a_1, \dots, a_{n-1} son n valores reales **únicos**, denominados las **coordenadas** del punto \dot{q} o del vector \vec{v} en el marco \mathcal{R} .

Coordenadas homogéneas y su espacio afín.

Las **coordenadas homogéneas** de un punto o un vector (relativos a un marco \mathcal{R}) forman un **vector columna** con $n + 1$ valores reales:

- Los n primeros valores son las coordenadas del punto o vector relativos a \mathcal{R}
- El último valor es 1 para puntos y 0 para vectores, se suele escribir w .

El conjunto de todas las posibles tuplas tiene una estructura de espacio afín:

- El subconjunto de las tuplas con $w = 1$ es un **espacio afín**, que llamamos A_n ya que dos de estas tuplas pueden restarse y se obtiene otra tupla con $w = 0$ (ya fuera de este subconjunto).
- El subconjunto de las tuplas con $w = 0$ es un **espacio vectorial**, y en concreto es el espacio vectorial asociado al espacio afín anterior, ya que estas tuplas pueden sumarse entre ellas y multiplicarse por un real.

Fijado un marco afín \mathcal{R} de A_n , hay una correspondencia biunívoca entre A_n y A_n

Correspondencia entre tuplas y puntos y vectores

Decimos que los puntos y vectores se obtienen **interpretando** sus coordenadas homogéneas en un marco $\mathcal{R} = (\vec{e}_0, \dots, \vec{e}_{n-1}, \dot{o})$, en el sentido siguiente:

- Si $\mathbf{u} = (a_0, \dots, a_{n-1}, 1)^T$ son las coordenadas de \vec{p} en \mathcal{R} , entonces podemos escribir la igualdad $\vec{p} = \mathcal{R}\mathbf{u}$, ya que

$$\vec{p} = 1\dot{o} + \sum_0^{n-1} a_i \vec{e}_i = (\vec{e}_0, \vec{e}_1, \dots, \vec{e}_{n-1}, \dot{o}) \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \\ 1 \end{pmatrix} = \mathcal{R}\mathbf{u}.$$

- Si $\mathbf{v} = (a_0, \dots, a_{n-1}, 0)^T$ son las coordenadas de \vec{v} en \mathcal{R} , entonces podemos escribir la igualdad $\vec{v} = \mathcal{R}\mathbf{v}$, ya que:

$$\vec{v} = 0\dot{o} + \sum_0^{n-1} a_i \vec{e}_i = (\vec{e}_0, \vec{e}_1, \dots, \vec{e}_{n-1}, \dot{o}) \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \\ 0 \end{pmatrix} = \mathcal{R}\mathbf{v}$$

Operaciones con las tuplas de coordenadas

Un marco afín \mathcal{R} en un espacio afín A_n puede verse como una correspondencia 1 a 1 entre A_n hacia A_n , ya que permite, a partir de una tupla $\mathbf{c} \in A_n$, obtener su correspondiente punto o vector $\mathcal{R}\mathbf{c} \in A_n$ (y al revés).

- Esa aplicación es **lineal** en el sentido de que **conserva las operaciones de los espacios afines**, es decir para cualquiera tuplas, \mathbf{u}, \mathbf{v} (en V_n), y \mathbf{p} (en A_n), y real a se cumple:

$$\mathcal{R}(\mathbf{p} + \mathbf{u}) = \mathcal{R}\mathbf{p} + \mathcal{R}\mathbf{u}$$

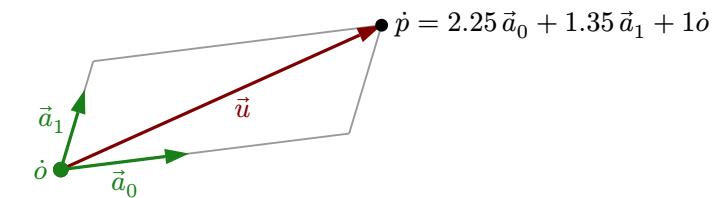
$$\mathcal{R}(\mathbf{u} + \mathbf{v}) = \mathcal{R}\mathbf{u} + \mathcal{R}\mathbf{v}$$

$$\mathcal{R}(a\mathbf{u}) = a(\mathcal{R}\mathbf{u})$$

- Esto implica que podemos **usar las operaciones con las tuplas para hacer cálculos con puntos y vectores**, esto permite implementar algoritmos que operan en espacios afines o vectoriales.

Ejemplos de coordenadas en un marco 2D

Consideramos el marco afín $\mathcal{R} \equiv \{\vec{a}_0, \vec{a}_1, \dot{o}\}$ y el punto \vec{p} que hemos visto antes.



Como el punto \vec{p} tiene coordenadas homogéneas $\mathbf{c} \equiv (2.25, 1.35, 1)^T$ (un vector columna) en el marco \mathcal{R} , por tanto podemos escribir esta igualdad:

$$\vec{p} = 2.25 \vec{a}_0 + 1.35 \vec{a}_1 + 1\dot{o} = (\vec{a}_0, \vec{a}_1, \dot{o}) \begin{pmatrix} 2.25 \\ 1.35 \\ 1 \end{pmatrix} = \mathcal{R}\mathbf{c}$$

Ventajas de las coordenadas homogéneas

Las coordenadas homogéneas en A_n se pueden usar para realizar cálculos con puntos y vectores en un ordenador, ya que las usamos como representaciones de los puntos y vectores abstractos de A_n .

El uso de las coordenadas homogéneas **simplifica muchísimo los cálculos con puntos y vectores en Informática Gráfica**:

- Permite representar de forma similar tanto los puntos como los vectores.
- Permite implementar la *transformaciones afines* con matrices.
- Simplifica los cálculos de proyección perspectiva.

Es importante recordar siempre que unas coordenadas no tienen sentido de forma independiente del marco de coordenadas al cual son relativas.

Marcos afines en IG

En Informática Gráfica se usan diversos marcos de coordenadas distintos:

- Marco de mundo: global a una escena.
- Marco de objeto: específico de un objeto concreto dentro de la escena.
- Marco de cámara: para coordenadas relativas a una cámara virtual posicionada y orientada de alguna forma concreta dentro de una escena.
- Marco del dispositivo: para coordenadas en una ventana o una imagen, en unidades de pixels.
- Marco normalizado de dispositivo: para coordenadas en una ventana o una imagen, en unidades normalizadas entre -1 y $+1$.

El resultado anterior nos indica que **podemos usar matrices para convertir las coordenadas relativas a un marco en las coordenadas relativas a otro marco**, lo cual es esencial en Informática Gráfica.

La base especial W_n de un espacio afín A_n

En Informática Gráfica, si representamos objetos reales usando un espacio afín abstracto A_n , entonces necesitamos trasladar a A_n los conceptos de *distancia* y *perpendicularidad* que usamos en el espacio físico real. Para ello, designamos una base de V_n llamada la **base especial**

$$W_n = (\hat{e}_0, \hat{e}_1, \dots, \hat{e}_{n-1})$$

cuyos vectores tienen **longitud unidad** y son perpendiculares dos a dos **por definición**. Usamos $\hat{\cdot}$ en lugar de $\vec{\cdot}$ para estos vectores, y se les llama **versores**. En 2D escribiremos $W_2 = (\hat{x}, \hat{y})$, y en 3D $W_3 = (\hat{x}, \hat{y}, \hat{z})$. Una vez seleccionada W_n

- podemos definir la *distancia* en A_n , (ya que W_n define las unidades de distancia en todas las direcciones posibles),
- podemos decir si dos vectores son perpendiculares o no,
- definimos el *ángulo* entre dos vectores no nulos, y
- podemos decir cuando un marco afín es un *marco cartesiano* o no lo es.

Subsección 1.3.

Producto escalar y vectorial de vectores.

Producto escalar de vectores

El **producto escalar (dot product)** es una función que se aplica dos vectores $\vec{u}, \vec{v} \in V_n$ y produce un valor real que se nota como $\vec{u} \cdot \vec{v}$.

El producto escalar cumple estas propiedades:

- Comutativa: $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$
- Distributiva respecto a la suma: $\vec{u} \cdot (\vec{v} + \vec{w}) = \vec{u} \cdot \vec{v} + \vec{u} \cdot \vec{w}$
- Asociativa respecto al producto por reales: $(a\vec{u}) \cdot \vec{v} = a(\vec{u} \cdot \vec{v})$
- Positivo: $\vec{u} \cdot \vec{u} \geq 0$

Muchas posibles definiciones distintas del producto escalar pueden cumplir estas propiedades, así que necesitamos también exigir que para dos vectores \hat{e}_i y \hat{e}_j de la base W_n se cumpla:

$$\hat{e}_i \cdot \hat{e}_j \equiv \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

Longitud de un vectores. Versores.

La **norma** (o **módulo**, o **longitud**) de un vector $\vec{u} \in V_n$ es el número real no negativo que se define como:

$$\|\vec{u}\| \equiv \sqrt{\vec{u} \cdot \vec{u}}$$

- La norma de un vector es siempre un valor real no negativo, y representa la longitud del desplazamiento que representa el vector.
- La **distanza** entre dos puntos \vec{p} y \vec{q} se define como el valor real $\|\vec{q} - \vec{p}\|$ (es decir, la longitud del vector que los une).
- Si sabemos que un vector tiene longitud 1, se le denomina **versor** (o **vector unitario**), y se escribe con un gorro: $\hat{x}, \hat{y}, \hat{z}, \hat{e}, \hat{a}$ etc...
- Los vectores de la base W_n son unitarios, y por eso los hemos escrito con un gorro.

Perpendicularidad y ángulo entre vectores

Una vez bien definido el producto escalar, podemos definir la noción de **perpendicularidad** y el **ángulo** entre dos vectores \vec{u} y \vec{v} (ninguno nulo):

- Si se cumple que $\vec{u} \cdot \vec{v} = 0$, entonces se dice que los vectores \vec{u} y \vec{v} son **perpendiculares** (u **ortogonales**).
- El **ángulo** θ entre \vec{u} y \vec{v} es un valor real (en radianes), en el rango $[0, \pi]$, se define como:

$$\theta \equiv \arccos\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \|\vec{v}\|}\right),$$

y por tanto, para dos versores \hat{a} y \hat{b} con ángulo θ entre ellos se cumple:

$$\cos(\theta) = \hat{a} \cdot \hat{b}.$$

- Los versores de W_n son **perpendiculares** dos a dos por la definición de W_n .

Producto vectorial en 3D

Consideramos ahora el espacio vectorial de las tuplas de 4 componentes con $w = 0$ (son coordenadas de vectores en 3D). En ese espacio se puede definir la operación de **producto vectorial (cross product)** entre dos vectores \vec{u}, \vec{v} , que produce un otro vector $\vec{w} = \vec{u} \times \vec{v}$. Esta operación cumple estas propiedades:

- **Anticomutativa**: $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$
- **Distributiva** respecto a la suma: $\vec{u} \times (\vec{v} + \vec{w}) = \vec{u} \times \vec{v} + \vec{u} \times \vec{w}$
- **Asociativa** respecto al producto por reales: $(a\vec{u}) \times \vec{v} = a(\vec{u} \times \vec{v})$
- **Perpendicularidad**: se cumple $\vec{u} \cdot (\vec{u} \times \vec{v}) = 0$, implica que $\vec{u} \times \vec{v}$ es perpendicular a \vec{u} y a \vec{v} .

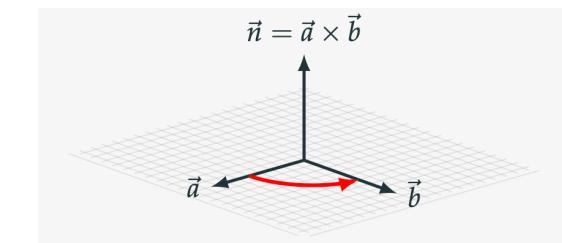
Hay muchas posibles definiciones del producto vectorial que cumplen estas propiedades. Para definirlo correctamente se exige que si $W_3 = (\hat{x}, \hat{y}, \hat{z})$, entonces se cumpla:

$$\hat{x} \times \hat{y} = \hat{z} \quad \hat{y} \times \hat{z} = \hat{x} \quad \hat{z} \times \hat{x} = \hat{y}$$

Dirección y sentido del producto vectorial en 3D

El producto vectorial constituye un método para obtener un vector perpendicular a otros dos vectores dados (no paralelos):

- El vector $\vec{n} \equiv \vec{a} \times \vec{b}$ es perpendicular al plano que forman \vec{a} y \vec{b} (y por lo tanto, perpendicular tanto a \vec{a} como a \vec{b})



- La dirección de \vec{n} es la dirección en la que avanza un tornillo (paralelo a \vec{n} y con la rosca hacia arriba, en el sentido en la que apunta \vec{n}) cuando se gira desde \vec{a} hacia \vec{b} .

Marcos ortogonales y ortonormales

Dado un marco $\mathcal{R} = (\vec{b}_0, \dots, \vec{b}_n, \dot{o})$, decimos que ese marco es:

Ortogonal: si los vectores de la base son perpendiculares dos a dos, es decir:

$$i \neq j \implies \vec{b}_i \cdot \vec{b}_j = 0$$

Ortonormal: si es ortogonal y además todos los vectores de la base son unitarios (son versores), es decir:

$$\vec{b}_i \cdot \vec{b}_i = 1$$

Definimos la **matriz de productos escalares** M asociada a \mathcal{R} como la matriz con $a_{ij} \equiv \vec{b}_i \cdot \hat{e}_j$, donde cada \hat{e}_i es un versor de W_n . Entonces se cumple:

- Si \mathcal{R} es ortogonal, M también lo es.
- Si \mathcal{R} es ortonormal, M también lo es (tendrá determinante $+1$ o -1).

Orientación de un marco. Marcos cartesianos.

Un marco \mathcal{R} cualquiera puede tener orientación a **derechas** o a **izquierdas**, la orientación depende de la base de \mathcal{R} , en concreto del signo del determinante de la matriz M de productos escalares asociada a \mathcal{R}

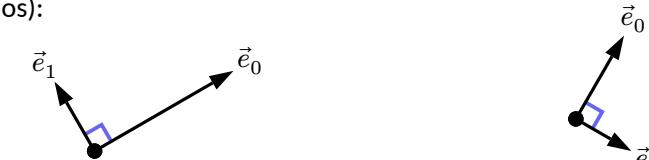
- Si es positivo, será a **derechas** (W_n es a derechas por **definición**).
- Si es negativo, será a **izquierdas**.

Un marco \mathcal{R} es **cartesiano** si y solo si es ortonormal y además tiene orientación a derechas (es decir: el determinante de M es $+1$).

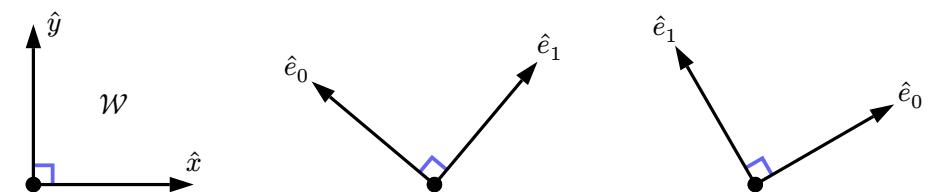
- Eso implica que los vectores en la base de \mathcal{R} se pueden hacer coincidir con los de W_n mediante una misma **rotación** aplicada a todos ellos.
- Con esta definición, un marco cuya base es W_n es **cartesiano** por definición, pero en un espacio afín hay otros muchos otros marcos también cartesianos.

Ejemplos de marcos ortogonales y ortonormales en 2D

Estos marcos son ortogonales pero no ortonormales (los vectores de la base no son unitarios):

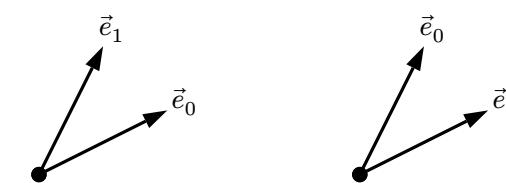


Estos marcos son ortonormales (los dos vectores de la base son unitarios y perpendiculares entre ellos). El marco W (a la izquierda) es ortonormal por definición:

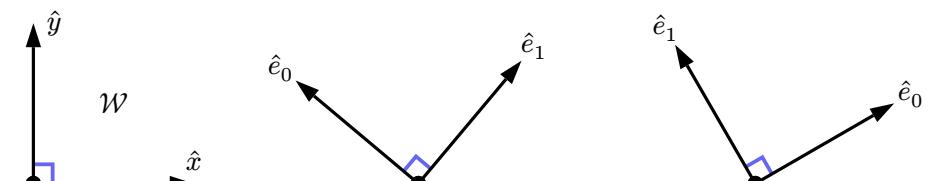


Ejemplos de orientación de marcos en 2D

Dos marcos 2D con las dos orientaciones posibles (a izquierdas y a derechas):

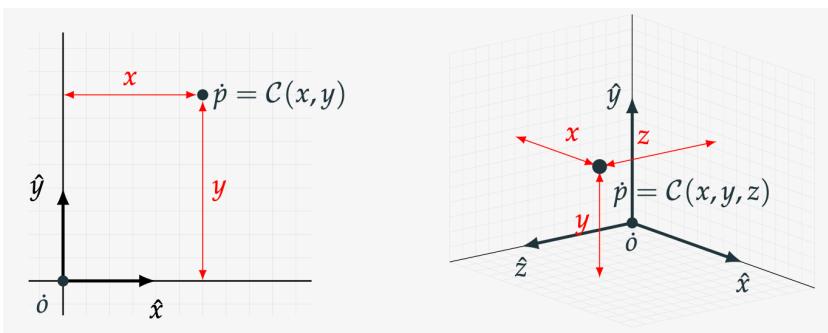


El marco W es a **derechas** (por definición). El marco del centro es a **izquierdas** (\hat{e}_0 a la izquierda de \hat{e}_1) y el de la derecha es a **derechas** (\hat{e}_0 a la derecha de \hat{e}_1)



Propiedades de las coordenadas cartesianas

Las coordenadas de un punto relativos a un marco cartesiano se llaman **coordenadas cartesianas**:



En un marco cartesiano \mathcal{C} :

- Si es 2D (izquierda), las coordenadas (x, y) son las distancias a las dos rectas.
- Si es 3D (derecha), las coordenadas (x, y, z) son las distancias a los tres planos.

Cálculo de producto vectorial con coordenadas cartesianas

En el espacio vectorial de las tuplas con $w = 0$ y $n = 3$ (coordenadas de vectores en 3D), el producto vectorial de dos tuplas de coordenadas $\mathbf{u} = (x_0, y_0, z_0, 0)$ y $\mathbf{v} = (x_1, y_1, z_1, 0)$ se escribe como $\mathbf{u} \times \mathbf{v}$ y se define como:

$$\mathbf{u} \times \mathbf{v} = \begin{pmatrix} y_0 z_1 - z_0 y_1 \\ z_0 x_1 - x_0 z_1 \\ x_0 y_1 - y_0 x_1 \end{pmatrix}$$

Si \mathcal{C} es un marco cartesiano, y \mathbf{u} y \mathbf{v} las coordenadas en \mathcal{C} de dos vectores \vec{u} y \vec{v} , entonces el producto vectorial de los vectores se puede calcular fácilmente usando el producto escalar de sus coordenadas, es decir, se cumple:

$$\vec{u} \times \vec{v} = (\mathcal{C}\mathbf{u}) \times (\mathcal{C}\mathbf{v}) = \mathcal{C}(\mathbf{u} \times \mathbf{v})$$

Esto se cumple para cualquier marco cartesiano, es decir, **el producto vectorial de coordenadas es invariante entre marcos cartesianos**.

Cálculo de producto escalar con coordenadas cartesianas

En el espacio vectorial de las tuplas con $w = 0$, el producto escalar de dos tuplas de coordenadas (de vectores) $\mathbf{u} = (a_0, \dots, a_{n-1}, 0)^T$ y $\mathbf{v} = (b_0, \dots, b_{n-1}, 0)^T$ se escribe como $\mathbf{u} \cdot \mathbf{v}$ y es igual a la suma del producto componente a componente:

$$\mathbf{u} \cdot \mathbf{v} = \sum_0^{n-1} a_i b_i$$

Si \mathcal{C} es un marco cartesiano, y \mathbf{u} y \mathbf{v} las coordenadas en \mathcal{C} de dos vectores \vec{u} y \vec{v} , entonces el producto escalar de los vectores se puede calcular fácilmente usando el producto escalar de sus coordenadas, es decir, se cumple:

$$\vec{u} \cdot \vec{v} = (\mathcal{C}\mathbf{u}) \cdot (\mathcal{C}\mathbf{v}) = \mathbf{u} \cdot \mathbf{v}$$

Esto se cumple para cualquier marco cartesiano, es decir, el producto escalar de coordenadas es invariante entre marcos cartesianos.

Problemas: cálculo del prod. escalar y vectorial.

Problema 3.1:

Demuestra que efectivamente el producto escalar de dos vectores se puede calcular (usando sus coordenadas en cualquier marco cartesiano) como la suma del producto componente a componente. Usa las propiedades que definen dicho producto escalar.

Problema 3.2:

Demuestra que el producto vectorial de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano según se ha indicado.

Problema 3.3:

Demuestra que el producto vectorial de dos vectores es perpendicular a cada uno de esos dos vectores.

Transformaciones geométricas

Una **transformación geométrica** T es una aplicación desde un espacio afín A_n en otro B_n con la misma dimensión, donde A_n y B_n pueden ser distintos o el **mismo espacio**.

La transformación T se aplica a los puntos de A_n y produce puntos en B_n . Si \dot{q} es la imagen por T de \dot{p} , escribimos:

$$\dot{q} = T(\dot{p})$$

En general, una transformación geométrica

- aplicada a una recta puede producir otra recta, o bien una curva,
- puede ser continua o no serlo,
- puede no tener inversa si mas de un punto de A_n tiene como imagen un mismo punto en B_n .

Como consecuencia de todo esto, una transformación geométrica en general **no puede aplicarse a los vectores libres**, únicamente a los puntos.

Función asociada a una transformación geométrica

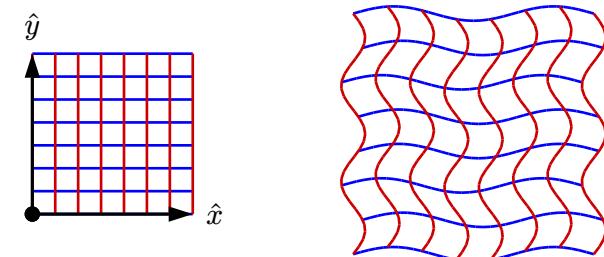
Si fijamos un marco \mathcal{R} del espacio A_n y una transformación geométrica T , entonces existirá una función F que asocia cada tupla c de coordenadas de un punto \dot{p} con la correspondiente tupla $c' = F(c)$ de coordenadas del punto transformado por T , es decir, se cumple:

$$T(\dot{p}) = T(\mathcal{R}c) = \mathcal{R}F(c) = \mathcal{R}c'$$

- La llamamos la **funciones asociadas a la transformación T en el marco \mathcal{R}** .
- Esa función es en realidad una transformación geométrica en el espacio de las tuplas de coordenadas de puntos, es decir, en A_n .
- La función F se puede usar para implementar la transformación en un programa.
- Por supuesto, esta función depende del marco \mathcal{R} que hemos seleccionado, para otro marco sería distinta.

Ejemplo de una transformación geométrica

Los puntos originales forman una rejilla (a la izquierda), cuando se les aplica una transformación geométrica T , la rejilla se puede deformar de manera arbitraria (a la derecha):



En este ejemplo, un punto de coordenadas cartesianas (x, y) (en $[0, 1]^2$) se transforma en otro punto de coordenadas cartesianas $(x', y') = F(x, y)$, según las fórmulas:

$$x' \equiv 2 + 1.5(x + 0.05 \sin(5\pi y))$$

$$y' \equiv -0.25 + 1.5(y + 0.03 \sin(3\pi x))$$

Transformaciones afines

Una transformación geométrica T es una **transformación geométrica afín** si T es continua y transforma cada paralelogramo en otro paralelogramo. Un paralelogramo está definido por 4 puntos, con lados iguales dos a dos. Formalmente:

- Para cualquiera cuatro puntos $\dot{p}, \dot{q}, \dot{r}, \dot{s}$, si forman un paralelogramo, entonces sus imágenes forman otro, es decir:

$$\dot{q} - \dot{p} = \dot{s} - \dot{r} \implies T(\dot{q}) - T(\dot{p}) = T(\dot{s}) - T(\dot{r})$$

- Esa propiedad permite **aplicar T a vectores y producir vectores**. Si $\vec{v} = \dot{q} - \dot{p}$, entonces se cumple que:

$$T(\vec{v}) = T(\dot{q} - \dot{p}) \equiv T(\dot{q}) - T(\dot{p})$$

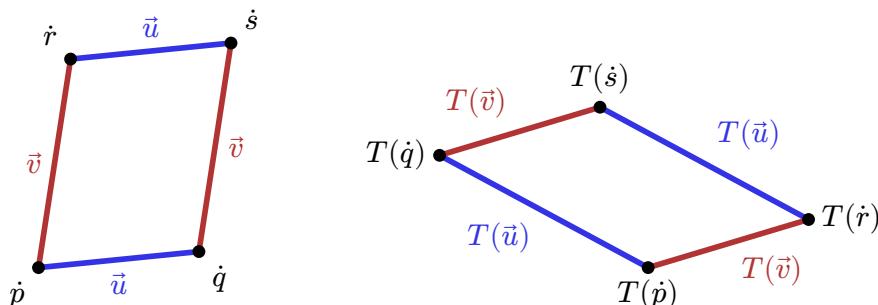
el vector resultado está bien definido pues no importa que par \dot{p} y \dot{q} seleccionemos: siempre que $\vec{v} = \dot{q} - \dot{p}$, se obtendrá el mismo vector $T(\vec{v})$.

Subsección 2.1.

Transformaciones afines

Ejemplo de transformación afín de un paralelogramo

Si T es afín y los puntos $\dot{p}, \dot{q}, \dot{r}, \dot{s}$ forman un paralelogramo, entonces sus imágenes $T(\dot{p}), T(\dot{q}), T(\dot{r})$ y $T(\dot{s})$ forman otro:



Los dos vectores \vec{u} y \vec{v} en las aristas originales se transforman:

- $\vec{u} = \dot{q} - \dot{p} = \dot{s} - \dot{r}$, se transforma en $T(\vec{u}) = T(\dot{q}) - T(\dot{p}) = T(\dot{s}) - T(\dot{r})$,
- $\vec{v} = \dot{r} - \dot{p} = \dot{s} - \dot{q}$, se transforma en $T(\vec{v}) = T(\dot{r}) - T(\dot{p}) = T(\dot{s}) - T(\dot{q})$.

Linealidad de las transformaciones afines

Se puede demostrar fácilmente que si T es afín, entonces para cualquiera dos vectores \vec{u} y \vec{v} se cumplirá que:

$$T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v}).$$

También sabemos que T es **lineal**: para un punto \dot{p} , un vector \vec{v} , y un real a :

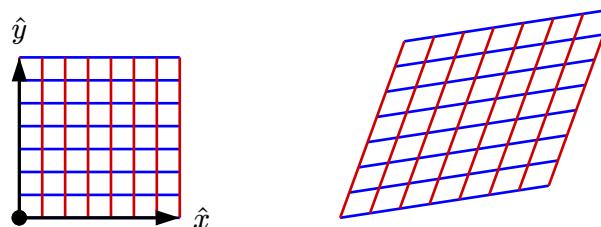
$$T(\dot{p} + a\vec{v}) = T(\dot{p}) + aT(\vec{v}).$$

Esto implica que T :

- **transforma líneas rectas en líneas rectas**, conservando la ecuación paramétrica de dichas rectas, y por tanto,
- **conserva las proporciones** entre las longitudes de vectores paralelos, y
- **conserva el paralelismo**: transforma dos líneas paralelas en otras dos líneas paralelas.

Conservación del paralelismo en una transformación de ejemplo

En este ejemplo de transformación afín, las líneas paralelas de la rejilla original se transforman en otra líneas paralelas en la rejilla transformada:



En este ejemplo, un punto de coordenadas cartesianas (x, y) (en $[0, 1]^2$) se transforma en otro punto de coordenadas cartesianas $(x', y') = F(x, y)$, según las fórmulas:

$$\begin{aligned} x' &\equiv 1.3x + 0.4y + 2 \\ y' &\equiv 0.2x + 1.1y + 0 \end{aligned}$$

Transformaciones afines singulares y no singulares

Podemos distinguir las siguientes dos clases de transformaciones afines, en función de si son una biyección o no:

- Una transformación afín T será **no singular** si siempre aplica puntos distintos en puntos distintos, es decir, se cumple:

$$\dot{q} - \dot{p} \neq \vec{0} \implies T(\dot{q}) \neq T(\dot{p})$$

es lo mismo que decir que la imagen de cualquier vector no nulo es otro vector no nulo. Una transformación no singular es biyectiva y **tiene inversa**, T^{-1} .

- Una transformación afín será **singular** si hay al menos dos puntos con la misma imagen, o equivalentemente, hay al menos un vector no nulo que se transforma en el vector nulo. Una transformación singular **no tiene inversa**.
- Un ejemplo de transformación afín singular es una que pone a cero una componente de los vectores (por ejemplo la componente X): anula cualquier vector paralelo a ese eje.

La matriz asociada a una transformación

Fijado un marco $\mathcal{R} = (\vec{a}_0, \dots, \vec{a}_{n-1}, \dot{o})$ y una transformación T (con función F en \mathcal{R}) consideramos los vectores de la base y el origen pero transformados por T , es decir los vectores $\vec{b}_j = T(\vec{a}_j)$ y el punto $\dot{p} = T(\dot{o})$. Sus coordenadas en \mathcal{R} son \mathbf{b}_j y \mathbf{p} , respectivamente, es decir:

$$\vec{b}_j = \mathcal{R}\mathbf{b}_j \quad y \quad \dot{p} = \mathcal{R}\mathbf{p} \quad \text{donde: } \begin{cases} \mathbf{b}_j = (b_{0,j}, \dots, b_{n-1,j}, 0)^T \\ \mathbf{p} = (p_0, \dots, p_{n-1}, 1)^T \end{cases}$$

Entonces podemos construir la **matriz asociada** a T (en coordenadas de \mathcal{R}), con $n+1$ filas y columnas, y que tiene las tuplas \mathbf{b}_j y \mathbf{p} dispuestas **por columnas**, y que escribimos como M_T :

$$M \equiv \begin{pmatrix} b_{0,0} & \dots & b_{0,n-1} & p_0 \\ \vdots & & \vdots & \vdots \\ b_{n-1,0} & \dots & b_{n-1,n-1} & p_{n-1} \\ 0 & \dots & 0 & 1 \end{pmatrix}$$

La función asociada a una transformación afín

Sea F la función asociada a la transformación afín T en el marco \mathcal{R} anterior. Sean $\mathbf{c} = (c_0, \dots, c_{n-1}, w)$ las coordenadas (en \mathcal{R}) de un punto o vector $\mathcal{R}\mathbf{c}$ cualquiera:

- Para un punto $\dot{p} = \mathcal{R}\mathbf{p}$ y un vector $\vec{v} = \mathcal{R}\mathbf{v}$, y un real a se cumple:

$$T(\dot{p} + a\vec{v}) = \mathcal{R}(F(\mathbf{p}) + aF(\mathbf{v}))$$

- Esto implica que F **implementa** T en \mathcal{R} , es decir:

$$T(\mathcal{R}\mathbf{c}) = \mathcal{R}F(\mathbf{c})$$

- A su vez, esto significa que podemos escribir esta implicación:

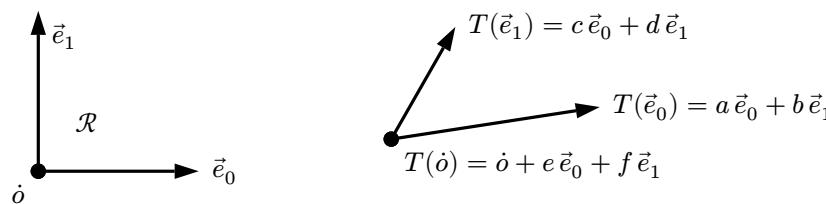
$$\mathcal{R}\mathbf{c} = w\dot{o} + \sum_{j=0}^{n-1} c_j \vec{a}_j \implies F(\mathbf{c}) = w\mathbf{p} + \sum_{j=0}^{n-1} c_j \mathbf{b}_j$$

- Lo cual supone que la función F se puede expresar usando la matriz M :

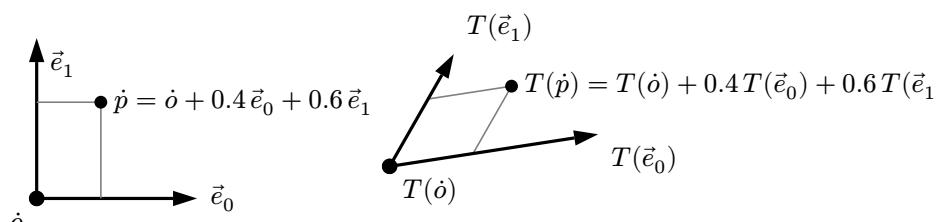
$$F(\mathbf{c}) = M\mathbf{c}$$

Matriz de una transformación afín 2D en un marco (2)

El mismo marco $\mathcal{R} = (\vec{e}_0, \vec{e}_1, \dot{o})$, el punto $T(\dot{o})$, y los vectores $T(\vec{e}_0)$ y $T(\vec{e}_1)$:



El punto \dot{p} de coordenadas $(0.4, 0.6, 1)^T$ (en \mathcal{R}) se transforma en el punto $T(\dot{p})$



Matriz de una transformación afín 2D en un marco

Dado un marco $\mathcal{R} = (\vec{e}_0, \vec{e}_1, \dot{o})$ y una transformación afín T , supongamos que:

- Las coordenadas de $T(\vec{e}_0)$ en \mathcal{R} son $(a, b, 0)^T$,
- Las coordenadas de $T(\vec{e}_1)$ en \mathcal{R} son $(c, d, 0)^T$,
- Las coordenadas de $T(\dot{o})$ en \mathcal{R} son $(e, f, 1)^T$ y
- Un punto \dot{p} tiene coordenadas $(x, y, 1)^T$ (en \mathcal{R}), es decir $\dot{p} = x\vec{e}_0 + y\vec{e}_1 + 1\dot{o}$.

Aplicando T a ambos lados de la igualdad, obtenemos:

$$T(\dot{p}) = xT(\vec{e}_0) + yT(\vec{e}_1) + 1T(\dot{o})$$

- El punto $T(\dot{p})$ tendrá unas coordenadas $(x', y', 1)^T$ (también en \mathcal{R}). Por tanto:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = x \begin{pmatrix} a \\ b \\ 0 \end{pmatrix} + y \begin{pmatrix} c \\ d \\ 0 \end{pmatrix} + 1 \begin{pmatrix} e \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Es decir, los 6 valores reales a, b, c, d, e, f forman la matriz M asociada a T y determinan completamente la transformación T

La matriz asociada a una transformación afín en 3D

Igualmente, en el caso 3D, consideramos el marco $\mathcal{R} = (\vec{e}_0, \vec{e}_1, \vec{e}_2, \dot{o})$

- las coordenadas de $T(\vec{e}_0)$ en \mathcal{R} son $(a, b, c, 0)^T$,
- las coordenadas de $T(\vec{e}_1)$ en \mathcal{R} son $(d, e, f, 0)^T$,
- las coordenadas de $T(\vec{e}_2)$ en \mathcal{R} son $(g, h, i, 0)^T$,
- las coordenadas de $T(\dot{o})$ en \mathcal{R} son $(j, k, l, 1)^T$ y
- un punto de coordenadas $(x, y, z, 1)^T$ (en \mathcal{R}) se transforma en otro punto de coordenadas $(x', y', z', 1)^T$ (también en \mathcal{R}).

Los 12 valores reales $a, b, c, d, e, f, g, h, i, j, k, l$ forman la matriz M y determinan completamente la transformación T , ya que:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = x \begin{pmatrix} a \\ b \\ c \\ 0 \end{pmatrix} + y \begin{pmatrix} d \\ e \\ f \\ 0 \end{pmatrix} + z \begin{pmatrix} g \\ h \\ i \\ 0 \end{pmatrix} + 1 \begin{pmatrix} j \\ k \\ l \\ 1 \end{pmatrix} = \begin{pmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Implementación de transformaciones afines con matrices

El resultado anterior nos dice que **podemos implementar una transformación afín usando su matriz asociada**:

- Eso permite implementar transformaciones afines en un programa, simplemente multiplicando la matriz (a la izquierda) por las coordenadas homogéneas del punto o vector a transformar (a la derecha).
- Si T es no singular, entonces M_T será invertible, y podremos implementar la transformación inversa T^{-1} usando la inversa de la matriz M_T^{-1} .
- La matriz M_T se puede descomponer como el producto de otras matrices cuadradas

$$M_T = D_T R_T$$

es decir, aplicar M_T equivale a aplicar primero R_T y luego D_T , donde:

- ▶ R_T es la matriz que transforma los vectores, y
- ▶ D_T es la matriz que desplaza el origen.

Composición de transformaciones y matrices. Inversas.

Las transformaciones afines se pueden componer: si T_1 y T_2 son dos transformaciones afines:

- La composición de ambas es otra transformación afín S , que se escribe como $T_2 \circ T_1$ y que supone transformar primero por T_1 y luego por T_2 , es decir, para un punto \vec{p} se cumple:

$$S(\vec{p}) = T_2(T_1(\vec{p}))$$

- En un marco fijo, la matriz M_S asociada a $T_2 \circ T_1$ se obtiene multiplicando las matrices M_2 y M_1 correspondientes a T_2 y T_1 , respectivamente:

$$M_S = M_2 M_1$$

- Puesto que $M_T = D_T R_T$, la inversa de M_T se puede escribir como:

$$M_T^{-1} = (R_T D_T)^{-1} = R_T^{-1} D_T^{-1}$$

Descomposición de matrices en 3D.

A modo de ejemplo, en 3D, consideramos un marco $\mathcal{R} = (\vec{e}_x, \vec{e}_y, \vec{e}_z, \vec{o})$ y una transformación afín T , consideramos las coordenadas (en \mathcal{R}) de los vectores de la base y el origen transformados:

$$\begin{aligned} T(\vec{e}_x) &= \mathcal{R}(x_0, y_0, z_0, 0)^T & T(\vec{e}_y) &= \mathcal{R}(x_1, y_1, z_1, 0)^T \\ T(\vec{e}_z) &= \mathcal{R}(x_2, y_2, z_2, 0)^T & T(\vec{o}) &= \mathcal{R}(d_x, d_y, d_z, 1)^T \end{aligned}$$

Entonces se cumple:

$$M_T = \begin{pmatrix} x_0 & x_1 & x_2 & d_x \\ y_0 & y_1 & y_2 & d_y \\ z_0 & z_1 & z_2 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 & x_1 & x_2 & 0 \\ y_0 & y_1 & y_2 & 0 \\ z_0 & z_1 & z_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = D_T R_T$$

Transformación de marcos afines

Dada una transformación T **no singular** y el marco afín $\mathcal{R} = (\vec{b}_0, \dots, \vec{b}_{n-1}, \vec{o})$ podemos definir el marco afín *transformado* \mathcal{S} así:

$$\mathcal{S} := (T(\vec{b}_0), \dots, T(\vec{b}_{n-1}), T(\vec{o})) \quad \text{donde:} \quad \begin{cases} T(\vec{b}_j) = \mathcal{R}(c_{0,j}, \dots, c_{n-1,j}, 0)^T \\ T(\vec{o}) = \mathcal{R}(p_0, \dots, p_{n-1}, 1)^T \end{cases}$$

Entonces se pueden escribir los vectores de la base y el origen transformados como combinaciones lineales de los vectores de la base y el origen originales:

$$T(\vec{b}_j) = 0\vec{o} + \sum_{i=0}^{n-1} c_{i,j} \vec{b}_i \quad \text{y} \quad T(\vec{o}) = 1\vec{o} + \sum_{i=0}^{n-1} p_i \vec{b}_i$$

Lo cual significa que podemos escribir el marco \mathcal{S} **multiplicando el marco \mathcal{R} por la matriz M_T** (por la derecha):

$$\mathcal{S} = \mathcal{R} M_T$$

donde M_T es la matriz asociada a T (en coordenadas de \mathcal{R}).

Cambio de coordenadas entre marcos afines

Dados dos marcos afines \mathcal{A} y \mathcal{B} , siempre existe una transformación afín T que transforma \mathcal{A} en \mathcal{B} . Dicha transformación tendrá asociada una matriz M (en \mathcal{A}) tal que $\mathcal{A}M = \mathcal{B}$. Podemos nombrar esa matriz como $M_{\mathcal{A}\mathcal{B}}$, tiene las coordenadas de \mathcal{B} en \mathcal{A} .

Si un punto o vector tiene coordenadas $\mathbf{c}_{\mathcal{A}}$ en \mathcal{A} y $\mathbf{c}_{\mathcal{B}}$ en \mathcal{B} , entonces se cumple:

$$\mathcal{A}\mathbf{c}_{\mathcal{A}} = \mathcal{B}\mathbf{c}_{\mathcal{B}}$$

Pero podemos sustituir \mathcal{B} por $\mathcal{A}M_{\mathcal{A}\mathcal{B}}$ y usando la asociatividad, obtenemos:

$$\mathcal{A}\mathbf{c}_{\mathcal{A}} = \mathcal{B}\mathbf{c}_{\mathcal{B}} = (\mathcal{A}M_{\mathcal{A}\mathcal{B}})\mathbf{c}_{\mathcal{B}} = \mathcal{A}M_{\mathcal{A}\mathcal{B}}\mathbf{c}_{\mathcal{B}} = \mathcal{A}(M_{\mathcal{A}\mathcal{B}}\mathbf{c}_{\mathcal{B}})$$

Puesto que las coordenadas en \mathcal{A} son únicas, se deduce que:

$$\mathbf{c}_{\mathcal{A}} = M_{\mathcal{A}\mathcal{B}}\mathbf{c}_{\mathcal{B}}$$

Es decir, la matriz $M_{\mathcal{A}\mathcal{B}}$ permite hacer el **cambio de coordenadas desde las coordenadas en \mathcal{B} a las coordenadas en \mathcal{A}** .

Interpretaciones de la acción de una matriz

Si tenemos una matriz M cualquiera (con determinante no nulo), y un marco afín \mathcal{A} , siempre existe una transformación afín T con matriz M que transforma \mathcal{A} en otro marco \mathcal{B} . Esto implica que podemos ver la matriz M de varias formas distintas:

- Como algo que implementa la función F de T : dadas las coordenadas \mathbf{c} de un punto o vector relativos a \mathcal{A} , nos da las coordenadas $F(\mathbf{c})$ del punto o vector transformado, también relativas a \mathcal{A} :

$$F(\mathbf{c}) = M\mathbf{c}$$

- Permite hacer el cambio de coordenadas desde las coordenadas en \mathcal{B} a las coordenadas en \mathcal{A} . Si $\mathbf{c}_{\mathcal{B}}$ son coordenadas en \mathcal{B} y $\mathbf{c}_{\mathcal{A}}$ las correspondientes en \mathcal{A} (ambas de un mismo punto o vector), entonces:

$$\mathbf{c}_{\mathcal{A}} = M\mathbf{c}_{\mathcal{B}}$$

- Como algo que transforma el marco \mathcal{A} en el marco \mathcal{B} ya que

$$\mathcal{B} = \mathcal{A}M$$

Subsección 2.3.

Tipos de transformaciones.

Transformaciones isométricas

Una transformación T es una transformación **isométrica** si conserva el valor absoluto del producto escalar, es decir, para dos vectores cualquiera \vec{u} y \vec{v} se cumple:

$$\| T(\vec{u}) \cdot T(\vec{v}) \| = \| \vec{u} \cdot \vec{v} \|$$

Como consecuencia T

- es una transformación afín, pero no conserva la orientación,
- conserva las distancias entre puntos, es decir conserva la longitud de los vectores: se cumple $\|T(\vec{v})\| = \|\vec{v}\|$,
- conserva el valor absoluto del ángulo entre dos líneas,
- conserva las áreas y volúmenes de los objetos, y
- tendrá asociada una matriz R_T (en un marco cartesiano cualquiera) que será ortonormal, y con determinante igual a $+1$ o a -1 .

Las **traslaciones, las rotaciones y las reflexiones** son ejemplos de isometrías.

Transformaciones rígidas.

Una transformación T que conserva el producto escalar (incluyendo el signo) es una transformación **rígida** u **ortogonal**. Para dos vectores cualquiera \vec{u} y \vec{v} se cumple:

$$T(\vec{u}) \cdot T(\vec{v}) = \vec{u} \cdot \vec{v}$$

Por tanto T :

- es afín y conserva la orientación,
- es isométrica,
- conserva las distancias, áreas y volúmenes,
- conserva los ángulos (en magnitud y signo), y
- tendrá asociada una matriz R_T (en un marco cartesiano cualquiera) que será ortonormal, y con determinante igual a +1.

Las **traslaciones** y **rotaciones** son rígidas (no las reflexiones).

Sección 3. Transformaciones usuales en Informática Gráfica.

1. Traslaciones
2. Escalados y reflexiones.
3. Cizallas
4. Rotaciones

Introducción

En esta sección estudiaremos las transformaciones usuales en Informática Gráfica, tanto en el espacio 2D como 3D.

- Traslaciones.
- Escalados: uniformes, no uniformes, reflexiones.
- Cizallas (*shearings*).
- Rotaciones: entorno a al origen (en 2D), entorno a los ejes de coordenadas o entorno a un eje arbitrario (en 3D).

En cada caso vemos las propiedades y las correspondientes matrices, relativas siempre a un marco \mathcal{R} en 2D o 3D (que en algunos casos será cartesiano). Para introducir las transformaciones se indica como se transforma el origen y los versores de dicho marco cartesiano, y eso nos da directamente la matriz.

Subsección 3.1. Traslaciones

Traslaciones en 2D y 3D

Una **traslación** T por un vector \vec{d} es una transformación afín rígida que a cada punto \vec{p} le asocia el punto $\vec{p} + \vec{d}$, y lógicamente no afecta a los vectores.

En el espacio 2D, para un marco $\mathcal{R} = (\vec{x}, \vec{y}, \vec{o})$, la traslación T por el vector \vec{d} de coordenadas $\mathbf{d} = (d_x, d_y, 0)^T$ en \mathcal{R} viene dada por:

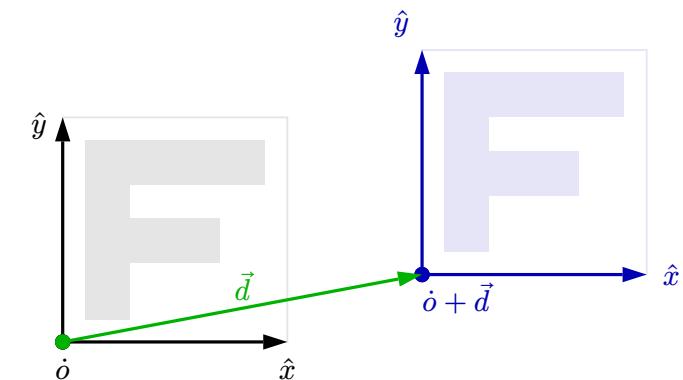
$$\begin{aligned} T_{\mathbf{d}}(\vec{x}) &= \vec{x} \\ T_{\mathbf{d}}(\vec{y}) &= \vec{y} \\ T_{\mathbf{d}}(\vec{o}) &= \vec{o} + \vec{d} \end{aligned} \quad M_T = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$$

En 3D, para un marco $\mathcal{R} = (\vec{x}, \vec{y}, \vec{z}, \vec{o})$, la traslación T por el vector \vec{d} de coordenadas $\mathbf{d} = (d_x, d_y, d_z, 0)^T$ en \mathcal{R} viene dada por:

$$\begin{aligned} T_{\mathbf{d}}(\vec{x}) &= \vec{x} \\ T_{\mathbf{d}}(\vec{y}) &= \vec{y} \\ T_{\mathbf{d}}(\vec{z}) &= \vec{z} \\ T_{\mathbf{d}}(\vec{o}) &= \vec{o} + \vec{d} \end{aligned} \quad M_T = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esquema de la traslación en 2D

Vemos una traslación en 2D por un vector $\mathbf{d} = (1.6, 0.3)$, en negro el marco original y en azul el desplazado. En verde vemos el vector de desplazamiento. Los ejes no cambian.



Subsección 3.2. Escalados y reflexiones.

Escalados.

Una **escalado** E_s es una transformación afín que multiplica las coordenadas de un punto o vector en un marco \mathcal{R} cualquiera por n factores escalares que forman la tupla $\mathbf{s} = (s_0, \dots, s_{n-1})$. Las distancias al origen se ven por tanto multiplicadas por esos factores. El origen de \mathcal{R} no cambia, y se llama **foco de escalado**.

En 2D, la tupla de los dos factores es: $\mathbf{s} = (s_x, s_y)$

$$\begin{aligned} E_s(\vec{x}) &= s_x \vec{x} \\ E_s(\vec{y}) &= s_y \vec{y} \\ E_s(\vec{o}) &= \vec{o} \end{aligned} \quad M_T = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

En 3D, la tupla es: $\mathbf{s} = (s_x, s_y, s_z)$

$$\begin{aligned} E_s(\vec{x}) &= s_x \vec{x} \\ E_s(\vec{y}) &= s_y \vec{y} \\ E_s(\vec{z}) &= s_z \vec{z} \\ E_s(\vec{o}) &= \vec{o} \end{aligned} \quad M_T = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Tipos de escalados

Las transformaciones de escalado pueden clasificarse en base a los factores de escala, suponiendo que se define en un marco \mathcal{C} cartesiano:

- Si todos los factores de escala son todos iguales a un valor s , se dice que el escalado es **uniforme**, conserva los ángulos y las proporciones de los vectores, ya que la longitud de un vector se multiplica por s al transformar.
- Si los factores son distintos, el escalado es **no uniforme**, y no conserva los ángulos ni las proporciones.

También en un marco cartesiano, podemos analizar el signo del producto de los factores (es el determinante de la matriz R_T)

- Si el producto es **positivo**, el escalado **conserva la orientación** del sistema de referencia.
- Si el producto es **negativo**, el escalado **invierte la orientación** del sistema de referencia.

Reflexiones

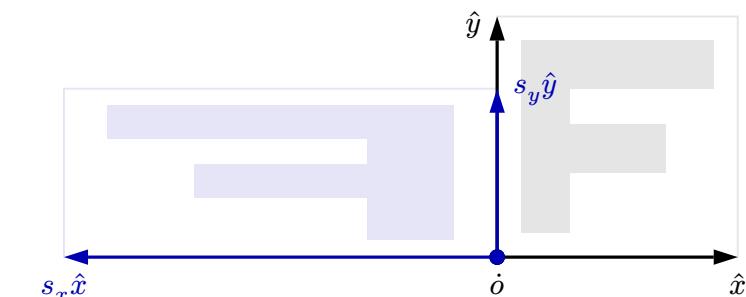
Una **reflexión** S_i es un caso particular de escalado. Son escalados definidos en un marco cartesiano \mathcal{C} que cumplen:

- Los factores de escala son todos 1, excepto uno de ellos, el correspondiente al versor \hat{e}_i que es -1 . Por eso, son su propia inversa, es decir $S_i^2 = I$.
- No conservan la orientación: en 2D y 3D convierten un marco a derechas en uno a izquierdas y al revés.
- Conservan las longitudes de los vectores, y la magnitud de los ángulos (no su signo). Así que son **isométricas**, pero no **rígidas**.
- La transformación invierte el signo de una de las coordenadas de cualquier punto o vector, con lo cual es una **reflexión** cuyo **eje o plano de reflexión** (una línea o plano invariante) es el la línea o el plano paralelo a \hat{e}_i que pasa por el origen.
- Las distancias al eje o plano se conservan, pero el punto o vector pasa al otro lado del mismo.

Esquema de escalados en 2D

Vemos un escalado en 2D por un vector, en negro el marco original y en azul el escalado. Se usan los factores de escala $s_x = -1.8$ y $s_y = 0.7$.

El cuadrado unidad con la figura en gris se transforma en el rectángulo con la figura en azul, invirtiendo la orientación del marco (ya que el signo de $s_x s_y$ es negativo).



Reflexiones con eje o planos arbitrarios

Una reflexión $S_{\hat{n}}$ puede hacerse teniendo como eje una línea cualquiera por el origen (en 2D) o con un plano de reflexión por el origen (en 3D), en cualquier caso la línea o el plano es perpendicular a un versor \hat{n} , no son necesariamente paralelos a los ejes de coordenadas.

Las coordenadas del versor \hat{n} serán $\mathbf{n} = (n_x, n_y, n_z)$ en 3D o $\mathbf{n} = (n_x, n_y)$ en 2D. En estas condiciones, la matriz de transformación asociada es la **Matriz de Householder** M_S

$$M_S \equiv I - 2(\mathbf{n}\mathbf{n}^T)$$

donde I es la matriz identidad y $\mathbf{n}\mathbf{n}^T$ es esta matriz (en 2D y 3D):

$$\mathbf{n}\mathbf{n}^T \equiv \begin{pmatrix} n_x n_x & n_x n_y & 0 \\ n_x n_y & n_y n_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \mathbf{n}\mathbf{n}^T \equiv \begin{pmatrix} n_x n_x & n_x n_y & n_x n_z & 0 \\ n_x n_y & n_y n_y & n_y n_z & 0 \\ n_x n_z & n_y n_z & n_z n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Cizallas.

Una **cizalla (shear)** C_{ij} es una transformación afín que, en un marco \mathcal{R} cualquiera, incrementa la i -ésima coordenada de una tupla, usando un incremento proporcional a la j -ésima coordenada de esa tupla (con $i \neq j$).

- El valor transformado c'_i de la i -ésima coordenada será $c'_i = c_i + ac_j$, donde a es un parámetro real que define la cizalla, y c_i y c_j las coordenadas originales. Las otras coordenadas c_j (con $i \neq j$) no cambian.
- La matriz asociada es igual a la matriz identidad, excepto que el valor en la fila i y columna j es a en lugar de 0.
- El marco transformado por C_{ij} es igual al original, excepto que en el marco transformado el eje $T(\vec{e}_j)$ se hace igual a $\vec{e}_j + a\vec{e}_i$.
- Las cizallas no conservan las longitudes ni los ángulos en general. Por eso son transformaciones **no rígidas**.
- Las cizallas únicamente conservan longitudes en líneas perpendiculares a \vec{e}_j . En 3D conservan ángulos entre vectores perpendiculares a \vec{e}_j .

Subsección 3.3.

Cizallas

Matrices de las cizallas en 2D.

En 2D hay dos tipos de cizallas, las llamamos C_{01} y C_{10} . Ambas dependen del parámetro real a .

- La cizalla C_{01} incrementa la coordenada X según la Y por tanto el marco transformado y la matriz son:

$$\begin{aligned} C_{01,a}(\vec{x}) &= \vec{x} \\ C_{01,a}(\vec{y}) &= \vec{y} + a\vec{x} \\ C_{01,a}(\dot{o}) &= \dot{o} \end{aligned} \quad M_T = \begin{pmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- La otra cizalla en 2D es C_{10} , que incrementa la coordenada Y según la X, así que ahora tenemos:

$$\begin{aligned} C_{10,a}(\vec{x}) &= \vec{x} + a\vec{y} \\ C_{10,a}(\vec{y}) &= \vec{y} \\ C_{10,a}(\dot{o}) &= \dot{o} \end{aligned} \quad M_T = \begin{pmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Matrices de las cizallas en 3D.

En 3D hay 6 tipos de cizallas, ya que hay 6 posibles pares i, j distintos.

Al igual que en 2D:

- La matriz es igual a la matriz identidad, excepto que en la fila i y columna j aparece el valor a en lugar de 0.
- El eje \hat{e}_j se incrementa por $a\hat{e}_i$.

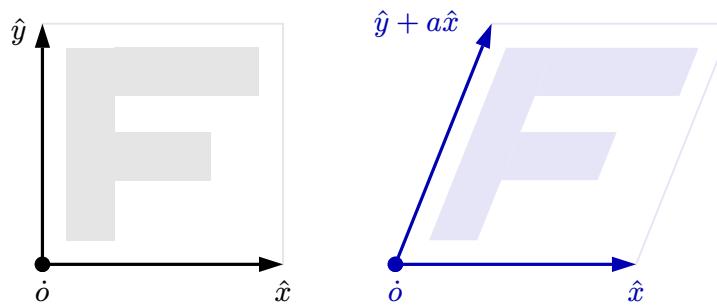
A modo de ejemplo, vemos la cizalla C_{21} que incrementa la coordenada Z ($i = 2$) en base a la Y ($j = 1$)

$$\begin{aligned} C_{21,a}(\vec{x}) &= \vec{x} \\ C_{21,a}(\vec{y}) &= \vec{y} + a\vec{z} \\ C_{21,a}(\vec{z}) &= \vec{z} \\ C_{21,a}(\dot{o}) &= \dot{o} \end{aligned} \quad M_T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & a & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esquema de una cizalla en 2D

Vemos la cizalla C_{01} en 2D con $a = 0.4$. Los puntos se desplazan en el eje X una distancia proporcional a su coordenada Y.

El cuadrado unidad con la figura en gris (a la izquierda) se transforma en el paralelogramo con la figura en azul a la derecha (la transformación no conlleva traslación, pero se han desplazado las figuras para apreciarlas mejor).



Subsección 3.4.
Rotaciones

Rotaciones en 2D

Una transformación de rotación R en un marco cartesiano $\mathcal{C} = (\hat{x}, \hat{y}, \hat{o})$, es una transformación afín rígida que:

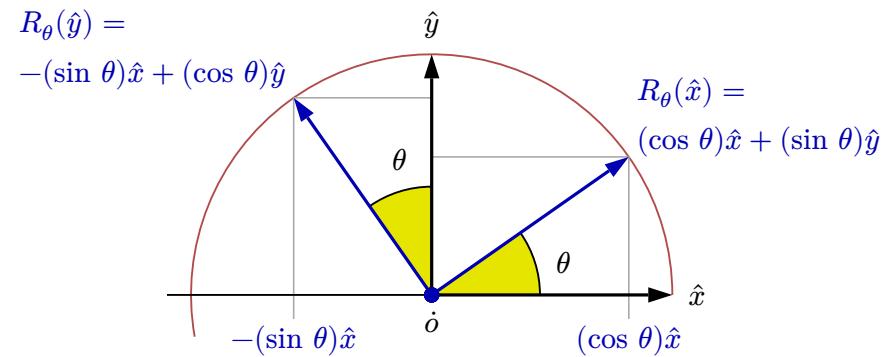
- Ileva cada punto p a otro a la misma distancia de \hat{o} que el original. Se dice que \hat{o} es el **centro de rotación**.
- depende de un parámetro θ , llamado **ángulo de rotación**, que es el ángulo en radianes (con signo) entre p y $R(p)$. Si $\theta > 0$, la rotación es antihoraria, y si $\theta < 0$ es horaria.

Así que la transformación de los versores y el origen, y la matriz son:

$$\begin{aligned} R_\theta(\hat{x}) &= (\cos \theta)\hat{x} + (\sin \theta)\hat{y} \\ R_\theta(\hat{y}) &= -(\sin \theta)\hat{x} + (\cos \theta)\hat{y} \\ R_\theta(\hat{o}) &= \hat{o} \end{aligned} \quad M_R = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Componentes de los ejes rotados en 2D

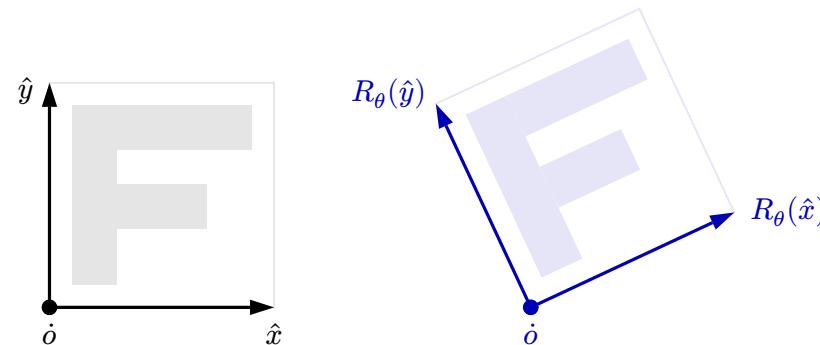
Vemos una rotación en 2D un ángulo $\theta = 35^\circ$, que es mayor que 0. En negro aparece un marco \mathcal{R} y en azul el marco transformado. Ambos tienen el mismo origen.



Esquema de una rotación en 2D

Vemos una rotación en R_θ con $\theta = 25^\circ$. Los puntos se rotan entorno al origen.

El cuadrado de lado 1 con la figura en gris (a la izquierda) se transforma en el cuadrado con la figura en azul a la derecha (la transformación no conlleva traslación, pero se han desplazado las figuras para apreciarlas mejor).



Problemas: rotaciones 2D (2/3)

Problema 3.6:

Demuestra que si rotamos en 2D un vector +90 grados o -90 grados, obtenemos otro vector perpendicular al original, es decir, si $\|\theta\| = \pi/2$ entonces

$$\vec{v} \cdot R_\theta(\vec{v}) = 0.$$

Problema 3.7:

Demuestra que una matriz de rotación en 2D es siempre ortonormal, independientemente del ángulo. Eso quiere decir que sus filas son perpendiculares dos a dos, e igual ocurre con sus columnas, y además cada fila y columna tienen longitud 1.

Problemas: rotaciones 2D (1/3)

Problema 3.4:

Demuestra que el producto escalar de vectores en 2D es invariante por rotación, es decir, que para cualquier ángulo θ y vectores \vec{u} y \vec{v} se cumple:

$$R_\theta(\vec{u} \cdot \vec{v}) = R_\theta(\vec{u}) \cdot R_\theta(\vec{v})$$

(para demostrarlo usa las coordenadas de los vectores en un marco cartesiano cualquiera)

Problema 3.5:

Demuestra que en 2D las rotaciones no modifican la longitud de un vector, es decir, que para cualquier ángulo θ y vector \vec{v} , se cumple:

$$\|R_\theta(\vec{v})\| = \|\vec{v}\|$$

Problemas: rotaciones 2D (3/3)

Problema 3.8:

Demuestra que, en 2D, el producto de una matriz de rotación y una de escalado no es comutativo en general, excepto si el escalado es uniforme.

Problema 3.9:

Demuestra que en 2D, el producto de una matriz de rotación y otra de traslación (por un vector no nulo) no es comutativo.

Rotaciones en 3D

Una transformación de rotación R_θ en un marco cartesiano $\mathcal{C} = (\hat{x}, \hat{y}, \hat{z}, \dot{o})$, es una transformación afín rígida que:

- lleva cada punto \vec{p} a otro a la misma distancia que \vec{p} de una línea que pasa por \dot{o} , y es paralela a un versor \hat{e} (a ese versor o a esa línea se les llama **eje de rotación**).
- también depende de un **ángulo de rotación** θ en radianes entre \vec{p} y $R(\vec{p})$.
- cuando $\theta > 0$, la rotación es antihoraria, vista desde un punto situado en el eje de rotación y mirando hacia el origen. Si $\theta < 0$, la rotación es horaria.
- el eje puede ser paralelo a uno de los tres versores de \mathcal{C} , o puede ser otro cualquiera.

Rotaciones en 3D entorno a los ejes cartesianos. Eje Z

En este caso el eje de rotación \hat{e} es alguno de los versores del marco \mathcal{C} , es decir es \hat{x} , \hat{y} o \hat{z} . Las coordenadas en el eje \hat{e} no cambian, ya que las rotaciones ocurren en un plano perpendicular a \hat{e} . Las otras dos coordenadas cambian igual que en 2D.

Nombramos las rotaciones indicando ángulo y eje. Para una rotación entorno al eje Z tenemos las mismas expresiones que en 2D para X e Y:

$$\begin{aligned} R_{z,\theta}(\hat{x}) &= (\cos \theta)\hat{x} + (\sin \theta)\hat{y} \\ R_{z,\theta}(\hat{y}) &= -(\sin \theta)\hat{x} + (\cos \theta)\hat{y} \\ R_{z,\theta}(\hat{z}) &= \hat{z} \\ R_{z,\theta}(\dot{o}) &= \dot{o} \end{aligned}$$

$$M_R = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotaciones en 3D entorno a los ejes cartesianos. Ejes Y y X

Para rotaciones entorno al eje Y:

$$\begin{aligned} R_{y,\theta}(\hat{x}) &= (\cos \theta)\hat{x} - (\sin \theta)\hat{z} \\ R_{y,\theta}(\hat{y}) &= \hat{y} \\ R_{y,\theta}(\hat{z}) &= (\sin \theta)\hat{x} + (\cos \theta)\hat{z} \\ R_{y,\theta}(\dot{o}) &= \dot{o} \end{aligned}$$

$$M_R = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para rotaciones entorno al eje X:

$$\begin{aligned} R_{x,\theta}(\hat{x}) &= \hat{x} \\ R_{x,\theta}(\hat{y}) &= (\cos \theta)\hat{x} + (\sin \theta)\hat{y} \\ R_{x,\theta}(\hat{z}) &= -(\sin \theta)\hat{x} + (\cos \theta)\hat{y} \\ R_{x,\theta}(\dot{o}) &= \dot{o} \end{aligned}$$

$$M_R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Problemas: rotaciones 3D entorno a los ejes cartesianos(1/n)

Considera las rotaciones en 3D entorno a \hat{e} , que es uno de los ejes \hat{x} , \hat{y} y \hat{z} de un marco cartesiano 3D.

Problema 3.10:

Demuestra que el producto escalar de vectores en 3D es invariante por estas rotaciones, y que tampoco modifican la longitud de un vector. Te puedes basar en los problemas similares en 2D.

Problema 3.11:

Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualquiera dos vectores \vec{u} y \vec{v} y un ángulo θ , se cumple:

$$R_{\theta,\hat{e}}(\vec{u} \times \vec{v}) = R_{\theta,\hat{e}}(\vec{u}) \times R_{\theta,\hat{e}}(\vec{v})$$

Rotaciones en 3D de ejes arbitrarios

En Informática Gráfica se puede necesitar hacer rotaciones entorno a ejes que son distintos de los ejes cartesianos.

Suponiendo que el eje es un versor \hat{e} (unitario), entonces se puede escribir el vector rotado usando la llamada **fórmula de Rodrigues**:

$$R_{\hat{e}, \theta}(\vec{v}) = (\cos \theta)\vec{v} + (1 - \cos \theta)(\vec{v} \cdot \hat{e})\hat{e} + (\sin \theta)\hat{e} \times \vec{v}$$

Si las coordenadas de \hat{e} en \mathcal{C} son $(e_0, e_1, e_2, 0)$, podemos escribir la matriz M_R como una suma:

$$M_R = \begin{pmatrix} a_{00} & a_{01} & a_{02} & 0 \\ a_{10} & a_{11} & a_{12} & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} c & -se_2 & se_1 & 0 \\ se_2 & c & -se_0 & 0 \\ -se_1 & se_0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ con } \begin{cases} a_{ij} := (1 - c)e_i e_j \\ c := \cos \theta \\ s := \sin \theta \end{cases}$$

Cuaterniones para rotaciones en 3D

Otra alternativa es usar los **cuaterniones (quaternions)** que son objetos que se pueden ver como una extensión de los números complejos a 4 dimensiones. Un cuaternion q tiene la forma:

$$q = a + bi + cj + dk$$

donde a, b, c y d son reales, y i, j y k son tres unidades imaginarias distintas. Al igual que los complejos, los cuaterniones se pueden sumar, restar y multiplicar entre ellos. Las propiedades de los cuaterniones permiten:

- Representar rotaciones en 3D de eje y ángulo arbitrarios con un cuaternion (es decir, con una tupla de 4 reales en memoria).
- Multiplicar cuaterniones para componer rotaciones.
- Representar cualquier vector en 3D como un cuaternion.
- **Rotar un vector alrededor de un eje arbitrario de forma muy eficiente**, usando los dos cuaterniones que representan la rotación y el vector y obteniendo el cuaternion del vector rotado.

Los ángulos de Euler para rotaciones en 3D

Otra forma alternativa de definir una rotación 3D de eje arbitrario es usando una tupla de tres valores reales (α, β, γ) llamados **ángulos de Euler**, cada uno de ellos en radianes.

Una rotación de eje arbitrario $R_{\hat{e}, \theta}$ será equivalente a la composición de tres rotaciones entorno cada uno de los tres ejes $\hat{x}, \hat{y}, \hat{z}$ de un marco cartesiano, usando los ángulos de Euler:

$$R_{\hat{e}, \theta} = R_{z, \gamma} \circ R_{y, \beta} \circ R_{x, \alpha}$$

Usar esto puede ser útil si conocemos los tres ángulos y queremos obtener la matriz (independientemente de \hat{e} y θ , que no conoceremos), un ejemplo son las *cámaras orbitales*. Sin embargo:

- Si partimos del eje \hat{e} y el ángulo θ no es sencillo calcular los ángulos de Euler que producen la misma rotación.
- Fijado \hat{e} y θ , puede haber más de una tripleta de ángulos de Euler que den lugar a la misma rotación.

Rotaciones de ejes arbitrarios: alternativas

Si una rotación debe ser aplicada a muchos vectores en la GPU, entonces es mejor calcular una vez la matriz M_R y usarla para aplicarla a cada vector. Se puede hacer de dos formas:

- Calculando la matriz M_T indicada antes.
- Componiendo: (1) una matriz de rotación R que alinea \hat{e} con \hat{x} , (2) la rotación por θ entorno a \hat{x} y finalmente (3) la rotación inversa R^{-1} de la primera. Es más complejo que calcular la matriz directamente.

Para rotar pocos vectores en la CPU puede ser más eficiente no calcular la matriz, se puede hacer de dos formas:

- Usando la fórmula de Rodrigues directamente.
- Usando cuaterniones.

La fórmula de Rodrigues y los cuaterniones son equivalentes, y para pocos vectores ambas formas son mucho más eficientes que calcular la matriz. Los cuaterniones, sin embargo, son un poco más eficientes que la fórmula de Rodrigues.

Introducción

Sección 4.
Transformaciones en Godot.

1. Tuplas de valores reales.
2. Matrices de transformación.

En esta sección estudiaremos como se representan las tuplas de coordenadas homogéneas en 2D y 3D en Godot (clases `Vector2` y `Vector3`), y las matrices de transformación (clases `Transform2D` y `Transform3D`).

También veremos como se implementan las transformaciones más comunes: traslaciones, rotaciones y escalados, mediante el atributo `transform` de las clases `Node2D` y `Node3D`.

Veremos como la transformación de un nodo en el árbol de escena afecta a ese nodo y a todos sus hijos.

Sesión 3: Espacios y transformaciones afines

Created 2025-12-30

Page 98 / 112.

4. Transformaciones en Godot..
4.1. Tuplas de valores reales..

Tuplas 2D. Clase `Vector2`.

Un objeto de la clase `Vector2` representa una tupla de dos valores reales, que se puede usar para guardar coordenadas de puntos o vectores en 2D.

- Contienen dos valores reales representados en coma flotante con 32 bits de precisión (equivalente a un `float` de C/C++, pero no a un `float` de GDscript, que tiene 64 bits).
- Los elementos son accesibles como `v.x` y `v.y`, o como `v[e]` donde `e` es una expresión entera que se evalúa a 0 o 1.
- Estos objetos se pueden sumar, restar y multiplicar por un escalar (un `float` de GDscript), usando los operadores binarios infijos `+`, `-` y `*`.
- También se pueden usar diversos métodos para otras operaciones:
 - ▶ `v.length()` devuelve la longitud del vector `v` (un `float`)
 - ▶ `v.normalized()` devuelve el vector $v/\|v\|$ (`Vector2`), si $\|v\| > 0$.
 - ▶ `v.dot(u)` devuelve el producto escalar de $v \cdot u$ (un `float`).

Hay otros muchos métodos, consultar:

docs.godotengine.org/en/stable/classes/class_vector2.html

Subsección 4.1.

Tuplas de valores reales.

Tuplas 3D. Clase `Vector3`.

Un objeto de la clase `Vector3` representa una tupla de tres valores reales, que se puede usar para guardar coordenadas de puntos o vectores en 3D.

- Contienen tres valores reales representados en coma flotante con 32 bits de precisión
- Los elementos son accesibles como `v.x`, `v.y`, `v.z`, o como `v[e]` donde `e` es una expresión entera que se evalúa a 0, 1 o 2.
- Estos objetos se pueden sumar, restar y multiplicar por un escalar (un `float` de GDscript), usando los operadores binarios infijos `+`, `-` y `*`.
- También se pueden usar diversos métodos para otras operaciones:
 - ▶ `v.length()` devuelve la longitud del `v` (`float`)
 - ▶ `v.normalized()` devuelve $v / \|v\|$ (`Vector3`), si $\|v\| > 0$.
 - ▶ `v.dot(u)` devuelve el producto escalar de $v \cdot u$ (`float`).
 - ▶ `v.cross(u)` devuelve el producto vectorial $v \times u$ (otro `Vector3`).

Consultar: docs.godotengine.org/en/stable/classes/class_vector3.html

Otros clases relacionadas

Existen otras clases para tuplas:

- Las clases `Vector2i` y `Vector3i` representan tuplas de dos y tres valores enteros (32 bits con signo). Una tupla `Vector3i` se puede usar, por ejemplo, para guardar los tres índices de un triángulo en una malla.
- La clase `Vector4` representa tuplas de cuatro valores reales (32 bits con signo). Se puede usar, por ejemplo, para representar un cuaternión.

Así como otros tipos de clases:

- **Basis**: bases del espacio vectorial en 3D.
- **Line2D**: líneas en 2D.
- **Plane**: planos en 3D.

Ahora veremos las clases para matrices de transformación:

- `Transform2D`: matrices 3×3 en 2D.
- `Transform3D`: matrices 4×4 en 3D.

Matrices de transformación 2D. Clase `Transform2D`

Un objeto de la clase `Transform2D` representa una matriz 3×3 que implementa una transformación afín en el espacio afín 2D A_2 .

- Se guardan seis valores reales de 32 bits, las dos primeras filas de la misma. La tercera fila no se guarda y siempre es $(0, 0, 1)$.
- Se pueden multiplicar (componer) entre ellas con el operadores binarios infijo `*`.
- Se pueden multiplicar o dividir por un `float` con el operador `*` o `/`.
- Se pueden aplicar a un `Vector2` con el operador `*` por la derecha, devuelve el `Vector2` resultado de aplicar la transformación (añadiendo $w = 1$ al `Vector2`, es decir, considerando el `Vector2` como las coordenadas de un punto).

Consultar: docs.godotengine.org/en/stable/classes/class_transform2d.html

Creación de objetos `Transform2D`. Propiedades

Hay varios métodos para crear objetos `Transform2D`. Supongamos que θ es un ángulo en radianes, a un real, x, y, s son `Vector2` (tuplas que representan vectores), $y o, p$ son `Vector2` que representan puntos. Entonces:

- `Transform2D()` crea la matriz identidad.
- `Transform2D(θ, o)` crea la matriz que rota θ radianes entorno a o .
- `Transform2D(x, y, o)` crea una matriz con x, y y o como 1a, 2a y 3a columnas, es decir, damos la base y el origen del marco transformado. Por tanto esto **permite construir cualquier matriz de transformación**.
- `Transform2D(θ, s, a, o)` crea la matriz componiendo una rotación por θ , un escalado por s , una cizalla por a y una traslación por o .

Se pueden usar las *propiedades* `x, y` y `origin` para consultar o actualizar los objetos `Vector2` con los valores en la 1a, 2a y 3a columnas, respectivamente. Permite leer o modificar directamente cualquier real de la matriz.

Transformaciones en 3D. La clase `Transform3D`

La clase `Transform3D` es similar a la clase `Transform2D`, solo que se guardan 4 columnas de 3 valores reales cada una (12 valores reales en total)

Los constructores, métodos y propiedades son similares, solo que se usan tuplas de tipo `Vector3` en lugar de `Vector2`. Destacamos estos dos constructores:

- `Transform3D()`: crea la matriz identidad.
- `Transform3D(x, y, z, o)`: crea una matriz cuyas columnas son x, y, z y o (tipo `Vector3` todos), es decir, damos la base y el origen del marco transformado.

Al igual que en 2D:

- estas matrices se pueden multiplicar entre ellas con `*`, y
- se pueden aplicar a un `Vector3` con `*`, por la derecha (considerando el `Vector3` como las coordenadas de un punto, es decir, añadiendo $w = 1$).

Métodos para componer una matriz 2D con otra

La clase `Transform2D` tiene varios métodos para crear una matriz B y componerla con otra matriz existente A , sin modificar A .

Hay varios métodos que devuelven BA (componen B por la **izquierda**: la acción es A seguida de B). Son:

- `A.rotated(θ)`: $B =$ rotación de θ radianes entorno al origen.
- `A.scaled(s)`: $B =$ matriz de escalado con factores s .
- `A.translated(t)`: $B =$ traslación por el vector t .

Muchas veces querremos componer las matrices creadas por la **derecha**, es decir obtener AB (acción de B seguida de A). Los correspondientes métodos son:

- `A.rotated_local(θ)`
- `A.scaled_local(s)`
- `A.translated_local(t)`

Métodos para componer una matriz 3D con otra

La clase `Transform3D` tiene varios métodos para crear una matriz B y componerla con otra matriz existente A , sin modificar A .

