VULKAN

Kunštek, Toni

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

VULKAN

Završni rad

Toni Kunštek

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 01.09.2021.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime studenta:	Toni Kunštek
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4229, 25.07.2018.
OIB studenta:	60222666607
Mentor:	Izv. prof. dr. sc. Alfonzo Baumgartner
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Vulkan
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	01.09.2021.
Datum potvrde ocjene Odbora:	08.09.2021.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 15.09.2021.

Ime i prezime studenta:	Toni Kunštek
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4229, 25.07.2018.
Turnitin podudaranje [%]:	5

Ovom izjavom izjavljujem da je rad pod nazivom**: Vulkan**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Alfonzo Baumgartner

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA

o odobrenju za pohranu i objavu ocjenskog rada

kojom ja __Toni Kunštek_____, OIB: __60222666607_____, student/ica Fakulteta elektrotehnike, računarstva i informacijskih tehnologija Osijek na studiju _Preddiplomski sveučilišni studij Računarstvo_, kao autor/ica ocjenskog rada pod naslovom: _Vulkan**,**

dajem odobrenje da se, bez naknade, trajno pohrani moj ocjenski rad u javno dostupnom digitalnom repozitoriju ustanove Fakulteta elektrotehnike, računarstva i informacijskih tehnologija Osijek i Sveučilišta te u javnoj internetskoj bazi radova Nacionalne i sveučilišne knjižnice u Zagrebu, sukladno obvezi iz odredbe članka 83. stavka 11. *Zakona o znanstvenoj djelatnosti i visokom obrazovanju* (NN 123/03, 198/03, 105/04, 174/04, 02/07, 46/07, 45/09, 63/11, 94/13, 139/13, 101/14, 60/15).

Potvrđujem da je za pohranu dostavljena završna verzija obranjenog i dovršenog ocjenskog rada. Ovom izjavom, kao autor/ica ocjenskog rada dajem odobrenje i da se moj ocjenski rad, bez naknade, trajno javno objavi i besplatno učini dostupnim:

a) široj javnosti

b) studentima/icama i djelatnicima/ama ustanove

c) široj javnosti, ali nakon proteka 6 / 12 / 24 mjeseci (zaokružite odgovarajući broj mjeseci).

**U slučaju potrebe dodatnog ograničavanja pristupa Vašem ocjenskom radu, podnosi se obrazloženi zahtjev nadležnom tijelu Ustanove.*

__Osijek, 15.09.2021._____ _____

(mjesto i datum) (vlastoručni potpis studenta/ice)

SADRŽAJ

1. UVOD

Tema ovoga završnog rada je „Vulkan”. Vulkan je grafičko aplikacijsko programsko sučelje (API) kojemu je cilj ostvarivanje bolje standardizirane komunikacije između koda programa i grafičkih uređaja. Autori ovoga grafičkog sučelja su Khronos grupa, ista grupa koja od 2006. godine kontrolira široko korišteno grafičko sučelje *OpenGL*. Glavna prednost Vulkana nad *OpenGL* sučeljem je mogućnost davanja zadataka grafičkom uređaju s više procesorskih jezgri, čime se maksimalno iskorištavaju resursi. Ovime se uklanja poznato „usko grlo“, jednojezgrenost procesora kod iscrtavanja *OpenGL* sučelja. Vulkan može biti korišten za razne druge potrebe, kao što su obavljanje matematičkih operacija, poboljšavanje računanja u fizici te procesiranje videa.

U poglavlju „2. Postavljanje okruženja“ objašnjeno je postavljanje i povezivanje potrebnih biblioteka s korištenim integriranim razvojnim okruženjem *Visual Studio* za rad unutar Vulkan okruženja. Poglavlje „3. Ključni čimbenici“ bavi se teorijskim dijelom unutar kojega su, uz popratne isječke koda objašnjeni ključni čimbenici Vulkana koji čine aplikaciju sposobnom za iscrtavanje slika. Poglavlje „4. Rezultati“ prikazuje rezultate dobivene pokretanjem programa s danim ulaznim podacima.

1.1. Zadatak završnog rada

U teorijskom dijelu rada potrebno je opisati Vulkan 3D biblioteku za razvoj 3D računalnih programa. Na primjeru pokazati kao se radi iscrtavanje jednostavnih oblika (kocka, kugla i slično). Opisati i napraviti primjer ugradnje gotovog 3D modela u formatu .obj u 3D scenu. Napraviti osnovno sjenčenje i po mogućnosti definirati neke jednostavne teksture ili shader-e na učitanom objektu.

2. POSTAVLJANJE OKRUŽENJA

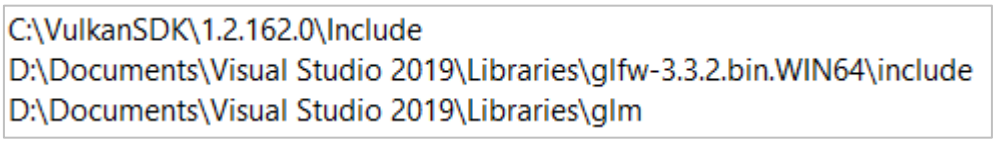
U svrhu programiranja unutar Vulkan okruženja potrebno je preuzeti VulkanSDK biblioteku. VulkanSDK biblioteka sadrži sve potrebne header datoteke, standardne validacijske slojeve, alate za pronalazak grešaka i učitavač Vulkan funkcija. Najnoviju stabilnu verziju moguće je preuzeti na stranici LunarG [1]. Vulkan biblioteka ne sadrži alate za kreiranje prozora za prikaz generiranih slika. Iz ovoga razloga je potrebno uključiti biblioteku koja se može pobrinuti za taj dio. Za kreiranje prozora za ovaj rad je korištena biblioteka GLFW. Također je korištena i GLM biblioteka koja sadrži operacije u vezi linearne algebre. U svrhu samog programiranja C++ jezikom korišteno je integrirano sučelje *Visual Studio 2019*. Potrebno je u projekt uključiti prethodno navedene biblioteke, prema koracima u nastavku:

I. Project > [Project] properties

U navedenom prozoru „[project] properties“, gdje je umjesto „[project]“ napisan naziv projekta, potrebno je odraditi ostale korake.

II. C/C++ > general > Additional include directories > edit

Potrebno je unijeti putanje do preuzetih biblioteka, do „include“ mape. GLM sadrži samo header datoteke pa je dovoljno unijeti putanju samo do prve mape. (slika 2.1.)



Slika 2.1. Putanje do preuzetih biblioteka (za C/C++)

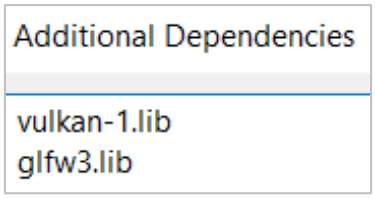
III. Linker > general > Additional library dependencies > edit

Također, kao u prethodnom koraku, potrebno je unijeti putanje do preuzetih biblioteka, ali u ovome slučaju do mape *Lib* za VulkanSDK biblioteku te *lib-vc####*, gdje *####* predstavlja verziju korištenog Visual Studio programa. (Slika 2.2.)

