

CC3501

Rendering

Pipeline

Eduardo Graells-Garrido & Iván Sipirán
Primavera 2022

¿Qué es rendering? ¿Por qué “pipeline”?

La renderización (rendering) se define como un proceso en secuencia que convierte la descripción de una escena en una imagen 2D.

La escena puede ser 2D, 3D, dinámica, etc. El proceso es el mismo.

El proceso es secuencial. De allí que sea una pipeline (línea de tuberías).

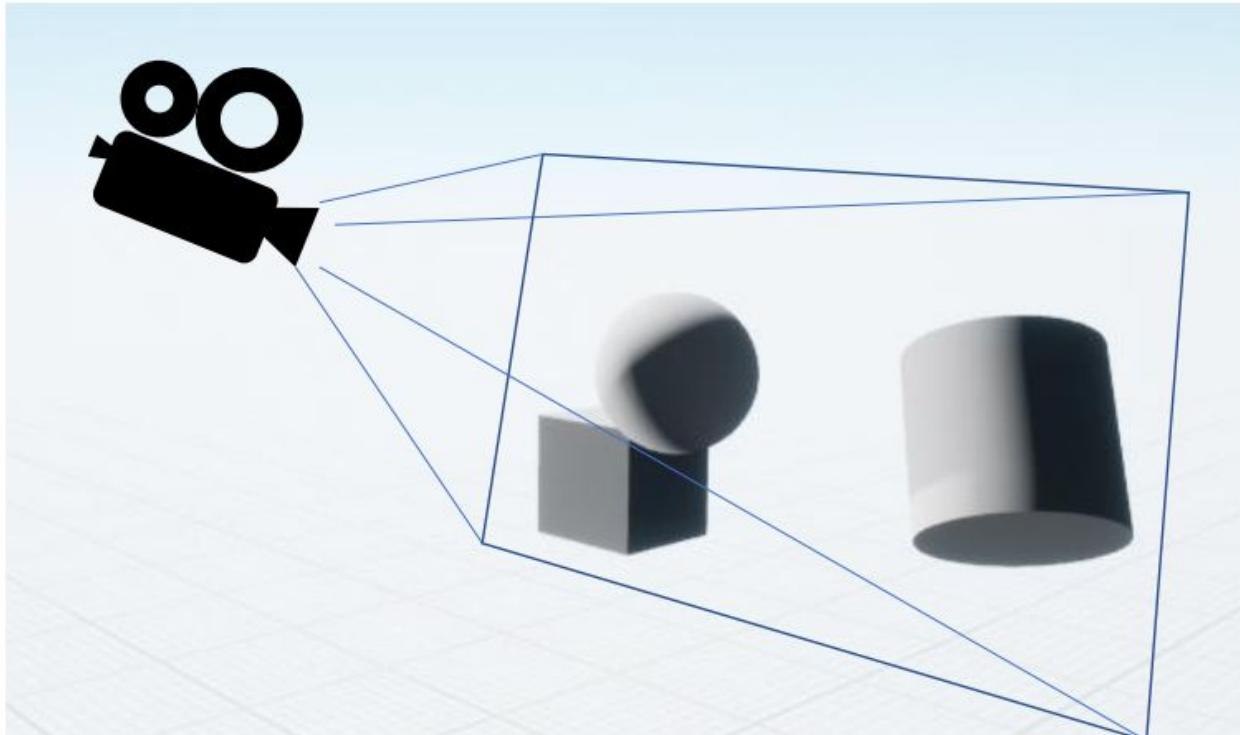
Descripción de la Escena: Modelos

Modelos geométricos, 3D o 2D.



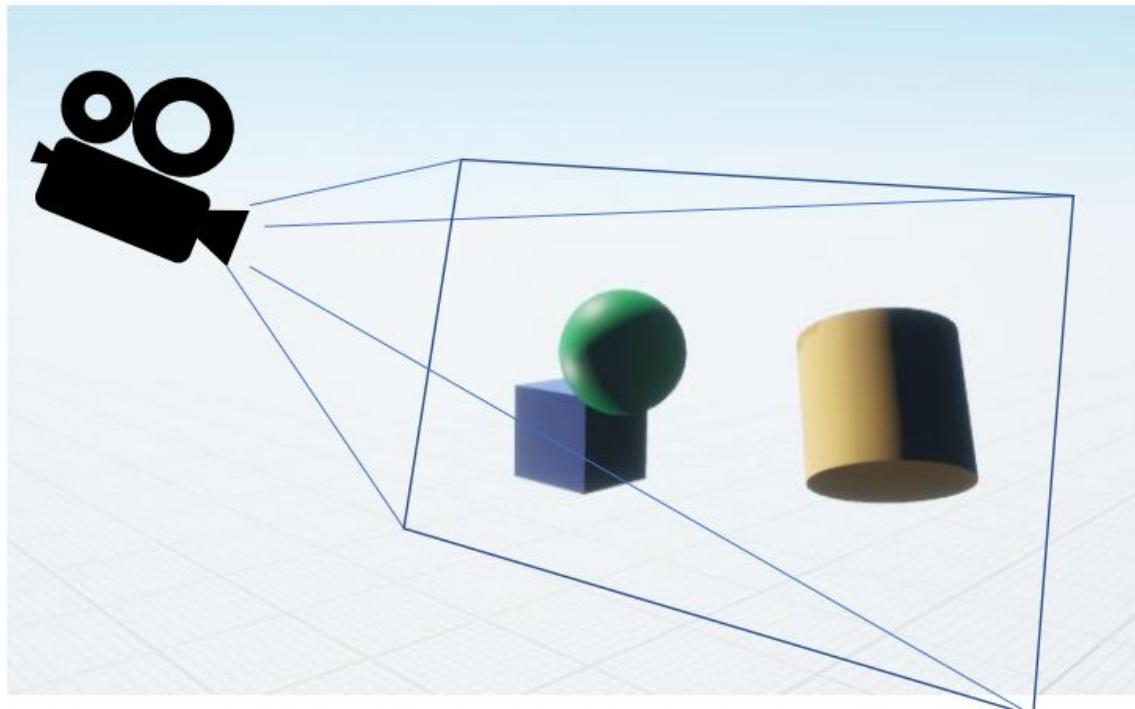
Descripción: Cámara y Escena

Los modelos se posicionan y orientan. Hay una cámara que los observa.



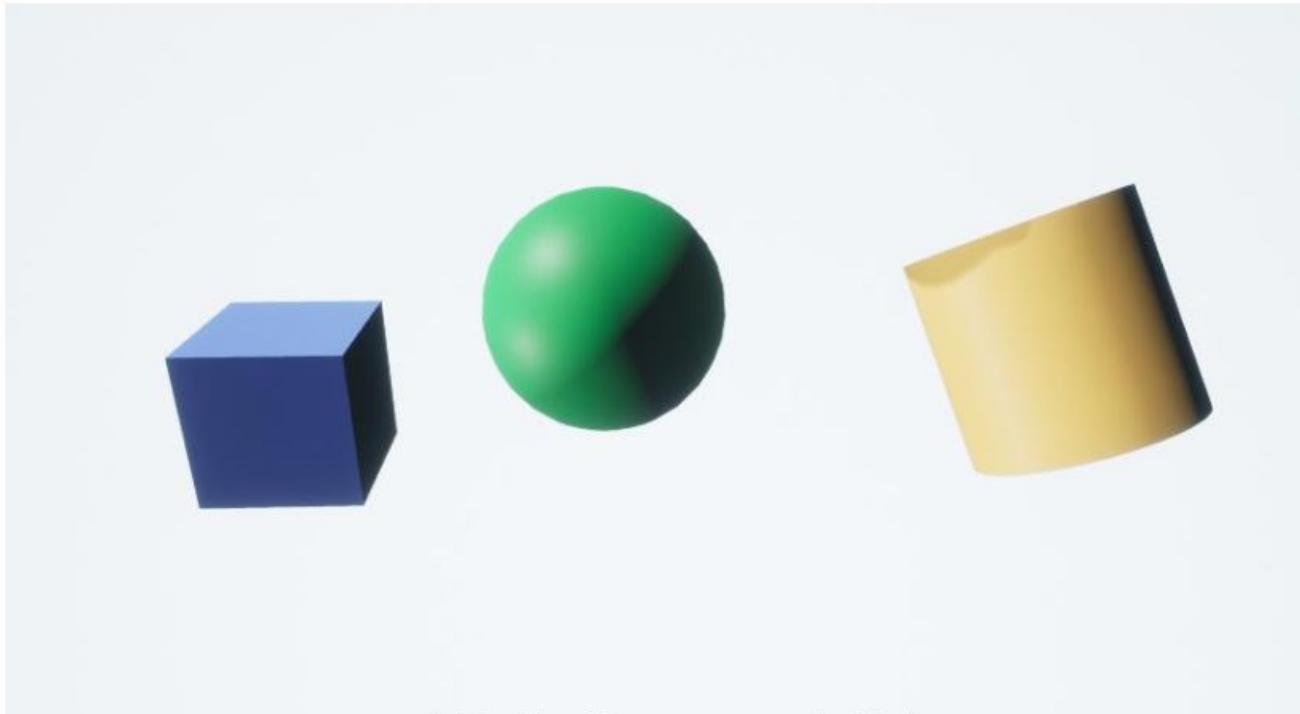
Descripción: Propiedades de Iluminación y Material

Colores, fuentes de luces, materiales.

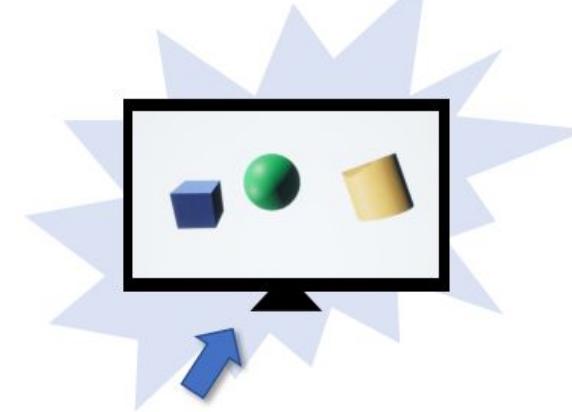
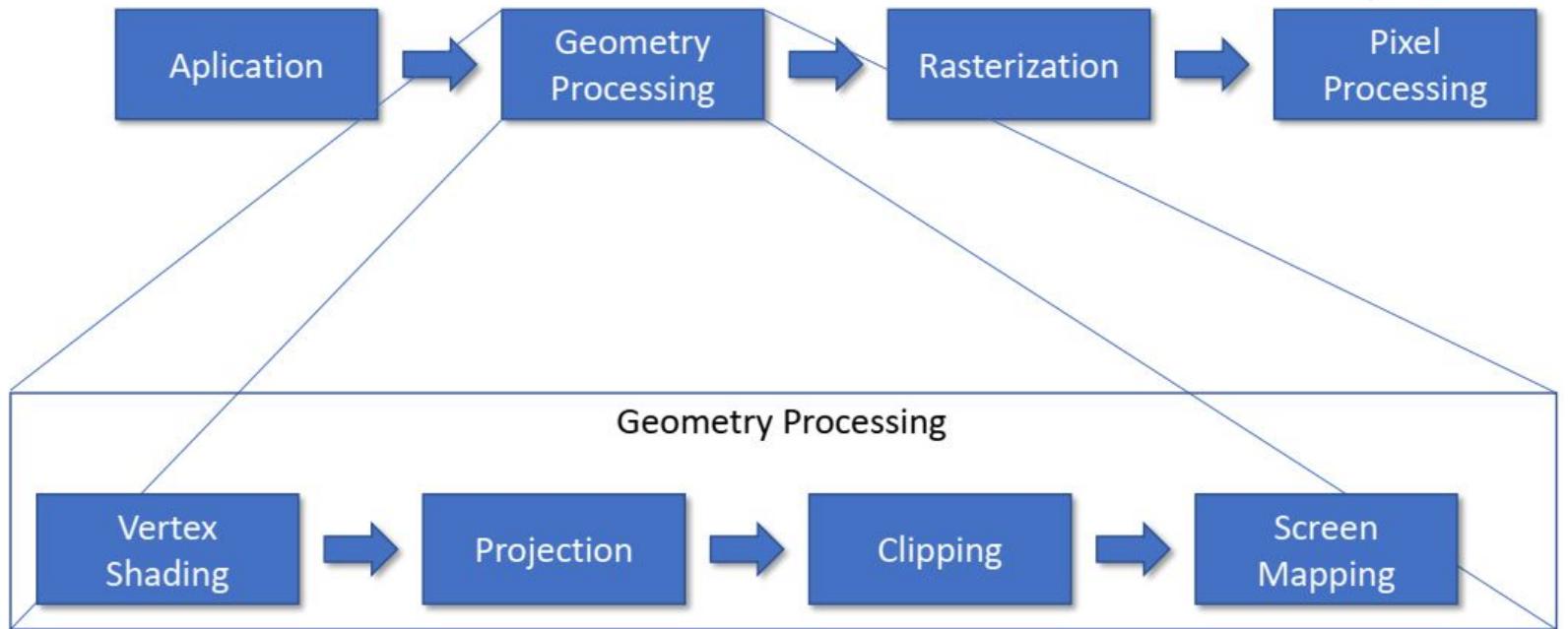


Resultado

La imagen como se ve desde la cámara.



