Computer Graphics

Basic Implementation

Subtopics

Framework

Components

- Textures
- Swap Chain
- Depth Buffer
- Descriptors
- DXGI
- COM

Features

- SDK Support
- Multisampling

System

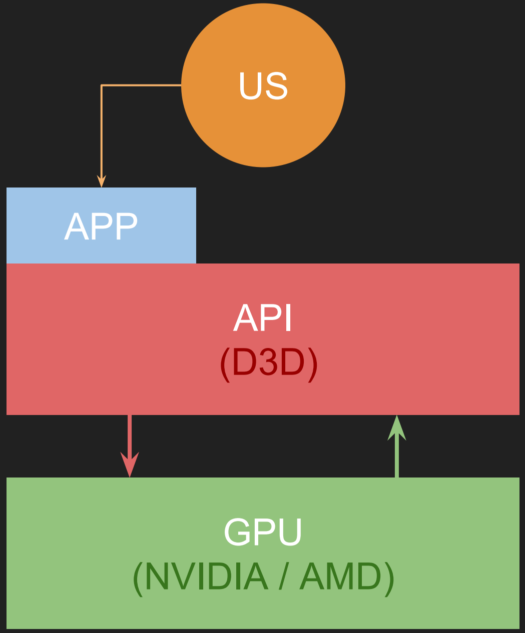
- CPU
- Residency
- Concurrency

Application

- Program’s Loop
- Initialization
- Components
- Basic Drawing
- Timing

Basics - Framework

The following model, simplifies the parts of the general applications development infrastructure we will work on:



We will develop an application based on some framework (OS, windowing system, I/O handling, etc…)

The application will:

- Obtain resources (GPU, buffers, etc) as COM smart pointers from the API
- Call API functions, using the above referenced instances

The API will then interface with the GPU.

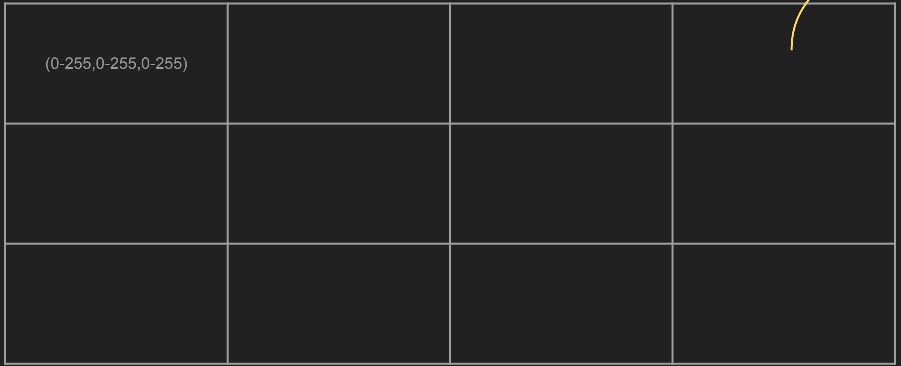
We will grow this picture as we cover all the different components.

Basics - Components - Textures

A texture is a resource, made out of a bidimensional matrix which holds picture information. Each [ith,jth] position of this 2D array, is a string of, for example, 3*32-bit floating values. This are provided by the DXGI_FORMAT enum.

We use textures to represent ANY drawable surface or general purpose resource, and generally speaking, each texture holds PIXEL information.

A common type used is DXGI_FORMAT_B8G8R8A8_UNORM - 8 bits for each color and alpha from [0,1]



For now we will stick to this limited definition, but moving forward, we will see there are many uses for textures as buffers.

Another very useful format, is:

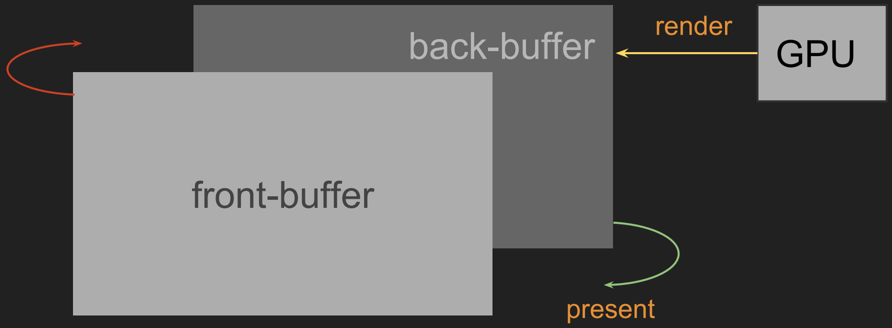
DXGI_FORMAT_R16G16B16A16_TYPELESS

Later we will see how that can be useful in the pipeline.