Slika 2.2. Putanje do preuzetih biblioteka (za linker)

IV. Linker > input > Additional dependencies > edit

Potrebno je unijeti tekst kao što je prikazano na slici 2.3.



Slika 2.3. Izlistavanje naziva dodanih zavisnosti

V. C/C++ > Language > C++ Language standard > „ISO C++17 Standard (/std:c++17)“

Potrebno je promijeniti standard jezika C++ na C++17 radi izbjegavanja određenih grešaka.

(Napomena: postavljanje okruženja po prethodno navedenim koracima vrijedi samo za rad na Windows uređaju. Za rad na drugim operacijskim sustavima postavljanje okruženja se razlikuje.)

3. KLJUČNI ČIMBENICI

U ovome poglavlju objašnjeni su ključni čimbenici, objekti i strukture koje je potrebno inicijalizirati za iscrtavanje slika unutar prozora. Više detalja o njima moguće je pronaći na službenoj stranici khronos grupe [2]. Za pomoć pri strukturiranju koda korištena je stranica *Vulkan tutorial* [3].

Na slici 3.1. nalazi se funkcija koja je pozvana prije inicijaliziranja ičega što je vezano za Vulkan. Funkcija kreira prozor koji će se koristiti za iscrtavanje funkcijama iz Vulkan biblioteke. Funkcije pozvane unutar *initWindow* su funkcije iz GLFW biblioteke. Funkcija *glfwWindowHint* naređuje programu da se ne kreira *OpenGL* kontekst jer će se umjesto njega koristiti Vulkan. Varijable *WIDTH* i *HEIGHT* predstavljaju varijable koje definiraju veličinu prozora.

```
void VulkanApplication::initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    glfwSetWindowUserPointer(window, this);
    glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
}
```

Slika 3.1. Inicijalizacija GLFW prozora

Vulkan ne može direktno manipulirati ovim prozorom, već je potrebno koristiti *Windows system integration* (*WSI*) ekstenziju. Objekt klase *VkSurfaceKHR* predstavlja abstraktni tip površine na kojoj se mogu prikazati slike. Ova površina će se pridodati već izrađenom GLFW prozoru, što se s lakoćom postiže pozivom funkcije *glfwCreateWindowSurface*, kao na slici 3.2.

```
void VulkanApplication::createSurface() {
    if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
        throw std::runtime_error("failed to create window surface!");
    }
}
```

Slika 3.2. Kreiranje površine prozora

3.1. Instanca

Instanca je veza između aplikacije i Vulkan biblioteke. Kreiranje objekta instance zahtjeva specificiranje detalja o aplikaciji potrebne programskoj podršci grafičkog uređaja (engl. *driver*). Kreiranje objekta instance prikazano je na slici 3.1.1.

```
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
}
```

Slika 3.1.1. Kreiranje objekta instance

Funkcije za kreiranje Vulkan objekata uglavnom prate isti obrazac. Funkcija *vkCreateInstance* prihvaća pokazivač na prijašnje kreiranu i ispunjenu strukturu kao prvi argument. Ova struktura je zadužena za sve opcije objekta koji se kreira. Zatim, kao drugi argument prima pokazivač na posebno specificirane algoritme za alociranje memorije. Uvijek se stavlja *nullptr* na ovo mjesto kada se pretpostavlja da uređaj ima dovoljno slobodne memorije. Kao treći argument funkcija prima pokazivač na varijablu koja sprema objekt. Gotovo sve Vulkan funkcije vraćaju vrijednost tipa *VkResult* koja vraća ili *VK_SUCCESS* ili kod greške, praćen izbacivanjem programa. Na slici 3.1.2 prikazano je kreiranje i ispunjavanje strukture koja je predana kao prvi argument funkcije *vkCreateInstance*.

```
VkInstanceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
auto extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
createInfo.ppEnabledExtensionNames = extensions.data();
```

Slika 3.1.2. Primjer kreiranja i ispunjavanja strukture *VkInstanceCreateInfo*

Svaka struktura unutar Vulkan okruženja sadrži *sType* kao prvi atribut kojime je potrebno specificirati koja verzija strukture se koristi. Ovaj atribut uglavnom služi tome da buduće verzije *VulkanSDK* biblioteke budu kompatibilne sa starijim verzijama. U tu istu svrhu postoji i atribut *pNext* koji predstavlja pokazivač na proširenje strukture, za buduće verzije. Ostali parametri prikazani na

slici 3.1.2. specifični su za kreiranje objekta instance. Unutar strukture *appInfo* definiraju se atributi kao što su naziv i verzija aplikacije.

Lista imena potrebnih proširenja za instancu *extensions* dobavlja se pomoću funkcije *getRequiredExtensions*. (Slika 3.1.3.)

```
std::vector<const char*> VulkanApplication::getRequiredExtensions() {
    uint32_t glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    std::vector<const char*> extensions(glfwExtensions, glfwExtensions + glfwExtensionCount);

    return extensions;
}
```

Slika 3.1.3. Provjera podržanosti ekstenzija

Funkcija *glfwGetRequiredInstanceExtensions* vraća polje imena Vulkan proširenja instance potrebnih za kreiranje Vulkan instance za GLFW prozor [4].

3.2. Validacijski slojevi

Validacijski slojevi predstavljaju slojeve unutar programa koji pronalaze greške aplikacije. Uobičajene operacije kojima se bavi validacijski sloj:

I. Provjerava vrijednosti parametara prema specifikaciji radi otkrivanja zlouporabe
II. Prati kreiranje i destrukciju objekata kako bi se prepoznalo curenje memorije
III. Provjerava sigurnost niti pratnjom poziva njihovih pozivača
IV. Ispisuje svaki poziv i dane parametre na standardni izlaz
V. Pronalazi Vulkan pozive za profiliranje i reprodukciju

VK_LAYER_KHRONOS_validation jedan je od slojeva koji se nalazi unutar *VulkanSDK* biblioteka. Ovaj sloj sadrži svu korisnu standardnu validaciju grupiranu unutar jednoga sloja. Svrha poruka validacijskog sloja mogu biti obavijest, upozorenje i greška. S lakoćom se može upravljati prikazanošću poruka tijekom popunjavanja strukture, prikazane na slici 3.2.1.

```
createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
createInfo.messageSeverity =
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;

createInfo.messageType =
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;

createInfo.pfnUserCallback = debugCallback;
```

Slika 3.2.1. Upravljanje prikazanošću poruka validacijskog sloja

Uklanjanjem enumeracijskih vrijednosti s određenim nastavcima filtriraju se određene vrste *debug* poruka. Također se uklanjanjem iz *messageType* filtrira tip poruka koje će u slučaju grešaka javiti, kao što su generalne poruke, poruke o validaciji te poruke o performansama.