Graphics Rendering Pipeline

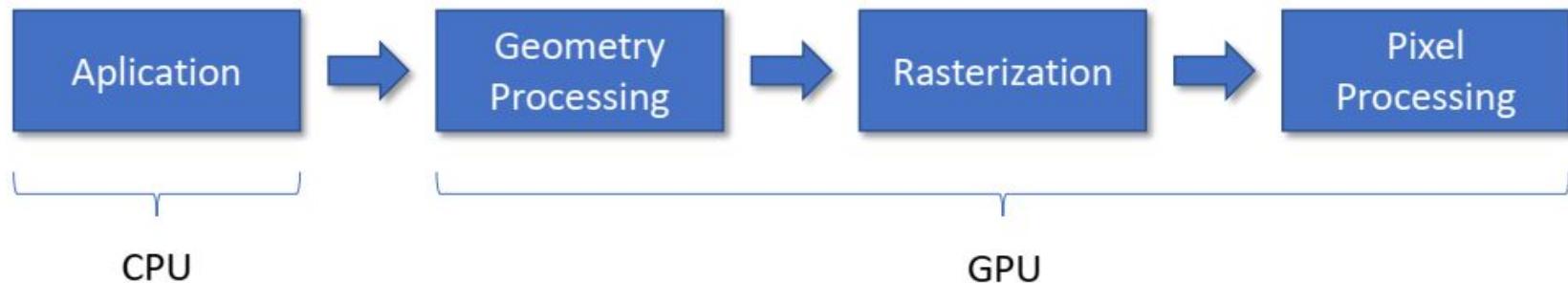


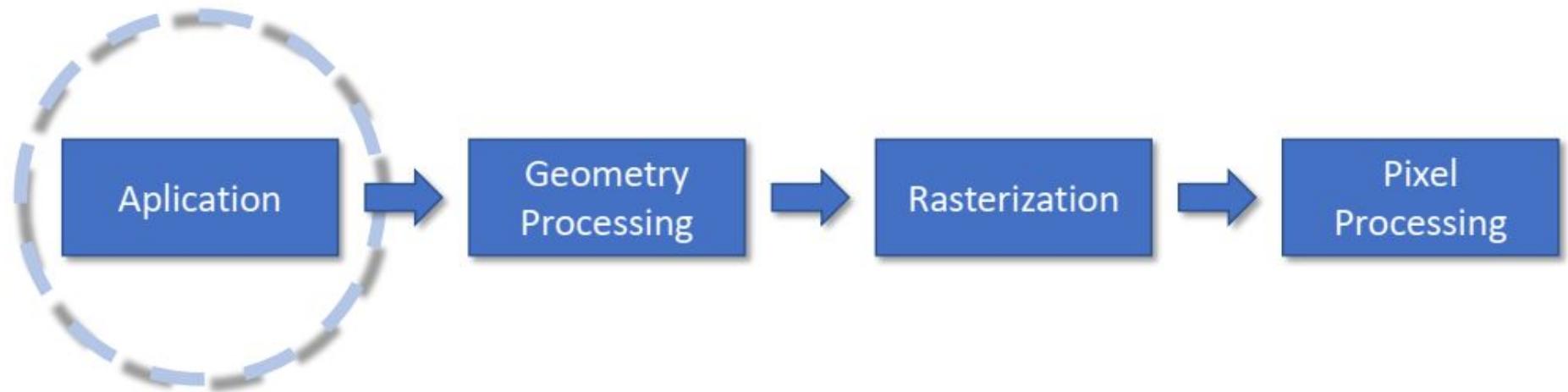
¿Dónde se realiza la computación?

A grandes rasgos, se divide de esta manera.

Cada etapa es, en sí misma, un pipeline.

¡En la actualidad hay CPU y GPU en todas las etapas! Pero si esquematizamos de acuerdo a la mayoría de los casos de uso, se obtiene esta división.





Aplicación

Se suele ejecutar en la CPU:

Simulación física

Interacción entre elementos (colisiones)

Input (controles)

Red

Todo lo que no sea gráfico

Y algunas cosas gráficas.

Aquí es donde **tenemos mayor control**, porque se ejecuta principalmente nuestro código.

La etapa de la Aplicación

Desde aquí generamos el input para la siguiente etapa, en forma de **primitivas gráficas**.

- Puntos, líneas y triángulos.

Son los elementos básicos con los que construimos elementos más complejos.

Primitivas

El “Conejo de Stanford” (Stanford Bunny) es el modelo 3D de un 🐰 de cerámica creado en 1994.

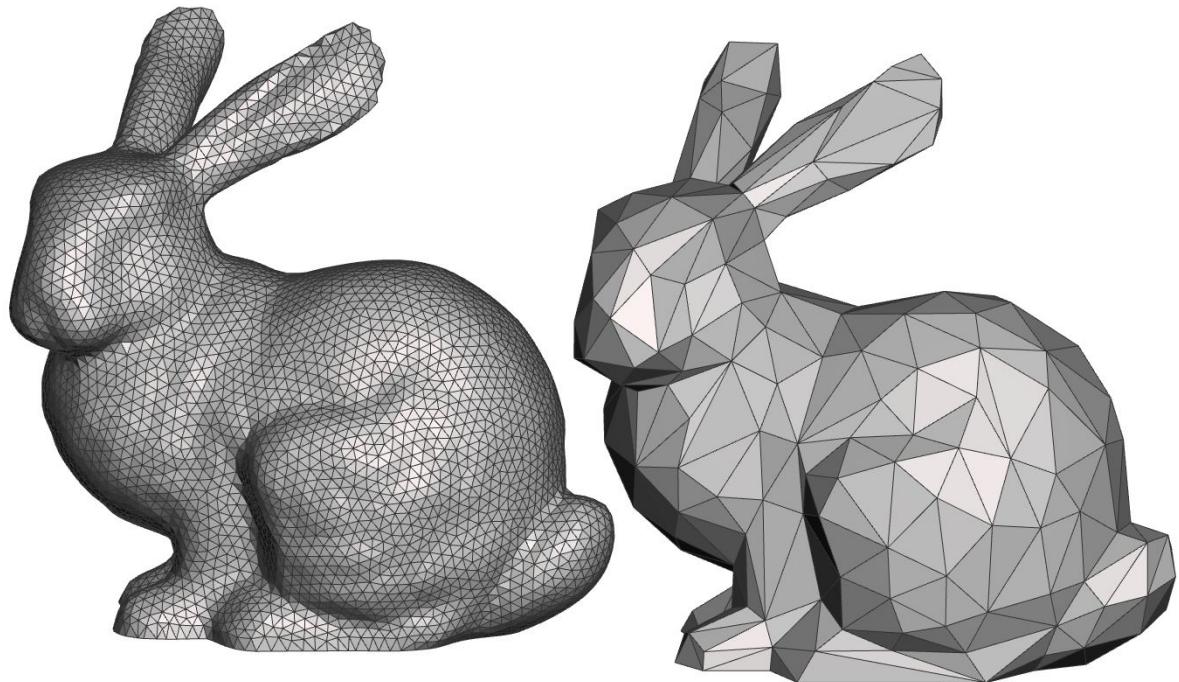
En sus distintas versiones lo podrán encontrar con distintas resoluciones. La original tiene 69,451 triángulos.

¿Alguna idea de porqué es así?

¿Por qué un rectángulo no es una primitiva?

Explóralo en Wikipedia:

https://en.wikipedia.org/wiki/Stanford_bunny#/media/File:Stanford_Bunny.stl



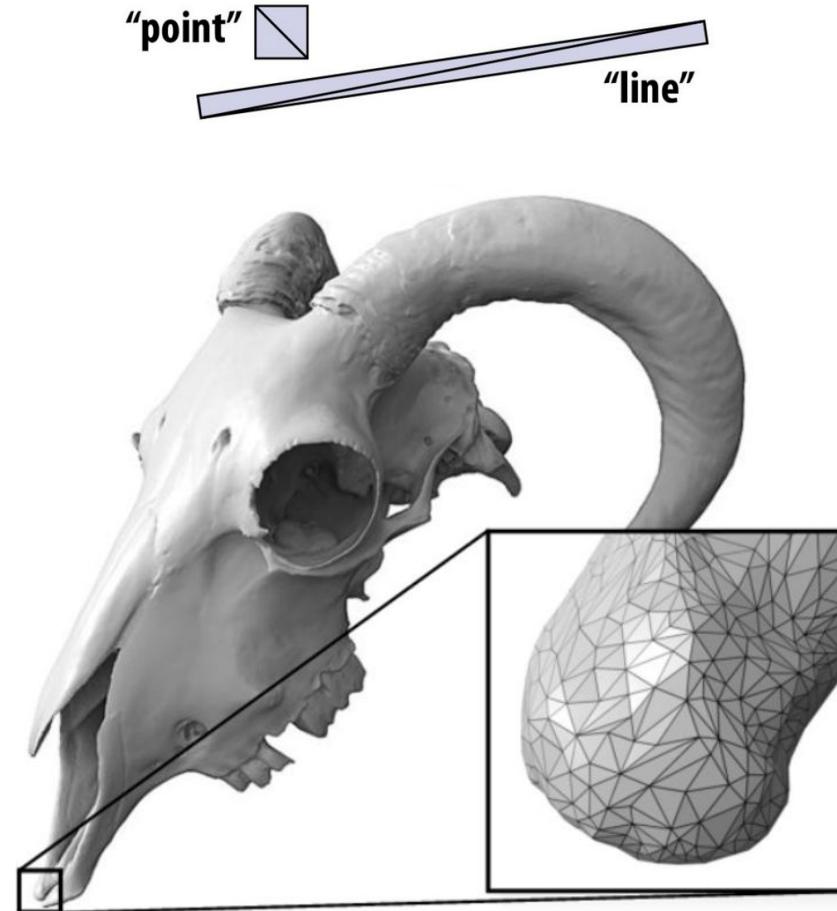
Triángulos

Pueden aproximar cualquier superficie.

Siempre son planares. Tienen una normal bien definida.

Usando coordenadas baricéntricas podemos interpolar cualquier dato de sus vértices en la superficie.

Más importante: todo se convierte en triángulos (incluso puntos y líneas). Por tanto, se ha trabajado en tener una cadena optimizada para graficarlos.



Cornell Box (1984)

El conejo en sí mismo es un objeto.

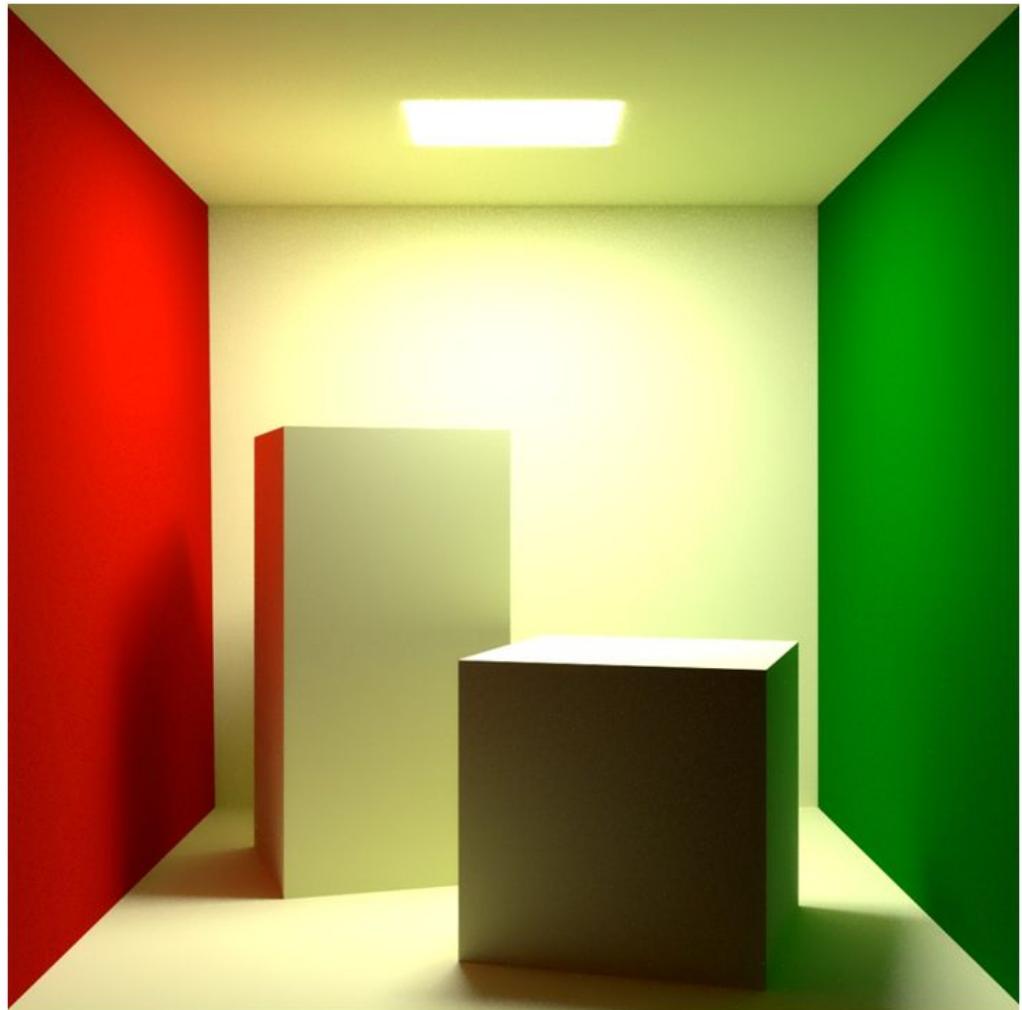
La caja de Cornell es una **escena**: contiene más de un objeto.