Basics - Components - Swap Chain

To avoid screen tearing, interruptions and other objects (malformations) during rendering, we can use a technique called double-buffering.

A buffers is a surface (represented as a texture) where the GPU will draw on a full scene, for then, present it to the user. The key, is that the drawing, happens on a buffer that is not visible, and once ready, it is presented and swapped for the previously displayed one. This is done by simply changing pointers references, not copying the buffers again.



In the application, the Swap Chain will:

- Store the buffers references
- Resize buffers
- Present the current front-buffer

Basics - Components - Depth Buffer

At the end of the day, a rendered image, is a 2D pixel array. To keep track of which object should be presented in order, the GPU has a depth-buffer, which is a 1-to-1 map of each pixel in the render buffer, but the difference that is keeps track of each pixel's depth value with range [0 - 1] , where 0 is the closest to the frustum (view).

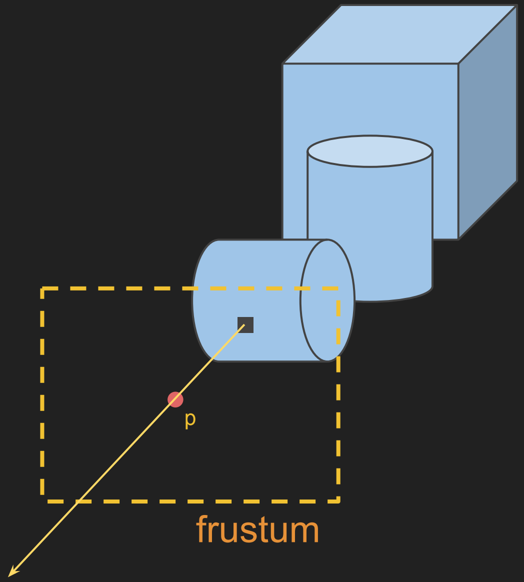
For each object being rendered, the following test is applied:

```
Before rendering ...
          clear depth_buffer = { 1.0 }
          for each object to be rendered
               If $p_{Can}$.depth < $p._{Cur}$.depth
                    update depth_buffer[$i^{th}$][$j^{th}$] = <$p_{Can}$,$p_{Can}$.depth>
```

This guarantees that regardless of the rendering order, objects will always (should) obscure each other correctly.

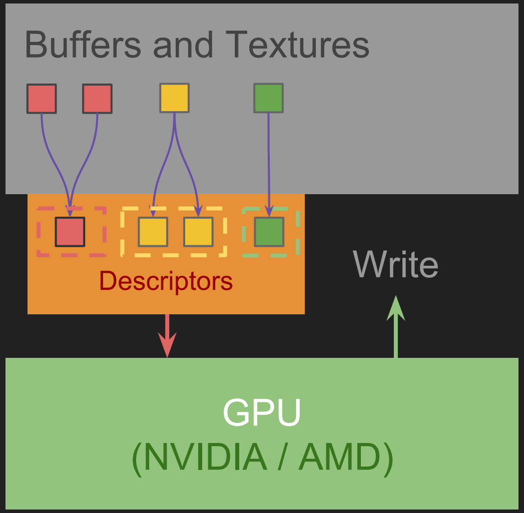
The depth/stencil buffer are described by D3D12_RESOURCE_DESC and might use (for example) formats:

DXGI_FORMAT_D24_UNORM_S8_UINT
DXGI_FORMAT_D32_FLOAT (with or without stencil)



Basics - Components - Descriptors

Every time before drawing, we need to describe what we are sending to the GPU.
These descriptors, are wrappers which will bind to the GPU's pipeline.



Each descriptor provides mainly two things:

- A reference to the resource
- Information about the data it points to (resources).
    - Type (Direct3D reference):
        - Buffers, shaders, unordered resources - (CBR, SRV, UAV)
        - Render Targets - (RTV)
        - Depth/ Stencil - (DSV)
        - Vertex Buffer - (VBV)
        - Samplers