Validacijske slojeve je moguće potpuno onemogućiti kada više nisu potrebni, kada se dokaže da aplikacija pravilno radi i da je spremna za objavu. Potrebno je samo otkomentirati prvu liniju koda prikazanu na slici 3.2.2. Onemogućavanjem validacijskih slojeva se dobiva na brzini izvođenja programa.

```
//#define NDEBUG = 1

#ifdef NDEBUG
const bool enableValidationLayers = false;
#else
const bool enableValidationLayers = true;
#endif
```

Slika 3.2.2. Onemogućavanje validacijskih slojeva

3.3. Odabir grafičkog uređaja

Neka računala sadrže više od jednog uređaja sposobnog za efektivno obavljanje grafičkih operacija. Često su to uređaji koji osim grafičke kartice sadrže i integriranu grafiku na procesoru. Kako bi bio odabran najbolji mogući ili najpoželjniji takav uređaj za korištenje, prvo je potrebno provjeriti podržava li pojedini uređaj sve potrebne značajke koje aplikacija zahtjeva.

Na slici 3.3.1. prikazana je provjera podržanosti uređaja. Popis svih dostupnih uređaja dobavljen je funkcijom *vkEnumeratePhysicalDevices* te se sprema u varijablu *devices*. Svaki uređaj se zatim predaje funkciji *isDeviceSuitable* koja provjerava podržava li sva proširenja i značajke potrebne za aplikaciju.

```
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());

for (const auto& device : devices) {
    if (isDeviceSuitable(device)) {
        physicalDevice = device;
        break;
    }
}
```

Slika 3.3.1. Odabir grafičkog uređaja

Osim same provjere podržava li grafički uređaj sučelje Vulkan i sve potrebne značajke, moguće je svakom uređaju dati ocjenu pa odabrati onaj kojemu je ocjena najviša. Na ovaj način je moguće automatizirati odabir „jačih“ grafičkih uređaja. Naravno, moguće je samo izlistati imena uređaja za odabir te uzeti najpoželjniji.

3.4. Logički uređaj

(engl. *Logical device*)

Nakon kreiranja fizičkog uređaja potrebno je napraviti i logički uređaj koji će predstavljati sučelje tog uređaja. Moguće je kreirati više od jednog logičkog uređaja ako postoje varijacije u zahtjevima. Kreiranje logičkog uređaja se svodi na specificiranje značajki uređaja

VkPhysicalDeviceFeatures koje se žele koristiti. Osim toga potrebno je priložiti reference na željene familije redova i proširenja. (Slika 3.4.1.)

```
VkPhysicalDeviceFeatures deviceFeatures{};
deviceFeatures.samplerAnisotropy = VK_TRUE;

VkDeviceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
createInfo.pQueueCreateInfos = queueCreateInfos.data();
createInfo.pEnabledFeatures = &deviceFeatures;
createInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
createInfo.ppEnabledExtensionNames = deviceExtensions.data();

if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
    throw std::runtime_error("failed to create logical device!");
}
```

Slika 3.4.1. Kreiranje logičkog uređaja

Gotovo svaka operacija unutar Vulkan okruženja, kao na primjer iscrtavanje i postavljanje tekstura, zahtjeva da se naredbe stave u red. Svaka familija redova dopušta spremanje samo određenih naredbi. Na slici 3.4.2. nalazi se struktura koja sadrži indekse dvaju vrsta familija redova.

```
struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
    std::optional<uint32_t> presentFamily;

    bool isComplete() {
        return graphicsFamily.has_value() && presentFamily.has_value();
    }
};
```

Slika 3.4.2. Struktura familija redova

Za spremanje indeksa familija redova korišten je klasni predložak *std::optional* [5] radi učinkovite provjere postoji li neka vrijednost u danom trenutku, uključujući i broj nula. Pomoću ovih indeksa se pristupa redovima. Redovi su automatski kreirani prilikom kreiranja logičkog uređaja. Za dohvaćanje redova, potrebno je pozvati *vkGetDeviceQueue* s odgovarajućim argumentima,

referencama na klasne *VkQueue* objekte nazvane *graphicsQueue* i *presentQueue*, kao što je prikazano na slici 3.4.3.

```
vkGetDeviceQueue(device, indices.graphicsFamily.value(), 0, &graphicsQueue);
vkGetDeviceQueue(device, indices.presentFamily.value(), 0, &presentQueue);
```

Slika 3.4.3. Dohvaćanje redova

Na slici 3.4.4. prikazan je vektor [6] unutar kojega je postavljena jedna ekstenzija uređaja korištena u aplikaciji za ovaj rad, a to je lanac zamjenjivanja. Ukoliko je potrebno dodati još neko proširenje uređaja, lako ga je moguće dodati u ovaj vektor.

```
const std::vector<const char*> deviceExtensions = {
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
};
```

Slika 3.4.4. Vektor omogućenih proširenja

3.5. Lanac zamjenjivanja

(engl. *Swap chain*)

Lanac zamjenjivanja sprema red slika koje čekaju zamjenu s onom koja je prikazana na zaslonu. Predstavlja infrastrukturu u kojoj se svaka slika u lancu pripremi prije nego se prikaže na ekranu. Glavna funkcija lanca je sinkronizacija prezentiranja slika s brzinom osvježavanja slika na zaslonu.

3.5.1. Detalji lanca

```
struct SwapChainSupportDetails {
    VkSurfaceCapabilitiesKHR capabilities;
    std::vector<VkSurfaceFormatKHR> formats;
    std::vector<VkPresentModeKHR> presentModes;
};
```

Slika 3.5.1. Struktura detalja o lancu zamjenjivanja

Na slici 3.5.1. prikazana je struktura unutar koje se nalaze detalji o mogućnostima lanca zamjenjivanja danog grafičkog uređaja. Lanac zamjenjivanja je podržan ako postoji barem jedan podržani format slika i prezentacijski način rada. Za kreiranje optimalnog lanca potrebno je odabrati optimalne postavke formata površine, prezentacijskog načina rada i rezoluciju slika unutar lanca.

Prvi član ove strukture je struktura *VkSurfaceCapabilitiesKHR* koja sadrži podatke o osnovnim mogućnostima prezentacijske površine kao što su minimalni i maksimalni broj slika unutar lanca te minimalna i maksimalna rezolucija slike. Rezolucija slika unutar lanca zamjenjivanja je gotovo uvijek ista rezoluciji prozora unutar kojega se iscrtavaju slike. Izražena je u pikselima. Zbog mogućnosti da je došlo do promjene rezolucije, potrebno je odraditi provjeru prikazanu na slici 3.5.2. Ukoliko je trenutna širina prozora različita od *UINT32_MAX*, tada se vraća trenutni opseg slike. Trenutna širina prozora biti će jednaka tome broju kada se rezolucije ne podudaraju. U tome slučaju je potrebno kreirati novi opseg slike *VkExtent2D* čije se dimenzije, širina i visina izjednačava s onom visinom i širinom koju dani argument *capabilities* određuje, to jest prvi član strukture *SwapChainSupportDetails*.

```
VkExtent2D VulkanApplication::chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {
    if (capabilities.currentExtent.width != UINT32_MAX) {
        return capabilities.currentExtent;
    }
    else {
        int width, height;
        glfwGetFramebufferSize(window, &width, &height);
        VkExtent2D actualExtent = {
            static_cast<uint32_t>(width),
            static_cast<uint32_t>(height)
        };
        actualExtent.width = std::max(capabilities.minImageExtent.width,
            std::min(capabilities.maxImageExtent.width, actualExtent.width));
        actualExtent.height = std::max(capabilities.minImageExtent.height,
            std::min(capabilities.maxImageExtent.height, actualExtent.height));
        return actualExtent;
    }
}
```

