Esplorare L'API Grafica Vulkan

Emanuele Franchi

Vulkan

- API grafica sotto forma di specifica
- Non ne esiste un'unica implementazione
- Implementata attraverso il driver della propria scheda grafica
- Sviluppata usando come modello l'architettura delle GPU odierne
- API di basso livello
- Richiede un certo know how da parte del programmatore
- Se il programmatore la utilizza in modo coscienzioso, può risultare in performance migliori rispetto alle API di vecchia generazione
- Multithreaded first
- Rilasciata nel 2016: molte GPU ancora in circolazione non la supportano

Inizializzare Vulkan

- Creare una Vulkan instance
- Creare una finestra usando l'API del sistema operativo
- Creare una presentation surface per collegare la finestra alla Vulkan instance
- Selezionare una GPU, dedicata o integrata
- Creare un device logico per interfacciarsi con la GPU selezionata
- Creare una swapchain per gestire la presentazione d'immagini sulla presentation surface

Renderizzare Un Colore: Demo

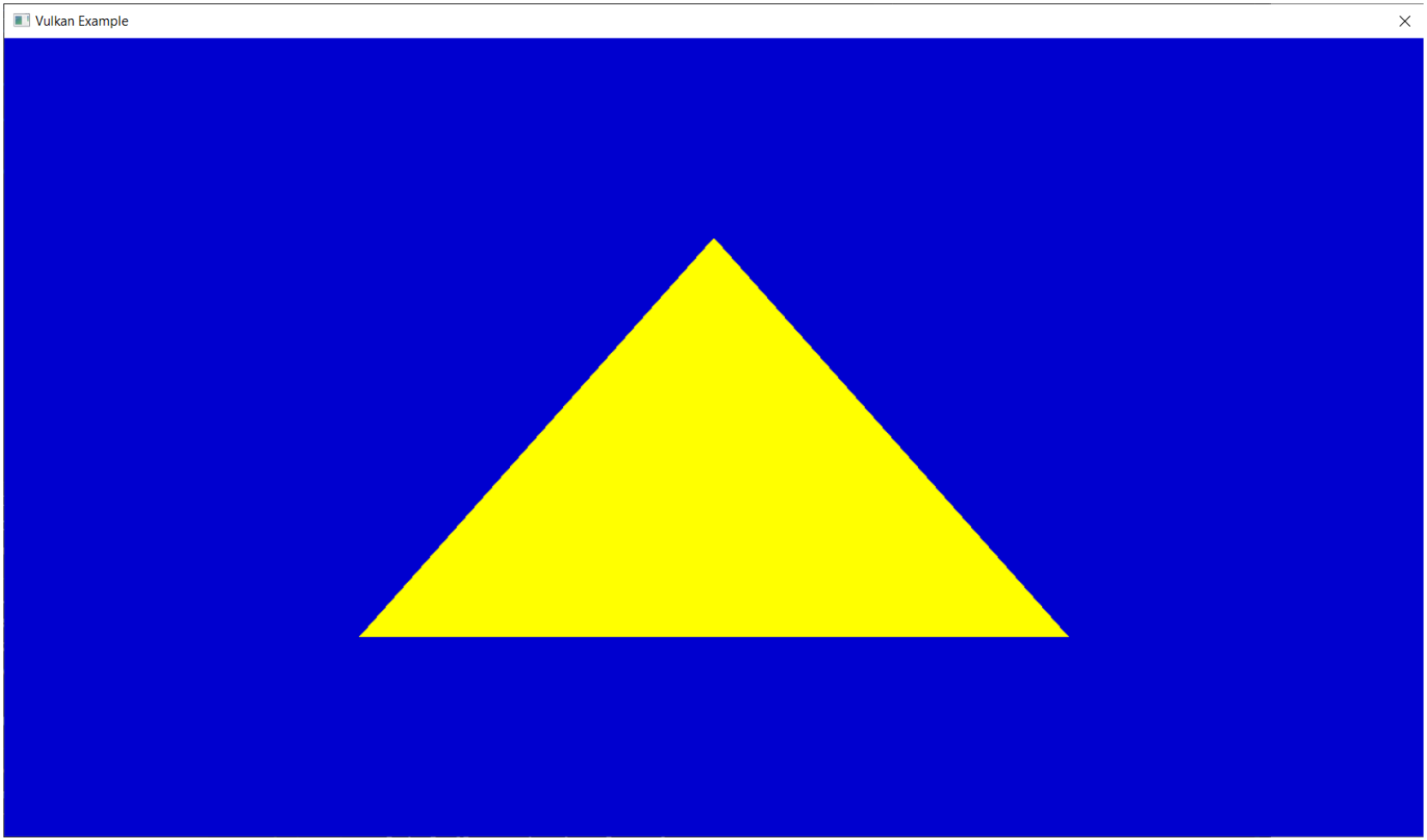