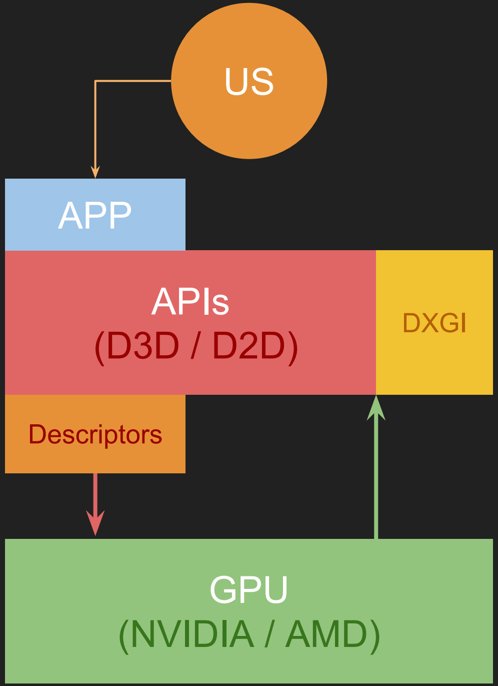
Each groups of descriptors of the same type, are stored in arrays (heaps)

Something worth mentioning is that descriptors are also known as "Views"

Note how above, many descriptors are bounded to the same RTV resource. This resource might have different roles during rendering.

Basics - Components - DXGI

In addition to Direct3D, we will also interact with DirectX Graphics Infrastructure



DXGI will centralize many enums, interfaces and many other resources, shares by different APIs (i.e. Direct2D), such as:

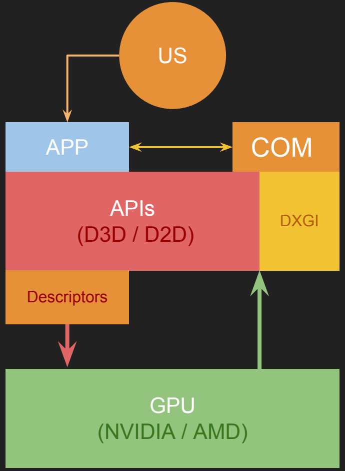
- Swap Chain interface - IDXGISwapChain
  - DXGI_SWAP_CHAIN_DESC
- Fullscreen management
- Display Adapters - IDXGIAdapter (GPU)
  - DXGI_ADAPTER_DESC
- Monitors - IDXGIOutput f/e Adapter
  - DXGI_OUTPUT_DESC
- Display Modes - DXGI_MODE_DESC
  - _RATIONAL, _FORMAR, _MODE_SCALING, Width, Height and more
- Data formats - DXGI_FORMAT_*

Basics - Components - COM

The Component Object Model is a operating system-wide generic (template) interface which we will use as “smart pointers” wrappers to count and manipulate resources. For our purpose, we only need to know:

- It is defined in the <wrl.h> header file - `Windows::WRL::ComPtr`
- Each instance is counted by the system, when it is out of scope, it is auto-collected, and the counter decreased.
- Each instance can be “released” either by using `.Reset()` or nullptr

  This ensures memory will be auto-collected
- Any wrapped resource will provide two methods:
  - `Get()` - Will return the pointer of the Interface we wrapped with COM
  - `GetAddressOf()` returns the & (address) of the wrapped interface
- In Practice:
  - Good	ComPtr<IDXGISwapChain> m_SwapChain;
  - Bad	IDXGISwapChain* m_SwapChain;



Basics - Features - Support

For backward compatibility and features check, Direct3D provides us with a way to check which is the highest Direct3D - and other features - level supported by the host.
Most common versions will be any in between 9 (legacy) and 12 (newest)
Checking for support is crucial to ensure we will not use any features not supported by the platform. Also, we could plan on falling back to older SDKs should the main one is not supported.

For this, D3D provides us with

```
HRESULT ID3D12Device::CheckFeatureSupport
```

D3D12_FEATURE - enum for the features we want to check: D3D12_DATA_*

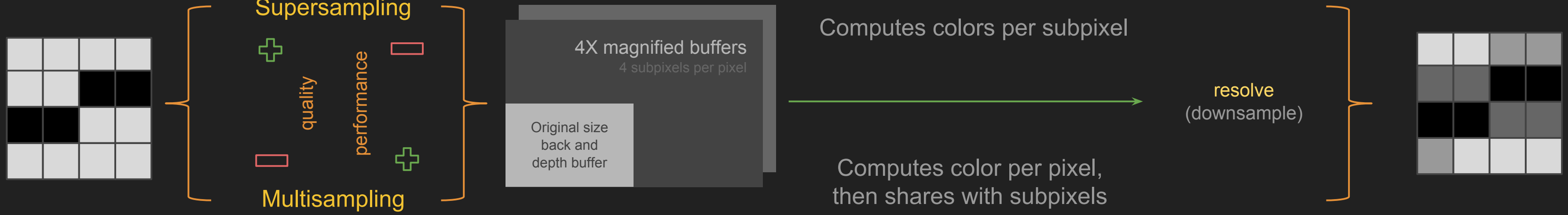
- _ARCHITECTURE, _FEATURE_LEVELS, _FROMAT_SUPPORT, _MULTISAMPLING_QUALITY_LEVELS

For example, DXGI_FEATURE_DATA_FEATURE_SUPPORT is the one we would use to obtain the maximum API level supported.