Varios métodos que devuelven BA (componen B por la **izquierda**: la acción es A seguida de B). Son:

- `A.rotated(e, θ)`: $B =$ rotación de θ radianes entorno al eje e (un `Vector3` que debe de tener longitud unidad).
- `A.scaled(s)`: $B =$ matriz de escalado con factores s (es un `Vector3`)
- `A.translated(t)`: $B =$ traslación por el vector t (`Vector3`)

Muchas veces querremos componer las matrices creadas por la **derecha**, es decir obtener AB (acción de B seguida de A). Los correspondientes métodos son:

- `A.rotated_local(e, θ)`
- `A.scaled_local(s)`
- `A.translated_local(t)`

La transformación de un nodo

Cada nodo de tipo `Node2D` o `Node3D` tiene una propiedad llamada `transform` que es un objeto de la clase `Transform2D` o `Transform3D`, respectivamente. Inicialmente es la matriz identidad.

- En los scripts podemos asignar valores a esa propiedad (en la función `_ready()` o `_init()`), con lo cual podemos cambiar la matriz de transformación que se aplica los vértices de ese nodo en tiempo de ejecución.
- En el editor, se pueden cambiar los valores iniciales de `transform` (posición, rotación y escalado, entre otras) en el inspector de propiedades del nodo.

También existen métodos en esas clases que permiten modificar la matriz de transformación de un nodo, sin necesidad de construir una nueva matriz y asignarla a `transform`. Estos métodos se explican más adelante.

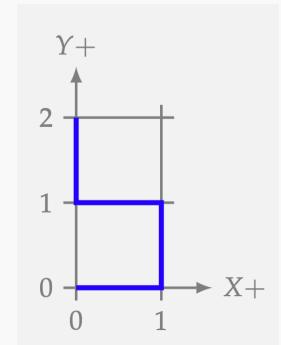
Problemas: implementación de transformaciones 2D en Godot

Para este problema y el siguiente, usa el mismo proyecto de Godot para visualización 2D que ya has usado en problemas previos.

Problema 3.12:

Crea un script global (`autoload`) con una función llamada `gancho` (sin parámetros) que crea y devuelve un objeto de la clase `Mesh` con una polilínea azul como la de la figura (los ejes se han dibujado por claridad).

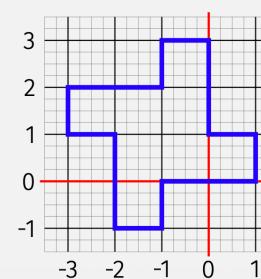
Crea en tu proyecto un nodo 2D de tipo `MeshInstance2D` y en `_ready` asígnale como malla (propiedad `mesh`) el objeto resultado de llamar a `gancho()`, ponle un color azul (propiedad `modulate`) y verifica que el gancho aparece en pantalla al ejecutar el proyecto.



Problemas: implementación de transformaciones 2D en Godot

Problema 3.13:

Crea un nodo 2D de tipo `Node2D` y llámalo `Gancho_x4`. En `_ready`, añádele cuatro nodos hijos de tipo `MeshInstance2D`, cada uno de ellos con un malla creada con la función `gancho` del problema anterior, pero con su `transform` modificada para que el objeto `Gancho_x4` se vea como en la figura (la rejilla y los ejes en rojo se han dibujado por claridad).



Fin de transparencias.

Informática Gráfica.

Sesión 4: Modelos de Objetos. Mallas indexadas..

Carlos Ureña, Sept 2025.

Dept. Lenguajes y Sistemas Informáticos.
Universidad de Granada.

Índice

Modelos geométricos	3
Modelos de fronteras: mallas de polígonos.	16
Representación de modelos de fronteras.	44
Problemas	80

Sección 1.

Modelos geométricos.

-
1. Modelos geométricos. Introducción.

Subsección 1.1.

Modelos geométricos. Introducción.

Modelos geométricos formales

Un **modelo geométrico** es un modelo matemático abstracto que sirve para representar un objeto geométrico que existe en un espacio afín E (2D o 3D).

- Los modelos deben permitir la visualización computacional de los objetos que representan.
- Los más usados hoy en día son los **modelos de fronteras**: son estructuras de datos que representan la frontera del objeto de forma exacta o aproximada, normalmente mediante **mallas de triángulos**, pero hay otras posibilidades.
- Un modelo alternativo son los **modelos de volúmenes**.
- Últimamente se usan bastante modelos basados en las **funciones de distancia con signo** (*signed distance functions*) o SDFs. Cada objeto se representa con un algoritmo que calcula la distancia (o una cota) desde cualquier punto al objeto.

En la asignatura nos centramos en las mallas de triángulos.

Representaciones computacionales

El modelo basado en subconjuntos de puntos del espacio:

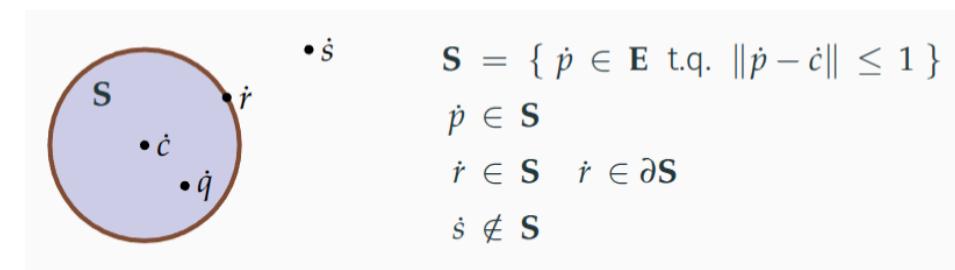
- Es un modelo válido para cualquier geometría (es el modelo **más general** posible).
- En muchos casos, **no se puede representar en la memoria** (finita, discreta) de un ordenador.

Hay representaciones aproximadas que usan una cantidad finita de memoria (**modelos geométricos computacionales**):

- **Enumeración espacial**: se partitiona el espacio en celdas o **voxels**, cada una se clasifica como interior o exterior al objeto.
- **Modelos de fronteras**: se representa la frontera (la superficie) en lugar de todo el interior, para ello se usan conjuntos finitos de polígonos planos o **caras**
- Otros tipos de modelos: algoritmos, SDFs, redes neuronales, etc..

Los conjuntos de puntos

Los modelos geométricos matemáticos abstractos más generales posibles son los **subconjuntos de puntos** de un espacio afín (típicamente 2D o 3D), por ejemplo, una esfera:

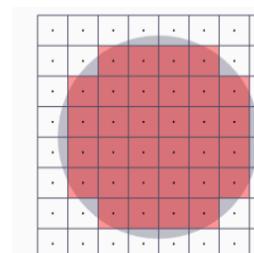


Cada subconjunto o **región** S es **cerrado** (incluye a su propia **superficie** o **frontera**, ∂S), además:

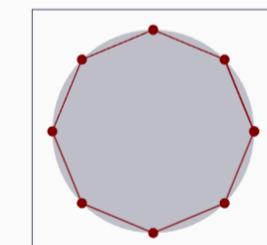
- Es **acotado** (no tiene extensión infinita),
- Su superficie es diferenciable (**plana** al menos a escala muy pequeña)

Ejemplo 2D de los modelos aproximados

Las dos formas computacionales de representar objetos son aproximadamente iguales al modelo ideal (un subconjunto de puntos), con un error que disminuye al aumentar la cantidad de memoria usada (la precisión o resolución).



Enumeración espacial



Modelos de fronteras

- **Modelos de fronteras**: usados en la mayoría de las aplicaciones.
- **Enumeración espacial**: muy útiles en aplicaciones específicas

Ejemplos de modelos de fronteras 3D (1/2)

Ejemplo de una **mallas de polígonos** (de las prácticas). A la izquierda el modelo con iluminación, a la derecha vemos las caras que forman el modelo:



Sesión 4: Modelos de Objetos. Mallas indexadas.

Created 2025-12-10

Page 9 / 91.

1. Modelos geométricos..
1.1. Modelos geométricos. Introducción..

Otros modelos de fronteras: nubes de puntos

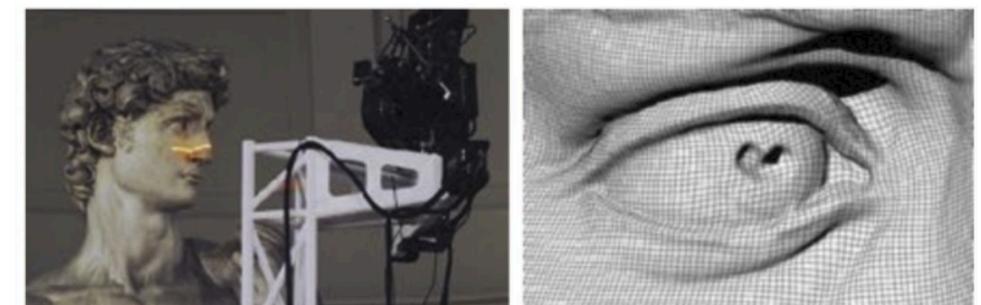
Una **nube de puntos** es un conjunto finito de puntos en el espacio 3D, cada uno con una posición y un color, que aproximan la superficie del objeto. No hay conectividad entre los puntos. Se suelen obtener a partir de escaneado 3D (LIDAR, o fotogrametría)



Imagen de la Web de Autodesk: www.autodesk.com/eu/solutions/point-clouds

Ejemplos de modelos de fronteras 3D (2/2)

Los modelos de fronteras (a muy alta resolución) permiten representar fielmente casi cualquier objeto real:



Escaneo 3D del David de Miguel Angel en Florencia en 1999.
Marc Levoy et al. The Digital Michelangelo Project: accademia.stanford.edu/mich

Sesión 4: Modelos de Objetos. Mallas indexadas.

Created 2025-12-10

Page 10 / 91.

Sesión 4: Modelos de Objetos. Mallas indexadas.

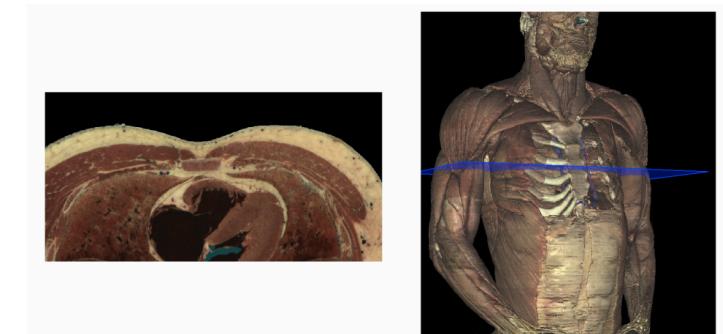
Created 2025-12-10

Page 10 / 91.

1. Modelos geométricos..
1.1. Modelos geométricos. Introducción..

Enumeración espacial 3D

Las **modelos volumétricos** se usan en aplicaciones donde interesa todo el volumen del objeto (p.ej. en la **Tomografía Axial Computerizada**, TAC, para Medicina y Arqueología, o en Geología y Climatología)



Captura de pantalla del software VH Dissector de Toltech:
www.toltech.net/anatomy-software/solutions/vh-dissector-for-medical-education

Modelos algorítmicos o procedurales

Se basan en modelar un objeto O mediante la implementación en el código de la aplicación de una función asociada al mismo (que se evalúa en cualquier punto p del espacio), sin usar una estructura de datos. Hay varias opciones:

- **Función de pertenencia de O :** la función produce un valor lógico, `true` si p está dentro de O o `false` si está fuera. La visualización usando estos modelos puede ser muy costosa en tiempo e inexacta.
- **Función de distancia con signo de O (Signed Distancia Function, o SDF):** la función devuelve la distancia más corta desde p a la frontera de O (negativa si está dentro de O , positiva si está fuera).

Se pueden usar para visualizar el objeto:

- Directamente, usando técnicas basadas en **ray-tracing**, o bien
- Indirectamente: mediante conversión a modelo de fronteras o volúmenes seguida de **rasterización**.

Modelos complejos basados en SDFs

En la actualidad las SDFs se usan para representar escenas y objetos complejos:

- Las SDFs constituyen la única forma de representar y visualizar (con un grado alto de exactitud visual) algunos tipos de objetos matemáticos, como los **fractales** (frontera no diferenciable en ningún punto). Se usan métodos numéricos iterativos (usualmente lentos).
- Se usan técnicas basadas en IA para entrenar **redes neuronales** usando imágenes o vídeos de escenarios reales, de forma que
 - ▶ La red neuronal es capaz de aproximar una SDF que codifica el modelo geométrico (y opcionalmente el de aspecto) del escenario real. La SDF se obtiene como la combinación de las SDFs de objetos simples (como por ejemplo elipsoides).
 - ▶ La visualización de la escena se puede hacer en tiempo real usando Ray-Tracing, incluso en escenarios muy complejos.

Modelos algorítmicos: ejemplo sencillo (esfera)

Por ejemplo, para una esfera O con centro en c y radio r , usando C++

Función de pertenencia: (F_O) se compara $\|\mathbf{p} - \mathbf{c}\|^2$ con r^2 :

$$F_O(\mathbf{p}) = \begin{cases} \text{true} & : \text{si } (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) \leq r^2 \\ \text{false} & : \text{en otro caso} \end{cases}$$

```
bool pertenece_esfera( vec3 & p, vec3 & c, float r )
{
    return dot( p-c, p-c ) <= r*r ;
}
```

Función de distancia con signo: (SDF_O) se restan distancia y radio:

$$SDF_O(\mathbf{p}) = \|\mathbf{p} - \mathbf{c}\| - r$$

```
float sdf_esfera( vec3 & p, vec3 & c, float r )
{
    return (p-c).length() - r ;
}
```

Sección 2. Modelos de fronteras: mallas de polígonos.

1. Introducción
2. Elementos y adyacencia.
3. Atributos de vértices.

Mallas de polígonos.

Una **malla de polígonos** (*polygon Mesh*) es un conjunto de puntos de un espacio afín que forman **caras** (*faces*) planas, usualmente adyacentes entre ellas, y que aproxima la frontera de un objeto en el espacio 3D

- El término **objeto** designa un conjunto de puntos como los descritos antes (de extensión finita, continuo), todos ellos en un mismo espacio afín.
- Una **cara** es un conjunto de puntos en un plano de dicho espacio afín, delimitados por un polígono.
- Las mallas aproxima una superficie, la cual
 - ▶ encierra completamente una región del espacio (el objeto tiene volumen), o bien
 - ▶ constituye en si misma el objeto, que tiene volumen nulo.

Las primeras son mallas ***cerradas*** y las segundas ***abiertas***.

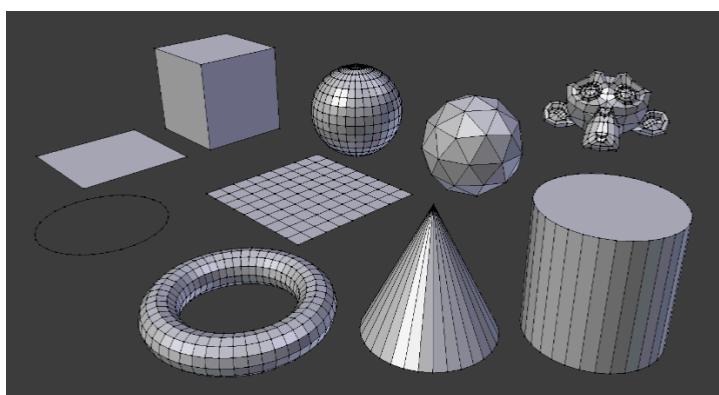
Subsección 2.1.

Introducción

2. Modelos de fronteras: mallas de polígonos..
2.1. Introducción.

Ejemplos de mallas

Aquí vemos varios ejemplos de mallas de polígonos (excepto la circunferencia que no lo es). Algunas son cerradas y otras abiertas:



Catálogo de objetos predefinidos de la aplicación Blender para modelado 3D:
docs.blender.org/manual/en/latest/modeling/meshes/primitives.html

Sesión 4: Modelos de Objetos. Mallas indexadas.

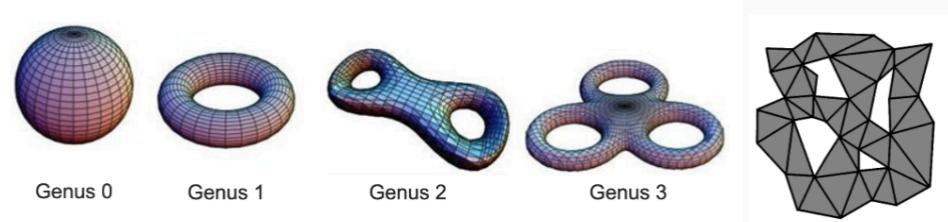
Created 2025-12-10

Page 18 / 91.

2. Modelos de fronteras: mallas de polígonos..
2.1. Introducción.

Características de las mallas

- Las mallas cerradas pueden tener cualquier **género topológico** (*genus*), aquí vemos mallas de género 0, 1, 2 y 3.
- Las mallas abiertas pueden tener huecos entre los polígonos:



Izquierda: Rudiger Westermann (Univ. Munich) - Computer Graphics Course slides
slideplayer.com/slide/4642205/

Derecha: [Stackoverflow](#)

Elementos de las mallas: vértices

Un **vértice (vertex)** es un par formado por un **punto** del espacio afín (en el extremo de alguna una arista), y un **valor entero único** (entre 0 y $n - 1$, donde n es el número de vértices de la malla).

Subsección 2.2.

Elementos y adyacencia.

- Al punto lo llamamos la **posición** del vértice.
- Al entero lo llamamos **índice** del vértice.
- Dos vértices distintos (con distinto índice) pueden tener la misma posición.
- Usar estos índices tiene estas ventajas:
 - ▶ Permiten expresar la **topología** de una malla independientemente de su **geometría**.
 - ▶ Facilita construir representaciones computacionales de las mallas con ciertas propiedades.

Elementos de las mallas: caras

Una **cara (face)** contiene un conjunto de puntos coplanares que están delimitados por un único polígono plano. Se determina por una secuencia ordenada de índices de vértices que forman dicho polígono.

- En la secuencia de índices, cada vértice comparte una arista con el siguiente (y el último con el primero). En esta secuencia
 - ▶ es indiferente cual índice es el primero, y
 - ▶ en principio, es indiferente en que sentido se recorren los vértices (solo hay dos posibilidades).
- Dos caras distintas no pueden tener asociado el mismo conjunto de índices, ni siquiera con distinto orden o empezando en distintos vértices.

Elementos de las mallas: aristas. Representación de mallas.

Una **arista (edge)** contiene el conjunto de puntos en un lado del polígono que delimita una cara, puntos que forman un segmento de recta. Se determina por un par único de índices de vértices.

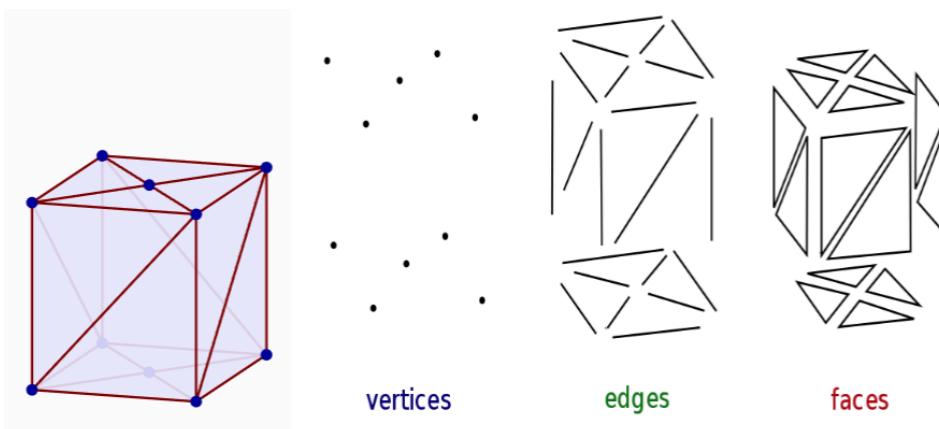
- Los dos índices de vértice de una arista no pueden coincidir.
- El orden en el que aparecen los índices en el par es irrelevante (las aristas no están orientadas).
- Dos aristas distintas no pueden tener el mismo par de índices de vértice, ni siquiera en distinto orden.

Esto implica que **una malla viene determinada por**:

- la secuencia $\{p_0, p_1, \dots, p_{n-1}\}$ de **posiciones** de sus n vértices.
- la secuencia de **caras**, cada una de ellas representada como una secuencia de k índices de vértice: $\{i_0, \dots, i_{k-1}\}$ (k puede ser distinto en cada cara).

Vértices, caras y aristas

Elementos de una malla que forman la frontera de un paralelepípedo:



Imágenes de la derecha tomadas de: [Wikipedia: Polygon Mesh](#).

Geometría y topología de las mallas

Una malla tiene una geometría y una topología:

Geometría: conjunto de puntos que están en alguna cara (eso incluye los puntos que están en alguna arista y las posiciones de los vértices).

Topología: conjunto de relaciones de adyacencia entre vértices, aristas y caras (sin tener en cuenta la geometría, es decir, considerando únicamente los índices de los vértices).

Esta definición permite que dos mallas distintas:

- Tengan la misma topología pero distinta geometría (p.ej., partimos de una malla y cambiamos las posiciones de sus vértices, pero mantenemos las adyacencias).
- Tengan la misma geometría pero distinta topología (p.ej., partimos de una malla con caras de cuatro aristas y dividimos cada cara en dos caras triángulares coplanares).

Adyacencia entre elementos de una malla

En una malla existen relaciones binarias de adyacencia entre estos elementos:

- Un vértice en el extremo de una arista es adyacente a la arista (por tanto, toda arista es adyacente a exactamente dos vértices).
- Una arista y una cara son adyacentes si la arista forma parte del polígono que delimita la cara.
- Dos vértices son adyacentes si hay una arista adyacente a ambos.
- Dos caras son adyacentes si hay una arista adyacente a ambas.
- Un vértice y una cara son adyacentes si hay una arista adyacente a ambos.

Puesto que los vértices están numerados, las relaciones de adyacencia se pueden expresar en términos de los índices de los vértices.

Características de las 2-variedades

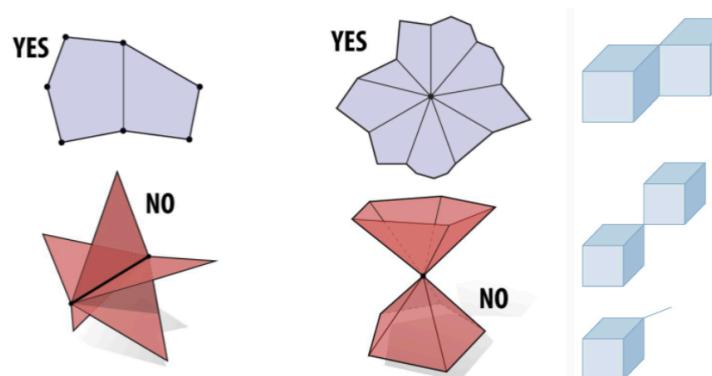
Vamos a usar exclusivamente mallas que son una **2-variedad (2-manifold)**, esto implica que:

- Un vértice siempre **es adyacente a dos aristas como mínimo** (no hay vértices aislados).
- Una arista siempre **es adyacente a una o a dos caras** (no hay aristas aisladas, ni aristas adyacentes a 3 o más caras).
- Todas las caras adyacentes a un vértice **se pueden ordenar en una secuencia en la cual cada cara es adyacente a la siguiente**.

Estas propiedades aseguran, entre otras cosas, que se pueden asignar ciertos atributos a cada vértice de forma única (por ejemplo, normales y coordenadas de textura), ya que el entorno de un punto de la superficie siempre es *equivalente a un plano*.

Ejemplos de mallas que no son 2-variedades

Varias mallas, dos representan 2-variedades y el resto no:



Izquierda: Carnegie Mellon Computer Graphics Course: slides (fall 2018):
15462.cs.cmu.edu/spring2018/lecture/meshes

Derecha: J.F. Hughes at al.: Computer Graphics: Principles and Practice (3rd ed.)

Conversión en 2-variedad

La topología de una malla que no es una 2-variedad puede modificarse para que lo sea, manteniendo la geometría:

- Se puede conseguir **replicando vértices**, es decir, añadiendo nuevos vértices con índices distintos en la misma posición de otros vértices ya existentes

En el ejemplo de la transparencia anterior, los dos conos unidos por el ápice se separan al insertar dos ápices en la misma posición, un ápice por cono.

- Esto puede implicar a veces también **replicar aristas**

En el ejemplo de la transparencia anterior: los dos cubos unidos con una arista en común se separan duplicando los dos vértices de dicha arista y la propia arista.

Aristas y vértices de frontera

Una arista es una **arista de frontera** (o de borde) (*boundary edge* o *border edge*) si es adyacente a una única cara.

Un vértice es un **vértice de frontera** si es adyacente a alguna arista de frontera.

A la derecha vemos una malla con diversos vértices y aristas de frontera (en amarillo)

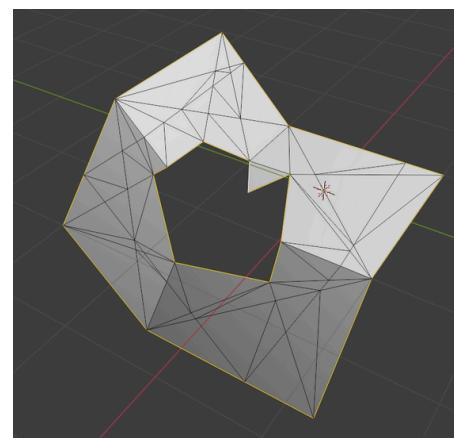


Imagen de StackOverflow:
stackoverflow.com/questions/78359671

Mallas abiertas y cerradas

Una malla es **cerrada** si y solo si no tiene aristas de frontera (todas las aristas son adyacentes a exactamente dos caras). Es *topológicamente equivalente* a una esfera (a la derecha).

Una malla es **abierta** si tiene al menos una cara de frontera (a la izquierda, las aristas de frontera están en rojo)

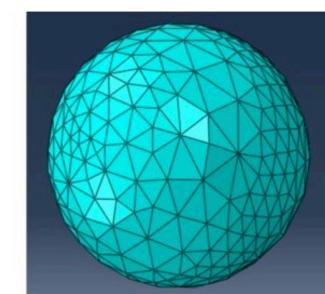
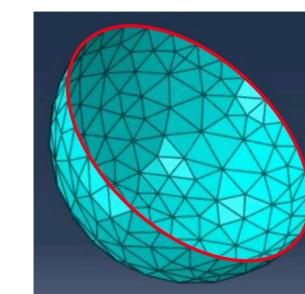


Imagen de Fangtao Yang: [Researchgate](#)

Lados y orientación de una cara.

En una cara se pueden identificar dos lados distintos, en función del orden en que aparecen los vértices en la secuencia de índices que define la cara:

- **Lado horario** es el lado en el cual se ven los vértices en orden de las agujas del reloj.
- **Lado antihorario** es el lado en el cual se ven los vértices en sentido contrario al de las agujas del reloj.

Cuando una cara se visualiza en una imagen **desde un punto de vista concreto**, se verá únicamente uno de los dos lados:

- Si se ve el lado horario, se dice que la cara **aparece orientada en sentido horario**.
- Si se ve el lado antihorario, se dice que la cara **aparece orientada en sentido antihorario**

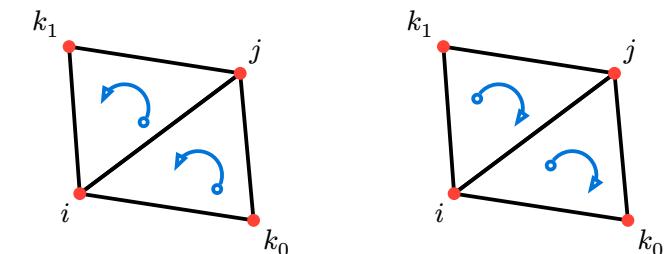
Cribado de caras traseras

Las APIs de rasterización suelen hacer una clasificación de las caras al visualizar una imagen. Según la orientación con que aparecen, se clasifican en **delanteras o traseras**.

- Esto permite el **cribado de caras traseras (*back face culling*)**, consiste en **visualizar en una imagen únicamente las caras delanteras**.
- Esto sirve para visualizar una malla cerrada opaca desde un punto de vista exterior a la malla, sin perder tiempo en rasterizar las caras que no se van a ver con seguridad, ya que están en la parte trasera (no visible) del objeto (las caras traseras).
- Este comportamiento se puede activar, desactivar, o configurar (se puede establecer que las caras delanteras son las de orientación horaria o antihoraria).
- En Godot el cribado está activado por defecto, y las caras delanteras son las que **aparecen en sentido horario**. Se puede configurar para cada malla.

Orientación coherente de las mallas.

Una malla tendrá **orientación coherente** si dadas dos caras adyacentes comparando una arista entre los vértices i y j , entonces, en una cara j es el siguiente a i , y en la otra cara i es el siguiente a j .



En un ejemplo de dos caras coplanares, es legal que se vean ambas caras orientadas en sentido horario, o ambas en sentido antihorario, pero no cada una con una orientación.

Marco de referencia de la malla.

La posición de cada vértice se representa en el ordenador por sus coordenadas respecto de un marco de referencia cartesiano \mathcal{R} único

- A dicho marco de referencia se le denomina **marco de referencia local de la malla**
- El i -ésimo vértice (en el punto p_i) tiene coordenadas $\mathbf{c}_i = (x_i, y_i, z_i, 1)$ en el marco \mathcal{R} , es decir:

$$p_i = \mathcal{R}c_i = \mathcal{R}(x_i, y_i, z_i, 1)^T$$

- A la tupla c_i se le denomina las **coordenadas locales** del vértice i -ésimo.
- No se suele almacenar en memoria la componente w ya que siempre es 1, aunque a veces se podría hacer para acortar algo el tiempo de procesamiento (a costa de usar más memoria).

Atributos de las mallas

Como modelos de objetos reales, las mallas suelen incluir más información geométrica o del aspecto del objeto:

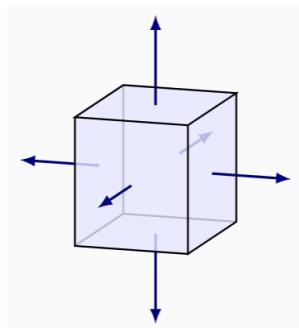
- **Normales (normals):** vectores de longitud unidad
 - ▶ **Normales de caras:** vector unitario perpendicular a cada cara, de longitud unidad, apuntando al exterior de la malla si es una malla cerrada. Precalculado a partir del polígono.
 - ▶ **Normales de vértices:** vector unitario perpendicular al plano tangente a la superficie en la posición del vértice.
- **Colores:** ternas (usualmente RGB) con tres valores entre 0 y 1.
 - ▶ **Colores de caras:** útil cuando cada cara representa un trozo de superficie de color homogéneo.
 - ▶ **Colores de vértices:** color de la superficie en cada vértice (la superficie varía de color de forma continua entre vértices).
- Otros atributos: coords. de textura, vectores tangente y bitangente, etc...

Subsección 2.3.

Atributos de vértices.

Normales de caras

Pueden ser útiles cuando el objeto que se modela con la malla está realmente compuesto de caras planas (p.ej., un cubo), o bien cuando se quiere hacer *sombreado plano*:



Para un polígono (con dos aristas \vec{a}, \vec{b} , vectores distintos, no nulos), su normal \vec{n} se define como:

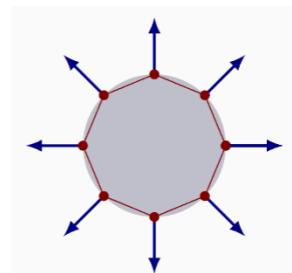
$$\vec{n} = \frac{\vec{m}}{\|\vec{m}\|} \quad \text{donde} \quad \vec{m} = \vec{a} \times \vec{b}$$

En estos casos la normal se puede precalcular y almacenar en la malla para lograr eficiencia en tiempo de visualización.

Normales de vértices para superficies suaves:

Tienen sentido cuando la malla aproxima una superficie curvada:

- A veces la superficie original es conocida, y las normales se definen fácilmente (p.ej. una esfera).
- Si la superficie original es desconocida, las normales se pueden definir exclusivamente usando la malla



Para un vértice (con k caras adyacentes) su normal \vec{n} se define como:

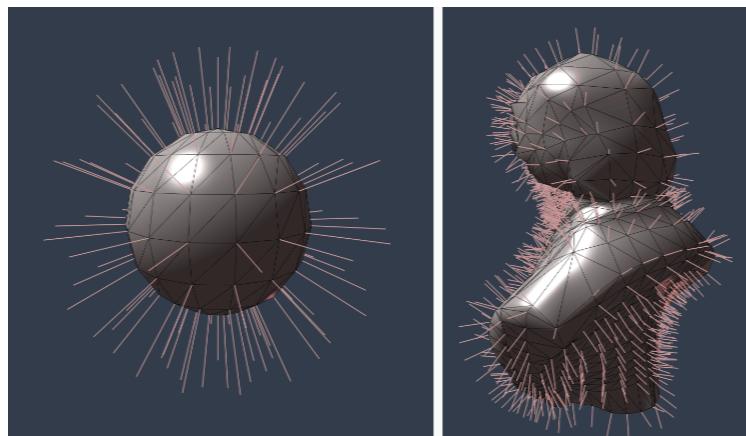
$$\vec{n} = \frac{\vec{s}}{\|\vec{s}\|} \quad \text{donde} \quad \vec{s} = \sum_{i=0}^{k-1} \vec{m}_i$$

donde $\vec{m}_0, \vec{m}_1, \dots, \vec{m}_{k-1}$ son las normales de las caras adyacentes al vértice.

Las coordenadas de estas normales también se pueden precalcular y almacenar.

Ejemplos de normales de vértices calculadas

Aquí vemos visualizadas las normales de una esfera de baja resolución (calculadas analíticamente), y de una malla arbitraria (calculadas promediando normales de caras):



Colores de vértices

En algunos casos, es conveniente asignar colores RGB a los vértices. La utilidad más frecuente de esto es hacer interpolación de color en las caras durante la visualización:

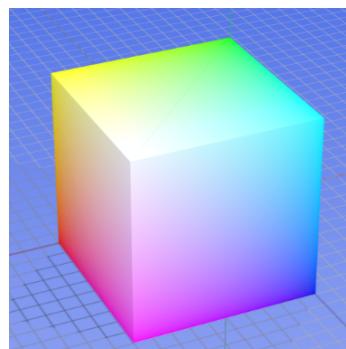
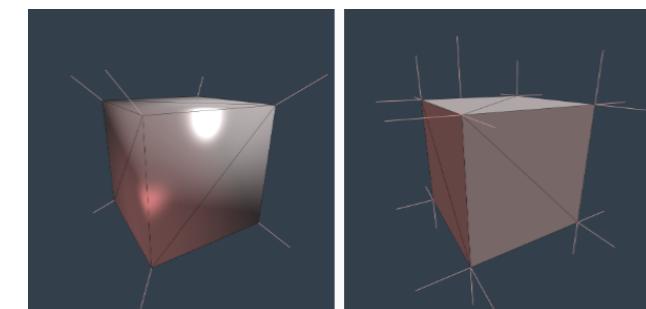


Imagen: en.wikipedia.org/wiki/File:RGB_color_solid_cube.png (Wikimedia Commons)

Discontinuidades de la normal

Algunos objetos reales presentan aristas o vértices donde la normal es discontinua (p.ej. un cubo), en ese caso promediar normales es mala idea, y es necesario replicar vértices y aristas (y después promediar):



El cubo de la izquierda tiene 8 vértices y el de la derecha 24 vértices. La iluminación es correcta a la derecha. El mismo problema puede aparecer con las coordenadas de textura, o los colores.

Sección 3. Representación de modelos de fronteras.

1. Triángulos aislados.
2. Tiras de triángulos.
3. Mallas indexadas de triángulos.
4. Aristas aladas
5. Formatos para archivos con mallas indexadas.

Representación en memoria

En esta sección veremos distintas formas de representar las mallas en la memoria de un ordenador:

- **Triángulos aislados, tiras de triángulos:** no representan explícitamente la topología.
- **Mallas indexadas:** lo más común, representan explícitamente la topología.
- **Aristas aladas:** extensión de las mallas indexadas para eficiencia en tiempo.

Godot está diseñado para visualizar directamente los triángulos aislados, las tiras de triángulos y las mallas indexadas, pero no las aristas aladas.

Por simplicidad, nos restringimos a **caras triángulares**, que es lo más común en la inmensa mayoría de las aplicaciones.

Subsección 3.1.
Triángulos aislados.

Tabla de triángulos aislados.

La más simple es usar una lista o tabla de **triángulos aislados**. La malla se representa como un vector o lista con tres entradas (tres variables de tipo **Vector3** o **Vector2**) para cada triángulo:

Malla TA (n triángulos)		
	x_0	y_0
0	x_0	y_0
1	x_1	y_1
2	x_2	y_2
3	x_3	y_3
4	x_4	y_4
5	x_5	y_5
:	:	
$3n - 3$	x_{3n-3}	y_{3n-3}
$3n - 2$	x_{3n-2}	y_{3n-2}
$3n - 1$	x_{3n-1}	y_{3n-1}

- Para cada triángulo se almacenan las coordenadas locales de cada uno de sus tres vértices (9 valores flotantes en total).
- La tabla se puede almacenar en memoria con todas las coordenadas contiguas.
- En total, incluye $9n$ valores flotantes.

Representación en Godot

La malla se representa como un array empaquetado de Godot con tres entradas (tres variables de tipo **Vector2** o **Vector3**) para cada triángulo:

```
var posiciones : PackedVector3Array = [
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0), # Triángulo 1
    Vector3(1,0,0), Vector3(1,1,0), Vector3(0,1,0), # Triángulo 2
    ...
]
```

También se puede usar un array normal de Godot, pero entonces debe convertirse a un array empaquetado antes de añadir las tablas a un objeto **Mesh** para agregarlo a un nodo:

```
var posiciones : Array[Vector3] = [
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0), # Triángulo 1
    Vector3(1,0,0), Vector3(1,1,0), Vector3(0,1,0), # Triángulo 2
    ...
]
var posiciones_em := PackedVector3Array( posiciones ) # ctor específico
```

Valoración.

Esta representación es **poco eficiente en tiempo y memoria**:

- Si un vértice es adyacente a k triángulos, sus coordenadas aparecen repetidas k veces en la tabla y se procesan k veces al visualizar.
- En una malla típica que representa una rejilla de triángulos, los vértices internos (la mayoría) son adyacentes a 6 triángulos, es decir, cada tupla aparece casi 6 veces como media.

Además, **no hay información explícita sobre la topología** de la malla:

- La topología se puede calcular comparando coordenadas de vértices, pero tendría una complejidad en tiempo cuadrática con el número de vértices y es poco robusto.

En algunos casos muy particulares, podría ser útil (objetos realmente compuestos de muchos triángulos realmente aislados, objetos muy sencillos).

Subsección 3.2. Tiras de triángulos.

Tiras de triángulos: motivación y características

Las representación en memoria usando **tiras de triángulos (triangle strip)** pretende reducir la memoria y el tiempo que necesitan la representación de triángulos aislados:

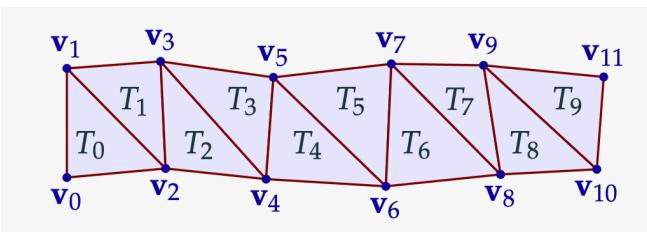
- Para conseguir esto, esta representación reduce el número de veces que aparecen replicadas unas coordenadas en memoria.
- Como consecuencia, se reduce el tiempo de procesamiento.

Sin embargo, esta representación

- No evita totalmente las redundancias (se siguen repitiendo coordenadas de vértices, aunque menos).
- Tampoco incluye información explícita sobre la topología de la malla.

Tiras de triángulos en una malla.

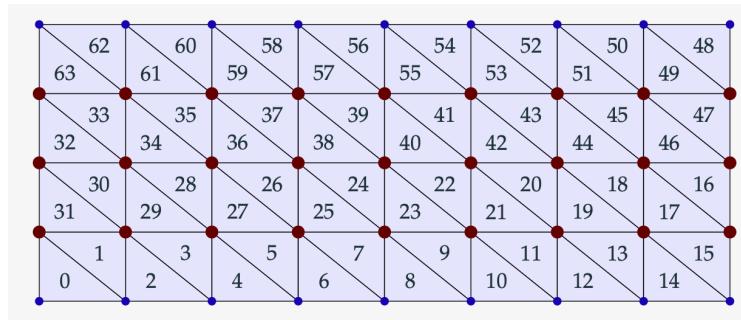
Podemos identificar una parte de una malla como una **tira de triángulos**: cada triángulo T_{i+1} en la secuencia es adyacente al anterior T_i , con lo cual T_{i+1} comparte con T_i una arista y sus dos vértices v_i y v_{i+1} , vértices cuyas coordenadas no tienen que ser repetidas en memoria:



- Cada tira de n triángulos necesita $n + 2$ tuplas de coordenadas de vértices (tres para el primer triángulo y después una más por cada triángulo adicional)
- Se almacena una tabla que en la i -ésima entrada almacena las coordenadas del i -ésimo vértice.

Tiras de triángulos para mallas no simples

En la mayoría de los casos, las tiras obligan a repetir algunas coordenadas de vértices (aunque en mucho menor grado que los triángulos aislados). Ejemplo de una tira en zig-zag:



- Las coords. de los vértices en rojo (grandes) se repiten dos veces.
- Las de los vértices en azul (pequeños) aparecen una sola vez.

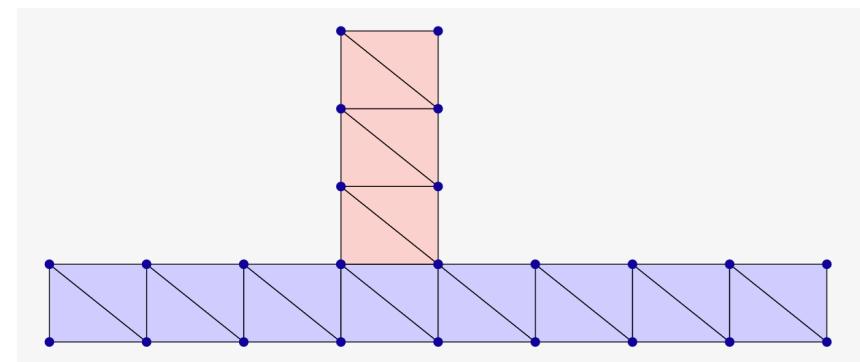
Representación en memoria

Una malla es una estructura con varias tiras. La tira número i (con n_i triángulos) es un array con $n_i + 2$ celdas, en cada una están las coordenadas maestras de un vértice.

Tira 0 (n_0 triángulos)	Tira 1 (n_1 triángulos)	Tira 2 (n_2 triángulos)
$x_0 \quad y_0 \quad z_0$ $x_1 \quad y_1 \quad z_1$ $x_2 \quad y_2 \quad z_2$ $x_3 \quad y_3 \quad z_3$ $x_4 \quad y_4 \quad z_4$ $\vdots \quad \vdots \quad \vdots$ $x_{n_0} \quad y_{n_0} \quad z_{n_0}$ $x_{n_0+1} \quad y_{n_0+1} \quad z_{n_0+1}$ $x_{n_0+2} \quad y_{n_0+2} \quad z_{n_0+2}$	$x_0 \quad y_0 \quad z_0$ $x_1 \quad y_1 \quad z_1$ $x_2 \quad y_2 \quad z_2$ $x_3 \quad y_3 \quad z_3$ $x_4 \quad y_4 \quad z_4$ $\vdots \quad \vdots \quad \vdots$ $x_{n_1} \quad y_{n_1} \quad z_{n_1}$ $x_{n_1+1} \quad y_{n_1+1} \quad z_{n_1+1}$ $x_{n_1+2} \quad y_{n_1+2} \quad z_{n_1+2}$	$x_0 \quad y_0 \quad z_0$ $\vdots \quad \vdots \quad \vdots$ $x_{n_2+2} \quad y_{n_2+2} \quad z_{n_2+2}$

Mallas con varias tiras

En algunos casos, es inevitable tener que recurrir a más de una tira para una única malla:



Por este motivo la implementación de una malla con tiras debe prever más de una tira en la misma malla.

Tiras de triángulos: valoración

La mejora de las tiras frente a los triángulos aislados es que usan menos memoria, sin embargo, tiene estos inconvenientes:

- Al requerir probablemente más de una tira de triángulos, la representación es algo más compleja.
- Se necesitan algoritmos (complejos) para calcular las tiras a partir de una malla representada de alguna otra forma. Se intenta optimizar de forma que el número de coordenadas a almacenar sea el menor posible.
- El numero promedio de veces que se repite cada coordenada en memoria es prácticamente siempre superior a la 1, y cercano a 2.
- Esta representación tampoco incorpora información explícita sobre la conectividad.

Tiras de triángulos: Implementación

Se pueden representar con un array de arrays, cada uno de los segundos con una tira:

```
var posiciones : Array[Array] = [
  [ # Tira 1
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0),
    Vector3(1,1,0), Vector3(2,0,0), ....
  ],
  [ # Tira 2
    Vector3(3,0,0), Vector3(4,0,0), Vector3(3,1,0),
    Vector3(4,1,0), ....
  ],
  .... ## otras tiras..
]
```

Para crear un nodo con una de estas mallas, es necesario construir un objeto **Mesh** por cada tira, y crear un nodo de tipo **MeshInstance3D** para cada uno de ellos. Los nodos **MeshInstance3D** se añaden como hijos del nodo que contiene la malla completa.

Mallas Indexadas.

Para solucionar los problemas de uso de memoria y tiempo de procesamiento de las soluciones anteriores, se puede usar una estructura con dos tablas:

- **Tabla de vértices:** tiene una entrada por cada vértice, incluye sus coordenadas
- **Tabla de triángulos:** tiene una entrada por triángulo, incluye los índices de sus tres vértices en la tabla anterior.

En esta solución:

- **No se repiten coordenadas de vértices:** se ahorra memoria y se puede visualizar sin repetir cálculos (se repiten índices enteros).
- **Hay información explícita de la topología (conectividad):** se almacenan explícitamente los vértice adyacentes a un triángulo y se pueden calcular fácilmente el resto de adyacencias.

Subsección 3.3.
Mallas indexadas de triángulos.

Estructura de datos

La tabla de triángulos (para n triángulos), almacena un total de $3n$ índices de vértices (enteros sin signo), y la de vértices $3m$ valores reales:

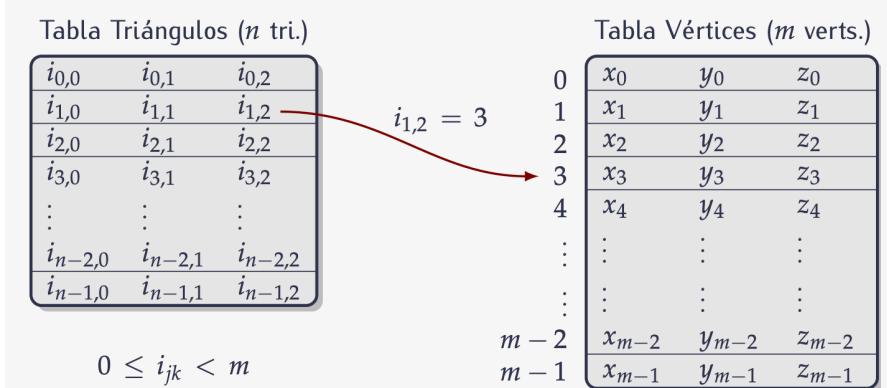


Tabla Triángulos (n tri.)			Tabla Vértices (m verts.)		
$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	x_0	y_0	z_0
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$	x_1	y_1	z_1
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$	x_2	y_2	z_2
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$	x_3	y_3	z_3
\vdots	\vdots	\vdots	x_4	y_4	z_4
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$	\vdots	\vdots	\vdots
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$	x_{m-2}	y_{m-2}	z_{m-2}
			x_{m-1}	y_{m-1}	z_{m-1}

$0 \leq i_{jk} < m$

Implementación de las mallas indexadas

En Godot se pueden usar arrays de Godot , es útil se queremos manipular la malla mediante algoritmos:

```
var posiciones : Array[Vector3] = [
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0), ....
]
var triangulos : Array[Vector3i] = [
    Vector3i(0,1,2), Vector3i(1,3,2), Vector3i(1,4,3), ....
]
```

Para añadir la malla a un nodo, habrá que convertir estos array a empaquetados, se puede hacer así:

```
var pos_em : PackedVector3Array( posiciones ) # ctor específico
var tri_em : PackedInt32Array([]) # inicialmente vacía

for t in triangulos: # añadimos índices con un bucle
    tri_em.append( t[0] ); tri_em.append( t[1] ); tri_em.append( t[2] )
```

Tiras de triángulos indexadas

Es posible representar una malla como una tabla de vértices y varias tiras de triángulos. Cada tira almacena índices de vértices en lugar de coordenadas de vértices.

- Las coordenadas no se repiten en memoria.
- Se repiten en memoria los índices de vértices, pero menos veces que con tabla de vértices y triángulos.
- El modelado usando tiras es más complejo.

Se pueden añadir a un nodo usando el tipo de primitiva tiras y además usando una tabla de índices.

- Las coordenadas de vértice se envían y se procesan una sola vez
- Los índices de vértices se envían repetidos, pero solo un par de veces de media aprox.

Motivación

La estructura de malla indexada permite, por ejemplo, consultar con tiempo en $O(1)$ si un vértice es adyacente a un triángulo, pero:

- Para consultar si dos vértices son adyacentes, hay que buscar en la tabla de triángulos si los vértices aparecen contiguos en alguno: esto requiere un tiempo en $O(n_t)$.
- No se guarda información de las aristas. Las consultas relativas a aristas se resuelven también en $O(n_t)$.
- En general, las consultas sobre adyacencia son costosas en tiempo.

Para poder reducir los tiempos de cálculo de adyacencia a $O(1)$, se puede usar más memoria de la estrictamente necesaria para la malla indexada. Veremos la estructura de **aristas aladas** (para mallas que encierran un volumen, es decir: siempre hay **dos caras adyacentes a una arista**, no necesariamente triangulares)

Estructura de aristas aladas: tabla de aristas.

Una malla se puede codificar usando una tabla de vértices (**tver**) (similar a la de las mallas indexadas), mas una tabla de aristas (**tari**). Esta última:

- Tiene una entrada por cada arista, con dos índices:

- ▶ **vi** = índice de **vértice inicial**
- ▶ **vf** = índice de **vértice final**

(es indiferente cual se selecciona como inicial y cual como final).

- Tambien tiene (cada arista es adyacente a dos triángulos):

- ▶ **ti** = índice del **triángulo a la izquierda**
- ▶ **td** = índice del **triángulo a la derecha**

(izquierda y derecha entendidos según se recorre la arista en el sentido que va desde vértice inicial al final)

- Esto permite consultas en O(1) sobre adyacencia **arista-vértice** y **arista-tríangulo**.

Aristas siguiente y anterior

Además de los datos anteriores, se guarda, para cada arista, los índices de otras cuatro adyacentes a ella.

- Si se recorren las aristas del triángulo de la izquierda aparecerá la arista en cuestión entre otras dos, cuyos índices se guardan en la tabla:

- ▶ **aai** = índice de la **arista anterior**
- ▶ **asi** = índice de la **arista siguiente**

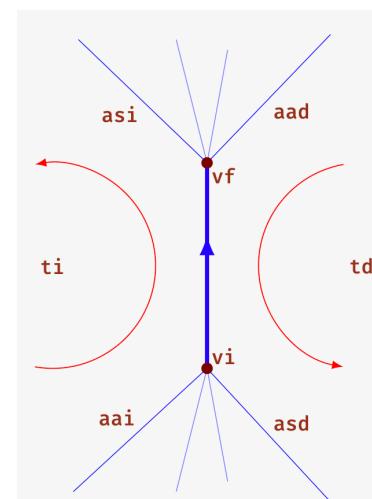
(el recorrido de las aristas se hace en sentido anti-horario cuando se observa el triángulo desde el exterior de la malla).

- Igualmente, se guardan los dos índices de arista anterior y siguiente relativas al recorrido anti-horario del triángulo de la derecha:

- ▶ **aad** = índice de la **arista anterior** (derecha)
- ▶ **asd** = índice de la **arista siguiente** (derecha)

Índices asociados a una arista

Índices (valores naturales) en la entrada correspondiente a la arista vertical en una malla (no necesariamente de triángulos):



Uso de las aristas siguiente y anterior.

El hecho de almacenar las aristas siguiente y anterior permite hacer recorridos por las entradas de la tabla de aristas siguiendo esos índices:

- Dada una arista y un triángulo adyacente (el izquierdo o el derecho), se pueden obtener estas listas:
 - ▶ aristas adyacentes al triángulo.
 - ▶ vértices adyacentes al triángulo
- Dada una arista y un vértice adyacente (el inicial o el final), se pueden obtener estas listas:
 - ▶ aristas que inciden en el vértice (esto permite resolver fácilmente adyacencias **arista-arista**)
 - ▶ triángulos adyacentes al vértice.

Con toda esta información (los 8 valores), se puede resolver directamente cualquier adyacencia que involucre una arista al menos (consultando su entrada). Aun no podemos resolver el resto.

Tablas adicionales. Uso.

Para hacer todas las consultas en $O(1)$, añadimos **taver** y **tatri**:

- **taver** = tabla de **aristas de vértice**: para cada vértice, almacenamos el índice de una arista adyacente cualquiera:
 - ▶ dado un vértice, permite recuperar todas las aristas, vértices y triángulos adyacentes.
 - ▶ por tanto, permite consultas de adyacencia **vértice-vértice** y **vértice-triángulo**.
- **tatri** = tabla de **aristas de triángulo**: para cada triángulo, se almacena el índice de una arista cualquiera adyacente:
 - ▶ dado un triángulo, permite recuperar todas las aristas, vértices y triángulos adyacentes,
 - ▶ por tanto, permite consultas de adyacencia **triángulo-triángulo** y **vértice-triángulo** (esta última se puede hacer de dos formas).

Subsección 3.5.

Formatos para archivos con mallas indexadas.

Formato PLY

El formato PLY fue diseñado por Greg Turk y otros en la Univ. de Stanford a mediados de los 90. Codifica una malla indexada en un archivo ASCII o binario (usaremos la versión ASCII). Tiene tres partes:

Cabecera: describe los atributos presentes y su formato, se indica el número de vértices y caras, ocupa varias líneas.

Tabla de vértices: un vértice por línea, se indican sus coordenadas X, Y y Z (flotantes) en ASCII, separadas por espacios.

Tabla de caras: una cara por línea, se indica el número de vértices de la cara, y después los índices de los vértices de la cara (comenzando en cero para el primer vértice de la tabla de vértices).

El formato es extensible de forma que un archivo puede incluir otros atributos (p.ej., colores de vértices). Su simplicidad hace fácil usarlo.

Ejemplo de archivo PLY: cabecera

En esta cabecera de ejemplo se indica que:

- hay 8 elementos **vertex** y 6 elementos **face** (caras),
- cada vértice tiene 3 **propiedades**, tipo (**float**) llamadas **x**, **y**, **z**,
- cada cara es una lista, primero su longitud (**uchar**) y luego una serie de enteros (**int**).

```
ply
format ascii 1.0
comment Archivo de ejemplo del formato PLY (8 vertices y 6 caras)
element vertex 8
property float x
property float y
property float z
element face 6
property list uchar int vertex_index
end_header
```

Ejemplo de archivo PLY: tablas de vértices y caras

A continuación vendría la tabla de vértices, con 8 líneas (una por vértice):

```
0.0 0.0 0.0
0.0 0.0 1.0
0.0 1.0 0.0
0.0 1.0 1.0
1.0 0.0 0.0
...
...
```

Y finalmente la tabla de caras, con 6 líneas (una por cara), en cada línea se indica el número de vértices de la cara seguido de los correspondientes índices (en este ejemplo todas son triángulos, por lo que el primer número es siempre 3):

```
3 0 1 2
3 0 3 1
3 4 0 1
3 4 1 5
...
...
```

Tablas de vértices, normales y coordenadas de textura

En un archivo **.obj**, cada línea puede comenzar con: **v** para vértices (coordenadas X,Y,Z), **vn** para normales (componentes X,Y,Z), **vt** para coordenadas de textura (componentes U,V). Por ejemplo:

```
v 0.123 0.234 0.345 1.0
v ...
...
vt 0.500 1 [0]
vt ...
...
vn 0.707 0.000 0.707
vn ...
...
```

Los vértices, normales y coordenadas de textura se numeran empezando en 0, **de forma independiente** entre ellos, es decir la numeración de los vértices es independiente de la de normales y de la de coordenadas de textura.

El formato OBJ

El formato **OBJ** (también llamado *Wavefront OBJ*) fue ideado por la empresa *Wavefront*, y se usa bastante hoy en día, es parecido a PLY, pero **con las normales y coordenadas de textura indexadas**:

- Incluye una tabla de vértices y una tabla de triángulos, igual que PLY
- Además, incluye tablas normales y coordenadas de textura. A diferencia de PLY, su tamaño no tiene porque coincidir con el de la tabla de vértices.
- En cada cara, cada vértice se representa por un índice de sus coordenadas de posición, y opcionalmente, otros índices independientes para su normal y sus coordenadas de textura.
- Permite añadir información de materiales y texturas (en archivos externos, con extensión **.mtl** y formato MTL, *Materials Template Library*).
- Godot puede importar archivos OBJ directamente.
- Librerías para carga disponibles: *Assimp*, *TinyOBJLoader*, etc.

Tabla de caras en formato OBJ

En un archivo **.obj**, cada cara se representa con una línea que comienza con **f**, seguida de una serie de grupos separados por espacios, uno por cada vértice de la cara. Cada grupo tiene la forma:

v_idx/vt_idx/vn_idx

donde **v_idx** es el índice del vértice en la tabla de vértices, **vt_idx** es el índice de las coordenadas de textura en la tabla de coordenadas de textura, y **vn_idx** es el índice de la normal en la tabla de normales. Los índices comienzan en 1.

Ejemplo de una cara triangular con vértices con índices 1, 8 y 16, coordenadas de textura con índices 34, 45 y 56, y normales con índices 0, 0 y 1:

```
f 1/34/0 8/45/0 16/56/1
```

Si no se usan coordenadas de textura o normales, los grupos pueden tener las formas **v_idx//vn_idx** o simplemente **v_idx**.

Formato OBJ: valoración

La ventaja frente a PLY (malla indexada de triángulos) es una mayor flexibilidad, lo que permite mayor eficiencia en memoria:

- Dos vértices en posiciones distintas pueden compartir normal (por ejemplo, una malla plana puede tener una única normal, en lugar de tantas como vértices)
- Un vértice único puede tener distintas coords. de textura o distintas normales en distintas caras, no es necesario replicarlo (por ejemplo, un cubo puede tener 8 vértices y solo 6 normales).

La principal desventaja es que las APIs de rasterización no pueden visualizar directamente este tipo de tablas, así que es necesario convertirlas a una malla indexada de triángulos, replicando normales y coordenadas de textura.

Tabla de aristas

En una malla indexada podría ser conveniente (para ganar tiempo de procesamiento en ciertas aplicaciones) almacenar explícitamente las aristas (ahora esa info. está implícita en la tabla de triángulos). Se puede hacer usando un **tabla de aristas**:

- Contiene una entrada por cada arista.
- En cada entrada hay dos índices (naturales) de vértices (los dos vértices en los extremos de la arista).
- El orden de los vértices en cada entrada es irrelevante, pero las aristas no deben estar duplicadas.

La disponibilidad de esta tabla permite, por ejemplo, dibujar en modo alambre con begin/end sin repetir dos veces el dibujo de aristas adyacentes a dos triángulos.

Los formatos glTF y GLB

El formato **glTF** (*GL Transmission Format*) es un formato estándar abierto para la transmisión y almacenamiento de modelos 3D y escenas. Fue desarrollado por el Khronos Group y está diseñado para ser eficiente en términos de tamaño de archivo y velocidad de carga.

- **glTF** utiliza una estructura basada en JSON para describir la geometría, materiales, texturas, animaciones y otros aspectos de un modelo 3D.
- El formato **GLB** es una versión binaria de glTF que empaqueta todos los datos en un solo archivo binario, lo que facilita su distribución y uso en aplicaciones web y móviles.
- Ambos formatos son ampliamente compatibles con motores gráficos modernos, incluyendo Godot, Unity y Unreal Engine.
- glTF/GLB soporta mallas indexadas, normales, coordenadas de textura, materiales PBR (*Physically Based Rendering*) y animaciones esqueléticas.

Problema: comparación de eficiencia en memoria (1/2)

Problema 4.1:

Supongamos que queremos codificar una esfera de radio $1/2$ y centro en el origen de dos formas:

- Por enumeración espacial, dividiendo el cubo que engloba a la esfera en celdas, de forma que haya k celdas por lado del cubo, todas ellas son cubos de $1/k$ de ancho. Cada celda ocupa un bit de memoria (si su centro está en la esfera, se guarda un 1, en otro caso un 0).
- Usando un modelo de fronteras (una malla indexada de triángulos), en el cual se usa una rejilla de triángulos y aristas que siguen los meridianos y paralelos, habiendo en cada meridiano y en cada paralelo un total de k vértices (se guarda únicamente la tabla de vértices y la de triángulos).

(continua en la siguiente transparencia)

Problema: comparación de eficiencia en memoria (2/2)

Problema 4.1 (continuación):

Asumiendo que un **float** y un **int** ocupan 4 bytes cada uno, contesta a estas cuestiones:

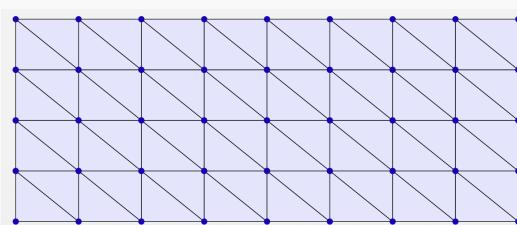
- Expresa el tamaño de ambas representaciones en bytes como una función de k .
- Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
- Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)

Compara los tamaños de ambas representaciones en ambos casos ($k = 16$ y $k = 1024$).

Problema: uso de memoria en mallas indexadas (1/2)

Problema 4.2:

Considera una malla indexada (tabla de vértices y caras, esta última con índices de vértices) con topología de rejilla como la de la figura, en la cual hay n columnas de pares de triángulos y m filas (es decir, hay $n + 1$ filas de vértices y $m + 1$ columnas de vértices, con $n, m > 0$, en el ejemplo concreto de la figura, $n = 8$ y $m = 4$).



(continua en la siguiente transparencia)

Problema: uso de memoria en mallas indexadas (2/2)

Problema 4.2 (continuación):

En relación a este tipo de mallas, responde a estas dos cuestiones:

- Supongamos que un **float** ocupa 4 bytes (igual a un **int**) ¿ que tamaño en memoria ocupa la malla completa, en bytes ? (tener en cuenta únicamente el tamaño de la tabla de vértices y triángulos, suponiendo que se almacenan usando los tipos **float** e **int**, respectivamente). Expresa el tamaño como una función de m y n .
- Escribe el tamaño en KB suponiendo que $m = n = 128$.
- Supongamos que m y n son ambos grandes (es decir, asumimos que $1/n$ y $1/m$ son prácticamente 0). deduce que relación hay entre el número de caras n_C y el número de vértices n_V en este tipo de mallas.

Problema: uso de memoria en tiras y mallas indexadas (1/2)

Problema 4.3:

Imagina de nuevo una malla como la del problema anterior, supongamos que usamos una representación como tiras de triángulos, de forma que cada fila de triángulos (con $2n$ triángulos) se almacena en una tira, habiendo un total de m tiras.

La tabla de punteros a tiras tiene un entero (el número de tiras) y m punteros, cada puntero suponemos que tiene 8 bytes de tamaño. De nuevo, asume que las coordenadas son de tipo **float** (4 bytes).

Responde a estas cuestiones:

(continua en la siguiente transparencia)

Problema: uso de memoria en tiras y mallas indexadas (2/2)

Problema 4.3 (continuación):

- Indica que cantidad de memoria ocupa esta representación, en estos dos casos:
 - Como función de n y m , en bytes.
 - Suponiendo $m = n = 128$, en KB.
- Para m y n grandes (es decir, cuando $1/n$ y $1/m$ son casi nulos), describe que relación hay entre el tamaño en memoria de la malla indexada del problema anterior y el tamaño de la malla almacenada como tiras de triángulos.
- Si suponemos que la transformación de cada vértice se hace en un tiempo constante igual a la unidad, describe que relación hay entre los tiempos de procesamiento de vértices para esta malla cuando se representa como una malla indexada y como tiras de triángulos.

Problema: número de vértices, aristas y caras

Problema 4.4:

Supongamos una malla cerrada, simplemente conexa (topológicamente equivalente a una esfera), cuyas caras son triángulos y cuyas aristas son todas adyacentes a exactamente dos caras (la malla es un *poliedro simplemente conexo de caras triangulares*). Considera el número de vértices n_V , el número de aristas n_A y el número de caras n_C en este tipo de mallas.

Demuestra que cualquiera de esos números determina a los otros dos, en concreto, demuestra que se cumplen estas dos igualdades:

$$n_A = 3(n_V - 2)$$

$$n_C = 2(n_V - 2)$$

(nótese que, al igual que en el problema anterior, sigue siendo cierto que el número de caras es aproximadamente el doble que el de vértices).

Problema: creación de la tabla de aristas

Problema 4.5:

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector **ari**, que en cada entrada tendrá una tupla de tipo **Vector2i** (contiene dos **int**) con los índices en la tabla de vértices de los dos vértices en los extremos de la arista. El orden en el que aparecen los vértices en una arista es indiferente, pero cada arista debe aparecer una sola vez.

Escribe el código de una función GDScript para crear y calcular la tabla de aristas a partir de la tabla de triángulos. Intenta encontrar una solución con la mínima complejidad en tiempo y memoria posible. Suponer que el número de vértices adyacentes a uno cualquiera de ellos es como mucho un valor constante $k > 0$, valor que no depende del número total de vértices, que llamamos n .

(continua en la transparencia siguiente)

Problema: creación de la tabla de aristas

Problema 4.5 (continuación):

Considerar dos casos:

- (a) Los triángulos se dan con orientación *no coherente*: esto quiere decir que si un triángulo está formado por los vértices i, j, k , estos tres índices pueden aparecer en cualquier orden en la correspondiente entrada de la tabla de triángulos. Además, no sabemos si la malla es cerrada o no.
- (b) Los triángulos se dan con orientación *coherente*: esto quiere decir que si dos triángulos comparten una arista entre los vértices i y j , entonces en uno de los triángulos la arista aparece como (i, j) y en el otro aparece como (j, i) . Además, asumimos que la malla es *cerrada*, es decir, que cada arista es compartida por exactamente dos triángulos.

Problema: cálculo del área de una malla indexada

Problema 4.6:

Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función GDScript que acepta ambas tablas (arrays de `Vector3` y de `Vector3i`) y devuelve el área.

Fin de transparencias.

Informática Gráfica.

Sesión 5: Modelos Jerárquicos.

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

Índice

Modelos Jerárquicos	3
Grafos de escena en Godot.	10
Ejemplo de un árbol 2D	36
Problemas	57

Sección 1. Modelos Jerárquicos

1. Grafos de escena.

Subsección 1.1. Grafos de escena.

Modelos jerárquicos.

En Informática Gráfica, un **Modelo Jerárquico** es una estructura de datos en forma de grafo que representa las relaciones espaciales entre las **componentes** de una aplicación interactiva.

- Es una herramienta fundamental que permite diseñar y gestionar modelos complejos mediante el uso de **componentes complejas constituidas de otras componentes más simples**
- Una **componente** es un objeto geométrico 2D o 3D sencillo, como una malla, o un grupo de varias componentes.
 - ▶ Permite la **reutilización** de componentes en distintas partes de un proyecto o en distintos proyectos.
- Permite a distintos desarrolladores (o al mismo desarrollador en distintos instantes) trabajar en diferentes componentes de un proyecto sin interferir entre sí, facilitando la colaboración en proyectos grandes.

Marcos afines y transformaciones asociados a los nodos.

La escena asociada a un grafo está compuesta de **objetos instanciados**.

- Un **objeto instanciado** es una réplica del objeto asociado a un nodo del grafo, pero con las coordenadas de sus vértices transformadas por una transformación afín propia de esa réplica.
- Un objeto puede aparecer instanciado varias veces en una escena, cada instancia puede tener asociada una transformación afín distinta.
- Las coordenadas de los vértices de un nodo son relativas a un marco afín propio de cada nodo, llamado **marco local** del nodo. A las coordenadas se les llama **coordenadas locales**.
- El marco local del nodo raíz se llama **marco de escena** y sus coordenadas se llaman **coordenadas de escena**.
- La transformación asociada a cada nodo determina como se convierten las coordenadas locales del nodo (relativas al marco del nodo) en coordenadas de escena (relativas al marco de escena).

Grafos de escena.

Un **Grafo de Escena** es un **Grafo Dirigido Acíclico**, cada uno de cuyos nodos contiene un modelo de un objeto. El grafo en sí es también un modelo (jerárquico) que representa un objeto compuesto (llamado **escena**) formado por **réplicas de los objetos cuyos modelos están en los nodos**.

- Un nodo N del grafo puede ser:

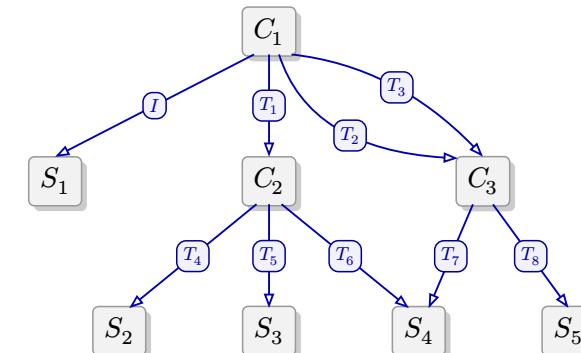
Terminal: si objeto asociado a N no está compuesto de otros objetos, típicamente mallas con vértices en determinadas posiciones en el espacio.

No terminal: si el objeto está compuesto de otros objetos en otros nodos del grafo, nodos llamados **nodos hijos** de N .

- En el grafo hay un **arco dirigido** desde cada nodo padre a cada uno de sus nodos hijos. Cada arco tiene asociada una **transformación geométrica** (afín).
- Todo nodo del grafo tiene al menos un parent (o más), excepto exactamente un nodo sin padres, que es el **nodo raíz**.

Ejemplo de grafo de escena

En un grafo de escena, cada arco dirigido tiene asociada una transformación afín:



- cada objeto compuesto de otros es un **nodo no terminal** (C_1, C_2, C_3)
- cada objeto simple (no compuesto) es un **nodo terminal** (S_1, S_2, S_3, S_4, S_5)
- cada arco se etiqueta con una transformación geométrica (I, T_1, T_2, \dots, T_8)

Instancias de objetos en el grafo de ejemplo

En la escena representada por un grafo **hay una instancia de cada nodo por cada camino posible desde la raíz al nodo**. Esa instancia tiene asociada la transformación que resulta de componer las transformaciones de los arcos del camino (de izquierda a derecha según se va desde la raíz al nodo).

A modo de ejemplo, en el grafo anterior hay las siguientes instancias de objetos terminales (no compuestos):

- Una instancia de S_1 , con la transformación I (la transformación identidad).
- Una instancia de S_2 , con la transformación $T_1 \circ T_4$.
- Una instancia de S_3 , con la transformación $T_1 \circ T_5$.
- Tres instancias de S_4 , con las transformaciones $T_1 \circ T_6$, $T_2 \circ T_7$ y $T_3 \circ T_7$.
- Dos instancias de S_5 , con las transformaciones $T_2 \circ T_8$ y $T_3 \circ T_8$.

En total hay 8 instancias de objetos terminales. Además, si se permite que los nodos no terminales tengan mallas (además de hijos), habría que añadir 3 instancias de los objetos C_2 y C_3 .

Sección 2.

Grafos de escena en Godot.

1. Escenas y nodos.
2. Transformaciones de los nodos.
3. La transformación de los nodos 2D.
4. La transformación de los nodos 3D.
5. Creación y actualización de árboles en tiempo de ejecución.

Proyectos, escenas y nodos en Godot

El desarrollo de aplicaciones gráficas en Godot se basa en los conceptos de **proyecto, escena y nodo**.

- Un **proyecto** es un conjunto de archivos que se usan para construir una aplicación. Se guardan en una *carpeta del proyecto* donde, entre otros, habrá un archivo **.godot** con datos del mismo.
- Una **escena** es un árbol de nodos, que se llama **árbol de escena**, con al menos un nodo. Una escena siempre tiene asociado un **nodo raíz** de la misma.
- Un proyecto incluye **una o varias escenas**. Una de las escenas será la **escena principal** del proyecto. En la carpeta del proyecto habrá un archivo **.tscn** por cada escena de dicho proyecto.
- Un **nodo** es un objeto (instancia de alguna clase Godot) que representa un elemento de la aplicación. Un nodo se incluye en una única escena, y aparece una sola vez en el árbol de dicha escena, por tanto un nodo tiene un único parente (excepto el nodo raíz de una escena).
- El nodo raíz de la escena principal se considera el **nodo raíz del proyecto**.

Clasificación de los nodos

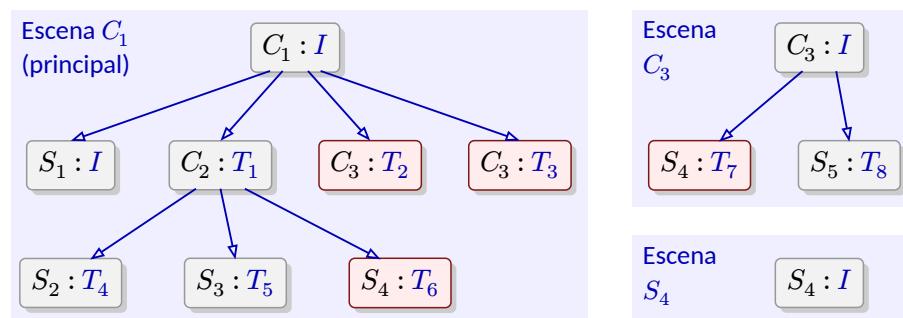
Los nodos puede clasificarse en estas dos categorías:

- **Nodos regulares:** son objetos instancias de la clase **Node** o sus derivadas, principalmente:
 - ▶ **CanvasItem:** para objetos 2D, puede ser de la clase **Node2D** o derivadas (objetos visibles) o **Control** (objetos de interfaz gráfica).
 - ▶ **Node3D:** objetos que contienen, entre otras cosas, mallas 3D, cámaras, fuentes de luz etc...
- **Nodos instancia de escena:** guardan **una referencia a una escena del proyecto** (que no puede ser la escena principal del proyecto). Este nodo es único en el árbol, pero varios nodos de este tipo pueden tener referencias a la misma escena.

Así que la única forma de que un nodo pueda instanciarse más de una vez en un grafo es hacer que ese nodo sea una escena de Godot.

Esquema de varias escenas de un proyecto Godot

Vemos aquí las escenas equivalentes (en Godot) al grafo de ejemplo que ya vimos, pero ahora cada nodo tiene asociada una transformación (después de :) y los nodos instancia de escena tienen fondo rosa.



Hay tres escenas, cuyas raíces son: C_1 , C_3 y S_4 . Godot ignora las transformaciones de los nodos raíz (ya que se redefinen en las instancias), excepto la transformación del nodo raíz de la escena principal.

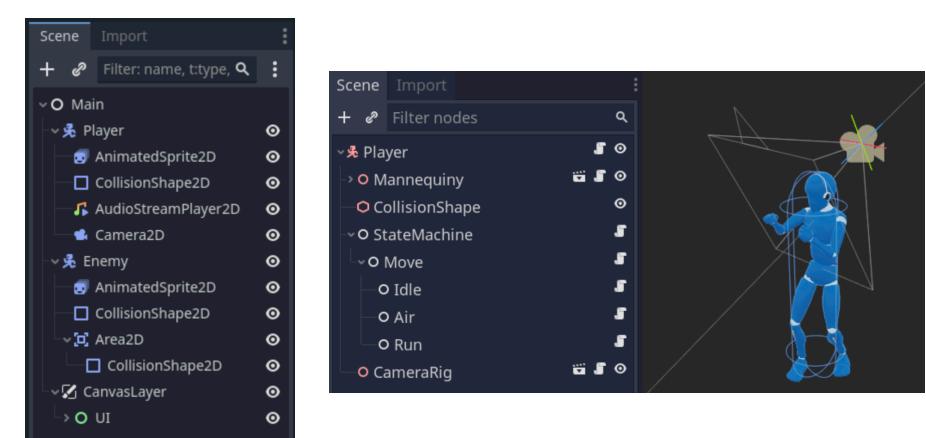
Instancias de nodos y escenas

En un árbol de escena:

- Cada nodo en un árbol de una escena tiene exactamente un parent (excepto el nodo raíz de una escena).
- Para cada escena (distinta de la escena principal del proyecto), puede haber varios nodos de tipo **instancia de escena** que refieren a esa escena, por tanto una escena puede estar:
 - ▶ instanciada en varios árboles de escena diferentes, o además
 - ▶ instanciada varias veces en un mismo árbol de escena.
- En el panel con el **árbol de escena** (arriba a la izquierda del editor), los nodos sub-escena aparecen como un nodo normal, pero con un icono de una placa. No se pueden expandir.
- En el panel del **sistema de archivos** (abajo a la izquierda del editor), podemos ver un archivo de extensión **.tscn** que guarda información de una escena. Si lo seleccionamos, en el panel **árbol de escena** pasaremos a ver el árbol de escena de esa escena.

Ejemplos de árboles de escena de Godot.

Aquí vemos dos ejemplos de dos árboles de escena de Godot.

