

Esplorare L'API Grafica Vulkan

Emanuele Franchi

- 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

Inizializzare Vulkan

- Creazione di una Vulkan instance
- Creazione di una finestra (OS)
- Creazione di una presentation surface
- Selezionare una GPU (device fisico)
- Creare un device logico per interfacciarsi con la GPU
- Creare una swapchain per gestire la presentazione d'immagini sulla presentation surface

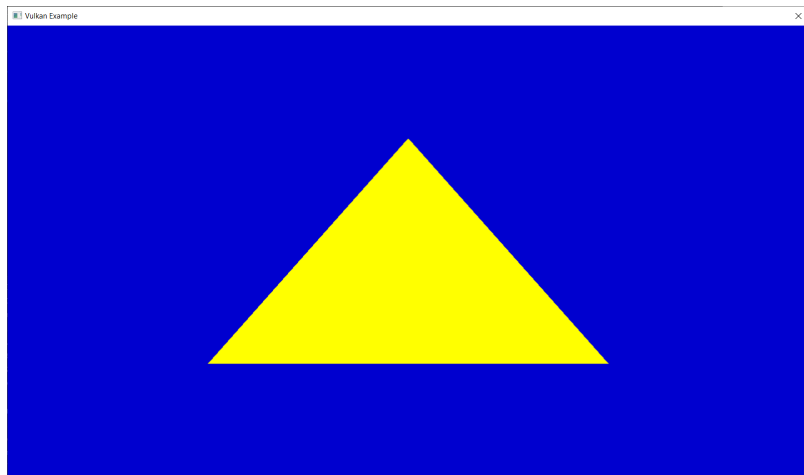
Renderizzare Un Colore: Demo



Renderizzare Un Colore

- Creazione di 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 utilizzano
- Creazione di un framebuffer
- Un framebuffer è l'insieme di attachment utilizzati da un'istanza di un render pass
- Creazione di un command buffer
- Scrittura di comandi su un command buffer
- Inviare un command buffer alla GPU
- Presentare un'immagine sulla presentation surface

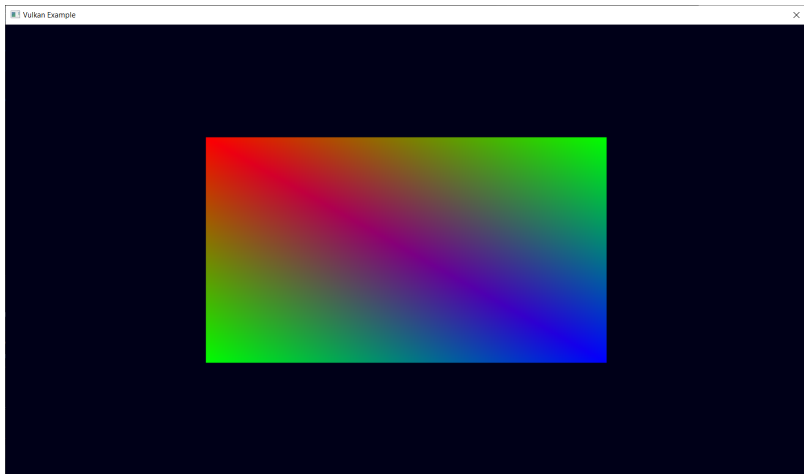
Renderizzare Un Triangolo: Demo



Renderizzare Un Triangolo

- Creazione di un pipeline state object
- Un pipeline state object descrive l'intero stato della pipeline grafica
- Shader: programmi eseguiti dalla GPU
- In particolare vertex shader e un fragment shader
- La GPU riceve come input una sequenza di vertici
- La GPU esegue il vertex shader per ogni vertice
- Il vertex shader genera della geometria
- La GPU esegue il fragment shader per ogni pixel coperto dalla geometria