Slika 3.5.2. Provjera jednakosti rezolucije trenutnog prozora

Drugi član strukture je vektor koji sprema listu enumeracijskih vrijednosti kojima se definiraju formati boja piksela i prostor boja koje grafički uređaj podržava. Svaki format površine (engl. *surface*

format) *VkSurfaceFormatKHR* sastoji se od dijela koji specificira format i dijela koji određuje prostor boja (engl. *color space*), na primjer: *VK_FORMAT_B8G8R8A8_SRGB*. Prvi dio (*B8G8R8A8*) predstavlja format boja. U ovome slučaju predstavlja 8 bita po boji, uključujući i prozirnost (*alpha*), na način da se sprema slijedno plava, zelena, crvena te na poslijetku prozirnost. Drugi dio („SRGB“) predstavlja standardni prostor boja. Ove podatke moguće je dobiti pozivom jedne funkcije *vkGetPhysicalDeviceSurfaceCapabilitiesKHR*.

Zadnji član sadrži vektor mogućih prezentacijskih načina rada (engl. *presentation mode*). Odabir prezentacijskog načina rada je jedno od najbitnijih postavki za lanac zamjenjivanja jer reprezentira same uvjete za prikaz slika na ekranu. U Vulkanu postoji 4 prezentacijska načina rada:

I. *VK_PRESENT_MODE_IMMEDIATE_KHR*
II. *VK_PRESENT_MODE_FIFO_KHR*
III. *VK_PRESENT_MODE_FIFO_RELAXED_KHR*
IV. *VK_PRESENT_MODE_MAILBOX_KHR*

Ako se koristi *IMMEDIATE*, slike su odmah prikazane na ekranu. U ovome slučaju može doći do tzv. efekta kidanja zaslona, greške u prikazu gdje je dio slike horizontalno pomaknut. Garantirano je da je *FIFO* način podržan. U tom slučaju, kada dođe novo vrijeme osvježenja slike, lanac zamjenjivanja prikazuje sliku koja je prva ušla u red. U isto vrijeme se dodaje nova slika na kraj reda. Ako nema mjesta u redu, tada program čeka. Način *FIFO_RELAXED* je sličan prethodnom. Jedina razlika je to što u trenutku osvježenja slike, ako nema slika u redu, kada napokon dođe nova slika u red, doći će do osvježenja čak iako nije vrijeme za osvježavanje. *MAILBOX* način je također sličan drugome. Umjesto da aplikacija blokira u trenutku kada je red pun, slike koje su već u redu će biti zamijenjene novima.

Na slici 3.5.3. prikazano je popunjavanje strukture *SwapChainSupportDetails*. Sve funkcije za dobavljanje detalja o lancu započinju s *vkGetPhysicalDeviceSurface* te nastavljaju ovisno o kojoj vrsti detalja se radi.

```
SwapChainSupportDetails details;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface, &details.capabilities);
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, nullptr);
if (formatCount != 0) {
    details.formats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, details.formats.data());
}
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, nullptr);
if (presentModeCount != 0) {
    details.presentModes.resize(presentModeCount);
    vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, details.presentModes.data());
}
return details;
```

Slika 3.5.3. Popunjavanje strukture *SwapChainSupportDetails*

3.5.2. Slikovni pogledi

Slikovni pogled je objekt unutar kojega se sprema slika *VkImage*. Na slici 3.5.4. prikazano je kreiranje slikovnih pogleda za lanac zamjenjivanja. Broj napravljenih slikovnih pogleda je određen mogućnostima korištenog grafičkog uređaja.

```
void VulkanApplication::createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());

    for (uint32_t i = 0; i < swapChainImages.size(); i++) {
        swapChainImageViews[i] = createImageView(swapChainImages[i],
            swapChainImageFormat, VK_IMAGE_ASPECT_COLOR_BIT);
    }
}
```

Slika 3.5.4. Kreiranje slikovnih pogleda

Kao argumente funkcije *createImageView* pridodaju se slika istoga indeksa, odabrani format površine te *VK_IMAGE_ASPECT_COLOR_BIT* čime se specificira da se radi o slici koja sadrži podatke o bojama. Ovi su argumenti potrebni za popunjavanje strukture kojom se kreira pojedini slikovni pogled, prikazana na slici 3.5.5. Slikovnom pogledu se prilikom kreacije pridodaje slika *VkImage*, tip i format pogleda te *aspectMask*, koji predstavlja vrstu podataka koji se sprema, na primjer podaci o boji ili dubinski podaci. Osim ovih podataka specificiraju se i podaci o razinama mipmapa te slojevima slike. Svaka iduća *mipmap* razina predstavlja umanjenu sliku, dimenzija jednog kvadranta prethodne *mipmap* razine.

```
VkImageViewCreateInfo viewInfo{};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
viewInfo.image = image;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = format;
viewInfo.subresourceRange.aspectMask = aspectFlags;
viewInfo.subresourceRange.baseMipLevel = 0;
viewInfo.subresourceRange.levelCount = 1;
viewInfo.subresourceRange.baseArrayLayer = 0;
viewInfo.subresourceRange.layerCount = 1;
```

Slika 3.5.5. Struktura za kreiranje slikovnog pogleda

3.5.3. Rekreacija lanca zamjenjivanja

U bilo kojem trenutku, tijekom izvođenja programa može doći do promjene dimenzija prozora na kojem se iscrtavaju slike. Ako dođe do ovoga, potrebno je rekreirati lanac zamjenjivanja tako da dimenzije lanca ponovo odgovaraju dimenzijama prozora. Na slici 3.5.6 prikazan je prvi dio funkcije zadužene za rekreiranje lanca.

```
void VulkanApplication::recreateSwapChain() {
    int width = 0, height = 0;
    glfwGetFramebufferSize(window, &width, &height);
    while (width == 0 || height == 0) {
        glfwGetFramebufferSize(window, &width, &height);
        glfwWaitEvents();
    }
    vkDeviceWaitIdle(device);
```