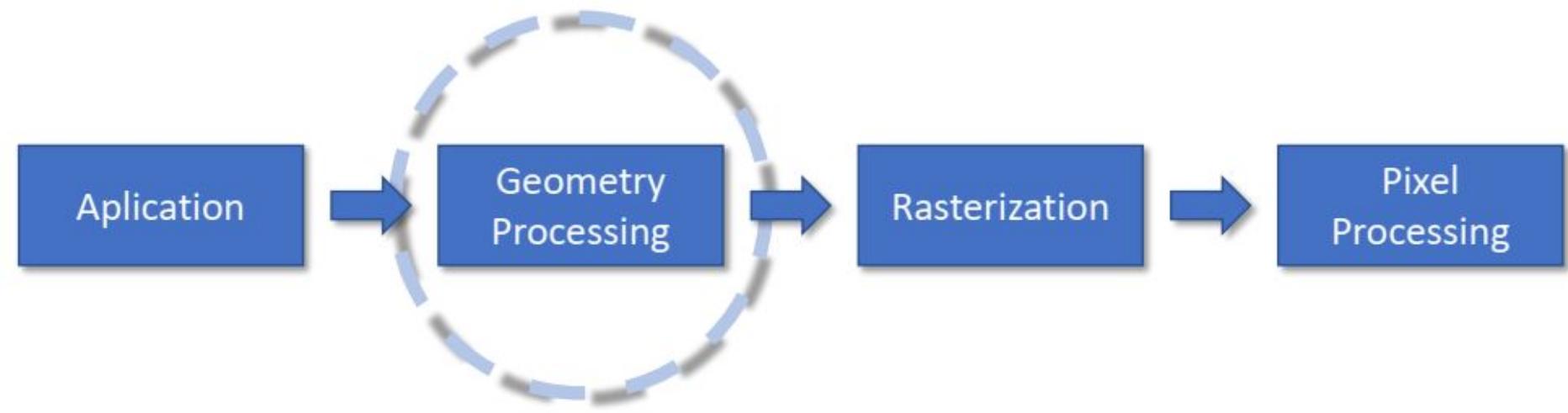
También contiene una fuente de luz.

Los elementos tienen propiedades: no tienen el mismo color.

En general, la etapa de Aplicación le entrega una escena a la siguiente etapa.

Lo que se entrega es la **lista de primitivas gráficas que conforman la escena**.





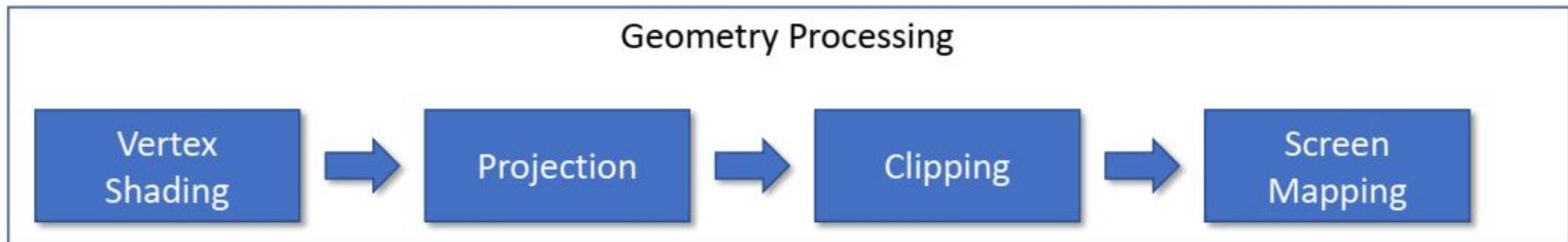
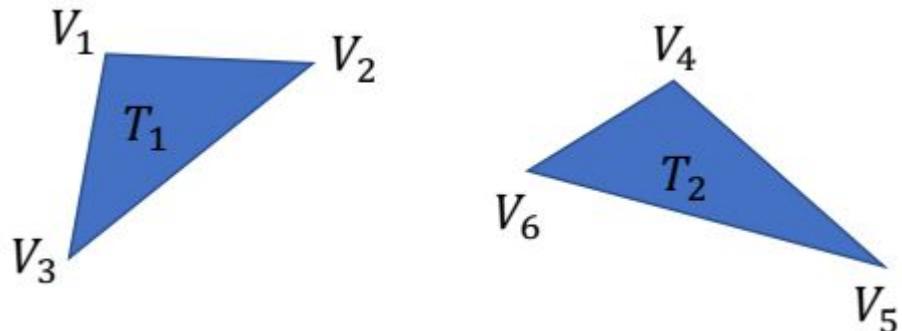
Geometry Processing

Esta etapa procesa las primitivas a nivel de **vértice**. Todo cálculo geométrico se hace a ese nivel.

Se divide, a su vez, en un conjunto estándar de sub-etapas.

En esta etapa usualmente no tenemos control.

Sí tenemos flexibilidad. Pero no control.



Vertex Shading

Transforma la posición de los vértices a un sistema de coordenadas estándar para visualización en pantalla, calcula atributos de los vértices (color, normal, etc)

Projection

Proyecta los vértices sobre un volumen definido por el campo de visión de la cámara utilizada

Clipping

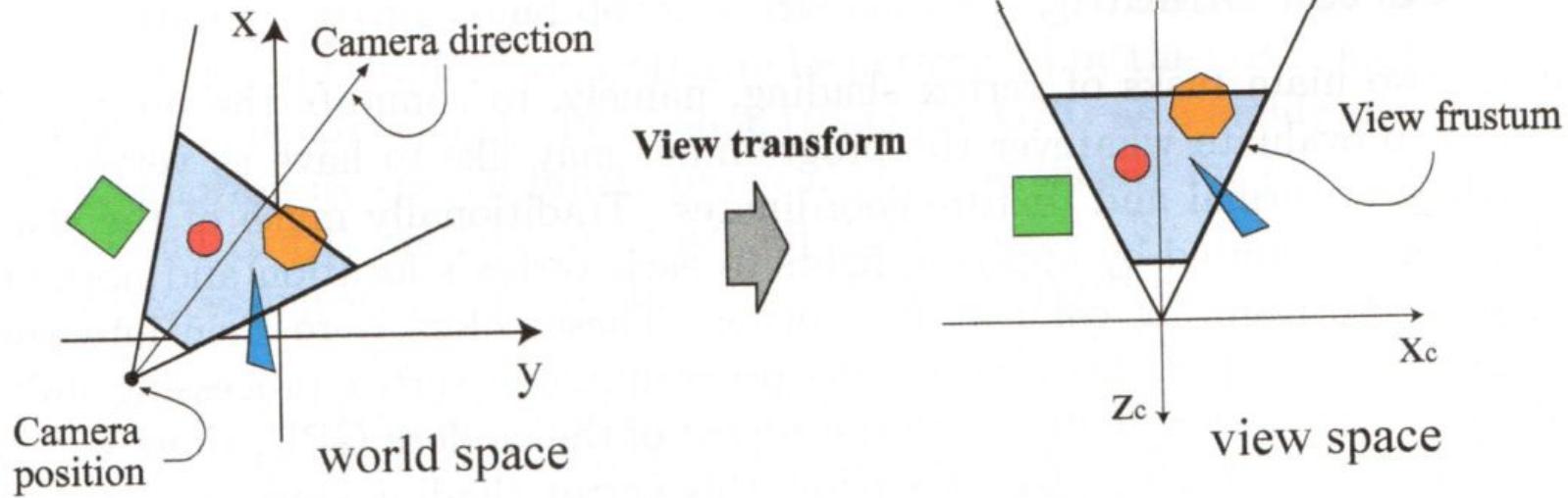
Filtra toda figura que no se encuentre en el volumen de visión

Screen Mapping

Asocia los vértices a las coordenadas de la pantalla

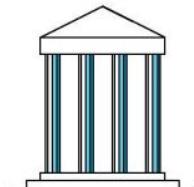
Vertex Shading

El proceso de transformación se llama “view transform” y busca llevar el mundo a un sistema de coordenadas estándar (más adelante veremos más detalles sobre esto).



Projection

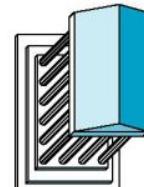
La transformación de vista define la manera en que veremos el mundo virtual. ¡Existen distintas proyecciones, la perspectiva “humana” es solo una de ellas!