Basics - Features - Multisampling

The GPU provides us with mechanisms for reduce “stair-steps” effect when rendering geometries. These is a natural effect of drawing with pixels, which are not infinite. The lower the resolution, the less available pixels, hence the effect. By increasing the resolution, with more pixels available, AA might not be needed.

Two major techniques to handle this issue are:


Supersampling
quality
performance
Multisampling
4X magnified buffers
4 subpixels per pixel
Original size back and depth buffer
Computes colors per subpixel
Computes color per pixel, then shares with subpixels
resolve
(downsample)

The final product’s (downsampled buffer) colors, are obtained by averaging the color of the each 4-subpixels block into each original pixel.

Basics - System - Residency

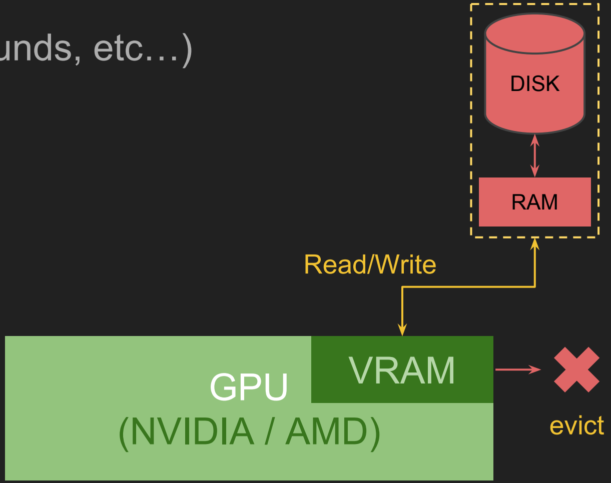
Just like in OS memory management, residency refers to a set of pages which are loaded in memory for the system (in our case the GPU) to use. Then, we have a working set, which is a subset of this resident one, and is defined by those pages which were recently used in a window of time.

When the GPU is not using a portion of memory, it will evict those blocks, freeing up space for optimization or bringing up more needed resources.

What are we loading (binding) in memory: RESOURCES (textures, sounds, etc…)

How can we manage their life-cycle:

- Create (load) and Destroy (evict) resources on demand
- Using D3D12 mechanisms:
  - `HRESULT D3D12MakeResident`
  - `HRESULT D3D12Evict`



Basics - System - CPU and Command Queues

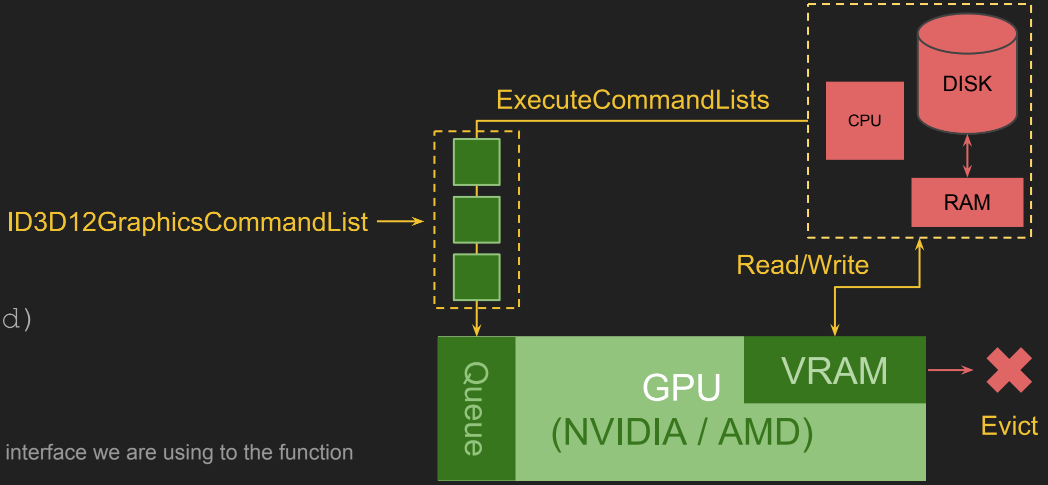
The GPU has its own processor, different than the system's CPU. Ideally, these two will work in parallel, with no serialized jobs, hence not idle waiting for the other to finish processing.

