내용

[Chapter 1 소개 4](#_Toc3149340)

[튜토리얼에 대하여… 4](#_Toc3149341)

[E-book 5](#_Toc3149342)

[튜토리얼 내용(Structure) 5](#_Toc3149343)

[Chapter 2 개요 7](#_Toc3149344)

[Vulkan의 기원 7](#_Toc3149345)

[삼각형을 그리기 위해 필요한 것 8](#_Toc3149346)

[Step 1 – 인스턴스와 물리 장치(Physical device) 선정 8](#_Toc3149347)

[Step 2 – 논리 장치(Logical device)와 큐 패밀리 8](#_Toc3149348)

[Step 3 – 창 표면(Window surface)와 스왑 체인 8](#_Toc3149349)

[Step 4 – 이미지 뷰 그리고 프레임버퍼 9](#_Toc3149350)

[Step 5 – 렌더 패스 9](#_Toc3149351)

[Step 6 – 그래픽 파이프라인 10](#_Toc3149352)

[Step 7 – 명령 풀 그리고 명령 버퍼 10](#_Toc3149353)

[Step 8 – 메인 루프 11](#_Toc3149354)

[요약 11](#_Toc3149355)

[API 개념 12](#_Toc3149356)

[코딩 관습(Convention) 12](#_Toc3149357)

[유효성 검사 레이어 13](#_Toc3149358)

[Chapter 3 개발 환경 15](#_Toc3149359)

[Windows 15](#_Toc3149360)

[Vulkan SDK 15](#_Toc3149361)

[GLFW 17](#_Toc3149362)

[GLM 17](#_Toc3149363)

[Visual Studio 설정하기 18](#_Toc3149364)

[Linux 28](#_Toc3149365)

[Vulkan SDK 28](#_Toc3149366)

[GLFW 28](#_Toc3149367)

[GLM 28](#_Toc3149368)

[Makefile 프로젝트 설정하기 28](#_Toc3149369)

[MacOS 28](#_Toc3149370)

[Vulkan SDK 28](#_Toc3149371)

[GLFW 28](#_Toc3149372)

[GLM 28](#_Toc3149373)

[Xcode 설정하기 28](#_Toc3149374)

[Chapter 4 기본 코드 29](#_Toc3149375)

[전반적인 구조 29](#_Toc3149376)

[자원 관리 31](#_Toc3149377)

[GLFW 통합하기 31](#_Toc3149378)

[Chapter 5 인스턴스 35](#_Toc3149379)

[인스턴스 생성하기 35](#_Toc3149380)

[확장 지원 확인 37](#_Toc3149381)

[정리 38](#_Toc3149382)

[Chapter 6 유효성 레이어 39](#_Toc3149383)

[유효성 레이어가 무엇인가? 39](#_Toc3149384)

[유효성 레이어 사용하기 40](#_Toc3149385)

[메시지 콜백 43](#_Toc3149386)

[구성 50](#_Toc3149387)

Chapter 1 소개

튜토리얼에 대하여…

이 튜토리얼은 Vulkan 그래픽스와 Compute API에 대해 가르칠 것입니다. Vulkan은 Khronos 그룹의 새로운 API로 현대(Modern) 그래픽 카드에게 더 향상된 추상화(Abstraction)을 제공할 것입니다. 이 새로운 인터페이스는 당신의 응용 프로그램이 더 좋은 성능을 이끌어내고, OpenGL과 Direct3D와 같은 API와 비교해서 덜 예외적인(Less surprising) 행동을 할 수 있도록 작성하기 쉽도록 제공되고 있습니다. 이러한 Vulkan의 아이디어는 Direct3D 12와 Metal과 비슷할 수 있습니다. 하지만 Vulkan은 완전한 크로스 플랫폼으로써 Windows, Linux 그리고 Android에서도 동시에 개발할 수 있다는 점에서 장점(Advantage)을 가지게 됩니다.

하지만 이러한 장점(benefits)를 취하기 위해서는 굉장히 장황한 API작업을 해야 합니다. 당신이 급조한 응용 프로그램에서도 프레임 버퍼 생성 초기화, 버퍼나 텍스처 이미지와 같은 메모리 관리자와 같은 그래픽 API와 관련해서 상세하게 작성(detail)해야 합니다. 그래픽 드라이버에 의존하는(less hand-holding)게 많이 줄어들기 때문에 올바른 행동을 할 수 있도록 더 많은 작업을 해야 합니다.

여기서 전달하는 메시지는 Vulkan은 모두를 위한 API가 아니라는 것입니다. 이는 고성능의 컴퓨터 그래픽에 열광하는(enthusiastic) 프로그래머와 기꺼이 그렇게 하려는 사람(are willing to put some work in)을 타겟으로 잡고 있습니다. 만약에 당신이 컴퓨터 그래픽스보다 게임 개발에 더 흥미가 있다면 Vulkan에 대한 흥미가 곧 떨어져 OpenGL이나 Direct3D을 다시 선호하게 될 수 있습니다. 다른 방법이라면 Unreal 엔진이나 유니티는 Vulkan을 사용할 수 있을 뿐만 아니라 더 높은 API를 제공할 것입니다.

이걸로 이제 이 튜토리얼을 따라오기 위한 준비물을 공개하겠습니다.

* Vulkan과 호환되는 그래픽 카드와 드라이버 (NVIDIA, AMD, Intel)
* C++에 대한 경험 (RAII와 초기화 리스트에 친숙)
* C++ 11이 호환되는 컴파일러 ( Visual Studio 2013 이상, GCC 4.8 이상)
* 3D 컴퓨터 그래픽스에 대한 어느정도의 경험

이 튜토리얼에서는 OpenGL이나 Directe3D의 개념(Concepts)에 대한 얘기는 하지 않을 것입니다. 그러나, 3D 컴퓨터 그래픽스에 대한 기초 지식은 갖추고 있어야 합니다. 원근 투영과 같은 수학적 배경을 설명하지 않기 때문에 컴퓨터 그래픽스 개념에 대해서는 온라인 자습서를 참조하는 게 좋습니다.

원한다면 C++ 대신 C를 사용할 수 있지만, 예제와 다른 선형 대수 라이브러리를 사용해야 하고, 본인만의 코드를 작성해야 합니다. 우리는 로직과 자원의 생존주기(Lifetime)을 설명하기 위해서 C++의 클래스와 RAII와 같은 기능을 이용할 것입니다.

E-book

E-book으로 이 튜토리얼을 읽고 싶으면 EPUB파일이나 PDF 파일을 다운로드 받을 수 있습니다.

* EPUB
* PDF

튜토리얼 내용(Structure)

우리는 첫 번째 삼각형을 화면에 띄우는 방법과, Vulkan의 작동 방식에 대한 개요로 시작하겠습니다. 모든 작은 단계의 목적은 당신이 전체적인 그림 안에서 기본적인 역할을 이해할 수 있도록 할 것입니다. 다음으로, Vulkan SDK와 선형 대수 계산을 위한 GLM 라이브러리, 창 생성을 위한 GLFW로 개발 환경을 설정할 것입니다. 튜토리얼은 Windows에서 Visual Studio, Ubuntu Linux에서 GCC를 이용한 설정 방법을 알려 줄 것입니다.

그리고 삼각형을 그리는데 필요한 Vulkan 프로그램의 요소들을 모두 구현할 것입니다. 각 챕터는 간략하게 다음과 같은 구조를 따르게 됩니다.

* 새로운 개념에 대한 소개와 목적 소개
* 관련 API를 모두 호출하고 프로그램에 통합
* 추상화 된 헬퍼 함수 작성

각 챕터는 이전 챕터의 후속 내용으로 작성이 되어있지만, Vulkan 기능을 소개하고 알아볼 수 있는 독립 챕터로써 읽을 수도 있습니다. 이는 사이트에서 유용한 참고 자료로 이용할 수도 있습니다. 모든 Vulkan 함수와 타입들은 사양서(Specification)에 링크되어 있으니, 더 알아보시고 싶으시면 클릭하시면 됩니다. Vulkan은 매우 새로운 API입니다. 그래서 사양서에 문제점(Shortcomings)가 있을 수 있습니다. 문제점을 발견하면 Khronos 저장소에 피드백을 남겨주시길 바랍니다.

이전에 언급했듯이, Vulkan API는 그래픽스 하드웨어를 최대한 통제하기 위해 많은 매개 변수(Paramter)가 포함된 장황한 API를 가지고 있습니다. 이는 간단한 텍스처 생성과 같은 작업에도 많은 단계를 거치고, 매번마다 반복해야 하는 작업을 야기할 수 있습니다. 그러므로 우리는 우리만의 튜토리얼에서 사용할 헬퍼 함수들을 작성할 것입니다.

모든 챕터는 코드가 작성되어 있는 링크를 포함합니다. 만약에 당신이 코드 구조에 대해서 궁금증(Doubts)이 들거나, 디버깅 또는 비교를 하고 싶을 때 알아볼 수 있습니다. 모든 코드 파일은 다양한 인증된 공급자(Vendors)에서 테스트되었습니다. 또한 각 챕터의 마지막에서는 특정 주제에 대해서 질문을 할 수 있는 섹션이 있습니다. 플랫폼과, 드라이버 버전, 소스코드 그리고 예상했던 행동과 실제 행동을 작성하면 도움을 드리겠습니다.

이 튜토리얼은 커뮤니티의 영향을 위해서 작성되었습니다. Vulkan은 여전히 새로운 API이며, 대표적으로 좋은 예제는 아직 없습니다. 만약에 튜토리일과 사이트에 대해서 피드백이 있다면 망설이지 마시고 GitHub 저장소에 issuse나 pull request를 남겨주시길 바랍니다. 저장소에 Watch를 해두신다면 튜토리얼 업데이트에 대한 알림을 받으실 수 있습니다.

당신이 Vulkan의 삼각형을 화면에 띄우는 절차를 마친 후에, 선형 변환을 포함한 텍스처, 3D 모델작업을 시작할 것입니다.

이전에 그래픽스 API를 다뤄보았다면 첫 번째 지오메트리를 띄우는데 많은 작업이 있음을 알 수 있습니다. Vulkan에는 많은 초기화 단계들이 있지만, 각각의 단계들이 이해하기 쉬울것이고, 중복되지 않다고 느낄 것입니다. 또한 보이는게 지루한 삼각형을 띄우고 나면 완벽히 그려진 3D 모델을 띄우는데 큰 노력이 들지 않으며, 각 단계를 넘을 때마다 보람참을 느끼는 게 중요합니다.

만약에 튜토리얼을 따라오던 중 문제를 맞닥뜨리게 되면, 첫 번째로 FAQ에서 당신의 문제와 해결법이 있는지 확인해보십시오. 이후에도 문제가 계속 된다면 가장 가까운 챕터의 질문 섹션에서 편하게 도움을 요청하세요.

고성능 그래픽스 API에 뛰어들 준비가 되셨나요? Let’s go!