Renderizzare Un Colore: Setup

- Creare un command buffer su cui scrivere i comandi grafici da eseguire
- Creare un render pass
- Un render pass descrive gli attachment che vengono utilizzati durante il rendering
- Un render pass raggruppa i comandi grafici in uno o più subpass in base a come e quali attachment questi utilizzano
- Usiamo un solo attachment
- Abbiamo un solo subpass durante il quale l'attachment viene usato come color render target

Renderizzare Un Colore: Main Loop

- Ottenere la prossima immagine della swapchain sulla quale renderizzare
- Aspettare che i comandi registrati nel command buffer abbiano terminato l'esecuzione
- Creare un framebuffer che contiene l'immagine della swapchain
- Un framebuffer raccoglie tutti gli attachment che vengono utilizzati durante un'istanza di un render pass
- Scrivere sul command buffer due comandi: uno per iniziare il render pass, un'altro per terminarlo
- Quando iniziamo il render pass, specifichiamo il clear color per l'immagine
- Inviamo il command buffer alla GPU, cosicché questa esegua i comandi registrati
- Inviamo un comando di presentazione alla GPU per presentare l'immagine renderizzata

Renderizzare Un Triangolo: Demo



Renderizzare Un Triangolo: Vertex Shader

```
#version 450
#extension GL_KHR_vulkan_glsl : enable

vec2 positions[3] = vec2[]
(
    vec2(+0.0, -0.5),
    vec2(+0.5, +0.5),
    vec2(-0.5, +0.5),
);

void main()
{
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
}
```

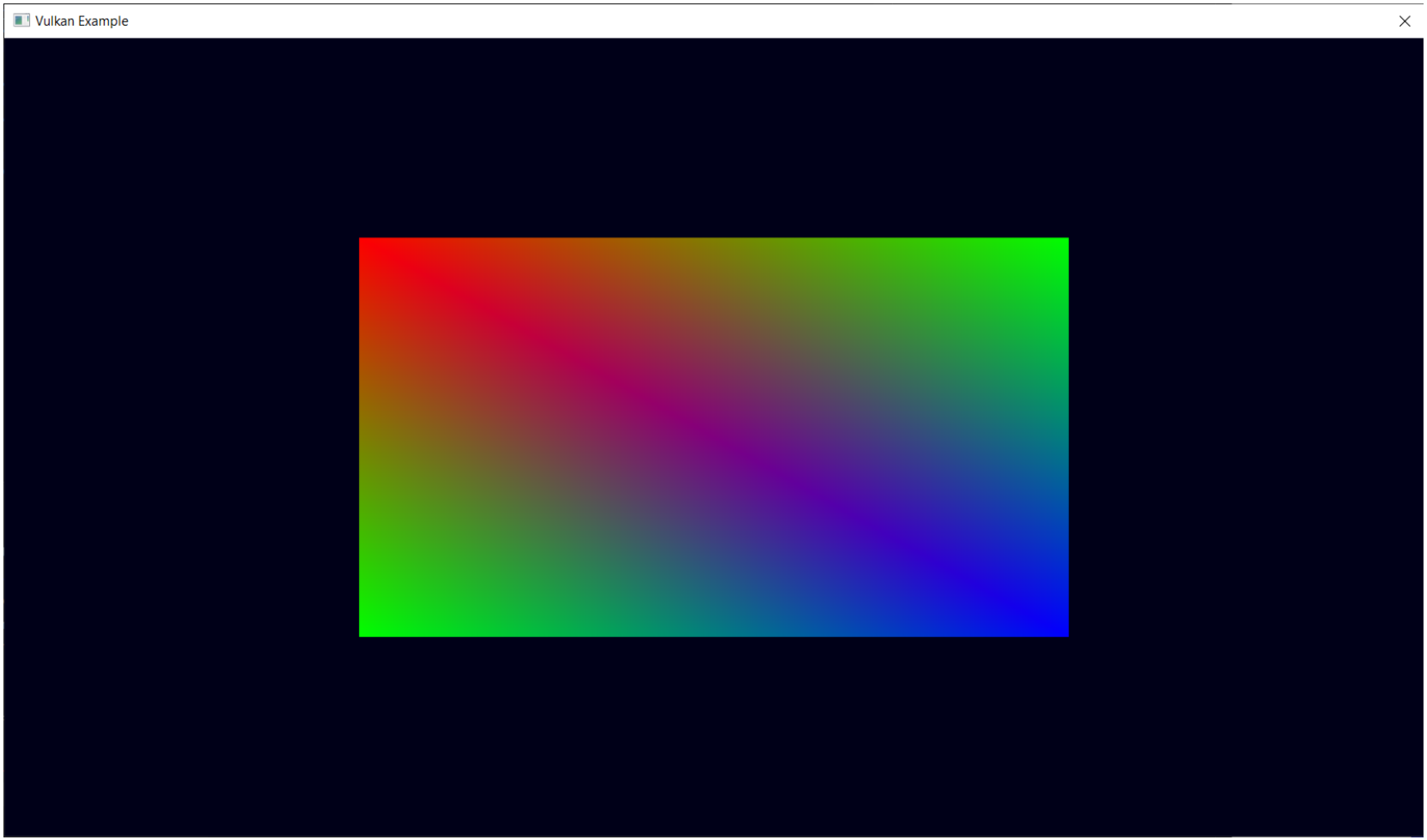
Renderizzare Un Triangolo: Fragment Shader

```
#version 450

layout(location = 0) out vec4 outColor;

void main()
{
    outColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```

Renderizzare Un Triangolo

- Usiamo un pipeline state object per descrivere l'intero stato della pipeline grafica
- Un pipeline state object descrive anche quali shader utilizzare
- Sul command buffer, scriviamo un comando che indica alla GPU di configurare lo stato della pipeline grafica usando il pipeline state object da noi creato