The CPU submits commands to the GPU via the GPU's ID3D12CommandQueue interface, then, the GPU will execute them in FIFO fashion (hence the queue).

The command queue, command lists and other interfaces are defined and created by the device interface.

The data structure and functions to utilize are:

```
-   ID3D12CommandQueue
-   ID3D12GraphicsCommandLists
-   ID3D12CommandAllocator (stay tuned)
```



IID_PPV_ARGS(<&ComPtrInterface>) → this macro pases the ID of the COM interface we are using to the function

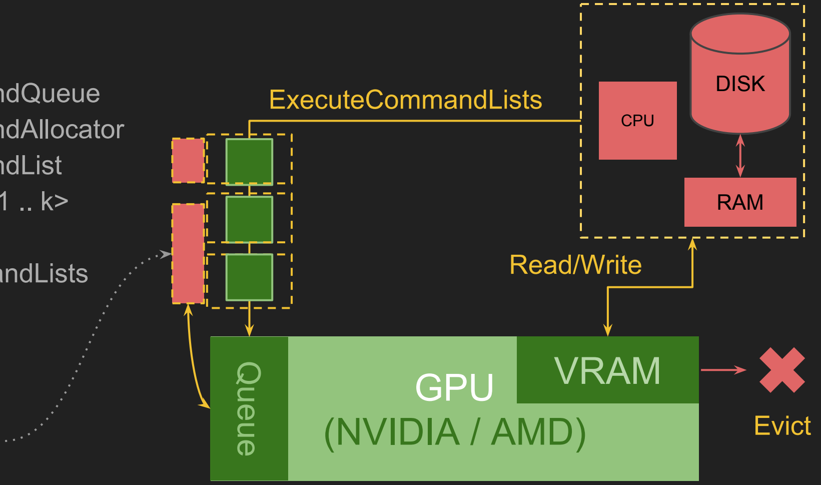
Basics - System - CPU and Command Queues

The lists will hold references to structures (commands) to be queued and then executed. The actual memory for these structures are held in an entity called a ID3D12CommandAllocator . There are two main types of allocators:

1) D3D12_COMMAND_LIST_TYPE_DIRECT
   - directly and sequentially executed.
2) D3D12_COMMAND_LIST_TYPE_BUNDLE
   - optimized lists of commands to be built on INITIALIZATION (preprocessed). Only used to better CPU usage utilization if needed (profiler showing overhead), otherwise, they should not be used.

Summarizing, the process is:

Obtain a command queue from the device	device->CreateCommandQueue
Create a command allocator	device->CreateCommandAllocator
Create a command list	device->CreateCommandList
Add commands	list-><record command 1 .. k>
Close the list	cList->Close
Execute the command list	queue->ExecuteCommandLists
Reset command lists’	cList->Reset

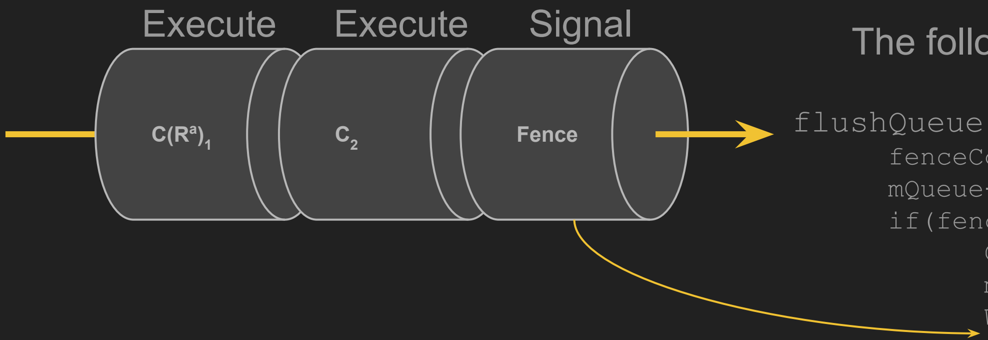


Many command lists can share the same allocator.
But only one can record at a time, rest must be closed.

Basics - System - Concurrency - Fence

Sometimes, we do need to synchronize the CPU and the GPU in order to avoid "race conditions" when updating resources, such as geometries positions. This is achieved by using Fences - `ID3D12Fence`.