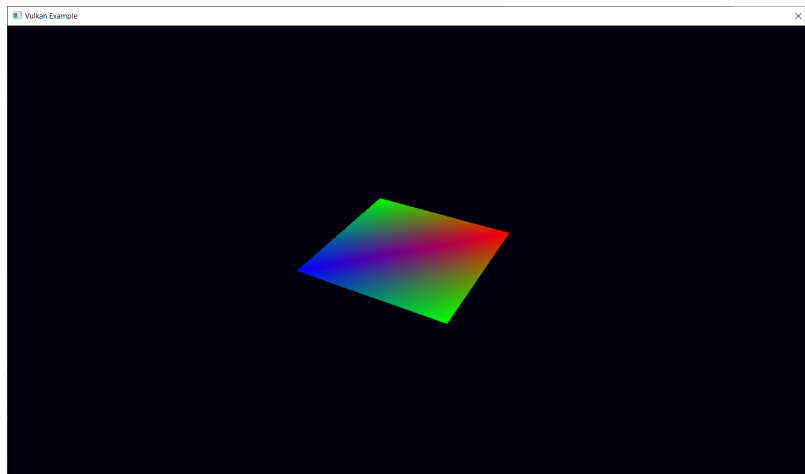
Vertex Buffer: Demo



Vertex Buffer

- I nostri vertici sono in RAM e noi dobbiamo caricarli nella memoria della GPU
- I vertici non cambiano durante l'esecuzione dell'applicazione
- Vogliamo rendere la lettura dei nostri vertici, da parte della GPU, il più veloce possibile
- Usiamo due buffer
- Uno staging buffer, allocato su memoria della GPU host visible (lenta)
- Scriviamo i dati dei nostri vertici su questo buffer
- Un vertex buffer, allocato su memoria della GPU device local (veloce)
- Inviame un comando alla GPU per trasferire il contenuto dello staging buffer nel vertex buffer

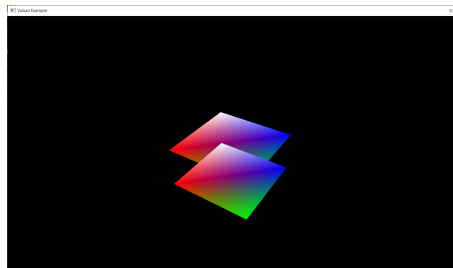
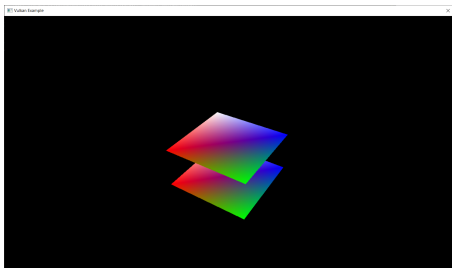
Uniform Buffer: Demo



Uniform Buffer

- Una variabile globale di uno shader è detta uniform
- Passiamo le variabili globali agli shader usando uno uniform buffer
- Siccome tali variabili cambiano frequentemente, allochiamo uno uniform buffer su memoria della GPU host visible
- Creiamo un pipeline layout object
- Un pipeline layout object descrive quali descriptor sono accessibili in quali shader
- Creiamo un descriptor set layout
- Un descriptor set layout descrive il contenuto di un descriptor set
- Creiamo un descriptor set basandoci su un descriptor set layout
- Una volta allocato, un descriptor set va popolato con i descriptor appropriati

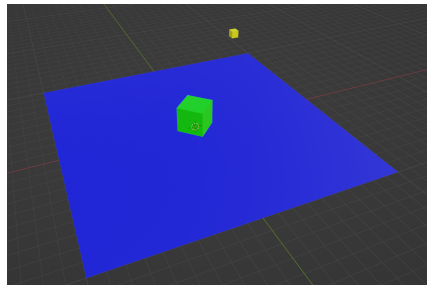
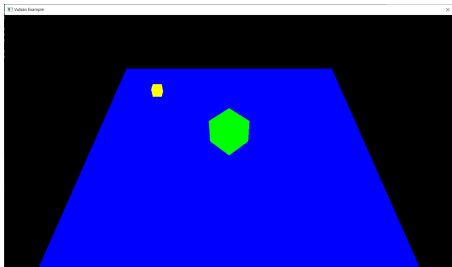
Depth Testing: Demo



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
- E se due oggetti si sovrappongono?
- Se gli oggetti da renderizzare sono opachi, usiamo un depth buffer
- Un depth buffer è un'immagine che codifica informazioni riguardanti la profondità dei frammenti
- Quando un frammento viene generato, compariamo la sua profondità con quella salvata nel corrispettivo texel del depth buffer
- Se è più grande, il frammento non viene ignorato
- Se è più piccola, il frammento viene utilizzato e la sua profondità viene salvata nel depth buffer
- Creiamo un'immagine da usare come depth stencil attachment
- Abilitiamo il depth testing quando creiamo il pipeline state object