Chapter 2 개요

이번 챕터에서는 Vulkan에 대한 소개와 현 시점의 문제점을 소개하고 끝마칠 것입니다. 그리고 나선 첫 번째 삼각형을 그리는데 필요한 구성 요소들을 살펴볼 것입니다. 이러한 방식으로 다음 챕터에 대한 큰 그림을 그릴 수 있도록 합니다. 우리는 Vulkan API의 구조와 일반적인 사용 패턴 등을 알아보고 끝을 맺을 것입니다.

Vulkan의 기원

다른 이전의 그래픽스 API와 비슷하게 Vulkan은 GPU에서 추상화된 크로스 플랫폼 API로 디자인되었습니다. 이 시대에서 디자인된 기능적 그래픽스 하드웨어의 대부분의 API들의 문제점은 구성 가능한(Configurable) 고정 기능(Functionality)에 제한된다는 점입니다. 프로그래머는 표준 포맷내의 버텍스 데이터를 제공해야 하고, GPU 제조사에서 제공하는 라이팅과 셰이딩 옵션에 감사(Mercy)해야할 따름입니다.

그래픽 카드 아키텍처가 성장(Matured)해갈수록 점점 더 많은 프로그래밍 가능한 기능들을 제공하기 시작할 것입니다. 이러한 모든 새로운 기능들은 어떻게 든 현재의 API에 통합해야 합니다. 이러한 결과는 불완전한 추상화를 낳을 수 있으며, 현대의 그래픽스 아키텍처에 대한 프로그래머의 의도에 대해서 그래픽 드라이버 단에서 많은 의혹(Guesswork)을 남기게 될 것입니다. 이러한 이유 때문에 게임의 성능을 향상시키기 위해 많은 드라이버 업데이트가 있습니다, 때때로는 상당한 기간을 두고 업데이트 하기도 합니다. 이런 드라이버들의 복잡성 때문에 응용 프로그램 개발자들은 셰이더를 위한 문법과 같은 공급자 간의 불일치성을 해결해야 합니다. 이러한 새로운 특징 외에도, 지난 10년 동안 강력한 그래픽 하드웨어를 갖춘 모바일 기기들이 쏟아져 나왔다. 이러한 모바일 GPU들은 각기 다른 에너지 그리고 공간 요구 조건에 대한 아키텍처를 기반으로 잡고 있습니다. 한 가지 예제를 뽑자면 타일 렌더링과 같은 기능을 프로그래머에게 더 제공함으로써 더 좋은 성능을 이끌어 낼 수 있도록 하는 것입니다. 제한된 멀티 쓰레딩 지원을 하는 API가 유래하는 또 다른 제약으로는, CPU 단에서 병목 현상이 일어날 수 있다는 점입니다.

Vulkan은 현대 그래픽스 아키텍처들을 위해서 이러한 문제점을 해결할 수 있도록 처음부터 설계되었습니다. 이는 프로그래머가 장황한 API를 사용해 작업에 대해서 깔끔하게 명시하여 드라이버 오버헤드를 줄일 수 있으며, 멀티 쓰레드를 사용해 명령을 동시에(parallel) 생성하고 입력할 수 있도록 합니다. 셰이더 컴파일에서 단일 컴파일러를 이용해 표준화된 바이트 코드 포맷으로 치환함으로써 불일치성을 줄일 수 있습니다. 마지막으로 그래픽스와 컴퓨트 기능을 하나의 API로 통합함으로 현대 그래픽 카드의 범용 계산 능력을 신뢰할 수 있습니다.

삼각형을 그리기 위해 필요한 것

이제 우리는 잘 작동하는 삼각형 그리기 Vulkan 프로그램의 모든 단계의 개요를 살펴볼 것입니다. 여기서 소개된 모든 개념들은 다음 챕터에서 상세하게 설명이 될 것입니다. 이번은 그냥 각각의 요소들에 대해서 큰 그림을 그릴 수 있도록 해드립니다.

Step 1 – 인스턴스와 물리 장치(Physical device) 선정

Vulkan 응용 프로그램은 **VkInstance**라는 Vulkan API를 설정함으로써 시작됩니다. 인스턴스는 당신의 응용 프로그램에 대한 설명과 사용할 확장 API을 기술함으로써 생성됩니다. 인스턴스를 생성한 후에는, Vulkan 지원 하드웨어를 질의(Query)할 수 있고, 명령을 실행하기 위한 하나의 또는 더 많은 **VkPhysicalDevices**를선택할 수 있습니다.원하는 장치들을 선택하기 위해 VRAM 크기와 장치 사양(Capabilities)와 같은 속성들을 질의할 수 있습니다.

Step 2 – 논리 장치(Logical device)와 큐 패밀리

사용할 하드웨어 장치를 선택하고 난 후, VkDevice를 생성해야 합니다. 여기서는VkPhysicalDeviceFeatures에서 다중 뷰포트 렌더링과 64bit 부동 소수점과 같은 사용할 기능들을 구체적으로 서술합니다. 또한 여기서 당신이 사용하고 싶은 큐 패밀리를 지정해야합니다. 그리기 명령이나 메모리 연산과 같은 대부분의 연산들은 VkQueue에 제출함으로써 비동기 시간적으로 실행됩니다. 큐들은 큐 패밀리로부터 할당되며, 각각의 큐 패밀리는 특정 집합의 명령어들을 각 큐들에게 지원할 수 있도록 합니다. 예를 들어, 그래픽스, 컴퓨트 그리고 메모리 전달 연산에 대해서 각각의 큐 패밀리가 존재할 수 있습니다. 큐 패밀리에 대한 유용성은 물리 장치 선정에서 다른(Distinguishing) 매개 변수를 가지게 되는 곳에서도 사용될 수 있습니다. 이는 Vulkan을 지원하는 장치가 그래픽 기능을 지원하지 않을 수도 있지만, 오늘날의 Vulkan을 지원하는 모든 그래픽 카드는 우리가 관심있는(interested) 큐 연산을 지원할 것입니다.

Step 3 – 창 표면(Window surface)와 스왑 체인

당신이 화면에 나타나지 않는(Offscreen) 렌더링에 관심이 있는 게 아니라면, 렌더된 이미지를 표현하는 창을 생성할 필요가 있습니다. Windows는 자체 지원(Native) 플랫폼 API나 GLFW와 SDL과 같은 라이브러리를 이용해 생성할 수 있습니다. 우리는 GLFW를 사용할 것입니다, 자세한 내용은 다음 챕터에서 알아볼 것입니다.

우리는 창에 실제로 렌더하기 위한 창 표면(a window surface)(VkSurfaceKHR)과 스왑 체인(a swap chain)(VkSwapChainKHR)라는 2가지 요소가 더 필요합니다. KHR 접두어에 대해서 얘기하자면 이러한 오브젝트들은 Vulkan 확장의 일부라는 뜻을 가지고 있습니다. Vulkan API는 스스로 완벽하게 특정 플랫폼에서 벗어나 있기 때문에 창 관리자(window manager)와 상호 작용하기 위해 표준화된 WSI(Window System Inteface) 확장을 이용해야 하는 이유입니다. 표면은 렌더링할 창들에 대해서 크로스 플랫폼으로 추상화되었으며, 일반적으로 Windows의 HWND와 같이 Native 창 핸들의 참조를 받아 인스턴스화 됩니다. 운 좋게도, GLFW 라이브러리는 이러한 플랫폼 세부 사항을 다루는 내장 함수가 존재합니다.

스왑 체인은 렌더 타겟들의 집합(Collection)입니다. 이것의 기본적인 목적은 현재 화면에서 렌더링 되고 있는 것과 우리가 렌더링 해야 할 이미지와 다른 것인지 확인하는 것입니다. 이는 완벽한 이미지만 보이게 하기 위한 중요한 작업입니다. 프레임을 그릴 때 마다 스왑 체인에게 렌더링할 이미지를 제공받아야 합니다. 프레임을 그리는게 끝냈다면, 특정한 시점에 화면에 표현하기 위해서 이미지를 스왑 체인에게 다시 반환합니다. 렌더 타겟의 수와 완성된 이미지를 화면에 표현하기 위한 조건은 현재 표현 모드(Present Mode)에 따라 결정됩니다. 일반적인 표현 모드는 이중 버퍼링(vsync) 그리고 삼중 버퍼링입니다. 우리는 이것들을 스왑 체인 생성 챕터에서 살펴볼 것입니다.

Step 4 – 이미지 뷰 그리고 프레임버퍼

스왑 체인으로부터 획득한 이미지를 그리기 위해서는 먼저 VkImageView와 VkFramebuffer로 감싸야 합니다. 이미지 뷰는 이미지에서 사용할 특정 부분을 참조하며, 프레임 버퍼는 이미지 뷰에서 사용할 색상, 깊이, 스텐실 타겟에 대해서 참조합니다. 왜냐하면 스왑 체인 내에는 다른 이미지가 많이 존재할 수 있기 때문에, 각각 이미지에 대해서 우선적으로 이미지 뷰와 프레임 버퍼를 생성하고, 이미지를 그릴 때 맞는 것을 선택합니다.

Step 5 – 렌더 패스

Vulkan의 렌더 패스는 렌더링 연산 중 사용하게 될 이미지의 타입을 기술하고, 얼마나 사용될 지, 그리고 어떻게 다루어 져야 할 지 설명합니다. 우리의 삼각형 렌더링 응용 프로그램의 초반에는 Vulkan에게 단일 이미지를 생상 타겟으로써 이용하고, 그리기 연산 전에 단색으로 지울 것을 지정합니다. 렌더 패스는 이미지의 타입만 기술하게 되므로, VkFramebuffer는 실제로 특정 이미지를 이러한 슬롯들에 바인딩하게 될 것입니다.

Step 6 – 그래픽 파이프라인

Vulkan에서의 그래픽 파이프라인은 VkPipeline 객체를 생성 하면서 설정을 합니다. 이는 뷰포트 크기나 깊이 버퍼 연산 그리고 VkShaderModule 객체를 사용한 프로그래밍 가능한 상태(The programmable state)와 같은 그래픽 카드의 설정 가능한 상태들을 기술합니다. VkShaderModule 객체는 셰이더 바이트 코드로 생성됩니다. 또한 드라이버는 렌더 패스로 명시된 파이프라인내에 어떤 어떤 렌더 타겟이 사용될 지 알아야 합니다.

다른 API와 비교해서 Vulkan의 특이한 기능 중 하나는 전문적으로 사용할 때 그래픽 파이프라인 대부분을 직접 구성할 수 있습니다. 이 뜻은 다른 셰이더로 교체하거나 버텍스 레이아웃에 자그만한 변화가 생긴다면, 그래픽 파이프라인을 완전히 새로 생성해야 합니다. 그래서 렌더링에 필요한 모든 조합에 대해서 미리 VkPipeline 객체를 많이 생성해 두어야 합니다. 뷰포트 크기나 초기화 색상과 같은 기초적인 구성들만 동적으로 바꿀 수 있습니다. 또한 모든 상태는 명시적으로 기술되어 있어야 합니다, 예를 들어 색상 혼합 상태는 기본 값이 없습니다.