Slika 3.5.6. Prvi dio funkcije *recreateSwapChain*

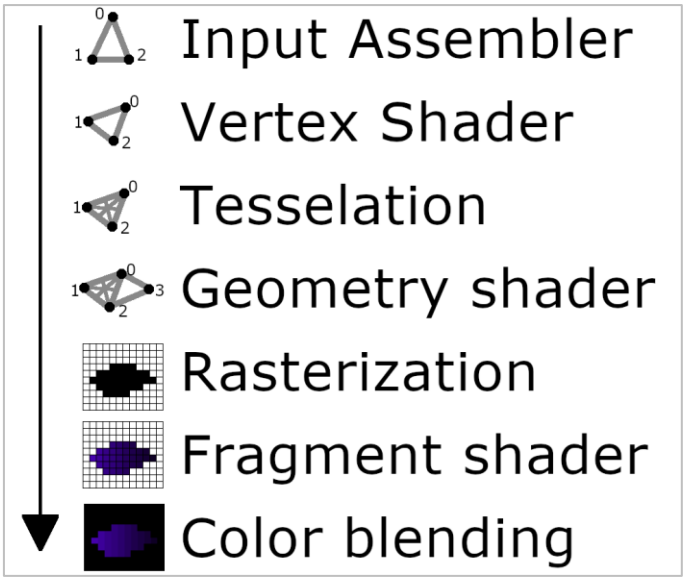
Nova visina i širina prozora dobavlja se funkcijom *glfwGetFramebufferSize*. Ako je bilo koja dimenzija jednaka nuli, program zapinje u petlji dok ne dobije dimenzije veće od nule. Kada dimenzije budu veće od nule, program nastavlja s drugim dijelom funkcije prikazan na slici 3.5.7. Funkcija *cleanupSwapChain* briše sve objekte koji se ponovo kreiraju i postavljaju funkcijama nakon nje.

```
    cleanupSwapChain();

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createDepthResources();
    createFramebuffers();
    createUniformBuffers();
    createDescriptorPool();
    createDescriptorSets();
    createCommandBuffers();

    imagesInFlight.resize(swapChainImages.size(), VK_NULL_HANDLE);
}
```

Slika 3.5.7. Drugi dio funkcije *recreateSwapChain*

3.6. Grafički cjevovod

(engl. *Graphics pipeline*)



Slika 3.6.1. Pojednostavljeni prikaz koraka grafičkog cjevovoda

Grafički cjevovod predstavlja sekvencu operacija koja pretvara podatke koji opisuju 3D modele u piksele prikazane na ekranu. Postupak započinje dohvaćanjem podataka o modelu kao što

su pozicije točaka, smjer vektora normala, koordinate tekstura i poligona. Zatim se pozicije svih točaka modela pretvaraju u točke prostora koje će biti prikazane na zaslonu. Za ovaj korak je zadužen *vertex shader*. Nakon toga slijedi rasterizacija. Rasterizacija je postupak unutar kojega se provjerava vidljivost svake točke na zaslonu generirane u prošlome koraku te se svaki vidljivi poligon pretvara u fragmente, tj. piksel elemente. *Fragment shader* je slijedno zadužen za pretvaranje tih fragmenata u prave, obojane piksele prikazane na zaslonu. Postoji još nekoliko koraka čije bi dodavanje poboljšalo krajnji rezultat. Između *vertex shadera* i rasterizacije se ubacuju dodatni shaderi za geometrijsko poboljšanje izgleda objekata. Nakon *fragment shadera* je poželjno dodati korak u kojemu će se obojani fragmenti spojiti sa susjednima radi dobivanja ljepše slike. (Slika 3.6.1.)

Na slici 3.6.2. prikazana je struktura koja grupira sve potrebne strukture i objekte za kreiranje cjevovoda. U nastavku su objašnjene funkcije pojedinih struktura i objekata.

```
VkGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;
pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = &depthStencil;
pipelineInfo.pColorBlendState = &colorBlending;
pipelineInfo.layout = pipelineLayout;
pipelineInfo.renderPass = renderPass;
pipelineInfo.subpass = 0;
pipelineInfo.basePipelineHandle = VK_NULL_HANDLE;
```

Slika 3.6.2. Struktura za kreiranje grafičkog cjevovoda

Polje struktura „*shaderStages*“ sadrži *VkPipelineShaderStageCreateInfo* strukture kojima se uključuju korišteni shaderi.

Za svaku varijablu unutar strukture točke *Vertex* potrebno je dodati jednu *VkVertexInputAttributeDescription* strukturu. Ove strukture su grupirane unutar jednoga vektora. Taj vektor je proslijeđen strukturi *„vertexInputInfo“* koja sadrži opis formata spremanja podataka o točkama koje će učitati *vertex shader*. Na slici 3.6.3. prikazano je ispunjavanje strukture

VkVertexInputAttributeDescription za varijablu *pos* strukture *Vertex*. Atribut *location* je poveznica s varijablom unutar shadera. Format mora dogovarati vrsti varijable. U ovome slučaju varijabla *pos* je trodimenzionalni vektor, stoga se u formatu nalazi *R*, *G* i *B*.

```
VkVertexInputAttributeDescription attributeDescription;
attributeDescription.binding = 0;
attributeDescription.location = 0;
attributeDescription.format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescription.offset = offsetof(Vertex, pos);
```

Slika 3.6.3. Primjer ispunjavanja strukture *VkVertexInputAttributeDescription*

Struktura *„inputAssembly“* specificira na koji način će se iscrtavati točke. Za određivanje ovoga je zadužen član *„topology“* navedenog objekta koji može imati vrijednosti *„VK_PRIMITIVE_TOPOLOGY_“* s nastavcima kao što su:

I. *POINT_LIST* – samo lista točaka
II. *LINE_LIST* – svaka crta ima svoje dvije točke, bez ponovnog korištenja
III. *LINE_STRIP* – druga točka svake crte je korištena kao prva točka iduće linije.
IV. *TRIANGLE_LIST* – svaki trokut ima svoje 3 točke, bez ponovnog korištenja
V. *TRIANGLE_STRIP* – druga i treća točka svakog trokuta su prve dvije idućeg trokuta.

Iz navedenog je moguće vidjeti da postoje dodatna mjesta za optimizaciju kod *LINE_STRIP* i *TRIANGLE_STRIP* načina, ali se mora voditi računa o redoslijedu točaka kod kreiranja samih modela.

Strukturom *„viewportState“* određuje se stanje prozora na zaslonu, tj. broj prozora i njihove dimenzije te broj škara (engl. *scissors*) kojima je moguće odrediti koji će dio pojedine slike biti prikazan na zaslonu.

Struktura *„rasterizer“* odlučuje o pretvaranju točaka u fragmente. S njome se može odrediti kako će se model prikazati; kao čvrsti, vidljivi model ili će se prikazati samo njegovi bridove (engl. *wireframe*). Također se bavi dubinskim testiranjem, tj. testiranjem koja je točka bliža kameri, odnosno točki pogleda.

Struktura *„multisampling“* specificira omogućavanje procesa poznatog kao *anti-aliasing*. Ovaj proces smanjuje grubost rubova prikazanih modela. Radi na način da kombinira više bliskih piksela različitih poligona u jedan piksel kojega prikazuje na zaslonu.

Struktura *„depthStencil“* određuje obradu dubinskih podataka (poglavlje 4.2).

Struktura *„colorBlending“* određuje hoće li se kombinirati boje piksela već prikazane slike s bojama piksela slike koja bi sljedeća trebala biti prikazana na zaslonu.

Objekt *„pipelineLayout“* sadrži poveznicu cjevovoda s deskriptorima (engl. *descriptor*). Deskriptori omogućavaju shaderima da slobodno pristupaju memorijskim resursima kao što su slike. Potrebno je odrediti njihov broj i referencu na njih prilikom kreiranja ovog objekta.