In a nutshell, a fence is a point at which the GPU will block the command queue, until it has been fully flushed, for then signaling the CPU to inform is yet again open for business.



The following pseudo code creates a blocking signal

```
flushQueue() {
        fenceCounter++; // UINT64
        mQueue->Signal(&mFence, fenceCounter)
        if(fence->GetVal < fenceCOunter)
                Create HANDLE eh
                mFence->SetEventOnCompletion(fenceCounter,eh)
                WaitForSignalObject(eh,<time>) // block CPU
                closeHandle(eh)
}
```

ID3D12CommandQueue API

Basics - System - Concurrency - Barrier

Similarly, imagine we have a resource R, which needs to be fully written on, before it can be read as a shader or any other resource. We achieve this by specifying aR's state using Barriers - D3D12_RESOURCE_BARRIER.

One type of barriers are the TRANSITIONS. We indicate the GPU on which state the resource is.

For example:

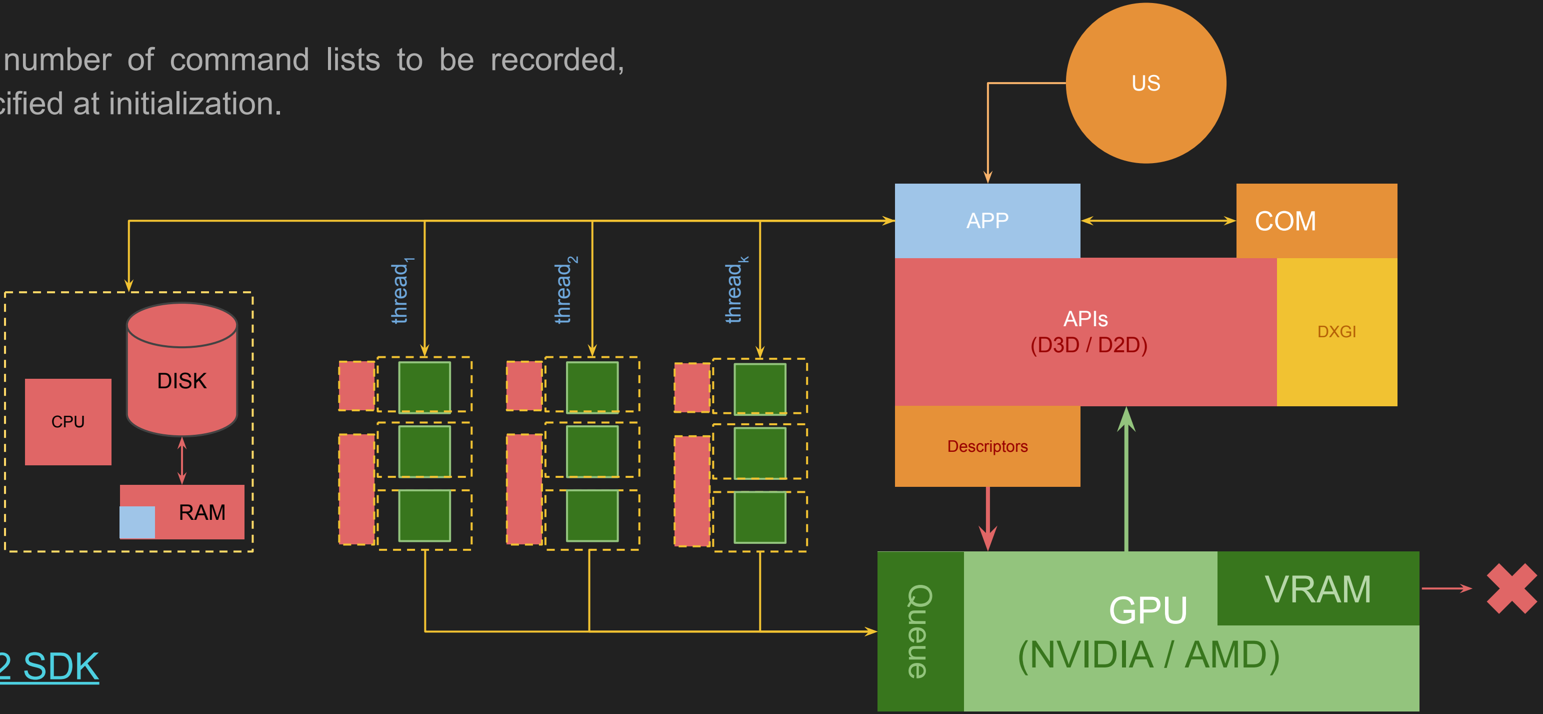
```
R.state = writable             commandList->ResourceBarrier(R,oldState,newState)
GPU.draw(R)
R.state = readable
GPU.draw(R)        // BLOCKED!
R.state = writable
GPU.draw(R)        // Ok
```

To simplify this implementation, MS provides the following header file: [d3dx12.h]()

Basics - System - Threading

Finally, it is important to understand how threads work with command lists. The queue is thread-safe (synchronized) but each thread cannot share their command list and allocator.