좋은 소식은 AOT 컴파일은 JIT컴파일에 동등성을 가지는 작업을 수행하며, 다른 그래픽 파이프라인으로 전환하는 것과 같은 큰 상태 변화가 매우 명확하기 때문에, 드라이버에게 더 많은 최적화 기회가 주어지고, 런타임 성능을 더 예측할 수 있게 됩니다.

Step 7 – 명령 풀 그리고 명령 버퍼

이전에 언급했듯이, 그리기 연산과 같은 우리가 실행하려는 Vulkan의 많은 연산들은 큐에 제출할 필요가 있습니다. 이러한 연산들은 첫 째로 큐에 제출하기 전에 VkCommandBuffer에 기록될 필요가 있습니다. 명령 버퍼들은 특정 큐 패밀리와 관련이 있는 VkCommandPool에서 할당이 됩니다. 간단한 삼각형을 그리기 위해선 명령 버퍼에 다음과 같은 연산들을 기록해야 합니다.

* 렌더 패스 시작
* 그래픽 파이프라인 바인드
* 버텍스 3개 그리기
* 렌더 패스 종료

프레임버퍼의 이미지는 스왑 체인에서 제공받은 특정 이미지에 따라 다르기 때문에, 각각의 가능한 이미지를 명령 버퍼에 기록하고, 그릴 때 맞는 것을 선택해야 합니다. 다른 방법으로는 매번마다 명렁 버퍼에 다시 기록하는 것이지만, 효율적이지는 않습니다.

Step 8 – 메인 루프

이제 명령 버퍼에 그리기 명령이 작성되었으므로(have been wrapped) 메인 루프틑 매우 간단합니다. 첫 번째로 vkAcquireNextImageKHR을 이용해 스왑 체인에서 이미지를 취득합니다. 그리고 해당 이미지에 대한 적절한 명령 버퍼를 선택하고 vkQueueSubmit으로 명령을 실행합니다. 마지막으로, 화면에 표현하기 위해서 vkQueuePresentKHR을 이용해 스왑 체인에게 이미지를 반환합니다.

큐에 제출된 연산은 비동기적으로 실행됩니다. 따라서 올바른 실행 순서에 대한 신뢰성을 위해서 세마포어와 같은 동기화 객체를 사용해야 합니다. 그리기 명령 버퍼에 대한 실행은 무조건 이미지 취득 과정이 끝날 때까지 기다려야 합니다. 그렇지 않다면, 화면에 표현하기 위한 이미지를 읽던게 렌더링이 시작되면서 나와버릴 수 있습니다. vkQueuePresentKHR은 렌더링이 완료될 때까지 기다려야 하며, 렌더링이 끝난 후에 끝났다는 신호를 위해서 두 번째 세마포어를 사용할 수 있습니다.

요약

이러한 정보들은 첫 번째 삼각형 그리기 작업 전에 간단한 이해를 위해 작성되었습니다. 실제 프로그램은 버텍스 버퍼 할당, 유니폼 버퍼 생성, 텍스처 이미지 업로딩과 같은 더 많은 단계가 포함되어 있으며, 챕터의 소단계에서 배우게 되지만, Vulkan은 충분히 높은 학습 곡선을 가지고 있기 때문에 간단하게 시작할 것입니다. 우리는 버텍스 버퍼를 사용하는 것 대신 버텍스 셰이더에 버텍스 좌표를 내장시켜 초기화 하는 편법을 이용할 것입니다. 이러한 이유는 버텍스 버퍼를 관리하려면 먼저 명령 버퍼에 대해 어느정도 친숙해져야 하기 때문입니다.

요약해서, 첫 번째 삼각형을 그리기 위해서는…

* VkInstance 생성
* 지원하는 그래픽 카드 선택 (VkPhysicalDevice)
* 그리고, 표현하기 위한 VkDevice와 VkQueue 생성
* 창, 창 표면, 스왑 체인 생성
* 스왑 체인 이미지를 VkImageView에 감싸기(Wrapping 하기)
* 렌더 타겟과 렌더 타겟의 취급법(usage)에 대한 렌더 패스를 생성합니다.
* 렌더 패스를 위한 프레임버퍼 생성
* 그래픽 파이프라인 설정
* 가능한 모든 스왑 체인 이미지를 위한 그리기 명령을 위한 명령 버퍼 할당 및 기록
* 취득한 이미지로 프레임 그리기, 그리기 명령 버퍼 제출, 스왑 체인에 반환하기

정말 많은 단계가 있지만, 각각의 개별 단계는 앞으로의 챕터들에서 매우 간단하고 깔끔하게 이루어질 것입니다. 전체적인 프로그램과 개별 단계와 비교하면서 혼란스럽다면 여기로 다시 돌아와서 참조할 수 있습니다.

API 개념

이번 챕터는 Vulkan이 낮은 수준에서 어떻게 구조화 되어있는지 간단한 소개와 함께 마치겠습니다.

코딩 관습(Convention)

모든 Vulkan 함수들, 열거형들과 구조체들은 LunarG에서 개발한 VulkanSDK의 **vulkan.h**에 정의되어 있습니다. 우리는 다음 챕터에서 VulkanSDK를 설치하면서 살펴볼 것입니다.

함수는 소문자 vk 접두사를 가지며, 열거형 같은 타입과 구조체는 Vk 접두사를 가지고, 열거형 값은 **VK\_** 접두사를 가집니다. API는 함수에 파라미터를 제공하기 위해 구조체를 아주 많이 사용합니다. 예를 들어 객체 생성은 일반적으로 다음 패턴을 따릅니다.



Vulkan의 많은 구조체들은 sType 멤버에 타입을 명시적으로 지정해야 합니다. **pNext** 멤버는 구조체 확장을 지정할 수 있지만, 이 튜토리얼 내에서는 계속 **nullptr**일 것입니다. 객체 생성이나 파괴 함수들은 VkAllocationCallbacks 매개 변수를 통해 드라이버 메모리를 위한 사용자 정의 할당자를 사용할 수 있습니다, 역시 이 튜토리얼에서는 nullptr로 남겨둘 것입니다.

대부분의 함수들은 VK\_SUCCESS나 에러 코드와 같은 VkResult를 반환합니다. 사양서에는 각 함수에 대한 에러코드와 뜻이 설명되어 있습니다.

유효성 검사 레이어

이전에 언급했듯이, Vulkan은 고성능과 적은 드라이버 오버헤드를 위해 설계되었습니다. 따라서 기본적으로 매우 제한적인 에러 확인과 디버깅 성능을 가지고 있습니다. 뭔가 잘 못되거나 잘 못했을 때 드라이버는 에러 코드를 반환하는 것 대신에 자주 충돌이 일어날 것입니다.

Vulkan은 유효성 검사 레이어라는 기능을 이용해 광범위한 검사를 사용할 수 있습니다. 유효성 검사 레이어는 API 사이에 삽입되어 있는 일련의 코드 조각들이고, 그래픽 드라이버는 함수 파라미터에 대한 추가 확인 작업과 메모리 관리 문제 추적과 같은 것들을 수행합니다. 좋은 점은 개발중에 활성화 해두고 릴리즈할 때는 완전히 해제해서 오버헤드를 없앨 수 있습니다. 누구나 자신만의 유효성 검사 레이어를 작성할 수 있지만, LunarG의 VulkanSDK는 이번 튜토리얼에서 사용할 표준 기능 집합 유효성 검사 레이어를 제공합니다. 또한 레이어로부터 디버그 메시지를 받기 위한 콜백 함수를 등록해야 합니다.

Vulkan은 모든 연산에 대해서 매우 명확하고, 유효성 검사 레이어는 광범위하기에, OpenGL이나 Direct3D에 비해서 왜 검은 화면인지 이유를 매우 쉽게 찾을 수 있습니다.

앞으로 우리가 코드를 작성하고 개발 환경을 설정하기에 하나의 단계만 남았습니다.

Chapter 3 개발 환경

이번 챕터에서는 Vulkan 프로그램을 개발하기 위한 환경 설정과 몇 가지 아름다운 라이브러리들을 설치할 것입니다. 컴파일러를 제외하고는 사용할 모든 툴들은 Windows, Linux, MacOS간에 호환이 됩니다. 하지만 각 설치 단계는 조금씩 다르기 때문에 여기서 나눠서 설명을 합니다.

Windows

Windows에서 개발한다면, 코드를 컴파일 하기 위해 Visual Studio 2017을 사용할 것으로 생각됩니다. 만약에 Visual Studio 2013, 2015 버전을 사용한다면 조금 과정이 다를 수 있습니다.

Vulkan SDK

Vulkan 프로그램을 개발하면서 가장 중요한 구성 요소는 SDK입니다. SDK는 헤더 파일과, 표준 유효성 검사 레이어, 디버깅 툴 그리고 Vulkan 함수 로더가 포함되어 있습니다. 로더는 런타임에 드라이버에서 함수들을 로드합니다(looks up). OpenGL의 GLEW와 친숙하다면 비슷하다고 느끼실수 있습니다.

SDK는 LunarG 사이트에서 다운로드할 수 있습니다. 계정을 생성하지 않아도 유용한 일부 추가 문서에 접근할 수 있습니다.

설치를 계속 진행하고, 설치 폴더에 주의하세요. 첫 번째로 그래픽 카드와 그래픽 드라이버가 제대로 Vulkan을 지원하는지 검증해야 합니다. SDK가 설치된 폴더로 가셔서, **Bin** 폴더를 연 후 **cube.exe** 데모를 실행하세요. 다음과 같이 되면 됩니다.

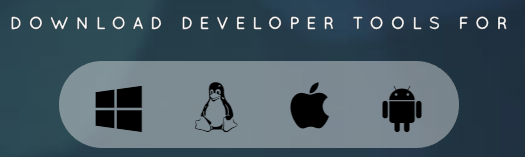


그림 3.1

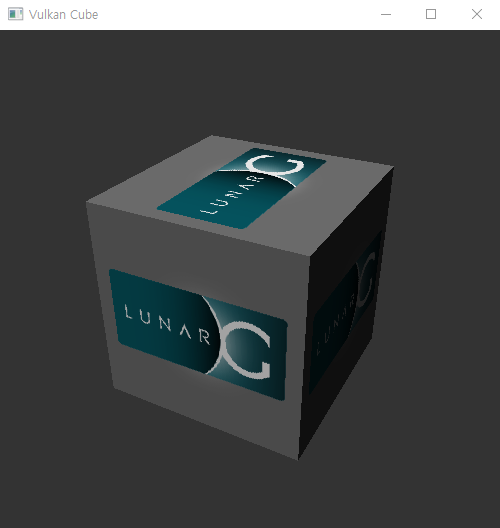


그림 3.2

만약 에러 메시지를 전달받았다면, Vulkan 런타임을 포함한 드라이버가 최신인지, 지원되는 그래픽 카드인지 확인하세요. 주요 드라이버 링크는 소개 챕터를 확인하세요.