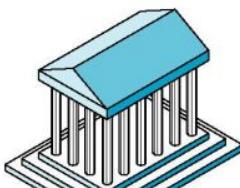
Front elevation



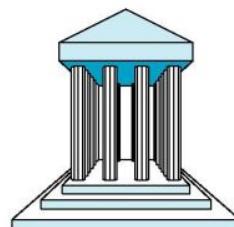
Elevation oblique



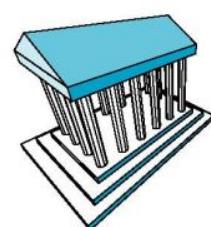
Plan oblique



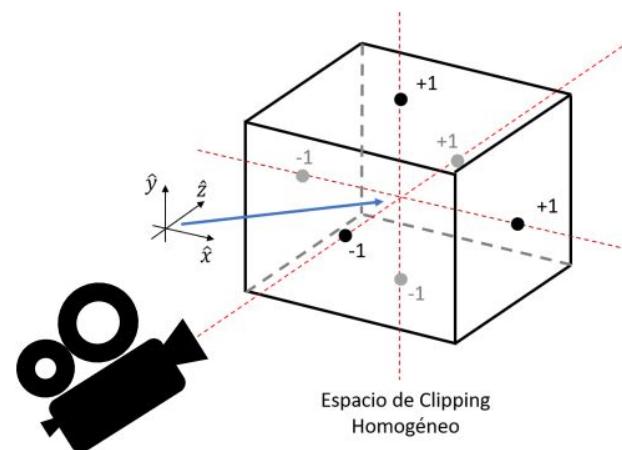
Isometric



One-point perspective

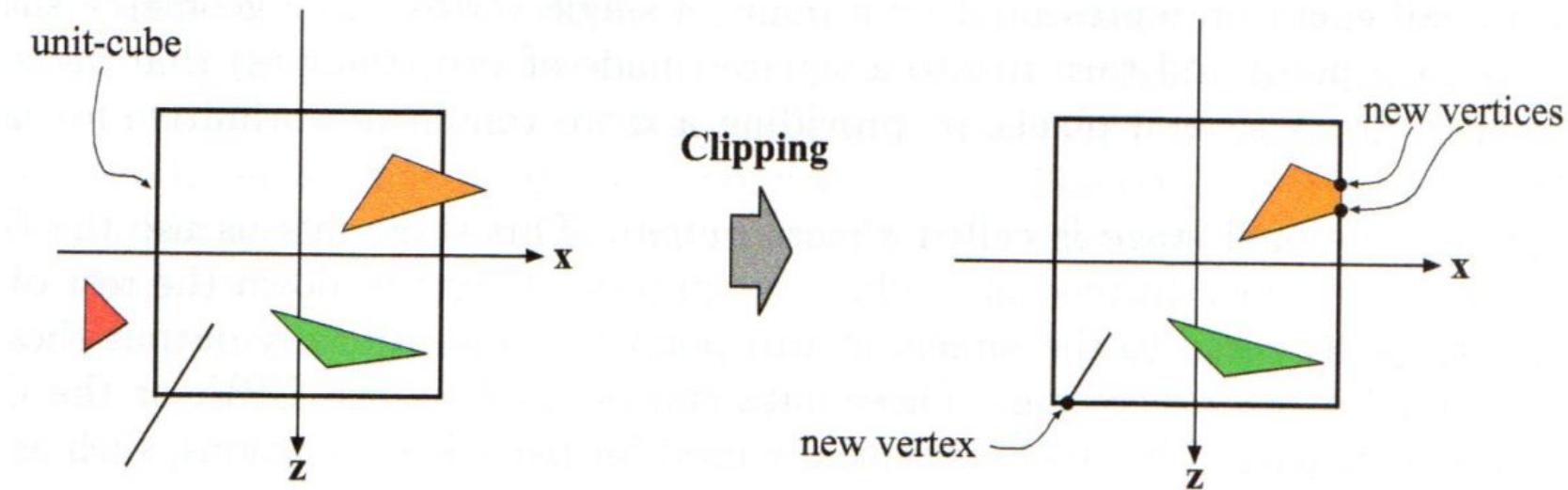


Three-point perspective



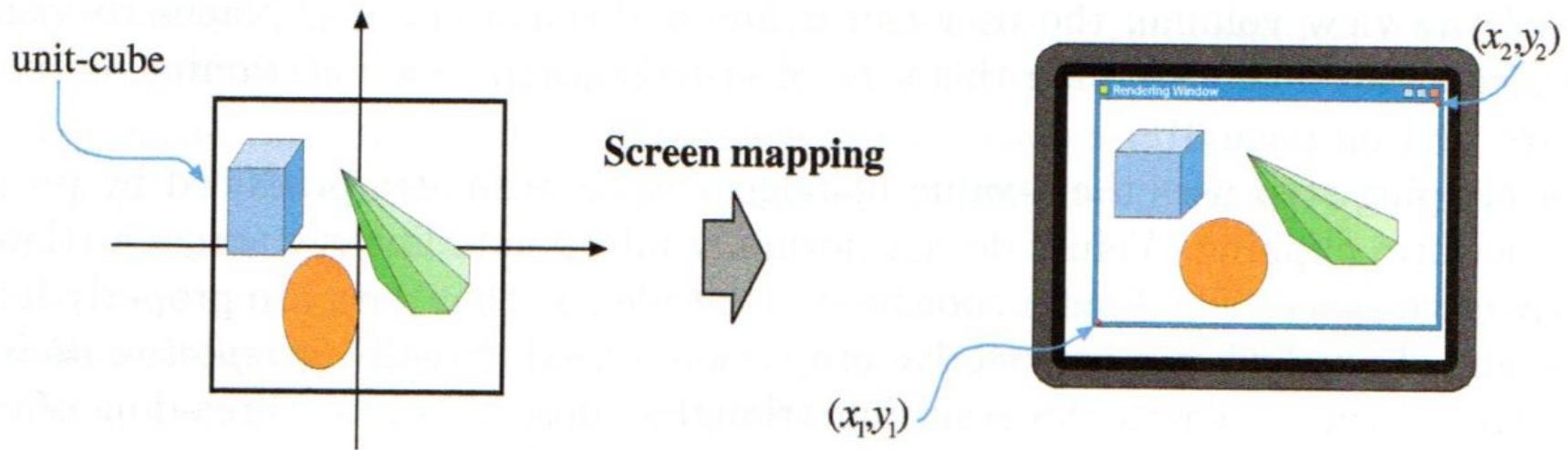
Clipping

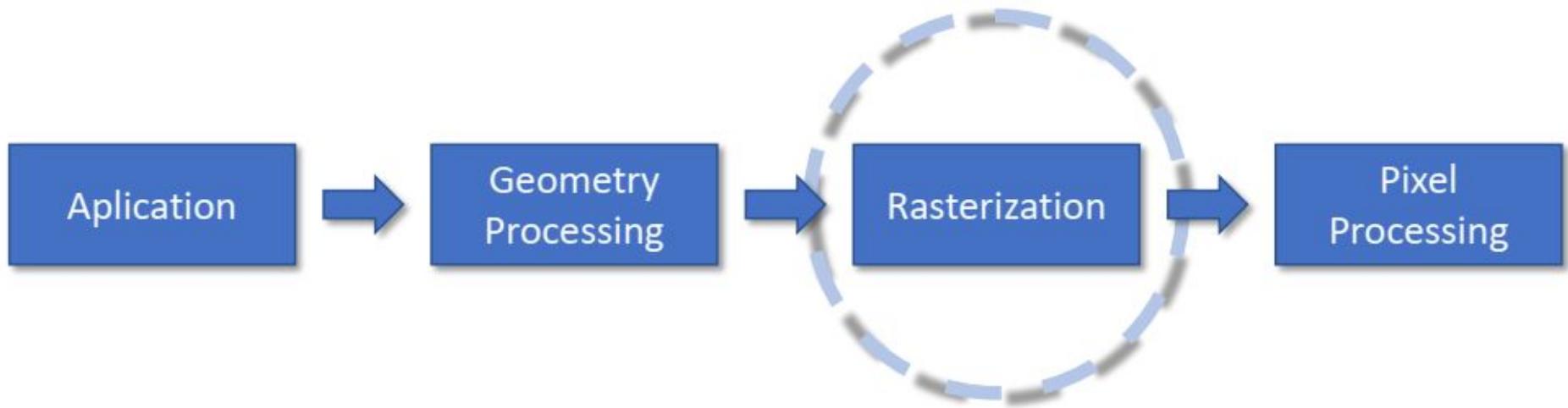
La operación de clipping permite usar nuestros recursos en lo que es visible, ya que el proceso de convertir la escena en pixeles es caro.



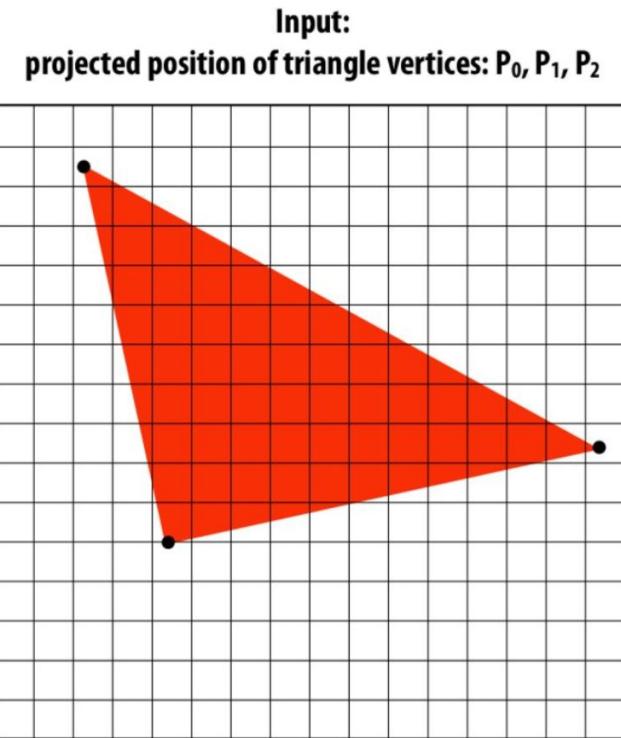
Screen Mapping

El proceso de screen mapping nos permite convertir las coordenadas de los vértices desde el volumen normalizado de vista hasta las coordenadas de la pantalla o ventana.

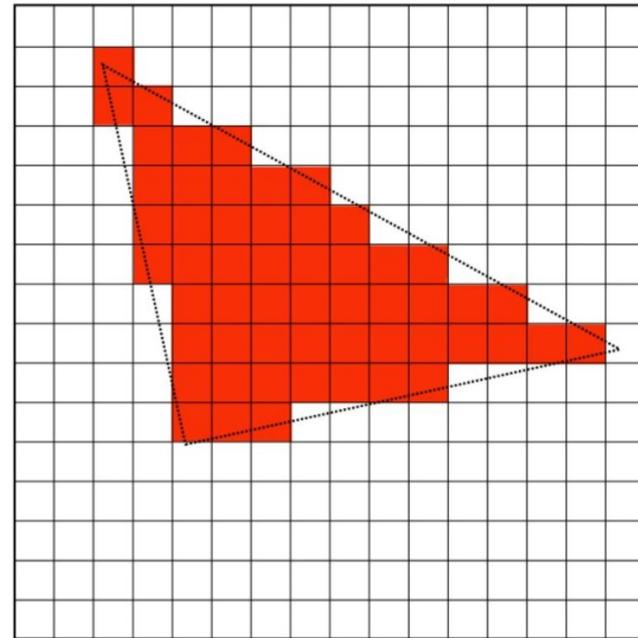




De Primitivas a Pixelles: Rasterización



Output:
set of pixels “covered” by the triangle



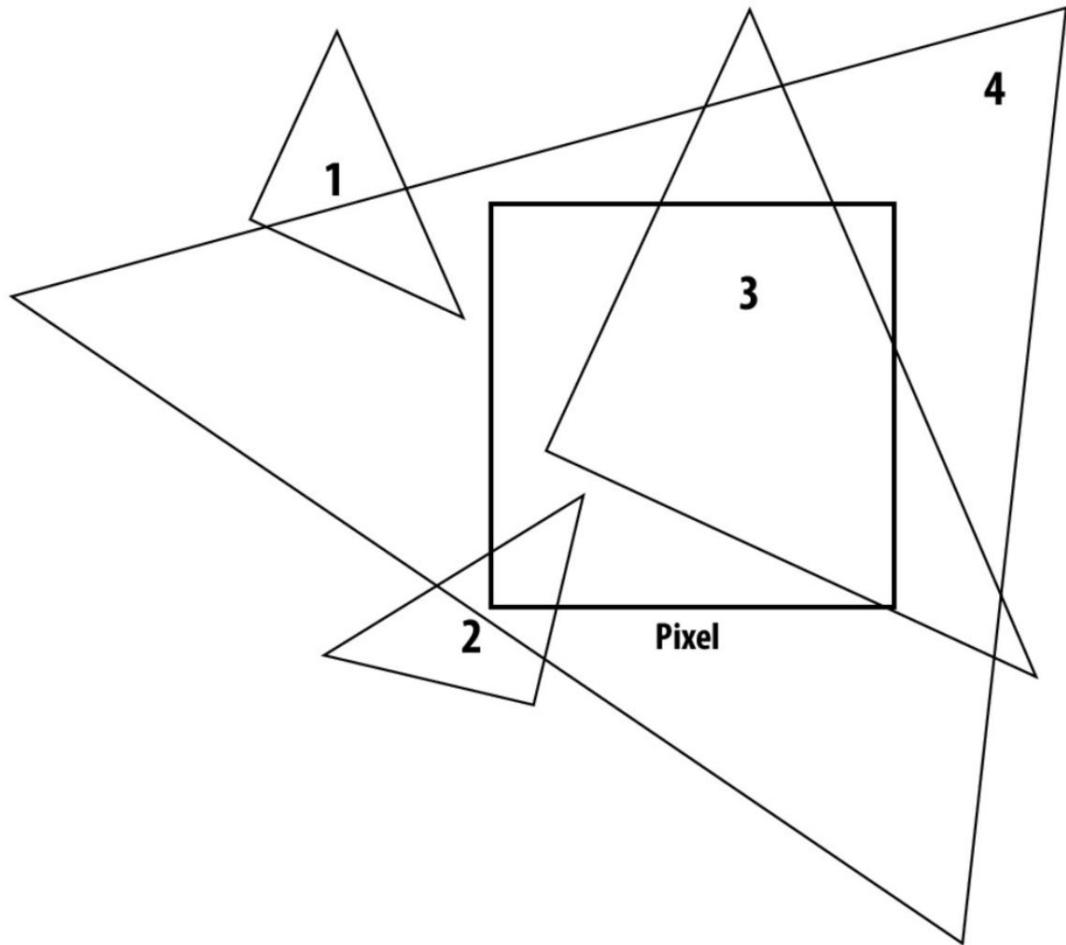
Pero...

¿Cómo determinar si un pixel es cubierto por un triángulo?

¿Qué hacer en caso de que un pixel no esté en los casos extremos (0% o 100% de cobertura)?

¿Cómo realizar ese proceso de manera eficiente? Recuerden que hay que rasterizar cada triángulo que esté en la pantalla.