Objekt *„renderPass“* grupira informacije o slikovnim međuspremnicima (engl. *framebuffer*), koliko memorije treba zauzeti za podatke o boji i za dubinske podatke, koliko će uzoraka koristiti za njih te kako će biti rukovani tijekom operacija iscrtavanja na ekran. Svaki prolazak iscrtavanja (engl. *render pass*) može se koristiti samo sa jednim grafičkim cjevovodom i slikovnim međuspremnikom. Slikovni međuspremnik je memorija predviđena za spremanje kolekcije svih podataka jednog video okvira koje jedan prolazak iscrtavanja zahtjeva.

```
swapChainFramebuffers.resize(swapChainImageViews.size());
for (size_t i = 0; i < swapChainImageViews.size(); i++) {
    std::array<VkImageView, 2> attachments = {
        swapChainImageViews[i],
        depthImageView
    };

    VkFramebufferCreateInfo framebufferInfo{};
    framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferInfo.renderPass = renderPass;
    framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
    framebufferInfo.pAttachments = attachments.data();
    framebufferInfo.width = swapChainExtent.width;
    framebufferInfo.height = swapChainExtent.height;
    framebufferInfo.layers = 1;

    if (vkCreateFramebuffer(device, &framebufferInfo, nullptr,
        &swapChainFramebuffers[i]) != VK_SUCCESS) {
        throw std::runtime_error("failed to create framebuffer!");
    }
}
```

Slika 3.6.4. Postupak kreiranja slikovnih međuspremnika

Broj slikovnih međuspremnika je isti kao i broj slikovnih pogleda unutar lanca zamjenjivanja. Svaki slikovni međuspremnik kreiramo s pripadajućom strukturom. U tu strukturu je potrebno dodati objekt *renderPass,* potrebne slikovne poglede kao *attachments* te izjednačiti širinu i visinu slike slikovnog međuspremnika s lancem zamjenjivanja. (Slika 3.6.4.)

Vulkan dijeli prolazak iscrtavanja na jedan ili više pod-prolazak (engl. *subpass*) koji omogućuju optimizaciju iscrtavanja na mobilnim uređajima. U predzadnjem redku koda na slici 3.6.2, broj „0“ predstavlja indeks pod-prolaska. Vulkan omogućava da se grafički cjevovod derivira od drugog, što je moguće ostvariti dodavanjem reference roditeljskog cjevovoda na *basePipelineHandle*.

3.7. Shaderi

Kako bi se programirao grafički uređaj, koriste se shaderi. Shader je ograničeni model programiranja s posebnim jezikom i funkcionalnošću, dizajniran za paralelizam grafičkih uređaja i integraciju s grafičkim cjevovodom [7]. Grafički uređaji, za razliku od centralnih procesora sadrže puno jezgri koje omogućavaju obradu velike količine podataka u istome trenutku. U Vulkanu je smanjena nekonzistencija shadera tako da je svaki podatak unutar shadera formatiran u veličini jednoga byte-a.

```
1  #version 450
2  #extension GL_ARB_separate_shader_objects : enable
3
4  layout(binding = 0) uniform UniformBufferObject {
5      mat4 model;
6      mat4 view;
7      mat4 proj;
8  } ubo;
9
10 layout(location = 0) in vec3 inPosition;
11 layout(location = 1) in vec3 inColor;
12 layout(location = 2) in vec2 inTexCoord;
13
14 layout(location = 0) out vec3 fragColor;
15 layout(location = 1) out vec2 fragTexCoord;
16
17 void main() {
18     gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
19     fragColor = inColor;
20     fragTexCoord = inTexCoord;
21 }
```

Slika 3.7.1. Datoteka „shader.vert“

Na slici 3.7.1. nalazi se *vertex shader*. *UniformBufferObject* sadržava sve potrebne matrice za izračunavanje transformacijske matrice potrebne za prikaz trodimenzionalnog prostora na zaslonu. Vektor pozicije *inPosition* prihvaća trodimenzionalne koordinate točke. Ugrađena varijabla *gl_Position* predstavlja izlaz te je jednaka umnošku projekcijske, pogledne i model matrice s pozicijskim vektorom. Zadnji množitelj je četverodimenzionalni vektor s prva 3 člana definiranim s ulaznim *inPosition* vektorom i dodanom vrijednosti *1.0* kao četvrtim članom. Na ovaj način se uspješno izračunava nova pozicija modela prilikom osvježenja slike s novom pozicijom modela [8]. Podaci o boji i koordinate tekstura se samo prenose s ulaza *inColor* i *inTexCoord* na izlaze *fragColor* i *fragTexCoord*, respektivno.

```
struct UniformBufferObject {
    alignas(16) glm::mat4 model;
    alignas(16) glm::mat4 view;
    alignas(16) glm::mat4 proj;
};
```

Slika 3.7.2. Struktura *UniformBufferObject*

Na slici 3.7.2. prikazana je struktura U*niformBufferObject* koja ima identične parametre kao istoimena varijabla unutar *shader.vert*. Podaci ovog objekta povezani su sa shaderom tijekom izvođenja pomoću deskriptora, kao što je prikazano na slici 3.7.3.

```
VkDescriptorSetLayoutBinding uboLayoutBinding{};
uboLayoutBinding.binding = 0;
uboLayoutBinding.descriptorCount = 1;
uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uboLayoutBinding.pImmutableSamplers = nullptr;
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

VkDescriptorSetLayoutBinding samplerLayoutBinding{};
samplerLayoutBinding.binding = 1;
samplerLayoutBinding.descriptorCount = 1;
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerLayoutBinding.pImmutableSamplers = nullptr;
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

std::array<VkDescriptorSetLayoutBinding, 2> bindings = { uboLayoutBinding, samplerLayoutBinding };
VkDescriptorSetLayoutCreateInfo layoutInfo{};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
layoutInfo.pBindings = bindings.data();
```

Slika 3.7.3. Kreiranje *VkDescriptorSetLayout* objekta

Polje *bindings* sadrži sve poveznice shadera s deskriptorima. Prva poveznica *uboLayoutBinding* povezuje ovaj objekt s onime istoga imena unutar *vertex* shadera. Druga poveznica *samplerLayoutBinding* povezuje objekt *textureSampler* s drugim, *fragment* shaderom. Ovaj objekt daje *fragment* shaderu pristup teksturama prilikom iscrtavanja modela. (Slika 3.7.3.)