- Sempre sul command buffer, scriviamo un comando che attiva la pipeline grafica per renderizzare tre vertici

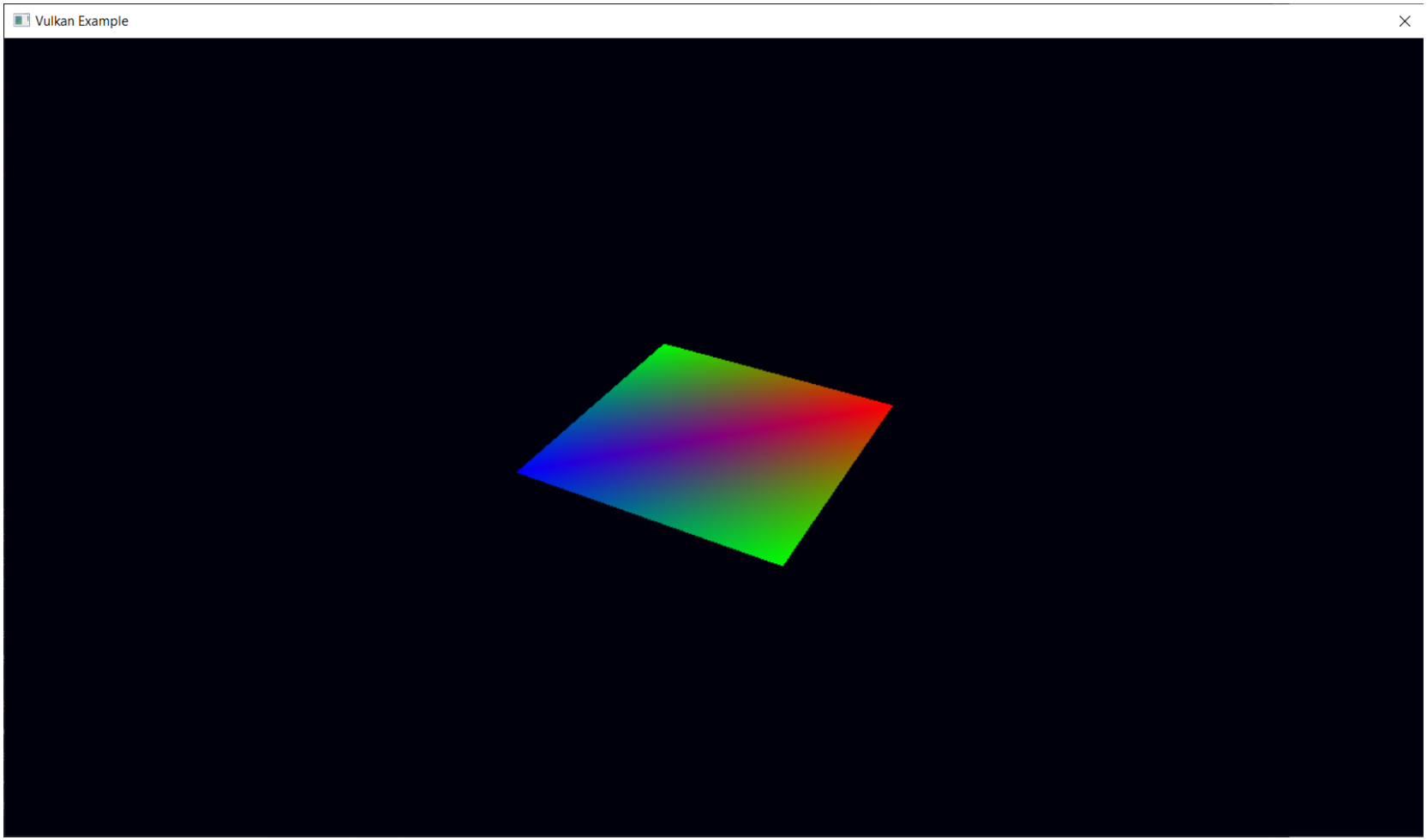
Vertex Buffer: Demo



Vertex Buffer

- I dati dei nostri vertici sono in RAM e noi dobbiamo caricarli nella memoria della GPU
- Usiamo due buffer
- Uno staging buffer, allocato su memoria della GPU host visible
- Scriviamo i dati dei nostri vertici su questo buffer
- Un vertex buffer, allocato su memoria della GPU device local
- Inviamo un comando che dice alla GPU di trasferire il contenuto dello staging buffer nel vertex buffer
- Modifichiamo la creazione del nostro pipeline state object per specificare come interpretare i dati nel vertex buffer
- Prima di registrare il comando per attivare la pipeline, registriamo un comando che indica alla pipeline grafica di usare come input il nostro vertex buffer

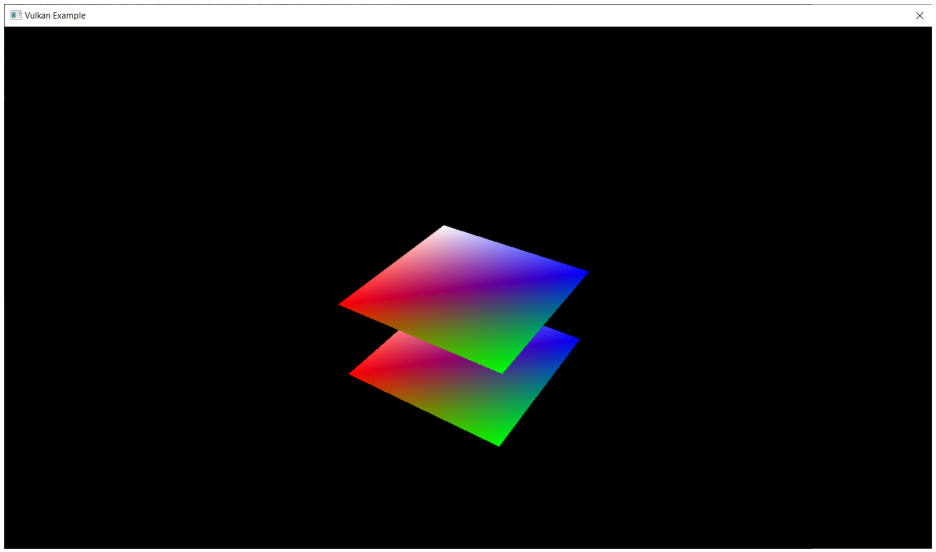
Uniform Buffer: Demo

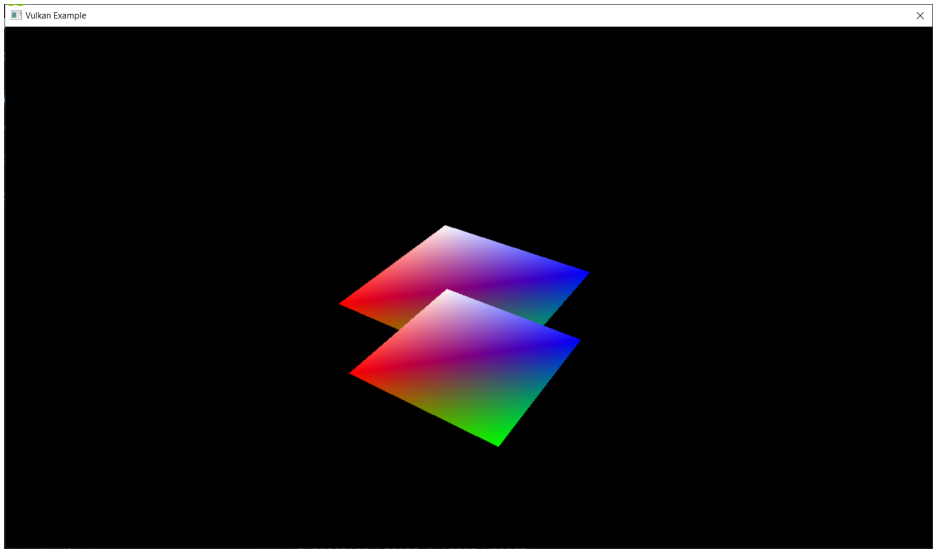


Uniform Buffer

- Passiamo variabili globali agli shader usando uno uniform buffer