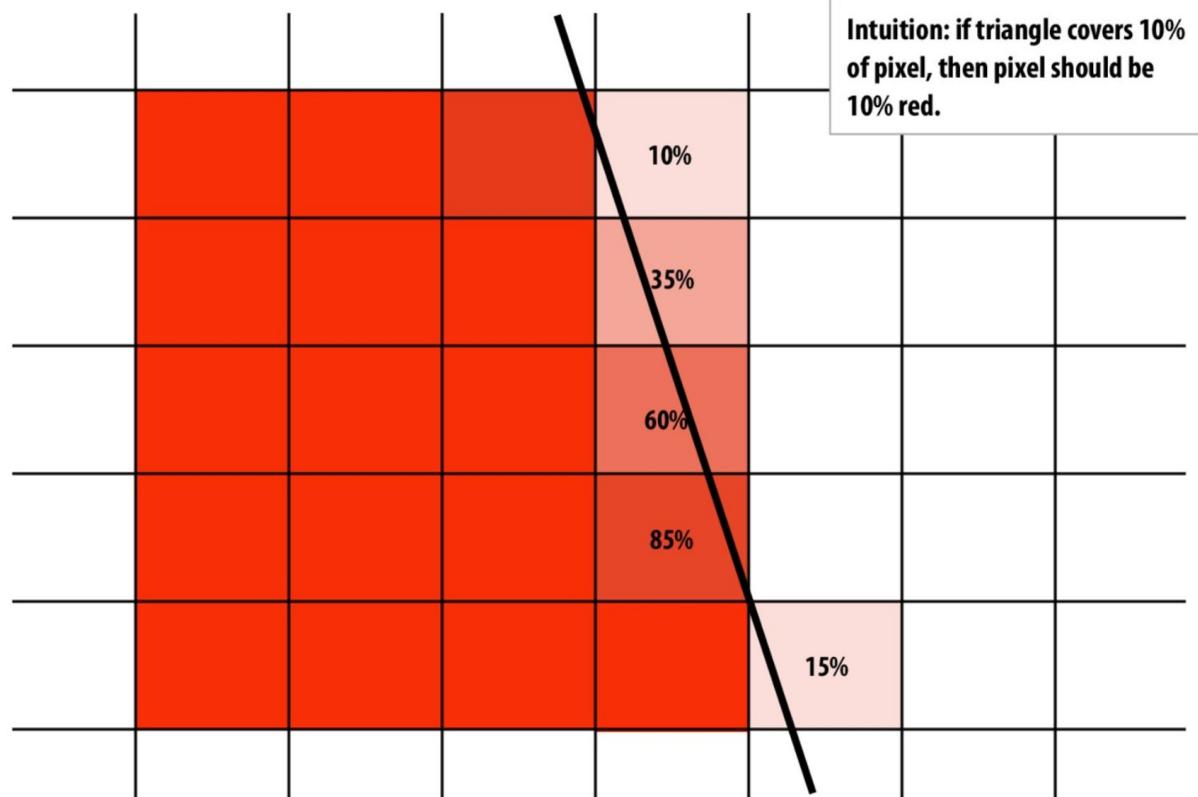
¿Cuáles de esos triángulos cubren al pixel?



Podríamos pensar en hacer un cálculo como éste y decidir el color del pixel en función del porcentaje de cobertura.

Pero, ¿cómo se calcularía? No sería muy costoso?

A fin de cuentas, estaríamos calculando una intersección de polígonos por cada pixel asociado a cada primitiva.

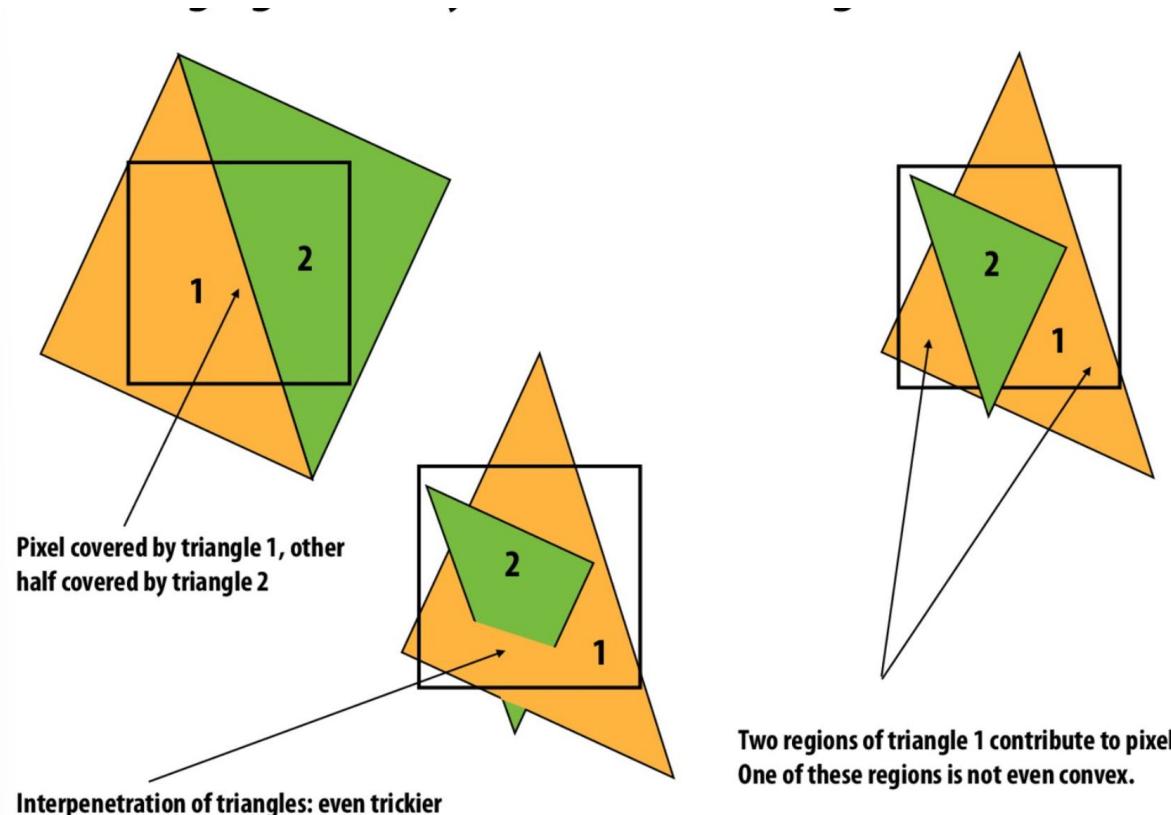


Estamos graficando escenas en 3D. Eso significa que la cantidad de configuraciones en las cuales un mismo píxel va a estar cubierto de manera simultánea por varios triángulos en múltiples configuraciones.

Habrá problemas de oclusión, transparencia, cortes no convexos, de costo.

¿Cómo lidiar con ello?

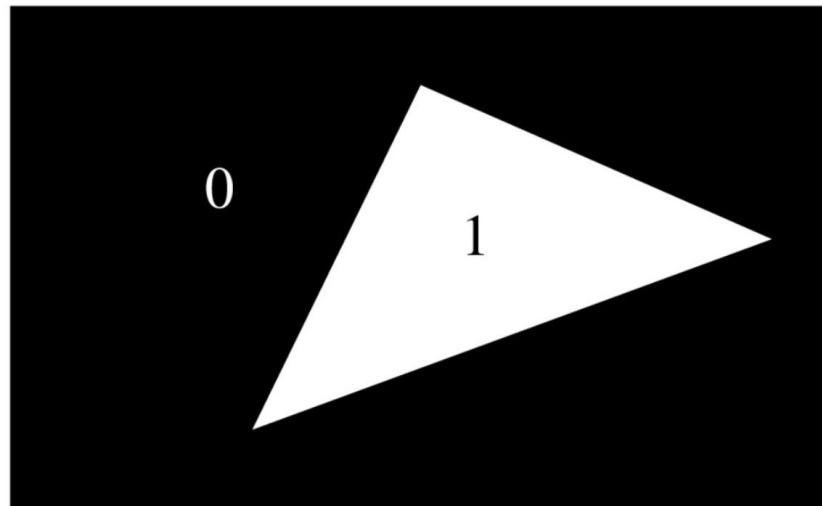
Como ven, **el problema no es trivial**.



¿Cómo resolverlo?

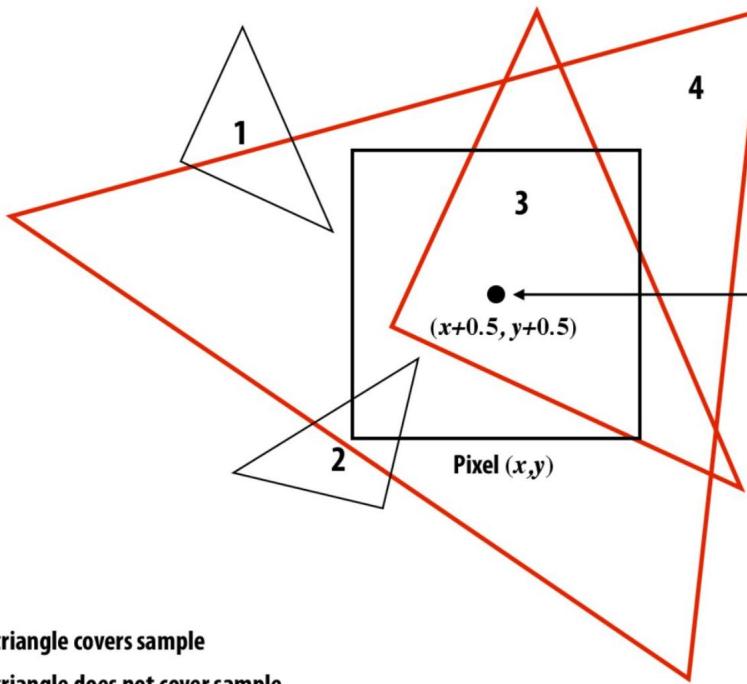
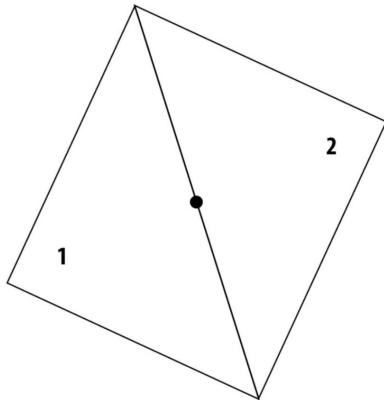
Con muestreo (**sampling**).

$$\text{coverage}(x, y) := \begin{cases} 1, & \text{triangle contains point } (x, y) \\ 0, & \text{otherwise} \end{cases}$$



Una manera simple de hacerlo es **considerar el punto central del pixel y ver cuáles triángulos lo contienen.**

Por supuesto, habrá casos de borde:



Example:
Here I chose the coverage sample point to be at a point corresponding to the pixel center.

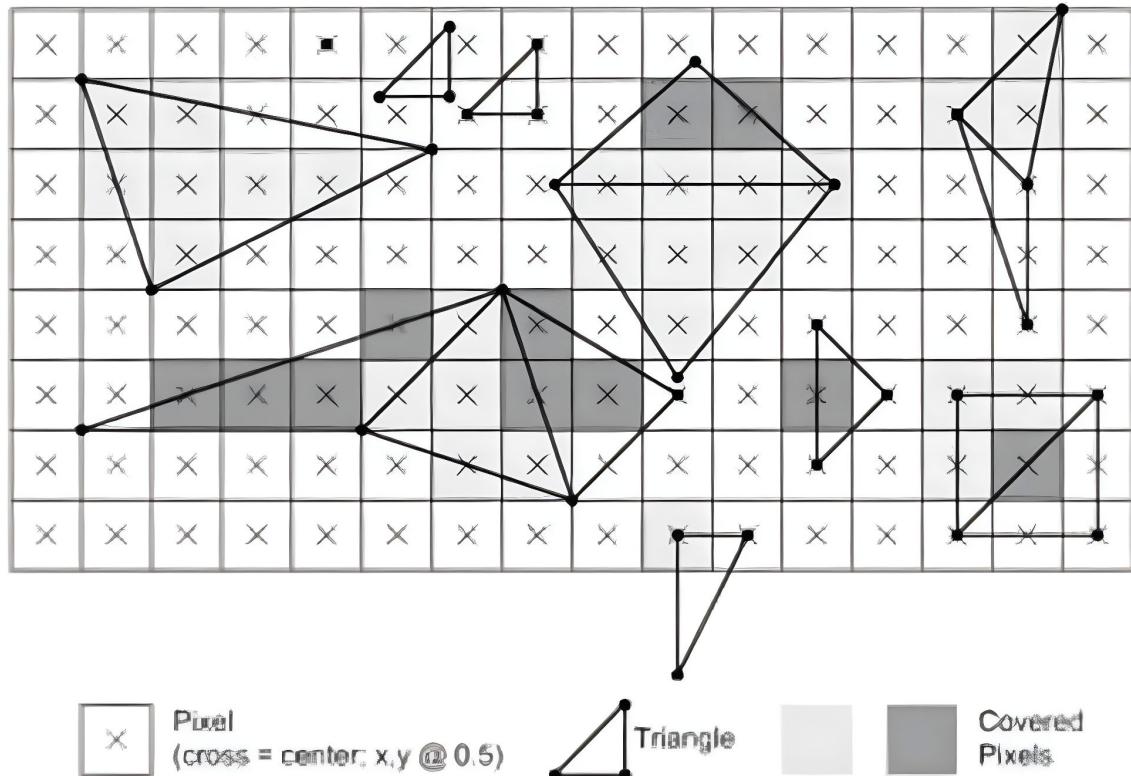
Que el centro de un pixel coincida con una arista de triángulo es muy común.