폴더 안에는 개발을 위한 유용한 프로그램이 더 있습니다. **glslangValidator.exe** 프로그램은 사람이 읽을 수 있는 GLSL 셰이더 코드를 컴파일 해 바이트 코드로 변환시킵니다. 이는 셰이더 모듈 챕터에서 더 자세히 알아볼 것입니다. **Bin** 폴더는 Vulkan 로더와 유효성 레이어의 바이너리가, **Lib** 폴더에는 그 라이브러리들이 포함되어 있습니다. **Doc** 폴더는 Vulkan SDK에 대한 유용한 정보와 오프라인 버전의 완전한 Vulkan 사양서가 포함되어 있습니다. 마지막으로 **Include** 폴더는Vulkan의헤더가있습니다. 자유롭게 다른 파일들을 탐험할 수 있지만, 이 튜토리얼에서는 그러지 않을 것입니다.

GLFW

이전에 언급했듯이 Vulkan은 플랫폼 독립적 API이기에 결과를 화면에 렌더링하기 위한 창을 생성하는 도구는 포함되어 있지 않습니다. 우리는 Win32와 같은 플랫폼 종속을 피하고, Vulkan의 크로스 플랫폼의 이점을 취하기 위해서 Windows, Linux, MacOS를 지원하는 GLFW 라이브러리를 이용해 창을 생성할 것입니다. 이러한 목적을 위해 SDL과 같은 다른 라이브러리를 사용할 수 있지만, 창 생성 이외에도 GLFW의 이점은 다른 Vulkan의 플랫폼 종속적인 것들을 추상화한다는 것입니다.

GLFW 공식 웹 사이트에서 최신 릴리즈 버전을 찾을 수 있습니다. 이 튜토리얼 에서는 64bit 바이너리를 사용할 것이지만, 32bit 모드로 빌드할 수도 있습니다. 이러한 경우는 **Lib** 폴더 대신 **Lib32** 폴더를 링크해야 합니다. 다운로드가 끝난 후 편한 위치에 압축을 풀어주세요. 저는 Visual Stdio 폴더 아래에 **Libraries** 폴더를 생성할 것입니다. 거기에 **libvc-2017** 폴더가 없어도 걱정하지 마세요. **Libvc-2015** 도 호환됩니다.

GLM

DirectX 12와 다르게 Vulkan은 선형 대수 연산 라이브러리를 포함하지 않기에 직접 다운로드 해야합니다. GLM은 그래픽 API와 같이 쓰이기 위해 디자인된 좋은 라이브러리입니다. OpenGL에서도 일반적으로 사용됩니다.

GLM은 헤더만 존재하는 라이브러리입니다. 최신 버전을 다운로드해서 편한 위치에 저장하면 됩니다. 다음과 같은 폴더 구조를 따라하셔도 됩니다.

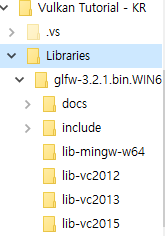


그림 3.3

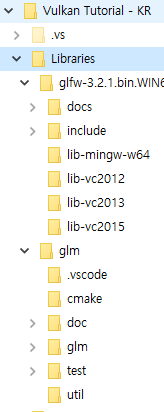


그림 3.4

Visual Studio 설정하기

이제 모든 종속 요소들을 설치했으니, Vulkan을 위한 기본적인 Visual Studio 프로젝트를 설정할 수 있습니다. 그리고 모든 게 잘 작동되도록 약간의 코드도 작성하겠습니다.

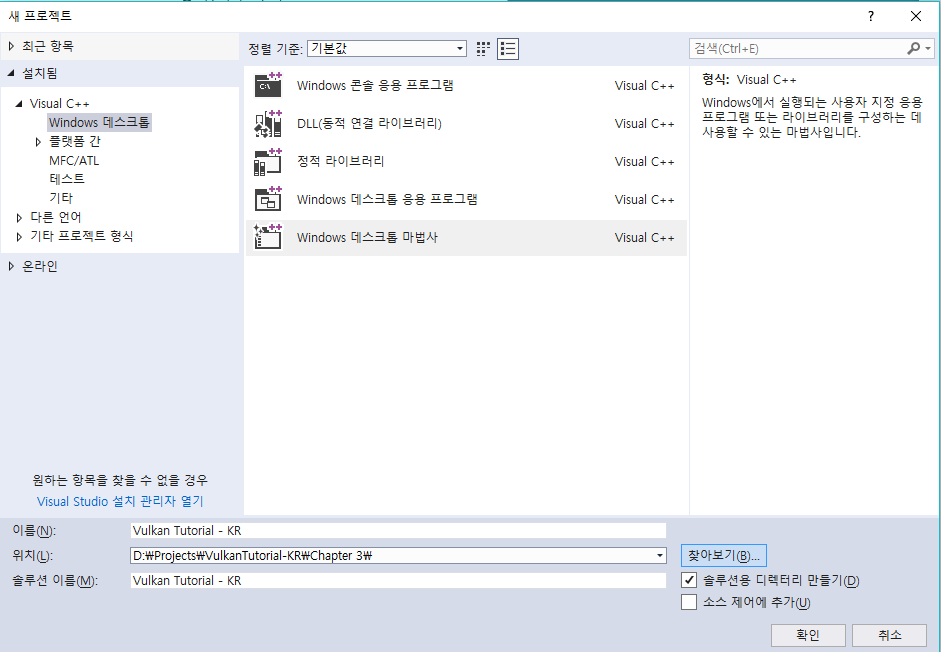


그림 3.5

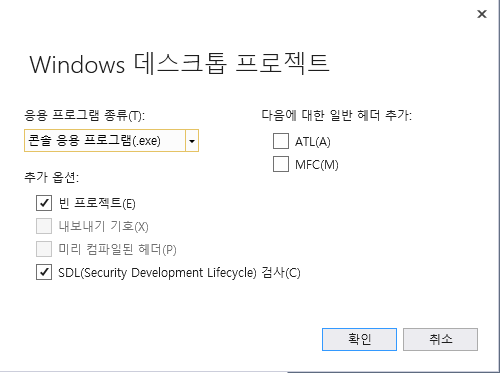


그림 3.6

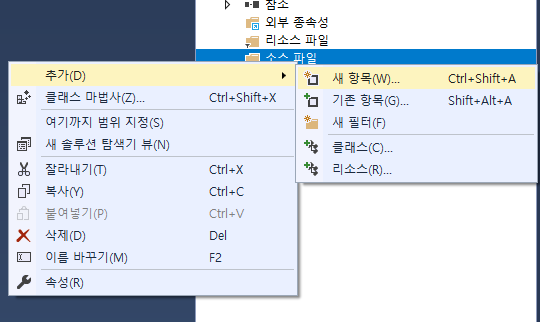
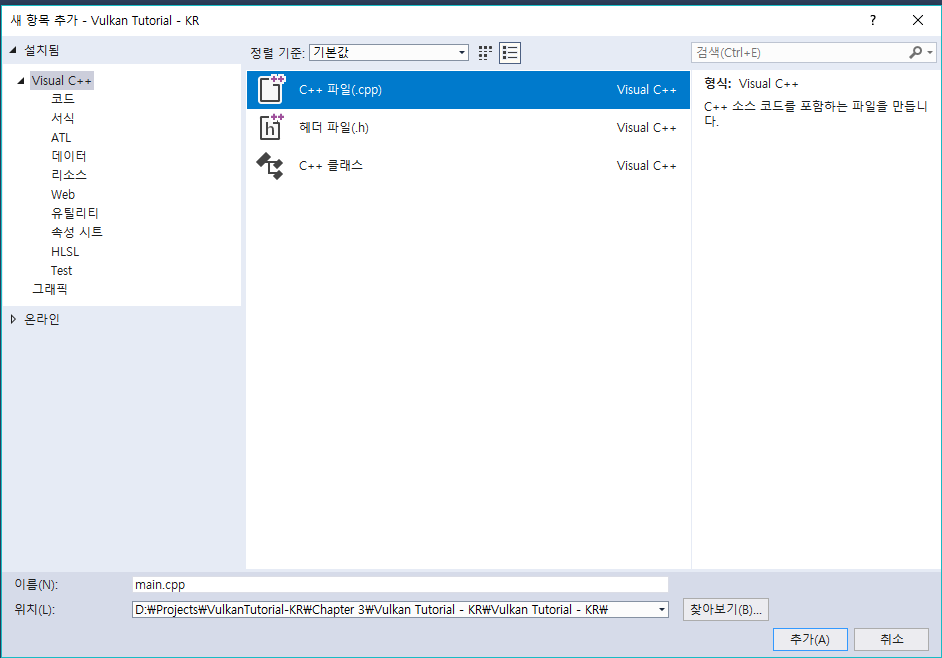


그림 3.7

그림 3.8

디버그 메시지를 출력하기 위해서 **콘솔 응용 프로그램(.exe)**를 선택합니다. 그리고 Visual Studio에서 특수 코드들을 추가하지 않도록 **빈 프로젝트**를 체크합니다.

**확인**을 눌러 프로젝트를 생성하고 C++ 소스 파일을 추가합니다. 당신이 이미 어떻게 하는지 알고 있어도, 완벽함을 위한 단계가 포함되어 있습니다.

이제 파일에 코드를 따라 칩니다. 굳이 이해하려고 노력하지 않으셔도 됩니다. 그냥 Vulkan 프로그램이 잘 작동되는지 확인하는 것입니다. 우리는 다음 챕터에서 간단하게 시작할 것입니다.



이제 프로젝트 설정을 통해서 에러들을 없에보겠습니다. 프로젝트 속성 창을 열고 설정을 **Debug**와 **Release** 둘다 적용하기 위해 **모든 구성**을 선택합니다.



그림 3.9

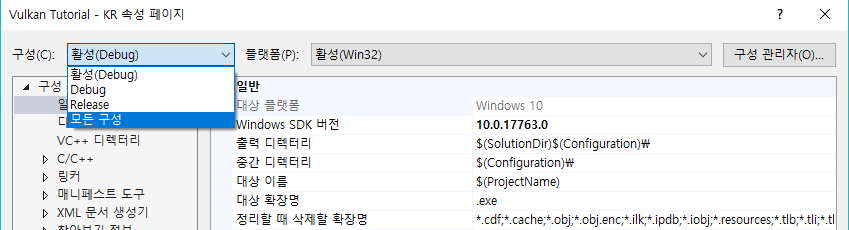


그림 3.10

**C++ -> 일반 -> 추가 포함 디렉터리**로 이동해서 **<편집…>** 드롭다운을 누릅니다. 그리고 자신이 압축해제한 Vulkan, GLFW, GLM 헤더 폴더를 지정합니다.

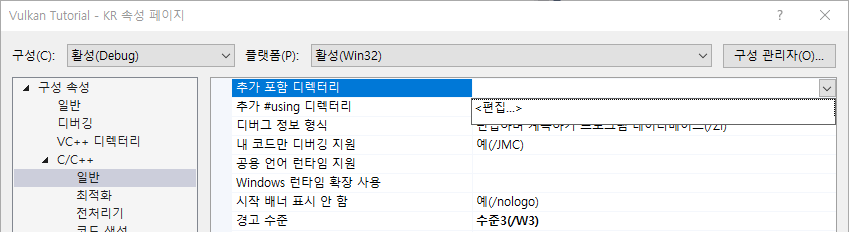


그림 3.11

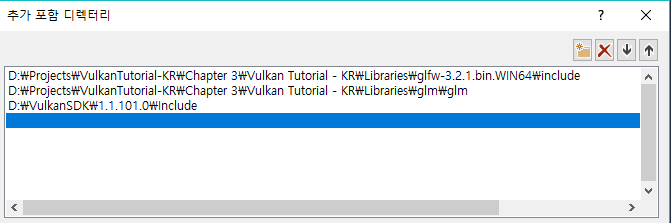


그림 3.12

다음으로 라이브러리 폴더를 지정하기 위해 **링커 -> 일반**으로 이동해 Vulkan과 GLFW의 오브젝트 파일 폴더(Lib, Library 폴더)를 지정합니다.