- Siccome tali variabili cambiano frequentemente, allochiamo uno uniform buffer su memoria della GPU host visible
- Informiamo di questo fatto la pipeline, usando un pipeline layout object
- Tale oggetto descrive quali descriptor (risorse) sono globalmente accessibili in quali shader
- Creiamo un descriptor set layout che descrive il numero e i tipi di descriptor
- Allochiamo un descriptor set basandoci sul descriptor set layout
- Una volta allocato, un descriptor set va popolato prima di essere utilizzato
- Prima di scrivere il comando per attivare la pipeline, scriviamo nel command buffer un comando per informare la GPU che stiamo usando un certo descriptor set

Depth Testing: Demo





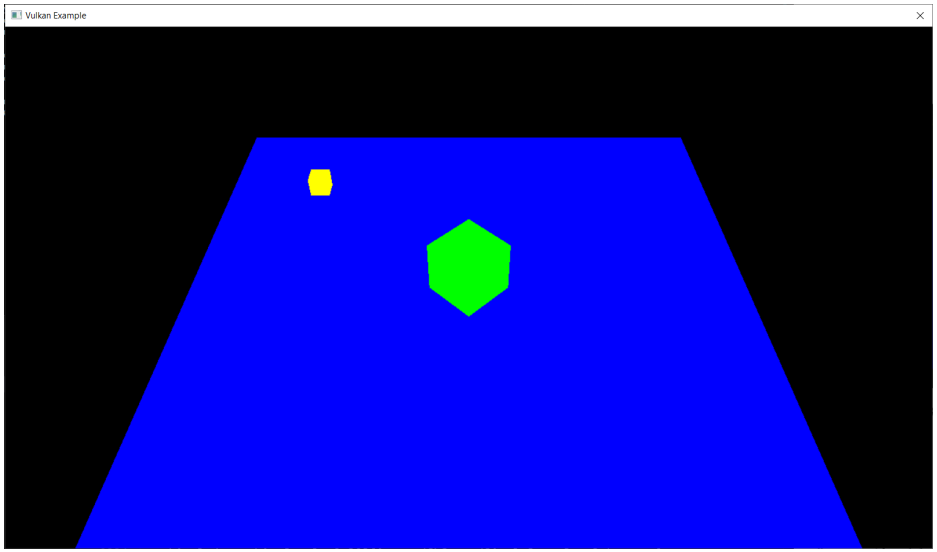
Depth Buffer: Idea

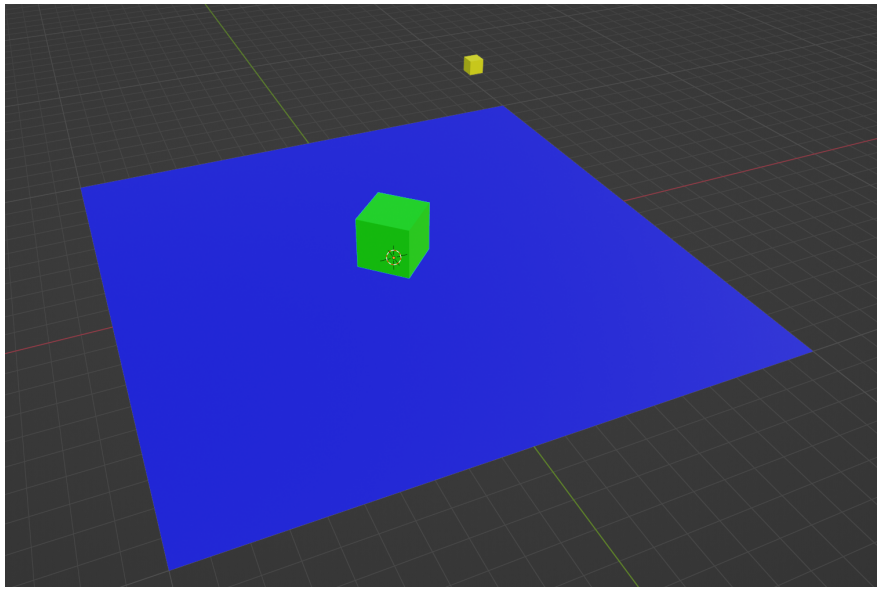
- Come renderizzare oggetti in modo tale che quelli più vicini possano nascondere quelli più lontani?
- Possiamo ordinare gli oggetti in base alla loro lontananza dalla camera