Se decide a través del tipo de arista:

Arista superior: es una arista horizontal que está sobre las demás del triángulo.

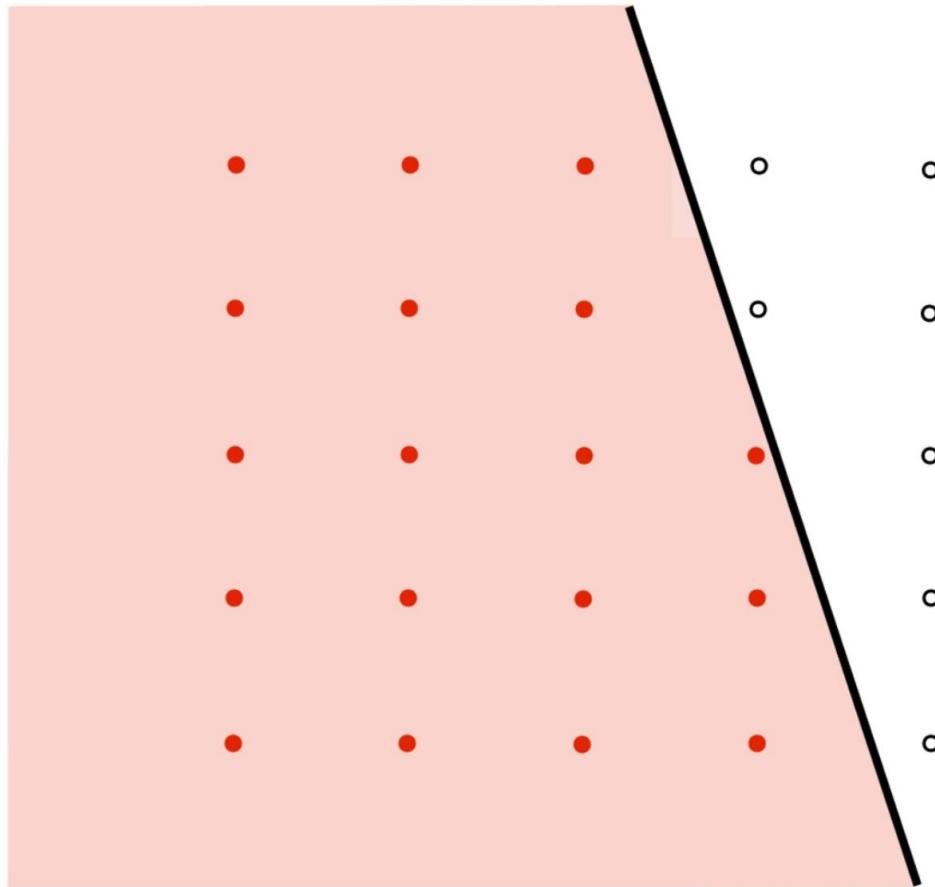
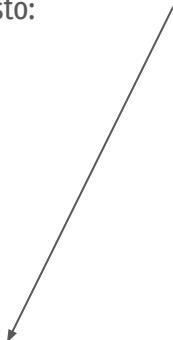
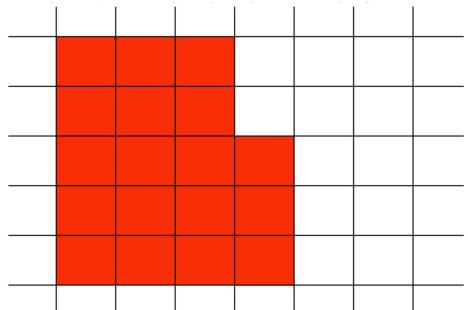
Arista izquierda: no es horizontal y está en el lado izquierdo del triángulo.

Solo en esos casos se considera que el pixel está cubierto por el triángulo.

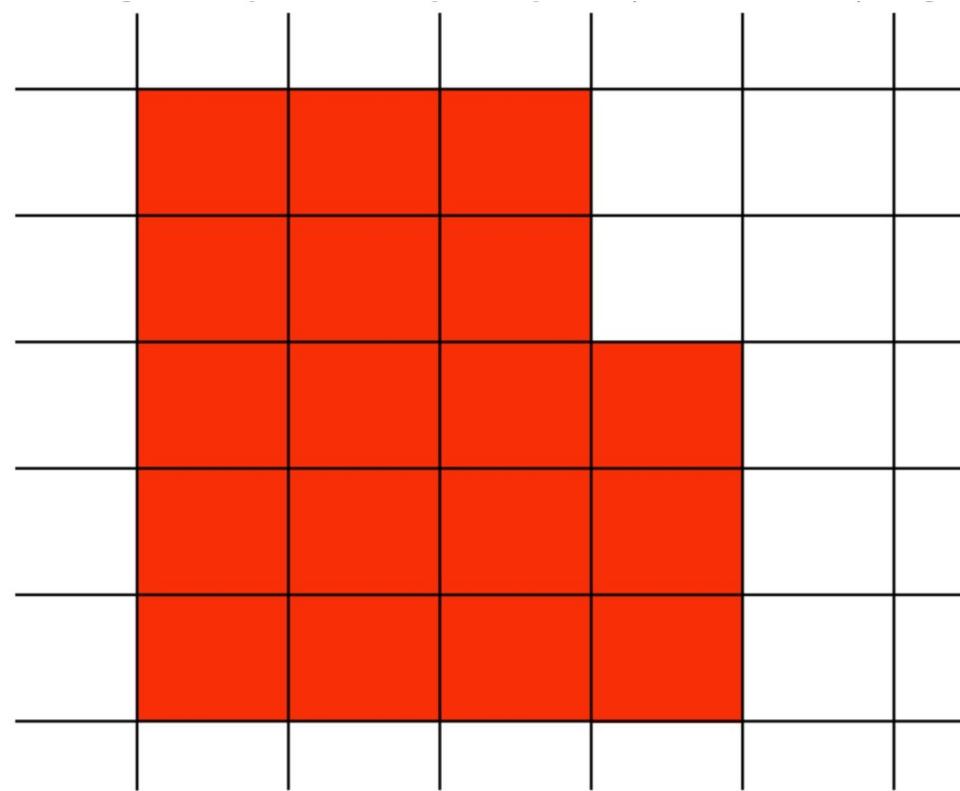


Así se vería la situación que vimos antes.

Cada uno de estos puntos es enviado a la pantalla para que la pantalla emita la luz correspondiente. Deberíamos ver esto:

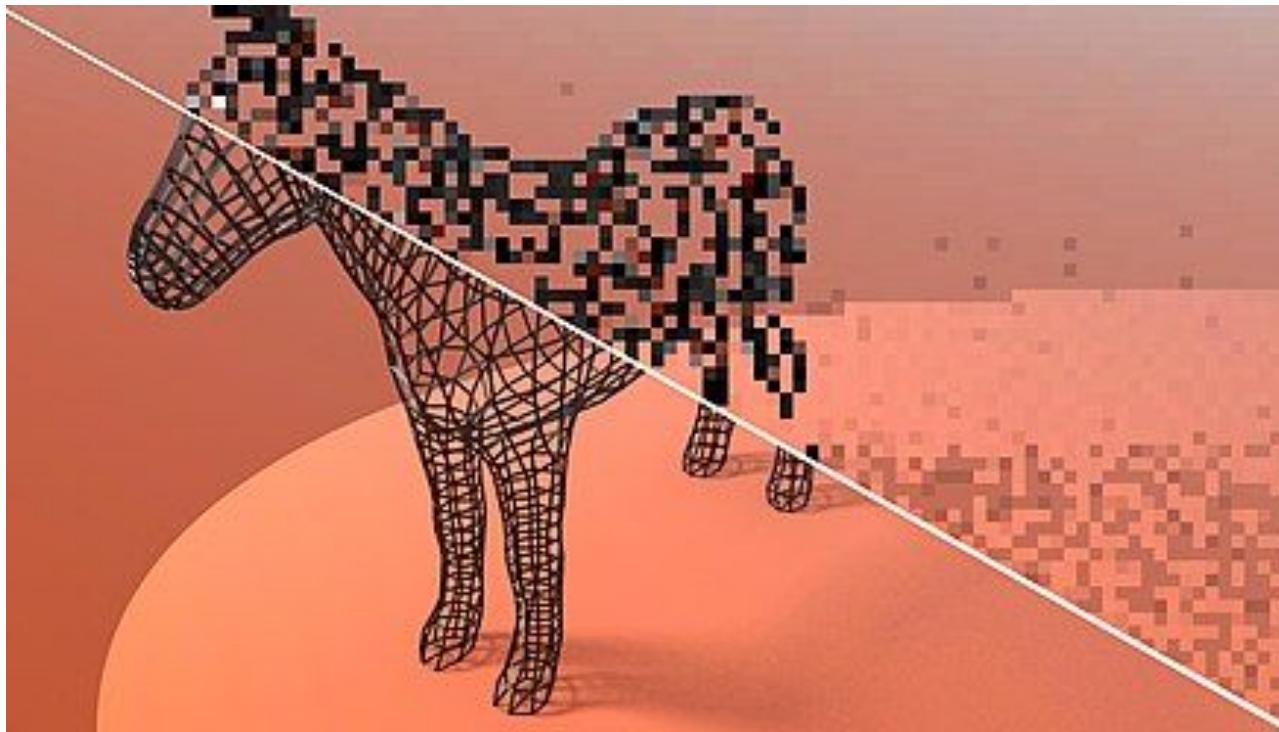


Aliasing: pérdida en la reconstrucción de una señal (de alta a baja frecuencia)



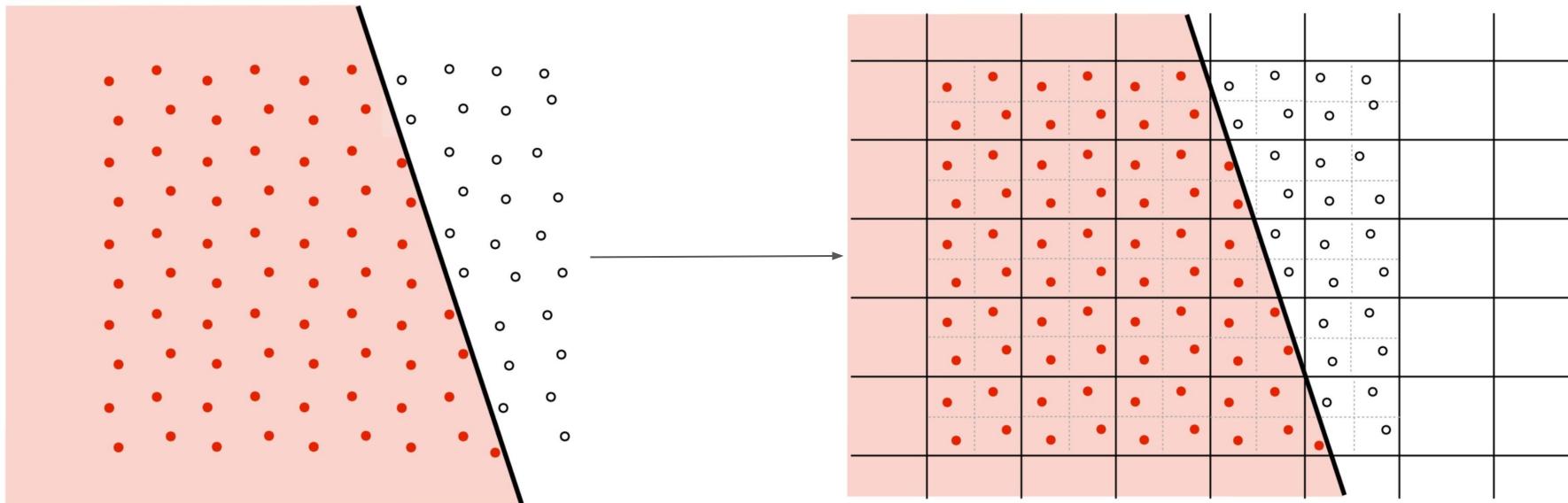


¿Es un asunto de resolución?