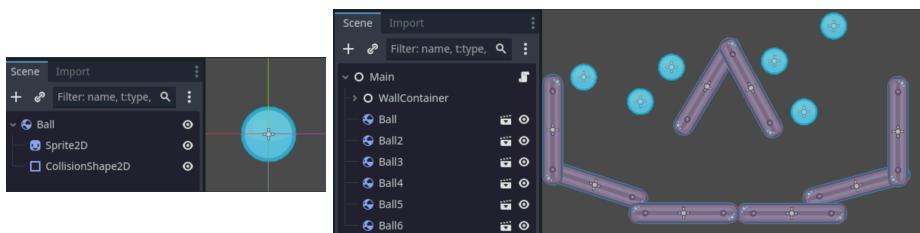


Imágenes obtenidas obtenidas de la documentación oficial de Godot:
docs.godotengine.org/es/4.5/getting_started/step_by_step/nodes_and_scenes.html

Escena instanciada en otra

A la izquierda vemos el disco azul que está modelado en la escena **Ball**, es un árbol formado por tres nodos.

A la derecha, los nodos **Ball**, **Ball2**, **Ball3**, etc.... son instancias de la escena **Ball**, cada una con su propia transformación (en concreto, están trasladadas a distintas posiciones del plano). En el panel aparecen con un icono de placa.



Imágenes obtenidas de la documentación oficial de Godot:

docs.godotengine.org/es/4.5/getting_started/step_by_step/instancing.html

Subsección 2.2.

Transformaciones de los nodos.

Reutilización de mallas

En las aplicaciones gráficas, los objetos tipo **Mesh** (mallas 2D o 3D) contienen los vértices y las coordenadas de textura, las normales, etc... y por tanto pueden ocupar mucha memoria. Por tanto:

- Se hace necesario evitar la duplicación de mallas en memoria, por ejemplo con dos nodos que incluyan la misma malla.
- Se pueden usar nodos de tipo **MeshInstance2D** o **MeshInstance3D** que contienen una referencia a una malla (propiedad **mesh**), pero no la malla en sí.
- En una aplicación puede haber distintos nodos referenciando la misma malla.
- Puesto que la clase **Mesh** es derivada de **RefCounted** (objetos con cuenta de referencias), en Godot se gestiona automáticamente la memoria de las mallas, siendo esta liberada cuando no hay más referencias a la misma.

El marco y la transformación asociados a un nodo.

Sea N un nodo (**Node2D** y **Node3D**) situado en un árbol de escena de Godot:

- El nodo N siempre asociado un marco afín (2D o 3D) que se llama **marco del nodo N** , lo llamamos \mathcal{N}
- El nodo N también tendrá asociado un **marco padre**, lo llamamos \mathcal{P} :
 - ▶ Si N no es el nodo raíz de una escena, \mathcal{P} es el marco del único nodo padre de N en el árbol de esa escena.
 - ▶ Si N es nodo raíz de una escena, entonces \mathcal{P} es el llamado **marco global de la escena**. Si esa escena es la escena principal de Godot, entonces a \mathcal{P} también se le llama **marco del mundo**.
- La transformación que convierte el marco \mathcal{P} en el marco \mathcal{N} se representa mediante una matriz M_N llamada **transformación (o matriz) del nodo**, tiene en sus **columnas** la base y el origen del marco \mathcal{N} , **expresadas en coordenadas relativas al marco \mathcal{P}** . Esto implica que

$$\mathcal{P}M_N = \mathcal{N}$$

El espacio de coordenadas local y el espacio padre

En relación a las coordenadas de los vértices en las mallas u objetos guardados en un nodo N :

- Se considera que están siempre expresadas en el marco \mathcal{N} del nodo, y se llaman **coordenadas locales de N** , también se dice que están expresadas en el **espacio de coordenadas local del nodo**.
- Al aplicarseles la transformación M_N , se convierten en coordenadas relativas al marco \mathcal{P} , se dice que esas coordenadas están expresadas en el **espacio de coordenadas padre de N** .
- En última instancia, todas las coordenadas de los vértices en una aplicación Godot acaban convertidas en la GPU en coordenadas relativas al marco global de la escena principal o marco del mundo, esas son **coordenadas de mundo**.
- Las coordenadas del mundo se usan por Godot para proyectar los vértices en pantalla y producir la imagen por rasterización en la GPU.

La propiedad `transform` de un nodo en Godot

Para cualquier nodo N , el objeto asociado tiene una propiedad de tipo `Transform2D` o `Transform3D`, llamada `transform`, que guarda la matriz asociada M_N .

La matriz de un nodo N se puede modificar asignando o actualizando su `transform`, se puede hacer de estas formas:

- En el editor, podemos cambiar el valor inicial de `transform` (le asignamos una posición, escalado o rotaciones).
- En tiempo de ejecución del videojuego, podemos modificar $N.transform$ directamente en el código GDScript. Se puede hacer de dos formas:
 - ▶ Mediante asignaciones a $N.transform$ o sus componentes.
 - ▶ Mediante asignaciones a propiedades relacionadas con `transform` (`position`, `rotation`, etc..., ver siguiente transparencia).
 - ▶ Usando métodos las clases `Node2D` o `Node3D` los cuales, aplicados a un nodo N , modifican $N.transform$.

Subsección 2.3.

La transformación de los nodos 2D.

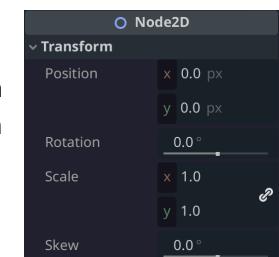
Atributos de transformación de los nodos 2D

Para cada nodo, en el panel de la derecha del editor podemos ver las propiedades que definen su transformación inicialmente. Godot mantiene los atributos siempre coherentes con la matriz `transform` del nodo. Los atributos son:

- position:** vector 2D (traslación)
- rotation:** ángulo de rotación en radianes
- scale:** vector 2D (factores de escala)
- skew:** real, ángulo de skew en grados (tipo de cizalla)

La matriz `transform` se corresponde con la composición de estas transformaciones (en este orden de derecha a izquierda):

1. Escalado por los factores `scale`
2. Skew por el ángulo `skew`
3. Rotación de `rotation` radianes
4. Traslación por vector `position`



Actualización de la transformación 2D en un script

Las propiedades de transformación se pueden cambiar en tiempo de ejecución. El siguiente código siempre produce el mismo resultado, independientemente del valor de `asignar_transform` y de las variables declaradas en el código

```
var factores_escala      := Vector2( 2.0, 1.0 )
var angulo_rot_radianes := 1.0
var vector_traslacion   := Vector2( 1.0, 0.0 )

if asignar_transform :
    var esc := Transform2D().scaled( factores_escala )
    var ske := Transform2D() ## identidad (skew nulo)
    var rot := Transform2D().rotated( angulo_rot_radianes )
    var tra := Transform2D().translated( vector_traslacion )
    transform = tra * rot * ske * esc ## el orden es esencial
else:
    scale     = factores_escala      ## (1)
    skew     = 0.0                  ## (2)
    rotation = angulo_rot_radianes ## (3)
    position = vector_traslacion   ## (4)
```

Modificación de la transformación de un nodo 2D.

Estos métodos, aplicados a un nodo N , modifican las propiedades de transformación, y después se recalcula la matriz `transform` del nodo.

- $N.\text{rotate}(\theta)$: añade θ radianes a la propiedad `rotation`, es decir, es equivalente a:

```
rotation += theta
```

- $N.\text{apply_scale}(s)$: multiplica los factores en la propiedad `scale` por los dos factores en s .

```
scale = Vector2( scale.x * s.x, scale.y * s.y )
```

- $N.\text{translate}(t)$: suma el vector t a la propiedad `position`.

```
position += vt
```

Atributos de transformación de los nodos 3D

Godot también tiene propiedades que definen la transformación de un nodo 3D. En el caso 3D no hay `skew` y además la rotación es la composición de 3 rotaciones entorno a los ejes en el orden Y, X, Z (aunque se puede cambiar el orden, o usar un cuaternion, o dar directamente los ejes transformados).

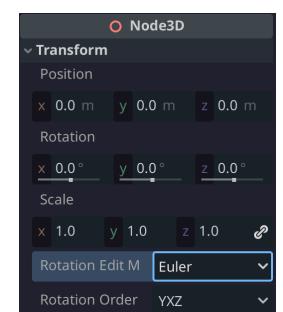
position: vector 3D (traslación)

rotation: vector con tres ángulos de rotación (Y, X, Z)

scale: vector 3D (3 factores de escala)

La matriz `transform` se corresponde con la composición de estas transformaciones (en este orden de derecha a izquierda):

1. Escalado por los factores `scale`
2. Rotaciones elementales con los ángulos contenidos en `rotation` (en radianes).
3. Traslación por vector `position`



Actualización de la transformación: rotaciones por la izquierda.

Al igual que en 2D, en 3D existen diversos métodos que permiten modificar las propiedades **rotation**, **scale** de un nodo 3D. En estos casos se componen matrices de rotación por la izquierda de la matriz actual.

- **N.rotate(v, θ)**: compone la rotación codificada en **rotation** con una rotación entorno al eje **v** (del padre). Si **position** es el vector nulo, entonces es equivalente a:

```
var rot := Transform3D().rotated( v, theta )
N.transform = rot * N.transform
```

- **N.rotate_x(θ)**: idem, donde **v** es el eje X.
- **N.rotate_y(θ)**: idem, eje Y.
- **N.rotate_z(θ)**: idem, eje Z.

Al usarse composición por la izquierda, el vector 3D **v** con el eje de rotación se interpreta como expresado en el **marco de coordenadas del padre**.

Actualización de la transformación: composición por la derecha.

Existen otros métodos que permite componer matrices por la derecha, es decir, la matriz que se compone actúa sobre las coordenadas **antes** que la transformación previa, es decir, se aplica a las coordenadas locales en primer lugar. Equivalente a:

```
var M : Transform3D = ... ## la matriz que se quiere componer
N.transform = N.transform * M
```

- **N.rotate_object_local(v, θ)**: $M :=$ rotación de θ entorno al eje **v**
- **N.translate_object_local(t)**: $M :=$ traslación por el vector **t**
- **N.translate(t)**: equivalente a la anterior.
- **N.scale_object_local(s)**: $M :=$ matriz de escalado con factores **s**.

Al usarse composición por la derecha, eso implica que las tuplas **v,t** y **s** (de tipo **Vector3**) que se indican aquí se interpretan como expresadas en el **marco de coordenadas local del nodo N**.

Creación, inserción y consulta de nodos en un árbol.

Los nodos se pueden crear en tiempo de ejecución con el método de clase **new()**. Si el constructor tuviese parámetros, habría que proporcionarlos como argumentos de **new()**. Por ejemplo, para crear un nodo de tipo **Node2D**:

```
var nodo := Node2D.new() ## al crearlo está "huérfano"
```

Para añadir un nodo (por ejemplo **hijo**) al árbol de escena, habría que usar el método **add_child()** del nodo padre (nodo **padre**). Lo añade como último nodo hijo. Por ejemplo:

```
padre.add_child( hijo )
```

Los hijos directos de un nodo padre **p** se pueden consultar con estos métodos:

- **p.get_child_count()**: devuelve el número de hijos del nodo.
- **p.get_child(i)**: devuelve el hijo número **i** (entero, empezando en 0).
- **p.get_children()**: devuelve un **Array** con todos los hijos del nodo.

Nombres de los nodos y búsqueda

Todo nodo tiene una propiedad **name** (cadena de caracteres, **String**) que se puede asignar y consultar en el editor o en tiempo de ejecución con scripts. Sirve para **identificar un nodo en el árbol**, y debe ser único entre los nodos hijos de un mismo padre. Los métodos para buscar nodos en un árbol son:

- ***p.get_node(s)***: devuelve un nodo descendiente (directo o indirecto) del nodo *p*, siguiendo la ruta (*path*) dada en la cadena *s*. La ruta tiene nombres de nodos separados por **/**, empezando por un hijo de *p*. Si no se encuentra da error. Aquí se obtiene el nodo de nombre «*bisnieto*»:

```
var nodo := n.get_node("hijo/nieto/bisnieto")
var nodo := $hijo/nieto/bisnieto ## forma equivalente.
```

- ***p.get_node_or_null(s)***: igual que el anterior, pero si no encuentra nada devuelve **null** en vez de dar error.
- ***p.find_node(s)***: igual que el anterior, pero permite caracteres comodín ***** y **?** en *s* y devuelve el primer nodo que encaja con la cadena, en un recorrido en profundidad del árbol. Si no encuentra nada, devuelve **null**.

Objetos de tipo **Mesh** compartidos

Como se ha indicado, en un árbol es conveniente **no replicar objetos **Mesh** completamente**. A modo de ejemplo, este código crea un **ArrayMesh** y dos **MeshInstance3D** que lo comparten. Cada **MeshInstance3D** está instanciado en una posición distinta:

```
var tablas := []
tablas.resize( Mesh.ARRAY_MAX )
GenerarTablas( tablas ) ## función que genera vértices, normales, etc...
var am := ArrayMesh.new()
am.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

var instancia1 := MeshInstance3D.new()
instancia1.mesh      = am
instancia1.position = Vector3( -3.0, 0.0, 0.0 )

var instancia2 := MeshInstance3D.new()
instancia2.mesh      = am
instancia2.position = Vector3( +3.0, 0.0, 0.0 )
```

Desconexión y destrucción de nodos

Se puede **desconectar un nodo hijo *h* de su nodo padre *p*** con el método ***p.remove_child(h)***. Esto lo elimina del árbol, pero no se destruye el nodo (aunque queda **huérfano** con seguridad, ya que todo nodo tiene un único parent, y no será visible en la escena).

Para **destruir** un nodo huérfano (es decir, liberar la memoria que ocupa ese nodo y todos sus hijos), se usa el método ***n.queue_free()***. Esto registra la solicitud de eliminar el nodo, lo cual ocurrirá al final del siguiente frame.

A modo de ejemplo, para desconectar y destruir un nodo hijo (con nombre «*borrar*») de un nodo parent *p*, haríamos:

```
var h := p.get_node("borrar")
p.remove_child( h ) ## queda huérfano
h.queue_free()      ## se puede usar hasta el final del siguiente frame.
```

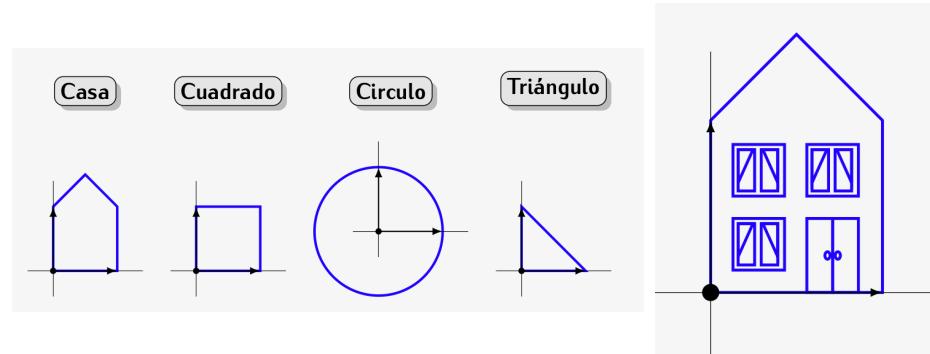
Sección 3.

Ejemplo de un árbol 2D

1. Diseño del grafo.
2. Implementación en Godot.
3. Implementación de Grafos parametrizados (y animados).

Objetos simples

Supongamos que queremos construir una figura como la de la derecha, usando varios objeto **ArrayMesh** simples (un cuadrado, un triángulo y un círculo), tal y como aparecen a la izquierda.



Subsección 3.1. Diseño del grafo.

3. Ejemplo de un árbol 2D.
3.1. Diseño del grafo..

Notación abreviada para grafos de escena

Para este diseño, usaremos una notación más expresiva para especificar el grafo de escena:

- Usaremos un grafo dirigido acíclico, en vez de un simple árbol.
- Los nodos son listas de:
 - ▶ Elementos simples (como los de la transparencia anterior), en negrita.
 - ▶ Instancias de otros nodos (subárboles), con una flecha a otro nodo.
 - ▶ Transformaciones: afectando a todas las entradas de la lista que le siguen en el nodo (y se aplican de abajo hacia arriba).
- La implementación de estos grafos en Godot requiere convertirlos en árboles, bien duplicando nodos, bien usando instancias de escenas.
- En nuestro caso, usaremos nodos duplicados, aunque compartirán objetos **ArrayMesh**.

Cada objeto aparece junto a su marco de referencia local, de forma que podamos entender mejor las transformaciones.

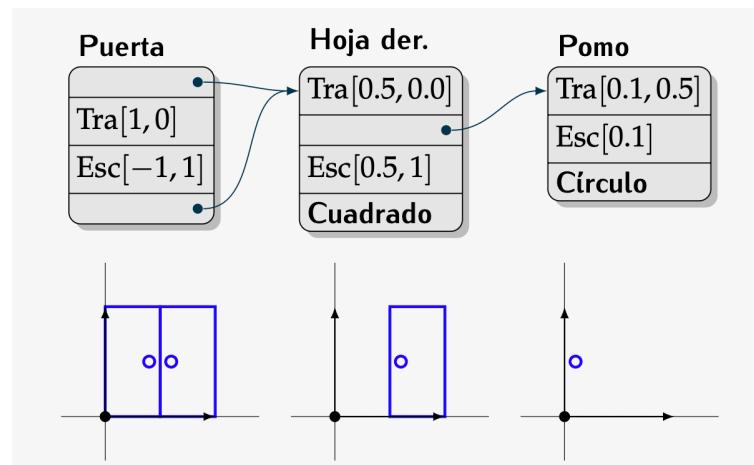
Sesión 5: Modelos Jerárquicos

Created 2025-12-10

Page 38 / 63.

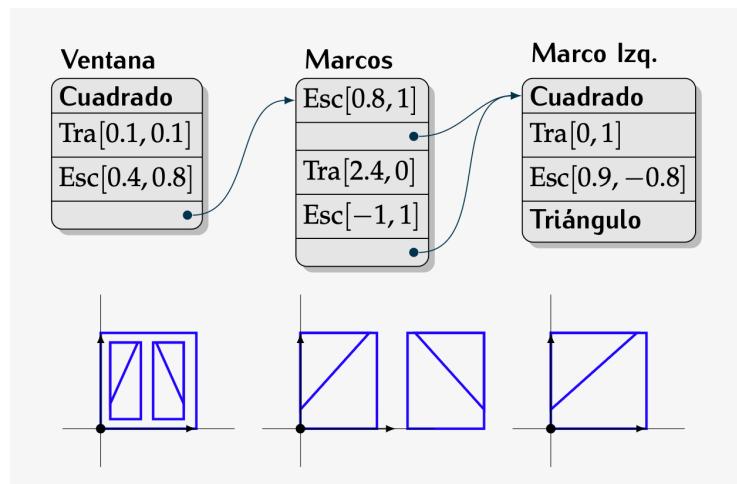
Objetos puerta, hoja derecha y pomo.

El nodo **Puerta**, contiene dos instancias de un nodo llamado **HojaDerecha**, que a su vez tiene un objeto **Pomo**



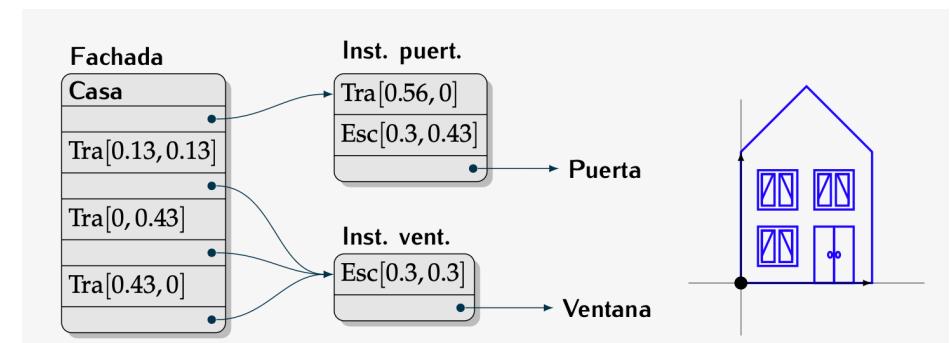
Objetos ventana, marcos y marco izquierdo

Objetos **Ventana**, compuesto de un **Cuadrado** y un objeto **Marcos**, a su vez compuesto de un **MarcoIzquierdo** (que es un **Cuadrado** y un **Triángulo**):



Objeto fachada, instancias de puertas y de ventana

Objeto **Fachada**, que es la raíz del árbol de escena. Contiene los objetos: **Casa**, **InstanciaPuerta** e **InstanciaVentana** (que a su vez incluyen **Puerta** y **Ventana**, respectivamente).



Estrategias de implementación.

El diseño del grafo de escena puede ser implementado en Godot de diversas formas:

- Creando los nodos en el editor, sin scripts, excepto para crear los objetos **ArrayMesh**
- Creación en tiempo de ejecución con uno o varios scripts.
- Combinación de ambos: algunos nodos creados en el editor, y otros en tiempo de ejecución.

En nuestro caso, usaremos un **único script asociado al nodo raíz**. Dicho nodo raíz se puede crear en el editor, y después, en tiempo de ejecución, en la función **_ready()**, se crean todos los nodos descendientes.

Subsección 3.2.

Implementación en Godot.

Funciones auxiliares.

Esta función crea un objeto **ArrayMesh** a partir de un array de posiciones de vértices (con tipo de primitiva polilínea abierta, **PRIMITIVE_LINE_STRIP**):

```
func CrearArrayMesh( v : PackedVector2Array ) -> ArrayMesh :
    var tablas : Array = [] ; tablas.resize( Mesh.ARRAY_MAX )
    tablas[ Mesh.ARRAY_VERTEX ] = v
    var am : ArrayMesh = ArrayMesh.new()
    am.add_surface_from_arrays( Mesh.PRIMITIVE_LINE_STRIP, tablas )
    return am
```

Esta función crea un nodo **MeshInstance2D** a partir de un **ArrayMesh** dado:

```
func CrearMeshInstance2D( am : ArrayMesh, tr : Transform2D ) ->
    MeshInstance2D:
    var mi := MeshInstance2D.new()
    mi.transform = tr
    mi.modulate = Color( 0.0, 0.0, 0.7 ) ## azul
    mi.mesh = am
    return mi
```

Objetos simples: circunferencia

La variable correspondiente a la circunferencia se puede crear así

```
var circunferencia : ArrayMesh = CrearCircunferencia( 64 )
```

Se usa una función auxiliar que genera los vértices de una circunferencia de radio unidad, con el número de segmentos indicado:

```
func CrearCircunferencia( n : int ) -> ArrayMesh :
    var v := PackedVector2Array()
    for i in range( n+1 ):
        var a : float = (float(i) * 2.0 * PI )/ float(n)
        v.append( Vector2( cos(a), sin(a) ) )
    return CrearArrayMesh( v )
```

Otra función auxiliar útil compone una transformación por la izquierda:

```
func TransformaNode2D( n : Node2D, tr : Transform2D ) -> Node2D :
    n.transform = tr * n.transform
    return n
```

Objeto simples: cuadrado, triángulo, casa

Los objetos simples se puede crear como variables únicas, de tipo **ArrayMesh**, de forma que se compartan entre todas las instancias que los usen.

```
var cuadrado := CrearArrayMesh( PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
    Vector2( 1.0, 1.0 ), Vector2( 0.0, 1.0 ),
    Vector2( 0.0, 0.0 )
]))
```



```
var triangulo := CrearArrayMesh( PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ), Vector2( 0.0, 1.0 ),
    Vector2( 0.0, 0.0 )
]))
```



```
var casa := CrearArrayMesh( PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
    Vector2( 1.0, 1.0 ), Vector2( 0.5, 1.4 ), Vector2( 0.0, 1.0 ),
    Vector2( 0.0, 0.0 )
]))
```

Funciones para el pomo, hoja derecha y puerta

```
func Pomo() -> Node2D :
    var tra = Transform2D().translated( Vector2( 0.1, 0.5 ) )
    var esc = Transform2D().scaled( Vector2( 0.06, 0.06 ) )
    return CrearMeshInstance2D( circunferencia, tra*esc )
```



```
func HojaDer() -> Node2D :
    var tra = Transform2D().translated( Vector2( 0.5, 0.0 ) )
    var esc = Transform2D().scaled( Vector2( 0.5, 1.0 ) )
    var n = Node2D.new()
    n.add_child( TransformaNode2D( Pomo(), tra ) )
    n.add_child( CrearMeshInstance2D( cuadrado, tra*esc ) )
    return n
```



```
func Puerta() -> Node2D :
    var tra = Transform2D().translated( Vector2( 1.0, 0.0 ) )
    var esc = Transform2D().scaled( Vector2( -1.0, 1.0 ) )
    var n = Node2D.new()
    n.add_child( HojaDer() )
    n.add_child( TransformaNode2D( HojaDer(), tra*esc ) )
    return n
```

Marco izquierdo y marcos

```
func MarcoIzq() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.0, 1.0 ) )  
    var esc = Transform2D().scaled( Vector2( 0.9, -0.8 ) )  
    var n = Node2D.new()  
    n.add_child( CrearMeshInstance2D( cuadrado, tr_identidad ) )  
    n.add_child( CrearMeshInstance2D( triangulo, tra*esc ) )  
    return n  
  
func Marcos() -> Node2D :  
    var esc1 = Transform2D().scaled( Vector2( 0.8, 1.0 ) )  
    var tra = Transform2D().translated( Vector2( 2.4, 0.0 ) )  
    var esc2 = Transform2D().scaled( Vector2( -1.0, 1.0 ) )  
    var n = Node2D.new()  
    n.add_child( TransformaNode2D( MarcoIzq(), esc1 ) )  
    n.add_child( TransformaNode2D( MarcoIzq(), esc1*tra*esc2 ) )  
    return n
```

Fachada

```
func Fachada() -> Node2D :  
  
    var tra1 = Transform2D().translated( Vector2( 0.13, 0.13 ) )  
    var tra2 = Transform2D().translated( Vector2( 0.00, 0.43 ) )  
    var tra3 = Transform2D().translated( Vector2( 0.43, 0.00 ) )  
  
    var n = Node2D.new()  
    n.add_child( CrearMeshInstance2D( casa, tr_identidad ) )  
    n.add_child( TransformaNode2D( InstPuerta(), tr_identidad ) )  
    n.add_child( TransformaNode2D( InstVentana(), tra1 ) )  
    n.add_child( TransformaNode2D( InstVentana(), tra1*tra2 ) )  
    n.add_child( TransformaNode2D( InstVentana(), tra1*tra2*tra3 ) )  
  
    return n
```

Ventana, instancia puerta e instancia ventana

```
func Ventana() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.1, 0.1 ) )  
    var esc = Transform2D().scaled( Vector2( 0.4, 0.8 ) )  
    var n = Node2D.new()  
    n.add_child( CrearMeshInstance2D( cuadrado, tr_identidad ) )  
    n.add_child( TransformaNode2D( Marcos(), tra*esc ) )  
    return n  
  
func InstPuerta() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.56, 0.0 ) )  
    var esc = Transform2D().scaled( Vector2( 0.3, 0.43 ) )  
    var n = Node2D.new()  
    n.add_child( TransformaNode2D( Puerta(), tra*esc ) )  
    return n  
  
func InstVentana() -> Node2D :  
    var esc = Transform2D().scaled( Vector2( 0.3, 0.3 ) )  
    var n = Node2D.new()  
    n.add_child( TransformaNode2D( Ventana(), esc ) )  
    return n
```

Subsección 3.3.

Implementación de Grafos parametrizados (y animados).

Diseño de grafos parametrizados

A veces los grafos de escena pueden depender de uno o varios **parámetros o grados de libertad**.

Son uno o varios valores reales (o vectores), de forma que cada uno de ellos **determina una o varias transformaciones de un modelo jerárquico**. Esto permite:

Animar de forma sencilla un modelo jerárquico: haciendo depender los valores de los parámetros del tiempo real transcurrido durante la ejecución.

Generar múltiples variantes de un diseño: ya que podemos construir nodos o sub-árboles similares, que solo difieren en los valores de los parámetros.

Modificar interactivamente un modelo en tiempo de ejecución: ya que se puede permitir al usuario usar controles o eventos de entrada para modificar los parámetros.

A modo de ejemplo, en un modelo de un coche, puede existir un parámetro que sea un ángulo de giro de las cuatro ruedas. Es un real en radianes que determina la transformación de rotación.

Variación lineal u oscilante

Para animar un modelo jerárquico, los valores de los parámetros se pueden variar en cada frame y recalculando las transformaciones de los nodos. Así se consigue animación o interacción. Hay muchas opciones, dos bastante sencillas y útiles son:

- Hacer que un parámetro **varíe linealmente con el tiempo**, por ejemplo, un ángulo de rotación r que aumenta a ritmo constante por cada segundo:

$$r = r_0 + 2\pi \cdot w \cdot t,$$

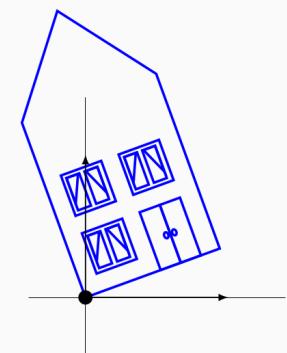
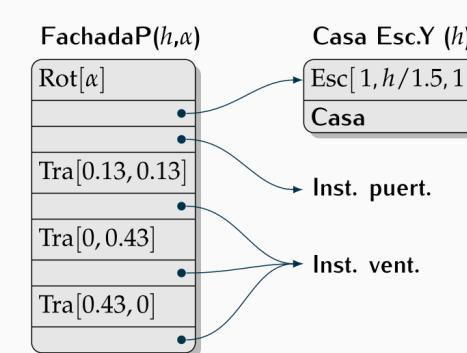
donde t es el tiempo transcurrido de animación en segundos, w es el número de vueltas por segundo, y r_0 el valor de r al inicio (cuando $t = 0$).

- Para escalados o desplazamientos queremos evitar que crezcan indefinidamente (no suele tener sentido), así que hacemos que v **oscile entre otros dos valores** a y b , y lo haga w veces por segundo:

$$v = a + (b - a) \cdot \frac{1 + \sin(2\pi \cdot w \cdot t)}{2}$$

Ejemplo de diseño de un grafo parametrizado

En el ejemplo anterior, podemos hacer que el grafo dependa de dos parámetros, α (rotación de la figura completa, en radianes) y h (altura de la fachada, real positivo):



Implementación de grafos parametrizados en Godot

En este ejemplo modificamos las transformaciones de dos nodos **n1** y **n2**

```

var a      := .... ## valor mínimo (e inicial) del parámetro 2 (oscilante)
var b      := .... ## valor máximo del parámetro 2 (oscilante)
var w1     := .... ## ciclos o vueltas por segundo (parámetro 1)
var w2     := .... ## ciclos o vueltas por segundo (parámetro 2)
var ang2 := 0.0  ## valor acumulado de 2*PI*w2*t (argumento de 'sin')

func _process( delta : float ) :
    if animacion_activada :

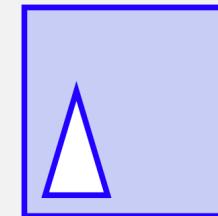
        ## actualizar transformación del nodo 'n1':
        n1.rotate( 2*PI*w1*delta ) ## simplemente acumulamos rotación

        ## actualizar transformación del nodo 'n2':
        ## (se calcula un factor de escala 'fe' oscilante)
        ang2 += 2.0 * PI * w2 * delta ## acumulamos en 'ang2'
        var fe : float = a + (b-a)*(1.0 + sin( ang2 ))/2.0
        n2.transform = Transform2D().scaled( Vector2( fe, 1.0 ) )
    
```

Escena simple

Problema 5.1:

Implementa un proyecto cuya escena principal tenga un de tipo **Node2D** con varios nodos hijos, que formen la figura con un cuadrado de lado 2, centrado en el origen, y con un triángulo inscrito. El cuadrado debe estar relleno de azul claro, el triángulo de blanco, y las aristas deben verse de color azul oscuro.



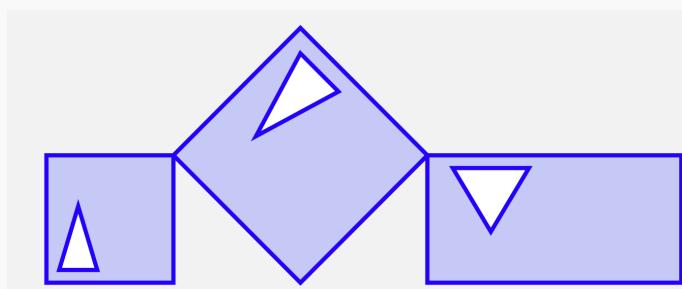
Sección 4.

Problemas

Proyecto con dos escenas.

Problema 5.2:

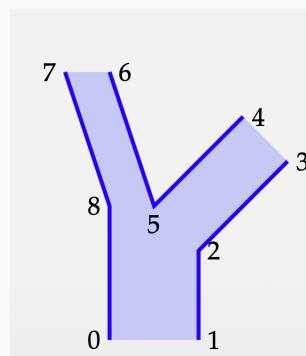
Crea un proyecto Godot con una escena principal con un nodo raíz compuesto. Ese nodo tendrá tres hijos, cada uno es una instancia de la escena del problema anterior, pero con una transformación distinta.



Escena simple

Problema 5.3:

Implementa un proyecto Godot con una función **Tronco**que crea y devuelve un **Node2D** con dos nodos hijos que forman la figura de aquí abajo (uno para el relleno y otro para las aristas).

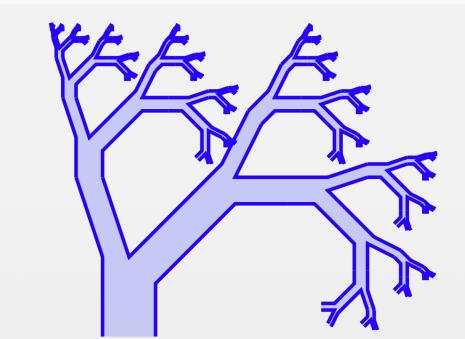


Índice	Coordenadas
0	(+0.0, +0.0)
1	(+1.0, +0.0)
2	(+1.0, +1.0)
3	(+2.0, +2.0)
4	(+1.5, +2.5)
5	(+0.5, +1.5)
6	(+0.0, +3.0)
7	(-0.5, +3.0)
8	(+0.0, +1.5)

Figura recursiva

Problema 5.4:

Implementa otro proyecto Godot que use la función del problema anterior para otra función, `Arbol(n)` que genera un árbol de escena con la figura de aquí abajo, que incluye múltiples instancias de `Tronco`, situadas recursivamente unas adyacentes a otras, hasta un nivel de recursividad dado por `n`.

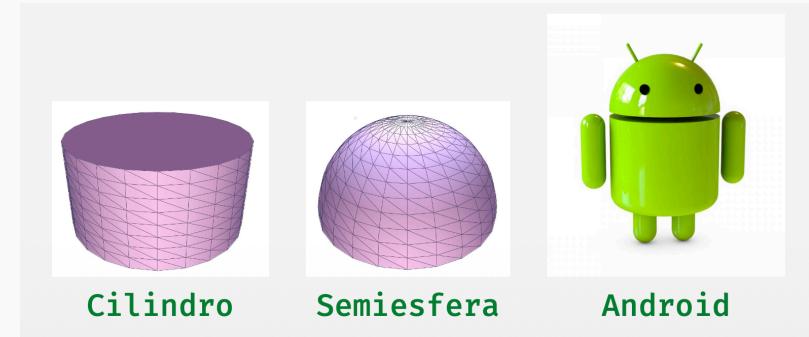


Fin de transparencias.

Árbol de escena 3D

Problema 5.5:

En un proyecto Godot 3D (puedes usar la práctica 2) para crear una figura como el logo de Android, usando únicamente dos objetos `ArrayMesh`, uno con un cilindro y otro con una semiesfera.



Informática Gráfica.

Sesión 6: Transformación de vértices.

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

Índice

Espacios de coordenadas y transformaciones.....	3
Transformación de vista	10
Transformación de proyección	28
Recortado y transformación del viewport.....	61
Problemas	76

1. Espacios de coordenadas y transformaciones..

Introducción.

El término **cauce gráfico (graphics pipeline)** se suele usar para referirnos al conjunto de pasos de cálculo que se realizan para visualizar polígonos en el contexto del **algoritmo de Z-buffer**

- El algoritmo de Z-buffer se usa para presentar polígonos incluyendo **eliminación de partes ocultas (EPO)** en 3D (es decir: lograr presentar únicamente las partes visibles de los polígonos que se dibujan).
- OpenGL, DirectX y otras librerías 3D usan Z-buffer.
- Estos pasos **se implementan en hardware** en las tarjetas gráficas modernas (GPUs: *Graphics Processing Units*).
- Estos pasos **no se aplican en otros algoritmos** de visualización y EPO en 3D, como por ejemplo en Ray-tracing.

Sección 1.

Espacios de coordenadas y transformaciones.

Pasos del cauce gráfico.

Los pasos del cauce gráfico suelen implementarse en secuencia, cada paso obtiene datos del anterior, los transforma de alguna manera y los entrega al siguiente paso. Los pasos (muy resumidos) son:

1. **Transformación** de coordenadas de vértices: cálculo de donde se proyecta en pantalla cada vértice.
2. **Recortado**: eliminación de partes de polígonos fuera de la zona visible.
3. **Rasterización y EPO**: cálculo de los píxeles donde se proyecta un polígono.
4. **Iluminación y texturización**: cálculo del color de cada pixel donde se proyecta un polígono.

la transformación y el recortado se pueden mezclar de diversas formas, ambos pasos son necesariamente previos a los otros dos, que también se pueden combinar de varias formas entre ellos

Sistemas de coordenadas (1/2)

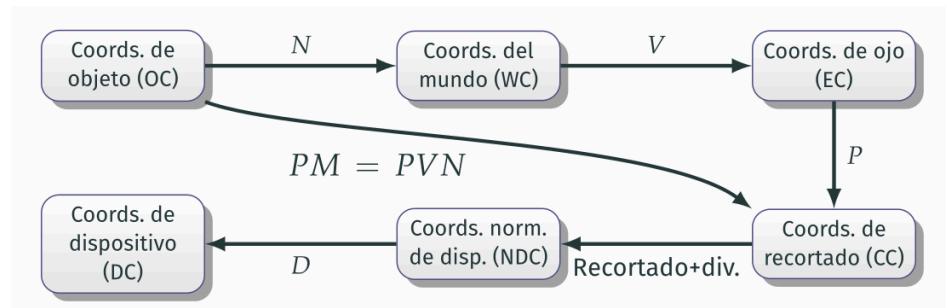
El cauce gráfico implementado en las engines y APIs de rasterización contempla una secuencia de 6 sistemas de coordenadas distintos. Las coordenadas de los vértices **son convertidas desde el primero al último**

La transformación de coordenadas entre cada sistema de referencia y el siguiente en la lista se hace mediante una matriz 4x4 específica de esa etapa (y en algún caso algo más).

1. **Coordenadas de objeto o maestras** (*Master coordinates*, OC): las coordenadas son distancias relativas a un sistema de referencia específico o distinto de cada objeto, que se crea en este espacio. En Godot, son las coordenadas de los vértices, relativas al nodo donde se encuentran.
2. **Coordenadas del mundo** (*World coordinates*, WC): son distancias relativas a un sistema de referencia único, común para todos los objetos de una escena.

Esquema de la transformación y recortado

En estas etapas del cauce gráfico, esencialmente los datos que se transforman son coordenadas de vértices y conectividad entre ellos. El esquema es el siguiente:



este esquema corresponde al recortado en CC (hay otras posibilidades, esta es la mejor).

Sistemas de coordenadas (2/2)

Los otros cuatro sistemas de coordenadas son:

3. **Coordenadas de cámara (o de ojo o de vista)** (*Eye coordinates or view-coordinates*, EC): son distancias relativas a un sistema de referencia posicionado y alineado con la *cámara virtual* en uso.
4. **Coordenadas de recortado** (*Clip coordinates*, CC): son distancias normalizadas (los vértices visibles están en el rango $[-1, +1]$ en los tres ejes), y con $w \neq 1$, relativas a un sistema asociado al rectángulo que forma la imagen en pantalla.
5. **Coordenadas normalizadas de dispositivo** (*Normalized device coordinates*, NDC): son similares a las coordenadas de recortado, pero con $w = 1$, y con todos los vértices en $[-1, +1]$ (los que están fuera se han eliminado).
6. **Coordenadas de dispositivo** (*Device coordinates*, DC): similares a NDC, pero ahora con las componentes X e Y en unidades de pixels.

Matrices de transformación

Las matrices de transformación (4x4) involucradas permiten convertir coordenadas en un sistema de coordenadas a coordenadas en otro:

- **La matriz de modelado y vista (*modelview*) M** , compuesta de:
 - ▶ **Matriz de modelado N** : convierte de OC a WC
 - ▶ **Matriz de vista V** : convierte de WC a EC
- **La matriz de proyección P** : convierte de EC a CC. (recibe coordenadas con $w = 1$, pero produce coordenadas en general con $w \neq 1$)
- **La matriz del viewport D** : convierte de NDC a DC (depende de la resolución de la imagen en pantalla y de la zona de esta donde se visualiza).

Las coordenadas de dispositivo (con $w = 1$ y en unidades de pixels) se usan como entrada para las siguientes etapas del cauce gráfico (rasterización, EPO, iluminación y texturas)

Sección 2. Transformación de vista

1. La matriz de vista 3D
2. Transformación de vista 3D en Godot

La transformación de vista.

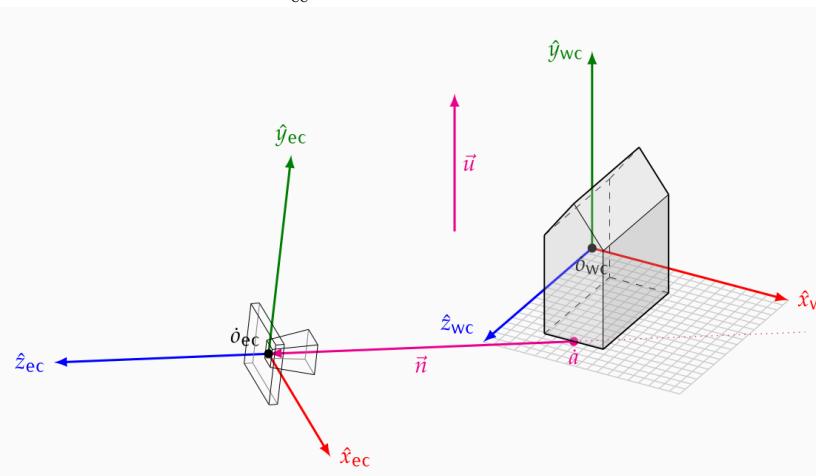
La **transformación de vista** es el cálculo que permite convertir **coordenadas de mundo** (*world coordinates*, WCC) en **coordenadas de ojo** (o de cámara) (*eye or camera coordinates*, ECC).

- Se usa un marco de referencia cartesiano $\mathcal{V} = \{\hat{x}_{\text{ec}}, \hat{y}_{\text{ec}}, \hat{z}_{\text{ec}}, \dot{o}_{\text{ec}}\}$, llamado **marco de cámara (o de vista)**, que está posicionado y alineado con la cámara virtual. Las **coordenadas de cámara (o de vista)** de un punto son las coordenadas de ese punto en el marco \mathcal{V} .
- Para hacer la conversión de coordenadas se debe usar la **matriz de vista**, la llamamos V .
- Puesto que el marco de coordenadas de mundo \mathcal{W} es cartesiano y \mathcal{V} también, la matriz V puede construirse fácilmente como la composición de una matriz de traslación por $\dot{o}_{\text{wc}} - \dot{o}_{\text{ec}}$ seguida de una matriz (ortonormal) de rotación, que tiene las coordenadas de mundo de \hat{x}_{ec} , \hat{y}_{ec} y \hat{z}_{ec} en **sus filas**.

Subsección 2.1. La matriz de vista 3D

El marco de coordenadas de vista o de cámara.

El **marco de coordenadas de vista** se construye usando el punto \hat{a} , el punto \hat{o}_{ec} , el vector \vec{u} y el vector \vec{n} . El observador está situado en \hat{o}_{ec} , y mira en la dirección de la rama negativa del eje Z ($-\hat{z}_{ec}$):



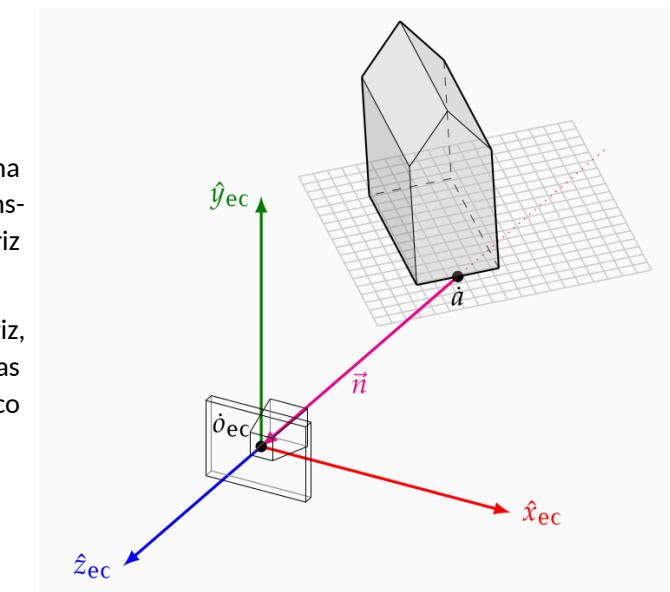
Cálculo del marco de vista.

El marco de referencia de vista \mathcal{V} , se define a partir de los siguientes parámetros:

- \hat{o}_{ec} = **posición del observador**: es el punto del espacio foco de la proyección, donde estaría situado el observador ficticio que contempla la escena (*projection reference point, PRP*)
- \vec{n} = **vector normal**: vector libre perpendicular al *plano de visión* (plano ficticio donde se proyecta la imagen perpendicular al eje óptico de la cámara virtual) (*view plane normal, VPN*).
- \hat{a} = **punto de atención**: punto en el eje óptico, por tanto se va a proyectar en el centro de la imagen (*look-at point*).
- \vec{u} = **vector hacia arriba**: es un vector libre que indica una dirección que el observador ve proyectada en vertical en la imagen (apuntando hacia arriba) (*view-up vector, VUP*)

De los tres parámetros \hat{o}_{ec} , \vec{n} y \hat{a} solo hay que especificar dos, ya que no son independientes, se cumple: $\hat{o}_{ec} = \hat{a} + \vec{n}$.

Escena posterior a transformación de vista.



Cálculo del marco de vista.

A partir de esos parámetros se obtiene se calculan los **versores del marco de vista**:

$$\hat{z}_{ec} = \frac{\vec{n}}{\|\vec{n}\|} \quad (\text{paralelo a } \vec{n} \text{ (VPN), normalizado})$$

$$\hat{x}_{ec} = \frac{\vec{u} \times \vec{n}}{\|\vec{u} \times \vec{n}\|} \quad (\text{perpendicular a } \vec{n} \text{ (VPN) y } \vec{u} \text{ (VUP), normalizado})$$

$$\hat{y}_{ec} = \hat{z}_{ec} \times \hat{x}_{ec} \quad (\text{perpendicular a los otros dos, normalizado})$$

Para que este cálculo pueda hacerse, los vectores \vec{u} y \vec{n} no pueden ser nulos ni paralelos, de forma que siempre $\|\vec{u} \times \vec{n}\| > 0$.

Coordinadas del mundo del marco de vista sC

El marco de referencia de vista se suele representar en memoria usando las coordenadas del mundo de los vectores y el punto (coordenadas relativas a \mathcal{W}), es decir:

$$\hat{x}_{\text{ec}} = \mathcal{W}(a_x, a_y, a_z, 0)^T = \mathcal{W}\mathbf{x}_{\text{ec}}$$

$$\hat{y}_{\text{ec}} = \mathcal{W}(b_x, b_y, b_z, 0)^T = \mathcal{W}\mathbf{y}_{\text{ec}}$$

$$\hat{z}_{\text{ec}} = \mathcal{W}(c_x, c_y, c_z, 0)^T = \mathcal{W}\mathbf{z}_{\text{ec}}$$

$$\dot{o}_{\text{ec}} = \mathcal{W}(o_x, o_y, o_z, 1)^T = \mathcal{W}\mathbf{o}_{\text{ec}}$$

Estas coordenadas se calculan a partir de las coordenadas de mundo (en el marco \mathcal{W}) de los vectores \vec{u}, \vec{n} y el punto \vec{o} como hemos visto. Esas coordenadas son \mathbf{u} , \mathbf{n} y \mathbf{o} , respectivamente.

La matriz V se puede construir directamente a partir de ellas.

Conversión de coordenadas: de vista a mundo

La **matriz de vista** V es la matriz que convierte desde coordenadas de mundo (en \mathcal{W}) hacia coordenadas de vista (en \mathcal{V}).

Su inversa V^{-1} hace la conversión contraria, **de vista a mundo**, y se puede escribir explícitamente V^{-1} , con las tuplas de coordenadas de mundo $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}, \mathbf{z}_{\text{ec}}$ y \mathbf{o}_{ec} dispuestas en sus columnas:

$$V^{-1} = \begin{pmatrix} a_x & b_x & c_x & o_x \\ a_y & b_y & c_y & o_y \\ a_z & b_z & c_z & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & o_x \\ 0 & 1 & 0 & o_y \\ 0 & 0 & 1 & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{T_{\mathbf{o}_{\text{ec}}}} \underbrace{\begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_R$$

Aquí se ha descompuesto V^{-1} en una **matriz ortonormal** R de alineamiento, seguida de una traslación al origen \mathbf{o}_{ec} de \mathcal{V} :

$$V^{-1} = T_{\mathbf{o}_{\text{ec}}} R$$

Cálculo de la matriz de vista

A partir de lo anterior es fácil obtener una expresión explícita de la **matriz de vista** V . Se puede escribir usando la composición inversa de la anterior:

$$V = (T_{\mathbf{o}_{\text{ec}}} R)^{-1} = R^{-1} T_{\mathbf{o}_{\text{ec}}}^{-1} = R^T T_{-\mathbf{o}_{\text{ec}}}$$

Donde se usa el hecho de que R es ortonormal. Expandiendo las matrices:

$$V = \underbrace{\begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{R^T} \underbrace{\begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{T_{-\mathbf{o}_{\text{ec}}}} = \begin{pmatrix} a_x & a_y & a_z & d_x \\ b_x & b_y & b_z & d_y \\ c_x & c_y & c_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde:

$$d_x = -\mathbf{x}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$$

$$d_y = -\mathbf{y}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$$

$$d_z = -\mathbf{z}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$$

Subsección 2.2.

Transformación de vista 3D en Godot

La clase Camera3D de Godot

En las aplicaciones 3D de Godot, la clase **Camera3D** es la encargada de **definir la matriz de vista** para el **viewport** (la zona de pantalla donde se visualiza cada frame).

- Es necesario situar una instancia de **Camera3D** (derivada de **Node3D**) en el árbol de escena.
- La instancia se registra como la cámara activa del primer viewport que se encuentra ascendiendo desde el nodo de la cámara hacia la raíz, si no hay ninguno, se registra en el viewport por defecto.
- Tiene una propiedad **transform** (una matriz, cuyas columnas son los ejes y origen de un marco del nodo). Ese marco de referencia es el **marco de vista V**
- La matriz **transform** permite transformar coordenadas de vista en coordenadas de mundo.
- La **matriz inversa** de **transform**, por tanto, es V^{-1} : **permite transformar coordenadas de mundo en coordenadas de vista.**

Desplazamientos locales una cámara

En Godot se puede desplazar la cámara en las direcciones de sus propios ejes, lo cual es muy típico en aplicaciones tipo *first person shooter*.

- En la aplicación el efecto es que el observador se mueve ciertas distancias **en las direcciones de los ejes del marco de cámara.**
- Puede ser hacia delante (eje Z negativo del marco de cámara), hacia atrás (Z+), hacia su izquierda (X-), su derecha (X+), arriba (Y+) y abajo (Y-).
- Para ello se puede usar el método **translate_object_local** de **Node3D**, aplicado al nodo cámara, lo cual desplaza el nodo en su propio sistema de referencia (compone traslaciones por la derecha):

```
cam.translate_object_local( Vector3( delta_x, delta_y, delta_z ) )
```

donde **delta_x**, **delta_y** y **delta_z** son las distancias a desplazar en las direcciones de los ejes X, Y y Z del marco de vista.

Situar y apuntar la cámara

En Godot se puede especificar las coordenadas de mundo de a (punto de atención, **p_atencion**), el vector **u** (vector hacia arriba **v_arriba**) y el origen o (punto **p_origen**) para apuntar una cámara (objeto **cam**) hacia dicho punto, para ello modificamos su sistema de referencia, se puede hacer de varias formas:

- Modificando la propiedad **transform** (de tipo **Transform3D**) del nodo cámara:

```
cam.transform.origin = p_origen  
cam.transform = cam.transform.looking_at( p_atencion, v_arriba )
```

- Modificando el nodo directamente, con el método **look_at** de **Node3D**:

```
cam.position = p_origen  
cam.look_at( p_atencion, v_arriba )
```

- Usando el método **look_at_from_position** de **Node3D**:

```
cam.look_at_from_position( p_origen, p_atencion, v_arriba )
```

Rotaciones locales de una cámara

También se puede rotar la cámara entorno a sus propios ejes, (de nuevo es muy típico en aplicaciones tipo *first person shooter*).

- En la aplicación el efecto es que el observador rota su punto de vista entorno a sus propios ejes.
- Puede ser rotar su vista hacia arriba y hacia abajo (entorno al eje X del marco de cámara), girar su vista hacia la izquierda y hacia la derecha (entorno al eje Y), y hacer un giro lateral (entorno al eje Z).
- Para ello se puede usar el método **rotate_object_local** de **Node3D**, aplicado al nodo cámara, lo cual rota el nodo en su propio sistema de referencia (compone rotaciones por la derecha):

```
cam.rotate_object_local( eje_rotacion, angulo_rad )
```

donde **eje_rotacion** es un vector unitario que indica el eje de rotación en el marco de vista, y **angulo_rad** es el ángulo de rotación en radianes.

Cámara orbital

Otra posibilidad es el uso de una **cámara orbital**, la cual *orbita* alrededor de un punto, usando dos ángulos de rotación y una distancia a dicho punto:

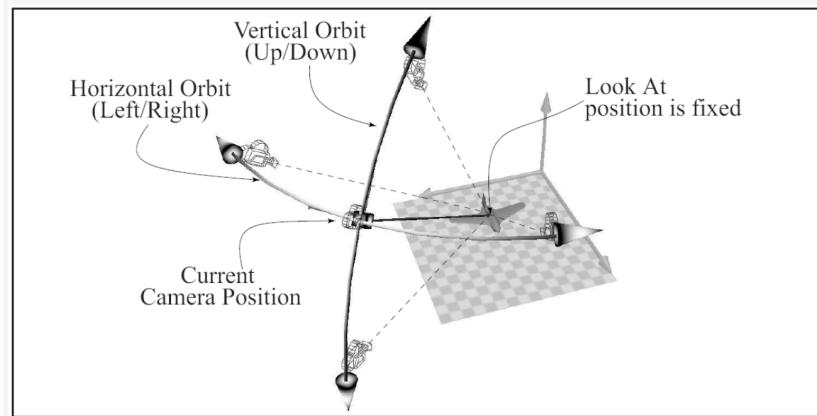
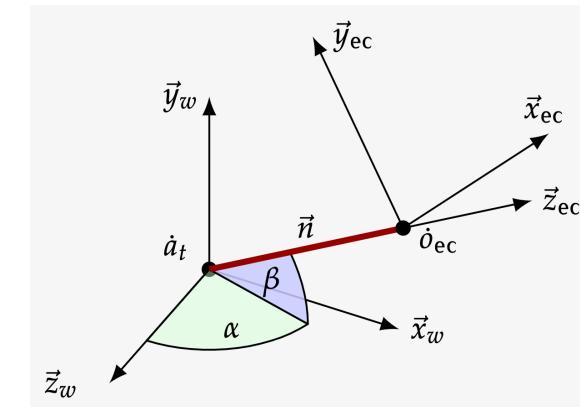


Figura obtenida de: K.Sung, P.Shirley, S.Baer **Essentials of Interactive Computer Graphics: Concepts and Implementation**. Ed. Routledge, 2008. Ver: [página del libro](#).

Marco de vista de la cámara orbital

El marco de vista está desplazado una distancia en el eje Z del mundo (a partir del punto de atención \vec{a}_t), y luego rotado usando dos ángulos α y β (entorno a los ejes Y y X, respectivamente) que determinan el vector \vec{n} :



Cámara orbital en Godot

En Godot, una posibilidad es usar el siguiente código en el nodo **Camera3D** en uso, que actualiza el marco de vista asignando su propiedad **transform**:

```
var ahr := ((45.0+float(dxy.x))*2.0*PI)/360.0 ## ang.horiz.radi.
var avr := ((30.0+float(dxy.y))*2.0*PI)/360.0 ## ang.vert.radi.
var tras := Transform3D().translated( Vector3( 0.0, 0.0, dz))
var rotx := Transform3D().rotated( Vector3.RIGHT, -avr )
var roty := Transform3D().rotated( Vector3.UP, ahr )
transform = roty*rotx*tras ## actualiza transform del nodo cámara
```

Los datos de entrada son **dxy** y **dz**

- **dxy** son dos enteros (**Vector2i**) con los ángulos de rotación en grados (initialmente a cero). A partir de ahí se calculan **ahr** (α) y **avr** (β), los ángulos en radianes.
- **dz** es un flotante con la distancia al origen (es decir, es $\|\vec{n}\|$)

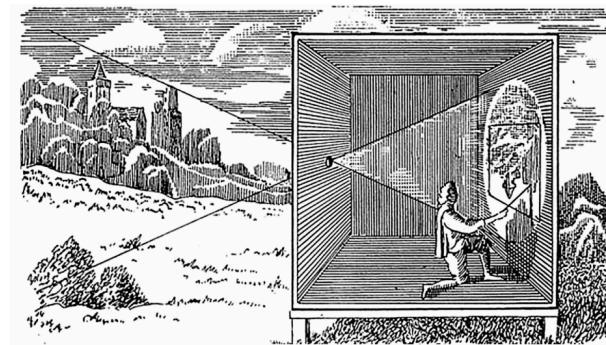
Estos tres valores se modifican interactivamente con el ratón y el teclado.

Sección 3.
Transformación de proyección

1. Tipos de proyección en 3D.
2. El *view-frustum*. Parámetros de proyección.
3. La matriz de proyección.

Introducción

La transformación de proyección 3D (perspectiva) emula la proyección que ocurre idealmente en una *cámara oscura*, sobre la pared opuesta a la apertura. Es similar a lo que ocurre en una cámara de fotografía, al proyectarse la escena sobre el sensor.



Grabado de una *Camera Obscura*, por Athanasius Kircher en *Ars Magna Lucis et Umbrae* (1645).
www.essentialvermeer.com/camera_obscura/co_one.html

Subsección 3.1.

Tipos de proyección en 3D.

El plano de visión. Tipos de proyección

Los vértices se proyectan sobre un plano alineado con el sistema de referencia de la cámara:

- Dicho plano se denomina **plano de visión (viewplane)**, es siempre perpendicular al eje Z del marco de vista.

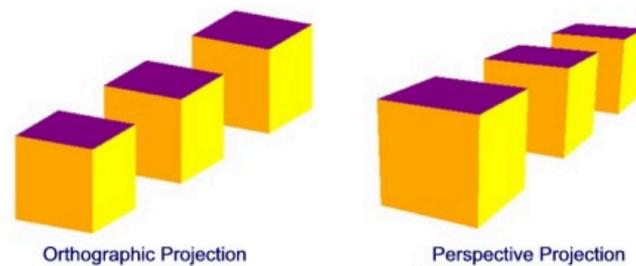
La proyección puede ser de dos tipos

Proyección perspectiva: los vértices se proyectan sobre el plano de visión usando líneas que van desde cada punto al origen del marco de coordenadas de la cámara (a esas líneas se les llama **proyectores**, el origen actua como **foco** de la proyección). La coordenada Z del plano de visión debe ser estrictamente positiva.

Proyección ortográfica: (o paralela) los proyectores son todos paralelos al eje Z. El plano de visión puede estar situado en cualquier valor de Z. Es un caso límite de la perspectiva, con el foco infinitamente alejado de la escena.

Comparación de proyecciones

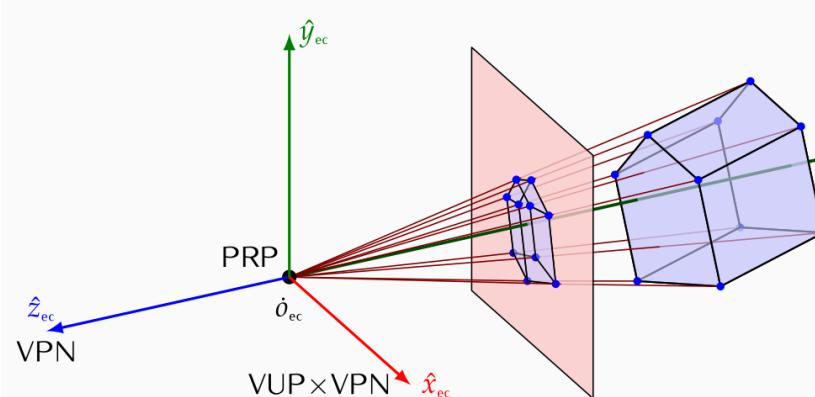
Aunque ninguna de las dos formas de proyección es igual al comportamiento del sistema visual humano, la proyección perspectiva nos parece más natural (la ortográfica es poco realista):



A la izquierda, se interpreta que el cubo más lejano es más grande que los otros, aunque en la imagen son los tres del mismo tamaño.

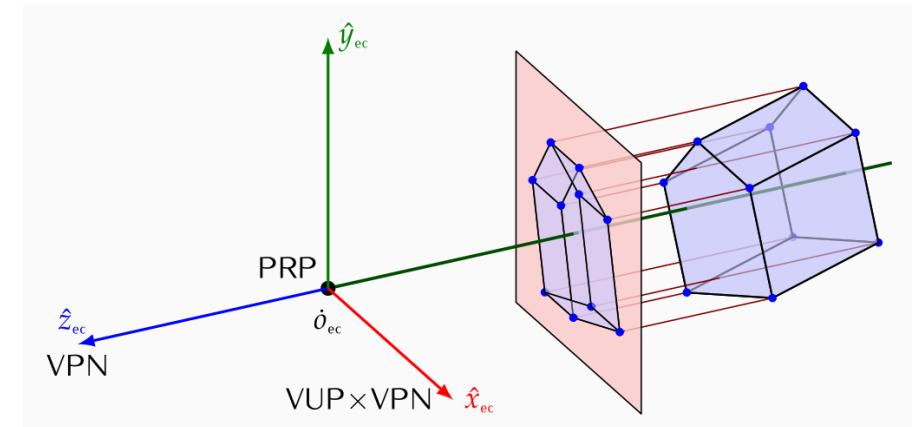
Proyección perspectiva

La **proyección perspectiva** cambia el tamaño de los objetos, usando un factor de escala s que crece de forma inversamente proporcional a la distancia (d_z) en Z desde el objeto al foco (s es de la forma $1/(ad_z + b)$)



Proyección paralela

La **proyección paralela** no cambia la escala, y la proyección se puede ver como una transformación afín:



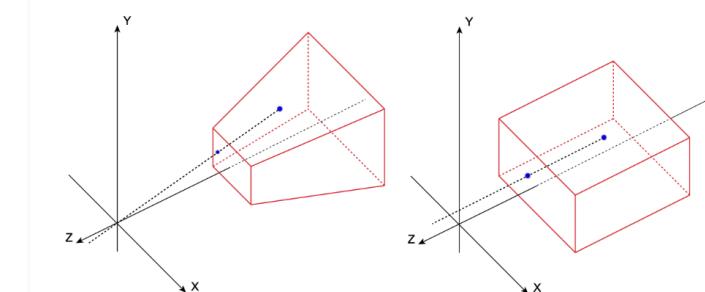
Subsección 3.2.

El **view-frustum**. Parámetros de proyección.

El **view-frustum**

El **view-frustum** designa la región del espacio de la escena que es visible en el viewport. Su forma depende del tipo de proyección:

- Perspectiva: es un tronco de pirámide rectangular (izq.).
- Ortográfica: es un paralelepípedo ortogonal u ortoedro (der.).



Transformación del view-frustum en un cubo

El view-frustum está determinado por los 6 planos que contienen a las 6 caras que lo delimitan.

- Estos planos se determinan por sus coordenadas en el marco de coordenadas de vista.
- La transformación de proyección transforma el view-frustum (en coordenadas de vista) en un cubo de lado 2 centrado en el origen, entre -1 y 1 en los tres ejes (en coordenadas de recortado, normalizadas).
- La proyección ortográfica es una transformación afín: una traslación seguida de escalado no necesariamente uniforme.
- La proyección perspectiva no es una transformación afín, aunque se puede expresar usando una transformación afín en coordenadas homogéneas 4D, seguida de una proyección de 4D a 3D.

Parámetros del view-frustum. Extensión en X e Y.

Respecto de los otros cuatro valores (l, r, b y t), determinan la extensión en X y en Y:

- l (**left**) y r (**right**) son los límites en X del view-frustum ($l \neq r$).
- b (**bottom**) y t (**top**) son los límites en Y ($b \neq t$).
- En proy. ortográfica:
 - ▶ El plano $x_{\text{ec}} = l$ en EC se transforma en el plano $x_{\text{ndc}} = -1$ en NDC.
 - ▶ El plano $x_{\text{ec}} = r$ en EC se transforma en el plano $x_{\text{ndc}} = +1$ en NDC.
 - ▶ El plano $y_{\text{ec}} = b$ en EC se transforma en el plano $y_{\text{ndc}} = -1$ en NDC.
 - ▶ El plano $y_{\text{ec}} = t$ en EC se transforma en el plano $y_{\text{ndc}} = +1$ en NDC.
- En proy. perspectiva:
 - ▶ El plano $-nx_{\text{ec}} = lz_{\text{ec}}$ (EC) se transf. en el plano $x_{\text{ndc}} = -1$ en NDC.
 - ▶ El plano $-nx_{\text{ec}} = rz_{\text{ec}}$ (EC) se transf. en el plano $x_{\text{ndc}} = +1$ en NDC.
 - ▶ El plano $-ny_{\text{ec}} = bz_{\text{ec}}$ (EC) se transf. en el plano $y_{\text{ndc}} = -1$ en NDC.
 - ▶ El plano $-ny_{\text{ec}} = tz_{\text{ec}}$ (EC) se transf. en el plano $y_{\text{ndc}} = +1$ en NDC.

Parámetros del view-frustum. Extensión en Z

Los 6 valores l, r, t, b, n y f (los **parámetros** de frustum) determinan la transformación de la tupla $(x_{\text{ec}}, y_{\text{ec}}, z_{\text{ec}})$, que está en coordenadas de vista en la tupla $(x_{\text{ndc}}, y_{\text{ndc}}, z_{\text{ndc}})$ (*coordenadas normalizadas de dispositivo*, entre -1 y 1):

- Los valores n (**near**) y f (**far**) son los límites en Z del view-frustum, pero cambiados de signo (se cumple $nn = f$).
 - ▶ El plano $z_{\text{ec}} = -n$ en EC se transforma en el plano $z_{\text{ndc}} = -1$ en NDC.
 - ▶ El plano $z_{\text{ec}} = -f$ en EC se transforma en el plano $z_{\text{ndc}} = +1$ en NDC.
- En la proyección perspectiva, se exige además $0 < n < f$.
- Aunque no se exige así, lo usual es que seleccione $n < f$, es decir, el view-frustum se extiende en Z en el intervalo $[-f, -n]$. En adelante supondremos $n < f$, de forma que:
 - ▶ El plano $z_{\text{ec}} = -n$ se llama **plano de recorte delantero**.
 - ▶ El plano $z_{\text{ec}} = -f$ se llama **plano de recorte trasero**.

Propiedades de la extensión en X e Y

Hay que tener en cuenta que:

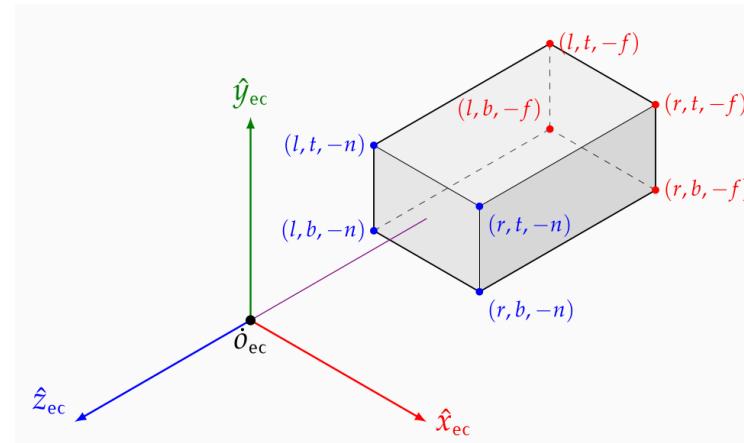
- Aunque esto no es requerido estrictamente, usualmente se seleccionan los parámetros de forma que $l < r$ y $b < t$.
- Cuando se cumple $l = -r$ y $b = -t$, decimos que el **view-frustum está centrado** (el eje Z pasa por el centro de las caras delantera y trasera). Esto es lo más usual, y se corresponde con lo que ocurre en una cámara.
- El valor $(r - l)/(t - b)$ suele coincidir con la relación de aspecto del viewport (ancho/alto, o bien núm.columnas/núm.filas). Si esto no ocurre los objetos aparecerán deformados en la imagen.

En adelante supondremos que siempre seleccionamos $l < r$ y $b < t$.

Parámetros en la proyección ortográfica.

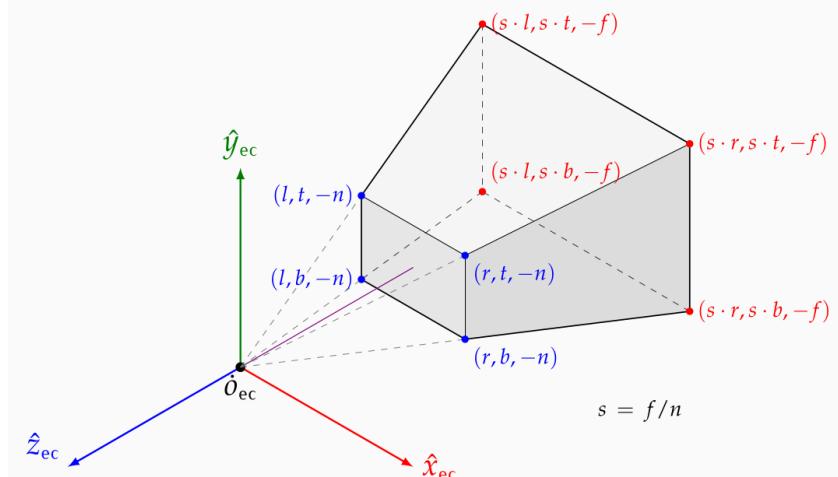
En pr. ortográfica el view-frustum es un **orthoedro**. Contiene los puntos cuyas coordenadas de cámara (x_{ec}, y_{ec}, z_{ec}) cumplen:

$$l \leq x_{ec} \leq r \quad b \leq y_{ec} \leq t \quad -f \leq z_{ec} \leq -n$$



Parámetros en la proyección perspectiva (1/2)

En perspectiva, el view-frustum es una **pirámide rectangular truncada**:



Parámetros en la proyección perspectiva (2/2)

Los puntos dentro del view-frustum son aquellos cuyas coordenadas de cámara (x_{ec}, y_{ec}, z_{ec}) cumplen:

En el eje X:

$$l \leq x_{ec} \left(\frac{n}{-z_{ec}} \right) \leq r$$

En el eje Y:

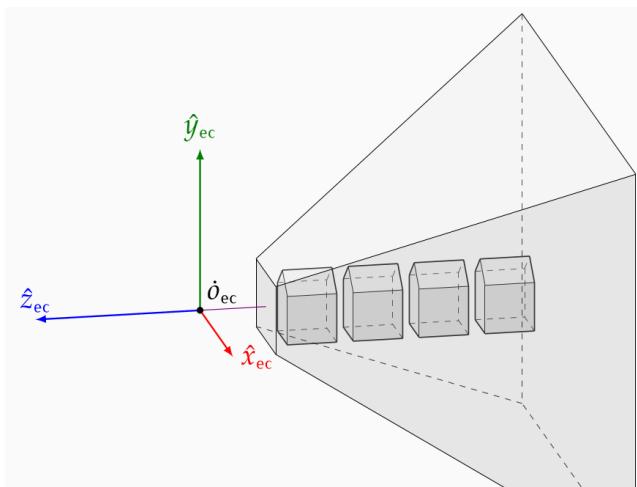
$$b \leq y_{ec} \left(\frac{n}{-z_{ec}} \right) \leq t$$

En el eje Z:

$$-f \leq z_{ec} \leq -n$$

Escena de ejemplo para transformación perspectiva

Suponemos que partimos de una escena que vamos a proyectar usando perspectiva. En el espacio de coordenadas de cámara, la escena es esta:



Proyección perspectiva sobre el plano delantero

Podemos suponer que los puntos se proyectan sobre el plano frontal del view-frustum(coord. Z igual a $-n$) con foco en \mathbf{o}_{ec} :

- Dado un punto $p = \mathcal{V}(x_{\text{ec}}, y_{\text{ec}}, z_{\text{ec}}, w_{\text{ec}})$ queremos calcular las coordenadas de su proyección (x', y', z', w') (en principio, con $w' = w_{\text{ec}} = 1$).
- Si se asume $z_{\text{ec}} < 0$ (el punto está en la rama negativa del eje Z), podemos hacer:

$$x' = \frac{n x_{\text{ec}}}{-z_{\text{ec}}} \quad y' = \frac{n y_{\text{ec}}}{-z_{\text{ec}}} \quad z' = \frac{n z_{\text{ec}}}{-z_{\text{ec}}}$$

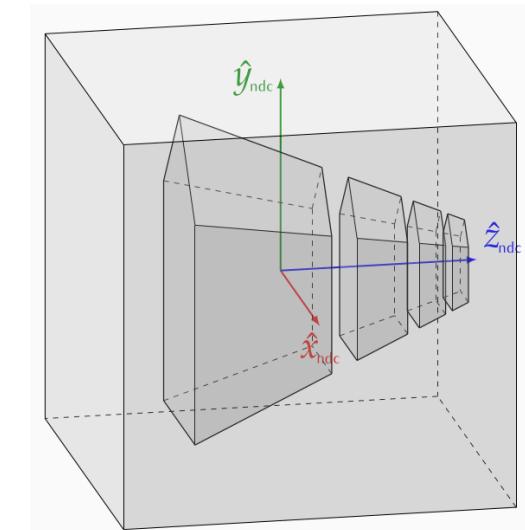
esta transformación tiene dos problemas:

- Las coordenadas resultado no están entre -1 y 1 .
- Colapsa o *aplana* todas las coordenadas Z (siempre $z' = -n$).

Escena proyectada por la transformación perspectiva

El efecto de la **transformación de proyección perspectiva** será:

- hacer más pequeños los objetos más alejados del observador, y
- situar la escena en un cubo de lado 2 y centrada en el origen del **marco de coordenadas normalizadas de dispositivo (NDC)**.



Normalización de coordenadas X e Y

Si el punto original está en el *view-frustum*, entonces:

- x' está en el intervalo $[l, r]$
- y' está en el intervalo $[b, t]$.
- queremos dejar ambas coordenadas en el intervalo $[-1, 1]$
- podemos usar un escalado y traslación adicionales en X e Y:

$$x'' = 2\left(\frac{x' - l}{r - l}\right) - 1 = \frac{a_0 x_{\text{ec}}}{-z_{\text{ec}}} - a_1 = \frac{a_0 x_{\text{ec}} + a_1 z_{\text{ec}}}{-z_{\text{ec}}}$$

$$y'' = 2\left(\frac{y' - b}{t - b}\right) - 1 = \frac{b_0 y_{\text{ec}}}{-z_{\text{ec}}} - b_1 = \frac{b_0 y_{\text{ec}} + b_1 z_{\text{ec}}}{-z_{\text{ec}}}$$

donde hemos definido estas cuatro constantes:

$$a_0 \equiv \frac{2n}{r - l} \quad a_1 \equiv \frac{r + l}{r - l} \quad b_0 \equiv \frac{2n}{t - b} \quad b_1 \equiv \frac{t + b}{t - b}$$

Información de profundidad y normalización en Z

El problema está de hacer $z' = -n$ está en que **se pierde información de profundidad en Z**, que es necesaria para EPO). Para evitarlo, se usa una función lineal de z con dos constantes c_0 y c_1 :

$$z'' = \frac{c_0 z_{\text{ec}} + c_1}{-z_{\text{ec}}} \quad \text{donde: } c_0 = \frac{n+f}{n-f}, \quad c_1 = \frac{2fn}{n-f}.$$

- los dos valores c_2 y c_3 se eligen de forma que, para $z_{\text{ec}} = -n$, se hace $z'' = -1$, y para $z_{\text{ec}} = -f$, se hace $z'' = 1$.
- es decir: el rango $[-f, -n]$ se lleva al rango $[-1, 1]$ (invirtiendo el orden).
- esta transformación conserva el orden (invertido) de las coordenadas Z (no *aplana*)
- ahora, **valores menores de Z implican puntos más cercanos al observador, y valores mayores, más lejanos.**

Coordenadas cartesianas del punto proyectado

En resumen, tenemos estas tres igualdades:

$$x'' = \frac{a_0 x_{\text{ec}} + a_1 z_{\text{ec}}}{-z_{\text{ec}}}$$

$$y'' = \frac{b_0 x_{\text{ec}} + b_1 z_{\text{ec}}}{-z_{\text{ec}}}$$

$$z'' = \frac{c_0 z_{\text{ec}} + c_1}{-z_{\text{ec}}} = \frac{c_0 z_{\text{ec}} + c_1 w_{\text{ec}}}{-z_{\text{ec}}}$$

esta transformación incluye una división, y por tanto

- no se puede implementar con una matriz** como hacíamos con las anteriores (no es lineal)
- aunque sí transforma líneas rectas en líneas rectas

Obtención de las coordenadas de recortado

Para solventar el problema anterior (para poder usar una matriz), se definen las **coordenadas de recortado (clip coordinates)**, a partir de las coordenadas de cámara del original:

$$\begin{aligned} x_{\text{cc}} &= a_0 x_{\text{ec}} + a_1 z_{\text{ec}} \\ y_{\text{cc}} &= b_0 y_{\text{ec}} + b_1 z_{\text{ec}} \\ z_{\text{cc}} &= c_0 z_{\text{ec}} + c_1 w_{\text{ec}} \\ w_{\text{cc}} &= -z_{\text{ec}} \end{aligned}$$

Esta transformación ya **sí se puede hacer con una matriz 4x4**:

- se ha eliminado la división por $-z_{\text{ec}}$, el resto es igual
- esta división se hace más adelante en el cauce gráfico
- para ello, el denominador de la división ($-z_{\text{ec}}$) queda guardado en w_{cc} (**que ya no es 1**).