Na slici 3.7.4. prikazan je *fragment shader*. Izlaz ovog shadera je boja *outColor*. Prikaz boje na modelu ovisi o učitanoj teksturi preko objekta *texSampler*. Funkcijom *vkCreateSampler* kreira se objekt *textureSampler* uz dodanu strukturu *samplerInfo* kao argument, prikazana na slici 3.7.5. Ovaj objekt je preko deskriptora povezan s *fragment* shaderom te služi kao filtar teksture.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 1) uniform sampler2D texSampler;

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;

layout(location = 0) out vec4 outColor;

void main() {
    outColor = texture(texSampler, fragTexCoord);
}
```

Slika 3.7.4. Datoteka „shader.frag“

```
VkSamplerCreateInfo samplerInfo{};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = properties.limits.maxSamplerAnisotropy;
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
samplerInfo.unnormalizedCoordinates = VK_FALSE;
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
samplerInfo.mipLodBias = 0.0f;
samplerInfo.minLod = 0.0f;
samplerInfo.maxLod = 0.0f;
```

Slika 3.7.5. Struktura za kreiranje objekta *textureSampler*

Vrijednosti *magFilter* i *minFilter* određuju vrstu interpoliranja piksela učitane teksture u slučaju da ih ima manje od broja fragmenata, odnosno da fragmenata ima manje od potrebnog broja za prikaz cijele učitane teksture.

Kada je tekstura prikazana pod velikim kutem, detalji na njoj postaju mutni. Riješenje za ovo je anizotropijsko filtriranje. Vrijednost maksimalne anizotropije se uzima iz mogućnosti grafičkog uređaja.

Vrijednost *addressMode* specificira ponašanje teksture na modelu za svaku koordinatu teksture koja je izvan njenih granica. Mod *REPEAT* ponavlja sliku koliko god je potrebno. Neke od ostalih modova za *addressMode* predstavljaju mod *CLAMP_TO_EDGE* koji kopira zadnji piksel svakog redka, odnosno stupca na ostatak modela i CLAMP_TO_BORDER kopira boju obruba na ostatak modela, što bi bila crna boja jer je određena pomoću *borderColor*.

Granice su upravljane s *unnormalizedCoordinates* parametrom. *VK_FALSE* uzrokuje da su granice teksture 0 i 1, a VK_TRUE bi uzrokovao da budu 0 i maksimalna veličina pojedine dimenzije slike. Ostali parametri, kao što su *mipmap* i *lod* razine (engl. *level of detail*), pomažu u optimizaciji modela, kao što je upravljanje smanjivanjem učitane teksture na modelu prilikom odmicanja pogleda od modela. Ovi parametri postaju vrlo korisni u dizajniranju većih prostora s puno različitih modela kroz koje se korisnik aplikacije može kretati.

Za učitavanje vanjskih teksturnih datoteka korištena je vanjska biblioteka *stb_image.h*. Ovu biblioteku je moguće preuzeti na linku [9]. Potrebno je ponoviti korak II iz poglavlja „2. Postavljanje okruženja“ s pripadajućom putanjom do datoteke, a zatim dodati dvije linije koda prikazane na slici 3.7.6.

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

Slika 3.7.6. Uključivanje *stb_image* biblioteke

Učitavanje podataka teksture iz vanjske datoteke je prikazano na slici 3.7.7. Putanja do datoteke je definirana varijablom *TEXTURE_PATH*. Funkcija *stbi_load* dobavlja informacije o širini, visini i broju kanala koje tekstura posjeduje. Zadnji argument *STBI_rgb_alpha* označava da se uz

učitane podatke teksture doda *alpha* kanal, ako nije postojao u datoteci. *Alpha* kanal određuje prozirnost slike. Povratna vrijednost je pokazivač na prvi element u polju vrijednosti piksela. Ovi podaci se spremaju u klasnu varijablu za teksturu koju *fragment* shader koristi za bojanje modela.

```
int texWidth, texHeight, texChannels;
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight, &texChannels, STBI_rgb_alpha);
VkDeviceSize imageSize = 4 * texWidth * texHeight;

if (!pixels) {
    throw std::runtime_error("failed to load texture image!");
}
```

Slika 3.7.7. Učitavanje podataka teksture iz vanjske datoteke

Nakon promjene bilo koje shader datoteke, potrebno je ponovo prevesti (engl. *compile*) shadere pokretanjem datoteke prikazane na slici 3.7.8.

```
C:/VulkanSDK/1.2.162.0/Bin32/glslc.exe shader.vert -o vert.spv
C:/VulkanSDK/1.2.162.0/Bin32/glslc.exe shader.frag -o frag.spv
pause
```

Slika 3.7.8. Datoteka „compile.bat“

3.8. Naredbena memorija

(engl. *command buffers*)

Narebe u Vulkanu nisu izvedene direktno koristeći pozive funkcija. Potrebno je snimiti sve naredbe koje su željene unutar objekata za spremanje naredbi. Ovo omogućava sav posao postavljanja naredbi za iscrtavanje unaprijed te omogućava paralelizam. Nakon spremanja naredbi u naredbenu memoriju, potrebno ih je samo pozvati unutar glavne petlje.

Količina memorije potrebne za snimanje naredbi određena je brojem okvira unutar lanca zamjenjivanja. Prije snimanja naredbi potrebno je zauzeti memoriju za njih funkcijom *vkAllocateCommandBuffers*. Ovoj funkciji pridodaje se struktura unutar koje se specificira *commandPool*, objekt kojime se uključuje familija redova za snimanje naredbi. Snimanje naredbi započinje funkcijom *vkBeginCommandBuffers* te završava pozivom funkcije *vkEndCommandBuffers*.

Ove funkcije su pozvane unutar *for* petlje jer se naredbe snimaju za svaki okvir lanca zamjenjivanja zasebno. (Slika 3.8.1.)

```
commandBuffers.resize(swapChainFramebuffers.size());

VkCommandBufferAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.commandPool = commandPool;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandBufferCount = (uint32_t)commandBuffers.size();

if (vkAllocateCommandBuffers(device, &allocInfo, commandBuffers.data()) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate command buffers!");
}

for (size_t i = 0; i < commandBuffers.size(); i++) {
    VkCommandBufferBeginInfo beginInfo{};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

    if (vkBeginCommandBuffer(commandBuffers[i], &beginInfo) != VK_SUCCESS) {
        throw std::runtime_error("failed to begin recording command buffer!");
    }
```

Slika 3.8.1. Zauzimanje memorije za naredbe i početak snimanja naredbi

Između *vkBeginCommandBuffers* i *vkEndCommandBuffers* funkcija nalaze se sve funkcije prikazane na slici 3.8.2. Svaka naredba koja se snima započinje s *vkCmd*.

```
vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
    vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
    VkBuffer vertexBuffers[] = { vertexBuffer };
    VkDeviceSize offsets[] = { 0 };
    vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);
    vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);
    vkCmdBindDescriptorSets(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS,
        pipelineLayout, 0, 1, &descriptorSets[i], 0, nullptr);
    vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
vkCmdEndRenderPass(commandBuffers[i]);

if (vkEndCommandBuffer(commandBuffers[i]) != VK_SUCCESS) {
    throw std::runtime_error("failed to record command buffer!");
}
```

Slika 3.8.2. Snimanje naredbi u naredbenu memoriju

Funkcije *vkCmdBeginRenderPass* i *vkCmdEndRenderPass*, prikazane na slici 3.8.2. obilježavaju početak i kraj snimanja naredbi za jedan prolazak iscrtavanja. Između ovih funkcija nalaze se *vkCmdBindPipeline*, funkcija kojom se određuje koji će se grafički cjevovod koristiti prilikom izvođenja naredbi. Ostale funkcije koje slijede nakon nje su potrebne za ostvarivanje konačnog rezultata (poglavlje 4). Funkcije *vkCmdBindVertexBuffers* i *vkCmdBindIndexBuffers* povezuju spremnike memorije gdje se nalaze točke modela te indeksi tih točaka. Ovi spremnici su popunjeni ili ručno ili učitavanjem već postojećih podataka iz datoteka koje opisuju 3D modele. Funkcija *vkCmdBindDescriptorSets* povezuje korištene deskriptore s korištenim grafičkim cjevovodom. Funkcija *vkCmdDrawIndexed* odrađuje sami dio iscrtavanja točaka na ekran. Funkcija *vkCmdBeginRenderPass* prima strukturu *renderPassInfo* kao argument, prikazana na slici 3.8.3.