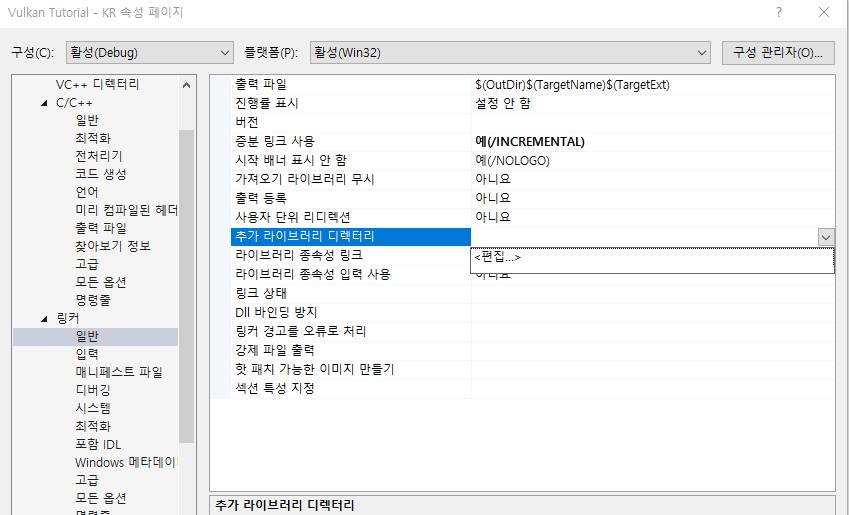
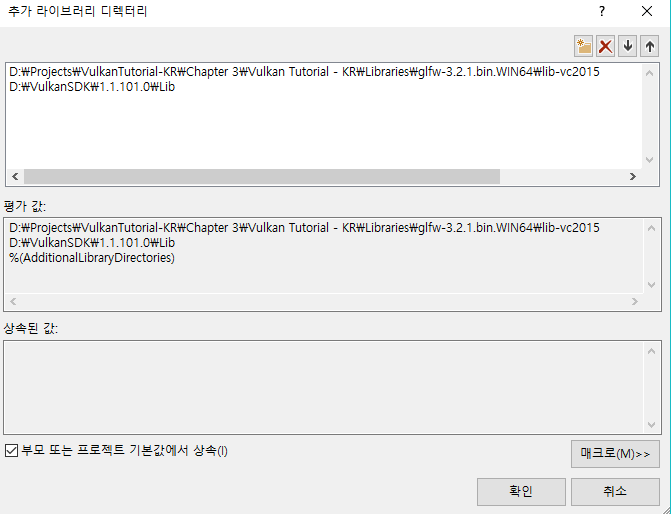
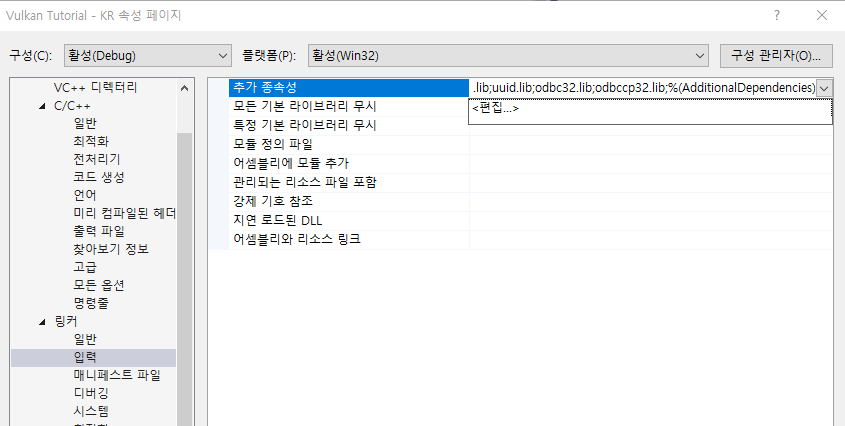


그림 3.13



**링커 -> 입력 -> 추가 종속성**의 **<편집…>** 버튼을 누릅니다.

그림 3.14



Vulkan과 GLFW의 오브젝트 파일(lib 파일, obj 파일)의 이름을 입력합니다. (Windows 경우 다음과 같이 입력).

그림 3.15

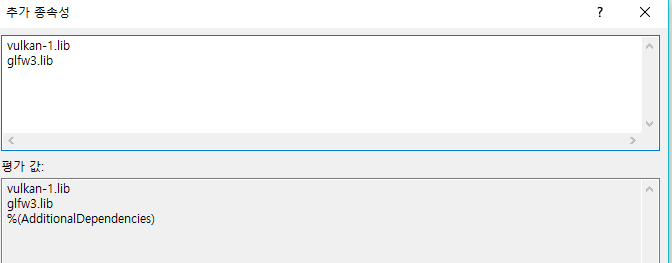


그림 3.16

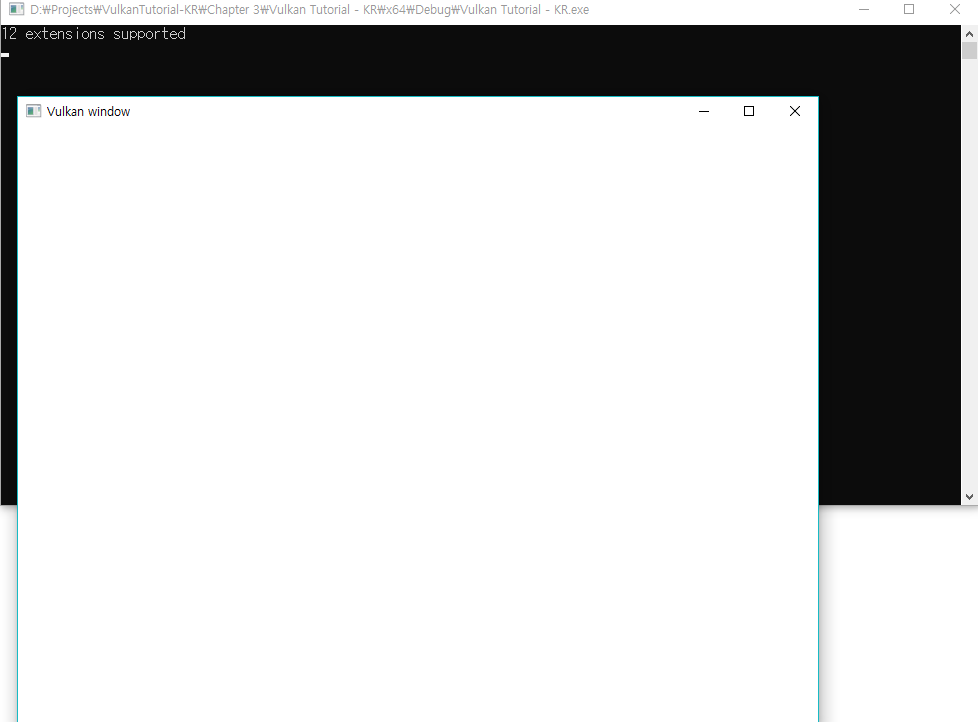
프로젝트 속성 창을 닫고 나면 코드에 에러 하이라이트가 바로 없어질 것입니다.

마지막으로 64비트 모드로 컴파일 합니다.



그림 3.17

**F5**를 눌러 프로젝트를 실행시키면 다음과 같이 팝업창과 함께 CMD 창이 뜰 것입니다.



확장의 수가 0이 아니여야 합니다. 축하합니다! 이제 Vulkan을 가지고 재미있게 놀 수 있게되었습니다!

Linux

Vulkan SDK

GLFW

GLM

Makefile 프로젝트 설정하기

MacOS

Vulkan SDK

GLFW

GLM

Xcode 설정하기

Chapter 4 기본 코드

전반적인 구조

이전 챕터에서 모든 적절한 설정과 간단하게 테스트한 샘플 코드를 가진 Vulkan 프로젝트를 생성했습니다. 이번 챕터는 다음 코드와 함께 밑바닥에서부터 시작할 것입니다.



첫 번째로 함수, 구조체, 열거자들이 제공되는 LunarG SDK의 Vulkan 헤더를 포함시켰습니다. **stdexcept와 iostream** 헤더는 전파되는 에러들을 보고하기 위해 포함되었습니다. **fuctional** 헤더는 자원 관리 섹션에서 람다 함수로써 사용될 것입니다. **cstdlib** 헤더는 **EXIT\_SUCCESS**와 **EXIT\_FAILURE** 매크로를 제공합니다.

프로그램 자체는 Vulkan 객체를 private 클래스 멤버로 저장하고 각각 초기화 하는 함수를 추가하는 클래스로 래핑됩니다. 이는 **InitVulkan** 함수에서 호출됩니다. 모든게 준비되고 나면 메인 루프에 프레임 렌더링을 위해 진입합니다. **MainLoop** 함수는 창이 닫히는 순간까지 루프를 돌게 됩니다. 창이 닫히고, **MainLoop** 함수가 반환될 때, **CleanUp** 함수를 이용해 자원들을 할당 해제할 것입니다.

만약 아무 종류의 치명적인 에러가 실행중에 나타나면, CMD창에 출력하기 위해 **main** 함수로 이유 메시지와 함께 **std::runtime\_error** 예외가 전파됩니다. 다양한 표준 예외 타입을 처리하기 위해 std::exception로 캐치하게 됩니다. 예를 하나 들면 특정 필수 확장이 지원되지 않는 다는 것을 찾아내는 것입니다.

간단하게 모든 챕터는 함수를 추가하고 **InitVulkan**에서 호출하고, 새로운 Vulkan 객체를 private 클래스 멤버에 추가하고, 그리고 필요할 경우 **CleanUp**에서 초기화 하는 과정을 거칩니다.

자원 관리

**malloc**을 통해 메모리 덩어리를 할당하고 **free**를 호출해주어야 하는 것과 같이, 모든 Vulkan 객체는 생성하고 난 후 사용하지 않는 다면 명확하게 파괴해주어야 합니다. 현대(Modern) C++ 코드는 **<memory>**헤더 안의 자동 자원 관리 유틸리티를 이용할 수도 있지만, 이 튜토리얼에서는 Vulkan 객체의 할당, 해제를 명확하게 작성하는 방법을 택하겠습니다. 명확하게 하는 게 모든 작업에서 실수를 피하기 좋고, 직접 생명 주기를 명시적으로 하므로 API를 배우기에는 좋습니다.