- Questa soluzione non funziona bene se due o più oggetti si sovrappongono in tutto o in parte
- Se gli oggetti da renderizzare sono opachi, allora possiamo usare un depth buffer
- Un depth buffer è un'immagine che codifica informazioni riguardanti la distanza dei frammenti dalla camera
- Quando un frammento viene generato, compariamo la profondità del frammento con quella salvata nel corrispettivo texel del depth buffer
- Se è più grande, il frammento non viene utilizzato
- Se è più piccola, il frammento viene utilizzato e la sua profondità viene salvata nel depth buffer

Depth Buffer: Implementazione

- Allochiamo un'immagine, sulla memoria della GPU, che possa essere usata come depth buffer
- Nel nostro render pass, aggiungiamo un nuovo attachment
- Questo attachment viene usato come depth stencil attachment durante il nostro subpass
- Durante la creazione del pipeline state object dobbiamo abilitare il depth testing
- Quando iniziamo un'istanza del render pass, specifichiamo un clear value di 1.0 per il depth buffer

Scena: Demo

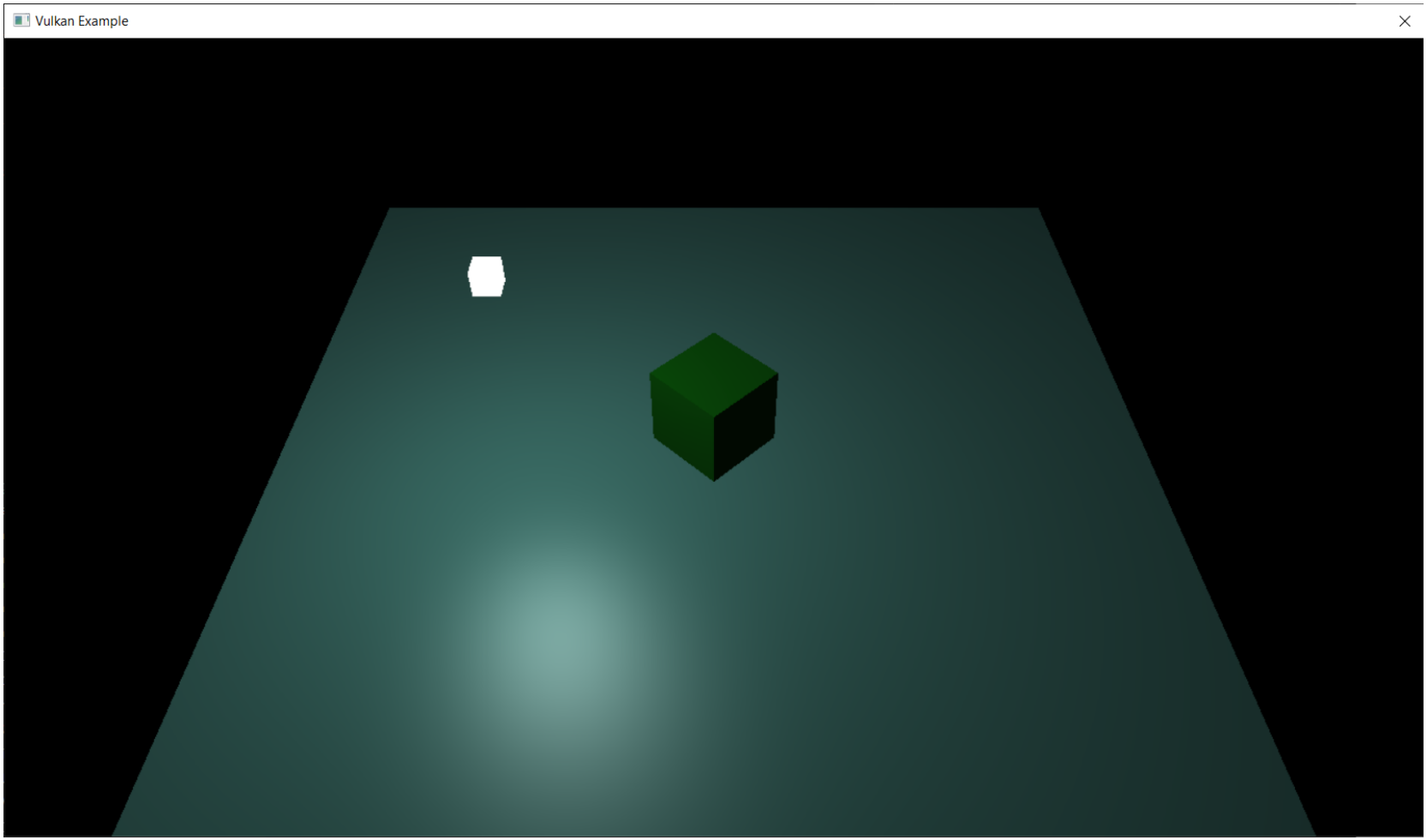




Scena

- Organizziamo gli oggetti da renderizzare in una scena
- Ogni oggetto ha una posizione, rotazione e scala (model matrix)
- Uno o più oggetti condividono un vertex buffer e un pipeline state object
- Più oggetti diversi possono usare gli stessi vertici e la stessa configurazione della pipeline
- Ogni oggetto ha un proprio uniform buffer
- Per renderizzare gli oggetti di una scena
  - Usiamo il pipeline state object dell'oggetto
  - Usiamo il vertex buffer dell'oggetto
  - Usiamo lo uniform buffer dell'oggetto
  - Attiviamo la pipeline
- Abbiamo una camera attraverso la quale guardiamo la scena (view e projection matrix)

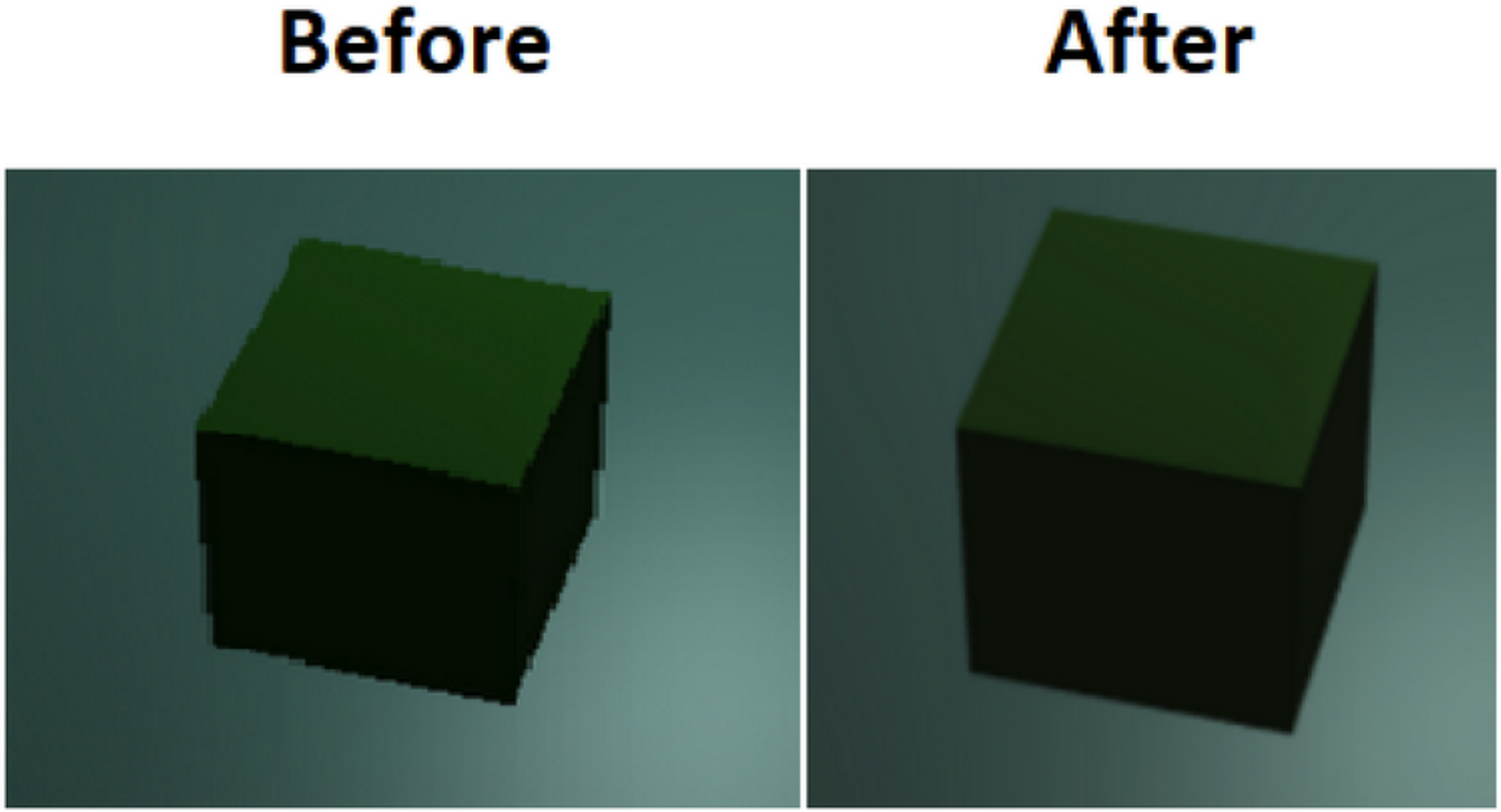
Blinn-Phong: Demo



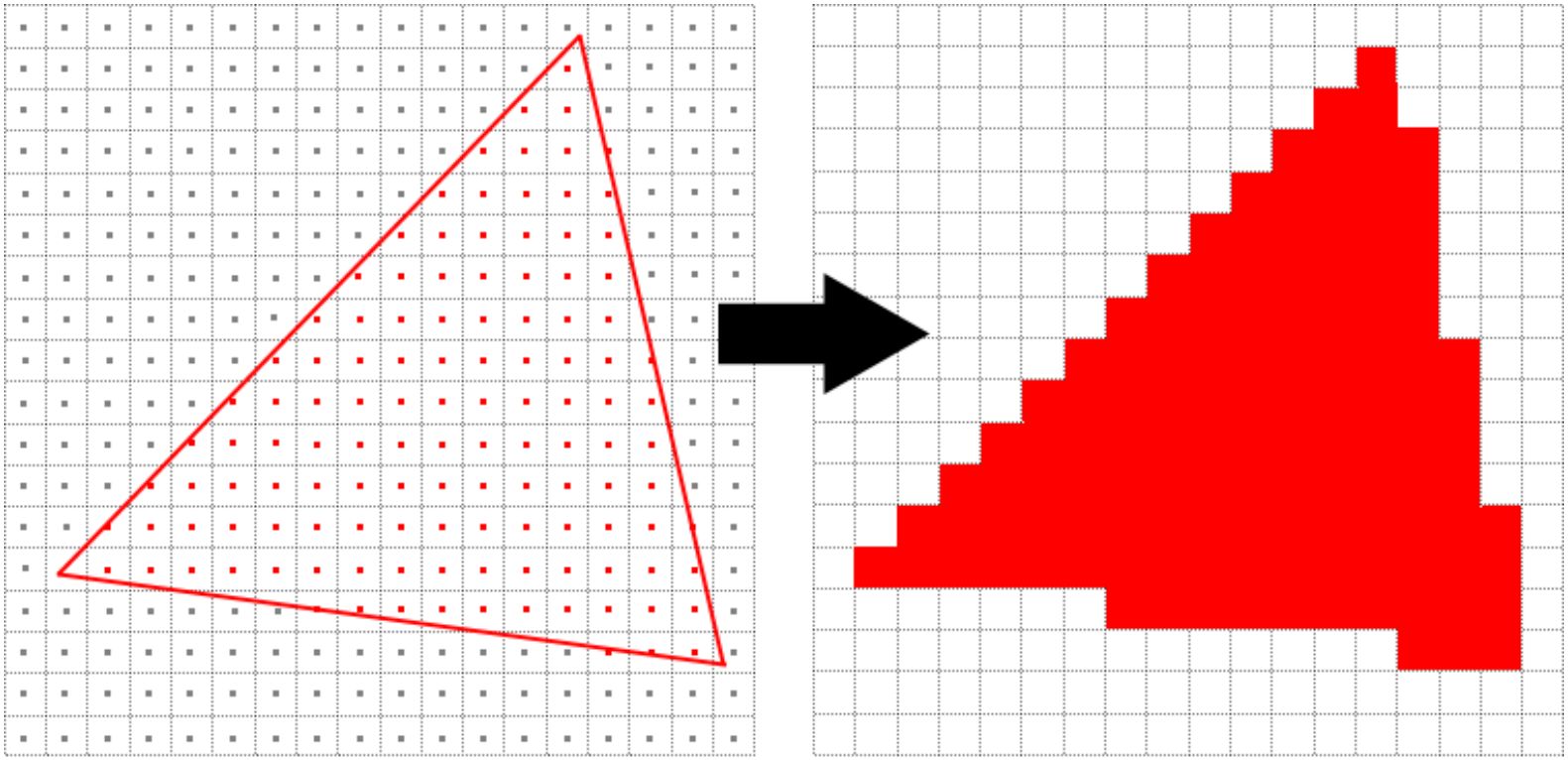
Blinn-Phong

- L'illuminazione viene divisa in tre componenti: ambient, diffuse e specular
- La componente ambientale modella il fatto che una scena non è mai totalmente buia
- La componente diffusiva simula l'impatto che la luce ha sugli oggetti opachi
