

## Vykreslovací engine

Vykreslovací engine vytváří 2D obrázek ze vstupních dat o scéně. 2D obrázek si můžeme představit jako 2 rozměrné pole, kde prvku se říká pixel. Ten drží nějakou hodnotu.

Vykreslovací engine je většinou součástí nějakého většího systému, například herního enginu. *Herní engine* je vícero enginů, například ještě fyzický, pro simulování kolizí objektů, apod. pro interaktivních zážitků.

### Snímek

Když vykreslíme kompletní obrázek, říká se mu snímek. Tedy při vykreslení tzv. pořídíme snímek.

### Framebuffer

Framebuffer je alokovaný kus paměti, do kterého zapisujeme námi vykreslený obrázek. Dnes už by bylo vhodné rozdělit framebuffer na screen buffer a off-screen buffer.

Screen buffer je kus paměti obrázku, který grafická karta má zobrazit na výstupní displej. Off-screen buffer je také kus paměti, ale využívá se na mezi výpočty. Většina vykreslování v moderních herních vykreslovacích enginech se provádí ve více fázích. Postupně tak vykresluje do obrázku, který následující fáze bere jako vstup a nějak ho upraví a zase vykreslí do nějakého framebufferu.

### Swapchain

Je fronta obrázků, které se prezentují na obrazovku. Screen bufferů tedy často bývá více. Je na vykreslovacím enginu, aby vykreslil do toho správného.

### Real-time

Mějme vykreslovací engine, který při platném vstupu, vytvoří snímek. Je-li schopný produkovat snímky do screen-bufferu sekvenčně takovou rychlostí, že při jejich prohazování se jeví jako pohyb, pracuje engine v reálném čase.

### Scéna

Scénu budeme definovat jako množinu objektů. Každý objekt bude mít tzv. mesh, což je množina vrcholů, hran a polygonů. Polygon je nějaký rovinný tvar, definovaný několika body, které jsou spojené hranami. Nejčastěji má polygon 3 hrany, protože jediné tak je jisté, že všechny body budou ležet na jedné rovině. To je důležité, protože pro například stínování se využívá normála, a na jedné rovině je jednoznačná.

### Rasterizace

Je proces, kdy jednotlivé polygony zobrazíme do obrázku. Obrázek pak můžeme zobrazit na monitoru pro uživatele. Celý proces se skládá z několika fází.

### Projekce

- Simulujeme kameru.
- Skládá se z transformace a projekce.

## Grafická API

### Vulkan

Vulkan je moderní, nízko-úrovňová grafická API. Oproti předchůdcům, jako DirectX 11 nebo OpenGL, dává daleko větší kontrolu nad celým procesem renderování. Mnoho funkcí a optimalizací do té doby dělali samotné ovladače. Vulkan je pouze API, a tím pádem to, jak ve skutečnosti pracuje

grafická karta, závisí na ovladači a Vulkan slouží jen jako rozhraní pro požadavky na grafickou kartu.

## Příkazy

Pomocí Vulkan API můžeme grafické kartě posílat požadavky. Tyto příkazy balíme jako balíčky do tzv. *příkazových bufferů*, aneb *command bufferů*. Např. v DirectX 12 se balíčky nazývají *command listy*, což je trochu výstižnější název. Dále jen *příkazové balíčky* nebo jen *balíčky*, bude-li *příkazové* zřejmé z kontextu.

Balíček má několik stavů, které určují jeho možnosti.

## Recording

Proces, kdy do balíčku postupně přidáváme příkazy, se nazývá *recording*, neboli *nahrávání*. Nejprve oznámíme začátek nahrávání konkrétního balíčku pomocí funkce *vkBeginCommandBuffer*. Poté přidáváme příkazy pomocí. Nakonec vše ukončíme pomocí *vkEndCommandBuffer*.

Pro představu:

```
vkBeginCommandBuffer(commandBuffer); // začni nahrávat

vkCmdCopyBufferToImage(commandBuffer, ...); // přidej příkaz: zkopíruj obsah nějakého
bufferu do obrázku

vkCmdDraw(commandBuffer, ...); // přidej příkaz: vykresli něco

vkEndCommandBuffer(commandBuffer); // ukonči nahrávání
```

Můžeme si všimnout, že volání, které přidá příkaz do balíčku, začíná *vkCmd*.