The maximum number of command lists to be recorded, need to be specified at initialization.



Multithreading 12 SDK

Basics - Application - Render-based Run loop

Generally speaking, frame-based rendering applications, run in a loop which continuously draws for each iteration. The body is the following:

```
void App::Run() {
     while(windowIsNotClosed) {
          read I/O
          if(timeToUpdate) {
               App::UpdateLogic()
               Draw
          }
     }
}
```

We will use a time controller class. Based on a target number of frames per second (usually 30, 60 unlimited)

Direct3D - Initialization

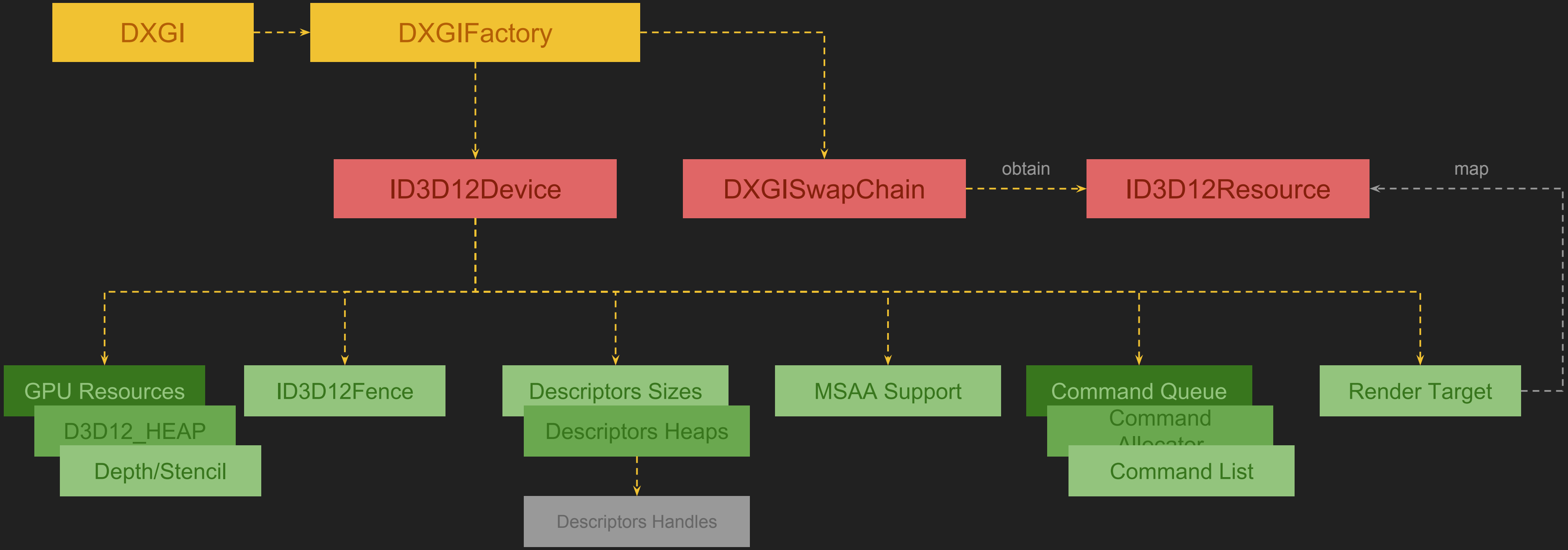
Initializing Direct3D is a lengthy process with enormous room for tweaking. Still we can summarize the following steps:

1. Obtain a DXGI factory
2. Create the main device
3. Create a fence
4. Obtain the resources Descriptors sized (from the current HW)
5. Check for Multisampling (antialiasing) support and primary output information (monitor)
6. Create the Command Queue, Allocator and Command List
7. Initialize the swap chain
8. Create the descriptors heaps and handles (head pointers of the heaps)
9. Create and set the render target (where to draw to)
10. Create and set the depth/stencil buffer
11. Set the Viewport