La matriz de proyección perspectiva Q

Con todo lo dicho, la proyección perspectiva se puede realizar usando una matriz Q , que se aplica a coordenadas de cámara (con $w_{\text{ec}} = 1$) y produce coordenadas de recortado (con $w_{\text{cc}} \neq 1$):

$$\begin{pmatrix} x_{\text{cc}} \\ y_{\text{cc}} \\ z_{\text{cc}} \\ w_{\text{cc}} \end{pmatrix} = \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & b_0 & b_1 & 0 \\ 0 & 0 & c_0 & c_1 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_{\text{ec}} \\ y_{\text{ec}} \\ z_{\text{ec}} \\ 1 \end{pmatrix}$$

evidentemente, podemos definir entonces la matriz Q de esta forma:

$$Q = \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & b_0 & b_1 & 0 \\ 0 & 0 & c_0 & c_1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

La proyección ortográfica.

En el caso de la **proyección ortográfica** (*orthographic projection*), se hace proyección en una dirección paralela al eje Z:

- Esta transformación **solo requiere la normalización de los rangos de valores en los tres ejes** (se usa traslación más escalado)
- (1) traslación T_{-c} (lleva el centro del paralelepípedo, c , al origen), donde

$$c \equiv \left(\frac{l+r}{2}, \frac{t+b}{2}, \frac{f+n}{2} \right)$$

- (2) escalado E_s con factores $s = (s_x, s_y, s_z)$ (deja los valores en $[-1, 1]$ en los tres ejes), donde:

$$s_x \equiv \frac{2}{r-l} \quad s_y \equiv \frac{2}{t-b} \quad s_z \equiv \frac{-2}{f-n}$$

(en Z se cambia de signo para *invertir* el eje Z).

Construcción de la matriz: ajuste en vertical

Para construir una matriz de proyección (perspectiva u ortográfica) a partir de los seis valores reales l, r, b, t, n y f , usando las constantes definidas antes.

- Otra posibilidad para la matriz perspectiva (más intuitivo) es usar los parámetros β y a (además de n y f):
 - ▶ $\beta \equiv$ es la **apertura vertical del campo de visión** (*fovy*), es el ángulo en grados (entre 0 y 180.0) que hay entre la cara superior y la inferior del frustum
 - ▶ $a_v \equiv$ es la **relación de aspecto vertical** (*vertical aspect ratio*) de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).

En este caso, los valores l, r, b y t se obtienen como:

$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv a_v t \quad l \equiv -r$$

La matriz de proyección ortográfica

La matriz de proyección ortográfica O se obtiene por tanto como composición de T_{-c} seguido de E_s , es decir $O = E_s \cdot T_{-c}$, o lo que es lo mismo:

$$O = \begin{pmatrix} a'_0 & 0 & 0 & a'_1 \\ 0 & b'_0 & 0 & b'_1 \\ 0 & 0 & c'_0 & c'_1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{donde:} \quad \begin{cases} a'_0 \equiv \frac{2}{r-l} & a'_1 \equiv -\frac{r+l}{r-l} \\ b'_0 \equiv \frac{2}{t-b} & b'_1 \equiv -\frac{t+b}{t-b} \\ c'_0 \equiv \frac{-2}{f-n} & c'_1 \equiv -\frac{f+n}{f-n} \end{cases}$$

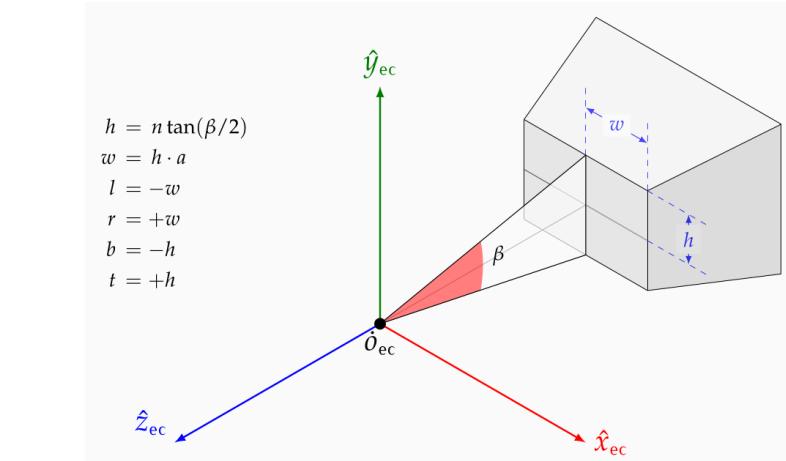
de forma que ahora hacemos:

$$(x_{cc}, y_{cc}, z_{cc}, w_{cc})^T = O(x_{ec}, y_{ec}, z_{ec}, w_{ec})^T,$$

donde w_{cc} ahora sí vale 1 con seguridad.

Parámetros de la matriz de proyección

El significado del valor β (*fovy*) se aprecia en esta figura, donde se han señalado los tamaños del plano delantero en horizontal $w = r - l$ y en vertical $h = t - b$:



Esta perspectiva es *centrada*, ya que $r = -l$ y $t = -b$

Construcción de la matriz: ajuste en horizontal

Alternativamente, se puede usar la apertura horizontal de campo (en lugar de la vertical).

Los parámetros son:

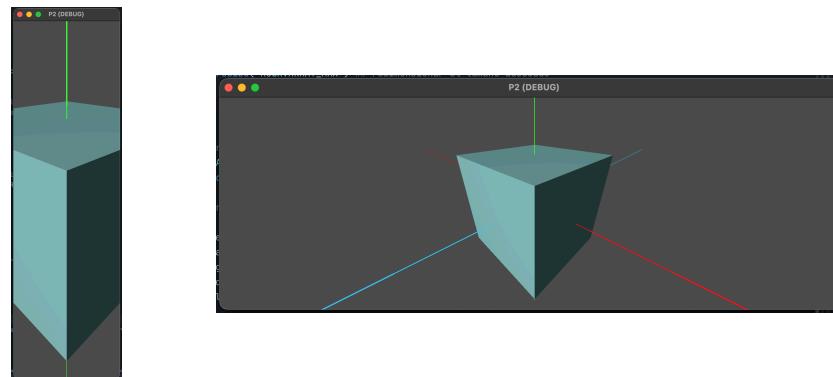
- $\alpha \equiv$ es la **apertura horizontal del campo de visión** (`fovX`), es el ángulo en grados (entre 0 y 180.0) que hay entre la cara izquierda y la derecha del frustum
- $a_h \equiv$ es la **relación de aspecto horizontal** (`horizontal aspect ratio`) de la imagen a producir: su alto (n. filas) dividido por el ancho (n. columnas), es la inversa de a_v .

En este caso, los valores l, r, b y t se obtienen como:

$$r \equiv n \tan\left(\frac{\alpha}{2}\right) \quad l \equiv -r \quad t \equiv a_h r \quad b \equiv -t$$

Ejemplo de ajuste en vertical

Por defecto, Godot usa **ajuste en vertical** (`KEEP_HEIGHT`), de forma que la apertura vertical del campo de visión es la que se fija con `fov`, y la apertura horizontal se ajusta según la relación de aspecto del viewport.



Al redimensionar la ventana, el cubo se ve siempre **completo en vertical**.

Matriz de proyección 3D en Godot

La clase `Camera3D` permite configurar la matriz de proyección mediante estas propiedades:

- **projection: tipo de proyección**, de tipo `ProjectionType`, puede valer `PROJECTION_PERSPECTIVE` o `PROJECTION_ORTHOGONAL`.
- **size: tamaño vertical u horizontal** del ortoedro visible (solo para `PROJECTION_ORTHOGONAL`), de tipo `float`.
- **fov: apertura del campo de visión** (en grados, `float`), vertical u horizontal (solo para `PROJECTION_PERSPECTIVE`), de tipo `float`. Por defecto es 75°.
- **keep_aspect: tipo de ajuste (vertical u horizontal)**, de tipo `KeepAspect`, puede valer `KEEP_WIDTH`, entonces las propiedades `fov` y `size` se interpretan en horizontal, o bien `KEEP_HEIGHT` (por defecto) y entonces se interpretan en vertical.
- **near: distancia al plano de recorte delantero** (valor n)
- **far: distancia al plano de recorte trasero** (valor f)

Al redimensionar la ventana, el cubo se ve siempre **completo en horizontal**.

Sección 4.

Recortado y transformación del viewport.

1. Recortado de primitivas y división por w
2. Transformación del viewport

4. Recortado y transformación del viewport..
4.1. Recortado de primitivas y división por w .

Recortado

Una vez se tienen las coordenadas de recortado de los vértices, se comprueban que primitivas estan dentro o fuera del viewfrustum (que en CC es un cubo de lado dos unidades centrado en el origen):

- Las primitivas completamente dentro de la zona visible **se mantienen**.
- Las primitivas completamente fuera de la zona visible **se descartan**.
- Las primitivas parcialmente dentro **se dividen en partes**: unas completamente dentro (se visualizan) y otras completamente fuera (que se descartan). Esto se hace mediante la **inserción de algunos vértices nuevos** justo en los planos que delimitan el view-frustum.

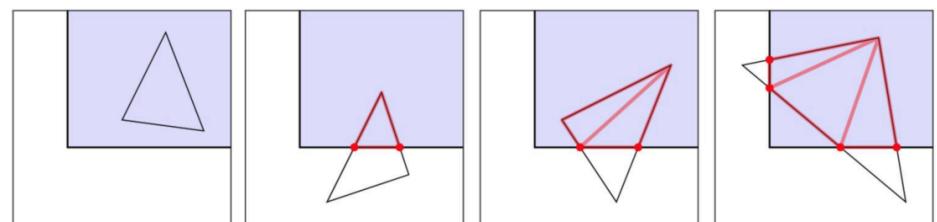
Subsección 4.1.

Recortado de primitivas y división por w

4. Recortado y transformación del viewport..
4.1. Recortado de primitivas y división por w .

Inserción de nuevos vértices y triángulos

Varios ejemplos de recortado de triángulos:



- Las coordenadas y otros atributos de los nuevos vértices se interpolan a partir de los vértices en los dos extremos de la arista donde se inserta el nuevo.
- Se hace recortado independiente por cada uno de los 6 planos de recorte.

Figura Copyright © 2012 de Jason L. McKesson, en
Learning Modern 3D Graphics Programming: paroj.github.io/gltut

División por W. Coordenadas normalizadas de dispositivo.

Los vértices (en el view-frustum) resultado del recorte tienen coordenadas de recorte con $w_{cc} \neq 0$ (si $P = Q$, entonces además $w_{cc} = 1$).

El siguiente paso es hacer la división por w_{cc} de las tres componentes. Se obtienen las **coordenadas normalizadas de dispositivo (NDC)**, con componente W de nuevo a 1:

$$(x_{ndc}, y_{ndc}, z_{ndc}, 1) = \frac{1}{w_{cc}} (x_{cc}, y_{cc}, z_{cc}, w_{cc})$$

Los valores x_{ndc} , y_{ndc} y z_{ndc} están los tres en el intervalo $[-1, 1]$, ya que los vértices ya han pasado el recortado y están todos dentro del ortoedro visible en NDC.

Subsección 4.2. Transformación del viewport

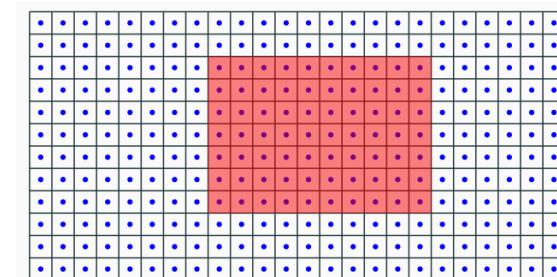
Transformación de Viewport

El siguiente paso consiste en calcular en qué posiciones de la imagen se proyecta cada vértice:

- Este paso se puede modelar como una transformación lineal que llamaremos **transformación de viewport**. El término **viewport** hace referencia a la zona rectangular de la ventana donde se proyectarán los polígonos que están en el cubo visible (un bloque rectangular de pixels)
- Esta transformación produce **coordenadas de dispositivo o de ventana (DC: device coordinates, o también llamadas screen coordinates, o window coordinates)**. Las coordenadas X e Y en DC se expresan en unidades de pixels.
- La transformación de viewport es lineal y consta simplemente de escalados y traslaciones.
- La coordenada Z se transforma y se conserva para poder hacer después *eliminación de partes ocultas*.

Coordenadas de dispositivo y pixels del viewport

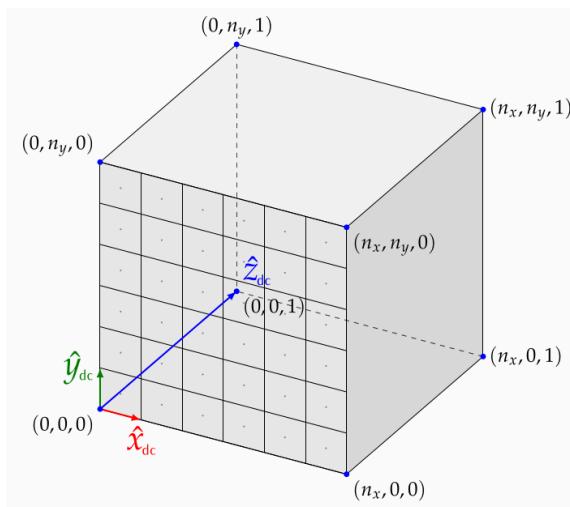
En coordenadas de dispositivo, podemos asociar una región cuadrada (de lado unidad) a cada pixel en el plano de la ventana. El **viewport** (en rojo) es un bloque rectangular de pixels, contenido en el bloque rectangular correspondiente a la ventana o imagen completa:



los centros de los pixels (puntos azules) tienen coordenadas de dispositivo con parte fraccionaria igual a 1/2. Los bordes entre pixels tienen coordenadas sin parte fraccionaria (enteras).

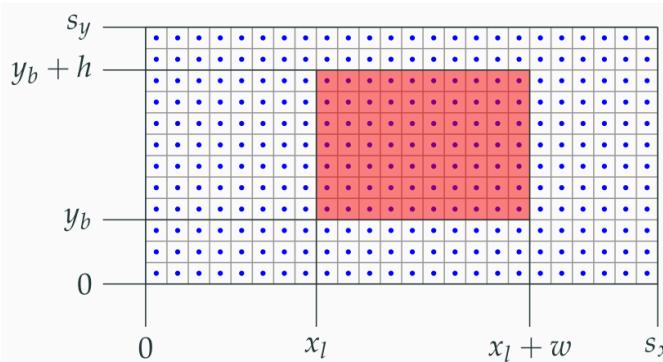
El espacio de coordenadas de dispositivo

En 3D el espacio de coordenadas de dispositivo es un ortoedro. Se puede visualizar como aparece aquí, incluyendo el marco de coordenadas de dispositivo:



Parámetros del viewport

Suponemos que la ventana tiene s_x columnas y s_y filas, y que el gestor de ventanas acepta coordenadas de pixels enteras no negativas:



Se cumplen las desigualdades: $\begin{cases} 0 \leq x_l < x_l + w \leq s_x \\ 0 \leq y_b < y_b + h \leq s_y \end{cases}$

Matriz del viewport. Parámetros.

Para convertir a coordenadas de dispositivo se usa una matriz D , que depende de estos parámetros (ver figura):

- x_l, y_b número de columna y fila (enteros no negativos) del pixel que ocupa, en la ventana, la esquina inferior izquierda del viewport.
- w, h (*width* y *height*) número total (enteros no negativo) de columnas y de filas de pixels (respectivamente) que ocupa el viewport.
- n_d, f_d rango de valores de salida en Z en DC. El valor n_d es la profundidad más cercana posible al observador, y f_d la más lejana. En OpenGL suele ser $n_d = 0$ y $f_d = 1$, en otras APIs puede ser $n_d = -1$.

Aunque los cuatro parámetros relevantes (x_l, y_b, w y h) son enteros, las coordenadas de dispositivos son valores reales, ya que las posiciones de los vértices en DC son en general no enteras (no coinciden necesariamente con los centros o bordes de los pixels).

La transformación de viewport

En NDC las coordenadas están en $[-1, 1]$, luego hay que hacer:

1. traslación de la esquina $p = (-1, -1, -1)$ al origen (matriz T_{-p})
2. escalado uniforme (por $1/2$) y por $(w, h, f_d - n_d)$ (matriz E_s)
3. traslación del origen a $q = (x_l, y_b, n_d)$ (matriz T_q)

Con lo cual la **transformación del viewport** D queda como el producto de estas tres matrices:

$$D = T_q E_s T_{-p}$$

Por tanto, las **coordenadas de dispositivo** $(x_{dc}, y_{dc}, z_{dc}, 1)$ se definen a partir de las normalizadas $(x_{ndc}, y_{ndc}, z_{ndc}, 1)$ de esta forma:

$$\begin{aligned} x_{dc} &= (x_{ndc} + 1)w/2 + x_l \\ y_{dc} &= (y_{ndc} + 1)h/2 + y_b \\ z_{dc} &= (z_{ndc} + 1)(f_d - n_d)/2 + n_d \end{aligned}$$

La matriz de viewport D

Como consecuencia de todo lo anterior, la **matriz de viewport D** debe definirse así:

$$D = \begin{pmatrix} \frac{w}{2} & 0 & 0 & x_l + \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y_b + \frac{h}{2} \\ 0 & 0 & \frac{f_d - n_d}{2} & n_d + \frac{f_d - n_d}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

de forma que:

$$(x_{dc}, y_{dc}, z_{dc}, 1)^T = D(x_{ndc}, y_{ndc}, z_{ndc}, 1)^T$$

La clase Viewport de Godot

La clase **Viewport** de Godot encapsula la información de un array de pixels en memoria sobre los que se pueden dibujar escenas 2D o 3D. Tiene dos sub-clases:

- Clase **Window**: el viewport ocupa **una ventana visible completa**, todo proyecto Godot tiene asociado un viewport de este tipo por defecto.
- Clase **SubViewport**: un viewport que no ocupa una ventana visible completa, hay de dos tipos:
 - ▶ El viewport **ocupa una parte de una ventana visible**, debe estar contenido en **SubViewportContainer**, que es una subclase de **CanvasItem**, es decir, es un nodo 2D.
 - ▶ El viewport es **una zona de memoria en la GPU** sobre la que se pueden visualizar imágenes, que no son visibles inmediatamente en pantalla, debe estar asignado a un objeto de la clase **ViewportTexture**, que es una subclase de **Texture2D**, es decir, una imagen de textura para 2D y 3D.

Consulta del viewport y sus propiedades

En algunos casos será necesario consultar (y modificar) las propiedades del viewport donde se está visualizando un nodo.

- Se puede recuperar el objeto **viewport** donde se está visualizando un nodo, para ello usamos el método **get_viewport** de la clase **Node**

```
var vp := nodo.get_viewport() ## 'vp' es el viewport de 'nodo'
```

- Una vez se ha obtenido el objeto **viewport** se pueden consultar sus propiedades y usar sus métodos, por ejemplo:

- ▶ Se puede acceder a su tamaño accediendo a la propiedad **size** de tipo **Vector2i**, con dos enteros que son el número de filas y de columnas de pixels del viewport.

Sección 5.
Problemas

1. Transformación de vista.
2. Transformación de proyección

Cámara que sigue un objetivo

Subsección 5.1.

Transformación de vista.

Problema 6.1:

Escribe el código GDScript para adjuntar a un nodo de tipo **Camera3D**, de forma que en cada frame la cámara apunte a un objeto móvil objetivo (por ejemplo un coche), con estos requerimientos:

- La posición y el vector de velocidad del objetivo (en coordenadas de mundo) se pueden obtener con dos funciones globales, llamadas **objetivo.posicion()** y **objetivo.velocidad()**, ambas devuelven un objeto de tipo **Vector3**.
- La cámara debe situarse detrás del objetivo, de forma que el punto devuelto por **objetivo.posicion()** se proyecte en el centro del viewport, y además la camara esté situada 3 unidades en horizontal por detrás del objetivo, y 2 unidades por encima (en el eje Y).

Parámetros para una vista concreta (1/3)

Problema 6.2:

Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja X, verde Y y azul Z), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

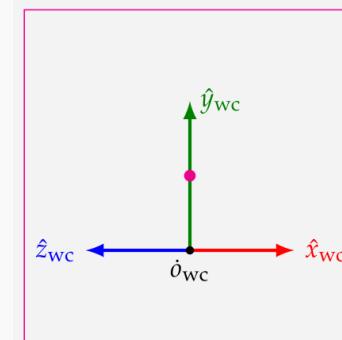
1. El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
2. El punto de coordenadas $(0, 0.5, 0)$ (aparece como un disco de color morado en la figura) debe aparecer en el centro del viewport
3. El observador (foco de la proyección) estará a 3 unidades de distancia del punto $(0, 0.5, 0)$

(continua en la siguiente transparencia).

Parámetros para una vista concreta (2/3)

Problema 6.2 (continuación):

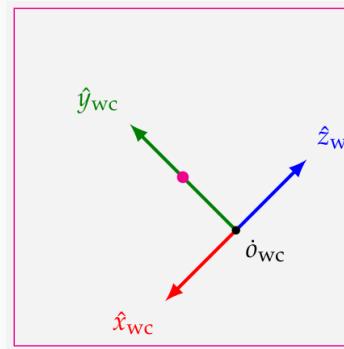
Escribe unos valores que podríamos usar para **a**, **u** y **n** de forma que se cumplan estos requisitos. En la figura se observa una vista esquemática de como quedaría la figura en un viewport cuadrado, no necesariamente a escala.



Parámetros para una vista concreta (3/3)

Problema 6.3:

Repite el problema anterior 6.2, pero ahora para esta vista:



Usa una rotación del marco de vista entorno a uno de sus propios ejes.

Construcción de la matriz de vista

Problema 6.4:

Escribe el código GDScript para calcular los vectores de coordenadas \mathbf{x}_{ec} , \mathbf{y}_{ec} , \mathbf{z}_{ec} y \mathbf{o}_{ec} que definen el marco de vista a partir de los vectores de coordenadas \mathbf{a} , \mathbf{u} y \mathbf{n} (todos estos vectores de coordenadas de mundo, en objetos de tipo **Vector3**).

Problema 6.5:

Partiendo de los vectores de coordenadas \mathbf{x}_{ec} , \mathbf{y}_{ec} , \mathbf{z}_{ec} y \mathbf{o}_{ec} que se calculan en el problema anterior, escribe el código que calcula explícitamente la matriz de vista, es una variable de tipo **Transform3D**.

Ajuste dinámico

Problema 6.6:

En una copia independiente del código de prácticas, modifica el nodo de la cámara orbital simple para conseguir que el fov mínimo (vertical u horizontal) sea siempre de 75° , serviría, por ejemplo, para ver el cubo de las prácticas siempre completo independientemente del ancho y alto de la ventana (ver transparencia 58 y siguiente). Para ello:

1. Añadir al script del nodo de cámara una función que se ejecute siempre que se redimensione la ventana (y al inicio), en esa función:
2. obtener el tamaño (alto y ancho) del viewport,
3. calcular la relación de aspecto (ancho/alto)
4. usar ajuste de la proyección en vertical si el viewport es más ancho que alto, y ajuste en horizontal en caso contrario.

Parámetros de matriz de proyección (1)

Problema 6.7:

Queremos visualizar una escena con mallas indexadas de la cual sabemos que tiene todos los vértices dentro de un cubo de lado s unidades cuyo centro es el punto de coordenadas del mundo $\mathbf{c} = (c_x, c_y, c_z)$.

Para construir la matriz de vista, se sitúa el observador en el punto $\mathbf{o}_{ec} = (c_x, c_y, c_z + s + 2)$, el punto de atención \mathbf{a} se hace igual a \mathbf{c} (el centro del cubo se ve en el centro de la imagen), y el vector \mathbf{u} es $(0, 1, 0)$. Se visualizará en un viewport cuadrado.

(continua en la siguiente página)

Parámetros de matriz de proyección (3)

Problema 6.8:

Repite el problema anterior 6.7, con exactamente los mismos requerimientos y suposiciones, excepto que ahora la escena, en lugar de estar contenida en un cubo de lado s unidades, está contenida en una esfera de radio r unidades (con centro igualmente en \mathbf{c}).

Problema 6.9:

Repite el problema anterior 6.7, con exactamente los mismos requerimientos y suposiciones, excepto que ahora, en lugar de suponer que el *viewport* es cuadrado, sabemos que tiene w columnas de pixels y h filas de pixels, y no podemos suponer que $w = h$.

Parámetros de matriz de proyección (2)

Problema 6.7 (continuación):

Queremos construir la matriz de proyección perspectiva Q de forma que se cumplan estos requerimientos:

1. No se recorta ningún triángulo.
2. El tamaño aparente de los objetos (proyectados en pantalla) es el mayor posible.
3. El valor del parámetro n es el mayor posible.
4. El valor del parámetro f es el menor posible.
5. Los objetos no aparecen deformados.

Con estos requerimientos, indica como calcular los valores l, r, t, b, n y f (para obtener la matriz Q de proyección), en función de s y $\mathbf{c} = (c_x, c_y, c_z)$.

Parámetros de matriz de proyección (4)

Problema 6.10:

Repite el problema anterior 6.7, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo β en grados que debe ser la apertura de campo vertical de la proyección perspectiva. Para ello, ahora tenemos libertad para situar al observador en la línea paralela al eje Z que pasa por \mathbf{c} , de forma que la apertura de campo vertical sea exactamente β .

Indica como calcular la coordenada Z que debemos usar ahora para \mathbf{o}_{ec} (la X y la Y son las mismas que antes), de forma que se cumpla lo dicho, también indica como debemos de calcular ahora los valores de l, r, t, b, n y f (todo ello en función de β, s y $\mathbf{c} = (c_x, c_y, c_z)$).

Fin de transparencias.

Informática Gráfica.

Sesión 7: Modelos de iluminación.

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

Índice

Radiación: percepción, emisión y reflexión	3
El modelo de <i>sombreado de Phong</i>	20
Modelos realistas de iluminación. La BRDF	49
Modelos de fuentes de luz	86
Problemas	99

Sección 1.

Radiación: percepción, emisión y reflexión.

1. Radiación y su percepción: el color.
2. Emisión y reflexión de la radiación.

Subsección 1.1.

Radiación y su percepción: el color.

La luz como radiación electromagnética

La luz que observamos es radiación electromagnética (variaciones periódicas del campo eléctrico y magnético) de naturaleza similar a las ondas que se usan para los móviles, wifi, radio y televisión:

- El sistema visual humano ha evolucionado para percibir esa radiación solo cuando su longitud de onda λ está aprox. entre 390 y 750 nanómetros (\equiv *espectro visible*).
- La emisión e interacción de las ondas en los átomos nos permite percibir el entorno.
- Físicamente, la radiación se describen como algo que tiene características de onda y de corpúsculo a la vez (modelos complementarios).
- En Informática Gráfica se usa más frecuentemente el *modelo de partículas* (óptica geométrica) en lugar del *modelo de ondas* (óptica física).

Brillo y color de la radiancia

Desde un punto p en una dirección v pueden emitirse o reflejarse una gran cantidad de fotones con longitudes de onda distintas.

- La intensidad o brillo de la luz depende de la cantidad de fotones (es decir, de la radiancia total sumada en todas las longitudes de onda)
- El color con el que percibimos la luz depende de la distribución de las longitudes de onda de los fotones en el espectro visible.

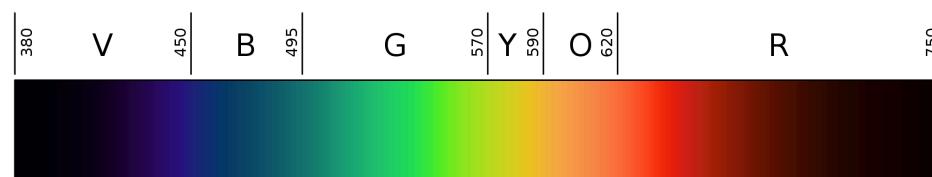


Figura de Wikipedia (Visible Spectrum): en.wikipedia.org/wiki/Visible_spectrum

El modelo de partículas. La radiancia.

Bajo este modelo, la radiación se puede describir de forma idealizada como un flujo en el espacio de partículas puntuales llamadas **fotones**, con trayectorias rectilíneas.

- Cada uno tiene una *energía radiante* que depende únicamente de su longitud de onda (es inv. prop.)
- En un entorno de punto p del espacio (típicamente en la superficie de un objeto) podemos medir la densidad de energía radiante por unid. de tiempo de los fotones de una longitud de onda λ que pasan por p en una determinada dirección v (un vector libre).

Esa energía se denomina **radiancia** y se nota como $L(\lambda, p, v)$.

La radiancia determina el tono de color y el brillo con el que observamos el punto p cuando lo vemos desde la dirección v .

Percepción de radiación visible

El ojo es la parte del *sistema visual humano* (SVH) capaz de enviar señales eléctricas al cerebro que dependen de las características de la luz que incide sobre las neuronas de su cara interna (la retina)

- En cada neurona de la retina, y para cada longitud de onda λ , se recibe una radiancia $L(\lambda)$ distinta.
- El ojo funciona de forma tal que *simplifica* esa gran cantidad de información y la reduce (en cada neurona) a tres valores reales positivos que forman una tupla (s, m, l) que depende de L , es decir, el ojo tiene asociada una función f tal que:

$$f(L) = (s, m, l)$$

- Esta simplificación es aprox. lineal, es decir si $f(L) = (s, m, l)$ y $f(L') = (s', m', l')$, entonces:

$$f(aL + bL') = a(s, m, l) + b(s', m', l')$$

donde a, b son valores reales arbitrarios, no negativos.

Los primarios RGB.

Si x es un valor real ($x > 0$), entonces:

- la señal $(x, 0, 0)$ enviada desde el ojo se interpreta o percibe en el cerebro (SVH) como de color rojo.
- la señal $(0, x, 0)$ se percibe de color verde.
- la señal $(0, 0, x)$ se percibe de color azul.

Como consecuencia, supongamos que tenemos tres distribuciones de radiancia

L_r, L_g y L_b tales que:

$$f(L_r) \approx (1, 0, 0) \quad f(L_g) \approx (0, 1, 0) \quad f(L_b) \approx (0, 0, 1)$$

A una terna de distribuciones L_r, L_g y L_b que cumplen lo anterior se le denomina una terna de **primarios RGB**, ya que son percibidos como rojo, verde y azul, respectivamente.

Mezcla aditiva de primarios

Podemos usar una *mezcla aditiva* (suma ponderada) de tres primarios RGB para producir una señal arbitraria (r, g, b) en el ojo, ya que se cumple:

$$f(rL_r + gL_g + bL_b) \approx (r, g, b)$$



Imagen obtenida de: en.wikipedia.org/wiki/RGB_color_model

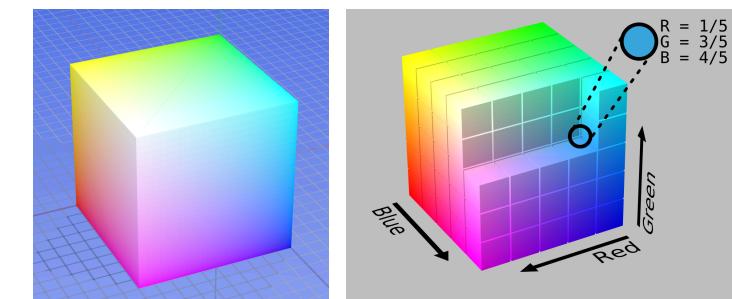
Reproducción de ternas RGB

Un dispositivo de salida de color (monitor, impresora, proyector) tiene asociados tres primarios RGB (las distribuciones obtenidas cuando se muestra el rojo, verde y azul a máxima potencia en el dispositivo)

- Como consecuencia, cualquier color reproducible en un dispositivo se puede representar por una terna (r, g, b) , con $0 \leq r, g, b \leq 1$.
- El valor 0 indica que el correspondiente primario no aparece.
- El valor 1 representa la máxima potencia del dispositivo para cada primario.
- Una misma terna (r, g, b) produce tonos de color ligeramente distintos en dispositivos distintos.
- Una misma terna (r, g, b) niveles de brillo que pueden variar mucho entre dispositivos.

El espacio RGB

Al conjunto de todas las ternas RGB con componentes entre 0 y 1 se le llama **espacio de color RGB**, y se puede visualizar como un cubo 3D con colores asociados a cada punto del mismo.



Imágenes de Wikipedia: *RGB color model* en.wikipedia.org/wiki/RGB_color_model

El espacio RGB no es el único esquema para representar computacionalmente los colores, pero sí el más usado hoy en día.

Dependencia del dispositivo de salida

El color que se obtiene con una terna RGB en un dispositivo de salida depende de los primarios RGB que se usen en dicho dispositivo y del brillo máximo que pueda alcanzar:



Imagen de: [sitio web de CBC News, Canadá](#)

Sesión 7: Modelos de iluminación

Created 2025-12-01

Page 13 / 103.

1. Radiación: percepción, emisión y reflexión..
1.2. Emisión y reflexión de la radiación..

Fuentes de luz y reflectores:

La radiación electromagnética visible se genera en las **fuentes de luz**, por procesos físicos diversos que convierten otras formas de energía en energía radiante. Hay de dos tipos:

- **Fuentes naturales:** Sol o estrellas, fuego, objetos incandescentes, órganos de algunos animales, etc...
- **Fuentes artificiales (luminarias):** filamentos incandescentes, tubos fluorescentes, LEDs, etc...

Los fotones creados en las luminarias interactúan con los átomos de la materia, que absorben su energía y después pueden radiar de nuevo una parte de ella, proceso conocido como **reflexión**:

- parte de la energía recibida se **convierte en calor**
- parte de la energía recibida se **convierte en radiación reflejada**
- la radiación reflejada puede reflejarse de nuevo varias veces

Subsección 1.2. Emisión y reflexión de la radiación.

1. Radiación: percepción, emisión y reflexión..
1.2. Emisión y reflexión de la radiación..

Modelo de la reflexión local en un punto

La radiancia $L(\lambda, p, v)$ se puede escribir como suma de:

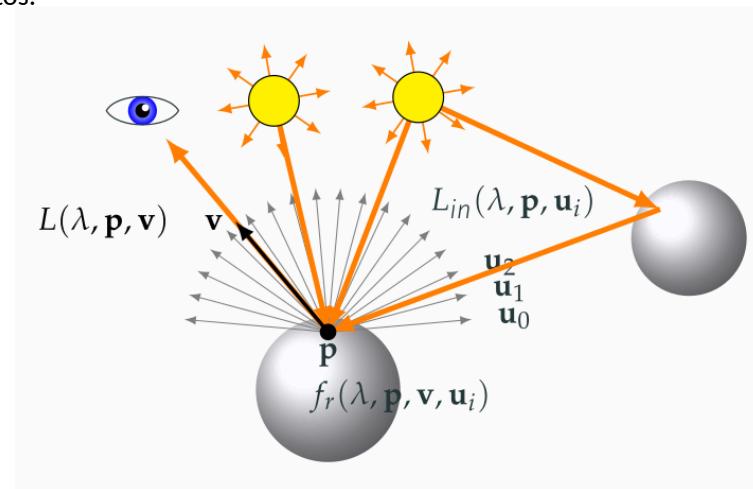
- la **radiancia emitida** desde p en la dirección v (0 si p no está en una fuente de luz), que llamamos $L_{\text{em}}(\lambda, p, v)$
- la **radiancia reflejada**, suma, para cada dirección u_i del producto de:
 - ▶ $L_{\text{in}}(\lambda, p, u_i) \equiv$ radiancia incidente sobre p desde u_i
 - ▶ $f_r(\lambda, p, v, u_i) \equiv$ fracción de radiancia que se refleja desde p en la dirección v , respecto del total incidente sobre p proveniente de la dirección u_i (con longitud de onda λ)

es decir, la radiancia es sume de emitida más reflejada, y la reflejada es una suma ponderada de la incidente desde cada dirección multiplicada por la fracción reflejada:

$$L(\lambda, p, v) = L_{\text{em}}(\lambda, p, v) + \sum_i f_r(\lambda, p, v, u_i) L_{\text{in}}(\lambda, p, u_i)$$

Reflexión local en un punto

Hay muchas trayectorias de fotones que no acaban siendo detectadas por el observador (la mayoría), además las que sí llegan pueden hacerlo por muchos caminos distintos:



Características de la ecuación de rendering

La ecuación de rendering tiene las siguientes características:

- La función L aparece a la izquierda y a la derecha (dentro de una integral).
- Por tanto, es una **ecuación integral**.
- Si conocemos las funciones L_{em} y f_r , podemos interpretar la función L como la *incógnita* de la ecuación, ya que L **está determinada** por f_r y L_{em} .
- No hay una expresión analítica cerrada general para escribir L en función de f_r y L_{em} .

Por tanto, **su resolución analítica** es imposible, así que hay dos estrategias para calcular L :

- Usar **modelos simplificados** de reflexión y emisión que permitan calcular L analíticamente a partir de f_r y L_{em} (a costa de realismo reducido).
- Usar **métodos numéricos** para calcular una aproximación a L .

La Ecuación de Rendering

La sumatoria anterior es incorrecta, ya que el espacio de posibles direcciones de entrada \mathbf{u} es continua (es la semiesfera de radio unidad, Ω), por tanto, si queremos ser rigurosos no se puede usar una sumatoria sino una integral, en la cual el diferencial es $(\mathbf{n} \cdot \mathbf{u})d\sigma(\mathbf{u})$

$$L(\lambda, p, v) = L_{em}(\lambda, p, v) + \int_{\Omega} L_{in}(\lambda, p, \mathbf{u}) f_r(\lambda, p, v, \mathbf{u}) (\mathbf{n} \cdot \mathbf{u})d\sigma(\mathbf{u})$$

En esta ecuación, el valor de L_{in} en realidad es la radiancia saliente desde otro punto q en la dirección contraria $-\mathbf{u}$. Sustituimos $L_{in}(\lambda, p, \mathbf{u})$ por $L(\lambda, q, -\mathbf{u})$:

La ecuación resultante se denomina **ecuación de rendering (rendering equation)**, tiene esta forma:

$$L(\lambda, p, v) = L_{em}(\lambda, p, v) + \int_{\Omega} L(\lambda, q, -\mathbf{u}) f_r(\lambda, p, v, \mathbf{u}) (\mathbf{n} \cdot \mathbf{u})d\sigma(\mathbf{u}) \quad (1)$$

El modelo de sombreado de Phong.

1. Componente ambiental.
2. Componente difusa.
3. Componente pseudo-especular.

Simplificaciones para modelos básicos

La ecuación de rendering es muy compleja de aproximar en corto tiempo de cálculo. Por tanto, se hacen varias simplificaciones:

1. No se considera la radiancia emitida.
2. No se considera la luz incidente que no provenga directamente de las fuentes de luz.
3. Las fuentes de luz son puntuales o unidireccionales, no extensas, y hay un número finito de ellas.
4. Los objetos o polígonos son totalmente opacos (no hay transparencias ni mat. translúcidos).
5. No se consideran sombras arrojadas (las fuentes son visibles desde cualquier cara delantera respecto de ellas).
6. El espacio entre los objetos no dispersa la luz (la radiancia se conserva en el espacio entre los objetos).
7. En lugar de considerar todas las longitudes de onda λ posibles, usamos el modelo RGB.

Modelo simplificado

El modelo que hemos visto antes en la ecuación (1) se simplifica:

- La iluminación indirecta se reduce a un término ambiente (no depende de v).
- De todas las direcciones u_i , solo es necesario considerar las que apuntan hacia cada una de las fuentes de luz l_i
- Todas las fuentes de luz son visibles desde un punto (no hay sombras arrojadas).
- Los valores de radiancia son tuplas (r, g, b) (no acotadas)
- Los valores de reflectividad (f_r) son tuplas (r, g, b) (entre 0 y 1)

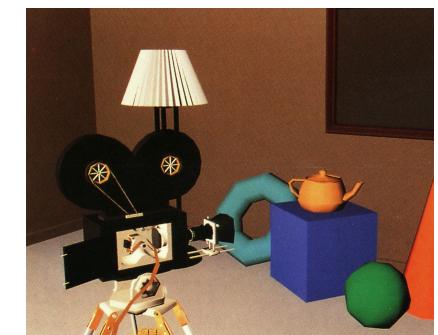
Por tanto ahora la radiancia saliente L se obtiene así:

$$L(\mathbf{p}, \mathbf{v}) = \sum_{i=0}^{n-1} L_{in}(\mathbf{p}, \mathbf{l}_i) f_r(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) \quad (2)$$

donde: $n \equiv$ número de fuentes de luz, $\mathbf{l}_i \equiv$ vector que apunta desde \mathbf{p} en la dirección de la i -ésima fuente de luz.

Efecto de las simplificaciones.

Aquí se observa una escena con iluminación compleja (izquierda) y simplificada (derecha)

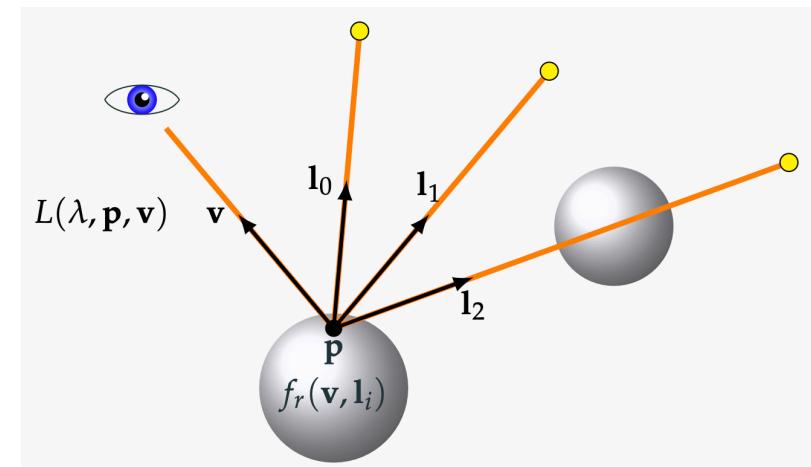


Imágenes de:

[Foley, van Dam, Feiner, Hughes Computer Graphics: Principles and Practice in C \(2nd ed.\)](#)

Modelo simplificado (figura)

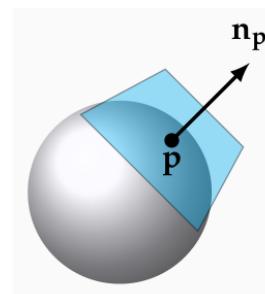
Ahora solo consideraremos trayectorias desde las luminarias hacia \mathbf{p} , las luminarias se cuentan aunque la trayectoria esté bloqueada (no hay sombras arrojadas)



El vector normal

La iluminación (la función f_r en la ecuación 2) depende la orientación de la superficie en el punto p . Esta orientación esta caracterizada por el **vector normal** n_p asociado a dicho punto:

- El vector normal n_p (en azul en la fig.) indica la orientación de la superficie en el punto p .
- Es de longitud unidad y depende de p .
- Idealmente es perpendicular al plano tangente a la superficie en el punto p (en azul en la fig.)
- En modelos de fronteras, puede calcularse de varias formas (depende del *método de sombreado*, que veremos más adelante).
- Constituye un parámetro de f_r



Tipos y atributos de las fuentes de luz

En el modelo de escena se puede incluir un conjunto de n fuentes de luz (numeradas de 0 a $n - 1$), cada una de ellas puede ser de **dos tipos**:

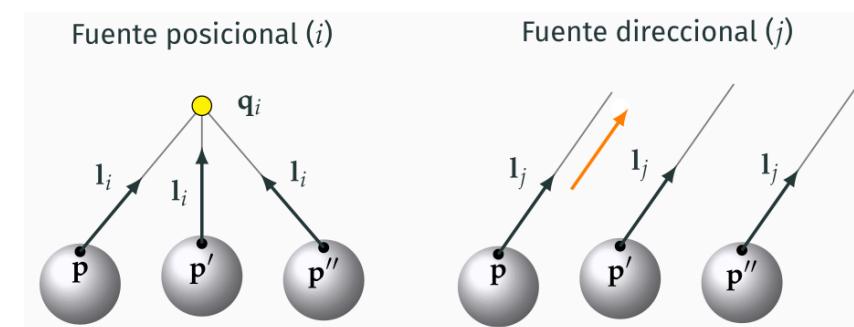
Fuentes de luz posicionales: ocupan un punto del espacio q_i . Dado un punto p , el vector unitario que apunta hacia la fuente de luz desde p se calcula como:

$$\mathbf{l}_i = \frac{\mathbf{q}_i - \mathbf{p}}{\|\mathbf{q}_i - \mathbf{p}\|}$$

Fuentes de luz direccionales: están en un punto a distancia infinita, por tanto hay un vector \mathbf{l}_j que apunta a la fuente y que es el mismo para cualquier punto p donde se quiera evaluar el MIL

Además de esto, **cada fuente de luz emite una radiancia** $S_i = (r, g, b)$ (en general no acotada).

Posición o dirección de las luminarias



Posicional: la dirección \mathbf{l}_i es distinta para cada punto p considerado. Es necesario recalcularla cada vez que se evalua el MIL.

Direccional: La dirección \mathbf{l}_j es igual para todos los puntos p considerados. Es una constante.

Radiancia incidente y tipos de reflexión. Componentes del MIL.

En la ecuación (2) los términos que aparecen pueden reescribirse en términos de los atributos de las fuentes de luz y el material

- El término $L_{in}(\mathbf{p}, \mathbf{l}_i)$ se hace igual a S_i (no tenemos en cuenta la distancia a la que está la fuente de luz)
- El término $f_r(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$ se descompone en tres sumandos o **componentes**
 - ▶ Luz indirecta reflejada, o término **ambiental**: $f_{am}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$.
 - ▶ Luz reflejada de forma **difusa**: $f_{dl}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$.
 - ▶ Luz reflejada de forma **pseudo-especular**: $f_{ph}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$.

La ecuación (2) queda como sigue:

$$L(\mathbf{p}, \mathbf{v}) = \sum_{i=0}^{n-1} S_i [f_{am}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{dl}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{ph}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)] \quad (3)$$

Color base del objeto en un punto

En cada punto p de la superficie de un objeto hay una terna RGB, que llamamos $C(p)$, con valores entre 0 y 1, que es el **color base** en el punto p .

- Para cada componente RGB, expresa la famcción de luz reflejada, y por tanto determina el color con el que apreciamos el objeto.
- Puede ser el mismo (constante) en todos los puntos p de la superficie de un objeto.
- Puede variar de un punto a otro dentro del mismo objeto. En rasterización, esto puede ocurrir de dos formas:
 - ▶ Por el uso de *texturas* (las veremos más adelante).
 - ▶ Por el uso de una tabla de colores como atributos de vértice (para cada punto p , su color $C(p)$ es el color RGB interpolado).
- El color del objeto afecta únicamente a las componentes ambiental y difusa (no a la componente especular).

Subsección 2.1.
Componente ambiental.

Componente ambiental.

Cada objeto puede reflejar más o menos cantidad de iluminación indirecta proveniente de la i -ésima fuente de luz.

- Esa iluminación indirecta es complicada de calcular y se ignora en este modelo.
- Por tanto, los objetos aparecerían negros donde no haya iluminación directa.
- Para suplir esto, se usa la **componente ambiental** f_{am} .

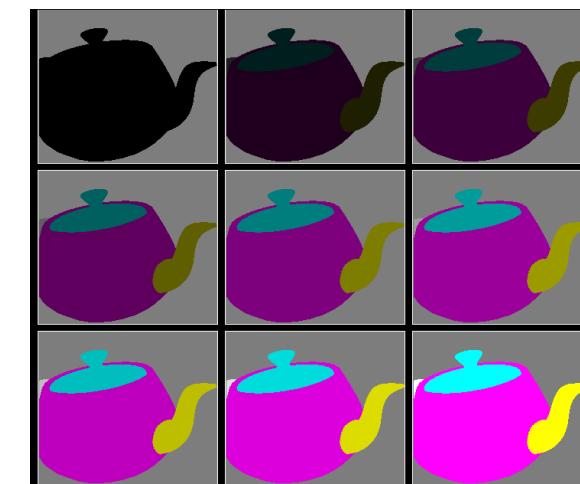
La componente ambiental, por tanto, no depende de v ni l_i , y se hace igual a:

$$f_{\text{am}}(p, v, l_i) = k_{\text{am}}(p) \cdot C(p), \quad (4)$$

donde $k_{\text{am}}(p)$ es un valor real entre 0 y 1 que determina la famcción de luz reflejada de esta forma

Reflectividad ambiental del objeto:

En este ejemplo, el color $C(p)$ depende de la parte de la tetera donde está p , mientras que k_a es constante en toda la tetera (aunque crece en sucesivas imágenes).



Componente difusa: expresión.

La **componente difusa o lambertiana (lambertian)** modela como se refleja la luz en los objetos mate o difusos *ideales*. El nombre *lambertiano* proviene de J.H. Lambert, científico francés del siglo XVIII, que fue el primero en describir este tipo de reflexión.

- La componente **sí depende** de la posición u orientación de la fuente de luz (es distinta según como esté orientada la fuente respecto de la superficie en p , es decir, depende de n_p y l_i),
- **no depende** de la dirección v en la que miramos p (el punto p se ve de un color igual desde cualquier dirección que lo veamos).

La expresión concreta de f_{dl} es esta:

$$f_{dl}(p, v, l_i) = k_{dl}(p) \cdot C(p) \cdot \max(0, n_p \cdot l_i), \quad (5)$$

donde $k_{dl}(p)$ es un valor entre 0 y 1 que indica la **famcción de luz reflejada de forma difusa**.

Subsección 2.2.

Componente difusa.

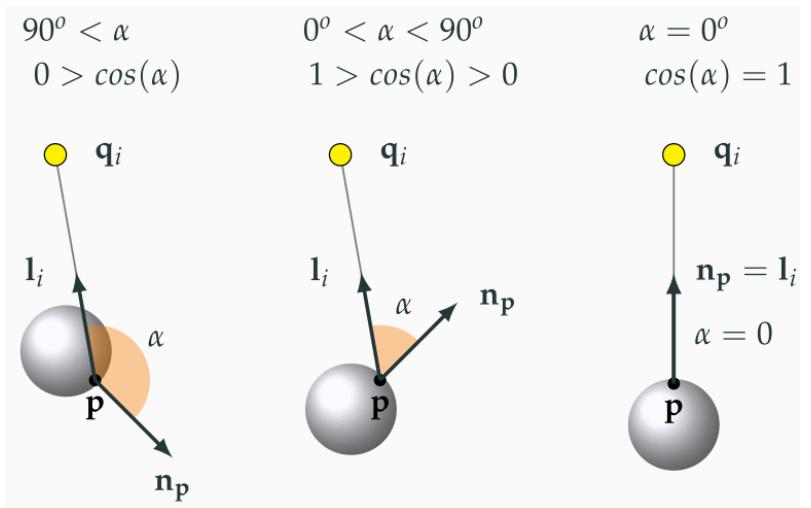
Orientación de la superficie

La orientación de la superficie respecto de la fuente de luz viene determinada por el valor α , que es el ángulo que hay entre los vectores n_p y l_i (el valor $n_p \cdot l_i$ es igual al coseno de α). Se pueden distinguir dos casos:

- Si $\alpha > 90^\circ$, entonces:
 - ▶ $\cos(\alpha)$ es negativo.
 - ▶ la superficie, en p , está orientada de espaldas a la fuente de luz.
 - ▶ la contribución de esa fuente debe ser 0.
- Si $0^\circ \leq \alpha \leq 90^\circ$, entonces:
 - ▶ la superficie, en p , está orientada de cara a la fuente de luz.
 - ▶ $\cos(\alpha)$ estará entre 0 y 1 (entre $\cos(90^\circ)$ y $\cos(0^\circ)$).
 - ▶ se puede demostrar que el valor $\cos(\alpha)$ es proporcional a la densidad de fotones por unidad de área que inciden en el entorno de p , provenientes de la i -ésima fuente de luz.

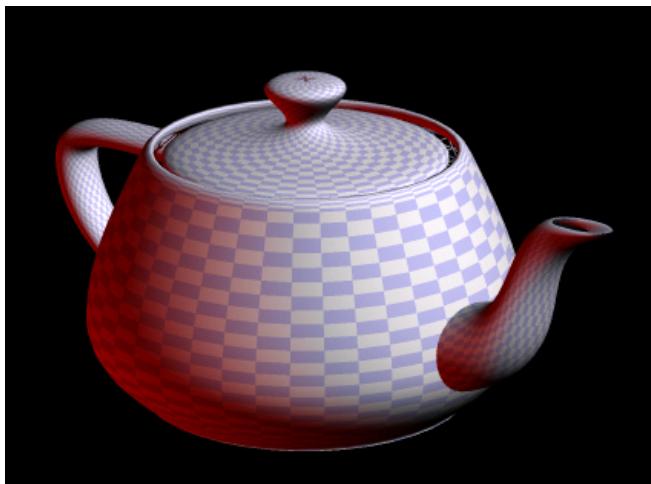
Orientación de la superficie (2)

Aquí se ilustran tres posibles casos:



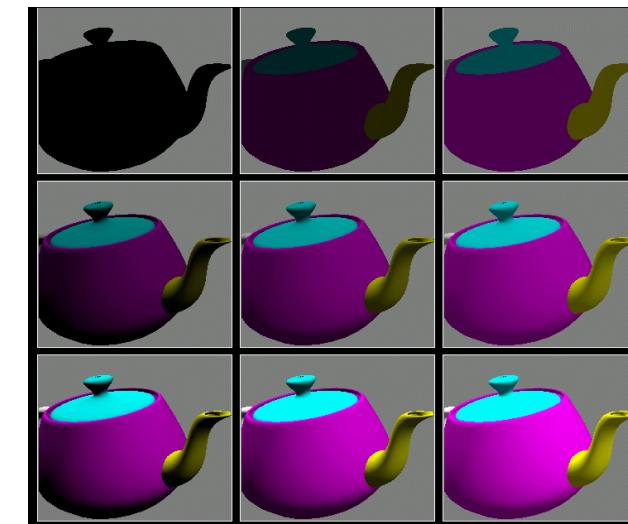
Material difuso

Ejemplo con dos fuentes de luz direccional, $k_{\text{am}}(\mathbf{p}) = 0$ y $k_{\text{dl}}(\mathbf{p}) = 1$ (solo hay componente difusa). Además, $C(\mathbf{p})$ varía de unos polígonos a otros:



Material difuso + ambiental

Aquí k_{am} crece de izquierda a derecha, y k_{dl} de arriba abajo:



Componente pseudo-especular. Modelo de Phong

La componente **pseudo-especular** o componente de **Phong** modela como se refleja la luz en los objetos brillantes, en los cuales dichas zonas brillantes dependen de la posición del observador:

- La componente **sí depende** de la posición u orientación de la fuente de luz (es distinta según como esté orientada la fuente respecto de la superficie en \mathbf{p}),
- **también depende** de la dirección en la que miramos \mathbf{p} (el punto \mathbf{p} se ve de un color diferente según la dirección en la que lo veamos).

La expresión ideada en 1973 por *Bui Tuong Phong* (*Bùi Tường Phong*), y conocida como **modelo de Phong** es esta:

$$f_{\text{ph}}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = k_{\text{ph}}(\mathbf{p}) d_i [\max(0, \mathbf{r}_i \cdot \mathbf{v})]^e , \quad (6)$$

donde $k_{\text{ph}}(\mathbf{p})$ es un valor real entre 0 y 1, representa la fracción de luz reflejada de forma pseudo-especular.

Parámetros y valor del modelo de Phong

En la expresión anterior:

r_i ≡ vector reflejado:

depende tanto de l_i como de n_p , y está en el plano formado por ambos, con n_p como bisectriz de ellos, se obtiene como:

$$r_i = 2(l_i \cdot n_p)n_p - l_i$$

el vector r_i indica la dirección desde p en la cual la i -ésima fuente de luz produce el máximo brillo.

e ≡ exponente de brillo:

un valor real positivo que permite variar el tamaño de las zonas brillantes (a mayor valor, menor tamaño y más pulida o especular).

d_i ≡ indicador de orientación:

vale 1 si $n_p \cdot l_i > 0$ (fuente de cara a la superficie), y 0 en otro caso (de espaldas).

Comp. pseudo-especular: modelo de Blinn-Phong

Una alternativa al modelo anterior consiste en usar el vector *halfway* h_i (bisectriz de l_i y v , normalizado).

- El modelo resultante se llama **modelo de Blinn-Phong** ya que esta modificación fue propuesta en 1977 por James F. Blinn.
- Ahora el brillo es proporcional al coseno del ángulo γ entre h_i y n_p (máximo cuando coinciden).

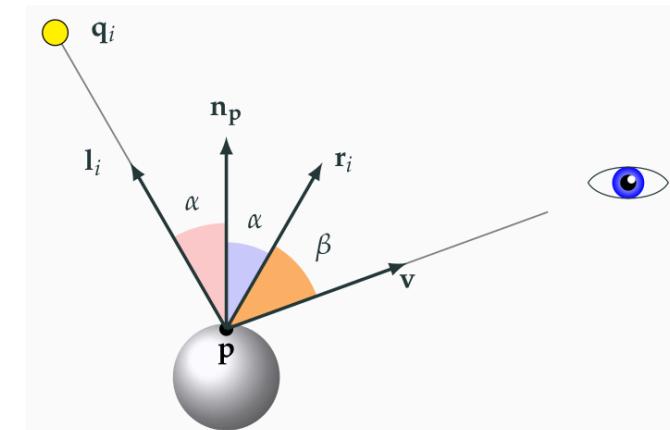
La expresión del Modelo de Blinn-Phong es la siguiente:

$$f_{bp}(p, v, l_i) = k_{bp}(p) d_i [n_p \cdot h_i]^e \quad (7)$$

Esta variante es más comúnmente usada que el modelo de Phong anterior. El vector h además, se usa en otros modelo de iluminación más avanzados.

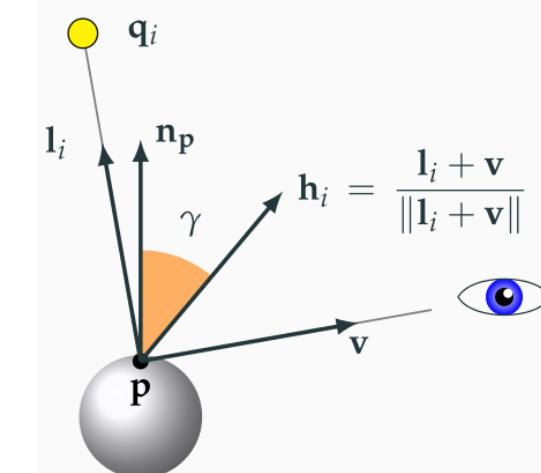
Vectores del modelo de Phong

El valor $r_i \cdot v$ es el coseno del ángulo β que hay entre la dirección de máximo brillo r_i y la dirección v hacia el observador. Cuando $r_i = v$ entonces $\beta = 0^\circ$, $\cos(\beta) = 1$, y el brillo es máximo:



Vectores de la componente de Blinn-Phong

El vector h_i es la bisectriz entre l_i y v , es decir, es el vector $l_i + v$, normalizado:



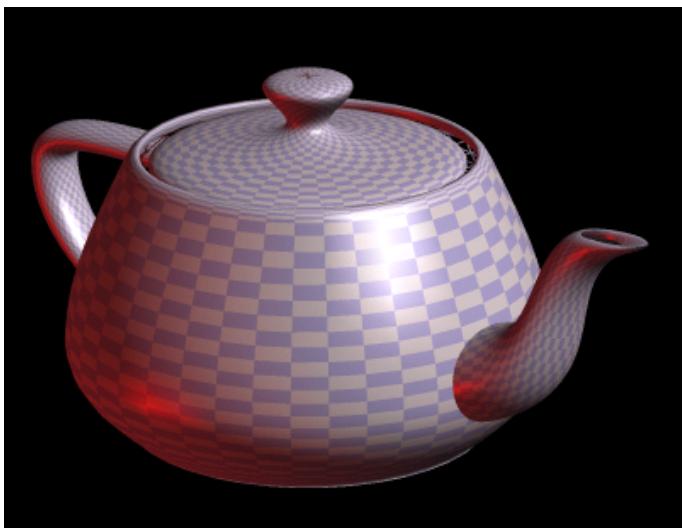
Ejemplo de material pseudo-especular

Aquí $k_a(\mathbf{p}) = 0$, $k_d(\mathbf{p}) = 0$, $k_s(\mathbf{p}) = 1$ y $e = 5.0$:



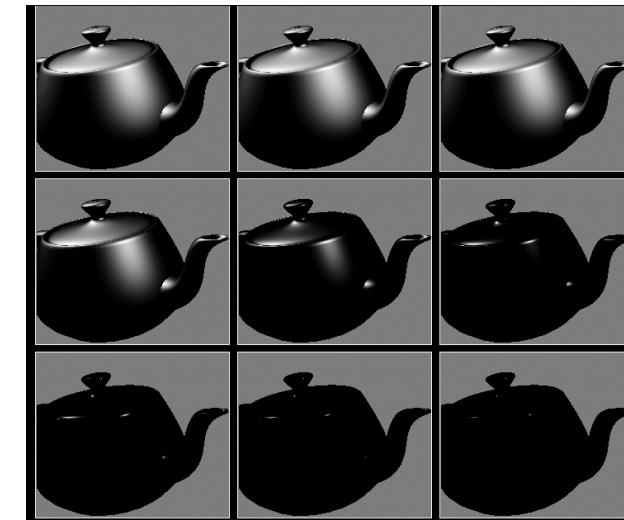
Ejemplo de material combinado

Combinación ambiental, más difusa, más pseudo especular:



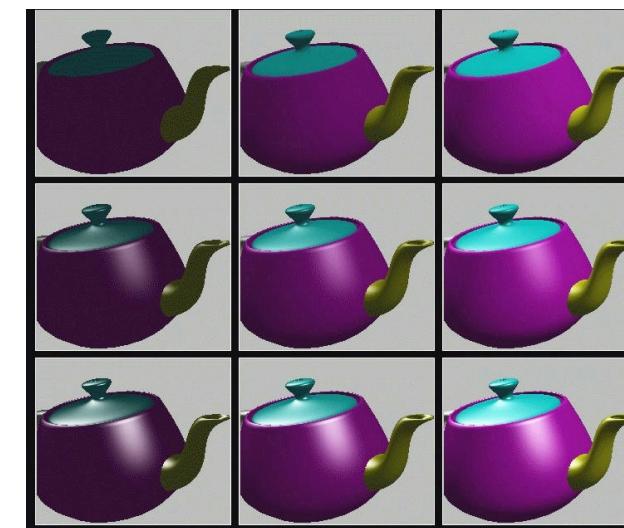
Efecto del exponente de brillo

Aquí el exponente e crece de izquierda a derecha y de arriba abajo:



Combinaciones material difuso + pseudo especular

Aquí k_d crece de izquierda a derecha y k_s de arriba abajo:



Modelos realistas

Aquí se describen otros modelos iluminación local, más realistas pero algo más costosos en tiempo. Se han diseñado teniendo en cuenta varios factores:

- Las **leyes físicas** que determinan como se produce la **reflexión y la refacción**.
- El comportamiento reflectivo de un **catálogo de materiales reales**, medido con dispositivos específicos y hecho público (MERL database)
- El uso de **parámetros intuitivos** adaptados a las necesidades expresivas de los diseñadores gráficos.

Destacamos estos modelos:

- Modelo de reflexión especular y refacción basado en las **Leyes de Fresnel**, para materiales dieléctricos.
- Modelo de reflexión difusa de **Burley**, como alternativa al modelo sencillo de Lambert.
- Modelo de reflexión en superficies de microfacetas, (modelo **GGX**), con la simplificación de **Schlick**, como alternativa al modelo de Phong o Blinn-Phong.

Sección 3.

Modelos realistas de iluminación. La BRDF.

1. Reflexión especular perfecta.
2. Reflexión especular y refracción simultáneas. Leyes de Fresnel.
3. Reflexión difusa. Modelos Lambertianos y de Burley.
4. Reflexión pseudo-especular. Modelos de microfacetas. GGX.
5. Combinaciones de BRDFs: suma ponderada y capas.

La Función Bidireccional de Distribución de la Reflectancia (BRDF)

En los modelos avanzados de iluminación local, la función de reflectividad f_r se define como una **BRDF (Bidirectional Reflectance Distribution Function)**.

- La BRDF es una función que relaciona la luz reflejada en un punto \dot{x} hacia una dirección de salida \mathbf{w}_o con la luz incidente desde una dirección \mathbf{w}_i .
- Por tanto, la BRDF depende únicamente de dos direcciones (y del punto \dot{x}).
- La BRDF se define como el cociente entre la radiancia reflejada L_r en la dirección \mathbf{w}_o y la irradiancia E incidente desde la dirección \mathbf{w}_i :

$$f_r(\dot{x}, \mathbf{w}_o, \mathbf{w}_i) = \frac{dL_r(\dot{x}, \mathbf{w}_o)}{dE(\dot{x}, \mathbf{w}_i)}$$

- La unidad de la BRDF es sr^{-1} (estereoradianes a la menos uno).
- La BRDF cumple el principio de conservación de la energía: **no puede haber más energía reflejada que la energía incidente**.

El marco de referencia local

Las expresiones de las BRDFs se suelen simplificar cuando se usan las coordenadas de los diversos vectores en el llamado **marco de referencia local** (ortonormal) asociado al punto \dot{x} , dicho marco:

- Está alineado con la normal $\mathbf{n}_{\dot{x}}$ a la superficie en \dot{x} , en concreto, el eje Z es paralelo a $\mathbf{n}_{\dot{x}}$.
- Los ejes X e Y son perpendiculares entre sí y perpendiculares a $\mathbf{n}_{\dot{x}}$. Es decir, son dos versores $\hat{\mathbf{t}}_x$ y $\hat{\mathbf{t}}_y$ **tangentes** a la superficie en \dot{x} . En principio es indiferente cuales sean esos dos ejes (siempre que sean perpendiculares entre ellos y a \mathbf{n}), pero en el caso de materiales **anisotrópicos** la orientación es relevante.
- Por tanto, consideraremos que \mathbf{w}_i , \mathbf{w}_o (y otros vectores que aparecerán), son en realidad tuplas de coordenadas relativas a este marco.

Reflexión especular perfecta

La reflexión especular perfecta es la que ocurre en los espejos o superficies metálicas pulidas perfectamente.



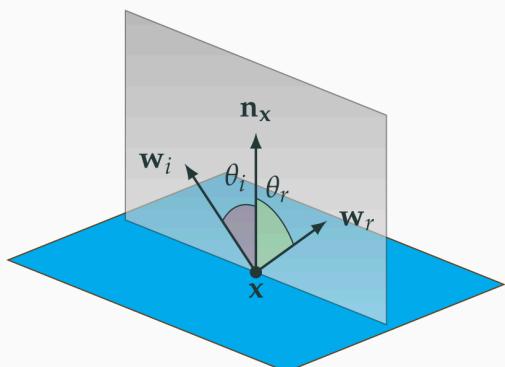
Imagen de la izquierda obtenida de: www.sphere-ball.com (Perfect mirror sphere)

Subsección 3.1.

Reflexión especular perfecta.

Vectores en la reflexión especular perfecta

En las superficies **especulares perfectas** (como un espejo) radiancia reflejada en un punto \dot{x} hacia una dirección cualquiera \mathbf{w}_r , se debe únicamente a la radiancia indicante sobre \dot{x} desde la dirección simétrica \mathbf{w}_i :



$$\frac{\mathbf{w}_i + \mathbf{w}_r}{\|\mathbf{w}_i + \mathbf{w}_r\|} = \mathbf{n}_{\dot{x}}$$

$$\cos \theta_i = \mathbf{w}_i \cdot \mathbf{n}_{\dot{x}}$$

$$\cos \theta_r = \mathbf{w}_r \cdot \mathbf{n}_{\dot{x}}$$

El ángulo de reflexión (θ_r), coincide con el de incidencia (θ_i) y el vector \mathbf{w}_i se encuentra en el plano definido por \mathbf{w}_r y $\mathbf{n}_{\dot{x}}$

Reflexión especular perfecta

La radiancia reflejada de forma especular perfecta no puede expresarse en términos de una expresión para la BRDF (tendría valores de densidad infinitos), así que se expresa directamente L_r en una dirección de salida \mathbf{w}_r en términos de L_{in} en la dirección simétrica \mathbf{w}_i , es decir:

$$L_r(\dot{x}, \mathbf{w}_r) = k_{ps}(\dot{x}) L_i(\mathbf{w}_i, \dot{x})$$

- El vector \mathbf{w}_i se calcula a partir de $\mathbf{n}_{\dot{x}}$ y \mathbf{w}_r :

$$\mathbf{w}_i = 2(\mathbf{w}_r \cdot \mathbf{n}_{\dot{x}})\mathbf{n}_{\dot{x}} - \mathbf{w}_r$$

- El valor $k_{ps}(\dot{x})$ estará entre 0 y 1, es la fracción de luz reflejada de esta forma. Si vale 0 no hay este tipo de reflexión, si vale 1 toda la reflexión es de esta forma.

Reflexión y refracción en medios dieléctricos

En el interfaz entre un medio dieléctrico (líquidos o cristales, por ejemplo) y otro (p.ej. el vacío) se produce a la vez **reflexión especular perfecta** y **refracción**. Aquí vemos la refracción en una bola de cristal sólida.



Subsección 3.2.

Reflexión especular y refracción simultáneas. Leyes de Fresnel.

3. Modelos realistas de iluminación. La BRDF.
3.2. Reflexión especular y refracción simultáneas. Leyes de Fresnel..

Reflexión y refracción en función del ángulo

En muchos casos, la reflexión y la refracción se presentan simultáneamente.

Cuando el **ángulo de incidencia** es grande (casi paralelo a la superficie) **predomina la reflexión**.

Si el ángulo es pequeño (perpendicularmente a la superficie), **predomina la refracción**:

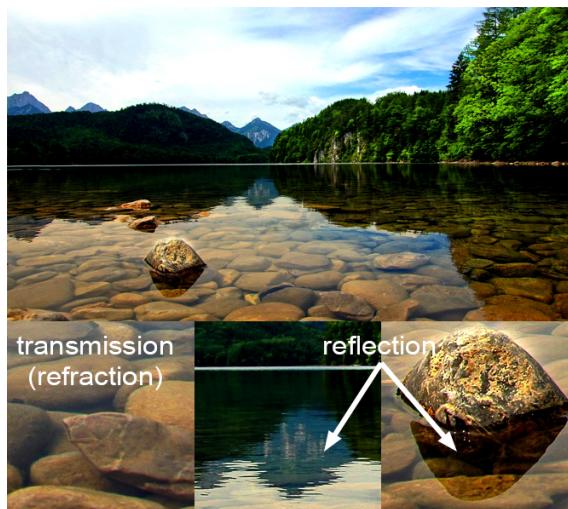


Imagen de: www.scratchapixel.com (Shading: Fresnel reflection and refraction.)

Imágenes: www.scratchapixel.com : reflection and refraction

Sesión 7: Modelos de iluminación

Created 2025-12-01

Page 58 / 103.

3. Modelos realistas de iluminación. La BRDF.
3.2. Reflexión especular y refracción simultáneas. Leyes de Fresnel..

Refracción y ancho del haz

La refracción se debe a la distinta velocidad de propagación (*velocidad de fase*) de la luz en los distintos medios.

- Los **índices de refracción** n_i y n_t son valores inversamente proporcionales a dicha velocidad en el medio de incidencia y en el medio de refracción.
- La sección del haz de luz cambia al refractarse (y por tanto cambia la radiancia).

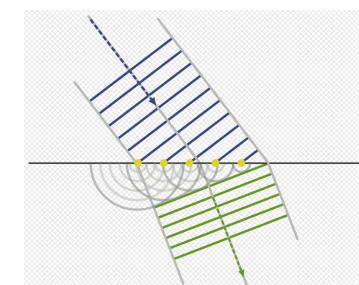


Imagen: [Anne Nordman \(wikipedia: Huygens-Fresnel principle\)](https://en.wikipedia.org/wiki/Huygens-Fresnel_principle)

Haz reflejado y haz refractado

La energía total del haz reflejado más el refractado coincide con la del haz incidente (se conserva la energía). Aquí vemos un haz que sufre refracción y reflexión dos veces en el interfaz entre un cristal y el aire:

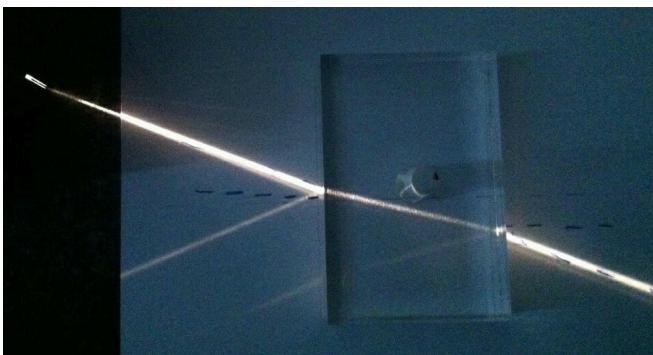


Imagen de [Physics Stack exchange](#)

Radiancia reflejada y refractada

La radiancia del haz reflejado (L_r) y la del haz refractado (L_t) son proporcionales a la radiancia incidente (L_i):

$$L_r(\dot{x}, \mathbf{w}_r) = F_R(\theta_i) L_i(\mathbf{w}_i, \dot{x})$$

$$L_t(\dot{x}, \mathbf{w}_t) = \left(\frac{n_t}{n_i}\right)^2 F_T(\theta_i) L_i(\mathbf{w}_i, \dot{x})$$

donde:

- La función F_R representa la fracción de luz reflejada, y F_T la refractada. Ambas funciones **suman la unidad**, y dependen de θ_i según las **Leyes de Fresnel**

$$F_R(\theta_i) + F_T(\theta_i) = 1$$

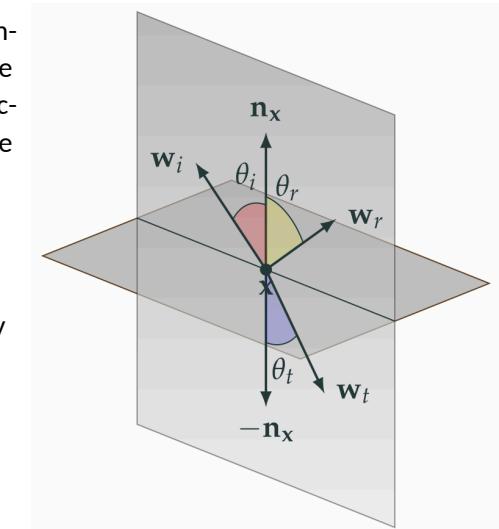
- El factor $(n_t/n_i)^2$ aparece por el cambio de ancho del haz.
- Los vectores \mathbf{w}_r y \mathbf{w}_t se calculan a partir de \mathbf{w}_i , $\mathbf{n}_{\dot{x}}$, n_i y n_t

Parámetros relevantes.

Por cada unidad de radiancia incidente desde \mathbf{w}_i (incidente en \dot{x}), parte se refleja de forma especular perfecta hacia la dirección \mathbf{w}_r y parte se refracta hacia \mathbf{w}_t :

- Se cumple $\theta_i = \theta_r$
- Sin embargo $\theta_i \neq \theta_t$
- El ángulo θ_t depende de θ_i , n_i y n_t .
- Definimos:

$$\begin{aligned} c_i &= \cos \theta_i = \mathbf{n}_{\dot{x}} \cdot \mathbf{w}_i \\ c_t &= \cos \theta_t = -\mathbf{n}_{\dot{x}} \cdot \mathbf{w}_t \end{aligned}$$



Cálculo de los vectores reflejado y refractado. Ley de Snell.