튜토리얼을 끝낸 후 **std::shared\_ptr**과 같은 걸로 오버로딩하여 자동 자원 관리를 구현하실 수도 있습니다. RAII를 사용하시는 게 큰 Vulkan 프로그램에서의 접근성에서 더 이점이 있으나, 학습적으로는 뒤에서 어떤 작업이 일어나고 있는지 항상 알고 계시는게 좋습니다.

Vulkan 객체는 **vkCreateXXX**와 같은 함수로 직접 생성할 수 있고, 또 다른 객체의 경우 **vkAllocateXXX**와 같은 함수로 할당할 수 있습니다. 만든 후에 객체가 더 이상 어디에도 사용되고 있지 않다면, **vkDestroyXXX**와 **vkFreeXXX**와 짝지어서 사용하여 파괴하면 됩니다. 이러한 함수들은 각 객체의 타입마다 다른 매개 변수를 가지고 있으나, **pAllocator** 매개 변수만은 모두 공유하고 있습니다. 이는 사용자 지정 메모리 할당자를 직접 지정할 수 있는 선택적 매개 변수입니다. 이 튜토리얼에서는 항상 nullptr으로 지정하여 무시하겠습니다.

GLFW 통합하기

화면에 렌더링하지 않으려고 한다면, Vulkan은 창 생성을 하지 않아도 완벽하게 작동합니다. 하지만 뭔가 보인다면 더 재미있을거에요!

먼저 **#include <vulkan/vulkan.h>** 구문을 다음과 같이 바꿉니다.



이러한 방법은 GLFW가 자체 정의를 포함(include)하고 Vulkan 헤더를 자동으로 포함(include)할 것입니다. **InitWindow** 함수를 추가하고 **Run**함수에서 다른 호출 구문 전에 호출합니다. 우리는 이 함수들로 GLFW를 초기화 하고 창 생성을 할 것입니다.



GLFW 라이브러리를 초기화 하는 **glfwInit()** 함수를 **InitWindow** 함수에서 맨 처음에서 호출되어야 합니다. GLFW는 원초적으로 OpenGL 컨텍스트(Context)를 생성하기 위해 설계되었기 때문에, 우리는 다음 호출로 OpenGL 컨텍스트를 생성하지 말라고 말해야 합니다.



창 리사이즈(resize)를 처리하는 것은 나중에 살펴볼 것이므로, 지금은 윈도우 힌트 호출로 비활성화 합니다.



이제 실제 창을 생성하는 일만 남았습니다. 참조(reference)를 저장하기 위해 **GLFWwindow\* window;** 구문을 private 클래스 멤버 변수에 추가하고, 다음과 같이 창을 초기화 합니다.

****

앞의 3개의 매개 변수는 너비와 높이, 창의 제목을 지정합니다. 4번째 매개 변수는 창을 열 모니터를 선택적으로 지정할 수 있고, 마지막 매개 변수는 OpenGL과 관련된 값입니다.

너비와 높이 값은 앞으로도 여러 번 참조하기 때문에, 리터럴 숫자 대신 하드코딩된 상수를 사용하는 게 좋습니다. 저는 HelloTriangleApplication 클래스 정의 위에 다음 구문을 추가했습니다.



그리고 창 생성 호출을 다음과 같이 바꿉니다.



이제 InitWindow 함수는 다음과 같이 보일 것입니다.



프로그램이 에러가 나타나거나 창이 닫힐 때까지 계속 작동해야 하므로, **MainLoop** 함수에 다음과 같이 이벤트 루프를 추가합니다.



이 코드는 상당히 이해하기 쉬우실 겁니다. 계속 반복하면서, X 버튼을 누르거나 유저에 의해 창이 닫히는 이벤트같은 걸 체크합니다. 여기서는 나중에 추가할 단일 프레임 렌더 함수도 호출됩니다.

창이 닫힐 때, 자원들을 파괴해서 정리합니다. 그리고 GLFW를 종료시킵니다. **CleanUp**코드는 다음과 같이 되야합니다.



이제 프로그램이 실행되고 프로그램이 종료되고 창이 닫히기 전까지 **Vulkan** 제목을 가진 창이 보일 겁니다. 이제 Vulkan 프로그램의 뼈대를 만들었으니, 첫 번째 Vulkan 객체를 만들어 봅시다!

[C++ Code](https://vulkan-tutorial.com/code/00_base_code.cpp) (원작자 코드 링크)

(Chapter 4 폴더의 프로젝트를 실행하시고, Vulkan SDK의 헤더 파일과 라이브러리 폴더를 링크 하시고 실행해도 동일한 결과가 출력됩니다.)

Chapter 5 인스턴스

인스턴스 생성하기

첫 번째 필요한 것은 instance를 생성함으로써 Vulkan 라이브러리를 초기화하는 것입니다. 인스턴스는 당신의 프로그램과 Vulkan 라이브러리 사이에서 연결이며, 이를 생성하기 위해서는 프로그램에 대한 세부 사항들을 드라이버에 지정해야 합니다.

**CreateInstance** 함수를 추가하고 **InitVulkan** 함수에서 호출함으로써 시작하겠습니다.



추가로 인스턴스를 조작할 핸들을 클래스 멤버에 추가합니다.



이제 인스턴스를 생성하기 위해 우리 프로그램에 대한 정보를 구조체에 채워야 합니다. 이 데이터들은 기술적으로 선택적이지만, 특정 특수 동작을 하는 잘 알려진 그래픽 엔진이 사용하기 때문에, 프로그램을 최적화하는데 유용한 정보를 제공할 수도 있습니다. 이 구조체의 이름은 **VkApplicationInfo**입니다.



이전에 언급했듯이, Vulkan의 많은 구조체들은 **sType**멤버에 명확하게 타입을 지정해야합니다. 또한 **pNext** 멤버를 가지고 있는 구조체들은 나중에 확장에 대한정보를 지정할 수 있습니다. 우리는 기본 초기화를 이용해 **nullptr**로 내버려 두겠습니다.

많은 Vulkan의 정보들은 함수 매개 변수 대신 구조체를 거지게 됩니다. 그리고 인스턴스를 생성할 때 충분한 청보를 제공하기 위해 구조체를 하나 더 채우겠습니다. 다음 구조체는 필수이며, Vulkan 드라이버에게 어떤 전역 확장 및 유효성 레이어를 사용할 것인지 알려줍니다. 여기서의 전역의 의미는 모든 프로그램에 적용되고, 특정 드라이버만 적용되는게 아닙니다. 다음 몇 챕터 내로 깔끔하게 이해가 되실겁니다.



앞의 두 매개 변수는 매우 직관적입니다. 다음으로는 원하는 전역 확장을 지정합니다. 개요 챕터에서 언급했듯이 Vulkan은 플랫폼에 비종속적인 API입니다. 이 뜻은 창 시스템 인터페이스 확장이 필요하다는 것입니다. GLFW는 구조체에 전달할 수 있는 확장 기능을 반환하는 편리한 내장 함수가 있습니다.



남은 두 구조체 멤버는 사용할 전역 유효성 레이어를 결정합니다. 이것에 대해서는 다음 챕터에서 더 깊게 얘기하도록 하겠습니다, 그러므로 지금은 비워두겠습니다.



이제 인스턴스를 생성하기 위한 모든 것을 지정했으니, **vkCreateInstance** 호출을 합니다.



보시다시피, Vulkan 객체 생성 함수 매개변수가 따르는 일반적인 패턴은 다음과 같습니다.

* create info의 구조체 포인터
* 사용자 지정 할당자 콜백, 이 튜토리얼에서는 항상 nullptr
* 새 객체를 다루고 저장할 포인터 변수

모든 것이 잘 진행되었으면 인스턴스 핸들은 **VkInstance** 클래스 멤버에 저장됩니다. 거의 모든 Vulkan 함수는 VK\_SUCCESS나 에러 코드같은 **VkResult** 타입을 반환 합니다.인스턴스가 성공적으로 생성되었는지 확인하기 위해, 결과를 저장하지 않고 다음과 같이 성공 값을 확인하면 됩니다.



이제 프로그램을 실행하면 인스턴스가 성공적으로 생성됩니다.

확장 지원 확인

만약에 vkCreateInstance 문서를 확인하셨다면 가능한 에러 코드중 VK\_ERROR\_EXTENSION\_NOT\_PRESENT 라는 에러를 확인하실 수 있습니다. 이런 에러 코드가 나타나면 간단하게 우리가 필요한 확장을 지정하고 끝낼 수 있습니다. 창 시스템 인터페이스와 같은 필수 확장에 대해서는 의미가 있지만, 다른 옵션 기능을 확인하려면 어떻게 해야할까요?

인스턴스를 생성하기 전에 지원하는 확장을 검색하려면, **vkEnumerateInstanceExtensionProperties** 함수가 있습니다. 이는 확장 개수의 변수가 저장된 포인터와 확장의 세부 정도가 담긴 **VkExtensionProperties** 배열을 받을 수 있습니다. 첫 번째 매개 변수에 특정 유효성 레이어에 의한 확장을 필터링할 수 있지만, 지금은 무시하겠습니다.

확장 세부 정보들을 저장하기 위한 배열을 할당하기 위해서는, 확장이 먼저 얼마나 있는지 알아야합니다. 뒤의 매개 변수를 빈칸으로 두면서 확장의 수를 요청할 수 있습니다.



이제 확장 세부 정보들을 저장할 배열을 할당할 수 있습니다. (**<vector>** 헤더파일 include)



마지막으로 확장 세부 정보들을 질의할 수 있습니다.



각각의 VkExtensionProperties 구조체는 확장의 이름과 버전이 담겨져있습니다. 간단한 루프를 이용해 조회해보겠습니다. (\t는 tap(들여쓰기) 입니다)



Vulkan 지원에 대한 세부 정보들을 제공하려고 하면 이 코드를 **CreateInstance** 함수에 추가할 수 있습니다. 한 번 **glfwGetRequiredInstanceExtensions**로 반환된 모든 확장이 **vkEnumerateInstanceExtensionProperties** 로 획득한 지원되는 확장 목록에 있는지 확인하는 함수를 만들어보세요.

정리

VkInstance는 프로그램이 종료되기 전에 반드시 파괴되어야 합니다. **vkDestoryInstance** 함수를 이용해 **CleanUp**에서 정리할 수 있습니다.



vkDestroyInstance 에서 사용되는 매개 변수는 매우 직관적입니다. 이전 챕터에서 언급했듯이, 할당 함수와 할당 해제 함수가 있는 Vulkan에는 선택적 할당자 콜백이 있습니다. 우리는 그것을 nullptr로 무시하고 있습니다. 모든 다른 Vulkan의 자원은 챕터를 따라서 생성하고, 인스턴스가 파괴되기 전에 정리되어야 합니다.

인스턴스 생성 이후 더 복잡한 단계들을 진행하기 전에, 유효성 검사를 통해 디버깅 옵션을 작동할 것입니다.

[C++ code](https://vulkan-tutorial.com/code/01_instance_creation.cpp)

Chapter 6 유효성 레이어

유효성 레이어가 무엇인가?