¡Podemos mejorar el resultado usando la misma resolución!

Lo que debemos hacer es hacer un **super muestreo** de nuestra señal (en este caso, la escena).



Este proceso es conocido como **antialiasing**.

Como tenemos más valores por pixel,
necesitamos promediar sus colores.



Pixel with sampling positions



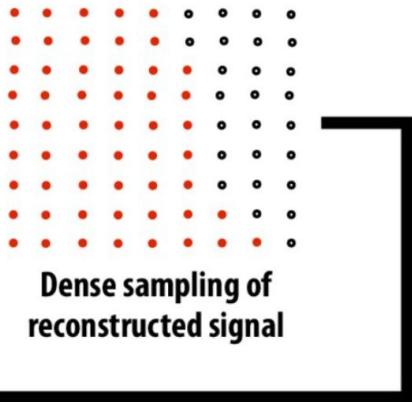
Sampled colours



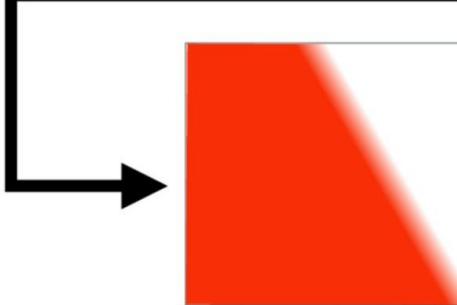
Average = displayed colour



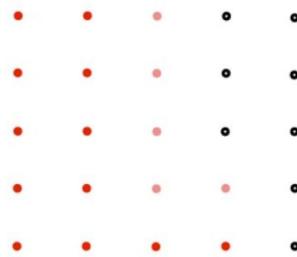
Original signal
(high frequency edge)



Dense sampling of
reconstructed signal



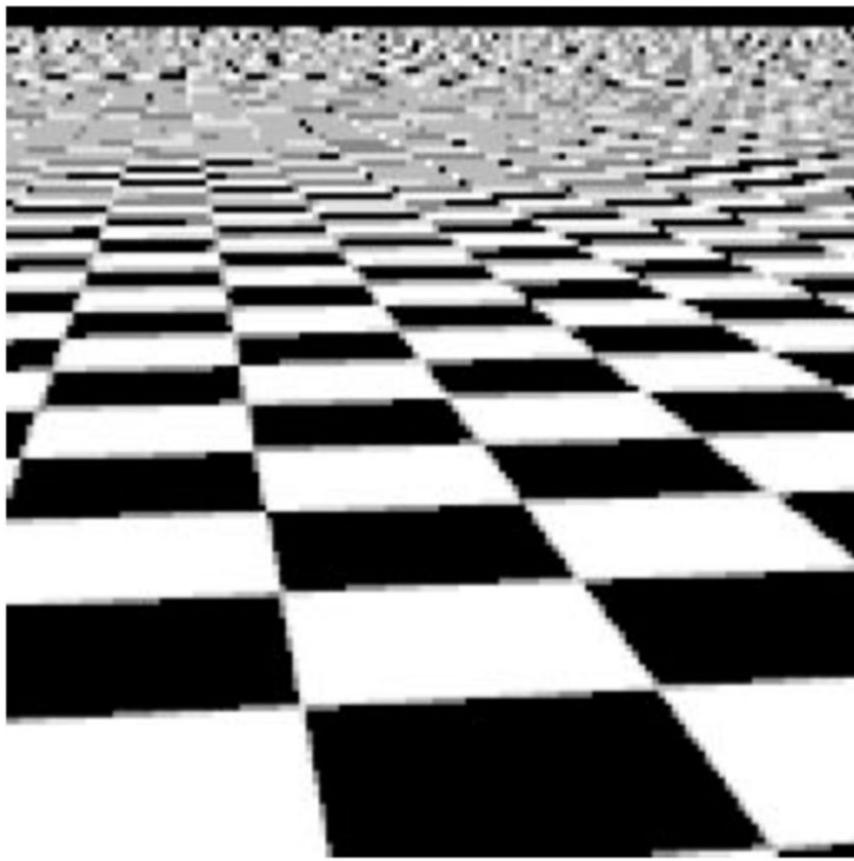
Reconstructed signal
(lacks high frequencies)



Coarsely sampled signal



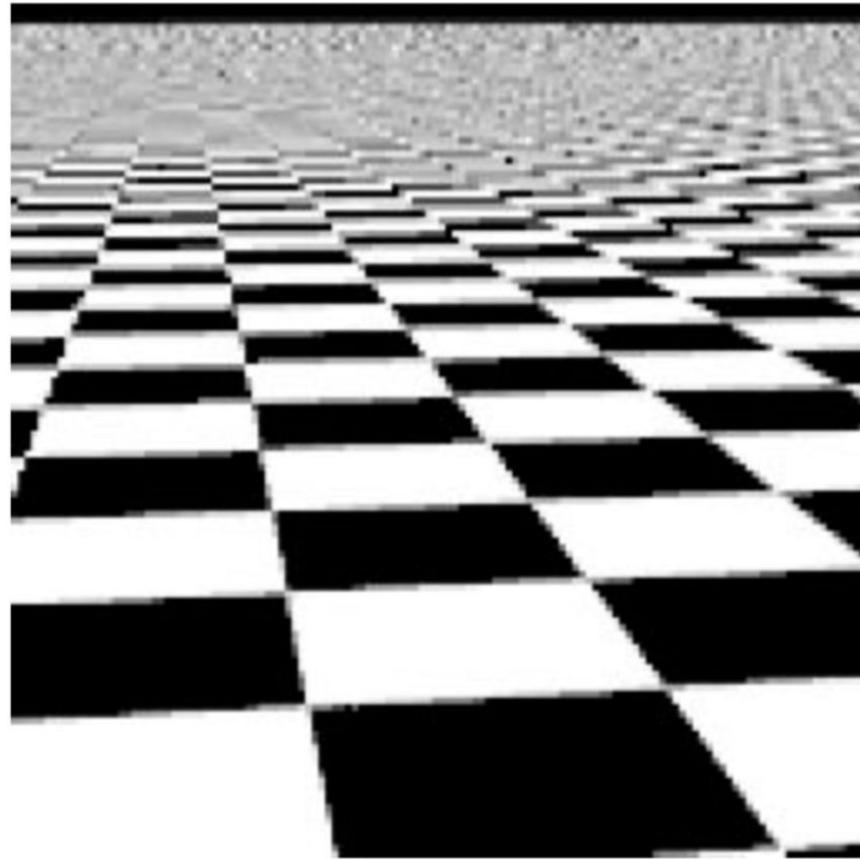
single sampling



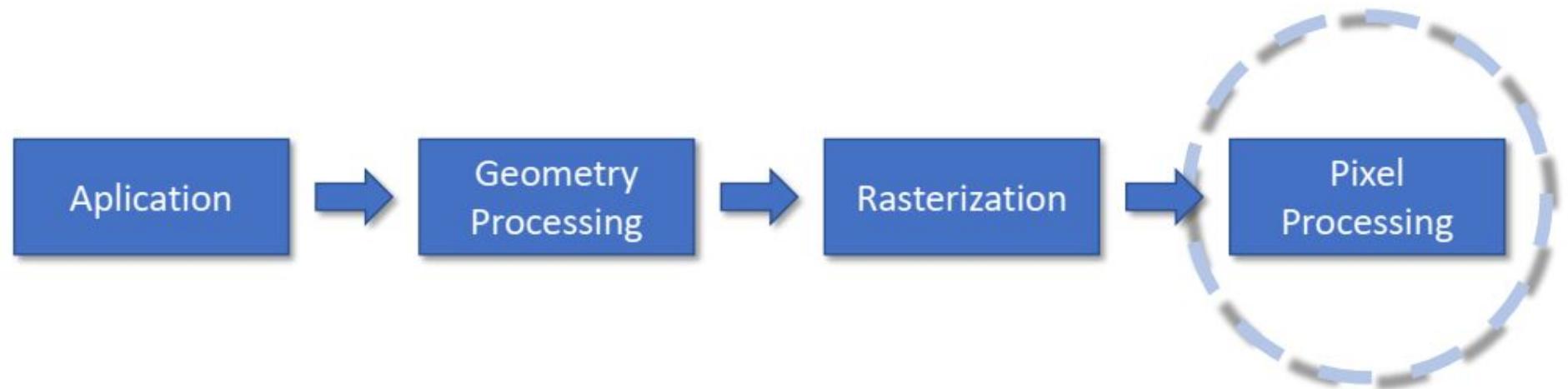
2x2 supersampling



single sampling



4x4 supersampling



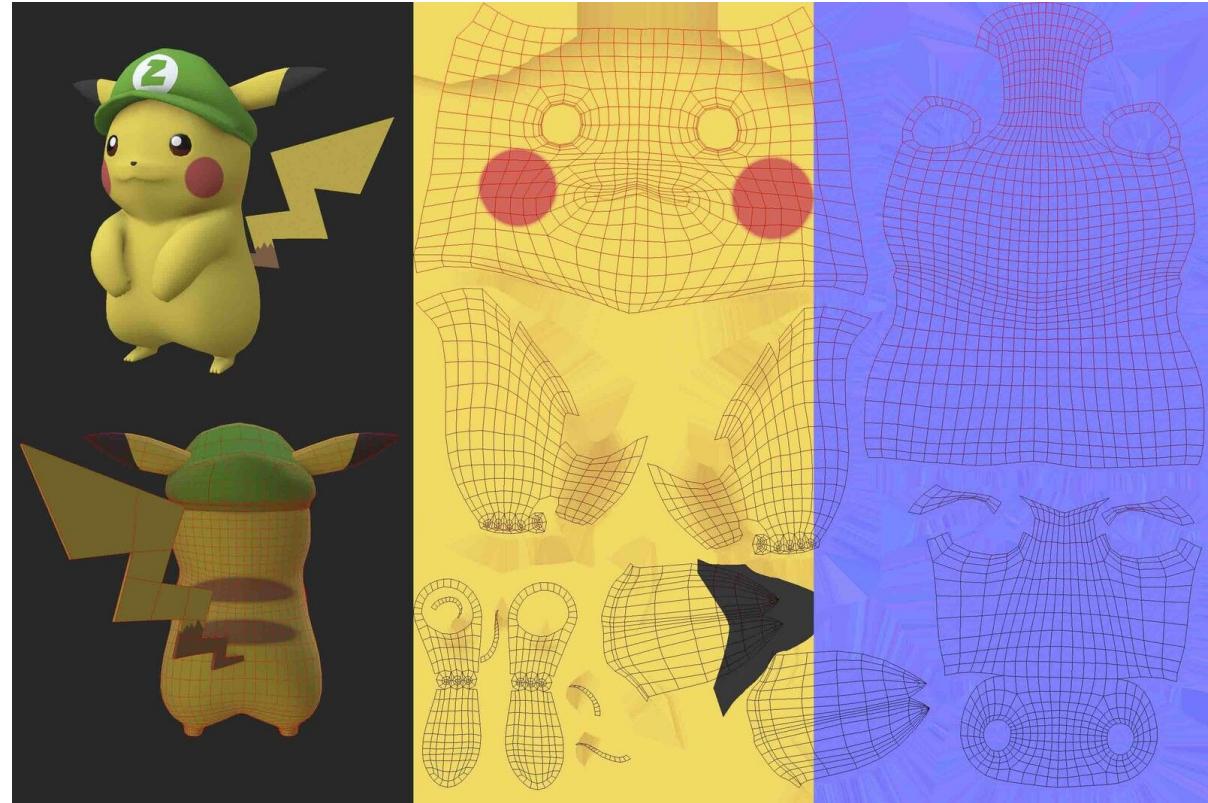
Pixel Processing

Para cada elemento de nuestra escena tenemos un conjunto de píxeles. Lo siguiente es decidir el color que le corresponde a cada uno.

Conocemos los atributos de los vértices: color, textura, normal, material, etc.