- La componente speculare simula il punto luminoso che una luce causa su oggetti lucidi (specular highlight)
- Ogni oggetto che deve essere illuminato ha un materiale: componente ambientale, diffusiva e speculare
- Un materiale ha una proprietà aggiuntiva che indica quanto un oggetto è lucido
- Una luce ha diverse intensità per la componente ambientale, diffusiva e speculare

MSAA: Demo

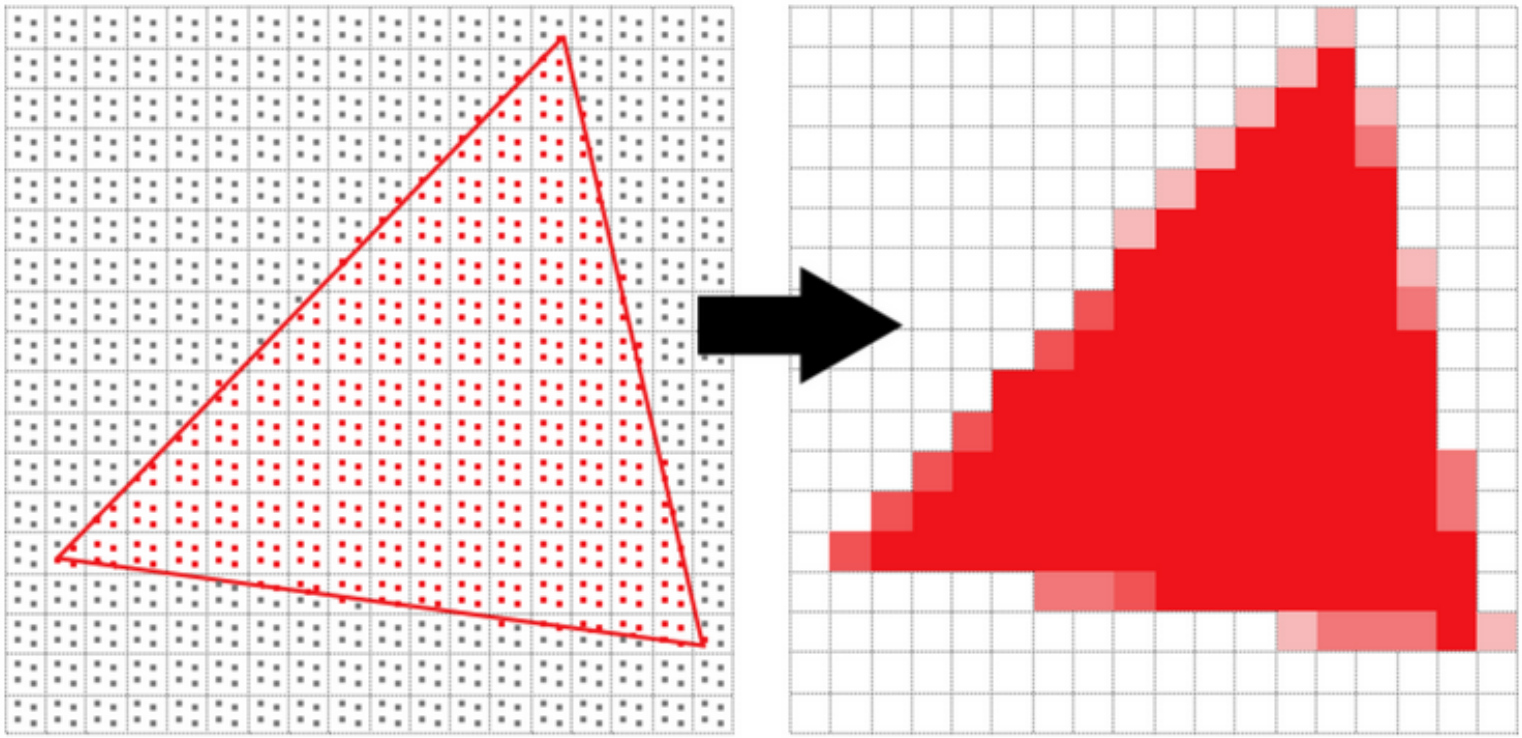


MSAA: Un Campione Per Pixel



- Sample point
- Sample point covered by the triangle

MSAA: Quattro Campioni Per Pixel



MSAA

- Le immagini che supportano il multisampling non sono presentabili
- Tutte le immagini presentabili devono avere un solo campione per pixel
- Creiamo una nuova immagine che abbia il numero di campioni per pixel che desideriamo
- Useremo questa immagine come nuovo color attachment nel render pass
- Usiamo l'immagine della swapchain come color resolve attachment
- Non dobbiamo dimenticarci di abilitare il multisampling quando creiamo un pipeline state object