Vulkan API는 최소한의 드라이버 오버헤드라는 아이디어로 설계가 되었고, 이러한 목표를 위한 현상 중 하나는 매우 제한적인 에러 확인이 기본적으로 설정되어 있다는 것입니다. 심지어 올바르지 않은 열거형을 설정하거나, 매개 변수에 nullptr를 전달하지 않는것과 같이 간단한 것도 명시적이지 않으면 충돌이 일어나거나 정의되지 않은 작동이 일어날 수 있습니다. Vulkan은 당신이 하려는 거의 모든 것에 대해서 매우 명시적이기 때문에, 새로운 GPU 기능을 사용하거나, 논리 장치 생성 때 새로운 기능을 요청하지 않는 등 아주 많은 작은 실수들을 만들기 쉽습니다.

하지만 API에 이러한 확인 기능들을 추가할 수 없다는 뜻은 아닙니다. Vulkan은 유효성 레이어라는 아름다운 시스템을 추가했습니다. 유효성 레이어는 추가적인 연산을 통해 Vulkan의 함수를 훅킹(Hook)하는 선택적 요소입니다. 유효성 레이어의 일반적인 연산은 다음과 같습니다.

* 사양서와 대조적으로 잘못되게 사용된 매개 변수 값
* 리소스 누수를 찾기 위한, 객체의 생성, 파괴 추적
* 호출이 발생된 쓰레드를 추적하여 쓰레드 안전성 확인
* 모든 호출과 그 매개 변수들을 표준 출력을 통해서 로그 출력
* 프로파일링을 위한 Vulkan 호출 트레이싱과 리플레이

다음은 매개 변수를 확인(diagnostics)하는 유효성 검사 레이어에서 함수 구현이 어떻게 생겼는지에 대한 예제입니다.



이러한 유효성 검사 레이어들은 당신이 원하는 모든 디버깅 기능들을 포함하도록 자유롭게 쌓이게 됩니다. 디버깅 빌드 때 간단하게 유효성 레이어를 활성화하고, 릴리즈 빌드 때 완전하게 비활성화 할 수 있습니다.

Vulkan은 아무런 내장 유효성 레이어를 제공하지 않으나, LunarG Vulkan SDK는 일반적인 에러 체크를 위한 좋은 레이어 집합을 제공하고 있습니다. 이것들은 완벽하게 오픈 소스이기 때문에, 직접 문제들을 확인하고, 직접 기여할 수 있습니다. 유효성 레이어를 사용하는 게 비정상적인 행동을 하는 다른 드라이버들에 의해 프로그램이 중지되는 것을 피할 수 있는 좋은 방법입니다.

유효성 레이어는 설치된 시스템에서만 사용할 수 있습니다. 예를 들어 LunarG의 유효성 레이어는 Vulkan SDK가 설치되어 있는 PC에서만 존재합니다.

Vulkan에는 일반적으로 두 타입의 유효성 레이어가 존재합니다. 하나는 인스턴스 레이어, 나머지 하나는 장치 특정(device specifie) 레이어입니다. 인스턴스 레이어는 인스턴스와 같은 전역 Vulkan 객체와 관련된 호출만 검사하는 방식입니다. 그리고 장치 특정 레이어는 특정 GPU와 관련된 모든 호출을 검사합니다. 장치 특정 레이어는 이제 더 이상 사용되지 않을 것이고(deprecated), 인스턴스 유효성 레이어가 모든 Vulkan 호출을 확인하게 됩니다. 사양서 문서에서는 일부 구현에 필요하기 때문에 여전히 호환성을 위해 장치 레벨의 유효성 레이어를 활성화하는 것을 권장하고 있습니다. 우리는 나중에 간단히 논리적 장치 수준에서의 인스턴스와 동일한 레이어를 지정할 것입니다.

유효성 레이어 사용하기

이번 섹션에서는 Vulkan SDK에서 제공되는 표준 진단 레이어를 활성화하는 방법을 알아볼것입니다. 확장때와 같이, 레이어의 이름을 직접 지정하면서 유효성 레이어를 활성화할 필요가 있습니다. 유용한 모든 레이어를 명시적으로 직접 지정하는 것 대신에, SDK의 **VK\_LAYER\_LUNARG\_standard\_validation** 레이어를 활성화하여 암시적으로 유용한 모든 진단 레이어를 활성화할 수 있습니다.

첫 번째로 두 구성 변수를 프로그램에 추가하여, 활성화할 레이어와 활성화 여부를 지정합니다. 여기서는 디버그 모드로 컴파일 됐는지, 아닌지 확인하는 기본적인 변수를 사용합니다. **NDEBUG**매크로는C++의 표준 매크로로 디버그 모드가 아님을 뜻합니다.



우리는 **CheckValidationLayerSupport** 함수를 추가하여 요청된 존재하는 모든 레이어를 확인합니다. 첫 번째로 **vkEnumerateInstanceLayerProperties** 함수를 이용해 존재하는 모든 레이어 리스트를 받아옵니다. 사용법은 인스턴스 생성 챕터에서 사용된 **vkEnumerateInstanceExtensionProperties**와 동일합니다.



다음으로, **validationLayers**안에 있는 모든 레이어가 **availableLayers** 리스트에 존재하는지 확인해야 합니다. **strcmp** 함수를 사용하기 위해 **<cstring>**을 포함시켜야 합니다.



이제 **CreateInstance** 함수에서 사용할 수 있습니다.



이제 프로그램을 디버그 모드에서 실행시키면, 에러가 나타나지 않아야 합니다. 만약 에러가 나타난다면, Vulkan SDK가 제대로 설치되었는지 확인하세요. 레이어가 하나도 없거나, 보여지는 레이어 수가 매우 적다면, 이슈를 확인해서 문제를 해결할 수 있습니다(LunarG 계정 필요). 이를 해결하는 데 도움을 받을 수 있을겁니다.

마지막으로 VkInstanceCreateInfo 구조체를 수정해서 유효성 레이어를 활성화 하기 위해 이름을 포함시킵니다.



이제 **VK\_ERROR\_LAYER\_NOT\_PRESENT** 에러 없이 **vkCreateInstance**가 성공적으로 수행되는지 확인하세요.

메시지 콜백

불행히도, 디버그 메시지를 우리의 프로그램에게 돌려줄 방법이 지금은 없기 때문에, 레이어를 그냥 활성화만 해서는 별로 도움이 안됩니다. 이러한 메시지를 받기 위해서는 콜백을 설정해야 합니다. 이는 **VK\_EXT\_debug\_utils** 확장이 요구됩니다.

첫 번째로 유효성 레이어가 활성화됐는지, 아닌지에 대해서를 기반으로 요구되는 확장의 리스트를 반환하는 **getRequiredExtensions** 함수를 만들겠습니다.



GLFW로 지정되는 확장들은 항상 요청되고 있습니다, 하지만 디버그 리포트 확장은 조건에 따라 추가됩니다. 오타를 피하기 위해서 VK\_EXT\_debug\_utils와 같은 리터럴 문자열 매크로인 **VK\_EXT\_DEBUG\_UTILS\_EXTENSIONS\_NAME**을 이용했습니다.

이제 **CreateInstance** 함수에서 쓸 수 있습니다.



프로그램을 실행하면 VK\_ERROR\_EXTENSION\_NOT\_PRESENT 에러가 뜨지 않아야 합니다. 이 확장에 대해서 딱히 확인할 필요가 없는 게, 유효성 레이어의 존재성으로 인해 암시적으로 되고 있기 때문입니다.

이제 콜백 함수를 보겠습니다. **DebugCallback**함수와 **PFN\_vkDebugUtilsMessengerCallbackEXT** 프로토타입과 함께 새로운 static 멤버 함수를 추가합니다. **VKAPT\_ATTR**과 **VKAPI\_CALL**은 Vulkan이 호출할 수 있게 하는 시그니처입니다.



첫 번째 매개 변수는 메시지의 심각성을 나타냅니다. 다음 플래그중 하나를 따릅니다.

* VK\_DEBUG\_UTILS\_MESSAGE\_SEVERITY\_VERBOSE\_BIT\_EXT : 진단 메시지
* VK\_DEBUG\_UTILS\_MESSAGE\_SEVERITY\_INFO\_BIT\_EXT : 자원 생성과 같은 정보적인 메시지
* VK\_DEBUG\_UTILS\_MESSAGE\_SEVERITY\_WARNING\_BIT\_EXT : 무조건 오류는 아니지만 프로그램의 버그일 수도 있는 메시지
* VK\_DEBUG\_UTILS\_MESSAGE\_SEVERTITY\_ERROR\_BIT\_EXT : 올바르지 않은 행동과 충돌을 야기할 수 있는 행동에 대한 메시지

이 열거형 값들은 비교 연산자를 이용하여 직접 메시지를 나타나게 하거나 등 설정할 수 있습니다.



**messageType**은 매개 변수는 다음 값을 따릅니다.

* VK\_DEBUG\_UTILS\_MESSAGE\_TYPE\_GENERAL\_BIT\_EXT : 사양서나 성능과는 무관한 어떠한 이벤트가 발생했을 때
* VK\_DEBUG\_UTILS\_MESSAGE\_TYPE\_VALIDATION\_BIT\_EXT : 사양서에 위반되거나 실수를 했을 때.
* VK\_DEBUG\_UTILS\_MESSAGE\_TYPE\_PERFORMANCE\_BIT\_EXT : 비최적화적인 사용의 가능성

pCallbackData 매개 변수는 VkDebugUtilsMessengerCallbackDataEXT 구조체를 가리키며, 메시지의 세부 사항을 가지는 중요한 멤버를 가지고 있습니다.

* **pMessage** : null 종료 문자열의 디버그 메시지
* **pObjects** : 메시지와 관련된 Vulkan 객체 배열
* **objectCount** : 배열안의 객체의 수

마지막으로, **pUserData** 매개변수는 설정할 때 지정한 콜백 설정중에 지정된 포인터가 포함되어 있어 자신의 데이터를 전달할 수 있습니다.

콜백의 Boolean의 반환값은 유효성 검사 메시지에 동작된 Vulkan 호출이 중단되어야 하는지 나타냅니다. 콜백 반환이 true라면, 호출은 중단되고 VK\_ERROR\_VALIDATION\_FAILED\_EXT 에러를 출력합니다. 이는 보통 유효성 레이어들을 테스트할 때만 사용됩니다, 따라서 항상 VK\_FALSE를 반환합니다.

이제 남은 것은 Vulkan에게 콜백 함수에 대해서 알려주는 것입니다. 약간 놀랄 수도 있지만, Vulkan은 디버그 콜백또한 명확하게 생성과 파괴를 할 필요가 있습니다. 콜백은 messenger라고 부르며, 원하면 더 많이 생성할 수도 있습니다. instance 아래에 이 핸들을 클레스 멤버로서 추가합니다.



이제 새로운 함수인 **SetupDebugCallback** 함수를 만들고 **InitVulkan**에서 **CreateInstance** 다음에 호출합니다.



콜백에 대한 세부 설명을 위해 구조체를 채워야 합니다.