El vector \mathbf{w}_r se calcula a partir de \mathbf{w}_i y $\mathbf{n}_{\dot{x}}$ (como en reflex. especular perfecta):

$$\mathbf{w}_r = 2(\mathbf{n}_{\dot{x}} \cdot \mathbf{w}_i)\mathbf{n}_{\dot{x}} - \mathbf{w}_i$$

El vector refractado \mathbf{w}_t tiene longitud unidad, y

- se determina por la **Ley de Snell**: $n_i \sin \theta_i = n_t \sin \theta_t$,
- la cual permite calcular \mathbf{w}_t así:

$$\mathbf{w}_t = (rc_i - c_t)\mathbf{n}_{\dot{x}} - r\mathbf{w}_i \quad \begin{cases} c_i = \mathbf{n}_{\dot{x}} \cdot \mathbf{w}_i \quad (= \cos \theta_i) \\ c_t = \sqrt{1 - r^2(1 - c_i^2)} \quad (= \cos \theta_t) \\ r = n_i/n_t \end{cases}$$

- si $1 - r^2(1 - c_i^2) < 0$, entonces el valor c_t no está definido y ocurre **reflexión interna total** (no hay refracción, toda la radiancia se refleja).

Ver: [Univ. Stanford Course CS-148 notes](#)

Coefficientes de reflexión y refracción. La leyes de Fresnel

El **coeficiente de reflexión** F_R (fracción de energía reflejada) depende del ángulo de incidencia θ_i , según las **leyes de Fresnel**.

Para luz *no polarizada* es:

$$F_R(\theta_i) = \frac{1}{2} \left[\left(\frac{n_i c_i - n_t c_t}{n_i c_i + n_t c_t} \right)^2 + \left(\frac{n_i c_t - n_t c_i}{n_i c_t + n_t c_i} \right)^2 \right]$$

La **aproximación de Schlick** para F_R es más sencilla de calcular:

$$F_R(\theta_i) \approx f_0 + (1 - f_0)(1 - c_i)^5 \quad \text{donde: } f_0 = \left(\frac{n_i - n_t}{n_i + n_t} \right)^2$$

El **coeficiente de transmisión** F_T (fracción de energía refractada) se obtiene teniendo en cuenta el principio de conservación de energía:

$$F_T(\theta_i) = 1 - F_R(\theta_i)$$

La BRDF difusa

La BRDF difusa produce un aspecto mate, ya que la radiancia se refleja con igual intensidad en todas las direcciones desde cada punto:



[Unreal Engine: update lighting in templates](#) (izq.) [The Citizenry: Terracotta clay vases](#) (dcha.).

Subsección 3.3.

Reflexión difusa. Modelos Lambertianos y de Burley.

Expresión de la BRDF difusa Lambertiana

La expresión que define a BRDF difusa de *Lambert* es la siguiente:

$$f_{dl}(\dot{x}, \mathbf{w}_i, \mathbf{w}_o) = \frac{1}{\pi} k_d(\dot{x}) \quad (8)$$

- Similar al ya visto, pero normalizado para la conservación de la energía.
- $k_{dl}(x)$ es un valor cualquiera entre 0 y 1, que indica la fracción de radiancia reflejada de forma difusa en \dot{x} (es el *albedo*).
- f_{dl} es **independiente de \mathbf{w}_o** : un punto se ve igual de brillante desde cualquier dirección

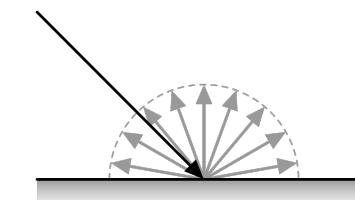


Imagen de Von Haarberg [Illustration of a diffuse BRDF \(wikipedia, CCO\)](#)

La componente difusa del modelo de Burley

El modelo de Lambert no se ajusta bien al comportamiento reflectivo real de los materiales difusos. Otros modelos intentan ser más realistas, se usa mucho el modelo propuesto por **Burley**, empleado de Disney, en 2012. La expresión es:

$$f_{bu}(\dot{x}, \mathbf{w}_i, \mathbf{w}_o) = \frac{k_{bu}(\dot{x})}{\pi} [1 + (f_{90} - 1)(1 - \mathbf{n}_{\dot{x}} \cdot \mathbf{w}_i)^5] [1 + (f_{90} - 1)(1 - \mathbf{n}_{\dot{x}} \cdot \mathbf{w}_o)^5]^9$$

Donde:

- El valor f_{90} se define como sigue:

$$f_{90} \equiv \frac{1}{2} + 2\alpha(\mathbf{w}_i \cdot \mathbf{h})^2 \quad \text{donde: } \mathbf{h} = \frac{\mathbf{w}_i + \mathbf{w}_o}{\|\mathbf{w}_i + \mathbf{w}_o\|}$$

- El valor k_{bu} está entre 0 y 1, como es habitual.
- El parámetro α es un real entre 0 y 1 que determina la **rugosidad (roughness)** del material. A mayores valores, más difuso es el material.

Voir: [Physically-Based Shading at Disney](#)

Sesión 7: Modelos de iluminación

Created 2025-12-01

Page 69 / 103.

3. Modelos realistas de iluminación. La BRDF..

3.4. Reflexión pseudo-especular. Modelos de microfacetas. GGX..

Modelos de BRDF basados en microfacetas

Los modelos más realistas y usados en la actualidad se basan en suponer que la superficie está formada (a pequeña escala) por **microfacetas (microfacets)** orientadas aleatoriamente, formando una *microsuperficie* \mathcal{M} , rugosa, cercana a la superficie \mathcal{G} (no rugosa):

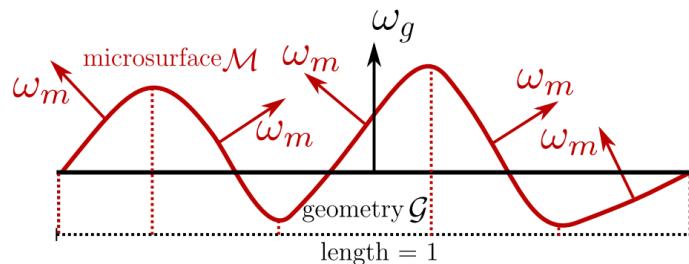


Imagen de: Eric Heitz JCGT (2014): *Understanding the Masking-Shading Function in Microfacet-Based BRDFs* <http://jcgtrg.org/published/0003/02/03/>

Subsección 3.4.

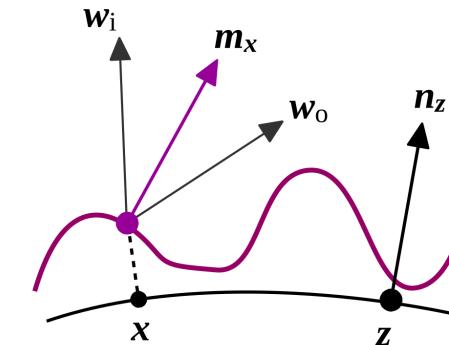
Reflexión pseudo-especular. Modelos de microfacetas. GGX..

3. Modelos realistas de iluminación. La BRDF..

3.4. Reflexión pseudo-especular. Modelos de microfacetas. GGX..

Reflexión especular en microfacetas. Normales.

Se asume que la reflexión en cada microfaceta ocurre de forma especular perfecta según las Leyes de Fresnel que hemos visto antes:



Para cada punto \dot{x} de la superficie, la normal a la superficie es $\mathbf{n}_{\dot{x}}$ (igual que antes), pero la normal de la microsuperficie es $\mathbf{m}_{\dot{x}}$. La reflexión especular entre \mathbf{w}_i y \mathbf{w}_o en \dot{x} solo ocurre si $\mathbf{m}_{\dot{x}}$ es la bisectriz de \mathbf{w}_i y \mathbf{w}_o (el caso de la figura).

Microfacetas: la distribución de normales D

La reflexión depende de las frecuencias relativas de las posibles orientaciones de la microsuperficie, es decir, de como se distribuyan sus normales:

- Las normales de la microsuperficie se distribuyen aleatoriamente según una función de distribución D (definida para cualquier versor normal \mathbf{m} en la semiesfera Ω^+).
- La función D depende de dos parámetros reales α_x y α_y , (ambos reales, > 0) que expresan la **rugosidad** de la superficie en las dos direcciones tangentes \mathbf{t}_x y \mathbf{t}_y , perpendiculares a la **normal a la macrosuperficie**, que es la normal en \dot{x} , es decir $\mathbf{n}_{\dot{x}}$.
- Para cualquier vector \mathbf{m} , la evaluación de $D(\mathbf{m})$ se hace teniendo en cuenta que el marco definido por \mathbf{t}_x , \mathbf{t}_y y $\mathbf{n}_{\dot{x}}$ es ortonormal (son tres vectores de longitud unidad y perpendiculares entre si)

La expresión de D

Para una normal a la microsuperficie \mathbf{m} cualquiera en un punto \dot{x} , podemos calcular $D(\mathbf{m})$:

- El vector \mathbf{m} es cualquier vector unitario, tal que $\mathbf{m} \cdot \mathbf{n}_{\dot{x}} \geq 0$ (está en la semiesfera superior definida por $\mathbf{n}_{\dot{x}}$, si no es el caso $D(\mathbf{m}) = 0$).
- La expresión de D para la **GGX anisotrópica** es:

$$D(\mathbf{m}) = \frac{1}{\pi \alpha_x \alpha_y \left(\left(\frac{\mathbf{m} \cdot \mathbf{t}_x}{\alpha_x} \right)^2 + \left(\frac{\mathbf{m} \cdot \mathbf{t}_y}{\alpha_y} \right)^2 + (\mathbf{m} \cdot \mathbf{n}_{\dot{x}})^2 \right)^2}$$

- En el caso de la **GGX isotrópica**, se particulariza para $\alpha_x = \alpha_y = \alpha$, simplificando la anterior expresión, obtenemos esta:

$$D(\mathbf{m}) = \frac{\alpha^2}{\pi \left((\mathbf{m} \cdot \mathbf{t}_x)^2 + (\mathbf{m} \cdot \mathbf{t}_y)^2 + (\alpha \mathbf{m} \cdot \mathbf{n}_{\dot{x}})^2 \right)^2}$$

La distribución de normales D

Para un dirección de entrada \mathbf{w}_i y otra de salida \mathbf{w}_o , la BRDF

- Es proporcional a la densidad de facetas orientadas en la dirección \mathbf{w}_h , que es la bisectriz entre \mathbf{w}_i y \mathbf{w}_o , es decir, es proporcional al valor $D(\mathbf{w}_h)$, donde:

$$\mathbf{w}_h = \frac{\mathbf{w}_i + \mathbf{w}_o}{\|\mathbf{w}_i + \mathbf{w}_o\|}$$

- Depende de las rugosidades α_x e α_y , que son las rugosidades en las direcciones tangenciales \mathbf{t}_x y \mathbf{t}_y , y hay dos tipos de distribuciones:
 - ▶ **GGX anisotrópica:** cuando $\alpha_x \neq \alpha_y$, la apariencia de la superficie depende de su orientación, es decir depende como se rote la superficie alrededor de la normal a la macrosuperficie $\mathbf{n}_{\dot{x}}$)
 - ▶ **GGX isotrópica:** cuando $\alpha_x = \alpha_y$, la apariencia de la superficie no depende de su orientación (rotación alrededor de \mathbf{n}).

Sombras y enmascaramiento

Parte de las microfacetas no son visibles desde \mathbf{w}_o (**enmascaramiento, masking**) o no lo son desde \mathbf{w}_i (**sombras arrojadas, shadowing**). Esto reduce la cantidad de luz reflejada desde \mathbf{w}_i hacia \mathbf{w}_o en una microfase perpendicular a \mathbf{m} :

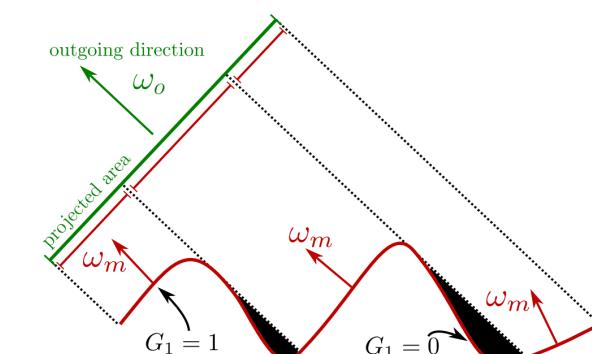


Imagen de: Eric Heitz JCGT (2014): Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs <http://jcg.org/published/0003/02/03/>

Sombras arrojadas y enmascaramiento

Para modelar las sombras/enmascaramiento se pueden usar diversas variantes, una realista y rápida de evaluar es la función G_2 llamada *height correlated masking and shadowing*, propuesta por E. Heitz en 2014, y definida así:

$$G_2(\mathbf{w}_i, \mathbf{w}_o, \mathbf{m}) = \frac{\chi^+(\mathbf{w}_o \cdot \mathbf{m}) \chi^+(\mathbf{w}_i \cdot \mathbf{m})}{1 + \Lambda(\mathbf{w}_i) + \Lambda(\mathbf{w}_o)}$$

donde la función Λ se define, para cualquier versor \mathbf{w} , como:

$$\Lambda(\mathbf{w}) = \frac{1}{2} \left(-1 + \sqrt{1 + \frac{\alpha_x^2 x^2 + \alpha_y^2 y^2}{z^2}} \right)$$

donde (x, y, z) son las coordenadas de \mathbf{w} , y donde

$$\chi^+(a) = \begin{cases} 1 & \text{si } 0 \leq a \\ 0 & \text{en otro caso} \end{cases}$$

La BRDF GGX

Teniendo en cuenta todos los factores, la BRDF f_r en un punto \dot{x} con normal (a la macrosuperficie) $\mathbf{n}_{\dot{x}}$ se puede expresar como aparece aquí:

$$f_{\text{ggx}}(\dot{x}, \mathbf{w}_i, \mathbf{w}_o) = \frac{F_R(\mathbf{w}_i, \mathbf{w}_h) D(\mathbf{w}_h) G_2(\mathbf{w}_i, \mathbf{w}_o, \mathbf{w}_h)}{4(\mathbf{w}_i \cdot \mathbf{n}_{\dot{x}})(\mathbf{w}_o \cdot \mathbf{n}_{\dot{x}})} \quad (10)$$

- Una variante de esta expresión fue introducida por primera vez por K.E. Torrance y E.M. Sparrow en 1967.
- Esta expresión es la que aparece en el artículo de B.Walter y otros (2007)
- Existen varias opciones para las funciones D y G , en la actualidad se usan las funciones D y $G = G_2$ que ya se han explicado, con estas funciones la BRDF se denomina **BRDF GGX**.

Artículos en PDF:

[Torrance y Sparrow, 1967, Theory for off-specular reflection from roughened surfaces](#)
[Walter et al., 2007, Microfacet Models for Refraction](#)

Factor de Fresnel con la aproximación de Schlick

Puesto que esta BRDF consiste en reflexión especular en las microfacetas (ignorando la componente de transmisión), es necesario tener en cuenta el factor de reflexión Fresnel que ya se ha descrito (F_R)

- El ángulo de incidencia de la reflexión (θ_i) es el ángulo entre los vectores \mathbf{w}_i y \mathbf{w}_h . Así que el término $F_R(\theta_i)$ se puede expresar como $F_R(\mathbf{w}_i, \mathbf{w}_h)$.
- Hay que tener en cuenta los índices de refracción del medio de incidencia (usualmente el aire), y el medio de refracción (el material de la superficie), son n_i y n_t , respectivamente.
- Se suele usar la aproximación de Schlick, es decir, se suele usar este factor:

$$F_R(\mathbf{w}_i, \mathbf{w}_h) \approx f_0 + (1 - f_0)(1 - (\mathbf{w}_i \cdot \mathbf{w}_h))^5 \quad \text{donde: } f_0 = \left(\frac{n_i - n_t}{n_i + n_t} \right)^2$$

Los parámetros de rugosidad

Los parámetros α_x y α_y expresan la rugosidad de la superficie:

- Cuando los valores son distintos, la BRDF es una BRDF **anisotrópica**. Se corresponde con materiales que están pulidos en una dirección.
- Lo más frecuente es que ambos valores coincidan (hablamos del valor α simplemente), entonces la BRDF es **isotrópica** (no hay una dirección preferida)
- Los valores de rugosidad α_x , α_y (o simplemente α) no pueden ser nulos, pero cuando decrecen hacia cero, la BRDF GGX se aproxima la BRDF de la reflexión especular perfecta.
- Cuando los valores de rugosidad crecen, la superficie es cada vez más rugosa (no es un espejo perfecto).

Ejemplo de rugosidad decreciente

En esta figura la rugosidad decrece de izquierda a derecha hasta llegar a casi el valor nulo. Menores valores de rugosidad significan menos variación en la dispersión de la luz reflejada (para una dirección indicente fija).

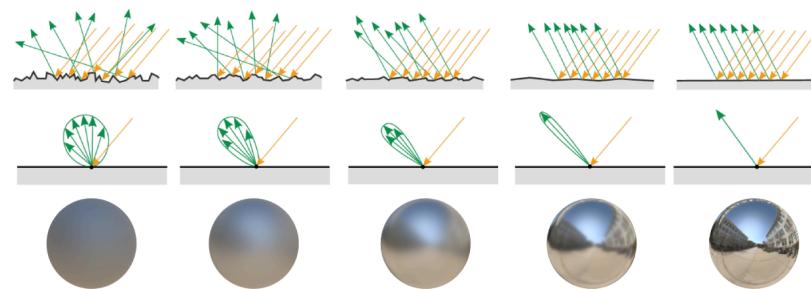


Imagen de: [S. Lagarde y C. de Rusiers, SIGGRAPH 2014 Course Notes](#)

Ejemplo de imagen con materiales anisotrópicos

En estas imágenes sintetizadas vemos el efecto de usar diversos pares de valores de α_x y α_y en los cuales $\alpha_x \neq \alpha_y$ (son materiales *anisotrópicos*)



Imagen de: [Physically Based Rendering in Filament](#)

Combinaciones de BRDFs

En una misma superficie se pueden combinar dos o más BRDFs, básicamente de dos formas:

- Mediante una suma ponderada de dos o más BRDFs, por ejemplo, se puede combinar una BRDF difusa lambertiana con peso a y una BRDF especular GGX, con peso $1 - a$ (con $a \in [0, 1]$)

$$f_r(\dot{x}, \mathbf{w}_i, \mathbf{w}_o) = a f_{\text{bu}}(\dot{x}, \mathbf{w}_i, \mathbf{w}_o) + (1 - a) f_{\text{ggx}}(\dot{x}, \mathbf{w}_i, \mathbf{w}_o)$$

donde f_{bu} es la BRDF difusa de Burley de la expresión (9), en la transparencia 69), y f_{ggx} es la BRDF GGX definida por la expresión (10) en la transparencia 79).

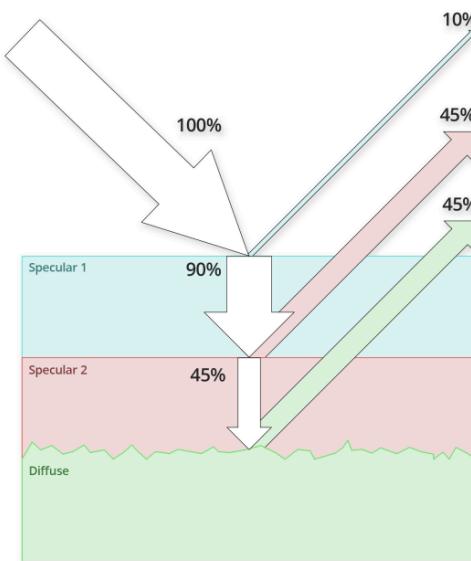
- Mediante capas: una capa superior con una BRDF (p.ej. GGX) y una capa inferior con otra BRDF (p.ej. difusa). La luz incidente atraviesa la capa superior, llega a la capa inferior, se refleja allí y vuelve a atravesar la capa superior. Este modelo es más costoso de evaluar, pero puede ser más realista.

Modelos multi-capas

La luz incidente sobre las capas transparentes superiores en parte se refleja hacia el medio de incidencia, y en parte se refracta (transmite) hacia la capa inmediatamente inferior.

En el sustrato difuso (blanco) se refleja el 100% de la luz incidente.

Imagen de Anders Langland, en:
[Physically Based Shading in Arnold](#)



Sección 4. Modelos de fuentes de luz.

Fuentes de luz realistas

En este documento ya se han descrito dos modelos sencillos de fuentes de luz (**fuentes puntuales** y **fuentes direccionales**), sin embargo en la actualidad los *engines* de render contemplan, además de esos, otros tipos de fuentes más realistas.

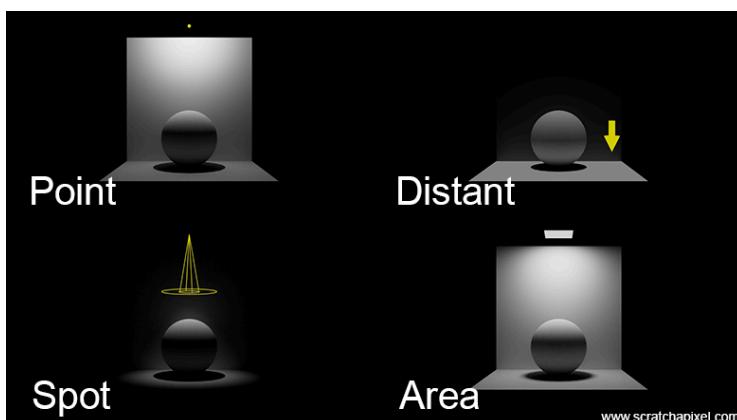


Imagen de Jean Colas Prunier en la web [Scratchapixel](#), página [introduction to lighting](#)

Tipos de fuentes de luz (1/2)

Aunque en la realidad hay muchísimos tipos de luminarias, los modelos más frecuentemente usado en los *engines* y aplicaciones gráficas interactivas son de alguno de estos tipos.

Los dos tipos que hemos visto ya son:

Fuente puntual (*point light*):

La radiancia se emite con la misma intensidad de luz en todas las direcciones desde un punto del espacio conocido. Si bien antes no se contempló, en la realidad esa intensidad decrece con el cuadrado de la distancia entre la fuente y el punto a iluminar.

Fuente direccional (*directional light* o *distant light*):

Se emite luz desde un punto muy lejano, con lo cual la iluminación solo depende de un vector de dirección fijo y conocido, el mismo para todos los puntos a iluminar.

Ejemplo de fuente puntual

Una fuente de luz puntual situada sobre la tetera.

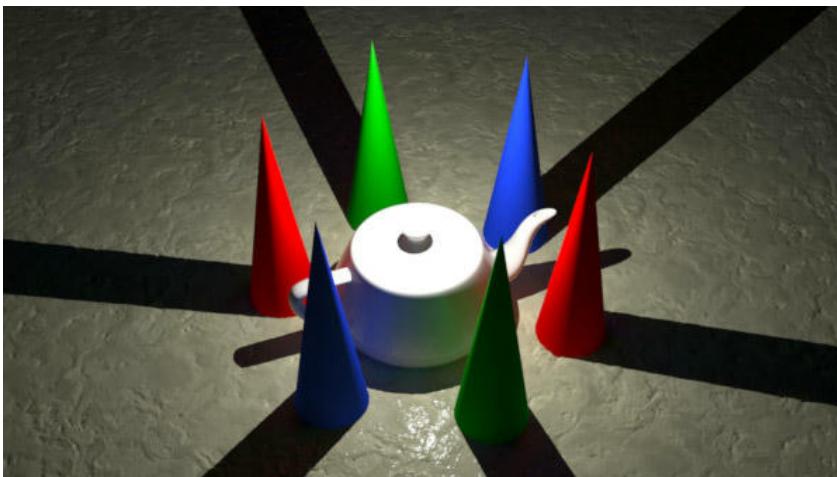


Imagen de [Manual de la aplicación Modo: Point Lights](#)

Tipos de fuentes de luz (2/2)

A los tipos anterior se le añaden otros tipos de luminarias:

Foco direccional (*spot light*):

Son un tipo de fuentes de luz puntuales, pero que no emiten la misma intensidad en todas las direcciones posibles.

Luz de área (*area light*):

En la realidad las luminarias siempre tienen cierto área, nunca emiten desde un punto, así que hay modelos que contemplan la emisión desde todos los puntos de una superficie (un rectángulo, un disco, una esfera, o en general una malla de polígonos).

Este tipo de luminarias son **mucho más realistas pero más complejas de simular que las anteriores**, y en principio no las vamos a tener en cuenta, aunque se usan mucho el los sistemas de rendering realista que simulan la *iluminación global*, y menos en los programas interactivos.

Ejemplo de fuente direccional

Una fuente de luz direccional, con la fuente en la dirección hacia arriba a la izquierda de la imagen. Estas fuentes se usan para simular la luz del sol.



Imagen de [Manual de la aplicación Modo: Directional light](#)

Ejemplo de fuente puntual de tipo Spot

Una fuente de luz puntual de tipo *Spot*, situada a la izquierda de la imagen.

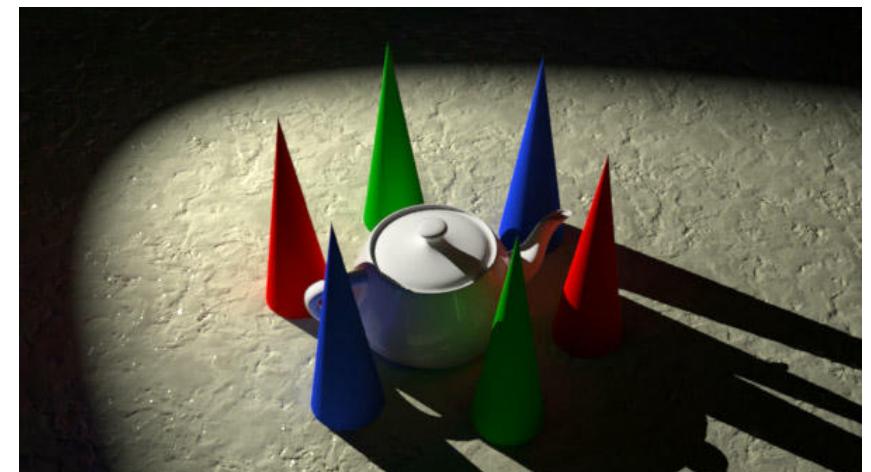


Imagen de [Manual de la aplicación Modo: Spot Lights](#)

Ejemplo de fuente de área

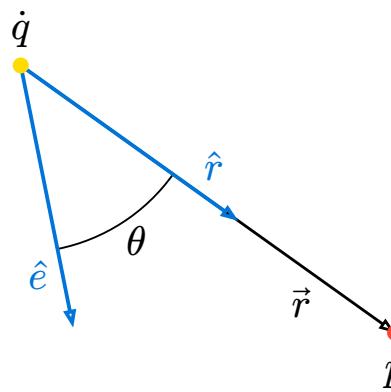
Una fuente de luz de área, en concreto un disco.



Imagen: *Physically Based Rendering: From Theory To Implementation* (3rd Ed.), Matt Pharr, Wenzel Jakob y Greg Humphreys. [Sección 12.5: Area Lights](#)

Parámetros de las fuentes puntuales y spot

En esta figura se aprecian los distintos vectores, distancias y ángulos.



$$\begin{aligned}\theta &= \arccos(\hat{r} \cdot \hat{e}) \\ \vec{r} &= \dot{p} - \dot{q} \\ r &= \|\vec{r}\| \\ \hat{r} &= \frac{\vec{r}}{r}\end{aligned}$$

Parámetros relacionados con las fuentes de luz puntuales

Si queremos calcular la iluminación en un punto \dot{p} iluminado por una fuente de luz puntual, en un punto \dot{q}

- El punto \dot{q} es la posición de la fuente de luz puntual.
- El vector que va desde \dot{q} hasta \dot{p} lo llamamos $\vec{r} \equiv \dot{p} - \dot{q}$.
- La distancia entre ambos puntos es $r \equiv \|\vec{r}\|$ (suponemos que $r > 0$).
- El vector normalizado desde la fuente hasta el punto \dot{p} es $\hat{r} \equiv \vec{r}/r$

En estas condiciones la intensidad de la luz reflejada en \dot{p} (debida la luz proveniente de \dot{q} e incidente sobre \dot{p}) será proporcional a:

- Una función $f(r)$ que indica como se atenúa la intensidad con la distancia, se cumple $f(r) \in [0...1]$.
- Una función $g(\hat{r})$ que depende de la dirección de salida \hat{r} .

Atenuación con la distancia

En la naturaleza la función f de la distancia es simplemente la inversa del cuadrado de la distancia $f(r) = 1/r^2$. Sin embargo, para aplicaciones interactivas a veces es conveniente usar otras expresiones:

- Usar tres constantes a , b y c y definir:

$$f(r) = \frac{1}{a + br + cr^2}$$

- Usar dos constantes d_0 y d_1 , con $d_0 < d_1$, y definir f de forma que vale 1 pero a partir de d_0 desciende suavemente hasta anularse a partir de d_1 , por ejemplo:

$$f(r) = \begin{cases} 1 & \text{si } r \leq d_0 \\ \left(1 - \frac{r - d_0}{d_1 - d_0}\right)^2 & \text{si } d_0 < r < d_1 \\ 0 & \text{si } d_1 \leq r \end{cases}$$

Luces de tipo Spot (dependencia de la dirección)

En una luz puntual por defecto la radiancia emitida es la misma para todas las direcciones de salida \hat{r} posibles, sin embargo, en una luz de tipo spot esa radiancia sí depende de la dirección \hat{r} .

- Las luces spot vienen caracterizadas por un vedor \hat{e} que es la dirección de máxima intensidad.
- Para una dirección de salida cualquiera \hat{r} , la radiancia, en general, será una función g decreciente con el ángulo θ entre \hat{r} y \hat{e} (donde $\theta = \arccos(\hat{r} \cdot \hat{e})$).
- Algunas opciones comunes:
 - ▶ Lóbulo coseno: $g(\theta) \equiv (\cos \theta)^n$, con $n \geq 1$.
 - ▶ Elegir dos ángulos θ_0 y θ_1 , con $0 < \theta_0 < \theta_1 < \pi/2$, y definir:

$$g(\theta) \equiv \begin{cases} 1 & \text{si } 0 < \theta \leq \theta_0 \\ \left(1 - \frac{\theta - \theta_0}{\theta_1 - \theta_0}\right)^2 & \text{si } \theta_0 < \theta < \theta_1 \\ 0 & \text{si } \theta_1 \leq \theta \end{cases}$$

Luces fotométricas

A veces se usan **luces fotométricas (photometric lights)**. Son un tipo de spot lights donde las funciones f y g se implementan de forma tabulada. Se usan para representar con fidelidad el comportamiento de **luminarias reales**.

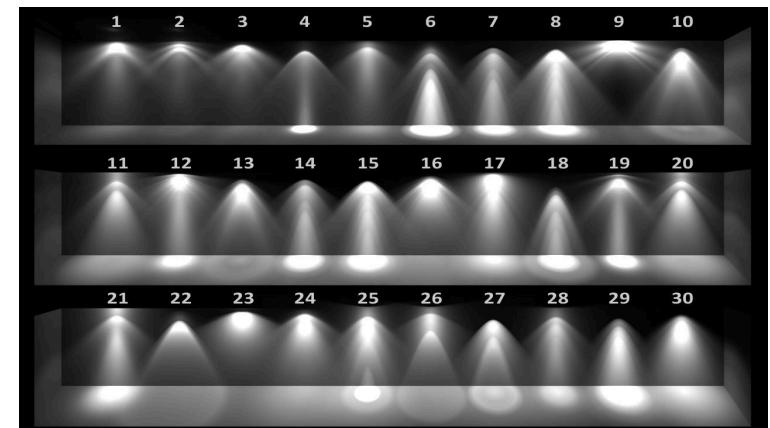


Imagen de [Manual de la aplicación Arnold de Autodesk: Photometric Lights](#)

Problemas: evaluación de la componente pseudo-especular del modelo de Phong

Problema 7.1:

Escribe el código GDScript de una función para calcular la reflectividad debida a la componente pseudo-especular del modelo de iluminación local de Phong, es decir, el valor real resultado de evaluar la expresión de f_{ph} que aparece en la ecuación (6) de la transparencia 40.

La función recibirá como parámetros los vectores unitarios \mathbf{n}_p , \mathbf{v} y \mathbf{l}_i , (todos de tipo **Vec3**), el exponente e y el valor escalar k_{ph} (ambos de tipo **float**). La función devolverá un valor de tipo **float**.

Escribe otra versión de la función, con los mismos parámetros, pero ahora para la componente pseudo-especular del modelo de Blinn-Phong, es decir, para evaluar la expresión de f_{bp} que aparece en la ecuación (7) de la transparencia 43.

Problemas: modelos de iluminación

Problema 7.2:

Suponemos que visualizamos una esfera de radio unidad centrada en el origen. Se ilumina con una fuente de luz puntual situada en el punto $p = (0, 2, 0)$. Consideramos únicamente la componente difusa del modelo de iluminación sencillo (es decir, usando únicamente la ecuación (5) de la transparencia 34). El observador está situado en $o = (2, 0, 0)$.

1. Describe razonadamente en qué punto de la superficie de la esfera el brillo será máximo ¿ es ese punto visible para el observador ?
2. Repite el razonamiento anterior asumiendo ahora que el material es puramente pseudo-especular, es decir, consideramos únicamente la ecuación (6) de la transparencia 40. Indica si dicho punto es visible para el observador.
3. Haz lo mismo para un material pseudo-especular, pero con el modelo de Blinn-Phong (ecuación (7) de la transparencia 43).

Problema: evaluación de la BRDF GGX

Problema 7.3:

Escribe el código GDScript de una función para calcular la reflectividad debida a la BRDF de microfacetas GGX, es decir, el color resultado de evaluar la expresión de f_{ggx} que aparece en la ecuación (10) de la transparencia 79.

La función recibirá como parámetros los vectores unitarios w_i, w_o, t_x, t_y y n_x , (todos de tipo **Vec3**), y los valores α_x y α_y (ambos de tipo **float**). La función devolverá un valor de tipo **float**.

Fin de transparencias.

Informática Gráfica.

Sesión 8: Texturas, sombreado y materiales.

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

Sección 1.

Texturas

1. Coordenadas de textura.
2. Asignación explícita de coords. de textura.
3. Asignación de coordenadas y consulta de texels.
4. Consulta de texels en texturas de imagen.
5. Problemas sobre texturas.

Índice

Texturas	3
Sombreado (<i>Shading</i>)	37
Luces, materiales y texturas en Godot.	55

1. Texturas.

Detalles a pequeña escala

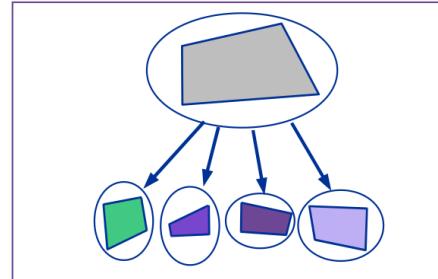
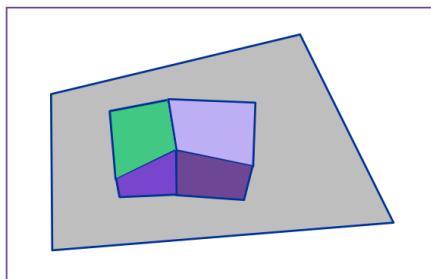
Los objetos reales presentan a veces detalles a pequeña escala, como son manchas, defectos, motivos ornamentales, rugosidades, o, en general, cambios en el espacio de los atributos que determinan su apariencia:



- Estas variaciones se pueden modelar como funciones que asignan a cada punto de la superficie de un objeto un valor diferente para algunos parámetros del MIL.
- Lo más usual es variar las reflectividades difusa y ambiente, pero se hace también con la normal (rugosidades), o a veces otros parámetros.

Implementación de detalles: polígonos de detalle

Para reproducir detalles a pequeña escala se pueden usar **polígonos de detalle**, son polígonos pequeños adicionales a los que definen la geometría de la escena, pero con materiales y/o orientación distintos entre ellos:



La desventaja de esta opción es su enorme complejidad en espacio (necesario para almacenar muchos polígonos pequeños) y tiempo (empleado en su visualización).

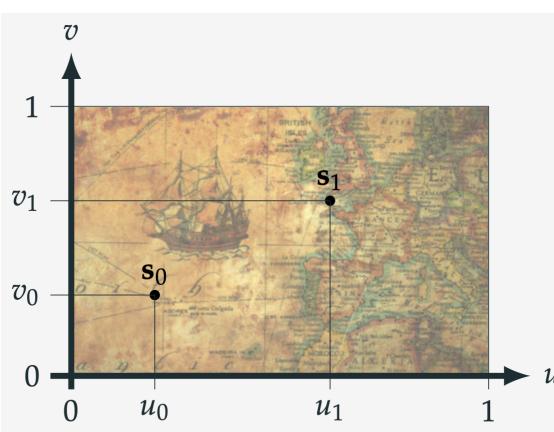
Texturas

Una opción mejor (mucho más eficiente) es usar imágenes para representar las funciones antes citadas. A estas imágenes se les llama (en el contexto de la Informática Gráfica) **texturas**:

- Una textura se puede interpretar como una función T que asocia a cada punto s de un dominio D (usualmente $[0, 1] \times [0, 1]$) un valor para un parámetro del MIL (típicamente el color $C(p)$). La función T determina como varía el parámetro en el espacio.
- La función T puede estar representada en memoria como una matriz de pixels RGB (una imagen discretizada), a cuyos pixels se les llama **texels** (*texture elements*). A esta imagen se le llama **imagen de textura**.
- La función T puede también representarse como un subprograma que calcula los valores a partir de s (que se le pasa como parámetro). A este tipo de texturas le llamamos **texturas procedurales**.

La textura como una función

En este ejemplo vemos una imagen de textura (bidimensional). El dominio D es $[0, 1]^2$. Cada punto del dominio es una par $s = (u, v)$. Los valores $T(s) = T(u, v)$ son ternas RGB.



Aplicación de texturas

Vemos varias formas de asignar colores a puntos del objeto:

- Evaluación del MIL con reflectividades blancas (izq. abajo)
- Eso directo de colores de la textura (izq. arriba)
- Evaluación del MIL con reflectividades obtenidas de la textura (der.)



Coordenadas de textura

Para poder aplicar una textura a la superficie de un objeto, es necesario hacer corresponder cada punto $\mathbf{p} = (x, y, z)$ de su superficie con un punto (u, v) del dominio de la textura:

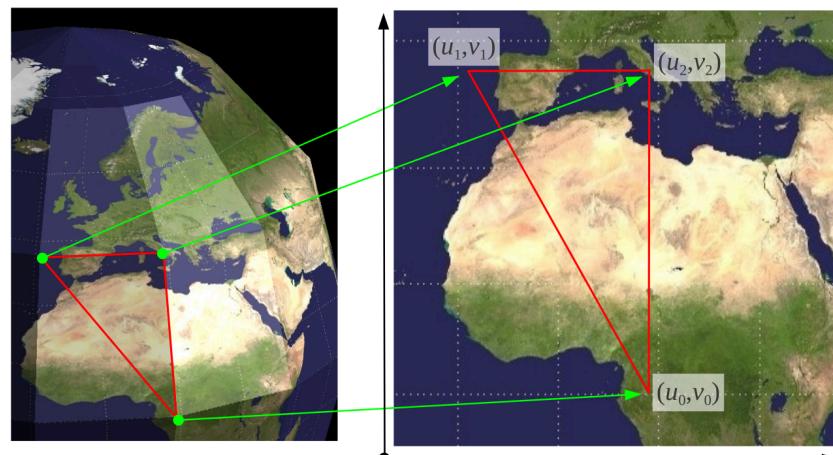
- Debe existir una función f tal que $(u, v) = f(x, y, z)$
- Si $(u, v) = f(x, y, z)$ entonces decimos que (u, v) son las **coordenadas de textura** del punto $\mathbf{p} = (x, y, z)$.
- Normalmente f se descompone en dos componentes f_u, f_v , de forma que $u = f_{u(x,y,z)}$ y $v = f_{v(x,y,z)}$
- La función f puede implementarse usando una tabla de coordenadas de textura de los vértices, o bien calcularse proceduralmente con un subprograma.

Subsección 1.1.

Coordenadas de textura.

Ejemplo de coordenadas de textura.

Vemos como a cada vértice de un triángulo del modelo se le asignan sus coordenadas de textura (u_i, v_i) (donde i es el índice del vértice en la tabla de vértices).



Asignación explícita o procedural

La asignación de coord. de text. se puede hacer usando:

- **Asignación explícita a vértices:** las coordenadas forman parte de la definición del modelo de escena, y son un dato de entrada al cauce gráfico, en forma de un vector o tabla de coordenadas de textura de vértices $(v_0, u_0), (v_1, u_1), \dots, (u_{n-1}, v_{n-1})$.
 - ▶ se puede hacer manualmente en objetos sencillos, o bien
 - ▶ de forma asistida usando software para CAD.

Esto hace necesario realizar una interpolación de coords. de text. en el interior de los polígonos.

- **Asignación procedural:** f se implementa con un subprograma que se invoca como `CoordText(p)` y que calcula las coordenadas de textura (para un punto \mathbf{p} devuelve el par $(u, v) = f(\mathbf{p})$ con las coords. de textura de \mathbf{p}).

Ejemplo de asignación explícita.

Esto es posible en objetos sencillos como este cubo construido con triángulos. En este ejemplo se busca una asignación de coordenadas de textura que sea continua en las aristas:

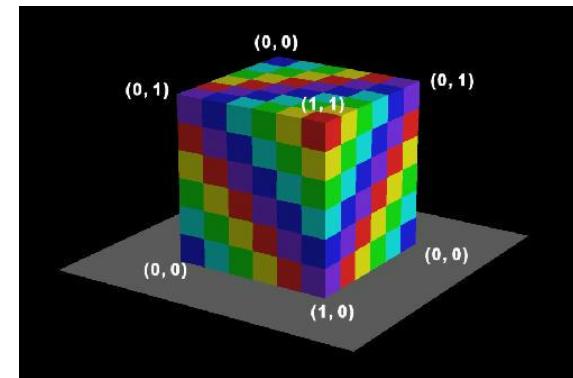


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

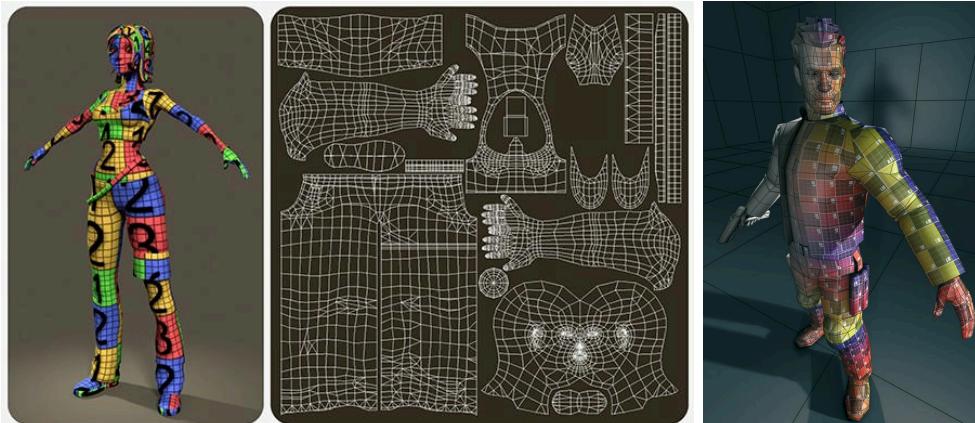
Subsección 1.2.

Asignación explícita de coords. de textura.

1. Texturas.
1.2. Asignación explícita de coords. de textura..

Ejemplo de uso de herramientas CAD.

En objetos complejos es necesario el uso de herramientas CAD:



Imágenes de [Sean Dixon](#) (izquierda, centro) y [Mayan Escalante](#) (derecha).

Subsección 1.3.

Asignación de coordenadas y consulta de texels.

Tipos de asignación procedural

Hay dos opciones:

- **Asignación procedural a vértices:** se invoca `CoordText(v_i)` para calcular las coordenadas de textura en cada vértice v_i , y las coordenadas obtenidas se almacenan y después se interpolan linealmente en el interior de los polígonos de la malla.
 - ▶ Funciona de forma totalmente correcta (exacta) solo cuando f es lineal, en otro caso es una aproximación lineal a trozos.
- **Asignación procedural a puntos:** se invoca `CoordText(p)` cada vez que sea necesario evaluar el MIL en un punto de la superficie p .
 - ▶ Permite exactitud incluso aunque f sea no lineal.
 - ▶ Esto requiere programación del cauce gráfico, invocando a `CoordText` en cada pixel desde el *fragment shader*.

Otras opciones para asignación procedural

Otras opciones (no lineales) son estas dos:

- **Coordenadas polares** (proyección en una esfera): el punto p se expresa en coordenadas polares como una terna (α, β, r) , los valores u y v se obtienen de α y β .
- **Coordenadas cilíndricas** (proyección en un cilindro): el punto p se expresa en coordenadas cilíndricas como una terna (α, y, r) , los valores u y v se obtienen de α e y .

Es muy complicado usarlas con asignación a vértices (α puede pasar de 360 a 0 en un triángulo, la textura se vería mal), y por tanto requieren usar asignación procedural a puntos (invocar `CoordText` desde los *fragment shaders*).

Funciones para asignación procedural:

Los tipos de funciones f más frecuentes son:

- **Funciones lineales** de la posición (proyección en un plano): el punto $p = (x, y, z)$ se proyecta sobre un plano y se expresa como un par (x', y') de coordenadas en dicho plano, que se interpretan como coordenadas de textura.
- **Coordenadas paramétricas:** se pueden usar si la malla aproxima una superficie paramétrica (p.ej. la tetera, hecha de superficies paramétricas tipo B-spline). Se usa asignación procedural a vértices. Se trata de funciones no lineales de la posición.

Funciones lineales (proyección).

En este caso el punto $p = (x, y, z)$ se proyecta en un plano, y se usan las coordenadas del punto proyectado (en el sistema de referencia del plano), como coordenadas de textura.

El plano estará definido por un punto por el que pasa (q) y por dos vectores libres (e_u y e_v , de longitud unidad y perpendiculares entre sí). En estas condiciones:

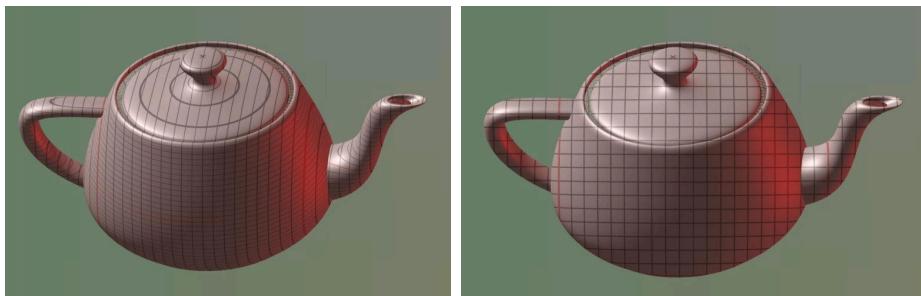
$$u = f_u(p) = (p - q) \cdot e_u \quad v = f_v(p) = (p - q) \cdot e_v$$

como casos particulares, y a modo de ejemplo, podemos hacer q igual al origen $(0, 0, 0)$, $e_u = x = (1, 0, 0)$ y $e_v = y = (0, 1, 0)$, y en este caso es una proyección paralela el eje Z, sobre el plano XY (descarta la Z)

$$\begin{aligned} u &= x = p \cdot x = (x, y, z) \cdot (1, 0, 0) \\ v &= y = p \cdot y = (x, y, z) \cdot (0, 1, 0) \end{aligned}$$

Ejemplo de proyección paralela a Z.

Las coordenadas de p que se usan en las funciones lineales pueden ser las coordenadas de objeto (izquierda) o bien o las coordenadas de mundo (derecha). Aquí vemos un ejemplo de una proyección paralela al eje Z:



Este método funciona mejor (menor deformación) cuando la normal es aproximadamente paralela a la dirección de proyección (parte frontal en el ejemplo de la izquierda).

Ejemplo de coordenadas paramétricas

Vemos un ejemplo de textura (izq.) y su aplicación a la tetera (der.). Este tipo de superficies se denominan superficies paramétricas de Bézier o B-spline, y se usan mucho en aplicaciones de diseño (CAD).



Esta imagen se ha generado asignando explícitamente en el programa a cada vértice sus coordenadas de textura, usando para ello sus coordenadas paramétricas.

Coordenadas paramétricas.

Una **superficie paramétrica** es una variedad plana de dos dimensiones (que puede ser abierta o cerrada), para la cual existe una función g (con dominio en $[0, 1] \times [0, 1]$) tal que, si p es un punto de la superficie, entonces existe un tupla de reales (s, t) tales que $p = g(s, t)$:

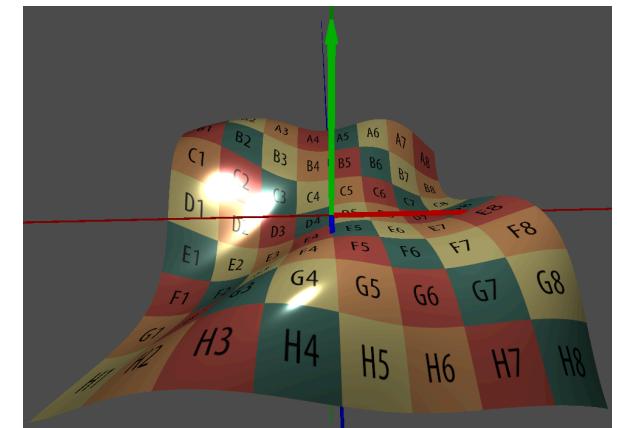
- En este caso, al par (s, t) se le llaman **coordenadas paramétricas** del punto p , y a la función g se le llama **función de parametrización** de la superficie.
- Usando la capacidad de evaluar g , podemos construir una malla que aproxima cualquier superficie paramétrica. La posición p_i del i -ésimo vértice se obtiene como $g(s_i, t_i)$, donde los (s_i, t_i) forman una rejilla en $[0, 1] \times [0, 1]$.
- En estas condiciones, podemos hacer $(u, v) = f(p) = (s, t)$, es decir, podemos usar las coordenadas paramétricas como coordenadas de textura.

Coordenadas paramétricas: campos de alturas

Una opción bastante simple es usar un **campo de alturas (height field)**, en el cual el punto $p = (x, y, z)$ se escribe en función de (u, v) así:

$$\begin{aligned} x &= u \\ y &= h(u, v) \\ z &= v \end{aligned}$$

donde h es una función que expresa la altura de cada vértice en el eje Y (podrían ser los otros).



Coordenadas esféricas

Se basa en usar las coordenadas polares (longitud, latitud y radio) del punto p:

- Equivale a una proyección radial en una esfera.
- Las coordenadas (α, β, r) se obtienen a partir de las coordenadas cartesianas (x, y, z) (normalmente coordenadas de objeto, con el origen en un punto central de dicho objeto). Hacemos:

$$\alpha = \text{atan2}(z, x) \quad \beta = \text{atan2}\left(y, \sqrt{x^2 + z^2}\right)$$

- Se obtiene α en el rango $[-\pi, \pi]$ y β en el rango $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Por tanto, podemos calcular u y v como sigue:

$$u = \frac{1}{2} + \frac{\alpha}{2\pi} \quad v = \frac{1}{2} + \frac{\beta}{\pi}$$

el valor de r no se usa y por tanto no es necesario calcularlo.

Coordenadas cilíndricas

Se usan las coordenadas polares (ángulo y altura) del punto p:

- Equivale a una proyección radial en un cilindro (cuyo eje es usualmente un eje vertical central al objeto).
- Las coordenadas (α, h, r) se obtienen a partir de las coordenadas cartesianas (x, y, z) (también con origen en el centro del objeto). Hacemos:

$$\alpha = \text{atan2}(z, x) \quad h = y$$

- El valor de α está en el rango $[-\pi, \pi]$ y h en el rango $[y_{\min}, y_{\max}]$ (el rango en Y del objeto). Por tanto, podemos calcular u y v como:

$$u = \frac{1}{2} + \frac{\alpha}{2\pi} \quad v = \frac{y - y_{\min}}{y_{\max} - y_{\min}}$$

tampoco el valor de r se usa ahora y por tanto no es necesario calcularlo.

Ejemplo de coordenadas esféricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

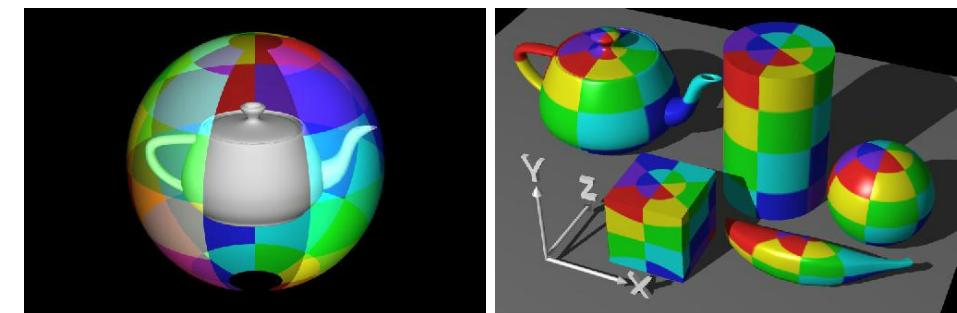


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

Ejemplo de coordenadas cilíndricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

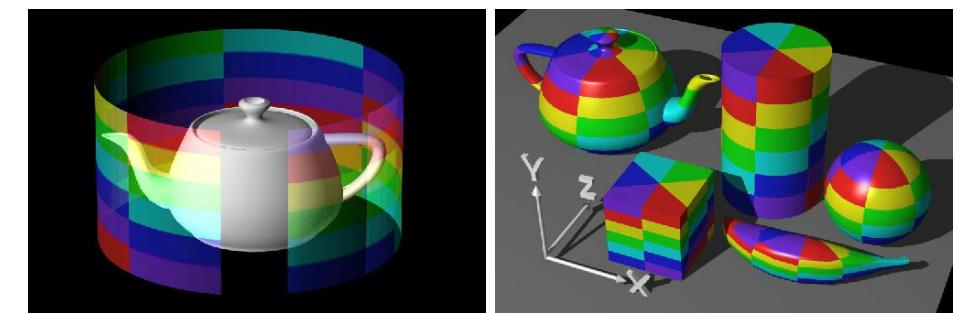


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

Consulta de texels en texturas de imagen.

En una textura de imagen con n_x columnas de texels y n_y filas, podemos interpretar que cada texel tiene asociada un pequeño rectángulo contenido en $[0, 1]^2$. El texel en la columna i , fila j tendrá un área con centro en el punto (c_i, d_j)

Subsección 1.4.

Consulta de texels en texturas de imagen.

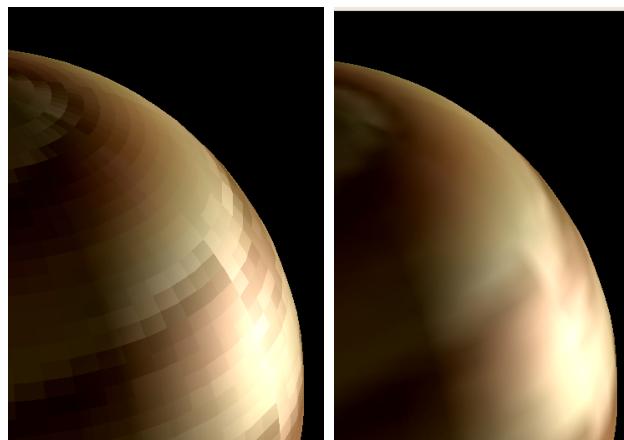
La consulta del color de la textura en un punto (u, v) puede hacerse de dos formas:

- **más cercano:** usar el color del texel cuyo centro sea más cercano a la posición (u, v) , es equivalente a seleccionar el texel cuya área contiene a (u, v) .
- **interpolación:** realizar un interpolación (bilineal) entre los colores de los cuatro texels con centros más cercanos al punto (u, v) .

Las diferencias entre ambos métodos son visibles cuando la proyección en la ventana de un texel ocupa muchos pixels.

Interpolación bilineal

Aquí vemos una textura de baja resolución, vista de cerca, que se visualiza usando los dos métodos:

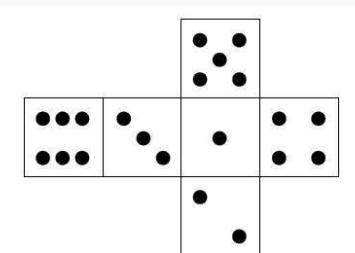


Subsección 1.5.
Problemas sobre texturas.

Problemas: coordenadas de textura (1/3)

Problema 8.1:

Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar un textura que incluya las caras de un dado. Para ello disponemos de una imagen de textura que tiene una relación de aspecto 4:3. La imagen aparece aquí:



(continua en la siguiente transparencia)

Problemas: coordenadas de textura (2/3)

Problema 8.2:

Considera de nuevo el cubo y la textura del problema anterior. Ahora supón que queremos que el cubo se visualize iluminado, para lo cual debemos de asignar normales a los vértices. Responde a estas cuestiones

- Describe razonadamente si sería posible usar la misma tabla de vértices y la misma tabla de coordenadas de textura que has escrito en el problema anterior, o es necesario usar otra tabla distinta.
- Si has respondido que no es posible usar las mismas tablas, escribe la nueva tabla de vértices, la nueva tabla de coordenadas de textura. Asimismo, escribe como sería la tabla de normales.

Problemas: coordenadas de textura (1/3, cont.)

Problema 8.1 (continuación):

Responde a estas cuestiones:

1. Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
2. Escribe la tabla de coordenadas de vértices, la tabla de coordenadas de textura, y la tabla de triángulos. Ten en cuenta que el cubo tiene lado unidad y su centro está en $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. Dibuja un esquema de la textura en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de textura.

Sección 2.

Sombreado (*Shading*)

1. Evaluación del MIL en rasterización.
2. Sombreado plano.
3. Sombreado en los vértices.
4. Sombreado en los píxeles.

2. Sombreado (*Shading*).

2.1. Evaluación del MIL en rasterización..

Alternativas

En el algoritmo de Z-buffer, la evaluación del MIL puede hacerse en tres puntos distintos del cauce gráfico:

- **Sombreado plano:** (*flat shading*) una vez por cada polígono que forma el modelo, asignando el resultado (una terna RGB única) a todos los pixels donde se proyecta el polígono.
- **Sombreado de vértices:** (*smooth shading* o *Gouraud shading*) una vez por vértice, cada color RGB obtenido se usa para interpolar los colores de los pixels en cada polígono.
- **Sombreado de pixel:** (*pixel shading* o *Phong shading*) una vez por cada pixel donde se proyecta el polígono

Subsección 2.1.

Evaluación del MIL en rasterización.

Subsección 2.2.

Sombreado plano.

Sombreado plano

Este método de sombreado es muy eficiente en tiempo si el modelo es sencillo (es decir, el número de polígonos es pequeño en comparación con el de pixels).

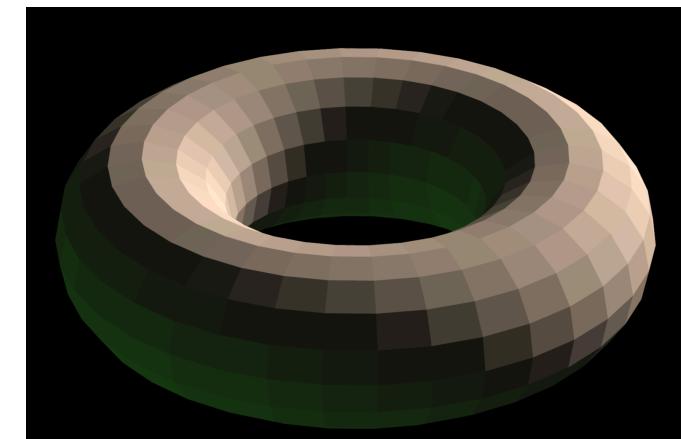
- Se debe seleccionar un punto cualquiera p de cada polígono, típicamente se usa un vértice, pero podría ser cualquier otro.
- Se usa la normal al polígono n_p .
- Se calcula el vector al observador v en p .

Las desventajas son:

- Puede no ser deseable que se aprecien los polígonos del modelo.
- Produce discontinuidades en el brillo de los pixels en las aristas.
- No es realista si el tamaño del polígono es grande en comparación con la distancia que lo separa al observador, en proyección perspectiva y/o brillos pseudo-especulares.

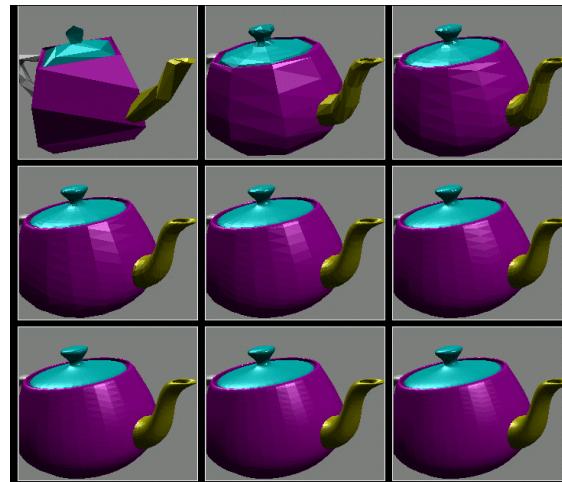
Resultados del sombreado plano.

Aquí vemos un objeto curvo aproximado con caras planas y visualizado con sombreado plano (MIL difuso).



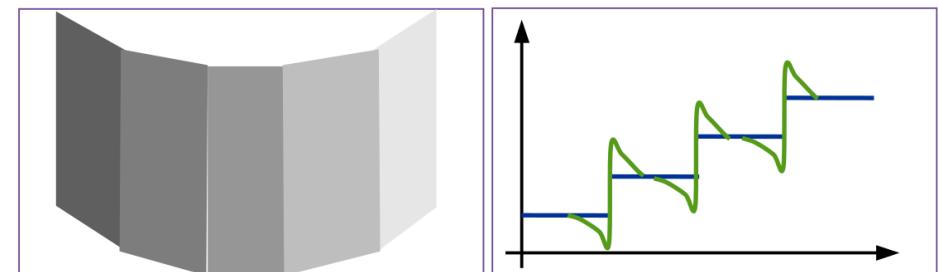
Resultados del sombreado plano

Aquí se observa la tetera, con sombreado plano, a distintas resoluciones. En este caso el MIL tiene una componente pseudo-especular no nula.



Bandas Mach

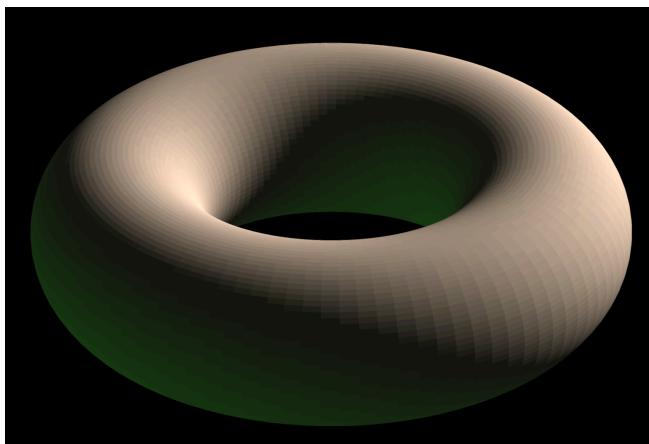
La **Bandas Mach** son una ilusión visual producida por la *inhibición lateral de las neuronas de la retina*, que es un mecanismo desarrollado evolutivamente para resaltar el contraste en aristas entre colores planos:



si no se quiere modelar un objeto formado realmente por caras planas, esta forma de visualizar produce resultados pobres. En algunos casos (objetos hechos de caras planas, iluminación puramente difusa) puede ser muy eficiente y realista.

Ejemplo de bandas Mach

En este objeto las bandas Mach son fácilmente apreciables:



Subsección 2.3.
Sombreado en los vértices.

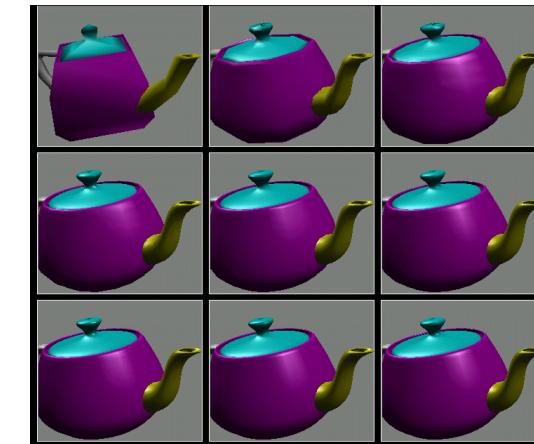
Sombreado en los vértices

En esta modalidad (*vertex shading*), el MIL se evalúa una vez en cada vértice del modelo.

- La posición p coincide con la posición del vértice.
- Si la malla de polígonos aproxima un objeto curvo, la normal n_p puede calcularse como el promedio de las normales de los polígonos adyacentes al vértice.
- La evaluación del MIL produce un color único para cada vértice
- Los valores en los vértices se usan como valores extremos para interpolar los colores de los pixels donde se proyecta el polígono
- La eficiencia en tiempo es parecida al sombreado plano.
- Los resultados son muchas veces más realistas que con sombreado plano.
- Pueden persistir problemas de bandas Mach y poco realismo.

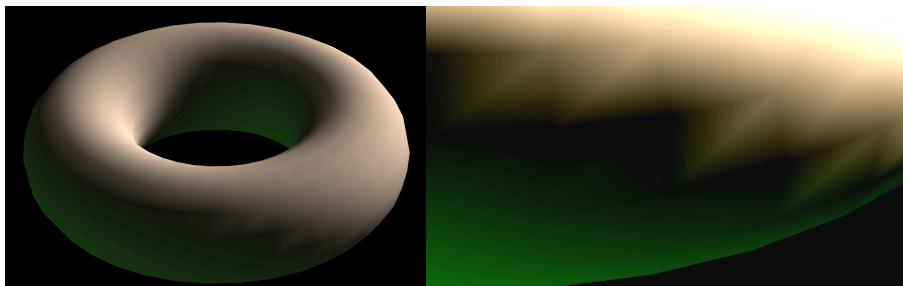
Pérdida de zonas brillantes

Los resultados mejoran, pero puede haber problemas de pérdida de zonas brillantes (componente pseudo-especular), en modelos con pocos polígonos que aproximan objetos curvos:



Discontinuidades en la derivada

A veces pueden aparecer problemas parecidos a las bandas Mach, en este caso por exageración en la retina de las discontinuidades de primer orden (cambios bruscos en la pendiente de la iluminación)



(a la derecha aparece una ampliación, con el brillo y contraste aumentado)

Sombreado en los píxeles.

En esta modalidad (*pixel shading*), el MIL se evalúa en cada pixel del viewport en el que se proyecta un polígono

- Requiere interpolar las normales asociadas a los vértices.
- Es computacionalmente más costoso que los anteriores, pero no cuando el número de polígonos visibles es del orden del número de pixels del viewport (o superior).
- Produce resultados de más calidad, hay muchos menos defectos por discontinuidades.
- Los resultados son más realistas incluso con pocos polígonos.
- La evaluación del MIL es la última etapa del cauce. Es decir, la evaluación del MIL se hace en cada instancia del *fragment shader*

Subsección 2.4.
Sombreado en los píxeles.

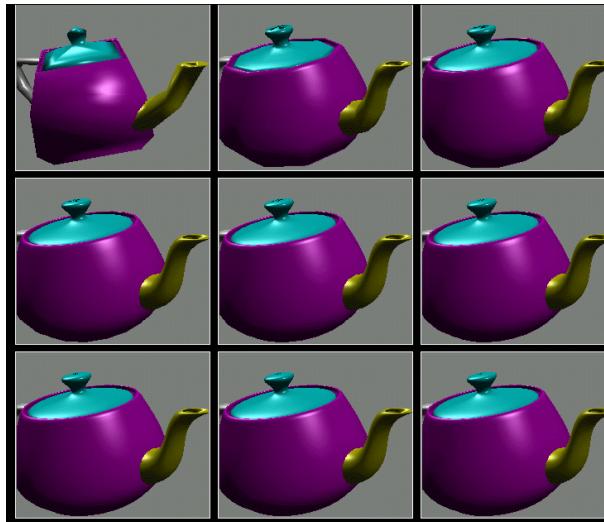
Ejemplo de sombreado

Esta imagen se ha creado con sombreado en los pixels:



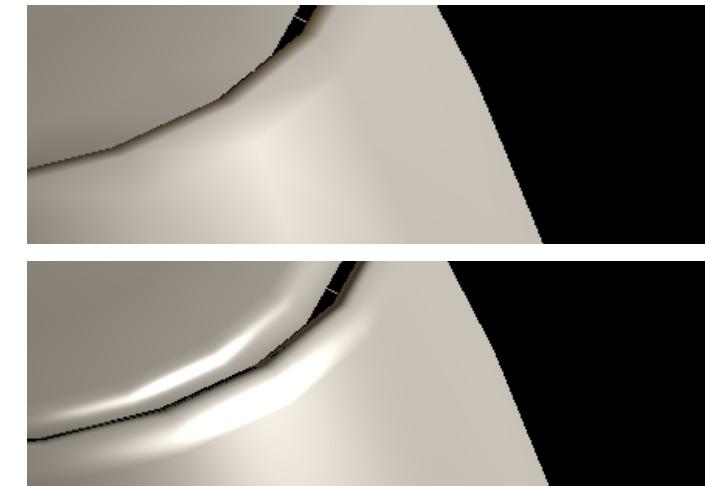
Reproducción de zonas de brillo

Con este sombreado se reproducen los brillos incluso a baja resolución:



Comparación de sombreado vértices y píxeles

Sombreado de vértices (arriba) y de píxeles (abajo), en iguales condiciones de iluminación, observador y atributos material:



Sección 3.

Luces, materiales y texturas en Godot.

1. Luces en Godot.
2. Materiales y texturas en Godot.

Subsección 3.1.

Luces en Godot.

Tipos de fuentes de luz en Godot

En Godot, las fuentes de luz emiten luz que se refleja en las superficies (con materiales con iluminación activada) y determina el color de los pixels. La luz puede provenir de varios tipos de fuentes en una escena:

1. Nodos del arbol de escena de tipo *fuente de luz*: son nodos de las clases `DirectionalLight3D`, `OmniLight3D` y `SpotLight3D`.
2. Luz ambiental en proveniente del entorno o fondo de la escena (un objeto de tipo `Environment`)
3. Luz reflejada indirecta, precalculada en una serie de puntos (usando la clase `ReflectionProbe`).
4. Luz reflejada indirecta, calculada usando técnica de *Iluminación global* (algoritmos implementados en las clases `LightmapGI`, o `VoxelGI` o `SDFGI`).
5. Desde la superficie de los objetos, cuando tienen definido un material con *color de emisión* y además están habilitada la iluminación indirecta en el espacio de pantalla (una aproximación a la iluminación global)

Veremos los dos primeros tipos de fuentes de luz.

Propiedades de las luces (clase `Light3D`)

Las propiedades comunes a todos los tipos de fuentes de luz son:

`light_color`: color RGB de la luz emitida por la fuente (tipo `Color`).

`light_energy`: es un valor `float` que multiplica a `light_color` (1.0 por defecto)

`shadow_enabled`: si es `true`, la luz produce sombras arrojadas (es decir, no ilumina puntos desde los que la luz no es visible). Si vale `false`, todos los puntos de las superficies (cuya normal está de cara a la luz) son iluminados.

`distance_fade_enabled`: para luces puntuales o spot, indica si hay una distancia a partir de la cual la luz no ilumina los puntos. Por defecto es `false`. Si se pone a `true`, hay otras propiedades `float` que controlan el efecto:

`distance_fade_begin`: distancia a partir de la cual la luz empieza a disminuir.

`distance_fade_length`: distancia adicional (a partir de `distance_fade_begin`) en la que la luz se atenúa hasta cero.

Nodos del árbol de tipo *fuente de luz*

En Godot se pueden situar, en un árbol de escena, nodos de tipo *fuente de luz*. La clase para esto es `Light3D`, derivada de `VisualInstance3D`, a su vez derivada de `Node3D`, es decir,

- Cada fuente tiene asociado su propio marco de coordenadas.
- Para usar una fuente de luz es necesario añadir un nodo `Light3D` a un árbol de escena.

Para cada tipo de nodo fuente de luz, existe una clase derivada de `Light3D`:

Clase `DirectionalLight3D`: luz direccional, como la luz del sol. La dirección de la fuente de luz es siempre el eje Z de su marco de coordenadas.

Clase `OmniLight3D`: luz puntual, que emite luz en todas las direcciones desde un punto, en concreto el origen de su marco de coordenadas.

Clase `SpotLight3D`: luz de tipo foco, que emite desde un punto (el origen del marco), con una distribución simétrica alrededor del eje Z del marco.

Luces direccionales (clase `DirectionalLight3D`)

Un nodo de tipo `DirectionalLight3D` representa una luz **direccional**, la cual ilumina cualquier punto de una superficie desde una dirección igual para todos los puntos:

- Por defecto dicha dirección es la rama positiva del eje Z (el vector hacia la fuente de luz, `l`, es $(0, 0, 1)$ para cualquier punto de la escena)
- Se puede modificar la dirección aplicando transformaciones de rotación al nodo (su posición es irrelevante).
- Por ejemplo, para apuntar a una dirección con coordenadas esféricas (θ, φ) , se puede usar la rotación:

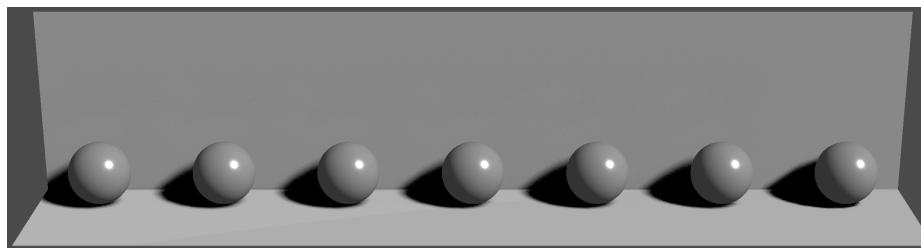
1. Rotar θ radianes alrededor del eje horizontal X (θ es el ángulo de `l` con el plano horizontal, negativo si es hacia arriba).

2. Rotar φ radianes alrededor del eje vertical Y.

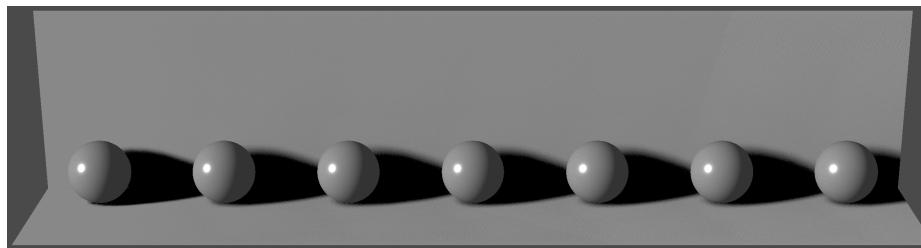
Se puede hacer directamente en el editor, o bien con GDScript usando los métodos `rotate_x` y `rotate_y` de la clase `Node3D`.

Ejemplo de luz direccional

Fuente de luz direccional, rotada con $\theta = -40^\circ$ y $\varphi = 60^\circ$



Fuente de luz direccional, rotada con $\theta = -20^\circ$ y $\varphi = -65^\circ$



Sesión 8: Texturas, sombreado y materiales

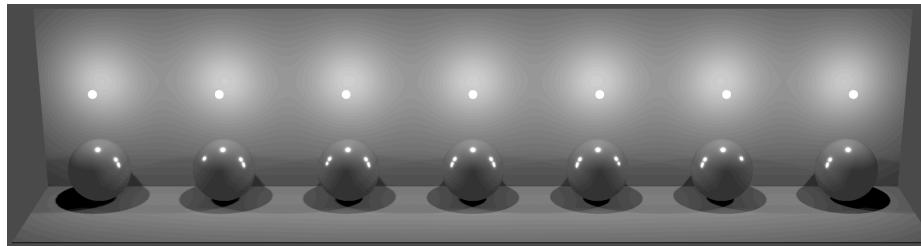
Created 2025-12-01

Page 61 / 87.

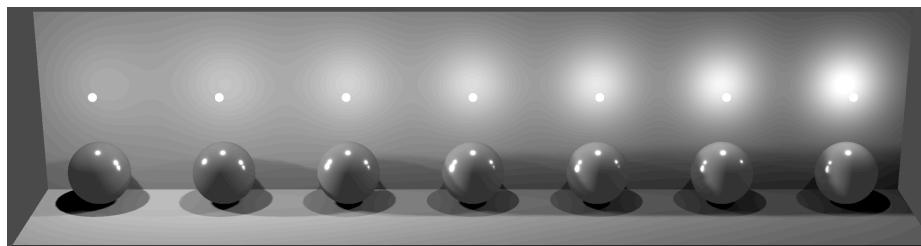
3. Luces, materiales y texturas en Godot..
3.1. Luces en Godot..

Ejemplo de luces puntuales

Ejemplo de luces puntuales, con atenuación fija $e = 1.5$,



Atenuación desde $e = 0$ (izquierda) hasta $e = 3.5$ (derecha):



Luces puntuales (clase `OmniLight3D`)

Un nodo de tipo `OmniLight3D` representa una **luz puntual**, la cual ilumina un punto \hat{p} en una superficie desde un punto \dot{q} fijo del espacio, asociado a la fuente.

- El punto \dot{q} es el **origen del marco de coordenadas del nodo**. Se puede cambiar asignando `position`, en un script o en el editor.
- La intensidad no depende de la dirección $\vec{r} = \dot{q} - \hat{p}$ pero sí puede depender de la distancia $r = \|\dot{q} - \hat{p}\|$. Se usa una propiedad `float`, con identificador `omni_attenuation` y llamada **atenuación**, si su valor es e , la intensidad será proporcional a:

$$g(r) = \frac{1}{r^e}$$

- Un valor $e = 0$ hace que la luz no se atenúa con la distancia. El valor $e = 2$ es **físicamente realista**, ya que en la naturaleza la luz se atenúa con el cuadrado de la distancia al punto de donde emana. A mayores valores del exponente e , más brillantes son las superficies cercanas a la fuente, y más oscuras las lejanas.