V předchozích API, jako OpenGL nebo DirectX 11, se příkazy takto neshlukovali. Bylo v režii ovladače zkusit příkazy optimalizovat a poslat do fronty. Nevýhodou bylo, že se muselo vše optimalizovat znovu a znovu každý snímek. Balíčky na druhou stranu můžeme nahrát jednou a posílat je už optimalizované vícekrát. Další výhodou je, že je možné nahrávat více balíčku současně, např. z různých vláken, a tím ještě více program zrychlit.

## Posílání (Submit)

Až bude vhodná doba, můžeme poslat balíček na grafickou kartu pro splnění.

### Fronta

Když chceme, aby se příkazy z balíčku vykonali, pošleme je do tzv. *fronty*. Protože CPU a GPU nejsou synchronizované, nemůžeme začít výpočet okamžitě. Proto se používá *fronta*, kterou grafická karta postupně splňuje.

Každá *fronta* je určité *rodiny*, kde každá *rodina* umí vykonávat určitou sadu příkazů. Obecně jsou 3 sady:

- Grafická: Především vykreslování, ale jestliže *rodina* umí tuto sadu, umí i *transfer* sadu
- Transfer: Pro přesun dat na grafické kartě.
- Compute:

Fronty ale nejsou tzv. *thread safe*, tedy nejsou připravené, aby se do nich nahrávalo z více vláken na CPU zároveň. Proto se jich vytváří více, většinou jedna pro každé vlákno, které k posílání budeme používat.

## Synchronizace

Je zaručeno, že grafická karta začne balíčky a příkazy v něm vykonávat ve stejném pořadí, jako do fronty přišli. Není ale zaručeno, že skončí ve stejném pořadí. To je problém, protože často se používá vícero iterací, než vznikne výsledný obrázek. Je tedy třeba, aby každá iterace proběhla ve správném pořadí. Například, rasterizujeme-li scénu a výsledný obrázek chceme rozmazat:



### Command bufferu

Balíčky můžeme synchronizovat pomocí semaforů. Semafor je synchronizační struktura, která má 3 stavy, tzv. zelenou a červenou. Při posílání balíčků do fronty upřesňujeme, které semaforey se mají signalizovat a na které se má čekat. Signalizovat znamená, že po poslání se tyto semaforey nastaví ze *zelené* na *červené*. Jakmile jsou veškeré příkazy z balíčku splněné, nastaví se semafor na *zelenou*. Poté se veškeré balíčky, které na tento semafor čekali, mohou začít vykonávat.

### Frontu

Příkazy ve frontě můžeme synchronizovat pomocí bariér. Zde je synchronizace omezená jen na typ a fázi příkazu, nikoliv na konkrétní příkaz. To znamená, že když vytvoříme bariéru, oznámíme, na který typ a fázi příkazu čekáme a od které fáze příkazu na to čeká.

### GPU to CPU

Pro synchronizaci mezi grafickou kartou a procesorem je tzv. fence.

## Náročnost

Dělat takovou režii ručně je již nadlidský úkol. Moderní hry mají desítky iterací, než se dostanou k výslednému obrázku, a synchronizovat vše ručně by způsobovalo spoustu chyb. Navíc se iterace mění dynamicky. Např. není třeba spouštět vykreslování vody, když např. žádná voda není v dohledu. Většina her také umožňuje měnit grafické nastavení a určité efekty třeba vypínat.

Tento problém jsem vyřešil Render Grafem, inspirované přednáškou od FrostBite<sup>1</sup>.

To, co se má v iteraci stát, jsem definoval jako tzv. *RenderPass*. Každý takový využívá nějaké zdroje, buď jako vstup nebo výstup. Tyto jsou v Render grafu virtualizované. To znamená, že žádný render pass nemá konkrétní zdroj "jen pro sebe", ale je mu přiřazen jen ukazatel a o samotné vytvoření, alokaci a správu se stará právě render graf.

Každý virtuální zdroj dostane unikátní název. *RenderPass* tento název použije v případě, že na něm chce záviset, nebo do něj naopak psát. Z toho všeho nám vznikne graf závislostí, který bude vypadat třeba takto:

---

<sup>1</sup>FrostBite je moderní herní engine od společnosti EA, známý především pro svou dechberoucí grafiku a zničitelné prostředí