**messageSeverity** 필드는 당신의 콜백에서 호출될 모든 타입의 위협도를 지정합니다. VK\_DEBUG\_UTILS\_MESSAGE\_SEVERITY\_INFO\_BIT\_EXT를 제외한 사용할 모든 타입을 지정했으며, 일반적인 디버그 정보들을 빼고 가능한 문제점들만 받기 위해서 제외했습니다.

**messageType** 필드는 콜백에서 어느 타입의 메시지를 받을 것인지 필터링하는 것에 대해서 비슷합니다. 여기는 간단히 모두 지정했으며, 당신이 필요없다고 생각하는 것은 제외하면 됩니다.

마지막으로 **pfnUserCallback** 필드는 콜백 함수 포인터를 지정합니다. 선택적으로 **pUserData** 매개 변수에 콜백 함수로 전달될 매개 변수를 전달합니다. 예로, **HelloTriangleApplication** 포인터를 전달할 수도 있습니다.

유효성 레이어와 디버그 콜백을 설정하는데에는 많은 방법이 있지만, 이 방법이 튜토리얼을 시작하기에 좋은 방법입니다. 다른 가능성에 대해서는 사양서를 참고하세요.

이 구조체는 **VkDebugUtilsMessengerEXT** 객체를 생성하기 위해 **vkCreateDebugUtilsMessengerEXT** 함수로 전달됩니다. 불행히도, 이 함수는 확장 함수이기 때문에 자동으로 로드되지 않습니다. 우리는 **vkGetInstanceProcAddr**을 이용해 직접 주소를 찾아야합니다. 우리는 백그라운드에서 처리될 우리만의 프록시(대체자, 대신 수행하는) 함수를 만들겠습니다. **HelloTriangleApplication** 정의 위에 추가합니다.



vkGetInstanceProcAddr 함수는 함수가 로드되지 않았다면 nullptr을 반환합니다. 이제 이 함수를 호출해 확장 객체를 생성할 수 있습니다.



뒤에서 두 번째 매개 변수는 nullptr로 설정할 선택적 할당자이고, 나머지 하나는 callback 함수입니다. 디버그 콜백은 Vulkan 인스턴스와 그 레이어에 지정되기 때문에, 첫 번째 매개변수로 인스턴스를 명확하게 지정해야 합니다. 또한 나중에 자식 객체에 대한 패턴을 보실 수 있을 겁니다. 잘 작동하는지 봅시다… 프로그램을 실행하고, 빈 창이 떴을 때 닫아줍니다. 명령 프롬프트에 다음과 같은 메시지가 출력된 걸 볼 수 있습니다

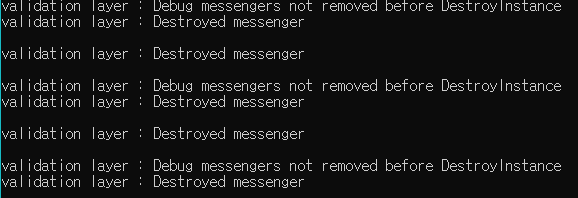


그림 6.1

이런! 벌써 우리 프로그램이 버그로 뒤덥혔습니다! **VkDebugUtilsMessengerEXT** 객체는 **vkDestroyUtilsMessengerEXT** 호출로 정리되어야 합니다. **vkCreateDebugUtilsMessengerEXT** 함수와 비슷하게 직접 로드해야 합니다. 일반 메시지가 여러 번 출력되는 걸 보세요 이러한 상황이 발생되는 이유는 여러 유효성 레이어가 디버그 메신저 삭제를 확인하기 때문입니다.

**CreateDebugUtilsMessengerEXT** 다음에 또 다른 프록시 함수를 생성합니다.



이 함수를 스태틱 클래스 함수나 클래스 밖의 함수처럼 작성하시면 됩니다. 이제 **CleanUp** 함수에서 호출합니다.



다시 프로그램을 실행하면 에러가 나타나지 않습니다. 어떤 호출이 메시지를 보냈는지 알고싶다면, 콜백 함수에 중단점을 찍어, 스택 트레이스를 하면 됩니다.

구성

유효성 레이어의 행동을 위한 정말 많은 설정과 그리고 **VkDebugUtilsMessengerCreateInfoEXT** 구조체에 플래그를 지정했습니다. Vulkan SDK의 Config 폴더를 찾아보세요. **vk\_layer\_settings.txt** 파일에 어떻게 레이어를 구성하는지 설명 되어있습니다.

당신만의 프로그램을 위한 레이어 설정을 구성하려면, 파일을 복사해서 당신의 프로젝트의 **Debug**와 **Release** 폴더에 복사합니다. 그리고 직접 원하는 행동의 명렁을 내리면 됩니다. 하지만 앞으로의 남은 튜토리얼에서는 기본 설정을 사용합니다.

튜토리얼에서 전반적으로 일부로 여러 실수들을 만들어서 유효성 레이어가 검출하게 함으로써, 얼마나 유용한지 그리고 Vulkan으로 무엇을 하고 있는지 가르칠 것입니다. 이제 시스템에서 Vulkan 장치들을 살펴볼 차례입니다.

[C++ 코드](https://vulkan-tutorial.com/code/02_validation_layers.cpp)

Chapter 7 물리 장치 및 큐 패밀리

물리 장치 선택

VkInstance로 Vulkan 라이브러리를 초기화하고 난 후, 우리가 필요한 기능들을 지원하는 그래픽 카드들을 살펴보고 선택해야 합니다. 사실은 여러 그래픽카드를 선택해 동시에 사용할 수도 있지만, 이번 튜토리얼에선 우리가 필요로 하는 첫 번째 그래픽 카드를 선택할 것입니다.

**PickPhysicalDevice** 함수를 추가하고, **InitVulkan** 함수에서 호출합니다.



그래픽 카드는 선택되고 나면 새로운 클래스 멤버로 추가할 VkPhysicalDevice 핸들에 저장됩니다. 이 객체는 VkInstance가 파괴될 때 명시적으로 같이 파괴됩니다. 그러므로 **CleanUp** 함수에 아무것도 추가하지 않아도 됩니다.



그래픽 카드의 목록을 불러오는 것은 확장 목록 불러오기와 매우 비슷합니다. 개수를 질의 하는 것으로 시작합니다.



Vulkan을 지원하는 장치가 0개인지 확인합니다.



그렇지 않다면 VkPhysicalDevice 핸들들을 저장할 배열을 할당할 수 있습니다.



모든 그래픽 카드가 똑같이 만들어지지 않기 때문에, 각 그래픽 카드들을 평가하고 우리가 원하는 행동의 연산들이 적합한지 확인해야 합니다. 다음 새로운 함수를 작성합니다.



그리고 새로운 함수를 통해 모든 물리 장치가 요구사항을 충족하는지 확인합니다.



다음 섹션에서는 IsDeviceSuitable 함수에서 확인할 첫 번째 요구 사항을 소개합니다. 이후 챕터에서 더 많은 Vulkan 기능을 사용하기 시작하므로, 더 많은 확인을 포함할 수 있도록 확장할 것입니다.

기본(Base) 장치 적합성 확인

장치의 적합성을 평가하기 위해서 일부 세부 사항을 질의하는 것으로 시작합니다. 이름, 타입, 지원하는 Vulkan 버전과 같은 기본(Basic) 장치의 속성은 vkGetPhysicalDeviceProperties로 질의할 수 있습니다.



텍스처 압축, 64 비트 부동 소수점, 다중 뷰포트 렌더링 (VR에 유용한) 과 같은 선택적 기능들의 지원은 vkGetPhysicalDeviceFeatures로 통해 확인할 수 있습니다.



더 많은 세부 정보들을 장치에 질의할 수 있지만, 장치 메모리와 큐 패밀리에 관하여는 다음에 논의해보겠습니다. (다음 섹션을 보세요)

예를 하나들자면, 우리의 프로그램이 지오메트리 셰이더를 지원하는 전용 그래픽 카드에서 동작한다고 합시다. 그러면 **IsDeviceSuitable** 함수는 다음과 같을 것입니다.



장치가 적합한 지에 대해서 확인하는 대신, 각 장치에 점수를 부여해서 가장 높은 것을 선택할 수도 있습니다. 이 방법은 높은 점수를 받은 전용 그래픽 카드를 사용할 수 있지만, 하나만 존재하는 경우 통합 GPU로 다시 돌아갈 수도 있습니다. 당신은 다음과 같이 구현할 수도 있을 겁니다.



이 튜토리얼에서는 직접 구현할 필요가 없으나, 이는 장치 선택 프로세스를 디자인하는데 아이디어를 제공할 수 있겠죠. 물론 선택 항목들을 화면에 표시하고 상용자가 선택할 수 있도록 할 수도 있습니다.

우리는 막 시작했기 때문에, Vulkan의 지원만으로도 우리가 필요한 것을 충족하기 때문에 지금은 모든 GPU에 만족할 것입니다.



다음 섹션에서는 실제로 필요한 첫 번째 기능을 확인해보겠습니다.

큐 패밀리

Vulkan의 그리기 작업부터 텍스처 업로딩에 이르기까지 대부분의 연산이 큐에 명령을 제출해야 합니다. 다른 큐 패밀리로부터 만들어지는 다른 타입의 큐들이 있으며, 각각의 큐 패밀리는 명령의 일부분만(subset) 허용됩니다. 예를 들어, 컴퓨트 명령만 수행하는 큐 패밀리가 있을 수 있고, 메모리 전송 관련 명령만 수행하는 큐 패밀리가 있을 수 있습니다.

우리는 어떤 큐 패밀리가 장치에 의해 지원되고, 어느 것이 우리가 사용할 명령들이 지원하는지 확인해야 합니다. 이러한 목적을 위해 **FindQueueFamilies** 함수를 만들어 우리가 필요한 큐 패밀리를 찾을 것입니다. 당장은 그래픽스 명령을 지원하는 큐를 찾아볼 것이지만, 나중에는 이 함수가 더 확장될 수도 있습니다.

이 함수는 원하는 속성들을 충족하는 큐 패밀리의 인덱스(Indices)를 반환합니다. 가장 좋은 방법은 구조체를 쓰는 것입니다. 인덱스가 -1이라면 “찾지 못함”을 표시합니다.



이제 FindQueueFamilies 함수를 구현할 수 있습니다.



큐 패밀리를 가져오는 것은 예상 하셨다 시피 **vkGetPhysicalDeviceQueueFamilyProperties**를 사용합니다.



VkQueueFamilyProperties 구조체는 지원하는 연산과 해당 패밀리를 기반으로 생성될 수 있는 큐의 수를 포함한 큐 패밀리에 대한 몇 가지 세부 정보가 들어 있습니다. 우리는 VK\_QUEUE\_GRAPHICS\_BIT를 지원하는 큐 패밀리를 적어도 하나를 찾아야 합니다.



이제 멋진 큐 패밀리 찾기 함수가 생겼으니, 장치에서 명령을 수행할 수 있도록 **IsDeviceSuitable** 함수에서 사용하여 확인합니다.



좋습니다, 이제 우리는 적절한 물리 장치를 찾았습니다. 다음 단계는 상호 작용할 논리 장치를 생성하는 것입니다.

[C++ 코드](https://vulkan-tutorial.com/code/03_physical_device_selection.cpp)