Sesión 8: Texturas, sombreado y materiales

Created 2025-12-01

Page 61 / 87.

Sesión 8: Texturas, sombreado y materiales

Created 2025-12-01

Page 62 / 87.

3. Luces, materiales y texturas en Godot..
3.1. Luces en Godot..

Luces de tipo foco (spot) (clase `SpotLight3D`)

Las luces de tipo **foco** (o **spot**) son una clase de luces puntuales que emiten luz desde un punto \dot{q} fijo hacia cualquier otro punto \hat{p} , con una intensidad que:

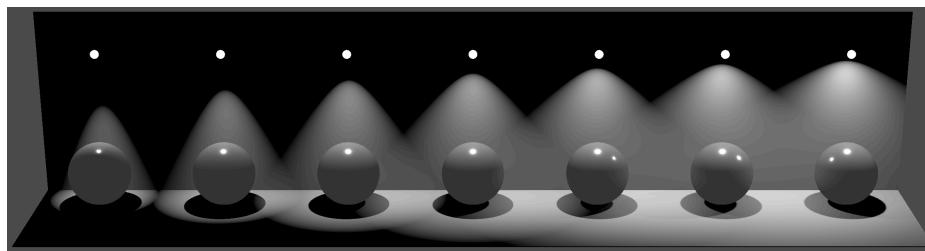
- Al igual que las luces puntuales, puede depender de una potencia de la distancia $r = \|\dot{p} - \dot{q}\|$, en este caso la propiedad se llama `spot_attenuation`, y se interpreta igual que `omni_attenuation` en `OmniLight3D`.
- Además, la intensidad depende del ángulo $\theta = \arccos(\hat{e} \cdot \hat{r})$ entre:
 - ▶ $\hat{e} \equiv$ versor Z negativo del marco de coordenadas de la fuente, llamado **eje del foco**, es decir, el vector con coordenadas locales $(0, 0, -1)$
 - ▶ $\hat{r} \equiv (\dot{p} - \dot{q})/r$, el versor desde \dot{q} hacia \dot{p} .

en concreto, la intensidad será nula cuando $\theta > \theta_{\max}$, donde el ángulo θ_{\max} se define con la propiedad `spot_angle` (llamada **apertura del foco**), que es un `float` con unidades en grados.

Ejemplo de luces de tipo foco

Aquí vemos un ejemplo de varias luces de tipo foco, en las cuales el eje \hat{e} tiene coordenadas de mundo $(0, -1, 0)$ (apunta en vertical hacia abajo), para ello se ha rotado cada nodo fuente un ángulo de -90° entorno al eje X (lo cual lleva el versor Z negativo hasta el versor Y negativo).

El ángulo de apertura va desde 20° a la izquierda hasta 60° a la derecha, y la atenuación es 1.0:



Mapa de entorno usando un panorama

Los mapas de entorno se pueden asociar a una escena mediante código o en el editor. Aquí vemos un ejemplo en el cual se asocia una imagen con un *panorama equirectangular* a una escena.

Para ello se añade un nodo **WorldEnvironment** y se usa un archivo de imagen **.jpg**. En GDScript se puede hacer asociando este script al nodo raíz de la escena:

```
func _ready() :  
  
    ## crear el objeto 'world environment' con el fondo en 'imagen.jpg'  
    var we := WorldEnvironment.new()  
    we.environment = Environment.new()  
    we.environment.background_mode = Environment.BG_SKY  
    we.environment.sky = Sky.new()  
    we.environment.sky.sky_material = PanoramaSkyMaterial.new()  
    we.environment.sky.sky_material.panorama = CargarTextura("imagen.jpg")  
  
    ## añadir el 'world environment' al árbol de escena  
    add_child( we )
```

Iluminación desde mapas de entorno

En Godot los objetos pueden recibir iluminación ambiental desde un mapa de entorno (un objeto de tipo **Environment** asociado a la escena).

- El árbol de escena puede tener un nodo de tipo **WorldEnvironment** con diversos parámetros de configuración, entre ellos está la propiedad **environment** que referencia a un objeto de tipo **Environment**.
- Los objetos de tipo **Environment** pueden incluir un modo en el cual el entorno es de tipo **cielo**, referenciado en la propiedad **sky** (de la clase **Sky**).
- Un objeto de tipo **Sky** tiene la propiedad **sky_material** de la clase **Material**, la propiedad es una referencia a un objeto de uno de estos tipos:
 - ▶ **PanoramaSkyMaterial**: usa una imagen panorámica de $360^\circ \times 180^\circ$. Se almacena en la propiedad **panorama** del material, de tipo **Texture2D**.
 - ▶ **ProceduralSkyMaterial**: degradados de color
 - ▶ **PhysicalSkyMaterial** : basado en física
 - ▶ **ShaderMaterial**: implementado en un shader.

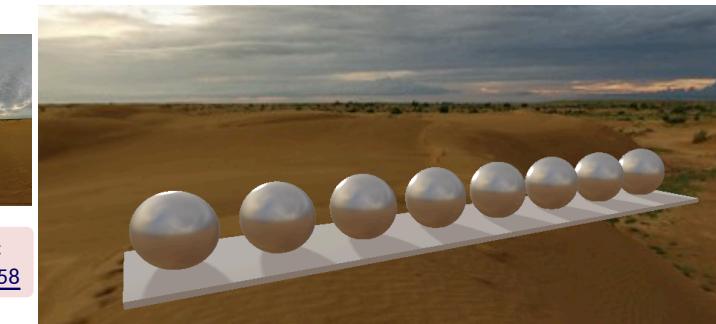
Mapa de entorno usando un panorama

Los mapas de entorno iluminan las superficies de los objetos en la escena, su luz se refleja en las superficies tanto difusas como especulares.

Aquí vemos un ejemplo (a la derecha) de una escena con varias esferas, iluminadas con una **fuente direccional** (ver las sombras arrojadas) y además con un mapa de entorno basado en un *panorama equirectangular* (es la imagen a la izquierda).

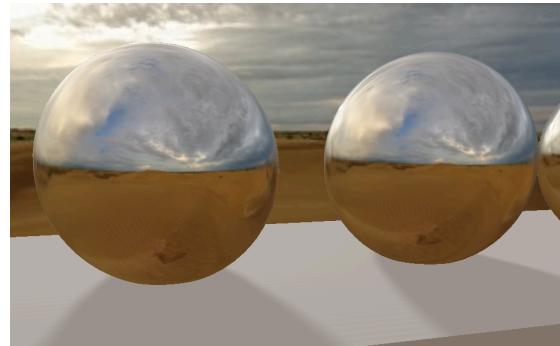


Imagen de Alexandre Duret-Lutz:
flickr.com/photos/gadl/595353758



La reflexión pseudo-especular o especular desde una superficie (iluminada por un mapa de entorno) únicamente incluye el mapa en sí, no los otros objetos.

En este ejemplo, las esferas especulares reflejan el mapa de entorno, pero no reflejan las otras esferas ni el suelo:



Para tener en cuenta estas reflexiones tendríamos que usar *Ray tracing*.

Materiales

En los *engines* de videojuegos y las aplicaciones gráficas 3D en general, un **material** es un objeto con una serie de parámetros que determinan como refleja la luz la superficie de un objeto (3D usualmente, pero también puede ser 2D). En concreto, un *material* encapsula:

- Los **tipos de BRDFs** asociadas al material, incluyendo los **pesos** con los que se combinan cuando hay varias (difusa, especular perfecta, metálica, etc)
- Los **parámetros de cada BRDF**, por ejemplo, el **exponente de brillo** del modelo de Blinn-Phong, las **rugosidades** del modelo GGX, el **índice de refracción** para refracción y reflexión en dieléctricos, etc...
- El color base de la superficie (a menudo llamado *albedo*), que puede ser un **color plano único** o puede ser una **textura de color** (ya sea almacenada en un archivo o generada proceduralmente).
- Texturas que controlan parámetros adicionales, por ejemplo texturas para controlar los pesos citados arriba (que así pueden variar punto a punto), o bien texturas para **mapas de perturbación de la normal**.

Subsección 3.2. Materiales y texturas en Godot.

Materiales en Godot

En Godot, los materiales se representan con objetos de la clase **Material**, que es una clase base abstracta. Existen diversas clases derivadas de **Material**, entre las cuales las más usadas son:

- **BaseMaterial3D**: clase base abstracta para los materiales de las superficies de los objetos de tipo malla de triángulos. Define la mayoría de las propiedades relevantes. No se puede usar directamente (es una clase abstracta), sino que se usan instancias de su clase derivada **StandardMaterial3D**.
- **ShaderMaterial**: material definido mediante un shader personalizado (escrito en el lenguaje de shaders de Godot). Permite un control total sobre la apariencia del material.
- Los materiales para entornos de tipo *cielo* (clase **PanoramaSkyMaterial**, **ProceduralSkyMaterial**, **PhysicalSkyMaterial**), no se usan para mallas de triángulos.

La clase StandardMaterial3D. Propiedades (1/2)

La clase **StandardMaterial3D** sirve para definir características de los materiales, y se implementa mediante un *fragment shader* interno específico. Las propiedades de esta clase son heredadas de su clase base abstracta **BaseMaterial3D**. Destacamos:

- **albedo_color**: color base del material (tipo **Color**).
- **albedo_texture**: textura con colores que multiplican a **albedo_color** por distintos factores en cada punto, permite variar el color base del material de un punto a otro (tipo **Texture2D**).
- **metallic**: (tipo **float**), entre 0.0 y 1.0 determina define el carácter metálico del material.(*metallicidad*).
- **metallic_specular**: (tipo **float**) entre 0 y 1 que define la intensidad del brillo pseudo-especular en materiales no metálicos.
- **roughness**: (tipo **float**), entre 0.0 y 1.0 que define la rugosidad del material

(continúa en la siguiente diapositiva)

El modelo de iluminación de Godot

El color reflejado I (antes de mult. por el color de la luz) puede escribirse así:

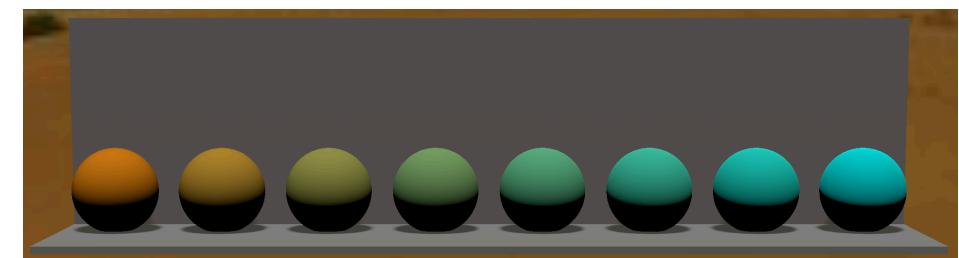
$$I = e + a + (1 - m)(b \cdot d + s \cdot p_\alpha) + m \cdot b \cdot p_\alpha$$

- e ≡ color en **emission** (**color emitido**)
- a ≡ **color ambiental** reflejado, depende del mapa de entorno.
- m ≡ factor real en **metallic** (**factor de metalicidad**)
- s ≡ factor real en **metallic_specular** (**factor pseudo-especular**)
- b ≡ **color base**, es **albedo_color**, multiplicado por **albedo_texture** (si existe).
- d ≡ **reflectividad difusa** según el modelo de Burley (o de Lambert).
- p_α ≡ **componente pseudo-especular**, es la suma de la reflectividad de una BRDF pseudo-especular (por defecto la BRDF GGX anisotrópica), más el color reflejado del mapa de entorno (si existe). Está afectada por la rugosidad α
- α ≡ valor real en **roughness** (**rugosidad**), va desde 0 (especular perfecto, como un espejo), hasta 1 (superficie muy rugosa, casi mate).

La clase StandardMaterial3D. Propiedades (2/2)

Más propiedades relevantes de la clase **StandardMaterial3D** (**BaseMaterial3D**):

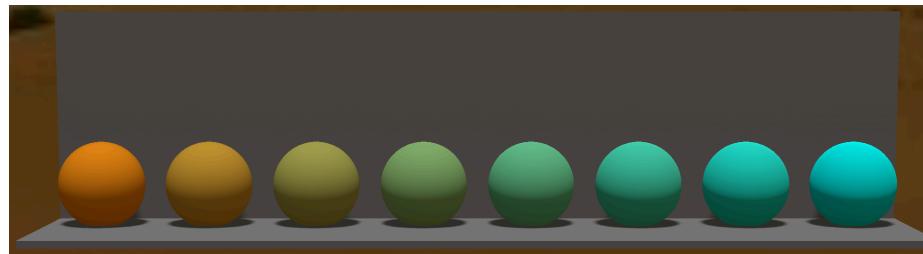
- **emission_enabled**: si es **true**, la superficie del objeto emite luz por sí mismo, si es **false** no emite luz. En la práctica este color se suma al reflejado, pero no afecta a otros objetos si no se activa algún método avanzado de iluminación.
- **emission**: valor **Color** con la luz emitida, cuando **emission_enabled** es **true**.
- **transparency**: modo de transparencia, permite desactivar transparencias (si se usa el valor **TRANSPARENCY_DISABLED**) o activarlas, si se activa, entre otras opciones, se puede ligar a la componente **alpha** del color base (usando el modo **TRANSPARENCY_ALPHA**).
- **disable_ambient_light**: si es **true**, el material no recibe luz ambiental del mapa de entorno.



Aquí vemos una serie de esferas puramente difusas, con el color base b (**albedo_color**) variando desde naranja (1.0, 0.6, 0.1) hasta verde (0.0, 1.0, 1.0), todo iluminado con una luz direccional, y con a a cero (**disable_ambient_light** puesto a **true**).

La componente ambiental y la difusa

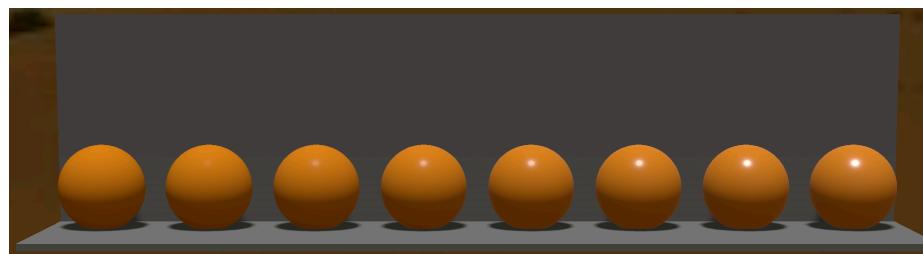
La componente ambiental a puede habilitarse si hay un mapa de entorno en la escena, en ese caso, las mismas esferas de la imagen anterior tendrán una cantidad de luz reflejada desde el entorno, lo cual es relevante en las partes que no dan a la fuente de luz (y donde d , por tanto, es nulo).



Aquí vemos la misma escena con `disable_ambient_light` a `false`, de modo que las esferas reciben luz ambiental desde el mapa de entorno.

Combinación difusa y pseudo-especular

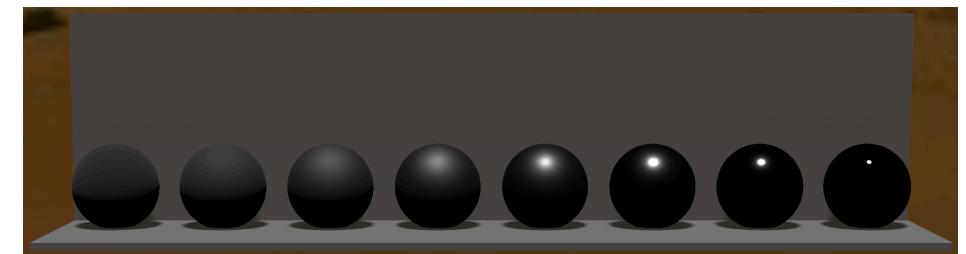
La componente pseudo-especular puede combinarse con la difusa para simular un material brillante (no afectado por el color base b) sobre un substrato difuso (afectado por b). En esta imagen variamos s desde 0 a la izquierda, hasta 0.6, con rugosidad $\alpha = 0.3$. El color base b se mantiene a naranja ($1.0, 0.6, 0.1$)



Se ha habilitado la luz ambiental.

La componente pseudo-especular

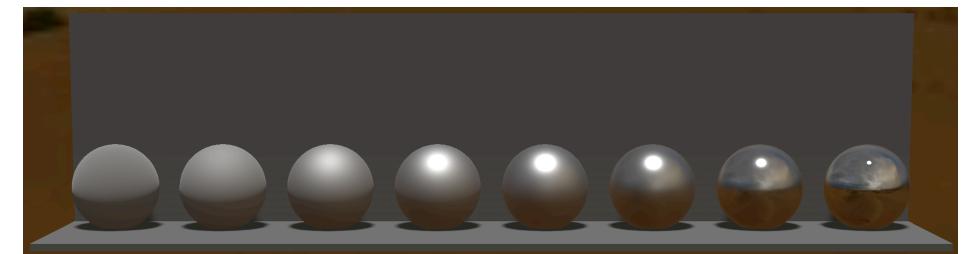
El factor s (propiedad `metallic_specular`) controla la cantidad de luz reflejada por la componente pseudo-especular. Este producto no está multiplicada por el color base, así que solo depende del color de la fuente de luz. Se puede observar haciendo $m = 0$ y $b = e = a = (0, 0, 0)$. Puesto que $I = s \cdot p_\alpha$, esta componente depende de la rugosidad α .



En esta serie vemos la componente pesudo-especular $s \cdot p_\alpha$, con s puesto a 1 y la rugosidad α bajando desde 1 hasta 0 (de izquierda a derecha). No hay luz ambiental, $a = (0, 0, 0)$.

Materiales metálicos

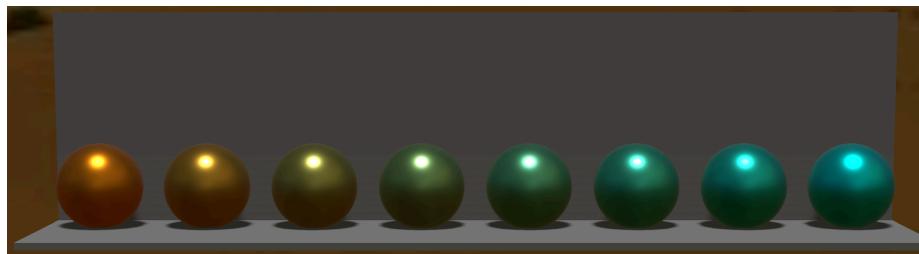
Para lograr que la componente pseudo-especular esté afectada por el color base b , podemos hacer $m = 1$ y entonces no se usa el valor s , ya que $I = m \cdot b \cdot p_\alpha$. Se consigue un aspecto metálico (a partir de cierto α), más o menos pulido, según α . Para $\alpha = 0$, se tiene un material reflectante como un espejo.



En esta serie hacemos $b = (1, 1, 1)$ y variamos la rugosidad desde $\alpha = 1$ a la izquierda hasta 0 a la derecha. Se tiene en cuenta la luz del entorno. El aspecto resultante es un aspecto metálico

Materiales metálicos afectados del color base

En esta serie se ha mantenido $m = 1$ y $\alpha = 0.3$. Aquí de nuevo se varía el color base b desde naranja a verde.



Este tipo de materiales puede simular metales como el oro u otros metales tintados.

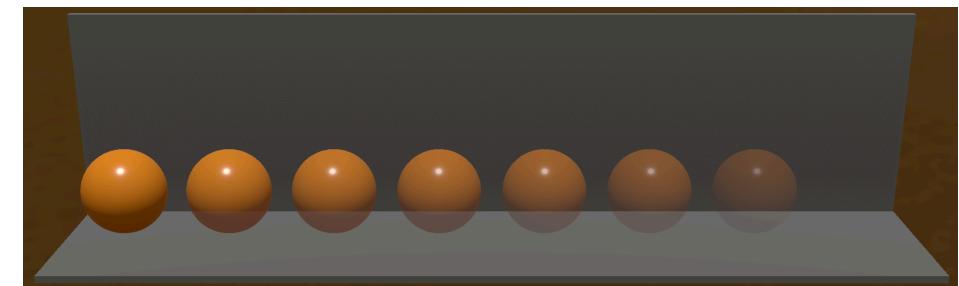
Objetos parcialmente transparentes

Además de los materiales, los nodos (de clases derivadas de `GeometryInstance3D`, es decir, todas las mallas, como `MeshInstance3D` u otras):

- Tienen una propiedad llamada `transparency`, es un `float` entre 0 y 1.
- Define la transparencia del objeto, pero se interpreta al revés que la componente `alpha` del albedo, ya que el valor 0 en `transparency` implica **totalmente opaco** y el valor 1 **totalmente transparente**.
- Este valor se compone con la transparencia del albedo.
- Los nodos con un valor de `transparency` mayor que 0 (es decir, parcialmente transparentes) **sí arrojan sombras**.
- Esta transparencia se ignora si se genera la aplicación en el modo de render de `compatibilidad` o para dispositivos móviles (solo se tiene en cuenta con el modo de render `Forward +`)

Materiales parcialmente transparentes

En Godot se puede hacer que un material sea parcialmente transparente, para ello se usa la propiedad `transparency` de la clase `StandardMaterial3D`. Si se activa la transparencia, el material puede usar la componente `alpha` del color base (`b` debe ser una tupla RGBA con 4 `float`) para definir el grado de transparencia, es un valor entre 0.0 (totalmente transparente) y 1.0 (totalmente opaco).



Aquí se la componente `alpha` de `b` varía desde 1 a la izquierda hasta 0 a la derecha. Con este modo, **no se tiene en cuenta la refracción, ni se proyectan sombras**.

Texturas en Godot. La clase `Texture2D`

En Godot, la clase `Texture2D` representa una imagen, con un rectángulo bidimensional con un color (albedo) asociado cada punto del mismo. Godot incorpora numerosas clases derivadas de `Texture2D`, podemos destacar estas:

- **ImageTexture**: usa una imagen cargada desde un archivo (PNG, JPG, etc) o creada en memoria. Es el tipo de textura **más frecuentemente usado**.
- **ViewportTexture**: se usan los pixels de un nodo `Viewport` (una ventana o una parte de una ventana sobre la que se visualiza una imagen).
- **GradientTexture2D**: degradado de color.
- **NoiseTexture2D**: colores calculados usando algoritmos de generación de ruido (ruido procedural) con un objeto tipo `Noise` (ruido Perlin, Cellular, Simplex).
- **CameraTexture**: usa la imagen capturada por una cámara del dispositivo que ejecuta la aplicación.
- **AtlasTexture**: textura que usa una región rectangular de otra textura (útil para optimizar el uso de memoria).

Asociación de materiales a objetos

Cualquier nodo derivado de **GeometryInstance3D** (por ejemplo, **MeshInstance3D**):

- tiene una propiedad llamada **material_override**, que es una referencia a un objeto de la clase **Material** (o a una clase derivada).
- adicionalmente, se puede definir su propiedad **material_overlay**, es un material que se *superpone* al material definido en **material_override**

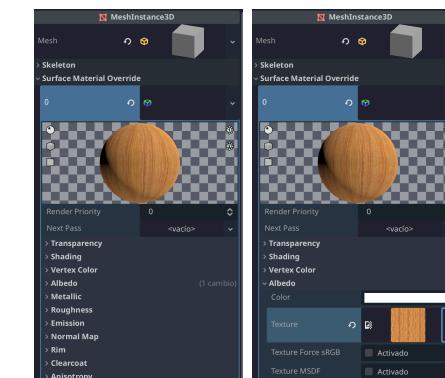
Si un nodo no tiene por defecto ningún material asignado, en ese caso se usa el material asociado a las mallas del nodo. En Godot,

- Los objetos de tipo **Mesh** (**ArrayMesh**, **IndirectMesh** y **PrimitiveMesh**) pueden tener una o varias *superficies* (mallas), y cada una puede tener asignado un material distinto.
- Ese material se puede consultar y cambiar usando los métodos **surface_get_material(i)** y **surface_set_material(i, m)** de la clase **Mesh**

Fin de transparencias.

Asignación de materiales y texturas en el editor

El editor de Godot puede usarse para crear un material específico para un nodo (de tipo derivado de **GeometryInstance3D**), y asignar texturas a ese material.



Aquí vemos dos capturas del panel de propiedades de un nodo tipo **MeshInstance3D**, en la primera se ven las propiedades dentro del **Surface Material Override**, y en la segunda igual, pero con las propiedades de textura desplegadas (en el apartado **Albedo** ⇒ **Texture**)

Informática Gráfica.

Sesión 9: Interacción.

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

Sección 1. Introducción

Índice

Introducción	3
Eventos en Godot	17
Interfaz de usuario y señales en Godot.	39
Selección	67

1. Introducción.

Sistemas Gráficos Interactivos

Un **Sistema Gráfico Interactivo** (SGI) es un sistema software cuya respuesta a cada acción del usuario

- ocurre (por lo general) en un tiempo corto (del orden de décimas de segundo como mucho) desde dicha acción del usuario.
- se presenta al usuario en forma de visualización gráfica 2D o 3D

Un sistema SGI, por lo general, mantiene en memoria una estructura de datos (un modelo) y ejecuta un ciclo infinito, en cada iteración

1. espera o detecta una acción del usuario.
2. obtiene los datos que caracterizan dicha acción.
3. modifica el estado del modelo según dichos datos.
4. visualiza una nueva imagen obtenida a partir del nuevo estado del modelo

Interactividad

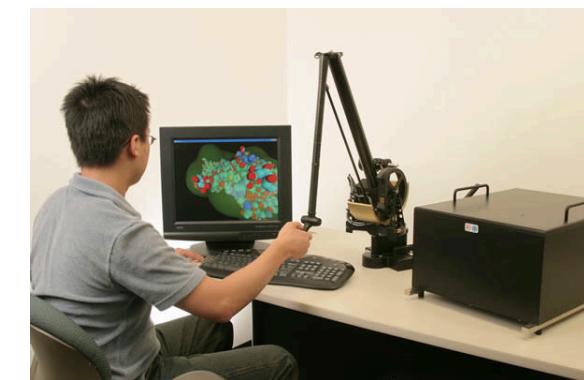
La incorporación de interactividad permite realizar aplicaciones que respondan ágilmente a las acciones de los usuarios y les ofrezcan retroalimentación sobre el efecto de dichas acciones.

La mayor parte de los sistemas gráficos son interactivos. Es esencial en:

- Videojuegos (*Videogames*) y Juego Serios (*Serious Games*).
- Sistemas de Diseño Asistido por Ordenador (CAD: *Computer Aided Design*)
- Sistemas de Realidad Virtual (VR: *Virtual Reality*)
- Sistemas de Realidad Aumentada (AR: *Augmented Reality*)
- Simuladores de aprendizaje (de conducción, de aviones, de barcos, etc...)

Dispositivos de entrada y salida

En un SGI el usuario debe disponer de al menos un dispositivo de entrada (p.ej. teclado, ratón) y un dispositivo de visualización (típicamente un monitor).



Hay otros dispositivos de entrada: tabletas digitalizadoras, sistemas de posicionamiento 3D.

Sistemas interactivos y de tiempo real

Los sistemas gráficos interactivos no siempre son **sistemas de tiempo real**:

- En un sistema interactivo se requiere que el retardo (latencia) entre la acción del usuario y la respuesta del sistema sea suficientemente pequeño como para que el usuario perciba una relación de causa-efecto.
- No obstante, eventualmente la respuesta puede demorarse algo más.
- En un **sistema de tiempo real** la latencia debe ser menor o igual que un tiempo máximo de respuesta prefijado en las especificaciones del sistema. Un retraso superior a ese límite se considera un fallo del sistema.

Algunas veces, sin embargo, se imponen **requerimientos de tiempo real estrictos**:

- Simuladores, sistemas de realidad virtual, gemelos digitales para vehículos o aviones, donde es necesario reproducir los tiempos de respuesta reales del sistema.
- Videojuegos de alta gama, donde se requiere una latencia mínima para una experiencia de usuario óptima.

Realimentación: utilidad

La realimentación es el mecanismo mediante el cual el sistema da información al usuario útil para permitir al usuario

- conocer más fácilmente el estado interno del sistema.
- ayudar a decidir la siguiente acción a realizar.

La información de realimentación que el sistema genera en cada momento depende lógicamente del estado del sistema y de la información previamente entrada por el usuario. Se puede usar con diferentes fines específicos:

- mostrar el estado del sistema
- como parte de una función de entrada
- para reducir la incertidumbre del usuario

Funciones de entrada en un SGI

Un sistema gráfico interactivo necesita normalmente funciones de entrada usuales en todo tipo de aplicaciones, p.ej:

- Entrada de una cadena de texto.
- Entrada de un valor numérico (directamente o con deslizadores, por ejemplo).
- Selección de un dato en una lista.

Además en un SGI se suelen necesitar otros tipos de entrada más específicos, por ejemplo, en un sistema CAD 3D podemos encontrar, entre otras, estas funciones:

- Lectura de posiciones 3D (selección de unas coordenadas específicas en el espacio de coordenadas del mundo)
- Selección de una componente de un modelo jerárquico 3D
- Entrada de los ángulos de rotación que determinan la orientación de un objeto.

Dispositivos físicos de entrada

El usuario introduce la información por medio de **dispositivos físicos de entrada**. Estos pueden ser

- de propósito general: p.ej. el teclado, o
- específicos para datos geométricos: p.ej. digitalizador.

Podemos clasificar los dispositivos de entrada gráfica atendiendo a la información que generan de forma directa. Esta puede ser:

- **Posiciones 2D**: tableta digitalizadora, lápiz óptico, pantalla táctil.
- **Posiciones 3D**: digitalizador, tracking.
- **Desplazamientos 2D**: ratón, trackball, joystick.
- **Imágenes o videos**: cámaras de fotografía o video.

Dispositivos lógicos de entrada

Un **dispositivos lógico de entrada** es una componente software que usa uno o varios dispositivos físicos de entrada para producir información de más alto nivel o mas elaborada, obtenida a partir de los datos recibidos directamente de los dispositivos físicos (o indirectamente de otros dispositivos lógicos). Ejemplos:

- **Puntero del ratón**: permite obtener coordenadas en pantalla, calculadas a partir de los desplazamientos físicos del ratón y del estado de sus botones.
- **Selector de componentes (Picker)**: permite seleccionar un componente de un modelo 3D usando el puntero de ratón.
- **Detección de gestos** a partir de una secuencia de vídeo en tiempo de real de las manos, se pueden reconocer determinados gestos (previamente definidos) y generar eventos de alto nivel asociados a dichos gestos. Por ejemplo: *mano abierta, puño cerrado, dedo índice apuntando*.

Lectura de datos dispositivos de entrada: modos de entrada.

Un **modo de entrada** es un método que usa una aplicación para decidir cuando debe consultar los datos relacionados con el estado (y los cambios de estado) de un dispositivo físico o lógico.

- Distintos tipos de dispositivos pueden tener asociados distintos modos de funcionamiento.
- Algunos tipos de dispositivos se pueden usar con más de un modo de funcionamiento.

Veremos los tres modos básicos más frecuentes:

- **Modo de muestreo**: la aplicación consulta del estado actual en instantes arbitrarios.
- **Modo de petición**: la aplicación espera hasta que se produzca un cambio de estado.
- **Modo de cola de eventos**: la aplicación recibe una lista de cambios de estado no procesados aún.

Estado y eventos de un dispositivo

Los cambios de estado que ocurren en un dispositivo de entrada (físico o lógico) se denominan **sucesos o eventos**.

- El **estado** de un dispositivo en un instante es el conjunto de valores de las variables gestionadas por el driver del dispositivo, y que representan en memoria su estado físico. P.ej:
 - ▶ En un teclado: un vector de valores lógicos que indican, para cada tecla, si dicha tecla está pulsada o no está pulsada.
 - ▶ En un ratón: estado de los dos botones (dos lógicos) y posición actual en pantalla del cursor (dos enteros).
- Un evento tiene asociados ciertos datos:
 - ▶ Instante de tiempo en el que el cambio ha ocurrido o se ha registrado
 - ▶ Información sobre: el estado inmediatamente después del evento, y sobre como ha cambiado el estado respecto al anterior al evento.

Modo de petición.

Para evitar perder eventos relevantes, la aplicación puede usar el **modo petición (request)**:

- La aplicación hace una petición y espera a que se produzca determinado tipo de evento.
- Cuando se produce, la aplicación recibe datos del evento.

Ventaja/Desventajas

- Nunca se perderá el siguiente evento tras hacer una petición.
- Puede perderse un evento si no se hace una petición antes de que ocurra.
- Se puede perder mucho tiempo esperando (no se puede hacer otras cosas).

Ejemplo: en un teclado, esperar hasta que se pulse una tecla alfanumérica, y entonces saber de qué tecla se trata.

Modo de muestreo

Un dispositivo puede usarse **modo de muestreo (sample)**:

- El software del dispositivo mantiene en memoria variables que representan el estado actual del dispositivo.
- La aplicación puede consultar dichas variables en cualquier momento, sin esperar alguna.

Ventajas/Desventajas

- Es muy eficiente en tiempo y memoria, y simple.
- Requiere a la aplicación emplear tiempo de CPU en muestrear a una frecuencia suficiente como para no perderse posibles cambios de estado relevantes.
- No hay información de cuando ocurrió el último cambio de estado.

Ejemplo: en un teclado, array de valores lógicos que indica, para cada tecla, si está pulsada o levantada.

Modos cola de eventos

En el modo **cola de eventos**

- Cada vez que ocurre un evento, el software del dispositivo lo añade a una cola FIFO de eventos pendientes de procesar.
- La aplicación accede a la cola, extrae cada evento y lo procesa.

Ventajas:

- No se pierde ningún evento.
- La aplicación no está obligada a consultar con cierta frecuencia, ni antes de cada evento.
- La aplicación no pierde tiempo en esperas si es necesario hacer otras cosas (se funciona en modo asíncrono).

Ejemplo: en un teclado, acceder a la lista de pulsaciones de teclas, ocurridas desde la última vez que se consultó.

Sección 2.

Eventos en Godot

1. Tipos de eventos en Godot. La clase *InputEvent*.
2. Gestión de eventos.

2. Eventos en Godot.

2.1. Tipos de eventos en Godot. La clase *InputEvent*.

Eventos en Godot. La clase *InputEvent*

Godot ofrece la clase *InputEvent* (un tipo de *Resource*) para eventos de entrada:

- Un evento es un objeto con datos sobre un cambio de estado en un dispositivo físico o lógico.
- Para cada tipo de evento, existe una clase específica derivada de *InputEvent*
- Muchos eventos se originan en un *viewport* cuya ventana tiene el foco.
- El sistema operativo y/o el sistema de ventanas asocian los eventos a un *viewport* y lo redirigen a la aplicación Godot.
- Cualquier nodo en un árbol de escena puede tener asociados métodos para gestionar eventos de entrada de uno varios tipos.
- Cada uno de esos métodos recibe como parámetro un objeto con los datos del evento, y puede declarar el evento como *consumido* o no.
- Un evento puede ser gestionado por varios nodos, hasta que no hay más nodos que lo gestionen o bien hasta que sea consumido por el último de ellos.

Subsección 2.1.

Tipos de eventos en Godot. La clase *InputEvent*.

2. Eventos en Godot.

2.1. Tipos de eventos en Godot. La clase *InputEvent*.

Tipos de eventos en Godot

Las clases derivadas de *InputEvent* son:

- *InputEventFromWindow*: eventos originados en una ventana, un *viewport* dentro de una ventana, o en una pantalla táctil. Las subclases son:
 - ▶ *InputEventWithModifiers* : eventos de teclado, ratón o gestos en pantallas táctiles.
 - ▶ *InputEventScreenDrag* : eventos de arrastre de elementos del GUI.
 - ▶ *InputEventScreenTouch* : eventos de toque generados en pantallas táctiles
- *InputEventJoypadButton*: eventos de botones de mando de juego (*joypad*)
- *InputEventJoypadMotion*: eventos de movimiento de mando de juego.
- *InputEventAction*: eventos asociados a acciones definidas en el *mapa de entradas* de Godot.
- *InputEventShortcut*: eventos de atajos de teclado
- *InputEventMIDI*: eventos recibidos de dispositivos MIDI.

Eventos de teclado, ratón y gestos

Los eventos más comunes son los de teclado, ratón, o gestos en pantallas táctiles. Se usan las clases que derivan de `InputEventWithModifiers`:

- `InputEventKey` : eventos de pulsar o levantar teclas de un teclado
- `InputEventMouse` : eventos de ratón: desplazamiento del ratón, de su rueda, pulsar o levantar botones del ratón.
- `InputEventGesture` : eventos de gestos producidos en pantallas táctiles.

Estos eventos tienen propiedades con datos sobre el estado de ciertas teclas *modificadoras* en el momento de producirse el evento, a saber:

- `alt_pressed` : si la tecla Alt pulsada vale `true`
- `shift_pressed` : tecla Shift
- `control_pressed` : tecla Control
- `meta_pressed` : tecla Meta (Windows, Cmd, etc.)

Eventos de teclado (`InputEventKey`)

La clase `InputEventKey` tiene las siguientes propiedades importantes:

- `keycode` (tipo `Key`): código de la tecla pulsada o levantada, independiente del idioma del teclado.
- `key_label` (tipo `Key`): código de la tecla, dependiente del idioma del teclado.
- `unicode` (tipo `int`): código Unicode del carácter generado por la tecla (si lo hay)
- `pressed` (tipo `bool`): vale `true` si la tecla se ha pulsado, o `false` si se ha levantado
- `echo` (tipo `bool`): vale `true` si el evento es producido por una repetición automática al mantener la tecla pulsada.
- `location` (tipo `KeyLocation`): para teclas con dos copias en el teclado (como Shift o Control), indica si la tecla está en la parte izquierda o derecha del teclado.

Eventos de ratón (`InputEventMouse`)

La clase `InputEventMouse` es la clase base para eventos de ratón. Propiedades:

- `position` (de tipo `Vector2`), con la posición del ratón en el *viewport* (en unidades de pixels) en el momento del evento.
- `global_position` (de tipo `Vector2`), idem, pero para el *viewport* raíz.
- `button_mask` (`MouseButtonMask`): bits con estado de los botones.

Hay dos sub-clases de `InputEventMouse`:

- `InputEventMouseButton`: pulsar o levantar botones del ratón. Propiedades:
 - ▶ `button_index` : constante que identifica el botón pulsado o levantado (tipo `MouseButton`)
 - ▶ `pressed` : vale `true` si el botón se ha pulsado, o `false` si se ha levantado
- `InputEventMouseMove` : eventos de movimiento del ratón. Propiedad:
 - ▶ `relative` (tipo `Vector2`): desplazamiento en pixels del ratón desde la última posición registrada (el último frame).

Subsección 2.2.

Gestión de eventos.

Métodos de gestión de eventos

En Godot, el desarrollador puede escribir código que se ejecutará cuando se produzca algún evento de entrada. Para ello, puede asociar a cualquier nodo del árbol de escena (objeto de tipo **Node**) **métodos de gestión de eventos**, que reciben como parámetro un objeto de tipo **InputEvent**. Son los siguientes (listados en orden de prioridad):

- (1) **_input**
- (2) **_shortcut_input** (únicamente para eventos **InputEventKey**, **InputEventShortcut**, or **InputEventJoypadButton**).
- (3) **_unhandled_key_input** (únicamente para eventos **InputEventKey**)
- (4) **_unhandled_input**

Cuando se produce un evento se ejecuta el primer método (**_input**) para todos los nodos en un árbol de escena que lo tengan definido, hasta que alguno de ellos **lo consume**. Si ninguno lo hace, se repite el proceso para el segundo método, y así hasta el cuarto. Entre (1) y (2) un evento **puede ser consumido por un control del GUI**.

Procesamiento y consumición de eventos

Estos métodos tienen un parámetro **event** de tipo **InputEvent**. En ellos:

- Se puede comprobar de qué clase es el evento, usando **is**. Por ejemplo:

```
if event is InputEventKey:  
    # código para gestionar evento de teclado
```

- Cuando ya se sabe el tipo se puede acceder a las propiedades específicas de ese tipo. Por ejemplo:

```
if event is InputEventKey:  
    if event.pressed:  
        # código para gestionar evento de pulsación de tecla
```

- Se puede invocar el método **set_input_as_handled** de la clase **Viewport**, para declarar el evento como consumido. Por ejemplo:

```
if event is InputEventKey:  
    if event.pressed and event.keycode == KEY_A:  
        get_viewport().set_input_as_handled() ## consume pulsación de 'A'
```

Orden de ejecución de métodos de gestión de eventos

Para cada uno de los 4 métodos de gestión de eventos anteriores, Godot ejecuta el método para todos los nodos del árbol (que lo tengan definido) en un orden **primero en profundidad**, pero **inverso entre hermanos** (inverso respecto al orden de los hijos en la lista del padre).

En esta captura de un árbol de escena de Godot, cada nodo tiene un nombre que indica su orden a efectos de gestión de eventos.

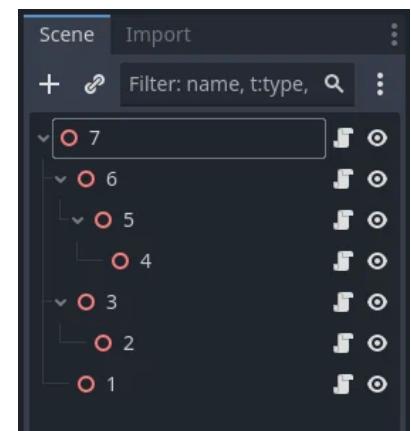


Imagen de la documentación de Godot: [Using InputEvent](#)

Ejemplo de procesamiento de eventos

Ejemplo tomado de la cámara orbital simple de las prácticas, el método **_input** responde a eventos de teclado y ratón. Para ello actualiza las variables **dxy**, **dz** y **bdrp**. En esta parte inicial se ve el código de gestión de las teclas:

```
func _input( event : InputEvent ):  
    var av : bool = true ## actualizar vista si/no  
  
    if event is InputEventKey and event.pressed: ## pulsaciones de teclas  
        match event keycode:  
            KEY_UP: dxy += Vector2( 0, -at )  
            KEY_DOWN: dxy += Vector2( 0, +at )  
            KEY_RIGHT: dxy += Vector2( -at, 0 )  
            KEY_LEFT: dxy += Vector2( at, 0 )  
            KEY_MINUS, KEY_PAGEDOWN, KEY_KP_SUBTRACT: dz *= 1.05  
            KEY_PLUS, KEY_PAGEUP, KEY_KP_ADD: dz = max( dz/1.05, 0.1 )  
            _: av = false  
  
    ## ..... continua ...
```

Ejemplo de procesamiento de eventos

En esta segunda parte vemos el código de gestión de eventos de ratón:

```
func _input( event : InputEvent ):

    ## ... código anterior ...

    elif event is InputEventMouseButton: ## botón ratón o rueda
        match event.button_index:
            MOUSE_BUTTON_RIGHT: bdrp = event.pressed ; av = false
            MOUSE_BUTTON_WHEEL_DOWN: dz *= 1.05
            MOUSE_BUTTON_WHEEL_UP: dz = max( dz/1.05, 0.1 )
            _: av = false

    elif event is InputEventMouseMotion and bdrp: ## movim. ratón
        dxy += ar * Vector2( -event.relative.x, event.relative.y )

    else: av = false # no actualizar la vista

    if av : ## actualizar la vista ....
```

Sesión 9: Interacción

Created 2025-12-01

Page 29 / 87.

2. Eventos en Godot.
2.2. Gestión de eventos..

Ejemplo de mapa de entrada en el editor

En este ejemplo se asocian pulsaciones de teclas y ejes del joypad a acciones de movimiento del personaje controlado por el jugador:

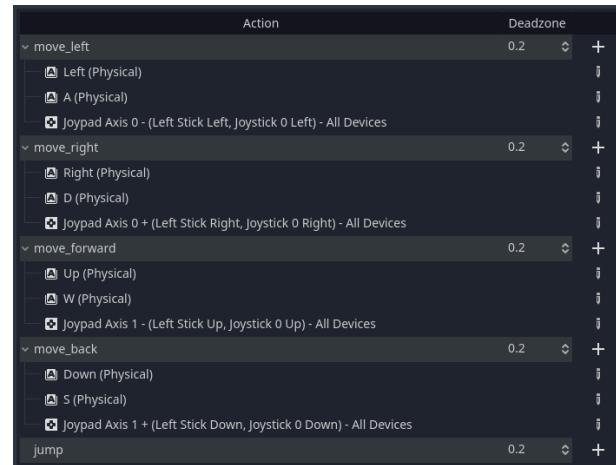


Imagen de la documentación de Godot: [First 3D game: player input](#)

El mapa de entrada. Eventos y acciones.

Durante la ejecución de una aplicación Godot, los eventos de entrada pueden ser mapeados a **acciones** definidas en el **mapa de entrada** (*Input Map*).

- El mapa de entrada es un objeto *singleton* (solo hay una instancia global), de la clase **InputMap**.
- Contiene una lista de acciones, cada una de ellas es una instancia de la clase **InputEventAction** que se identifica con una cadena única llamada **nombre de la acción** (propiedad **action** de la clase, de tipo **StringName**).
- Se puede configurar en el editor, o bien desde código, con los métodos **add_action** y **action_add_event** de la clase **InputMap**.
- Cada acción del mapa de entrada tiene asociados una o varias instancias de la clase **InputEvent** (o de cualquier subclase). Son los eventos que **disparan** esa acción.

La **ventaja** del mapa de entrada es que **permite separar el tipo de evento** que las dispara del **código que las gestiona**.

Sesión 9: Interacción

Created 2025-12-01

Page 30 / 87.

2. Eventos en Godot.
2.2. Gestión de eventos..

Acciones predefinidas en el mapa de entrada

Godot incorpora varias acciones predefinidas (*built-in actions*) en el mapa de entrada (se pueden visualizar con el editor si se activa la opción correspondiente):

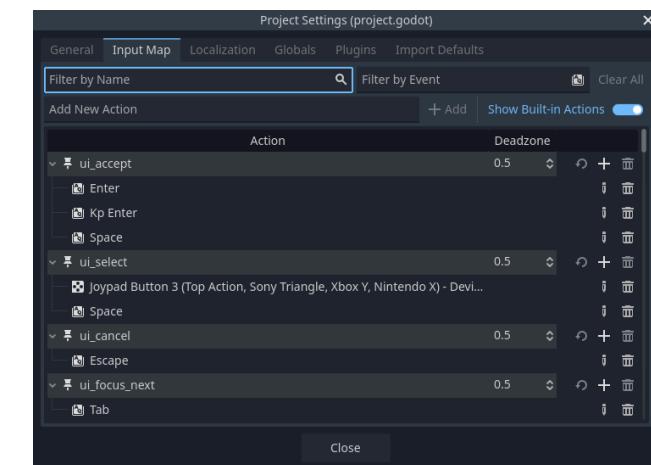


Imagen de la documentación de Godot: [First 3D game: player input](#)

Definición de acciones usando código

Las acciones se pueden definir y asociar a eventos usando código GDScript, por ejemplo, este código (asociado a un nodo cualquiera) crea una acción que se disparará cuando se pulse o se levante la tecla A o la tecla flecha arriba:

```
## crear instancia de evento: pulsar o levantar tecla 'A'  
var evento_tecla_A := InputEventKey.new()  
evento_tecla_A.keycode = KEY_A  
  
## crear instancia de evento: pulsar o levantar tecla 'flecha arriba'  
var evento_tecla_arriba := InputEventKey.new()  
evento_tecla_arriba.keycode = KEY_UP  
  
## añadir la acción al mapa de entrada  
InputMap.add_action( "accion_creada_script" )  
  
## asociar los eventos a la acción  
InputMap.action_add_event( "accion_creada_script", evento_tecla_A )  
InputMap.action_add_event( "accion_creada_script", evento_tecla_arriba )
```

Sesión 9: Interacción

Created 2025-12-01

Page 33 / 87.

2. Eventos en Godot.
2.2. Gestión de eventos..

Muestreo de dispositivos. La clase `Input`

En ocasiones puede ser útil usar **muestreo de estado (polling)** de los dispositivos de entrada en lugar de las funciones de gestión de eventos.

- Para ello Godot tiene la **clase singleton `Input`** con métodos para consultar el estado de teclas, botones de ratón, acciones, ejes o botones de *joypads*, giroscopios, acelerómetros, magnetómetros, etc...

Podemos consultar el **estado de acciones** con estos métodos de `Input`:

- `is_action_pressed`**: es `true` si la acción **está pulsada en el momento de la llamada**.
- `is_action_just_pressed, is_action_just_released`**: son `true` si la acción se ha pulsado o levantado **desde el último frame anterior a la llamada**.
- `is_action_just_pressed_by_event, is_action_just_released_by_event`**: es `true` si la acción se ha pulsado o levantado **desde el último frame anterior a un evento**.

Respondiendo a acciones en métodos de eventos

Una vez definidas las acciones en el mapa de entrada, se puede escribir código para responder a eventos asociados con una acción, usando el método `is_action` de la clase `InputEvent`.

Por ejemplo, este código (asociado a un nodo cualquiera) responde a la acción creada en el ejemplo anterior (`accion_creada_script`) y a otra acción llamada `accion_tecla_Q`. En ambos casos el evento es consumido:

```
func _input( event : InputEvent ):  
  
    if event.is_action( "accion_tecla_Q" ) :  
        print("Nodo raíz: 'accion_tecla_Q' disparada")  
        get_viewport().set_input_as_handled()  
  
    elif event.is_action( "accion_creada_script" ) :  
        print("Nodo raíz: 'accion_creada_script' disparada")  
        get_viewport().set_input_as_handled()
```

Sesión 9: Interacción

Created 2025-12-01

Page 34 / 87.

Sesión 9: Interacción

Created 2025-12-01

Page 34 / 87.

Ejemplo de muestro de una acción asociada a un tecla

En este ejemplo se usa el método `_input` y además muestro en cada frame (en el método `_process`). Se usa la acción llamada `accion_tecla_Q`, asociada a eventos de pulsar y levantar la tecla Q.

```
func _input( event : InputEvent ) : ## se ejecuta en cada evento  
    if event.is_action("accion_tecla_Q"):  
        print("_input: 'accion_tecla_Q' - pressed es ", event.pressed)  
  
    var contador : int = 0 ## contador de frames.  
  
    func _process( delta : float ) : ## se ejecuta en cada frame  
        contador = contador+1  
        if Input.is_action_just_pressed("accion_tecla_Q"):  
            print("_process ",contador,": 'accion_tecla_Q' pulsada ahora")  
        if Input.is_action_just_released("accion_tecla_Q"):  
            print("_process ",contador,": 'accion_tecla_Q' levantada ahora")  
        if Input.is_action_pressed("accion_tecla_Q"):  
            print("_process ",contador,": 'accion_tecla_Q' está pulsada")
```

Sesión 9: Interacción

Created 2025-12-01

Page 35 / 87.

Sesión 9: Interacción

Created 2025-12-01

Page 36 / 87.

Ráfaga de mensajes asociados a una pulsación en el ejemplo

Con el código anterior, si el usuario pulsa y levanta la tecla Q, se produce una ráfaga de mensajes como los que se muestran a continuación:

```
_input: 'accion_tecla_Q' - pressed es true  
_process 161: 'accion_tecla_Q' pulsada ahora  
_process 161: 'accion_tecla_Q' está pulsada  
_process 162: 'accion_tecla_Q' está pulsada  
_process 163: 'accion_tecla_Q' está pulsada  
_process 164: 'accion_tecla_Q' está pulsada  
_input: 'accion_tecla_Q' - pressed es false  
_process 165: 'accion_tecla_Q' levantada ahora
```

El número de frames en los que se repite el mensaje “*accion_tecla Q*” *está pulsada* depende de la duración de la pulsación de la tecla por parte del usuario (esto puede usarse para comportamientos que puedan depender de dicho tiempo).

Problema

Problema 9.1:

En una aplicación Godot cualquiera, añade código al nodo raíz de forma que cada vez que se pulse y luego se levante una tecla (por ejemplo la tecla P), se imprima en pantalla un mensaje con el tiempo total en segundos que dicha tecla ha estado pulsada, en los casos en los que ha permanecido pulsada al menos el tiempo de un frame.

Sección 3.

Interfaz de usuario y señales en Godot.

1. Interfaz de usuario.
2. Clases de controles.
3. Señales

Subsección 3.1.

Interfaz de usuario.

Interfaz de usuario (IU)

En general, el término **Interfaz Gráfica de Usuario** (o **GUI**, de **Graphical User Interface**, o simplemente **UI**) se refiere a elementos visuales que aparecen en la pantalla durante la ejecución de una aplicación y que

- permiten presentar información al usuario en forma de texto, valores numéricos, colores, iconos, tablas de datos, infografías, etc...
- permiten al usuario introducir información como textos, valores lógicos, cantidades, colores, selecciones de una opción de entre una lista, etc..

En particular, en el contexto de las **aplicaciones gráficas interactivas 2D o 3D**, el término **interfaz de usuario (user interface)** se suele referir a los elementos visuales distintos de las proyecciones 2D o 3D de los objetos de la escena, elementos que sirven únicamente para presentar o pedir información al usuario, pero no son representaciones visibles del modelo 3D o 2D que la aplicación gestiona.

Ejemplo de GUI creado con Godot

En esta captura vemos los elementos del GUI rodeando la vista 3D central.

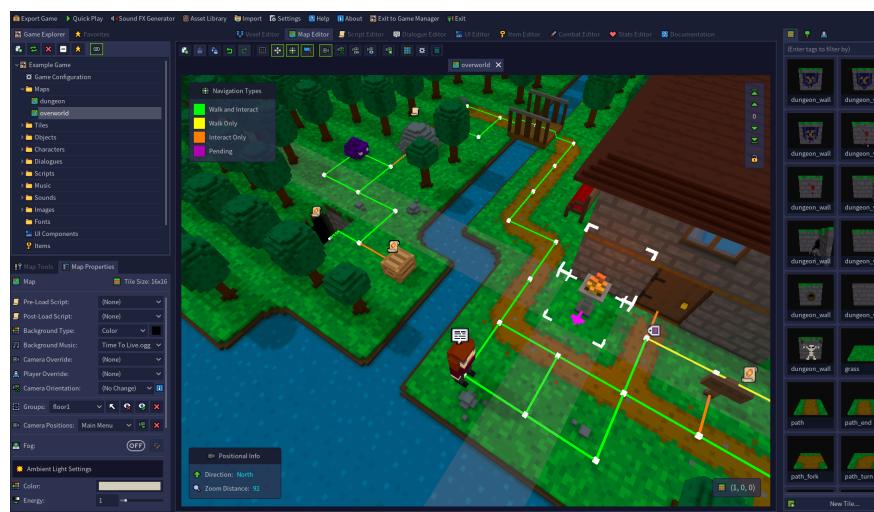
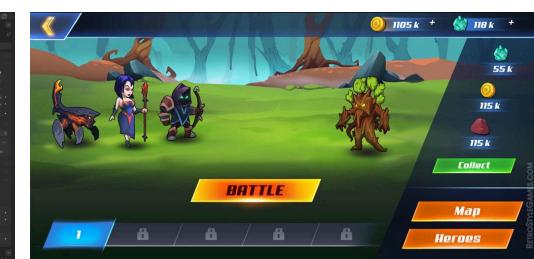
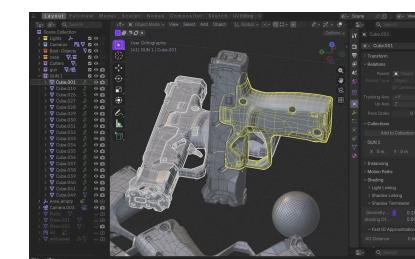


Imagen del editor de mundos virtuales **RPG in a Box** zeromatrix.itch.io/rpginabox

Ejemplos de interfaces de usuario

En estos ejemplos de aplicaciones gráficas interactivas, el **UI** está formado por todos los elementos visuales en pantalla (usualmente en 2D) que se superponen (o rodean) a la imagen 2D o 3D del modelo de escena y objetos.

Estos elementos permiten presentar información de estado al usuario o pedirle que introduzca datos o selecciones.



Imágenes de: *Blender extensions. SHEK: [Theme Plasticity](#)* (izquierda)
Padel Konstantinov [What is UI in Games ?](#) (derecha)

El interfaz de usuario en Godot

Godot permite incorporar elementos de interfaz de usuario:

- Se usa la clase base **Control**, derivada de **CanvasItem**, a su vez derivada de **Node**.
- Cada instancia de una clase derivada de **Control** es un elemento visual 2D, que ocupará un **área rectangular** en un **viewport** y cuyo **estilo** es configurable.
- Los objetos **Control** capturan y responden a **eventos de entrada**.
- Algunos controles actúan como **contenedores** de otros, lo cual permite organizarlos jerárquicamente.
- El árbol de escena de la aplicación puede incorporar nodos de clases derivadas de **Control**, donde la relación padre-hijo se interpreta en términos de **contenedor-contenido**, es decir, **el rectángulo del hijo está incluido dentro del del padre**.
- Al igual que el resto de nodos, los nodos de tipo **Control** pueden crearse y configurarse **mediante scripts y/o con el editor**

Propiedades de la clase **Control**

La forma y posición del rectángulo ocupado por un control en el *viewport* viene determinada por las siguientes propiedades de la clase **Control**:

- **position** (tipo **Vector2**): posición de la esquina superior izquierda del rectángulo, relativo a la posición del control padre (o del *viewport* si no tiene).
- **size** (tipo **Vector2**): ancho y alto del rectángulo, en unidades de pixels.
- **rotation** (tipo **float**): ángulo de rotación (en radianes) alrededor del pivote
- **scale** (tipo **Vector2**): factor de escala en los ejes X e Y, alrededor del pivote.
- **pivot_offset** (tipo **Vector2**): posición relativa del pivote en pixels.

El estilo se puede configurar usando estas propiedades:

- **theme** (tipo **Theme**): permite definir y aplicar estilos coherentes a múltiples controles.
- **theme_type_variation** (tipo **StringName**): permite aplicar variaciones de estilo definidas en el tema.

Subsección 3.2.

Clases de controles.

Galería de controles de Godot

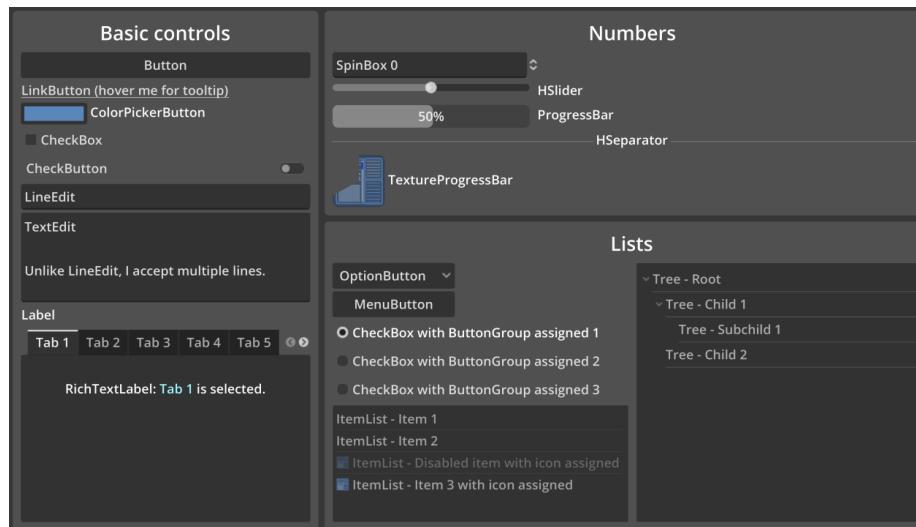


Imagen de Godot Asset Library: Control Galery Demo

godotengine.org/asset-library/asset/2766

Algunas clases derivadas de **Control**

- **BaseButton**: clase base para botones.
- **Container**: clase base para contenedores de otros controles.
- **Range**: clase base para valores numéricos en un rango.
- **ItemList**: lista vertical de items seleccionables.
- **Label**: caja con texto plano, puede ser multilínea.
- **LineEdit**: entrada de texto de una línea.
- **MenuBar**: barra de menús.
- **RichTextLabel**: caja con texto enriquecido (con íconos o imágenes).
- **Separator**: línea horizontal separadora.
- **TabBar**: barra de pestañas en horizontal.
- **TextEdit**: editor de textos de múltiples líneas.
- **Tree**: visor de estructuras jerárquicas de items.
- **VideoStreamPlayer**: reproductor de videos.

Otras clases derivadas de **Control**

Otras clases derivadas directamente de **Control** son:

- **GraphEdit**: editor gráfico interactivo de grafos de nodos, cada nodo es un objeto **GraphNode**, se usa típicamente para editar interactivamente estructuras de datos que representan diagramas de flujos de datos, o cualquier otra estructura de tipo grafo.
- **ColorRect**: rectángulo de color sólido.
- **NinePatchRect**: rectángulo con una imagen dividida en 9 rectángulos de igual tamaño, de forma que al escalar el control, cada rectángulo se escala en X y/o Y dependiendo de su posición.
- **ReferenceRect**: rectángulo con un borde resaltado.
- **TextureRect**: rectángulo que muestra una imagen (textura).
- **Panel**: rectángulo con estilo configurable (una instancia de **StyleBox**).

Clases para valores numéricos (derivadas de **Range**)

Las clases que permiten entrada de valores numéricos son:

- **SpinBox**: entrada de valores numéricos, mediante el teclado, y adicionalmente con dos botones para incrementar o decrementar el valor.
- **Slider**: entrada de valores numéricos arrastrando con el ratón un elemento en un segmento o *carril*, hay dos sub-clases **HSlider** (horizontal) y **VSlider** (vertical)
- **ScrollBar**: barra de scroll. También hay dos sub-clases **HScrollBar** (horizontal) y **VScrollBar** (vertical).

Otras clases también derivadas de **Range** permiten mostrar valores numéricos en forma gráfica, son:

- **ProgressBar** barra de progreso, muestra como cambia en el tiempo el porcentaje completado de un proceso
- **TextureProgressBar** igual, pero usando una imagen.

Clases para botones (derivadas de **BaseButton**)

Las clases derivadas de **BaseButton** permiten crear botones con diferentes apariencias y comportamientos:

- **Button**: botón estándar con texto o ícono. Se puede usar directamente, o bien via algunas de las clases derivadas, a saber:
 - ▶ **CheckBox**: entrada de valores lógicos (true/false), o en general selección de una entre dos opciones excluyentes.
 - ▶ **CheckButton**: similar a la anterior, pero con apariencia de botón.
 - ▶ **ColorPickerButton**: selector de colores, permite seleccionar un color usando uno de entre varios modelos de color (RGB, HSV, etc...)
 - ▶ **MenuButton**: despliega un menú flotante (instancia de **PopupMenu**)
 - ▶ **OptionButton**: despliega un **PopupMenu** para seleccionar una opción, cuyo texto se muestra en el botón.
- **LinkButton**: botón con una *URI* que se abre con el navegador.
- **TextureButton**: botón con una imagen en lugar de un texto o un tema.

Clases contenedoras (derivadas de **Container**) (2/2)

- **AspectRatioContainer**: contenedor que mantiene una relación de aspecto fija (ancho/alto) para el área ocupada por sus hijos.
- **CenterContainer**: contenedor que centra su(s) hijo(s) en el área disponible.
- **FlowContainer**: contenedor que dispone sus hijos en filas (**HFlowContainer**), pasando a la siguiente fila cuando no hay espacio disponible (como se disponen palabras en un párrafo de texto). También pueden configurarse por columnas (**VFlowContainer**).
- **FoldableContainer**: un contenedor que puede plegarse a una sola línea con texto o expandirse y mostrar todos sus nodos hijos.
- **GraphElement** contenedor para elementos de un control **GraphEdit**. Tiene dos subclases: **GraphNode** y **GraphFrame**.
- **ScrollContainer** contenedor que añade barras de scroll automáticamente a sus nodos hijos, si el área disponible es menor que la requerida.

Sesión 9: Interacción

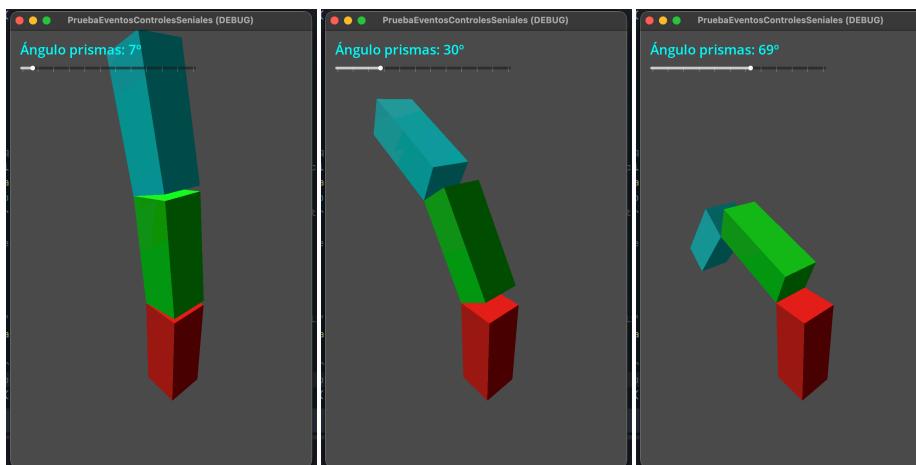
Created 2025-12-01

Page 53 / 87.

3. Interfaz de usuario y señales en Godot..
3.2. Clases de controles..

Ejemplo: deslizador (slider) y etiqueta (label)

Vemos un ejemplo de creación con GDScript de un control deslizador horizontal **HSlider** y una etiqueta **Label** que muestra el valor actual del deslizador. Controla un grado de libertad de un modelo jerárquico (ángulo entre prismas).



Sesión 9: Interacción

Created 2025-12-01

Page 55 / 87.

Creación del GUI

Los controles pueden crearse tanto en el editor como mediante scripts.

- **En el editor**, se añade un árbol de nodos, todos ellos derivados de **Control**, donde los nodos contendores son nodos no terminales que incluyen a sus nodos hijos.
- **Mediante código GDScript**, creando y configurando cada nodo, y añadiendo los nodos hijos a sus nodos. Se puede hacer, por ejemplo, en el método **_ready()** del nodo raíz de una escena (o en cualquier otro).

En cualquier caso, los controles aparecerán en el viewport asociado al nodo contenedor raíz del GUI.

Sesión 9: Interacción

Created 2025-12-01

Page 54 / 87.

3. Interfaz de usuario y señales en Godot..
3.2. Clases de controles..

Ejemplo: código GDScript del deslizador y la etiqueta (1/2)

Para crear una caja con el texto y el deslizador podemos añadir código al método **_ready** de cualquier nodo asociado al viewport donde queremos ver el GUI:

```
var etiqueta : Label = null
var deslizador : HSlider = null

func _ready(): ## es el método '_ready' del nodo raíz (u otro nodo)

    ## crear y configurar el deslizador
    deslizador = HSlider.new()
    deslizador.size_flags_horizontal = Control.SIZE_EXPAND_FILL
    deslizador.ticks_on_borders = true # mostrar barras verticales
    deslizador.tick_count = 12        # número de barras verticales
    deslizador.min_value = 0.0        # valor a la izquierda (mínimo)
    deslizador.max_value = 120.0      # valor a la derecha (máximo)
    deslizador.value     = 30.0       # valor inicial
    deslizador.step      = 1          # incremento entre valores posibles

    ## .... continúa
```

Sesión 9: Interacción

Created 2025-12-01

Page 56 / 87.

Sesión 9: Interacción

Created 2025-12-01

Page 56 / 87.

Ejemplo: código GDScript del deslizador y la etiqueta (2/2)

```
func _ready(): ## es el método '_ready' del nodo raíz (u otro nodo)
## .... código anterior

## crear y configurar la etiqueta de texto
etiqueta = Label.new() ## etiqueta
etiqueta.size_flags_horizontal = Control.SIZE_EXPAND_FILL
etiqueta.text = "Ángulo prismas: %.0f°" % deslizador.value
etiqueta.add_theme_font_size_override( "font_size", 40 )
etiqueta.add_theme_color_override( "font_color", Color( 0, 1, 1 ) )

## crear caja contenedora vertical
var vbox := VBoxContainer.new() # crea el objeto con caja vertical
vbox.position = Vector2( 30, 30 ) # relativa al viewport de este nodo
vbox.size.x = 520 # ancho de la caja en pixels
vbox.add_theme_constant_override( "separation", 20 ) # en pixels

## crear sub-árbol de controles
vbox.add_child( etiqueta )
vbox.add_child( deslizador )
add_child( vbox )
```

Señales. Señales en Godot.

En general, en el contexto de la programación, el término **señal** se refiere a un mecanismo de comunicación entre diferentes componentes de un sistema, donde un componente (el emisor) puede emitir una señal para notificar a otros componentes (los receptores) que ha ocurrido un evento o que se ha producido un cambio de estado.

En particular, en Godot, una señal es un objeto, instancia de la clase **Signal**, creado por un **nodo emisor** para notificar la ocurrencia de un cambio de estado, de forma que se ejecuten automáticamente (sin llamada explícita) uno o varios métodos en otros nodos (receptores).

La gran ventaja de usar señales es la **separación entre emisores y receptores**:

- Un nodo emisor detecta el cambio de estado relevante y emite la señal, sin necesidad de conocer qué nodos receptores la van a recibir.
- Uno o varios nodos receptores pueden ejecutar código cuando ocurre el cambio de estado, sin necesidad de conocer qué nodo emisor ha emitido la señal.

Subsección 3.3.

Señales

La clase **Signal**

Un objeto de la clase **Signal** se construye especificando:

- Un objeto (de cualquier clase derivada de **Object**) que podrá actuar como **emisor** de la señal.
- El nombre único de la señal (una cadena de texto, de tipo **StringName**)

Se pueden usar los siguientes métodos de la clase **Signal**:

- **connect** conecta la señal a un objeto **invocable** (**Callable**) receptor.
- **disconnect** desconecta la señal de un objeto receptor.
- **emit** emite la señal, ejecutando automáticamente los métodos de los objetos receptores conectados (o las funciones conectadas).

Un **objeto invocable** es una instancia de la clase **Callable** y puede ser:

- Una función global (no método de ningún objeto)
- Un método de un objeto concreto.

Señales emitidas por un nodo

Cualquier nodo (objeto de una clase derivada de **Node**) puede emitir un conjunto de señales, que dependen del tipo del nodo. De especial interés son las señales emitidas por los nodos del interfaz (de clases derivadas de **Control**)

En el editor podemos ver la señales emitidas por un nodo seleccionado (en el panel, pestaña **Nodo**, ícono **Señales**). A modo de ejemplo, algunas emitidas por un control **Button**:

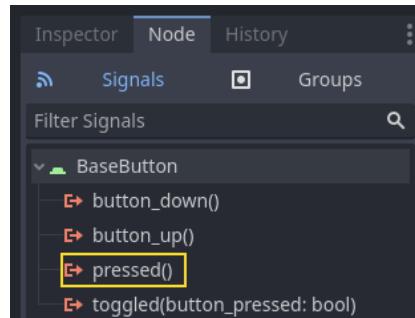


Imagen de la documentación oficial de Godot: [using signals](#)

Conexión de señales predefinidas mediante GDScript

La conexión de señales también puede hacerse mediante código GDScript, usando el método **connect** de la clase **Signal**. En el ejemplo anterior, después de crear el deslizador, conectamos su señal **value_changed** al método **_valor_deslizador_actualizado** del nodo actual:

```
var deslizador : HSlider = null
var etiqueta : Label = null

func _ready():
    ## ... código anterior de creación de los controles
    ## conectar señal y actualizar los prismas la 1a vez.
    deslizador.connect( "value_changed", _valor_deslizador_actualizado )
    $Prismas.fijar_rotacion_nodos( deslizador.value )

func _valor_deslizador_actualizado( valor : float ):
    etiqueta.text = "Ángulo prismas: %.0f°" % valor # actualizar texto
    $Prismas.fijar_rotacion_nodos( valor ) # actualizar los prismas
```

La señal **value_changed** lleva un parámetro con el valor actual.

Conexión de señales predefinidas en el editor

Usando el editor se puede conectar una señal emitida por un nodo (identificada por su nombre) a un método de otro nodo receptor. En este caso se conecta la señal de nombre **pressed** al método **on_button_pressed** del nodo del árbol con nombre **Sprite2D**

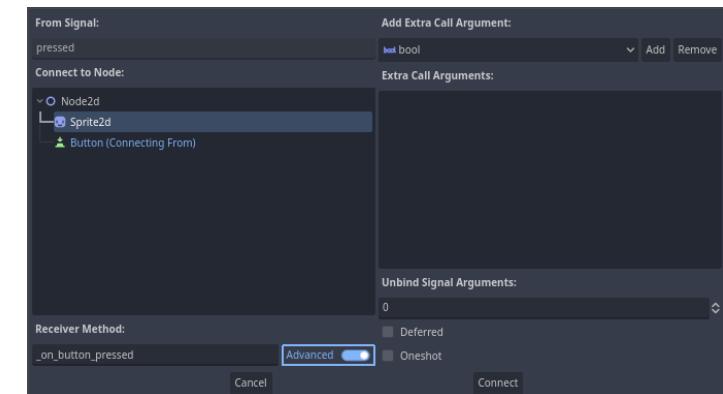


Imagen de la documentación oficial de Godot: [Using signals](#)

Acoplamiento entre emisores y receptores de señales

En el script del ejemplo anterior:

- El nodo con el script es el nodo receptor de la señal, y **conoce** al nodo emisor (el nodo **deslizador**).
- Asimismo, el objetivo de la señal es ejecutar un método en el nodo de nombre «**Prismas**», así que en el script se **conoce** ese nodo de los prismas.

Por tanto:

- Añadir nuevos emisores **requiere extender el script**: será necesario conocer los otros posibles emisores para conectar la señal.
- Añadir nuevos nodos objetivo **requiere extender el script**: será necesario conocer los nuevos nodos objetivo para invocar sus correspondientes métodos.

En definitiva **hay excesivo acoplamiento entre emisores, receptores y nodos objetivo**, lo cual dificulta extender el código. Esto se puede solucionar usando señales definidas por el programador.

Señales definidas por el programador: declaración y emisión

Además de las señales predefinidas en los nodo de tipo **Control**, el programador puede definir sus propias señales (**custom signals**), usando la palabra clave **signal**. Todo esto es **independiente de qué nodos reciban la señal**, para ello usamos un módulo **gdscript** global, por ejemplo en el módulo **Utilidades**:

```
extends Node
## .... otras declaraciones globales en 'Utilidades.gd' ....

## Declaraciones de señales accesibles en todos los scripts:
signal señal1          ## sin parámetros
signal señal2( p : float ) ## un parámetro float
```

Desde cualquier script en un nodo emisor podemos emitir las señales, usamos el método **emit** de **Signal**. Por ejemplo, para emitir las señales anteriores:

```
## emitir las señales (independiente de los posibles receptores):
Utilidades.señal1.emit()      ## sin parámetros
Utilidades.señal2.emit( 34.56 ) ## debemos dar una expresión float
```

Señales definidas por el programador: conexión y recepción

Cualquier nodo interesado en recibir estas señales puede convertirse en receptor de las mismas, conectándose a ellas con el método **connect** de **Signal**. Esto es **independiente de qué nodos emitan la señal**. Por ejemplo, un nodo cualquiera puede suscribirse a las señales anteriores, si se le asigna un script con este código:

```
extends Node

func _ready() -> void:
    ## ... otro código de inicialización ...
    ## conectar las señales con los métodos receptores de este nodo
    Utilidades.señal1.connect( _metodo_señal1 )
    Utilidades.señal2.connect( _metodo_señal2 )

func _metodo_señal1(): # se ejecuta al recibir 'señal1'
    print("Receptor: 'señal1' recibida.")

func _metodo_señal2( v : float ): # se ejecuta al recibir 'señal2'
    print("Receptor: 'señal2' recibida, v == ",v)
```

Sección 4. Selección

1. Selección en aplicaciones gráficas interactivas
2. Selección en Godot.
3. Problemas: selección por intersecciones

Subsección 4.1. Selección en aplicaciones gráficas interactivas

El proceso de selección

En una aplicación gráfica interactiva (2D o 3D), la **selección** es el proceso mediante el cual el usuario usa la aplicación para designar uno o varios objetos o componente de la escena que se está visualizando en pantalla.

- Es una operación fundamental en la interacción usuario-aplicación.
- Permite al usuario indicar uno o varios objetos para: ver o editar sus propiedades, borrarlos, duplicarlos, agruparlos, desagruparlos, moverlos, transformarlos, etc ...
- La selección puede hacerse de diversas formas o con diversos tipos de dispositivos.
- Lo más común es hacer un *click* con el ratón sobre la proyección de un objeto en pantalla (*click* en un pixel donde se proyecta el objeto).
- De esta forma se pueden seleccionar objetos individuales. Para seleccionar varios objetos, se puede hacer uno a uno, añadiendo nuevos objetos a la selección actual en cada paso.