These is processed when resizing

Direct3D - Main Components

The following is a hierarchical representation of components dependencies:



Direct3D - Basic Drawing Draw Call

Draw() will be invoked once per frame. The basic steps are:

1) Reset the command lists and allocators
2) Set the current buffer as a RENDER TARGET state
3) Set the viewport / scissors
4) Clear the render target
5) Clear the back buffer
6) Use the pipeline
7) Set the render target
8) Set the current buffer back to PRESENT state
9) Close the list
10) Present the new buffer
11) Swap the current buffer counter
12) Flush the commands in the GPU’s queue

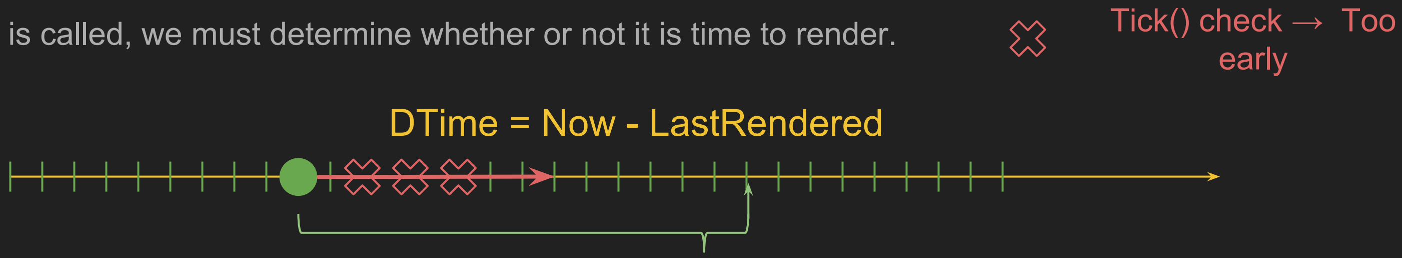
Direct3D - Timing

In rendering based applications, timing is really important due to various reasons:

1) Rendering update
2) Animations
3) Logic processing
4) I/O

It is really important we could measure how long has it passed since we rendered the last frame, or how many frames per second we are drawing on the screen.

Every time Tick() is called, we must determine whether or not it is time to render.



Let TargetFPS = 60 then TargetTime = (1 / FPS)

If (Dtime >= TargetTime) → Render

Direct3D - Timing

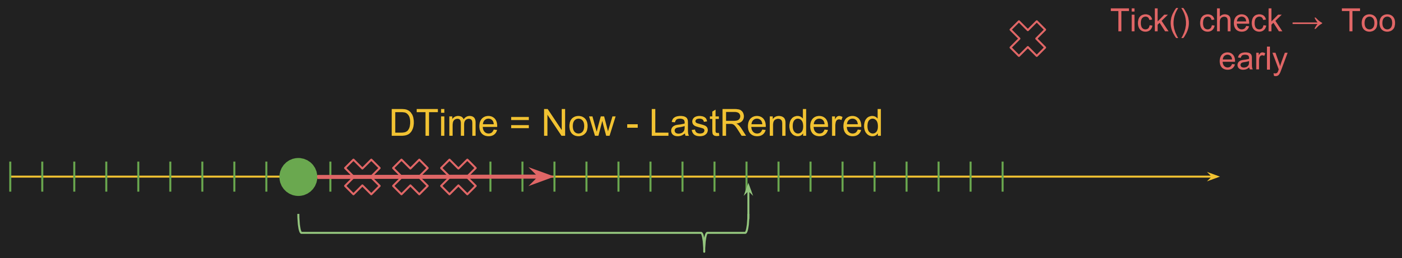
In our examples, we will use a very precise CPU measure function called QueryPerformanceCounter.

Given a long integer **n**, calling QueryPerformanceCounter(**n**) is analogous to querying current time (in different measure units).

Before this, we must obtain the CPU's constant number of units of execution per second. Let this be **k**, it can be obtained with QueryPerformanceFrequency(**k**). We can then set our frames to be updated every TargetTime = (k / TargetFPS) times a second.

Finally, we can standardize another useful measure, which is DeltaTimeSecs (Actual seconds) from frame to frame the following way:

```
DeltaTimeSecs = (Now - LastRendered) * (1/k); // converting CPU counts into seconds.
```



Let TargetFPS = 60 then TargetTime = (k / FPS)

If (DTime >= TargetTime) → Render → Reset