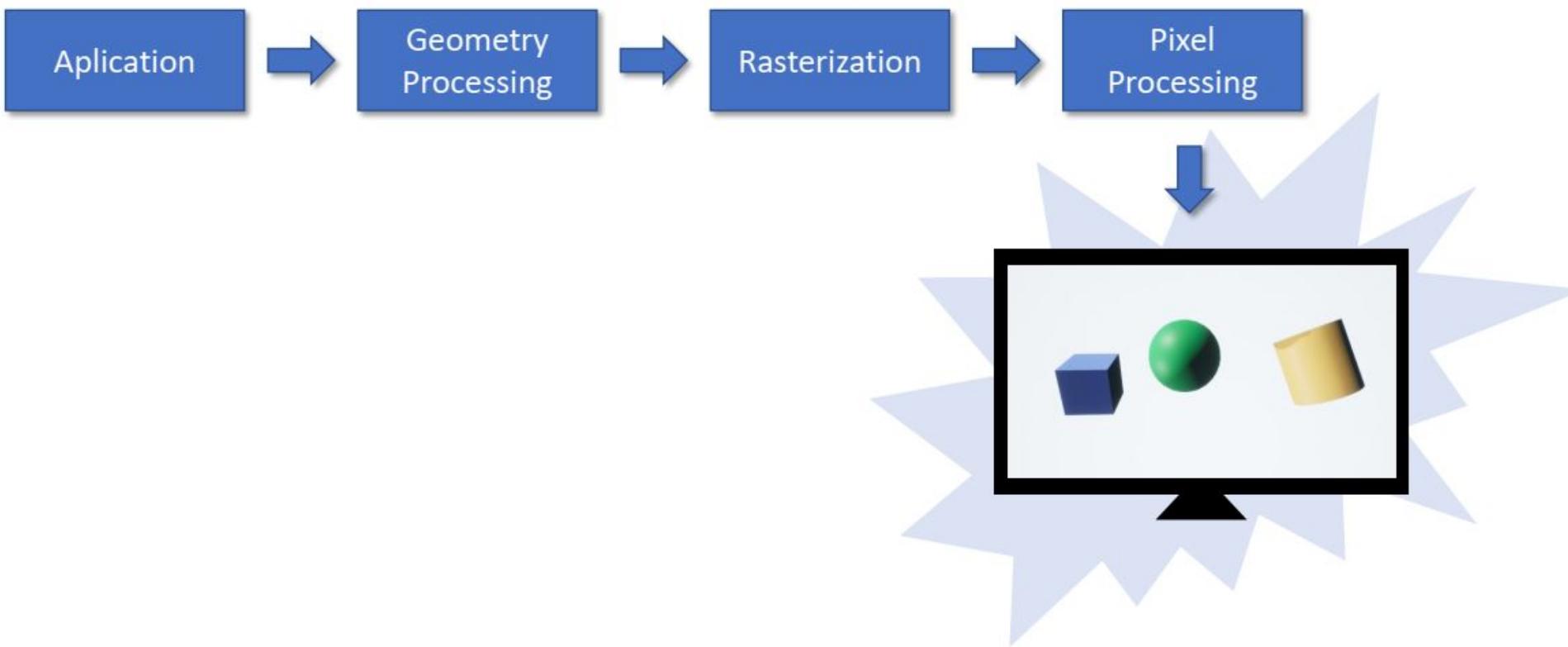
Tenemos una coordenada asociada a cada pixel y podemos interpolarla.

Como resultado, podemos utilizar un programa llamado **fragment shader** que determina el color final del pixel considerando la información del triángulo o incluso información adicional en forma de **mapas de textura**.



pd=aunque este Pikachu tenga rectángulos, **al hacer rendering se convierte todo a triángulos.**

<https://www.artstation.com/artwork/zAkxJ6>



Esto es una mirada general y simplificada

A lo largo del curso veremos cada una de estas etapas por separado.

Las veremos **en detalle** y experimentaremos con su **implementación**.

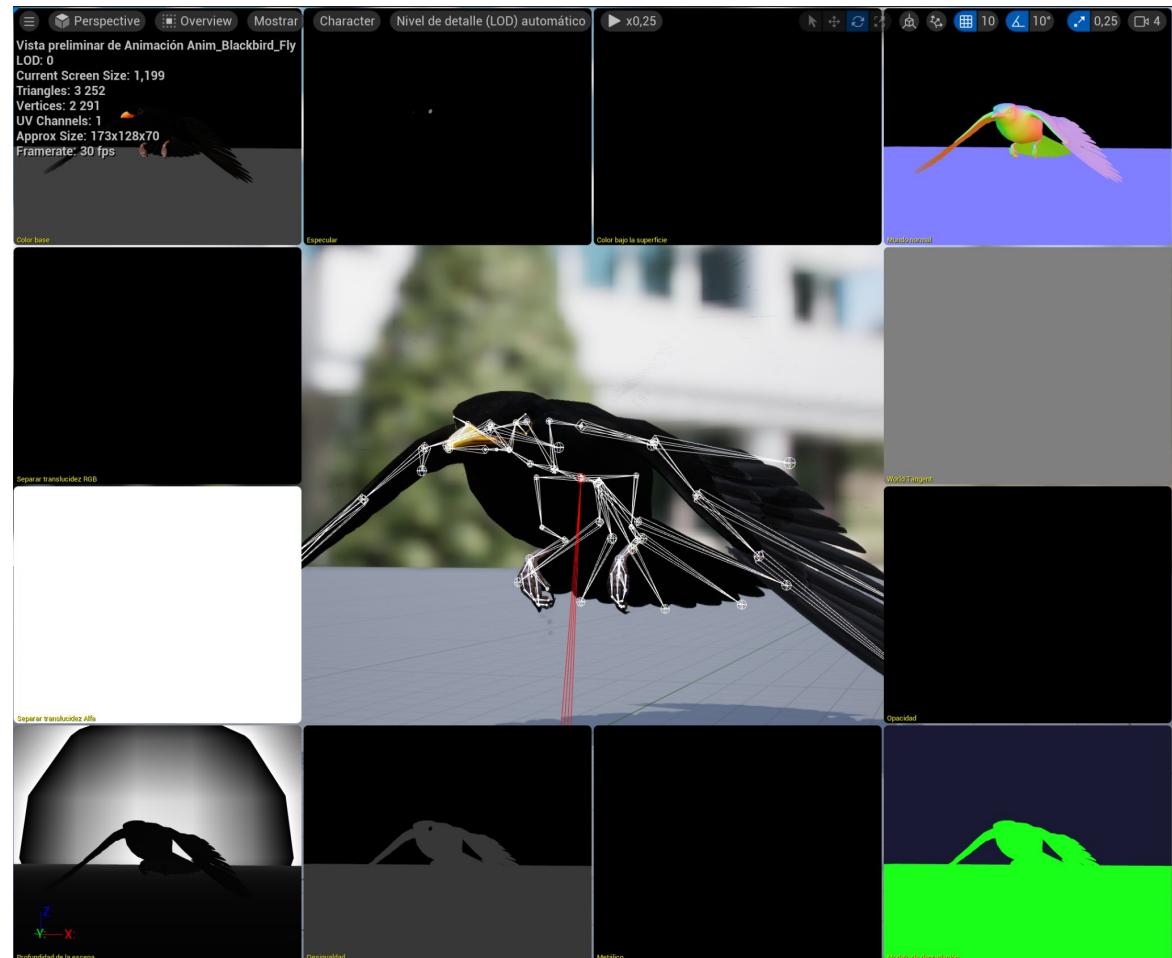
Importante: el proceso de rendering suele ser el mismo siempre. Como desarrolladores/as, usualmente nos encargamos de configurar el mundo, sus reglas y su apariencia. Un motor gráfico se encarga del rendering.

Pregunta propuesta: ¿cómo evaluar de manera eficiente si un punto (pixel) es cubierto por un triángulo? ¿Cuáles son los casos donde esa operación es costosa?

Motor: Unreal Engine 5

Disponible desde este año. Pueden descargarlo y probar distintas maneras de visualizar el contenido de una escena 3D.

<https://www.unrealengine.com/en-US>



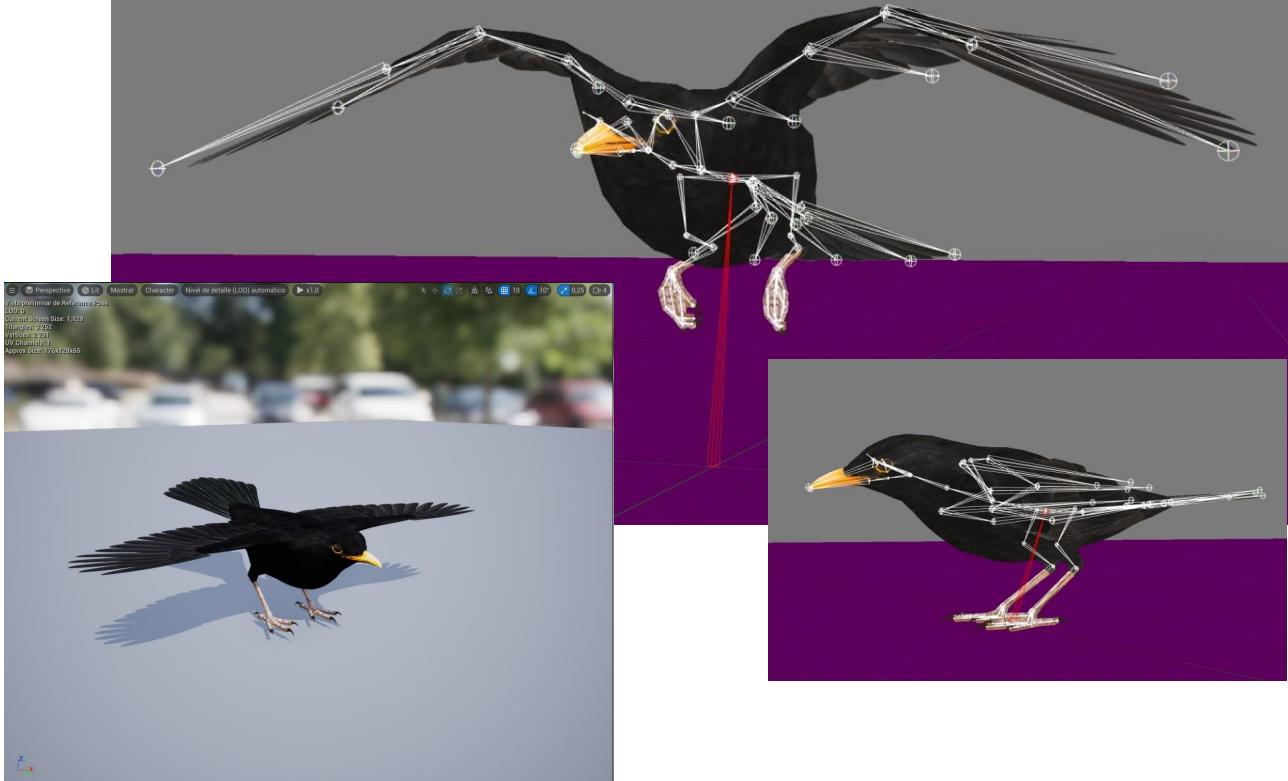
Rendering de un Pajarito

Aquí vemos el modelo 3D de un zorzal negro (*turdus merula*). Tiene animación provista por el esqueleto asociado al modelo 3D.

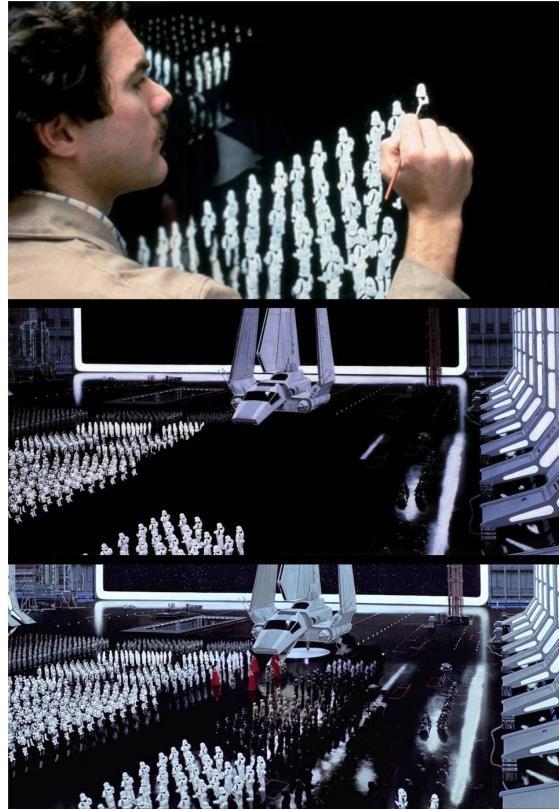
El modelo tiene una **pose de referencia** y sus vértices son modificados de acuerdo a la animación que tenga el esqueleto.

Eso se llama “**skinning**” y es una técnica común utilizada en videojuegos y animación.

Veremos distintas demostraciones con el pajarito a lo largo del curso.



Rendering Hoy



THE VIRTUAL PRODUCTION OF

THE

MANDALORIAN





MATRIX AWAKENS

AN **UNREAL ENGINE 5**
EXPERIENCE



¿Preguntas?