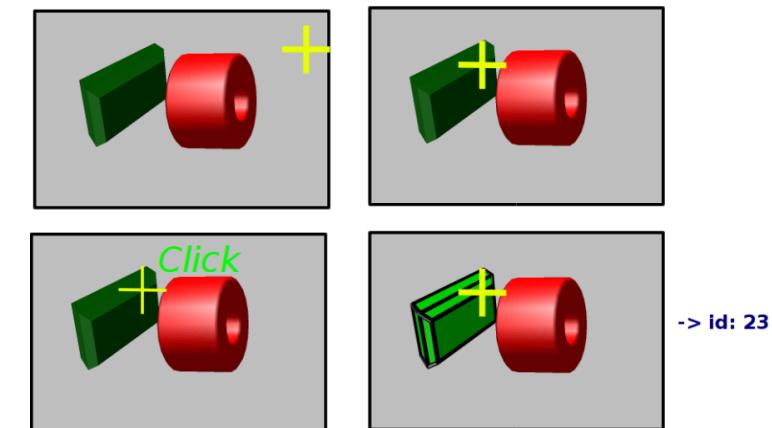
Implementación de la selección

Existen básicamente dos forma distintas de implementar la selección:

- Usando **rasterización** sobre un objeto *framebuffer* oculto:
 - ▶ Este *framebuffer* es una zona de memoria en la GPU en la cual se renderiza (por rasterización) la escena. En lugar de usar los colores RGB resultado de texturas e iluminación, a todos los pixels donde se proyecta un objeto se les **asigna un valor entero que identifica al objeto**.
 - ▶ Cuando el usuario hace *click* en un pixel, se crea el *framebuffer* y se **lee el identificador de dicho pixel**.
- Usando ***ray casting***, es decir, calculando intersecciones de un rayo:
 - ▶ En este caso, se calcula una semirecta (**rayo**) que pasa por todos los puntos de la recta que se proyecta en el punto central a un pixel.
 - ▶ Cuando el usuario hace *click* en un pixel, se calcula el correspondiente rayo y se interseca con todos los polígonos o triángulos de la escena. **Se selecciona el más cercano**.

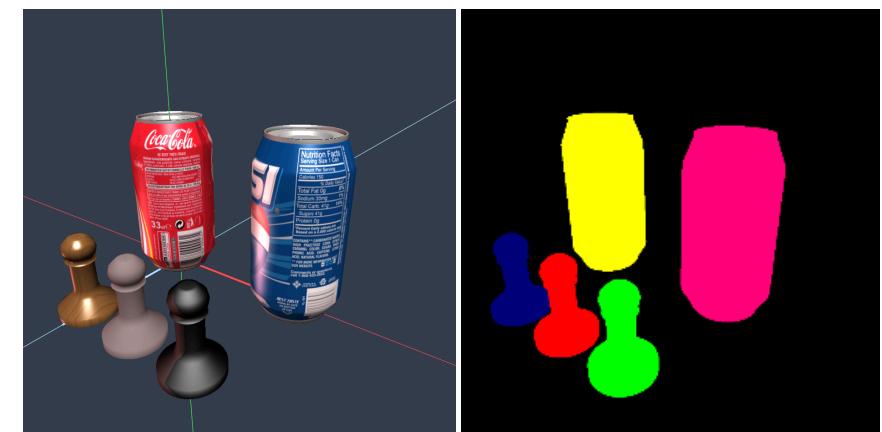
Ejemplo de selección con un *click*

Vemos un ejemplo de selección de un objeto entre varios. Lo usual es que el objeto seleccionado cambie de aspecto de alguna forma (un color resaltado, aristas de contorno resaltadas, etc....).



Ejemplo de *framebuffer* de selección

Vemos un ejemplo de selección usando un *framebuffer* oculto. En la imagen de la izquierda se ve la escena renderizada normalmente, y en la imagen de la derecha se ve el *framebuffer* oculto, donde cada pixel tiene un asociado un entero (que aparece como un color RGB en la imagen).



Selección en Godot

En Godot, la selección se puede hacer usando ray-tracing. Se puede:

- Calcular el rayo que pasa por el centro de un pixel (usando la cámara).
- Calcular intersecciones entre un rayo y los llamados **nodos colisionadores**. Son objetos con mallas **no visibles** pero **detectables** en el cálculo de intersecciones.
- Este tipo de nodos son de alguna clase derivada de **CollisionObject3D** (clases **StaticBody3D**, **RigidBody3D** entre otras), en los ejemplos usaremos exclusivamente **StaticBody3D**.
- La **forma (geometría) del colisionador** la debe definir otro nodo de clase **CollisionShape3D**, que debe ser **hijo del nodo colisionador**, y que llamamos **nodo de forma**.
- Todo **CollisionShape3D** tiene la propiedad **shape** de clase **Shape3D**, es el objeto que define la forma, puede ser de varias subclases de **Shape3D**:
 - ▶ Objetos simples (**BoxShape3D**, **SphereShape3D**, etc...)
 - ▶ Mallas arbitrarias (**ConcavePolygonShape3D** o **ConvexPolygonShape3D**).

Subsección 4.2.

Selección en Godot.

4. Selección.
4.2. Selección en Godot..

Creación de mallas colisionadoras

Los nodos colisionadores se pueden crear a partir de un nodo de la clase **MeshInstance3D**, que contiene la **malla visible** (la que se visualiza). Es necesario:

- Crear un nodo **StaticBody3D**, puede ser hijo del **MeshInstance3D**, para que tenga la misma transformación.
- Añadir un nodo **CollisionShape3D** como hijo del nodo **StaticBody3D**.
- Asignar un objeto de alguna clase derivada de **Shape3D** en la propiedad **shape** del **CollisionShape3D**.

Esto se puede hacer de dos formas, a partir de un nodo **MeshInstance**

- **Con el editor:** creando el nodo colisionador y el nodo de forma en el editor, o bien usando el menú **Malla/Mesh** y la opción **Crear forma de colisión** (ver siguiente transparencia).
- **En un script:** creando el nodo colisionador y el nodo de forma con código, o bien simplemente **invocando el método `create_trimesh_collision`** del objeto **MeshInstance3D** (ver transparencia).

Sesión 9: Interacción

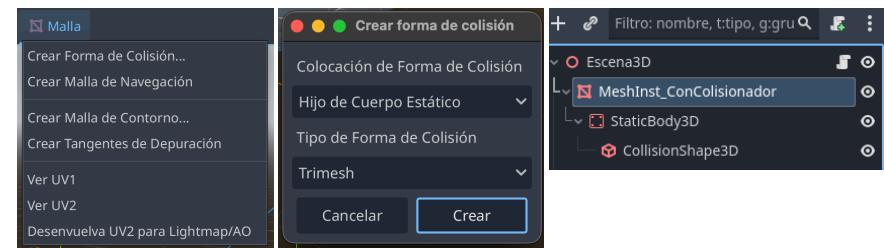
Created 2025-12-01

Page 74 / 87.

4. Selección.
4.2. Selección en Godot..

Creación de mallas colisionadoras en el editor

En el editor, podemos crear un colisionador para un **MeshInstance3D** cualquiera, para ello se selecciona el nodo **MeshInstance3D** y en la parte superior se usa la opción **Crear forma de colisión** en el menú **Malla** (o **Mesh**), creará un hijo **StaticBody3D** y un nieto **CollisionShape3D** con una forma **ConcavePolygonShape3D**.



Vemos: el menú **Malla/Mesh** (izquierda), las opciones para crear el colisionador (centro), y los nodos del árbol con el objeto visible (**MeshInstance3D**), el colisionador (**StaticBody3D**), y su forma (**CollisionShape3D**).

Creación de mallas colisionadoras con un script

En un script, se puede conseguir el efecto equivalente:

- Se parte de un nodo `MeshInstance3D` (con la malla visible), al cual ya se le debe de haber asignado alguna malla en su campo `mesh` (se puede hacer, por ejemplo, en el método `_ready()`, después de la inicialización de `mesh`).
- Se invoca el método `create_trimesh_collision()` del `MeshInstance3D`, que crea automáticamente un hijo `StaticBody3D` y un nieto `CollisionShape3D` con una forma `ConcavePolygonShape3D`.

```
func _ready():
    var tablas : Array = []
    # ..... aquí va el código de inicialización de las tablas.
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

    # crear colisionador automáticamente
    create_trimesh_collision()
```

Ejemplo de cálculo de selección con un click (1/2)

En el siguiente ejemplo, se asocia a un nodo un script que responde a eventos de `click` calculando el objeto que se proyecta en el pixel donde se ha hecho `click`.

```
func _unhandled_input( event : InputEvent ):
    if event is InputEventMouseButton :
        if event.button_index == MOUSE_BUTTON_LEFT and event.pressed :

            ## 1. Buscar nodo de la cámara
            ## .....

            ## 2. Construir objeto con parámetros de intersección
            ## .....

            ## 3. Calcular la intersección
            ## .....

            ## 4. Procesar el resultado
            ## .....
```

Intersecciones rayo-malla colisionadora

En Godot, el cálculo de intersecciones rayo-malla se puede hacer usando el método `intersect_ray()` de la clase singleton `PhysicsDirectSpaceState`, que calcula la intersección entre un rayo y los colisionadores del árbol de escena.

- El rayo se define por dos puntos: el origen y un punto final (no es una semirecta, sino un segmento).
- El rayo se puede calcular a partir de las coordenadas del pixel donde se ha hecho `click`, usando los métodos: `project_ray_origin()` (para el origen) y `project_ray_normal()` (para el vector director, desde el origen hasta el final) ambos de la clase `Camera3D`
- El método `intersect_ray` devuelve información sobre el objeto colisionador (`StaticBody3D`) intersecado más cercano al origen del rayo (si hay intersección), incluyendo una referencia a dicho objeto.
- Para usar este método, es necesario obtener primero el objeto `PhysicsDirectSpaceState` de la escena. Esto se hace con la propiedad `direct_space_state` de la clase singleton `World3D`.

Buscar cámara y construir objeto de parámetros

Para buscar el nodo de la cámara, suponemos (en este ejemplo) que dicho nodo es hermano del nodo que tiene el script (está en el mismo nivel del árbol de escena). Se puede hacer de muchas otras formas.

```
## 1. Buscar nodo de la cámara
var cam : Node = get_node_or_null("../CamaraOrbitalSimple")
assert( cam != null , "No encuentro 'CamaraOrbitalSimple'" )
```

A continuación se usa la posición del `click` (está en `event.position`) para construir un objeto `query` de la clase `PhysicsRayQueryParameters3D`, que contiene el rayo y los parámetros para el cálculo de la intersección:

```
## 2. Construir objeto con parámetros de intersección
var query := PhysicsRayQueryParameters3D.new()
query.collision_mask = 0xFFFFFFFF # todas las capas
query.from = cam.project_ray_origin( event.position )
query.to = query.from+100*cam.project_ray_normal( event.position )
```

Calcular las intersecciones y procesar resultado

Para calcular las intersecciones se invoca el método `intersect_ray()` del `space_state`:

```
## 3. Calcular las intersecciones
var space_state = get_world_3d().direct_space_state
var result = space_state.intersect_ray(query)
```

Finalmente, si hay alguna intersección, comprobamos el objeto padre del colisionador, si existe y tiene el método `cuando_click`, lo invitamos:

```
## 4. Procesar el resultado
if result:
    print("SÍ hay objeto en ese punto.")
    var padre : Node = result.collider.get_parent()
    if padre != null :
        print("Objeto padre intersecado: ", padre.name )
        if padre.has_method("cuando_click"): padre.cuando_click()
        else: print("Objeto padre no tiene 'cuando_click'")
    else: print("No hay objeto padre.")
else: print("NO hay objeto en ese punto.")
```

Comportamiento de objetos clickados

En los nodos con hijos con colisionadores se puede definir un método `cuando_click()` para definir el comportamiento al ser seleccionados:

```
var color1 = Color( 1.0, 0.5, 0.2 )
var color2 = Color( 0.2, 0.5, 1.0 )
var mat : StandardMaterial3D = null
var estado : bool = false ## va alternando al ser clickado.

func _ready(): ## crear un material con 'color1':
    mat = StandardMaterial3D.new()
    mat.albedo_color = color1
    material_override = mat

func cuando_click(): ## cambiar el color al ser clickado:
    print("El DONUT ha sido clickado")
    estado = not estado
    if estado : mat.albedo_color = color2
    else: mat.albedo_color = color1
    material_override = mat
```

Problema: intersección rayo-triángulo (1/2)

Problema 9.2:

Una posibilidad para hacer selección en mallas de triángulos es usar cálculo de intersecciones entre un rayo (una semirrecta que pasa por el centro de un pixel) y cada uno de los triángulos de la malla. Diseña un algoritmo en pseudo-código para el cálculo de intersecciones entre un rayo y un triángulo:

- El rayo tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla \mathbf{o} , y como vector de dirección la tupla \mathbf{d} (la suponemos normalizada).
- Las coordenadas del mundo de los vértices del triángulo son $\mathbf{v}_0, \mathbf{v}_1$ y \mathbf{v}_2 .
- El algoritmo debe de indicar si hay intersección o no, y, en caso de que la haya, calcular las coordenadas del mundo del punto de intersección.
(ver la siguiente transparencia).

Problema: intersección rayo-triángulo (2/2)

Problema 9.2 (continuación):

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe $t > 0$ tal que el punto $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$ está en dicho plano. Equivale a decir que el vector $\mathbf{p}_t - \mathbf{v}_0$ es perpendicular a la normal al plano \mathbf{n} (el producto escalar de ambos es nulo).
2. El punto \mathbf{p}_t citado arriba está dentro del triángulo. Es decir, hay dos valores reales no negativos a y b (con $0 \leq a + b \leq 1$) tales que el vector $\mathbf{p}_t - \mathbf{v}_0$ es igual a $a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$.

(a los tres valores a , b y $c \equiv 1 - b - a$ se les llama *coordenadas baricéntricas* de \mathbf{p}_t en el triángulo, se usan en ray-tracing).

Problema: cálculo de rayos para selección

Problema 9.3:

Para implementar la selección usando intersecciones es necesario calcular el rayo que tiene como origen el observador y pasa por centro del pixel donde se ha hecho click. Escribe el pseudo-código del algoritmo que calcula el rayo a partir de las coordenadas del pixel donde se ha hecho click:

- Tenemos una vista perspectiva, y conocemos los 6 valores l, r, t, b, n, f usados para construir la matriz de proyección.
- También conocemos el marco de coordenadas de vista, es decir, las tuplas $\mathbf{x}_{ec}, \mathbf{y}_{ec}$ y \mathbf{z}_{ec} con los versores y la tupla \mathbf{o}_{ec} con el punto origen (todos en coordenadas del mundo).
- El viewport tiene w columnas y f filas de pixels. Se ha hecho click en el pixel de coordenadas enteras x_p e y_p

El algoritmo debe producir como salida las tuplas \mathbf{o} y \mathbf{d} (normalizado) que definen el rayo.

Fin de transparencias.

Informática Gráfica: Teoría. Sesión 10. Realismo y Ray-tracing.

Carlos Ureña

2025-26

Grado en Informática y Matemáticas

Grado en Informática y Administración y Dirección de Empresas

Dpt. Lenguajes y Sistemas Informáticos

ETSI Informática y de Telecomunicación

Universidad de Granada

1. Técnicas realistas en rasterización
2. Ray tracing

Informática Gráfica, curso 2025-26.
Teoría. Sesión 10. Realismo y Ray-tracing.

Sección 1.
Técnicas realistas en rasterización.

Informática Gráfica, curso 2025-26.
Teoría. Sesión 10. Realismo y Ray-tracing.
Sección 1. Técnicas realistas en rasterización

Subsección 1.1.
Mipmaps..

- 1.1. Mipmaps.
- 1.2. Perturbación de la normal
- 1.3. Sombras arrojadas
- 1.4. Superficies transparentes. Refracción.
- 1.5. Superficies especulares

En muchos casos la resolución a la que se ve la textura no coincide con la de la imagen sintetizada:

- ▶ Si la resolución es menor (objeto lejano), en un pixel se proyectan muchos texels.
- ▶ Si la resolución es mayor, un texel se proyecta en muchos pixels.

en ambos casos el efecto es una pérdida de realismo. El primer problema se puede solucionar usando anti-aliasing, o de forma mucho más eficiente usando la técnica de *mipmaps*

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 5 de 94.

Acceso a los *mipmaps*

Durante el sombreado, en cada punto **p** a sombrear es necesario saber qué versión de la textura debemos de leer:

- ▶ Se usará la textura M_i , donde i crece linealmente con el logaritmo de la distancia d entre **p** y el observador (menos resolución a mayor distancia).
- ▶ Esta solución puede presentar cambios bruscos de la resolución al pasar bruscamente de una resolución a otra en pixels cercanos. La solución consiste en interpolar entre las dos texturas más apropiadas en función de $\log(d)$

Un *ipmap* (*de multum in parvo maps*) es una serie de $n + 1$ texturas (bitmaps) obtenida a partir de una imagen o textura de $2^n \times 2^n$ texels.

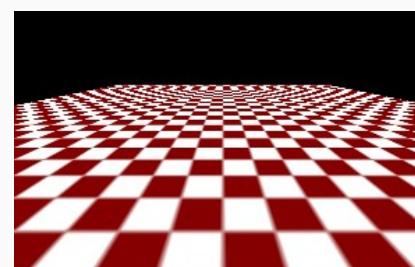
- ▶ La primera imagen (imagen M_0) coincide con la original
- ▶ La i -ésima imagen (M_i) tiene como resolución $2^{n-i} \times 2^{n-i}$ texels.
- ▶ Cada texel de la imagen $i + 1$ (M_{i+1}) se obtiene a partir de cuatro texels de la imagen número i , promediándolos:

$$M_{i+1}[j, k] = \frac{1}{4} (M_i[2j, 2k] + M_i[2j + 1, 2k] + M_i[2j, 2k + 1] + M_i[2j + 1, 2k + 1])$$

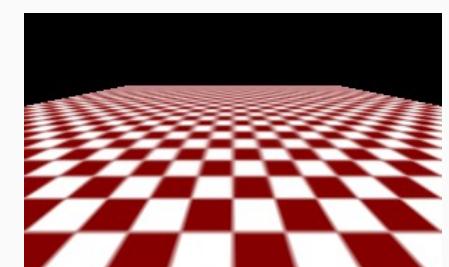
GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 6 de 94.

Ejemplo de *mipmapping*

En la parte más cercana se usa en ambos casos la textura original. Con *mipmaps*, a distancias mayores se usan sucesivamente texturas de menos resolución.



sin mipmapping



con mipmapping

http://www.flipcode.com/archives/Advanced_OpenGL_Texture_Mapping.shtml

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 7 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 8 de 94.

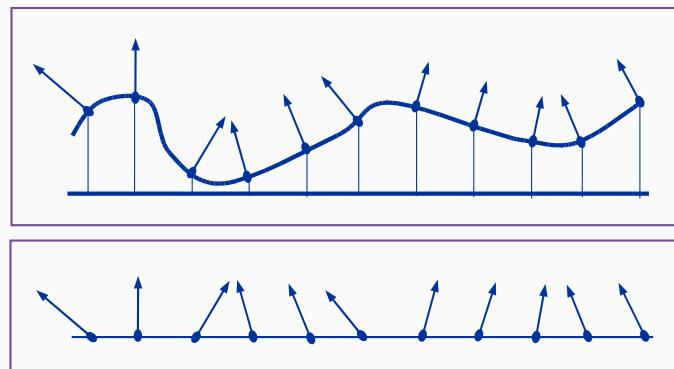
Subsección 1.2. Perturbación de la normal.

Algunos tipos de superficies presentan cambios de orientación a pequeña escala (rugosidades)

- ▶ Esto se puede reproducir con mallas de polígonos con muchos polígonos pequeños, o con polígonos de detalle de diferente orientación. En cualquier caso, la complejidad en tiempo y espacio del proceso de rendering es muy alta.
- ▶ Una solución consiste en usar un texture para modificar a pequeña escala el vector normal que se usa en el MIL, a esto se le llama *mapas de perturbación de la normal (bump-maps)*.

Rugosidades definidas por un campo de alturas

Es necesario usar una función real f_h , tal que para cada par de coordenadas de textura (u, v) , el valor real $f_h(u, v)$ se interpreta como la altura de la superficie rugosa respecto del plano del polígono en el punto de coordenadas de textura (u, v)



Codificación del campo de alturas

Para evaluar $f_h(u, v)$ dados u y v se pueden usar dos opciones:

- ▶ f_h puede representarse como una función con una expresión analítica conocida y evaluable con algún algoritmo que tiene a u y v como datos de entrada (se llaman *texturas procedurales*).
- ▶ la opción más usual es que f_h este codificada como una texture cuyos texels son valores escalares (tonos de gris) que codifican la altura. Para evaluar $f_h(u, v)$ se usa el mismo método visto para acceso a textures en la sección anterior (se usan los texels más cercanos a (u, v) en el espacio de coords. de textura).

El procedimiento de perturbación de la normal usa como parámetros las derivadas parciales de f_h (d_u y d_v):

$$d_u = \frac{\partial f_t(u, v)}{\partial u} \quad d_v = \frac{\partial f_t(u, v)}{\partial v}$$

- si f_h está definido por una función analítica conocida y derivable, estas derivadas se pueden conocer evaluando las expresiones de las derivadas parciales de f_h .
- si f_h está codificada con una textura, se usan diferencias finitas

Cuando el campo de alturas f_t se codifica con una textura de grises, los valores de d_u y d_v se deben aproximar por diferencias finitas:

$$d_u \approx k \frac{f_h(u + \Delta, v) - f_h(u - \Delta, v)}{2\Delta}$$

$$d_v \approx k \frac{f_h(u, v + \Delta) - f_h(u, v - \Delta)}{2\Delta}$$

donde:

- Δ es usualmente del orden de $1/n_t$ (n_t = resol. de la textura).
- k es un valor real que sirve para atenuar o exagerar el relieve

La superficie como una función de las c.t.

Los puntos de los polígonos que forman las superficies de los objetos pueden interpretarse como una función f_p de las coordenadas de textura, es decir, si las coord. de textura de un punto q son (u, v) , entonces:

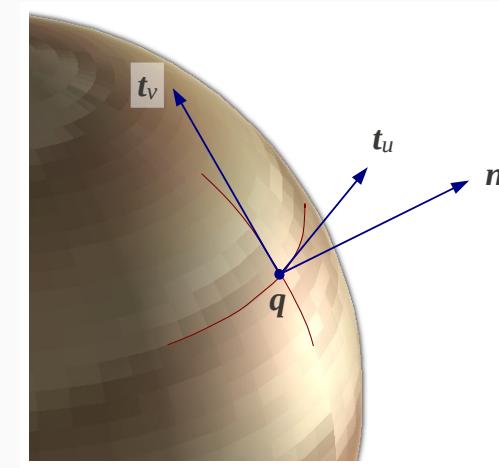
$$q = f_p(u, v)$$

para calcular la normal modificada es necesario conocer las derivadas parciales de f_p (dos vectores t_u y t_v)

$$t_u = \frac{\partial f_p(u, v)}{\partial u} \quad t_v = \frac{\partial f_p(u, v)}{\partial v}$$

Las tangentes y la normal

A los vectores t_u y t_v se les suele llamar *tangente* y *bitangente*. Ambos definen un plano tangente, perpendicular a la normal.



Estos vectores son tangentes a la superficie del objeto, ya que la normal original \mathbf{n} es colineal con $\mathbf{t}_u \times \mathbf{t}_v$. Existen varias alternativas para obtenerlos:

- ▶ Para objetos sencillos, los vectores tangentes son constantes o muy fáciles de calcular
- ▶ Para mallas de polígonos:
 - ▶ Se pueden calcular como constantes en cada polígono, a partir de las coordenadas de textura.
 - ▶ Se pueden asignar a los vértices (igual que las c.t.) y realizar una interpolación en el interior de los polígonos (igual que se interpola la normal).

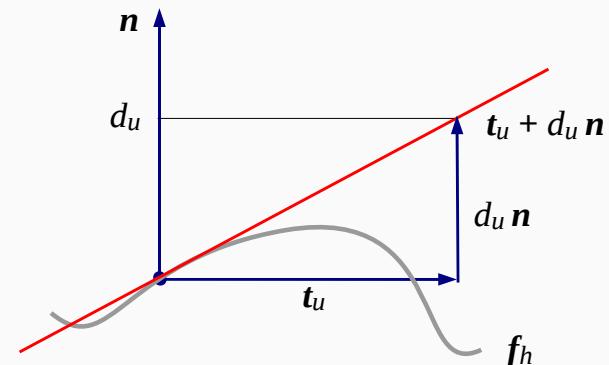
GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 17 de 94.

la normal modificada \mathbf{n}' es perpendicular a estos dos vectores, por tanto se calcula usando su producto vectorial (y normalizando)

$$\mathbf{n}' = \frac{\mathbf{n}''}{\|\mathbf{n}''\|} \quad \text{donde: } \mathbf{n}'' = \mathbf{t}'_u \times \mathbf{t}'_v$$

Los vectores tangentes \mathbf{t}'_u y \mathbf{t}'_v a la superficie rugosa son:

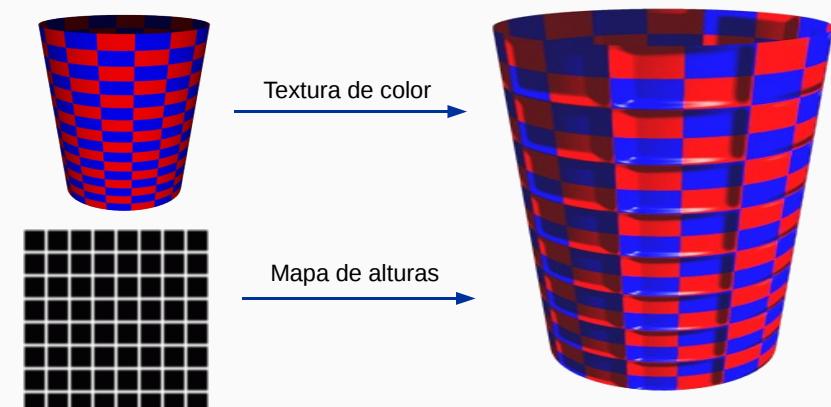
$$\mathbf{t}'_u = \mathbf{t}_u + d_u \mathbf{n} \quad \mathbf{t}'_v = \mathbf{t}_v + d_v \mathbf{n}$$

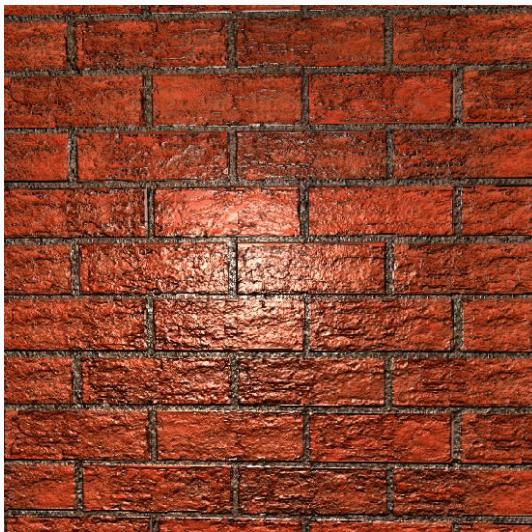


GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 18 de 94.

Ejemplo de texturas + perturbación de la normal (1)

Imágenes de Fredo Durand y Barb Curtler:
<http://groups.csail.mit.edu/graphics/classes/6.837/>





GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 21 de 94.

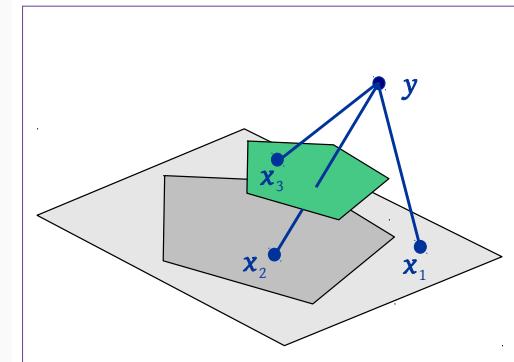
Sombras arrojadas y el MIL

Ninguna de las técnicas anteriores tiene en cuenta la existencia de sombras arrojadas.

- ▶ Se supone que todas las fuentes son visibles desde todos los puntos de la superficie, lo cual no siempre es cierto.
- ▶ Si asumimos que los polígonos son opacos, y las fuentes puntuales (o direccionales), para cada punto en una superficie y para cada fuente de luz, el punto y la fuente pueden ser mutuamente visibles o no.
- ▶ Cuando la fuente no ilumina el punto, el sumando del MIL correspondiente a la fuente no debe añadirse para obtener el color reflejado.

La función de visibilidad V

La visibilidad de la fuente de luz (en y) está controlada por la función V :



$$V(\mathbf{x}_1, \mathbf{y}) = 1 \quad V(\mathbf{x}_2, \mathbf{y}) = 0 \quad V(\mathbf{x}_3, \mathbf{y}) = 1$$

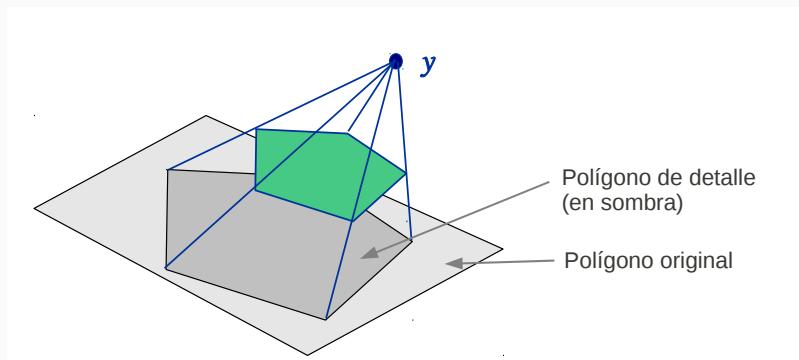
El problema de las sombras arrojadas es, por tanto, semejante al problema de la visibilidad:

- ▶ Se pueden usar algoritmos con precisión de objetos: se producen en la salida los polígonos (parte de los originales) iluminados por (visibles desde) las fuentes de luz.
- ▶ Se pueden usar algoritmos con precisión de imagen: se obtiene el primer punto visible en el centro de cada pixel de un plano de visión asociado a una fuente de luz.

El papel del observador lo juega la fuente de luz. Puede ser posicional (observador a distancia finita) o direccional (observador a distancia infinita: proyección ortogonal).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 25 de 94.

Algoritmo de fuerza bruta (2)



- ▶ tiene complejidad cuadrática con el número de polígonos
- ▶ se puede usar solo para un único polígono receptor y unos pocos que arrojan sombras.

Supondremos escenas formadas por poliedros opacos delimitados por caras planas o polígonos planos individuales.

- ▶ El algoritmo más sencillo consiste en proyectar todos los polígonos contra todos, usando la fuente de luz como foco.
- ▶ Para cada par de polígonos P y Q se calcula el polígono de sombra arrojada S que proyecta P sobre Q (si hay alguna), y se recorta S usando Q como polígono de recorte.
- ▶ Los polígonos producidos se tratan como polígonos de detalle. Son polígonos superpuestos a los originales en los cuales la fuente de luz no es visible.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 26 de 94.

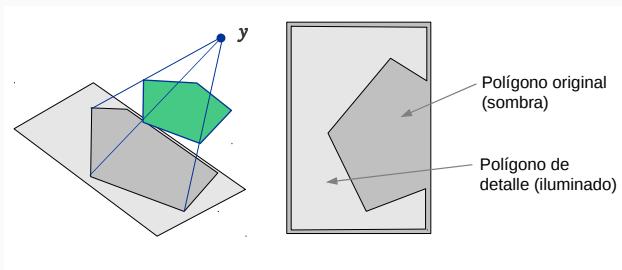
Algoritmo de Weiler-Atherton-Greenberg

Otros algoritmos de sombras arrojadas (más eficientes) están basados en algoritmos de eliminación de partes ocultas ya existentes. Un ejemplo es el algoritmo de Weiler-Atherton-Greenberg (1978) para sombras arrojadas:

- ▶ Se usa el algoritmo de Weiler-Atherton para eliminación de partes ocultas
- ▶ Se produce un modelo con polígonos iluminados asociados a los originales (son también polígonos de detalle).
- ▶ La complejidad en tiempo es mucho menor que cuadrática en el caso medio.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 27 de 94.

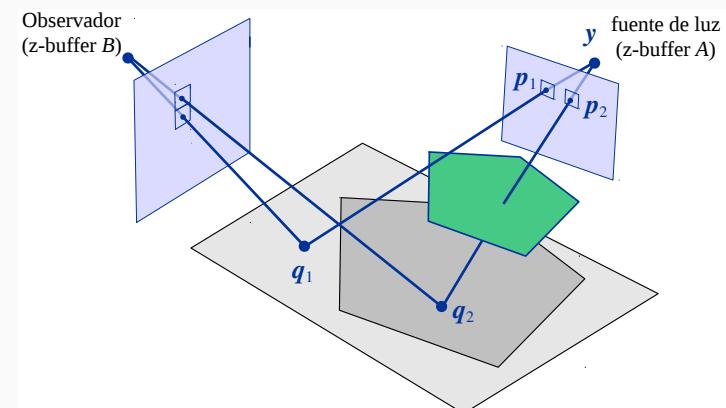
GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 28 de 94.



En general, los algoritmos con precisión de objetos para sombras son:

- ▶ muy complejos en tiempo para escenas complejas
- ▶ para algunas aplicaciones son los más idóneos (cuando se necesita un resultado en forma de dibujo vectorial).

Otra posibilidad (mucho más eficiente) es usar Z-buffer para sombras arrojadas:

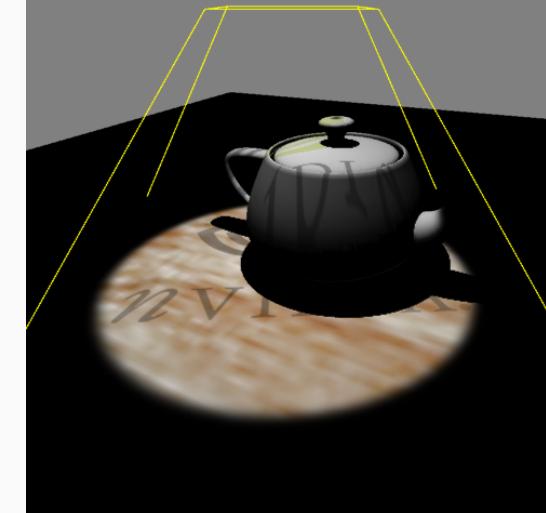


Z-buffer para sombras arrojadas (2)

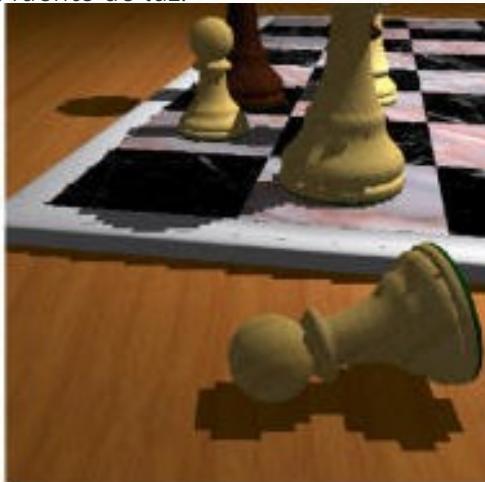
- ▶ En la primera pasada se calcula el Z-buffer A asociado a la fuente de luz (se proyectan los objetos contra la fuente)
- ▶ La segunda pasada es semejante al Z-buffer normal, se calcula el Z-buffer B , para cada punto visible q_i desde el observador en un pixel, se debe calcular el color con el que se ve q_i , y por tanto es necesario comprobar si es visible desde la fuente, para ello:
 - ▶ se calcula p_i (la proyección de q_i en el plano de visión asociado a la fuente de luz)
 - ▶ se accede al pixel del Z-buffer A correspondiente a p_i , que contiene una distancia d
 - ▶ si $d < \|q_i - y\|$, entonces q_i no está iluminado (este es el caso de la figura), en otro caso q_i sí está iluminado.

Ejemplo de Z-buffer para sombras

(se proyecta una textura desde la fuente en los objetos):



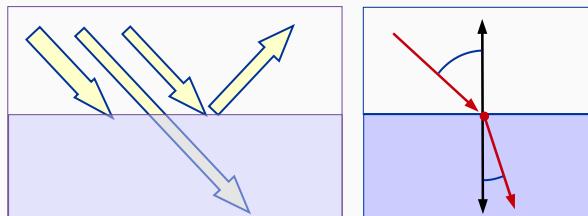
son visibles si el observador está cerca de ellas en comparación con la distancia a la fuente de luz:



GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 33 de 94.

Materiales transparentes

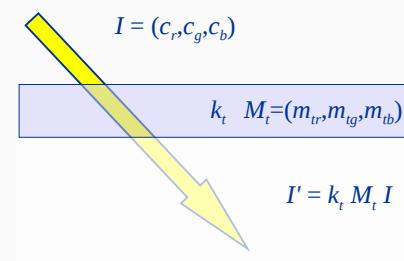
Hay objetos sólidos o líquidos que permiten pasar (además de reflejar o absorber) algunos fotones de la luz que los alcanza. Su estructura atómica permite a los rayos de luz viajar en línea recta.



la luz cambia de dirección debido a su progreso más lento en estos medios (debido al retraso por múltiples eventos de dispersión de fotones)

Cambio del color en la refracción

Al pasar por un objeto delgado transparente, la cantidad de luz que no es absorbida en el medio (y atraviesa el objeto) depende de la longitud de onda:



- ▶ La fracción global de luz refractada es k_t , que está entre 0 y 1
- ▶ En cada longitud de onda se refracta una fracción distinta, en RGB estas fracciones son un color $M_t = (m_{tr}, m_{tg}, m_{tb})$

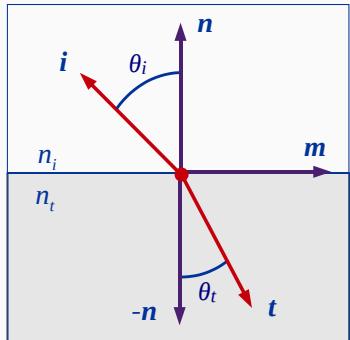
Por tanto, estos materiales están caracterizados por k_t y M_t

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 35 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 36 de 94.

El vector \mathbf{t} puede calcularse a partir de \mathbf{n} , \mathbf{i} , y los índices de refracción n_i y n_t , teniendo en cuenta la ley de Snell:

$$n_i \sin \theta_i = n_t \sin \theta_t$$



$$\begin{aligned}\mathbf{t} &= (r \cos \theta_i - \cos \theta_t) \mathbf{n} - r \mathbf{i} \\ \cos \theta_i &= \mathbf{i} \cdot \mathbf{n} \\ \cos \theta_t &= \sqrt{1 - r^2 [1 - (\mathbf{i} \cdot \mathbf{n})^2]} \\ r &= n_i / n_t\end{aligned}$$

Si $1 < r^2 [1 - (\mathbf{i} \cdot \mathbf{n})^2]$ entonces no hay refracción (hay *reflexión interna total*). Solo puede ocurrir cuando $n_t < n_i$, para $\theta_i > \theta_{\max}$.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 37 de 94.

El método de Z-buffer solo puede tener en cuenta, para un pixel, los colores de los puntos en el proyector o rayo que pasa por el centro del pixel hacia el observador.

- ▶ Cuando $n_i \neq n_t$ los rayos se desvían, y es lo que no puede reproducirse con Z-buffer
- ▶ Si suponemos que $n_i = n_t$, entonces no hay cambio de dirección debida a la refracción, y por tanto $\mathbf{t} = -\mathbf{i}$
- ▶ Con esta simplificación, se puede adaptar el método de Z-Buffer para incluir polígonos transparentes.

a continuación vemos un esbozo del método

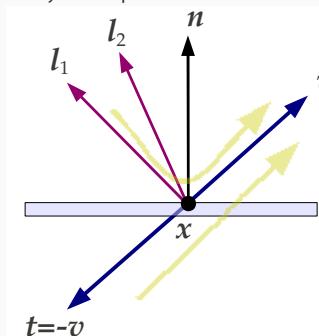
Z-buffer adaptado a superficies transparentes

El algoritmo dibuja primero los polígonos opacos, y después los transparentes o semi transparentes (en cualquier orden)

- 1: Inicializar Z-buffer (Z) e Imagen (I)
- 2: **for** cada polígono opaco P **do**
- 3: Rasterizar P , actualizando Z e I
- 4: **for** cada polígono transparente Q **do**
- 5: k_t = fracción de luz refractada de Q
- 6: M_t = color transparente de Q
- 7: **for** cada pixel (x, y) ocupado por Q **do**
- 8: **if** Q es visible en (x, y) **then**
- 9: $I[x, y] = k_t M_t I[x, y]$

Combinación de reflexión y refracción

En la superficie entre una lámina de material transparente y el espacio (vacío) entre objetos puede también reflejarse la luz:

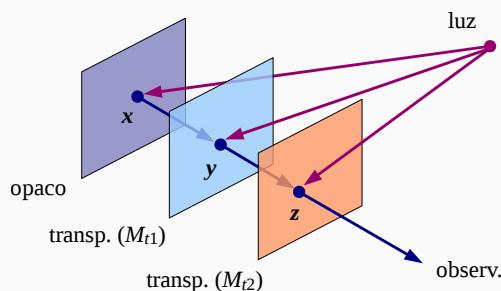


Si $k_t > 0$, al MIL debe sumársele la luz refractada proveniente del otro lado del polígono, en la dirección de v

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 39 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 40 de 94.

El color I que percibe el observador depende de los colores I_x , I_y y I_z reflejados en los puntos x , y y z :



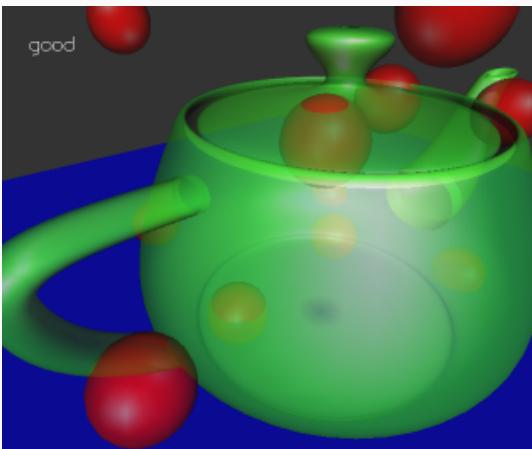
Este color depende del orden de los polígonos:

$$I = I_z + M_{t2}I_y + M_{t1}M_{t2}I_x \neq I_z + M_{t1}I_x + M_{t2}M_{t1}I_y$$

Los polígonos transp. deben dibujarse en orden de Z:

-
- 1: Inicializar Z-buffer (Z) e Imagen (I)
 - 2: **for** cada polígono opaco P **do**
 - 3: Rasterizar P , actualizando Z e I
 - 4: **for** cada polígono transparente Q (en orden de Z) **do**
 - 5: k_t = fraccion de luz refractada de Q
 - 6: M_t = color transparente de Q
 - 7: **for** cada pixel (x, y) ocupado por Q **do**
 - 8: **if** Q es visible en (x, y) **then**
 - 9: I_m = resultado de evaluar MIL
 - 10: $I[x, y] = I_m + k_t M_t I[x, y]$
-

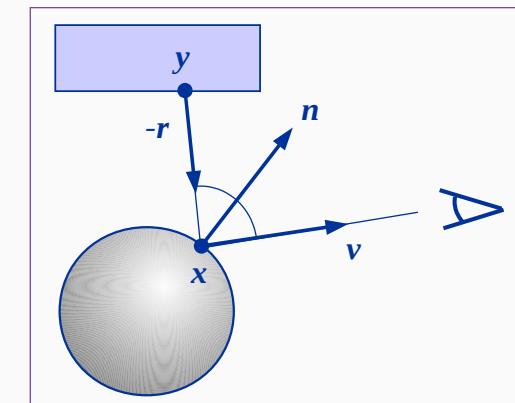
Ejemplo de materiales transparentes



Algunos objetos pulidos reflejan la luz de forma especular perfecta, como los espejos planos

- ▶ La componente especular perfecta es una componente más del modelo de iluminación local, que se suma a las anteriores (se suele dar en combinación con la refractada en los objetos de cristal).
- ▶ Este efecto no puede reproducirse con ninguno de los métodos que hemos visto, pues la iluminación no procede de la dirección del rayo central a un pixel, sino de otras direcciones.

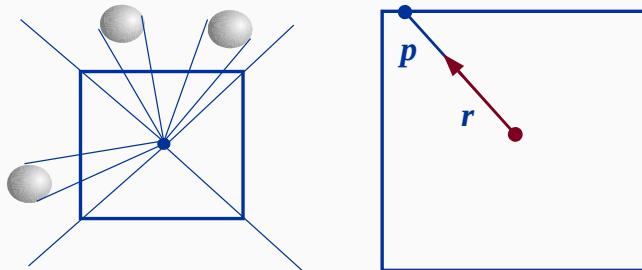
Si la esfera es perfectamente reflectante, el color de x visto desde v es igual al color de y visto desde la dirección $-r$ (el vector reflejado r , cambiado de signo).



GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 45 de 94.

Mapas de entorno tipo caja (box map)

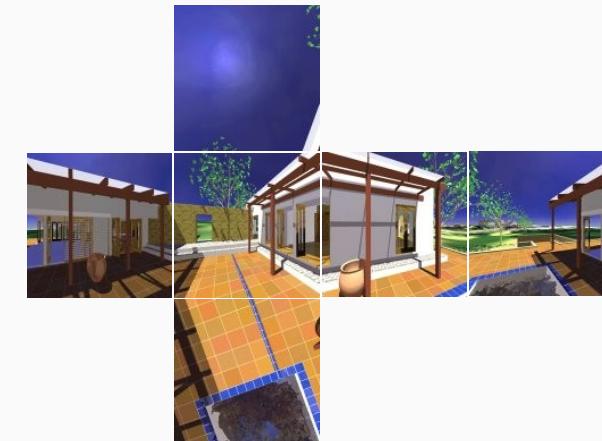
En esta técnica, el entorno se proyecta en las 6 caras de un cubo centrado en el objeto reflectante, obteniéndose 6 texturas.



En tiempo de rendering, el vector r se proyecta sobre la cara que corresponda (se calcula p), y se obtienen el color RGB del texel que contiene a p .

Texturas para mapas de entorno tipo caja

Ejemplo de 6 texturas para mapas de entorno

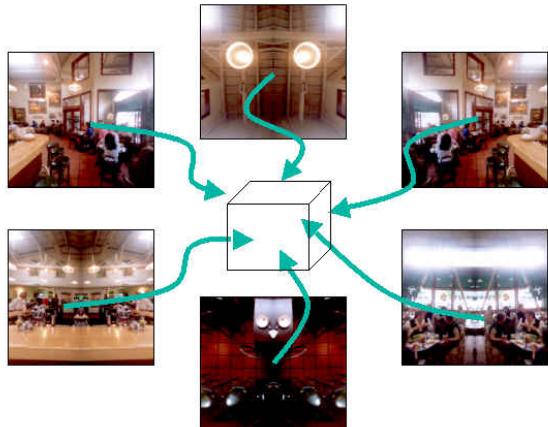


http://developer.nvidia.com/object/cube_map_ogl_tutorial.html

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 47 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 48 de 94.

Tambien es posible usar fotografías de un entorno real



http://developer.nvidia.com/object/cube_map_ogl_tutorial.html

Una sola imagen codifica el color reflejado para todas las posibles orientaciones de la normal



se asume una proyección ortográfica fija, en la cual el vector v es constante y paralelo al eje Z.

Panoramás equirectangulares

Es una sola imagen que codifica, en cada texel (u, v) , la irradiancia en una dirección de coordenadas polares $\alpha = au$ y $\beta = bv$:



se suelen obtener a partir de múltiples fotografías de un entorno. A su vez, pueden servir para crear mapas de entorno esféricicos.

Mapa de entorno esféricico

Ejemplo del mapa de entorno esféricico anterior en la tetera:



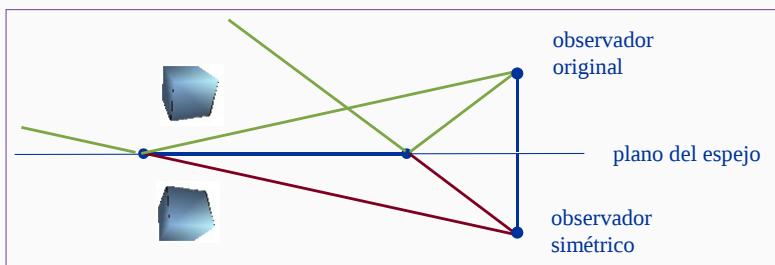
Ejemplo de combinación con texturas y perturbación de la normal.
Además, la tetera se muestra en el entorno que refleja:



GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 53 de 94.

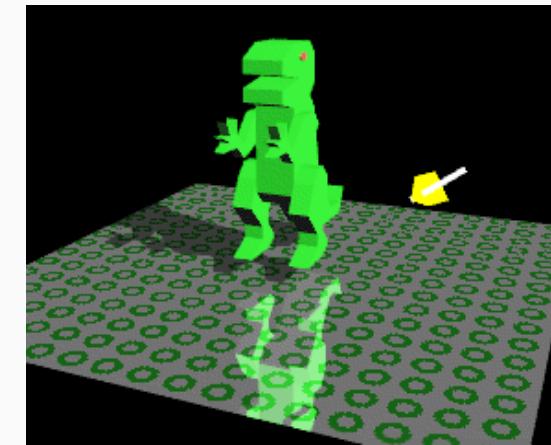
Reflexión en espejos planos

En estos objetos, la escena reflejada es simétrica respecto de la original respecto del plano del espejo:



Se pueden reproducir las reflexiones sintetizando la imagen vista por una cámara simétrica respecto de la original.

Ejemplo de duplicación de entorno:



GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 55 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 56 de 94.

- 2.1. El algoritmo de Ray-Tracing
- 2.2. Intersecciones rayo-objeto y rayo-escena
- 2.3. Problemas: Cálculo de intersecciones
- 2.4. Ejemplos

Introducción

Hemos visto bastantes efectos:

- ▶ Superficies difusas y pseudo-especulares
- ▶ Texturas y mapas de perturbación de la normal
- ▶ Sombras arrojadas
- ▶ Superficies transparentes
- ▶ Superficies especulares perfectas

Considerarlos todos en visualización por rasterización lleva a software que es bastante complicado de implementar. Además los tiempos de síntesis de imágenes pueden hacerse bastante altos.

La técnica de *Ray-Tracing*

Existe un algoritmo no muy eficiente en tiempo, pero bastante sencillo, que tiene en cuenta todos los efectos anteriores:

- ▶ Este algoritmo es el algoritmo de *Ray-Tracing* (*seguimiento de rayos*, usualmente traducido por *trazado de rayos*)
- ▶ Describo completamente por primera vez por Turner Whitted en 1979-80.
- ▶ Está basado en la EPO por Ray-Casting, combinada con evaluación del MIL
- ▶ Es conceptualmente muy sencillo, y fácil de implementar.
- ▶ Obtiene un grado de realismo muy superior a Z-buffer, a costa de tiempos de cálculo usualmente más altos.

Frente a rasterización, Ray-Tracing puede visualizar objetos

- ▶ curvos (con superficie definida por ecuaciones implícitas).
- ▶ especulares perfectos (que reflejan el entorno).
- ▶ transparentes (con índices de refracción arbitrarios).
- ▶ con sombras arrojadas por otros.

Existen diversos algoritmos basados en Ray-Tracing con funcionalidad adicional:

- ▶ **Ray-Tracing distribuido**: fuentes de luz extensas, profundidad de campo, desenfoque de movimiento (*motion-blur*).
- ▶ **Path-Tracing**: iluminación indirecta o global.

El algoritmo de **Path-Tracing** y derivados se usa en generación por ordenador de películas y efectos especiales.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 61 de 94.

MIL de Ray-Tracing. Iluminación directa.

La iluminación directa $L_{\text{dir}}(\mathbf{p}, \mathbf{v}, \mathbf{l})$ es la radiancia reflejada en \mathbf{p} hacia \mathbf{v} debida a iluminación directa proveniente de una fuente de luz que está en la dirección \mathbf{l} . Se define así:

$$\begin{aligned} L_{\text{dir}}(\mathbf{p}, \mathbf{v}, \mathbf{l}) &\equiv k_a(\mathbf{p}) \cdot C(\mathbf{p}) \\ &+ k_d(\mathbf{p}) \cdot C(\mathbf{p}) \cdot \max(0, \mathbf{n} \cdot \mathbf{l}) \\ &+ k_s(\mathbf{p}) \cdot d_i \cdot [\max(0, \mathbf{r} \cdot \mathbf{v})]^e \end{aligned}$$

esta fórmula incorpora las componentes ambiental, difusa y pseudo-especular o *glossy*, de forma similar a como vimos en el tema 3 para rasterización. Aquí:

- ▶ \mathbf{r} ≡ vector reflejado en \mathbf{p} (depende de \mathbf{v} y \mathbf{n}).
- ▶ \mathbf{n} ≡ normal en \mathbf{p} (depende de \mathbf{p} y O).
- ▶ k_a, k_d, k_s ≡ parámetros del material en \mathbf{p} .
- ▶ $C(\mathbf{p})$ ≡ color del objeto en \mathbf{p}

Ray-Tracing calcula la radiancia $L_{\text{in}}(\mathbf{q}, \mathbf{u})$ incidente sobre un punto \mathbf{q} proveniente de una dirección \mathbf{u} (un vector unitario).

$$\begin{aligned} L_{\text{in}}(\mathbf{q}, \mathbf{u}) &= L(\mathbf{p}, \mathbf{v}) \\ &= M_E(\mathbf{p}) + L_{\text{ind}}(\mathbf{p}, \mathbf{v}) + \sum_{i=0}^{n_L-1} v_i \cdot S_i \cdot L_{\text{dir}}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) \end{aligned}$$

donde:

- ▶ El punto \mathbf{p} es el primero visible desde \mathbf{q} en la dirección de \mathbf{u} .
- ▶ El vector \mathbf{v} es la dirección de salida de radiancia (es $-\mathbf{u}$).
- ▶ $M_E(\mathbf{p})$ es la emisividad en \mathbf{p} .
- ▶ El valor v_i es la visibilidad de la i -ésima fuente de luz (vale 0 o 1).
- ▶ Las funciones L_{dir} y L_{ind} representan la **iluminación directa** y la **iluminación indirecta**, respectivamente.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 62 de 94.

MIL de Ray-Tracing. Iluminación indirecta.

El término $L_{\text{ind}}(\mathbf{p}, \mathbf{v})$ incluye la radiancia ambiente global, la reflejada perfectamente y la refractada. Se define **recursivamente** en términos de L_{in} :

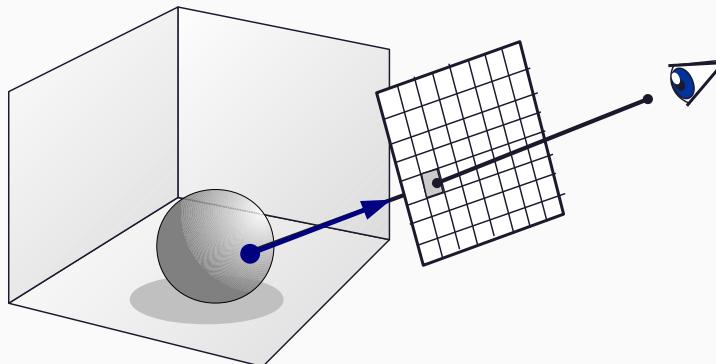
$$L_{\text{ind}}(\mathbf{p}, \mathbf{v}) \equiv A_G(\mathbf{p}) + k_{ps}(\mathbf{p})M_{PS}(\mathbf{p})L_{\text{in}}(\mathbf{p}, \mathbf{r}) + k_t(\mathbf{p})M_T(\mathbf{p})L_{\text{in}}(\mathbf{p}, \mathbf{t})$$

- ▶ $A_G(\mathbf{p})$ ≡ radiancia ambiente global (suple ilum. indirecta).
- ▶ $k_t(\mathbf{p})$ ≡ fracción de luz refractada en \mathbf{p} .
- ▶ $k_{ps}(\mathbf{p})$ ≡ fracc. de luz refl. de forma especular perfecta en \mathbf{p} .
- ▶ $M_{PS}(\mathbf{p})$ y $M_T(\mathbf{p})$ son ternas RGB (con valores en $[0, 1]$) que permiten modular el color de la componente refejada perfectamente o refractada.
- ▶ \mathbf{r} ≡ vector reflejado en \mathbf{p} (depende de \mathbf{v} y \mathbf{n}).
- ▶ \mathbf{t} ≡ vector refractado en \mathbf{p} (depende de \mathbf{v} y \mathbf{n}).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 63 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 64 de 94.

Los pixels se procesan secuencialmente, en cada uno se crea un rayo (llamado *rayo primario* o *rayo de cámara*) y se determina el primer objeto visible por Ray-Casting:



GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 65 de 94.

La función RAYTRACING

Esta función calcula la radiancia incidente sobre el punto **o**, proveniente de la dirección **u** (como una terna RGB). El entero **n** es el nivel de profundidad en las llamadas recursivas.

```

1: function RAYTRACING( punto o, vector u, entero n)
2:   if n > max then // si se ha superado máximo nivel de recursión
3:     return (0,0,0)           // devolver radiancia nula
4:   O := primer objeto visible desde o en la dir. u // (por ray-casting)
5:   if no existe ningun objeto visible then
6:     return radiancia de fondo correspondiente a u
7:   p := punto de O intersecado
8:   return EVALUAMILREC( O, p, -u, n )

```

El pseudocódigo del algoritmo, que recorre todos los pixels, puede quedar así:

```

1: o := posición del observador, en coords. del mundo
2: for cada pixel  $(i, j)$  de la imagen do
3:   q := punto central (en WCC) del pixel  $(i, j)$ 
4:   u := vector desde o hasta q normalizado
5:   rad := RAYTRACING(o,u,1)
6:   fijar el pixel  $(i, j)$  al valor rad

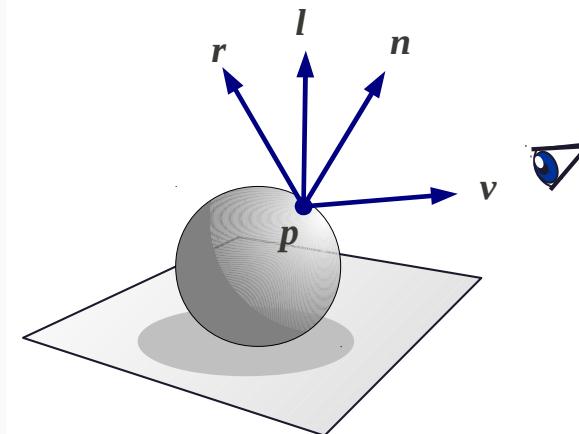
```

- ▶ La función RAYTRACING es recursiva, devuelve un color, y tiene un parámetro que sirve para que la recursión no se haga infinita.
- ▶ Los colores de los pixels no están acotados, así que puede ser necesario un paso de post-proceso para normalizar la imagen.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 66 de 94.

Evaluación del MIL

Una vez se conoce el punto **p**, se obtienen la normal **n**, y los parámetros del MIL, que es evaluado:

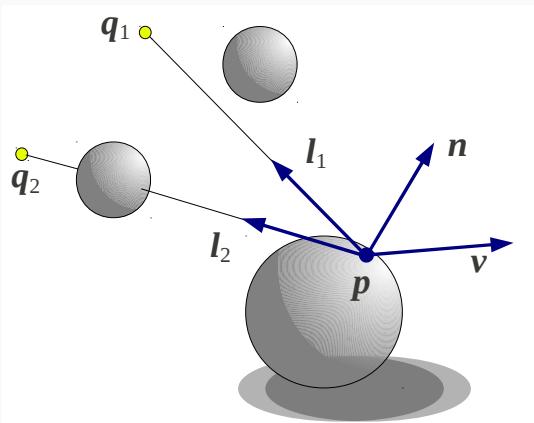


podemos considerar objetos curvos (esferas, cilindros, conos, etc..)

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 67 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 68 de 94.

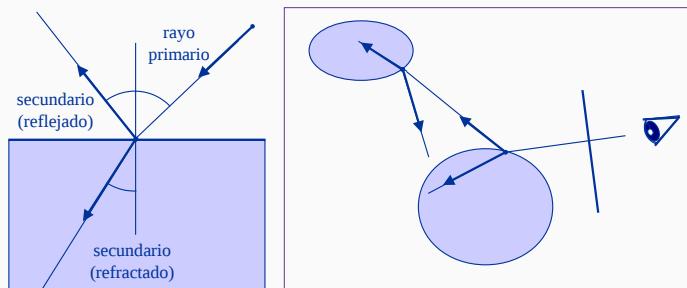
Este método permite incorporar sombras arrojadas, usando Ray-Casting para visibilidad. Se comprueba si un segmento de recta desde p hasta (o hacia) la fuente interseca algún objeto de la escena, es decir, se evalua $V(p, q_i)$



GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 69 de 94.

Rayos secundarios y recursividad

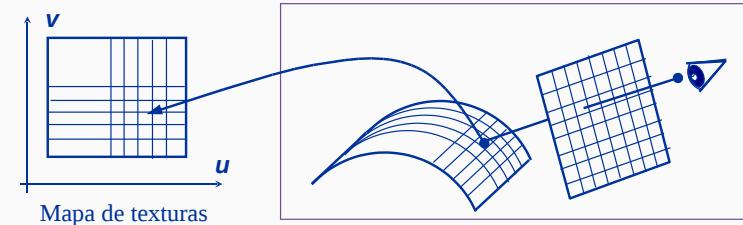
También es posible tener en cuenta superficies perfectamente especulares y/o perfectamente transparentes. Esto se hace creando *rayos secundarios*, e invocando recursivamente al algoritmo.



Da lugar a un árbol de rayos asociado al árbol de llamadas recursivas.

El objeto en el que está p puede tener asociadas texturas (o mapas de pert. de la normal)

- ▶ A partir de p se obtienen las coordenadas (u, v) en el espacio de la textura
- ▶ A partir de (u, v) se consulta la textura o texturas asociadas al objeto.



GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 70 de 94.

La función EVALUAMILREC

La función que evalua el MIL (EVALUAMILREC) llama recursivamente a la función de RAYTRACING,

```

1: function EVALUAMILREC( O, p, v, n )
2:   Obtener paráms. del material de O en p (n, C, ka, kd, ks, kps, MPS, MT, kt)
3:   Obtener parámetros de fuentes de luz (nL, li, Si)
4:   rad := ME(p) + AG(p)           // emisividad y comp. ambiente global
5:   for i := 0 to nL - 1 do          // para cada fuente de luz
6:     if la fuente i es visible desde p then // (que sea visible)
7:       rad := rad + Si · DIRECTA(p, v, li) // ilum. directa
8:     if kt > 0 and n < max then
9:       t := vector refractado respecto de v
10:      rad := rad + kt · MT · RAYTRACING( p, t, n + 1 ) // componente refractada
11:     if kps > 0 and n < max then
12:       r := vector reflejado respecto de v
13:       rad := rad + kps · MPS · RAYTRACING( p, r, n + 1 ) // componente reflejada
14:   return rad

```

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 71 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 72 de 94.

Esta función evalúa L_{dir} (las componentes ambiental, difusa y especular correspondientes a una fuente de luz).

```

1: function DIRECTA( p, v, l )
2:   rad :=  $k_a \cdot C$ 
3:   if  $k_d > 0$  then
4:     rad := rad +  $k_d \cdot C \cdot \max(0, \mathbf{n} \cdot \mathbf{l})$ 
5:   if  $k_s > 0$  then
6:     rad := rad +  $k_s \cdot d_i \cdot [\max(0, \mathbf{r} \cdot \mathbf{v})]^e$ 
7:   return rad

```

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 73 de 94.

Cálculo de intersecciones: rayo-objeto y rayo-escena

El algoritmo de Ray-Tracing emplea la mayor parte de su tiempo en:

- ▶ Evaluación de MIL avanzados: a los subprogramas que los evaluan se les llama *shaders*.
- ▶ Cálculo de intersecciones. Dado un punto \mathbf{o} y un vector unitario \mathbf{v} (en coordenadas de mundo, codificando una **semirecta** o **rayo**), se trata de calcular el menor valor real $t > 0$ tal que el punto $\mathbf{p}(t) \equiv \mathbf{o} + t\mathbf{v}$ está en la frontera o superficie de algún objeto de la escena.

Hay dos algoritmos fundamentales:

- ▶ Intersección rayo-objeto: cada tipo de geometría posible tiene asociado un algoritmo.
- ▶ Intersección rayo-escena: podemos recorrer exhaustivamente los objetos, pero es $O(n_o)$. Se reducen los tiempos con *indexación espacial*, que es $O(\log n_o)$.

Intersecciones rayo-objeto: método general

Calcular la intersección de un rayo (\mathbf{o}, \mathbf{v}) con un objeto cuya geometría es $O \subseteq \mathbb{R}$ consiste en obtener el mínimo valor de $t > 0$ (si hay alguno) tal que $\mathbf{o} + t\mathbf{v} \in \partial O$. El objeto O puede estar caracterizado por estas dos funciones:

- ▶ Ecuación implícita: un campo escalar F tal que si $\mathbf{p} \in \partial O$ entonces $F(\mathbf{p}) = 0$.
- ▶ Condiciones adicionales: un predicado o función lógica C tal que $\mathbf{p} \in \partial O$ si y solo si $F(\mathbf{p}) = 0$ y $C(\mathbf{p})$.

El algoritmo requiere:

1. Calcular el conjunto $S = \{t_0, t_1, \dots, t_{n-1}\}$ con las raíces de la ecuación $F(\mathbf{o} + t\mathbf{v}) = 0$.
2. Eliminar de S los t_i que no cumplen $C(\mathbf{o} + t_i\mathbf{v})$.
3. Si $S \neq \emptyset$ la solución es $t = \min(S)$. Si $S = \emptyset$ no hay inters.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 75 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 76 de 94.

Hay dos tipos de algoritmos para obtener la menor raíz t :

- ▶ Directos: t se obtiene con una fórmula explícita.

Por ejemplo: si ∂O es un subconjunto de una **superficie cuádrica**, entonces

$$F(\mathbf{o} + t\mathbf{v}) = 0 \iff \exists a, b, c \in \mathbb{R} \quad \text{t.q: } at^2 + bt + c = 0$$

- ▶ Iterativos: t se obtiene por sucesivas aproximaciones

Por ejemplo: se puede usar si F es la ***signed distance function*** (SDF) de O , es decir, si

$$F(\mathbf{p}) = s \cdot \left(\min_{\mathbf{q} \in \partial O} \|\mathbf{p} - \mathbf{q}\| \right) \quad \text{donde: } s \equiv \begin{cases} -1 & \text{si } \mathbf{p} \in O \\ +1 & \text{si } \mathbf{p} \notin O \end{cases}$$

Si no se dispone de SDF existen alternativas (Bisección, Newton, cotas de distancia, etc...).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 77 de 94.

El algoritmo *Sphere Tracing*

Un método iterativo, llamado **Sphere Tracing**, usa las SDFs F_i de los objetos. El algoritmo se basa en avanzar un punto a lo largo del rayo. A cada paso se avanza la mínima distancia del punto a los objetos de la escena, hasta que diverge o converge a un punto de intersección:

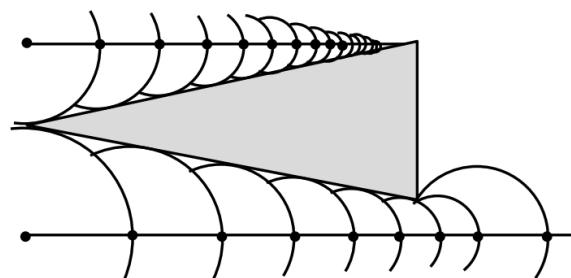


Figura obtenida del artículo *Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces*, de John C. Hart (1995). [PDF aquí](#).

Para calcular las intersecciones rayo-escena podemos usar los algoritmos directos de intersección rayo-objeto para las distintos tipos de objetos que forman la escena:

- ▶ Las escenas usualmente consisten en mallas de triángulos.
- ▶ Solución de *fuerza bruta*: comprobar la intersección con cada triángulo de la escena (tiempo en $O(n_t)$).
- ▶ La **indexación espacial** se basa en descartar partes de la escena si el rayo no interseca un volumen englobante V de todos los triángulos de esa parte de la escena.
- ▶ La intersección rayo- V es muy rápida de calcular.
- ▶ Se hace jerárquicamente (recursivamente), y se obtiene un árbol de volúmenes englobantes, con conjuntos de triángulos en los nodos terminales. Se obtiene tiempo en $O(\log n_t)$.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 78 de 94.

Intersecciones rayo-escena con *sphere tracing*

Si se disponen de SDFs F_0, \dots, F_{n-1} de los objetos O_0, \dots, O_{n-1} se puede usar este algoritmo (donde $0 \lesssim \epsilon$ es la tolerancia):

```

1: function INTERSECCIONSDF( o, v,  $O_0, \dots, O_{n-1}$  )      //  $\|\mathbf{v}\| = 1$ 
2:   p = o // mejor punto de intersección encontrado hasta ahora
3:    $d = 0$  //  $d \equiv$  máxima distancia que se puede avanzar desde p
4:   repeat
5:     p = p +  $d\mathbf{v}$  // avanzamos a lo largo del rayo
6:      $d = \min\{F_0(\mathbf{p}), \dots, F_{n-1}(\mathbf{p})\}$  // actualizamos  $d$  en p
7:   until  $\|d\| \leq \epsilon$  (o hasta un número máximo de iteraciones)
8:   if  $\|d\| \leq \epsilon$  then // si p está ya muy cerca de la superficie
9:     Hay intersección con  $O_i$  en p (con  $i$  tal que  $d \equiv F_i(\mathbf{p})$ )
10:  else
11:    No hay intersección con ningún objeto

```

Hay numerosos ejemplos de esta técnica en [Shadertoy](#)

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 79 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 80 de 94.

Subsección 2.3. Problemas: Cálculo de intersecciones.

Problema 10.1.

Supongamos que un *rayo* (una semirecta en 3D) tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla **o**, y como vector de dirección la tupla **d** (la suponemos normalizada).

Además sabemos que un *disco* de radio *r* tiene como centro el punto de coordenadas de mundo **c** y está en el plano perpendicular al vector **n**

Con estos datos de entrada, diseña un algoritmo para calcular si hay intersección entre el rayo y el disco.

(ten en cuenta las indicaciones que hay en la siguiente transparencia).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 82 de 94.

Problema: intersección rayo-disco (2/2)

Problema 10.1. (continuación)

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe $t > 0$ tal que el punto $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$ está en dicho plano. Equivale a decir que el vector $\mathbf{p}_t - \mathbf{v}_0$ es perpendicular a la normal al plano **n**.
2. El punto \mathbf{p}_t citado arriba está dentro del disco, es decir, su distancia a **c** es inferior al radio.

Problema: intersección rayo-esfera

Problema 10.2.

Diseña un algoritmo para calcular la primera intersección entre un rayo (con origen en **o** y vector **d**, normalizado) y una esfera de radio unidad y centro en el origen, si hay alguna.

Ten en cuenta que un punto cualquiera **p** está en esfera si y solo si el módulo de **p** es la unidad, es decir, si y solo si $F(\mathbf{p}) = 0$, donde F es el campo escalar definido así:

$$F(\mathbf{p}) \equiv \mathbf{p} \cdot \mathbf{p} - 1$$

Describe como podría usarse ese mismo algoritmo para calcular la intersección con una esfera con centro y radio arbitrarios (este problema puede reducirse al anterior si el rayo se traslada a un espacio de coordenadas donde la esfera tiene centro en el origen y radio unidad).

Problema 10.3.

Describe como podemos definir el campo escalar cuyos ceros son los puntos en un cilindro con altura unidad y radio unidad (sin considerar los discos que forman la base ni la tapa).

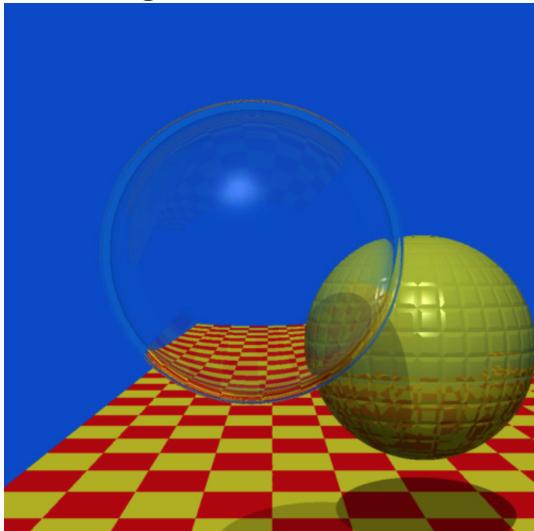
Usando esa definición diseña el algoritmo para calcular la intersección rayo-cilindro.

Describe asimismo el campo escalar y el algoritmo correspondientes a un cono de altura unidad y radio de la base unidad (sin considerar el disco de la base).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 85 de 94.

Ejemplos: primeras imágenes

Una de las primeras imágenes creadas con esta técnica (en 1980)



Turner Whitted (1980) An Improved Illumination Model for Shaded Display (CACM)
Entrevista a T. Whitted, vídeo sitio web de nVidia

Ejemplos: reflexión y refracción

Incluye: texturas, sombras arrojadas, reflexiones, *bump-mapping*.



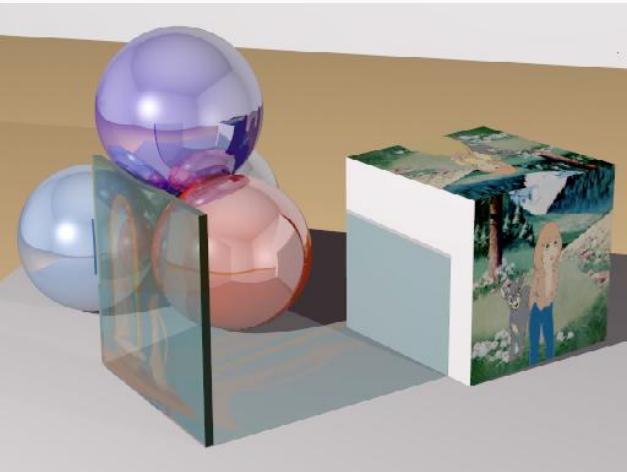
Universidad de Cornell (1998) Computer Graphics programme: Reflection and Transparency.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 87 de 94.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 88 de 94.

Ejemplos: reflexiones especulares, sombras arrojadas.

Incluye: reflexiones, texturas, sombras arrojadas con texturas.

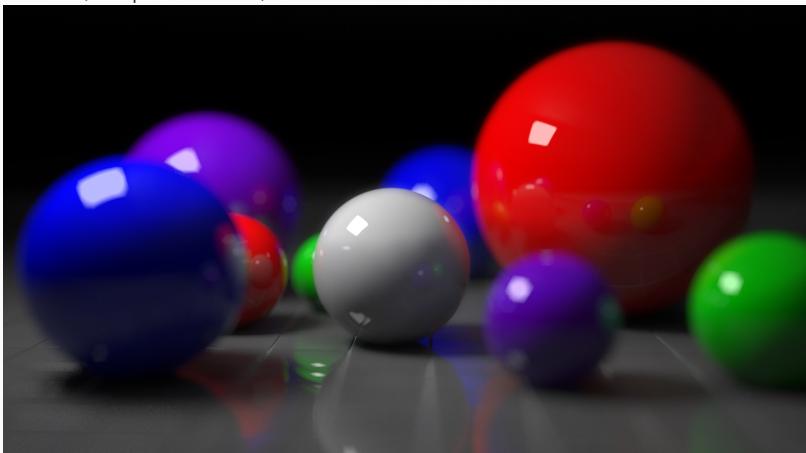


Universidad de Cornell (1998) [Computer Graphics programme: Reflection and Transparency](#).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 89 de 94.

Ejemplos: *Distributed Ray-Tracing*

Se usa *Distributed Ray-Tracing* (Cook, 1984): se simula la profundidad de campo (*Depth of Field*) y las penumbras generadas por luces de extensas (no puntuales):



By Mimigu at [English Wikipedia: Ray Tracing \(Graphics\)](#). CC BY 3.0.

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 91 de 94.

Ejemplos: escenas complejas, indexación espacial.

Escena compleja: múltiples objetos complejos. Indexación espacial.

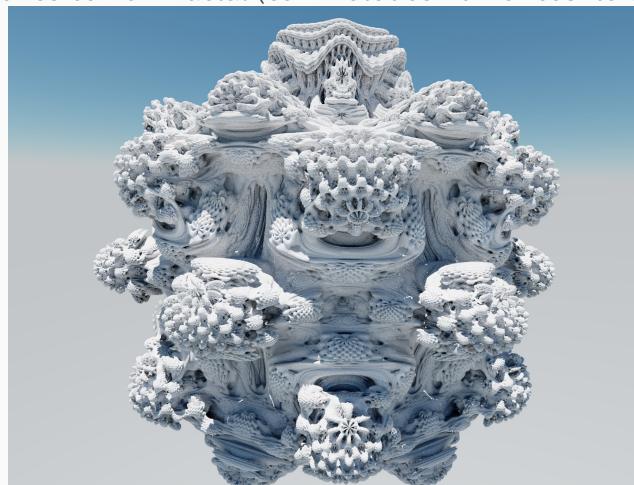


Autor: Gilles Tran [Sitio web Oyonale](#). (Public Domain)

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 90 de 94.

Ejemplos: *Path-tracing, fractales*.

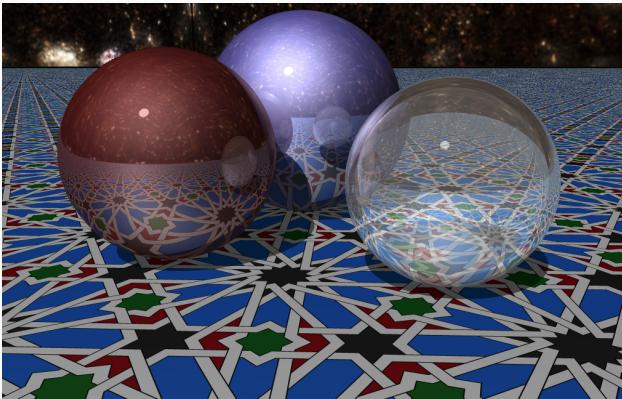
Se usa *Path-tracing* (Kajiya, 1986,*Global Illumination*), e intersecciones con un fractal (con métodos numéricos iterativos).