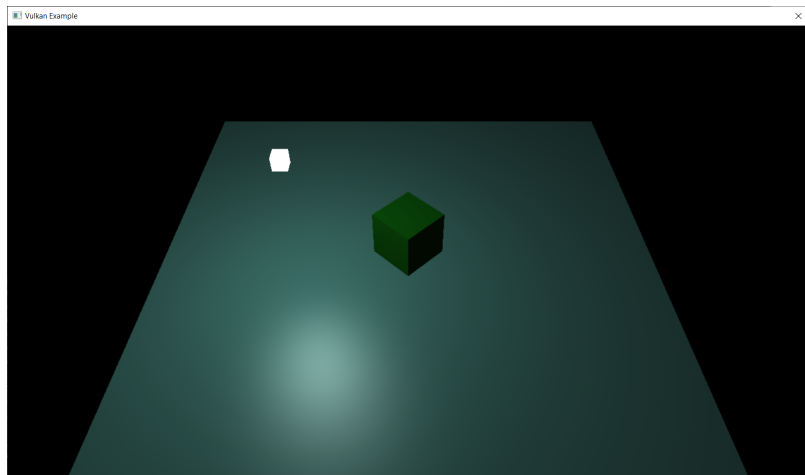
Scena: Demo



Scena

- Organizziamo gli oggetti da renderizzare in una scena
- Abbiamo una camera attraverso la quale guardiamo la scena
- Possiamo mettere uno o più oggetti all'interno della scena
- Questi oggetti possono condividere un vertex buffer e un pipeline state object
- Infatti, oggetti diversi possono usare gli stessi vertici e la stessa configurazione della pipeline
- Ogni oggetto ha un proprio uniform buffer

Blinn-Phong: Demo

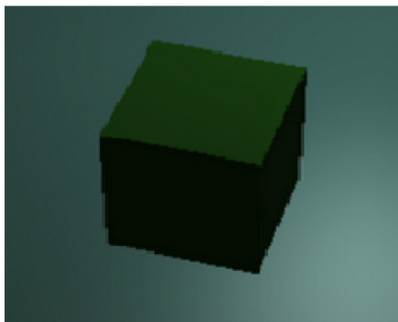


Blinn-Phong

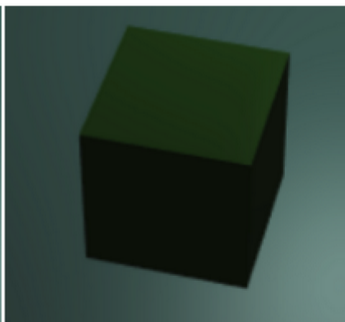
- L'illuminazione viene divisa in tre componenti: ambient, diffuse e specular
- La componente ambientale modella il fatto che una scena non è mai totalmente buia
- La componente diffusiva simula l'impatto che la luce ha sugli oggetti opachi
- La componente speculare simula il punto luminoso che una luce 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

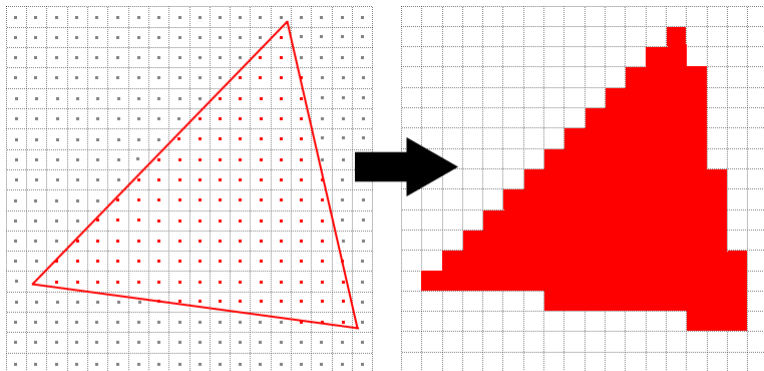
Before



After

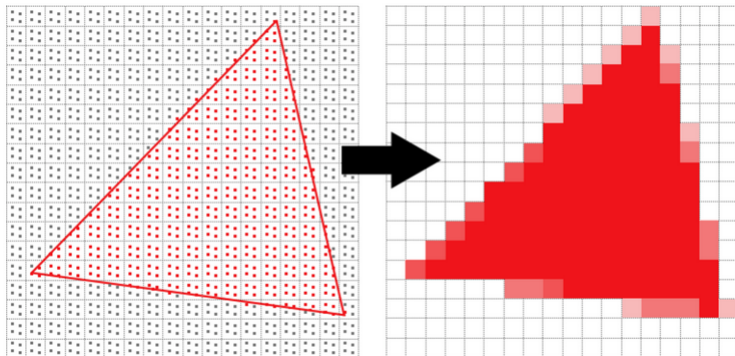


MSAA: Un Campione Per Pixel



- Sample point
- Sample point covered by the triangle

MSAA: Quattro Campioni Per Pixel



- 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