Por: Mikael Hvidtfeldt Christensen [Blog sobre arte generativo](#).
(<https://www.flickr.com/photos/syntopia/>).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 92 de 94.

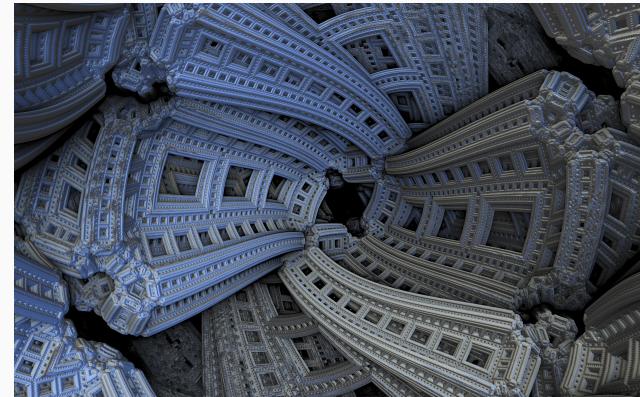
Ray-tracing en tiempo real en la GPU usando un *fragment shader*.
Incluye: reflexiones, refracciones, texturas procedurales (grupos cristalográficos del plano: pmm6)



Animación disponible on-line en Shadertoy: <https://www.shadertoy.com/view/4sdfzn>

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 93 de 94.

Se usan objetos definidos por su SDF (*Signed Distance Function*) para calcular intersecciones iterativamente. Se hace en la GPU en un *fragment shader*.



Menger Journey por Mikael Hvidtfeldt Christensen. Animación on-line en [Shadertoy](#).

GIM-GIADE: Informática Gráfica- curso 25-26- creado el 24 de noviembre de 2025 – transparencia 94 de 94.

Fin de la presentación.

Informática Gráfica.

Sesión 11: Animación.

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

Índice

La animación por ordenador. Ejemplos sencillos.	3
Técnicas de animación	49
Interpolación y curvas para animación	62
Animaciones en Godot	97

1. La animación por ordenador. Ejemplos sencillos..

Resumen de la sección

Sección 1.

La animación por ordenador. Ejemplos sencillos.

1. La animación clásica
2. Animación por ordenador
3. Estructura de los sistemas de animación
4. Implementación de sistemas de animación interactiva.
5. Ejemplos de animaciones sencillas.

En esta sección se realiza una introducción a la animación y la animación por ordenador

- Una breve introducción a la animación clásica: dispositivos mecánicos y cine de animación tradicional.
- Conceptos básicos de la animación por ordenador. Tipos de animación: *off-line* e *interactiva*

La Animación clásica

La palabra **Animación** (en sentido clásico) se refiere al proceso de darle apariencia de movimiento a objetos estáticos.

- Desde tiempos muy antiguos se ha intentado desarrollar dispositivos o mecanismos para lograr esto.
- La idea básica es presentar una secuencia de imágenes impresas (en papel o celuloide) a un ritmo que produzca *ilusión* de movimiento continuo.
- Los primeros dispositivos (diseñados a principios del siglo XIX) usan un efecto *estroboscópico*: el *Zootropo* (*Zoetrope*), y múltiples variantes:
- Con la aparición del cine, se crea la animación como una industria y un arte: se dibuja cada imagen y se fotografía sobre una película de celuloide.

Subsección 1.1.

La animación clásica

1. La animación por ordenador. Ejemplos sencillos..
1.1. La animación clásica.

Dispositivos mecánicos para animación

El efecto estroboscópico consiste en iluminar la imagen del sujeto en una posición fija del disco a intervalos regulares de tiempo, de forma que se incrementa la sensación de movimiento.

(no se percibe el movimiento del disco, sino la secuencia de imágenes)



GIF animado de: *Simon Ritter von Stampfer*, Public domain, via Wikimedia Commons:
commons.wikimedia.org/wiki/File:Prof.Stampfer%27s_Stroboscopische_Scheibe_No.X.gif

Sesión 11: Animación

Created 2025-12-09

Page 6 / 112.

Cine de animación clásico

A lo largo del siglo XX se desarrollan técnicas de animación clásica (no por ordenador) y *stop motion*

- Se comienza dibujando cada cuadro a mano de forma independiente, se hace una fotografía de cada cuadro (se crea un fotograma de la película).
- En *stop motion* se crea un escenario real estático que se cambia incrementalmente para cada fotografía.
- Despues se desarrolla el *keyframing / inbetweening*: unos artistas dibujan unos cuadros clave y otros interpolan los cuadros intermedios.
- Al principio cada cuadro era una imagen era única.
- Despues se forma cada cuadro superponiendo varias *capas* de dibujo, cada una de ellas dibujada en un soporte transparente.

Cine de animación clásico: keyframes e in-betweens

Ejemplo de tres keyframes y los in-betweens:

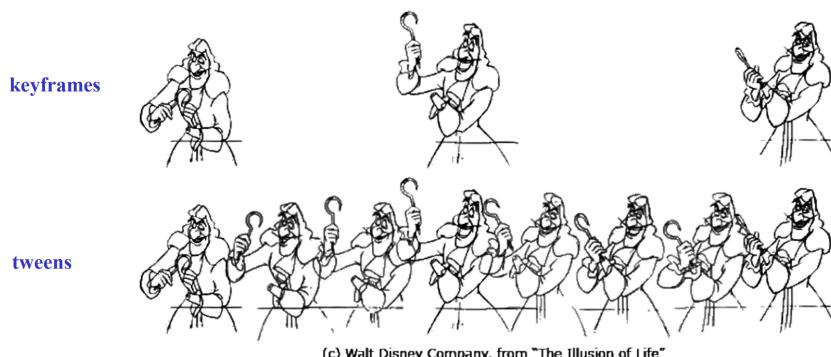


Imagen del libro de Frank Thomas y Ollie Johnston: **The Illusion of Life** (1981):
<https://books.disney.com/book/the-illusion-of-life/>

Sesión 11: Animación

Created 2025-12-09

Page 9 / 112.

1. La animación por ordenador. Ejemplos sencillos..
1.1. La animación clásica.

Cine de animación clásico: cámaras multiplano

Este tipo de cámaras tienen varios grados de libertad que permitían mover las capas de dibujo de forma independiente:

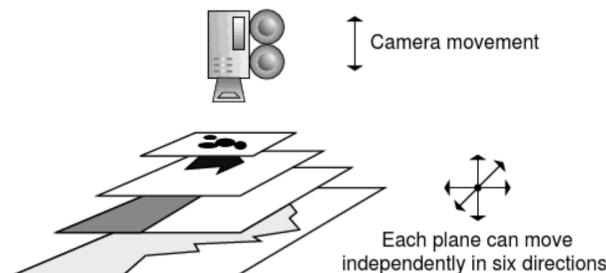


Imagen del libro: Rick Parent **Computer Animation Algorithms and Techniques** (3rd Edition)
Ed. Elsevier. 2012. ISBN: 9780124158429.

Cine de animación clásico: cámaras multiplano

Los cuadros de una animación pueden estar compuestos de varias capas, para eso se diseñaron las **cámaras multiplano**:

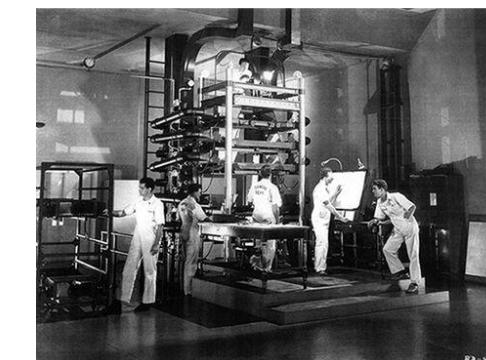


Imagen de The Walt Disney Family Museum Blog:
waltdisney.org/blog/machine-imagination-walt-disneys-pinocchio-and-multiplane-camera

Sesión 11: Animación

Created 2025-12-09

Page 10 / 112.

1. La animación por ordenador. Ejemplos sencillos..
1.1. La animación clásica.

Sesión 11: Animación

Created 2025-12-09

Page 10 / 112.

Subsección 1.2. Animación por ordenador

Animación por ordenador

A partir de los años 70, se comienza a dar soporte para la animación usando ordenadores, de forma progresivamente mayor hasta la actualidad, donde los ordenadores son una herramienta esencial para las animaciones. El ordenador se usa para:

- Modelado de personajes, escenarios y elementos 2D o 3D.
- Iluminación y texturas realistas.
- Definición de la dinámica y comportamiento de los objetos de la escena.
- Render de los fotogramas.
- Mezcla de imagen real fotografiada o filmada con elementos generados por ordenador.
- Composición de vídeo y efectos post-render.

Eventos. Tipos.

En el contexto de la animación, un **evento** es un cambio de estado relevante, de varios tipos:

Eventos de entrada: como los que ya hemos estudiado, son eventos producidos directa o indirectamente por el usuario usando algún dispositivo, de uno de estos dos tipos:

- **Físico:** teclado, joystick, ratón, etc...
- **Virtual:** un botón u otros elementos de un interfaz de usuario. También los hemos llamado **señales**

Eventos de la escena: eventos que resultan de la interacción de los elementos de la escena entre ellos, p.ej.:

- La trayectoria de un objeto alcanza una pared.
- Un coche supera la línea de meta.
- Un NPC (*personaje no jugable*) ha entrado en una habitación.
- El personaje del jugador ha perdido toda su energía.

Animación por ordenador: terminología

Usaremos los siguientes términos:

- **Cuadro o fotograma** (o **frame**): cada una de las imágenes estáticas que componen una animación.
- **Fotogramas por segundo** (o **frame rate, FPS**): número de fotogramas por unidad de tiempo real (usualmente segundos) a la que se reproduce la animación, puede ser constante o variar. A veces es un objetivo que no siempre se consigue en todos los *frames*.
- **Período (T)**: tiempo (usualmente en milisegundos) entre fotogramas consecutivos (es 1000/FPS).
- **Período objetivo**: período que se quiere lograr en una animación.
- **Tiempo**: usamos este término para referirnos, durante la reproducción de una animación, al **tiempo real** transcurrido desde su inicio o desde un instante de referencia en la misma.

Animación por ordenador no interactiva (*off line*)

En la animación *off line* usada en cinematografía no hay interacción con el usuario durante el proceso de render de las imágenes.

- Se planean con antelación las animaciones de cada elemento de cada secuencia de animación.
- Se decide de antemano un *framerate* fijo (normalmente entre 24 y 60 FPS).
- Cada fotograma tiene asociado un instante de tiempo fijo y conocido.
- Cada fotograma producido es almacenado en el sistema de archivos.
- El tiempo que se tarda en sintetizar cada cuadro no está relacionado con el *frame rate* (puede ser muy superior al período).

Animación por ordenador interactiva (*on line*)

En la animación *interactiva* (típicamente usada en videojuegos y simuladores), el usuario interactúa con el sistema durante la reproducción de la animación, influyendo en la misma:

- El usuario produce *eventos de entrada* (teclado, ratón, joystick, etc..)
- Se define como cada elemento reacciona a los posibles eventos y como evoluciona con el tiempo.
- Cada fotograma producido es mostrado en un monitor.
- Se elige un *frame rate* mínimo (un período máximo) como objetivo. A veces puede no lograrse.
- Por tanto: el tiempo que se tarda en sintetizar cada cuadro está, en principio, acotado por el período objetivo (en un sistema monohebra).

Subsección 1.3. Estructura de los sistemas de animación

El bucle de animación

La producción de una animación *off-line* o de una animación interactiva supone la ejecución de un **bucle de animación**

- En animaciones interactivas, se ejecuta hasta que se decide parar la animación o acabar el programa.
- En animaciones *off line* se ejecuta hasta que se han visualizado todos los *frames* que se planeaban producir.
- En cada iteración:
 - ▶ Se actualiza el estado del modelo de escena (ya que cambia con el tiempo).
 - ▶ Se produce una imagen.
- La actualización del estado puede suponer cálculos complejos: iluminación avanzada, física, IA de personajes, post-procesamiento de imagen, etc...)

Modelo de escena en animación

El modelo de escena debe incluir (igual que en IG en general):

- **Modelos geométricos:** mallas de triángulos, árbol de escena, transformaciones, etc...
- **Modelos de aspecto:** texturas, materiales, luces, etc...

Además, en animación se debe añadir a dicho modelo:

- **Modelos dinámicos:** cómo evolucionan los modelos geométricos o de aspecto de los objetos de la escena con el tiempo, p.ej. qué trayectoria seguirá un objeto.
- **Modelos de comportamiento:** para animación interactiva, hay que definir cómo reaccionará cada elemento a los posibles eventos que pueden ocurrir, p.ej.: qué hará el personaje jugador cuando el usuario pulsa una tecla determinada, o cuando atraviesa una puerta.

Estado de los objetos

En animación interactiva cada objeto u elemento visible tiene un estado que puede depender en todo o en parte del tiempo. El estado puede tener distintas componentes:

- **Estáticas:** independiente del tiempo (constantes) (p.ej: la geometría en coordenadas maestras y la textura de una bala de cañón).
- **Dinámicas y continuas:** variables continuas cuyo valor es una función no constante del tiempo (p.ej: la posición, velocidad, aceleración y orientación de la bala, las cuales son funciones del tiempo transcurrido desde que se disparó).
- **Dinámicas y discretas:** variables que toman un valor en un conjunto discreto de valores (p.ej: la bala puede estar en estado de reposo, de movimiento o explotando). Los cambios del valor se llaman **transiciones** y se suelen modelar con **máquinas de estados**

Cambio y evolución del estado en los objetos

Cada objeto dinámico de una escena animada debe de incorporar métodos para:

- **Actualizar el estado:** calcular el nuevo estado del objeto para un frame que se va a sintetizar, a partir del estado actual y del instante de tiempo t de dicho frame.
- **Procesar evento** ejecuta una posible transición tras un evento, es decir: calcula el nuevo estado del objeto a partir de un evento de entrada que ha ocurrido en un instante de tiempo. Se conocen:
 - ▶ El tipo y atributos del evento (p.ej: la posición del ratón, etc...)
 - ▶ A veces, el instante de tiempo real en el que se produjo el evento.

El bucle de animación interactiva

El esquema en pseudo-código del bucle de animación interactiva es:

```
1: while se esté ejecutando la animación do
2:   t ← instante de tiempo real actual
3:   for each evento  $E$  pendiente de procesar do
4:     for each objeto  $O$  afectado por  $E$  do
5:       Hacer que  $O$  procese el evento  $E$ .
6:     end
7:   end
8:   for each objeto dinámico  $O$  de la escena do
9:     Actualizar el estado de  $O$  al instante  $t$ .
10:  end
11:  Visualizar la escena sobre el framebuffer actual.
12:  Presentar el framebuffer actual en pantalla.
13:  Esperar bloqueado hasta  $t + T$            ▷ (si sobra tiempo)
14: end
```

Sistemas de animación y tiempo real

Los sistemas de animación tienen algunas características de *sistemas de tiempo real*:

- Un *sistema de tiempo real (STR)* es un software que tiene requerimientos no funcionales estrictos en cuanto a los tiempos de respuestas de sus componentes
- Por ejemplo: el sistema que detecta una colisión y despliega un airbag de un coche debe hacer eso en un tiempo limitado, del orden de milisegundos, es un **STR estricto**
- En animación interactiva, el *frame rate* es un objetivo que se intenta alcanzar, pero no siempre se consigue. Por eso se puede considerar un sistema de tiempo real **no estricto**.

Herramientas de tiempo real

Se requieren herramientas software (librerías o soporte del lenguaje) de tiempo real para

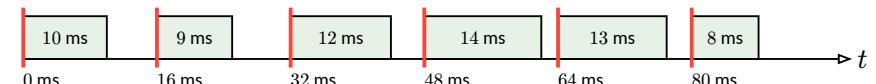
- Ser capaz de medir el tiempo transcurrido con precisión, para
 - ▶ conocer el instante de tiempo real de cada fotograma que se va a producir, y para
 - ▶ conocer el tiempo sobrante o de retraso hasta el siguiente fotograma.
- Ser capaz de esperar hasta el fin del período entre dos frames, se puede hacer dos formas alternativas:
 - ▶ hacer una espera bloqueada explícita (en aplicaciones de escritorio), o bien
 - ▶ requerir que se ejecute una función pasado un determinado período de tiempo (en aplicaciones Web).

Todos estos aspectos suelen ser transparentes al programador en **engines de videojuegos** como Godot, Unity, Unreal Engine, etc...

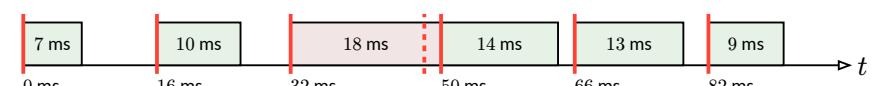
Diagrama de una animación

Suponiendo un período $T = 16$ milisegundos (ms), cada caja representa la actualización de estado y el render de un frame (tarda un tiempo posiblemente distinto en cada frame)

Todos los frames se pueden calcular dentro del período:

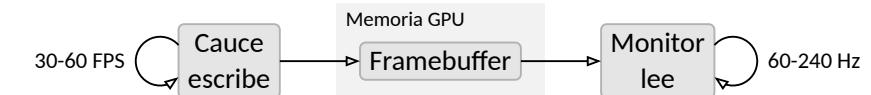


Un frame (en rojo) tarda más del período, el siguiente se retrasa:



La tasa de refresco del monitor

En un esquema sencillo, el cauce escribe en un *framebuffer* (memoria de imagen), el cual a su vez está conectado al ordenador:



- El cauce escribe periódicamente en el *framebuffer* (30-60 FPS).
- El monitor está continuamente leyendo la imagen del *framebuffer* y presentándola en pantalla
- La **tasa de refresco del monitor (monitor refresh rate)** es el número de veces por segundo que eso ocurre (entre 60 y 240 Hz)
- No tiene sentido usar para una animación interactiva un *frame rate* superior a la tasa de refresco del monitor (algunos frames producidos no se verán en pantalla).

Parpadeo debido a framebuffer único

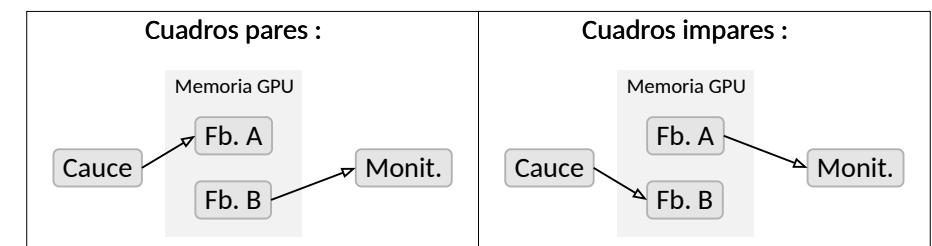
Si se usa un único framebuffer (como en el ejemplo de arriba)

- En una animación, el cauce no está sincronizado con el refresco del monitor (el *framerate* de la animación es distinto del *refresh rate* del monitor).
- Por tanto, ocurrirá que el refresco del monitor se realiza cuando el cauce está a mitad de visualizar una escena en el *framebuffer*.
- La imagen que se presenta en el monitor será incompleta (solo se visualizan algunos triángulos de la escena).
- Al efecto visible se le llama en general **flickering (parpadeo)**

Framebuffer doble para animación interactiva

Para evitar problemas se usa un *doble buffer*: consiste en usar al menos dos *framebuffers* en lugar de uno solo:

- Uno de ellos se muestra en el monitor mientras el otro se actualiza.
- Cada vez que se termina de renderizar un frame, los roles de ambos buffers se intercambian (se hace *swap* de los buffers).
- Los frames mostrados en el monitor están siempre completos.



Subsección 1.5.

Ejemplos de animaciones sencillas.

Animación de las agujas de un reloj

Supongamos que queremos visualizar una animación de un reloj con tres agujas: una para las horas, otra para los minutos y otra para los segundos.

- Tenemos una malla de polígonos que es un modelo de una aguja en posición vertical (paralelo al eje Y, hacia arriba), con el origen en el punto del eje del reloj.
- Para visualizar las tres agujas usamos matrices de rotación entorno el eje Z, cada una es una rotación distinta.
- Cada una de esas matrices de rotación depende del tiempo t , que es el tiempo en segundos transcurrido desde el comienzo del un día determinado.
- Lógicamente, cada una de esas matrices usa un ángulo que depende de t , pero con una constante de proporcionalidad distinta.

La animación de tres agujas:

Llamamos θ_h , θ_m y θ_s a los ángulos de las tres agujas en un instante t (es un tiempo conocido en segundos)

- Las matrices de rotación serán $R_z(\theta_h)$, $R_z(\theta_m)$ y $R_z(\theta_s)$, donde, en general, para un ángulo γ cualquiera (en radianes) la expresión $R_z(\gamma)$ representa la matriz de rotación entorno al eje Z de un ángulo γ (en el sentido de las agujas del reloj si $\gamma > 0$ y lo vemos desde Z+). Por tanto:

$$R_z(\gamma) = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{donde: } \begin{cases} c = \cos(\gamma) \\ s = \sin(\gamma) \end{cases}$$

Cálculo de los ángulos

Los tres ángulos θ_h , θ_m y θ_s dependen linealmente de t (en segundos):

- El segundero da una vuelta completa (2π radianes) cada minuto (60 segundos):

$$\theta_s = \frac{2\pi}{60} \cdot t$$

- El minutero da una vuelta cada hora ($= 60^2$ segundos):

$$\theta_m = \frac{2\pi}{60^2} \cdot t$$

- Las horas dan una vuelta cada 12 horas ($= 12 \cdot 60^2$ segundos):

$$\theta_h = \frac{2\pi}{12 \cdot 60^2} \cdot t$$

Modelo del reloj

Por tanto, modelamos el reloj usando un grafo de escena en el cual el nodo raíz es un objeto compuesto que contiene:

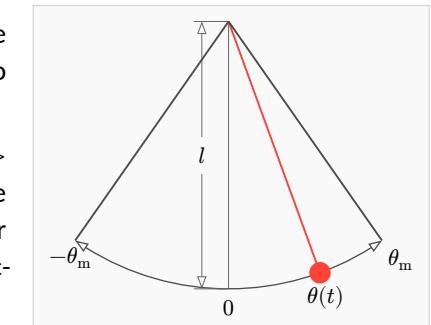
- un nodo para la esfera (que no cambia con el tiempo), con centro en el origen.
- un nodo para cada aguja, cada uno de ellos contiene
 - la matriz de transformación de la aguja.
 - un nodo hijo (el modelo de la aguja en coordenadas maestras).

La función para actualizar el estado simplemente asigna las matrices de rotación según el valor de t recibido, usando las fórmulas que hemos visto antes.

Modelo del péndulo

Supongamos el modelo de un péndulo: es un disco con cierta masa colgando de un punto fijo por una cuerda de longitud l . El ángulo θ entre la cuerda y la vertical varía con el tiempo t , es una función $\theta(t)$ del tiempo t (en segundos).

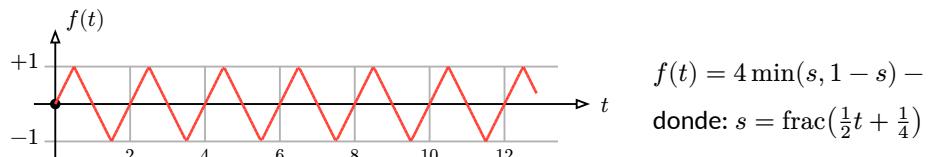
- El péndulo oscila periódicamente desde un ángulo $-\theta_m$ hasta el ángulo θ_m
- Esa oscilación tiene un período $T > 0$ expresado en segundos, es decir: se tardan T segundos en volver a pasar por el mismo sitio con la misma dirección.



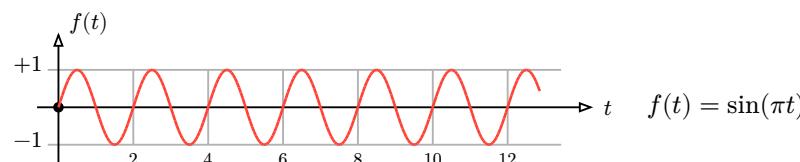
Función oscilante auxiliar

Necesitamos una función $f(t)$ que oscile entre -1 y $+1$ con un período de 2 unidades de tiempo.

Una posibilidad es una función **lineal a trozos** (cambios bruscos)

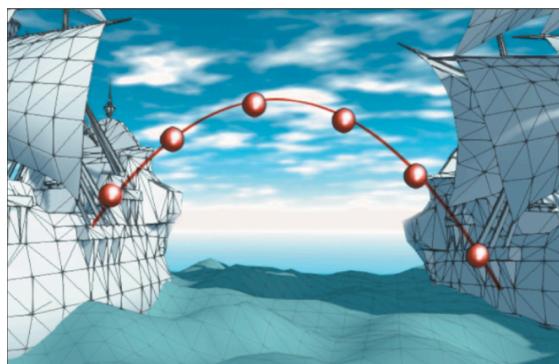


Para lograr un movimiento más suave, usamos una **sinusoidal**



Ejemplo de la bala de cañón

Supongamos que queremos simular la trayectoria de una bala de cañón esférica que es lanzada desde un punto (se eleva hasta que vuelve a la altura inicial)



Evolución del ángulo θ con el tiempo.

Así que ya podemos usar la función f para modelar como evoluciona el ángulo θ con el tiempo:

$$\theta(t) = \theta_m \cdot f(2t/T)$$

Donde:

- La amplitud de la oscilación es $2 \cdot \theta_{\max}$ en lugar de 2
- El período es T en lugar de 2.

El modelo del péndulo será un modelo jerárquico que incluye una matriz de rotación entorno al eje Z, $R_z(\theta)$, con ángulo θ (respecto de la vertical).

Supuestos

Suponemos las siguientes condiciones:

1. La bola sale del cañón en una posición inicial y con una velocidad inicial conocidas.
2. La bola está sujeta a la gravedad.
3. La bola es esférica y de un color plano (la orientación o rotación es irrelevante)
4. La bola no tiene efectos de fricción con el aire
5. Nos interesa simular la trayectoria hasta que alcanza de nuevo la altura inicial.

En estas condiciones, la trayectoria de la bala **solo depende de la velocidad y dirección iniciales**.

Trayectoria con aceleración constante

Las condiciones que determinan la trayectoria son:

- La bala tiene (en el instante inicial $t = 0$) su centro en una posición $\mathbf{p}(0) = (p_x, p_y, p_z)$ y una velocidad inicial $\mathbf{v}_0 = (v_x, v_y, v_z)$, ambos son dos vectores en 3D, conocidos, con unidades de metros (m) y metros por segundo (m/s), respectivamente. Se cumple $v_y > 0$ (se lanza hacia arriba).
- La aceleración (debida a la gravedad) es un vector $\mathbf{a} = (0, -g, 0)$ constante en el tiempo, donde $g = 9.8$ (en m/s²).

En estas condiciones, la posición (en metros) de la bala en un instante cualquiera t es $\mathbf{p}(t)$, definido como:

$$\mathbf{p}(t) = \mathbf{p}(0) + \mathbf{v}_0 t + \frac{1}{2} \mathbf{a} t^2$$

Implementación de la trayectoria

La función $\mathbf{p}(t)$ es un ejemplo de una **curva paramétrica** en el espacio 3D (una secuencia continua de puntos que depende de un valor real, en este caso el tiempo).

Para visualizar la animación se usa:

- Un modelo en coordenadas maestras de la bala,
- Una matriz de traslación $T_{\mathbf{p}(t)}$, que traslada el centro del modelo a la posición $\mathbf{p}(t)$ cada vez que se pide una actualización del modelo para el instante t .

La duración de la animación en segundos es

$$t = \frac{2v_y}{g}$$

(es el tiempo que tarda la bala en volver a la altura inicial p_y).

Ejemplo de un perseguidor

Las anteriores animaciones se basan en modificar un ángulo o una posición usando una expresión para una matriz que depende del tiempo según una función simple conocida.

- No se requieren variables de estado (adicionales a la posición, el ángulo y algunas constantes conocidas).
- En muchas animaciones más complejas, se requieren variables de estado (discretas o continuas) para modelar el comportamiento.

Un ejemplo sería un modelo de un **perseguidor**

- Es un objeto que está en una posición pero se desplaza hacia una posición objetivo.
- Puede sufrir cambios de estado en respuesta a eventos externos: pausar, reanudar, fijar un nuevo objetivo.

Condiciones del comportamiento

El comportamiento del perseguidor es este:

- Cuando el perseguidor está pausado, no realiza ningún movimiento.
- Cuando el perseguidor no está pausado, se desplaza en línea recta hacia el objetivo. Lo hará a velocidad constante (1m/s), hasta que su posición se encuentre sobre el objetivo (o se fije otro objetivo).
- Cuando el perseguidor no está pausado pero su posición ya coincide con la del objetivo, no se mueve.
- Los eventos externos producen cambios de estado, cada uno de esos eventos tiene asociado un instante en el tiempo. Se implementan como funciones que cambian el estado del perseguidor, las cuales reciben un valor de tiempo (nunca inferior al de la llamada previa).

Variables de estado del perseguidor

Las diversas variables de estado que se necesitan para implementar su comportamiento son:

Estado: una variable discreta que solo puede tomar dos valores: *persiguiendo*, y *pausado*. Inicialmente es *pausado*.

Posición: (*p*) posición actual en el espacio del objeto (un vector). Inicialmente está en el origen.

Objetivo: (*o*) posición del objetivo en el espacio (un vector). Inicialmente es el origen.

Último instante: (*u*) instante en el que se actualizó por última vez la posición del objeto, por tanto es el instante al que corresponde la posición. Inicialmente es 0.

Eventos que producen cambios de las variables

Los eventos u operaciones que producen cambios de alguna de las variables de estado tienen todas asociado un instante *t*, son:

Actualizar estado: se llama (al menos) antes de visualizar un nuevo *frame*, para llevar la posición a la correspondiente al instante *t*

Fijar objetivo: supone cambiar la variable *q* por un nuevo valor y pasar al estado *persiguiendo* (si no estaba ya en ese estado).

Pausar: si el estado no es *pausado*, cambia el estado a *pausado*, a partir de ese instante el perseguidor no cambia su posición (pero mantiene el objetivo que tenía)

Reanudar: si el estado no es *persiguiendo*, cambia el estado a *persiguiendo*, a partir de ese instante su posición puede cambiar con el tiempo.

Actualización de estado

La función de actualización recibe un instante *t* y actualiza la posición *p*:

```
1: function ACTUALIZARESTADO(t)
2:   if estado == persiguiendo then
3:     d ← o - p                         ▷ d es el vector hasta el objetivo
4:     if \|d\| > 0 then
5:       r ← min(t - u, \|d\|)             ▷ r es el tiempo en mov.
6:       p ← p + rd/\|d\|                ▷ actualizar posición
7:     end
8:   end
9:   u ← t
10: end
```

Nota: el valor *r* es el tiempo en movimiento (desde la última actualización previa) hasta llegar al objetivo o hasta esta actualización (lo que ocurra antes). Se asume velocidad unidad.

Implementación de eventos

La implementación de las operaciones es:

```
1: function FIJAROBJETIVO(t, q)
2:   ACTUALIZARESTADO(t)
3:   o ← q
4: end
```

```
1: function PAUSAR(t)
2:   ACTUALIZARESTADO(t)
3:   estado ← pausado
4: end
```

```
1: function REANUDAR(t)
2:   ACTUALIZARESTADO(t)
3:   estado ← persiguiendo
4: end
```

Resumen de la sección

En esta sección se abordan diversas técnicas de animación muy usadas en la producción de infografías y desarrollo de videojuegos:

Keyframes e Inbetweens: (cuadros clave y cuadros intermedios)

Esta técnica agiliza el diseño e implementación de las animaciones, ya que reduce la complejidad de estos procesos (reduce el número de cuadros que se diseñan manualmente)

Rigging y Skinning: (esqueleto y piel)

También facilita el diseño de objetos complejos (como personas o animales), ya que evita el tener que adaptar manualmente el modelo de la piel o ropa a las distintas poses de un modelo articulado.

Sección 2.

Técnicas de animación

1. Keyframes e Inbetweens
2. Rigging y Skinning

Sesión 11: Animación

Created 2025-12-09

Page 50 / 112.

2. Técnicas de animación.

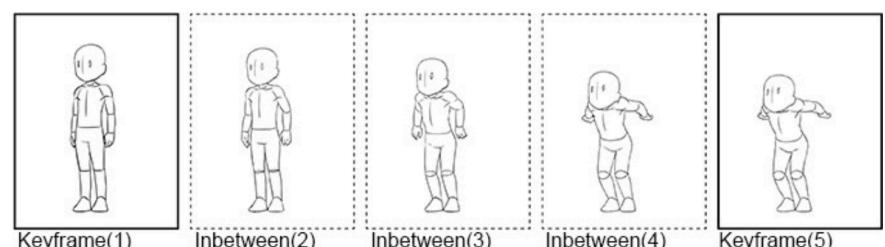
2.1. Keyframes e Inbetweens.

Keyframes e Inbetweens

La técnica de *keyframing* permite una mayor agilidad en la producción de animaciones, ya que separa la tarea de definir unos **cuadros clave (keyframes)** de la tarea de producir los frames **intermedios** entre ellos (**inbetweens**)

Subsección 2.1.

Keyframes e Inbetweens



Cuadros clave para animación de grafos de escena

Para animaciones por ordenador de grafos de escena parametrizados, se puede usar la metodología de los **cuadros clave**:

- Cada cuadro clave tiene asociada un valor de tiempo (t_j)
- El diseñador o animador especifica el vector de valores de los parámetros en cada cuadros clave. Por tanto, conocemos el valor $v_{i,j}$ del parámetro i en el cuadro clave número j
- Para los cuadros intermedios (entre cuadros clave) se hace **interpolación**: cada parámetro se interpola entre el valor de ese parámetro en el cuadro clave anterior y el posterior.

Edición de las curvas de interpolación

Aquí vemos una captura del *editor de curvas* de 3Ds Max (Autodesk). El usuario está editando 6 curvas correspondientes a 6 parámetros:

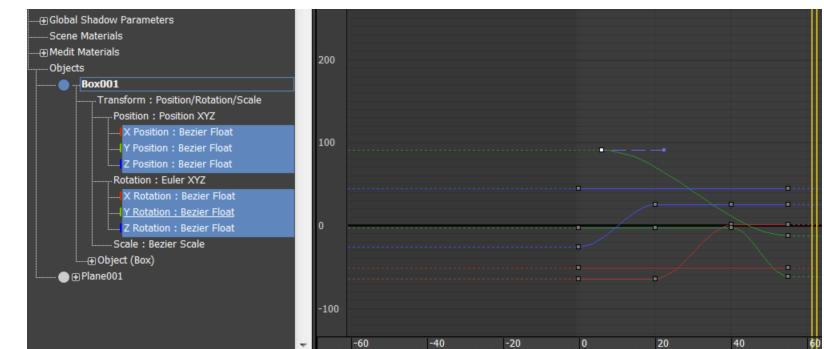


Figura obtenida de la página [Curve Editor Introduction](#) en el sitio Web [Autodesk 3Ds Max Learning Center](#) (Abril 2024).

Edición de las curvas de interpolación

También es posible: (a) controlar las derivadas, y (b) que los instantes clave t_i de cada parámetro sean distintos:

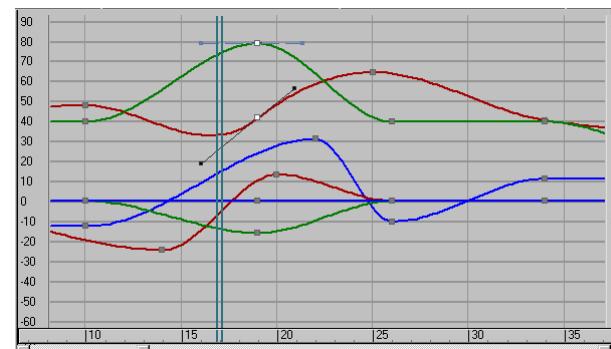


Figura obtenida de la página [Track View Workspace](#) en el sitio Web [Autodesk 3Ds Max Learning Center](#) (Abril 2024).

Subsección 2.2.
Rigging y Skinning

Modelado de personajes

Si queremos modelar mallas de triángulos que representen personajes o animales:

- Se puede usar un grafo de escena jerárquico parametrizado, hecho de partes rígidas: cabeza, torso, muslo, pierna, pie, antebrazo, brazo, manos, los dedos, falanges, etc...
- Cada elemento es un nodo del grafo que se puede rotar respecto a la articulación que lo une al padre. El vector de parámetros son los ángulos de esas articulaciones (permite definir poses e interpolar entre ellas).
- El problema es modelar de forma realista la piel o la ropa, que se extienden de forma continua entre distintas partes y se *adaptan* a las mismas: es poco realista modelarlo como una malla rígida.

Rigging y Skinning: Rigging

La solución consiste en separar el problema en dos partes, llamadas *Rigging* y *Skinning*

Rigging: se define un grafo de escena parametrizado (*esqueleto*)

- Está hecho de segmentos de recta rígidos (*huesos*), con extremos en las articulaciones
- El modelo tiene asociados n parámetros (típicamente ángulos de rotación, u otros)
- Cada hueso tiene una matriz de transformación asociada.
- Este esqueleto se usa para definir las poses del modelo.

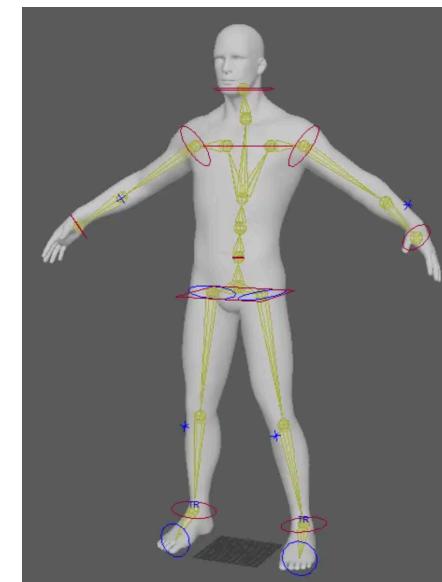
Rigging y Skinning: Skinning

Una vez está definido el *rig* (esqueleto), se procede al *skinning*

Skinning: se define una malla indexada de triángulos (*skin*, piel) que se adapta al modelo en una pose *neutra*.

- Cada vértice de la malla se asocia a uno o varios de los huesos (usando un vector de pesos), según su distancia a los mismos.
- Para una pose arbitraria (distinta de la neutra), la matriz de transformación de cada vértice de la piel es una suma ponderada de las matrices de transformación de los huesos asociados a dicho vértice.

Ejemplo de un esqueleto (*Rig*) y su piel (*Skin*)

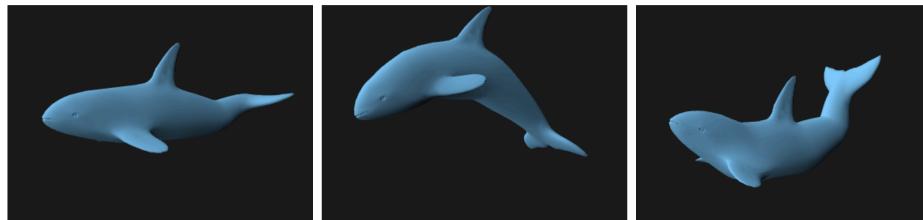


Se observa el modelo jerárquico (segmentos) del esqueleto, así como la malla de triángulos que representa la piel del modelo. En la página se observa una animación interactiva del modelo.

Figura obtenida de la página
[Rigging and Skinning](https://gamedevinsider.com/) en el sitio Web
<https://gamedevinsider.com/> (Abril 2024).

Implementación en un vertex shader

Aquí podemos ver una implementación sencilla del *rigging* basado en vectores de pesos de cada vértice, en un *vertex shader* con WebGL, para un modelo simple:



Página con descripción de la técnica y la animación en **WebGL Fundamentals**:

<https://webglfundamentals.org/webgl/lessons/webgl-skinnning.html>

Enlace directo a la animación:

<https://webglfundamentals.org/webgl/webgl-skinnning-3d-gltf-skinned.html>

Sección 3.

Interpolación y curvas para animación

1. Interpolación de valores reales.
2. Curvas paramétricas.
3. Curvas e interpolación en 2D/3D.

Resumen de la sección

En esta sección se exponen las metodologías para implementar funciones reales de variable real que interpolan entre dos o más valores reales, dados como datos de entrada, pudiéndose controlar la forma de la función fijando sus derivadas

Esta técnica es útil para el diseño de animaciones usando funciones o curvas paramétricas, que permiten, entre otras cosas, realizar interpolación de variables continuas (especialmente para *in betweening*) y diseñar trayectorias de movimiento, todo ello de forma flexible y eficiente.

Subsección 3.1.

Interpolación de valores reales.

El problema de la interpolación

Queremos reproducir una animación entre dos instante de tiempo, de forma que a lo largo del intervalo entre ambos, una variable v tome un valor dado al inicio de la animación y otro distinto al final. La variable puede ser de varios tipos:

Valor real: por ejemplo: una posición en una línea, un ángulo, etc...

Vector de reales: por ejemplo: una posición en 2D o en 3D, o una velocidad, etc...

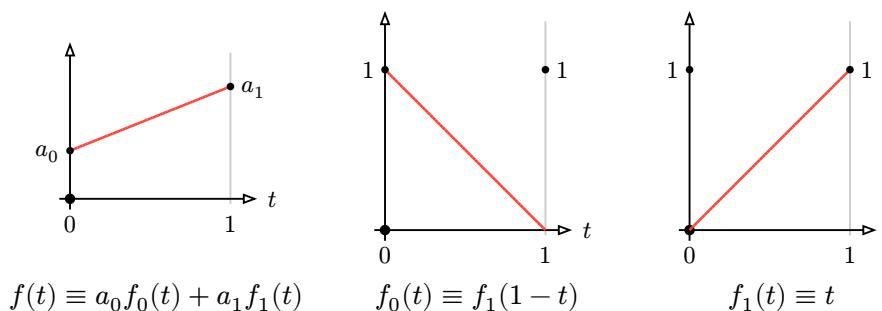
Matriz de reales: por ejemplo: una transformación geométrica, una matriz de cámara, etc...

Por simplicidad, inicialmente consideramos que:

el intervalo de tiempo va de 0 a 1 y que la variable es un valor real

Función lineal.

Una posibilidad muy sencilla es usar una función f **lineal**:



El problema es que la pendiente es constante, pero nos gustaría usar una curva cuya pendiente vaya tiendiendo a cero en los extremos (para una entrada y salida suaves). Por tanto queremos que $f'(0) = f'(1) = 1$.

Interpolación en el intervalo unidad

Por tanto, tenemos el siguiente problema de interpolación:

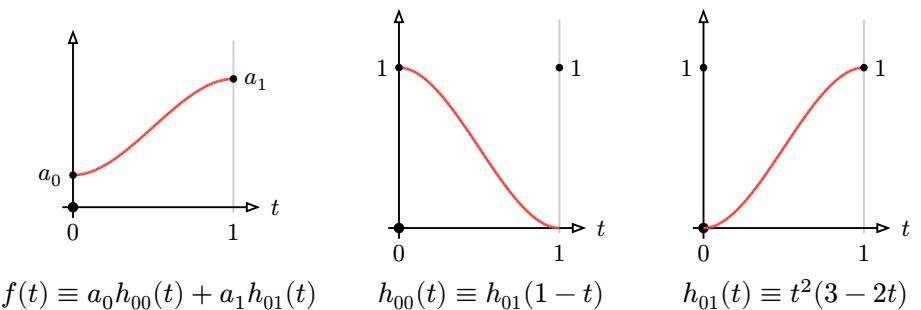
- El instante inicial es $t = 0$ y el final es $t = 1$.
- El valor inicial de la variable v es a_0 y el final es a_1 , ambos son datos conocidos de entrada (pueden ser positivos o negativos).
- Queremos encontrar una función f tal que:
 - ▶ f tiene un parámetro real t , siempre en el intervalo $[0, 1]$
 - ▶ f produce valores reales (positivos o negativos).
 - ▶ Se cumple $f(0) = a_0$ y $f(1) = a_1$

Cualquier función que usemos para interpolar entre a_0 ya a_1 se puede poner como la suma ponderada de dos funciones f_0 y f_1 :

$$f(t) \equiv a_0 f_0(t) + a_1 f_1(t) \quad \text{donde: } \begin{cases} f_1(0) = 0 \\ f_1(1) = 1 \\ f_0(t) = f_1(1-t) \end{cases}$$

Curvas cuadráticas

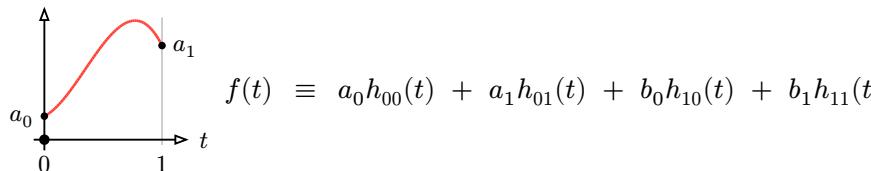
Para lograr una curva con pendiente nula en los extremos se puede usar el polinomio de grado 2 que cumple las 4 condiciones que le hemos impuesto:



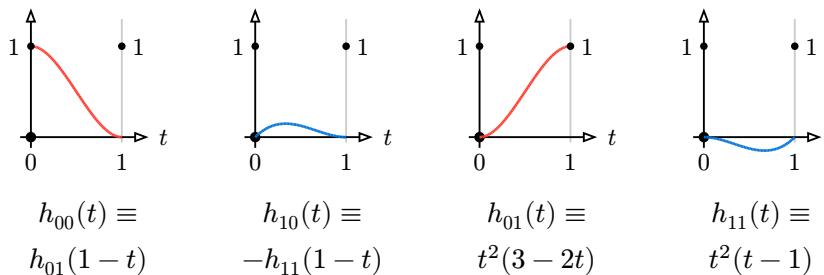
Sin embargo, en muchas aplicaciones queremos controlar los valores de las derivadas de f en 0 y 1 (hacerlos iguales a cualquier valor, distinto de 0).

Control de derivadas con splines cúbicos de Hermite

Ahora f es un polinomio de grado 3 conocido como el *Spline cúbico de Hermite*:



Aquí b_0 y b_1 son las derivadas de f en 0 y en 1. Las funciones base son:

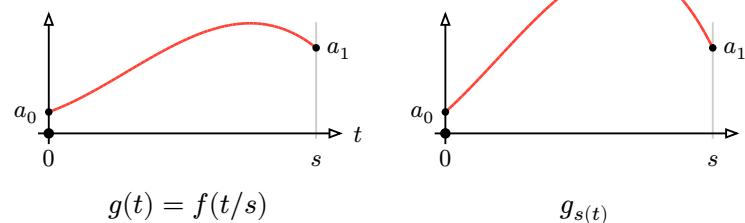


Splines de Hermite en intervalos arbitrarios

Para fijar las derivadas b_0 y b_1 en un intervalo de longitud $s > 0$, usaríamos una versión modificada (g_s) del Spline cúbico de Hermite:

$$g_s(t) \equiv a_0 h_{00}(t/s) + a_1 h_{01}(t/s) + sb_0 h_{10}(t/s) + sb_1 h_{11}(t/s)$$

donde se multiplican los coeficientes de las derivadas por s . Aquí vemos el resultado:



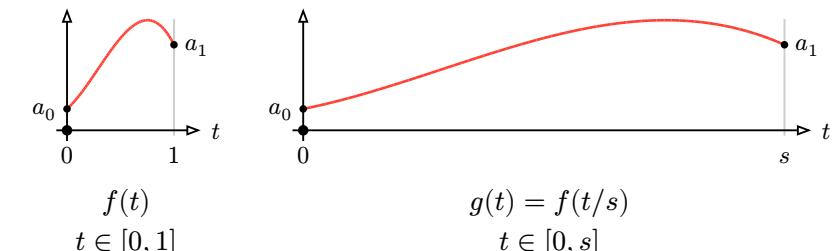
Interpolación en intervalos de longitud arbitraria

Supongamos ahora que queremos interpolar en un intervalo $[0, s]$ con una longitud $s > 0$ no necesariamente igual 1.

- Si tenemos una función f que interpola en $[0, 1]$, podemos construir otra función g que interpola en $[0, s]$, sin más que hacer

$$g(t) \equiv f(t/s) \quad \text{donde: } t \in [0, s]$$

- Las derivadas de la función g , sin embargo, se dividen por s



En muchas aplicaciones, queremos que una función f tome una serie de valores arbitrarios a_0, a_1, \dots, a_{n-1} en una secuencia ordenada de puntos de tiempo conocidos $t_0 < t_1 < \dots < t_{n-1}$, y con derivadas b_0, b_1, \dots, b_{n-1}

- Buscamos una función f tal que $\forall i \in \{0, 1, \dots, n-1\}$:
 1. $f(t_i) = a_i$ y $f'(t_i) = b_i$
 2. f interpola suavemente entre a_i y a_{i+1} en el intervalo $[t_i, t_{i+1}]$
- Para lograr esto: **se usan $n - 1$ splines cúbicos de Hermite**, cada uno de ellos desplazado al intervalo $[t_i, t_{i+1}]$ y con los parámetros adecuados

El spline de Hermite en el intervalo $[t_i, t_{i+1}]$

Para valores de t en el intervalo $[t_i, t_{i+1}]$ usaremos, por tanto, el Spline g_i , definido así:

$$g_i(t) \equiv a_i h_{00}(u_i) + a_{i+1} h_{01}(u_i) + s_i b_i h_{10}(u_i) + s_i b_{i+1} h_{11}(u_i),$$

donde:

- u_i = variable (dependiente de t) que va desde 0 hasta 1 cuando t va desde t_i hasta t_{i+1} , es decir:

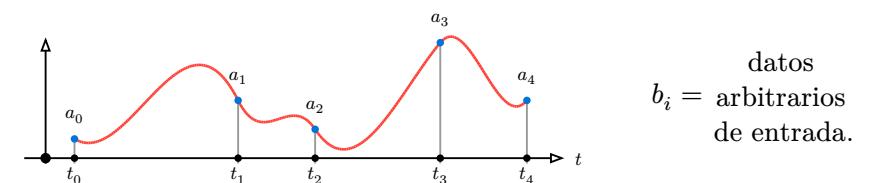
$$u_i \equiv (t - t_i)/s_i$$

- s_i = longitud del intervalo (> 0):

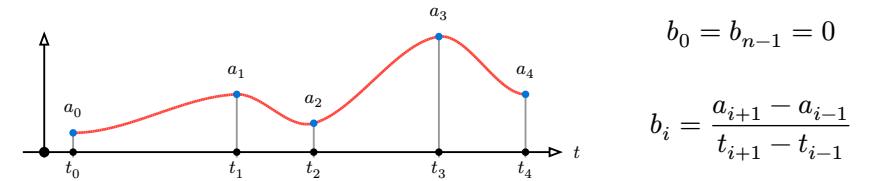
$$s_i \equiv t_{i+1} - t_i$$

Gráfica del spline de Hermite en varios intervalos

Aquí vemos un ejemplo de una curva que interpola entre 5 valores



Se pueden asignar las derivadas de forma que la curva sea más suave:



Subsección 3.2.

Curvas paramétricas.

Curvas paramétricas

Una **curva paramétrica** es una función f que asigna a cada valor de un parámetro t real (en el intervalo $[a, b]$) un punto $f(t)$ en el espacio.

- El *espacio* puede ser el plano (2D) o el espacio tridimensional (3D).

Para expresar una trayectoria de un objeto en el espacio, usaremos *curvas paramétricas*:

- El valor de t es el *tiempo transcurrido* (en segundos, por ejemplo)
- Los puntos $f(t)$ son las posiciones del objeto en el espacio a lo largo del tiempo.
- El intervalo $[a, b]$ es el intervalo de tiempo en el que se está moviendo el objeto, típicamente $a = 0$, y b es la duración.
- Las curvas que vamos a usar son típicamente *continuas* (no hay saltos bruscos de un punto a otro distinto).

Componentes de una curva

Puesto que $\mathbf{f}(t)$ es un punto en el espacio 2D o 3D, sus coordenadas (en algún marco de referencia \mathcal{R}) siempre serán funciones reales de variables real:

- En el espacio 2D, hay dos de estas funciones f_x y f_y :

$$\mathbf{f}(t) = (f_x(t), f_y(t))$$

- En el espacio 3D, hay tres funciones (f_x , f_y y f_z)

$$\mathbf{f}(t) = (f_x(t), f_y(t), f_z(t))$$

Ejemplo de curva paramétrica: línea recta

Un ejemplo sencillo de curva paramétrica en 2D es una curva que representa un simple trayectoria rectilínea entre dos puntos del plano $\mathbf{p}_0 = (x_0, y_0)$ y $\mathbf{p}_1 = (x_1, y_1)$, en el intervalo $[0, 1]$.

- La función \mathbf{f} que define la curva será:

$$\mathbf{f}(t) = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1$$

- Se puede descomponer en dos funciones f_x y f_y cada una de ellas es, por supuesto, una función de t

$$f_x(t) = (1 - t)x_0 + tx_1$$

$$f_y(t) = (1 - t)y_0 + ty_1$$

- La velocidad es constante e igual a la distancia entre los dos puntos (ya que se recorre en una unidad de tiempo).

Ejemplo de curva paramétrica: movimiento circular

Otro ejemplo podría ser un movimiento circular en el plano, con período $T > 0$ segundos, con centro en el origen y radio r , constante

- La función \mathbf{f} que define la curva (suponiendo t en segundos) será:

$$\mathbf{f}(t) = r \cdot (\cos(2\pi t/T), \sin(2\pi t/T))$$

- Al igual que en el ejemplo anterior, obviamente se puede descomponer en dos funciones f_x y f_y

$$f_x(t) = r \cdot \cos(2\pi t/T)$$

$$f_y(t) = r \cdot \sin(2\pi t/T)$$

- Teóricamente está definida para cualquier $t > 0$ (no tiene porque parar nunca).

Ejemplo de curva paramétrica: espiral

Un ejemplo algo más complejo es usar la curva anterior, pero hacer depender el radio r también del tiempo, es decir, se usa una expresión $r(t)$ en lugar de la constante r :

- Una trayectoria en espiral que se aleja indefinidamente del origen, hacemos $r(t) = 1.5t$

$$\mathbf{f}(t) = 1.5t \cdot (\cos(8\pi t), \sin(8\pi t))$$

- Una trayectoria circular que se aleja y acerca al origen, periódicamente:

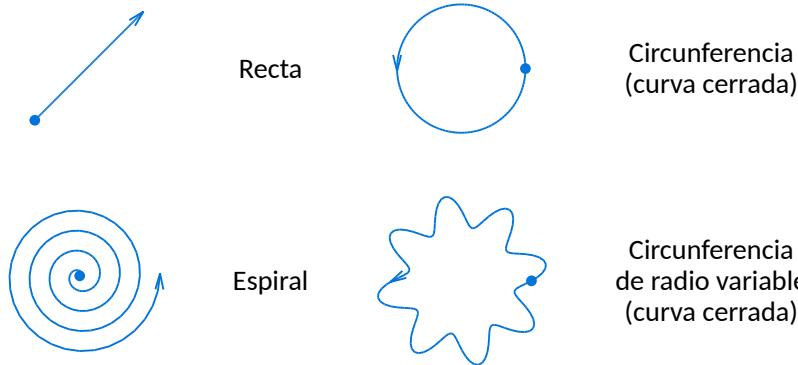
$$\mathbf{f}(t) = r(t) \cdot (\cos(2\pi t), \sin(2\pi t))$$

donde el radio oscila entre 1 y 1.6 (con 8 ciclos por vuelta)

$$r(t) = 1 + 0.3 \cdot (1 + \sin(8 \cdot 2\pi t))$$

Gráficas de las curvas

Aquí vemos, a modo de ejemplo, las gráficas de las cuatro curvas que hemos descrito (se ve un disco en $t = 0$)



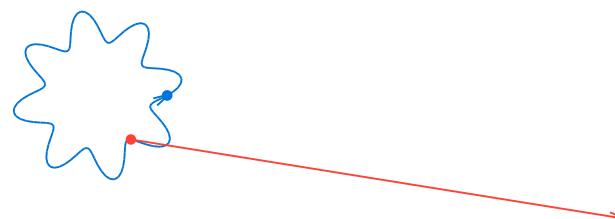
Una curva cerrada es una en la que $f(0) = f(1)$

Ejemplos del vector tangente

En una circunferencia:



En la circunferencia de radio variable:



Vector de velocidad (tangente)

En un punto t a lo largo de una curva f (con componentes f_x, f_y y f_z) se puede definir el vector tangente a la curva en ese punto $v(t)$ como la derivada de f respecto de t .

$$v(t) \equiv \frac{df(t)}{dt} = \left(\frac{df_x(t)}{dt}, \frac{df_y(t)}{dt}, \frac{df_z(t)}{dt} \right)$$

- La dirección de $v(t)$ es tangente a la curva, es la dirección en la que se mueve en t un punto que recorra la curva.
- La longitud de $v(t)$ (se nota como $\|v(t)\|$) es la velocidad a la que se mueve el punto en el instante t .

Cálculo del vector tangente

En muchas aplicaciones queremos usar el vector tangente a una curva en un punto,

- Típicamente querremos alinear un objeto con la dirección de la curva en un punto dado (por ejemplo, si el objeto es un coche, debe estar alineado con la curva que está recorriendo).
- A veces se conoce la expresión analítica exacta de la tangente (p.ej. en una circunferencia, la tangente es perpendicular al radio)
- Si no se conoce la derivada analíticamente, se puede aproximar numéricamente con una diferencia finita:

$$v(t) \approx \frac{f(t + \Delta) - f(t - \Delta)}{2\Delta} \quad \text{donde } \Delta \text{ es pequeño}$$

Curvas 2D y 3D para trayectorias

Subsección 3.3.

Curvas e interpolación en 2D/3D.

3. Interpolación y curvas para animación.
3.3. Curvas e interpolación en 2D/3D..

Splines cúbicos de Hermite

El uso splines cúbicos de Hermite permite definir una curva suave que pasa por una serie de puntos dados y que tiene una dirección y velocidad dadas en cada uno de esos puntos.

- Se definen por tramos, cada tramo es un polinomio cúbico de Hermite.
- Cada tramo se define por dos puntos de control y dos vectores de control (uno en cada extremo).
- La curva es continua en posición y dirección.

Para ello, se considera las componentes X, Y (y Z, si procede) de forma independiente, y se usan las funciones reales de variable real que ya hemos visto.

En muchas aplicaciones será necesario diseñar trayectorias 2D y 3D que sean suaves y que:

- Pasan (o se acercan) a determinados puntos dados como entrada.
- Tienen una dirección y velocidad determinada en esos puntos.

Se pueden usar diversas técnicas de interpolación:

- Splines cúbicos de Hermite (los hemos visto para funciones, ahora los generalizamos a curvas).
- Curvas de Bezziers.
- Curvas B-Spline.

Sesión 11: Animación

Created 2025-12-09

Page 86 / 112.

3. Interpolación y curvas para animación.
3.3. Curvas e interpolación en 2D/3D..

Splines cúbicos de Hermite en 2D

En el caso 2D, se tienen como datos de entrada:

- Una secuencia de instantes de tiempo $t_0 < t_1 \dots t_{n-1}$ en $[0, 1]$
- Una serie de puntos $\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{n-1}$ en el plano, por los cuales se quiere que pase la curva ($\mathbf{p}_i = (x_i, y_i)$)
- Una serie de vectores $\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1}$ que indican la dirección y velocidad de la curva en cada uno de los puntos ($\mathbf{v}_i = (x'_i, y'_i)$)

El problema es encontrar la curva $\mathbf{f} = (f_x, f_y)$ que cumple las condiciones.

- Para eso se resuelve el problema de interpolación para cada una de las funciones f_x y f_y .
- Se pueden usar los Splines cúbicos de Hermite, que ya hemos visto.

Splines cúbicos de Hermite en 2D

Por tanto, imponemos estas condiciones:

- La función f_x cumple: $f(t_i) = x_i$ y $f'(t_i) = x'_i$
- La función f_y cumple: $f(t_i) = y_i$ y $f'(t_i) = y'_i$

Así que usamos los Splines de Hermite en cada tramo para definirlas

$$f_x(t) \equiv x_i h_{00}(u_i) + x_{i+1} h_{10}(u_i) + s_i x'_i h_{01}(u_i) + s_i x'_{i+1} h_{11}(u_i)$$

$$f_y(t) \equiv y_i h_{00}(u_i) + y_{i+1} h_{10}(u_i) + s_i y'_i h_{01}(u_i) + s_i y'_{i+1} h_{11}(u_i)$$

donde i es el índice del tramo donde está t . Vectorialmente, se escribe:

$$\mathbf{f}(t) \equiv \mathbf{p}_i h_{00}(u_i) + \mathbf{p}_{i+1} h_{01}(u_i) + s_i \mathbf{v}_i h_{10}(u_i) + s_i \mathbf{v}_{i+1} h_{11}(u_i),$$

donde $u_i := (t - t_i)/s_i$ y $s_i := t_{i+1} - t_i$

Problema: curva Hermite para una trayectoria

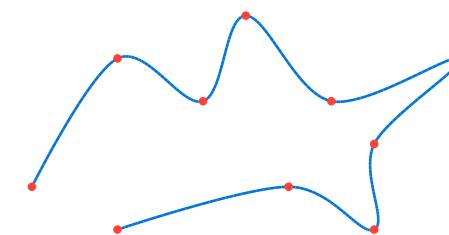
Problema 11.1:

Implementar un proyecto en Godot en el cual el nodo raíz tiene un script que define dos arrays con: una serie de n puntos $\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{n-1}$ del plano $y = 0$, y: una serie de instantes de tiempo t_0, t_1, \dots, t_{n-1} (en segundos) con $t_0 = 0$

- Sitúa en cada uno de esos puntos un disco pequeño visible, a modo de marcador.
- Incluye una función para calcular la posición y velocidad de la curva de Hermite que pasa por los puntos en los instantes dados, a partir de un t en $[0, t_{n-1}]$. En cada punto \mathbf{p}_i el vector de velocidad \mathbf{v}_i se calcula a partir de los puntos anterior y siguiente (como en el ejemplo anterior).
- En el método `_process(delta)` del script, calcula la posición y velocidad de la curva en el tiempo transcurrido desde el inicio, y mueve un objeto (un coche, por ejemplo) a esa posición, alineado con la dirección de la curva (usando el vector de velocidad como vector de dirección).

Curva Hermite por varios puntos

Una curva (en azul) que pasa por varios puntos (en rojo).



$$\mathbf{v}_0 = \mathbf{v}_{n-1} = (0, 0)$$

$$\mathbf{v}_i = \frac{\mathbf{p}_{i+1} - \mathbf{p}_{i-1}}{t_{i+1} - t_{i-1}}$$

- En el primer y último puntos la velocidad es nula (se comienza y acaba de forma suave).
- En el resto de puntos, la velocidad es el vector que va desde el punto anterior al siguiente (dividida por el tiempo entre uno y otro).
- Este tipo de curvas son continuas, con derivada (tangente) continua, pero la curvatura es discontinua.

Curvas de Beziers (cuadráticas)

La **Curva de Beziers** permite diseñar una trayectoria desde un punto \mathbf{p}_0 hasta otro punto \mathbf{p}_2 , pero de manera que la forma de la curva está influenciada por un tercer punto \mathbf{p}_1 . Para ello:

- Se define la curva $\mathbf{q}_0(t)$, que interpola linealmente desde \mathbf{p}_0 hasta \mathbf{p}_1 (es un segmento de recta), e igualmente, se define $\mathbf{q}_1(t)$ como la recta desde \mathbf{p}_1 hasta \mathbf{p}_2 :

$$\mathbf{q}_0(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1 \quad \mathbf{q}_1(t) = (1-t)\mathbf{p}_1 + t\mathbf{p}_2(t)$$

- La **Curva de Beziers (cuadrática)** (\mathbf{B}) es la curva \mathbf{f} resultado de interpolar entre $\mathbf{q}_0(t)$ y $\mathbf{q}_1(t)$, es decir:

$$\mathbf{B}(t) \equiv (1-t)\mathbf{q}_0(t) + t\mathbf{q}_1(t)$$

Propiedades de la curva de Beziers cuadrática

Esta curva cumple las siguientes propiedades:

- La curva sale de p_0 (en $t = 0$) y llega a p_2 (en $t = 1$)
- No pasa necesariamente por p_1 , pero este punto determina la dirección y velocidad al salir de p_0 y al llegar a p_2 .
- La curva (en X y en Y) se puede expresar como un polinomio de grado 2 en t

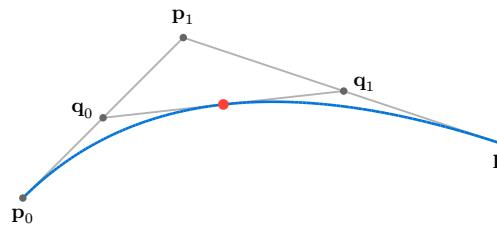
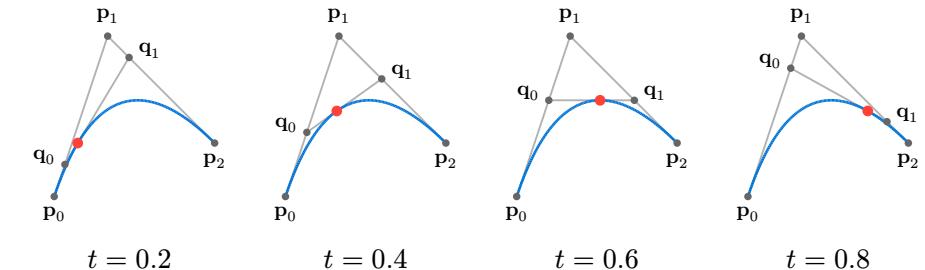


Diagrama de una curva de Beziers cuadrática

Aquí vemos los tres puntos, las dos rectas entre p_0 , p_1 y p_2 , así como la recta entre q_0 y q_1 , y el punto en la curva (en rojo):



Curva de Beziers cúbica, y generalización

Las **curvas de Beziers cúbicas** usan 4 puntos (p_0, p_1, p_2, p_3) en vez de 3

- Con esos puntos se pueden definir dos curvas de Beziers cuadráticas: la curva B_0 que usa p_0, p_1, p_2 , y la curva B_1 que usa p_1, p_2, p_3 .
- A partir de B_0 y B_1 se define la **Curva de Beziers cúbica** (polinomios de grado 3), interpolando:

$$B(t) \equiv (1-t)B_0(t) + tB_1(t)$$

- El esquema se puede generalizar recursivamente a n puntos usando polinomios de grado $n - 1$
- Las curvas pasan por el primer y último puntos.
- Los puntos suelen denominarse **puntos de control** de las curvas.

Curvas de B-spline

Las **Curvas B-spline** son parecidas a las curvas de Beziers, pero permiten mayor control sobre la forma de la curva.

- Los puntos distintos del primero y el último se suelen llamar **puntos de control**.
- Se permite controlar mejor las velocidades de la curva, usando un vector de valores reales, llamado **vector de nodos**.
- Al igual que las curvas de Beziers, se pueden definir recursivamente, para cualquier grado de los polinomios.
- Son ampliamente usadas en diseño asistido por ordenador y animación.

Introducción

Sección 4.

Animaciones en Godot

- 1. Animaciones con *AnimationPlayer*.
- 2. Animaciones mediante scripts

Godot ofrece diversas opciones para incorporar animaciones en un proyecto:

- Animaciones creadas con el editor usando nodos de tipo *AnimationPlayer*.
- Animaciones programadas mediante scripts, usando el método *_process*
- Animaciones relacionadas con la simulación física, usando el método *_physics_process*

Sesión 11: Animación

Created 2025-12-09

Page 98 / 112.

4. Animaciones en Godot.

4.1. Animaciones con *AnimationPlayer*.

Los nodos de tipo *AnimationPlayer*. Pistas.

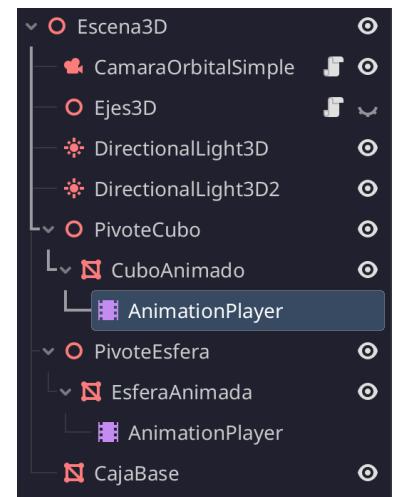
Subsección 4.1.

Animaciones con *AnimationPlayer*.

La clase *AnimationPlayer* (derivada de *Node*) permite crear animaciones para aplicaciones 2d o 3D, mediante un editor gráfico.

Cada nodo *AnimationPlayer* puede coneter una o varias **pistas** (*tracks*). Una pista es una especificación de como varía con el tiempo una **propiedad de algún otro nodo del árbol de escena**.

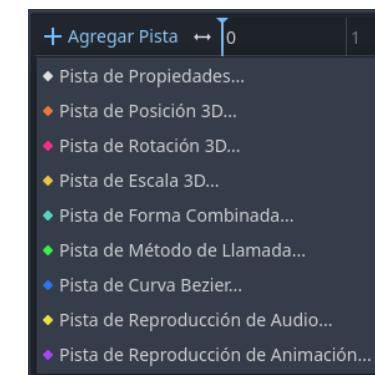
Un árbol de escena puede contener uno o varios nodos de este tipo.



Tipos de pistas.

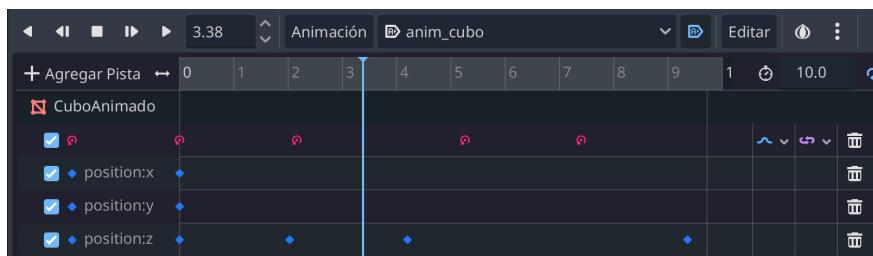
Una pistas pueden asociarse a distintos tipo de propiedades:

- Propiedades asociadas a la transformación del objeto: posición, rotación, y escala.
- Cualquier propiedad numérica, que varía con el tiempo según una curva de Bezziers.
- Cualquier propiedad no numérica del objeto, por ejemplo su textura, el material, o la malla usada (en un *MeshInstance*).



Animación del cubo

Aquí vemos el panel correspondiente al nodo *AnimationPlayer* del cubo. La animación dura 10 segundos, se repite, y se activa al iniciar la aplicación.



Tiene 4 pistas (tracks):

- Una controla la rotación. Produce rotaciones entorno al eje Y.
- Otras tres controlan la posición en los ejes X, Y y Z. Cada keyframe tiene asociada una posición distinta en el eje Z. Son de tipo B-spline.

Keyframes (cuadros clave)

En cada pista se debe añadir al menos un **keyframe** (lo usual es añadir varios)

- Un keyframe asocia un valor concreto a la propiedad en un instante de tiempo concreto.
- Para propiedades numéricas, se puede configurar como se interpola el valor entre dos keyframes consecutivos:
 - ▶ linealmente,
 - ▶ usando una curva suave
 - ▶ mediante curvas de Bezziers (B-spline). Estas se pueden editar interactivamente.
- Para propiedades no numéricas, el valor establecido en un keyframe se mantiene hasta el siguiente keyframe o hasta el final de la animación.

Curvas de la posición Z del cubo

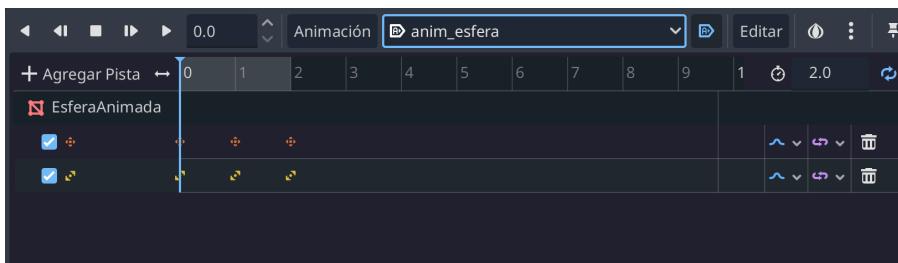
La posición del cubo se controla mediante tres curvas de tipo B-spline (una por cada eje). Aquí vemos la curva correspondiente a la posición en Z (que es la que varía):



Cada punto blanco es un **keyframe**. Se pueden arrastrar para cambiar su tiempo y su valor. Los segmentos que hay sobre ellos permiten modificar la derivada (pendiente) en cada uno de ellos.

Animación de la esfera

Aquí vemos el panel correspondiente al nodo ***AnimationPlayer*** de la esfera. La animación dura dos segundos, se repite, y se activa al iniciar la aplicación.

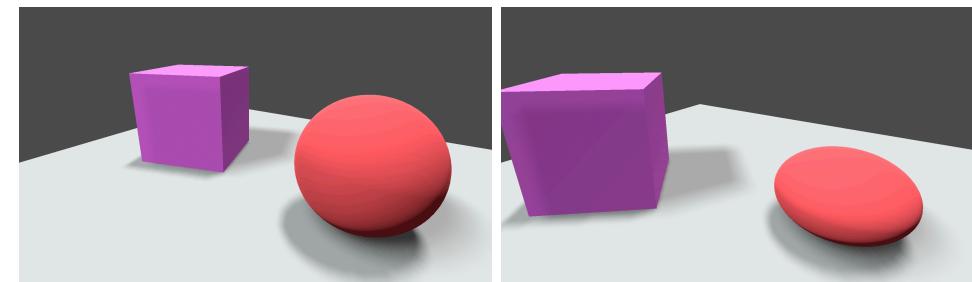


Tiene dos pistas (*tracks*):

- Una para controlar la posición. Cada *keyframe* tiene asociada una posición con una altura en Y distinta.
- Otra para controlar la escala. Ahora se asocia un escalado en Y diferente en cada *keyframe*.

Una y otra imagen

Aquí vemos dos capturas de la aplicación en dos estados de la animación:



- El cuadrado de la izquierda se desplaza en horizontal y rota.
- La esfera de la derecha se desplaza en vertical y se escala.

El método *process*

Otra forma de crear animaciones es mediante *scripts*. Se hace escribiendo código en el método ***_process*** (de la clase ***Node***).

- Se invoca para cada nodo presente en un árbol de escena, antes de cada frame.
- Tiene un parámetro ***delta***, de tipo ***float***, que indica el tiempo (en segundos) transcurrido desde el último frame, se trunca a un límite máximo si hay un retardo grande (en una prueba ese límite es de 133 ms).
- Normalmente representa el tiempo real, pero si se trunca al valor máximo, ya no es así.
- En cada frame, el orden de invocación es:
 - ▶ En orden creciente de la ***prioridad***, es decir, el valor de la propiedad entera ***process_priority*** (vale 0 por defecto en todos los nodos).
 - ▶ Entre nodos de igual prioridad, **se usa un recorrido en pre-orden del árbol** (de arriba a abajo en el panel del árbol de escena del editor).

Animaciones programadas

El código asociado al método `_process` puede, a modo de ejemplo:

- Modificar cualquier propiedad de un nodo, por ejemplo:
 - ▶ la posición, rotación o escala de un nodo **Spatial** (3D) o **Node2D** (2D)
 - ▶ la visibilidad de un nodo
 - ▶ el color o cualquier otro atributo del material de un nodo
- Mover cámaras, cambiar sus parámetros.
- Mover fuentes de luz, o insertar o destruir fuentes de luz.
- Crear o destruir nodos.
- Insertar nuevos nodos en un árbol.
- Cambiar el lugar de un nodo.
- Poner en marcha o parar audios o vídeos.
- Cualquier otra acción programable mediante código GDScript.

Problemas: animaciones de ejemplo

Problema 11.3:

Desarrolla un proyecto Godot para el ejemplo de animación de un **reloj con tres agujas** que se indica en la subsección 1.5 de este PDF.

Problema 11.4:

Desarrolla un proyecto Godot para el ejemplo de animación de un **péndulo** que se indica en la subsección 1.5 de este PDF.

Problema 11.5:

Desarrolla un proyecto Godot para el ejemplo de animación de una **bala de cañón** que se indica en la subsección 1.5 de este PDF.

Problema: posición oscilante

Problema 11.2:

Crea un proyecto Godot con una animación de una esfera cuya posición en X oscile periódicamente, con estas condiciones:

- El centro de la esfera tiene coordenada Z igual a 0, su coordenada Y es igual al radio, y su coordenada X varía entre $-s$ y $+s$, donde $s > 0$ es una constante declarada en el script.
- El período (tiempo en volver al mismo punto viajando en la misma dirección) es una constante $T > 0$ declarada en el script (con unidades de segundos).
- La esfera se mueve siempre a velocidad constante en magnitud (es siempre s/T), y el signo depende de la dirección.
- Tu animación debe producir esa velocidad constante, incluso teniendo en cuenta que los sucesivos valores de **delta** pueden cambiar entre frames. Especialmente, la magnitud de la velocidad debe ser constante aunque entre dos frames haya ocurrido un cambio de dirección en un extremo.

Fin de transparencias.