```
VkRenderPassBeginInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[i];
renderPassInfo.renderArea.offset = { 0, 0 };
renderPassInfo.renderArea.extent = swapChainExtent;
std::array<VkClearValue, 2> clearValues{};
clearValues[0].color = { 0.0f, 0.0f, 0.0f, 1.0f };
clearValues[1].depthStencil = { 1.0f, 0 };
renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());
renderPassInfo.pClearValues = clearValues.data();
```

Slika 3.8.3. Davanje informacija o prolasku iscrtavanja

Objekt *renderPass* predstavlja korišteni objekt prolaska iscrtavanja. Polje *swapChainFrameBuffers* sadrži sve okvire lanca zamjenjivanja nad kojima će se naredbe izvršavati. Iduća dva parametra, *offset* i *extent* definiraju pomak i veličinu površine za iscrtavanje. Polje *clearValues* sadrži određene vrijednosti koje se postavljaju prilikom čišćenja površine za iscrtavanje. Prvi element sadrži boju postavljenu na crnu boju s maksimalnom prozirnošću. Drugi element se odnosi na čišćenje dubinskih podataka (poglavlje 4.2.). (Slika 3.8.3.)

3.9. Sinkronizacija

Funkcija *drawFrame* dobavlja sliku iz lanca zamjenjivanja, zatim obavlja željene naredbe spremljene u naredbenoj memoriji nad tom slikom te nakon toga vraća sliku lancu zamjenjivanja. Pozvana je unutar glavne petlje.

```
void VulkanApplication::mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();
    }
    vkDeviceWaitIdle(device);
}
```

Slika 3.9.1. Glavna petlja programa

Na slici 3.9.1. prikazana je glavna petlja programa. Funkcija *glfwPollEvents* provjerava događaje kao što su unos na tipkovnici i pritisak miša. Funkcija *vkDeviceWaitIdle* je potrebna za pravilno zatvaranje programa.

Funkcija *mainLoop* je pozvana nakon inicijalizacije svih prijašnje navedenih čimbenika unutar funkcija *initWindow* i *initVulkan*. Kada program završi, kada izađe iz glavne petlje, pozvana je funkcija *cleanup* unutar koje se oslobađa sva zauzeta memorija programa. (Slika 3.9.2.)

```
void VulkanApplication::run() {
    initWindow();
    initVulkan();
    mainLoop();
    cleanup();
}
```

Slika 3.9.2. Redoslijed obavljanja glavnih zadataka programa

Poslovi koje obavlja funkcija *drawFrame* trebaju biti slijedni, ali funkcije koje poziva završavaju svoje poslove asinkrono. U svrhu sinkronizacije poslova koje obavlja potrebno je dodati semafore. Jedna vrsta semafora će signalizirati kada je slika dobivena od lanca zamjenjivanja, a druga kada je slika spremna za iscrtavanje. Pojedini semafor unutar navedenih vrsta je zadužen za

sinkronizaciju obrade jednog slikovnog okvira od slika „u letu“ (engl. *in-flight*). Slike „u letu“ predstavljaju sve slike koje se paralelno obrađuju radi optimalnije iskorištenosti resursa.

Funkcija *vkAcquireNextImageKHR* dobavlja sliku lancu zamjenjivanja. Četvrti argument unutar poziva ove funkcije predstavlja odabrani semafor iz spremljenog polja semafora vrste *imageAvailableSemaphores*. (Slika 3.9.3.)

```
VkResult result = vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
    imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);
```

Slika 3.9.3. Dobavljanje sljedeće slike

Na slici 3.9.4. prikazana je struktura *submitInfo* unutar koje je specificirano čekanje *imageAvailableSemaphores* semafora. Ova struktura pridodaje se pozivu funkcije *vkQueueSubmit* koja tada predaje naredbe iz naredbene memorije za izvršavanje. Polje semafora *signalSemaphores* sadrži sve semafore koji će biti obaviješteni kada se naredbe izvrše.

```
VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

VkSemaphore waitSemaphores[] = { imageAvailableSemaphores[currentFrame] };
VkPipelineStageFlags waitStages[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;

submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffers[imageIndex];

VkSemaphore signalSemaphores[] = { renderFinishedSemaphores[currentFrame] };
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
```

Slika 3.9.4. Postavljanje semafora

Vraćanje obrađene slike lancu zamjenjivanja prikazano je na slici 3.9.5. Funkciji *vkQueuePresentKHR* pridodaje se struktura unutar koje je specificiran lanac zamjenjivanja te index slike. Njoj je također kroz strukturu naređeno da čeka iste one *signalSemaphores* semafore prije izvedbe.

```
VkPresentInfoKHR presentInfo{};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = signalSemaphores;

VkSwapchainKHR swapChains[] = { swapChain };
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = swapChains;

presentInfo.pImageIndices = &imageIndex;

result = vkQueuePresentKHR(presentQueue, &presentInfo);
```

Slika 3.9.5. Vraćanje obrađene slike

Do sada je objašnjeno kako se sinkroniziraju poslovi obrade pojedine slike pomoću semafora, no potrebno je još dodati i ograde (engl. *fences*). Ograde su druga vrsta sinkronizacijskih objekata kojima se operacije obavljane na grafičkom uređaju sinkroniziraju s procesorom. Na slici 3.9.6. prikazano je upravljanje ogradama. Zakomentirani dijelovi koda na ovoj slici predstavljaju linije koda prikazane na slikama 3.9.3, 3.9.4 i 3.9.5, slijedno.

```
vkWaitForFences(device, 1, &inFlightFences[currentFrame], VK_TRUE, UINT64_MAX);

// ...

if (imagesInFlight[imageIndex] != VK_NULL_HANDLE) {
    vkWaitForFences(device, 1, &imagesInFlight[imageIndex], VK_TRUE, UINT64_MAX);
}
imagesInFlight[imageIndex] = inFlightFences[currentFrame];

// ...

vkResetFences(device, 1, &inFlightFences[currentFrame]);

if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit draw command buffer!");
}

// ...

currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
```

Slika 3.9.6. Upravljanje ogradama

Funkcija *vkWaitForFences* blokira program sve dok nije obaviještena da propusti program. U ovome slučaju funkcija *vkQueueSubmit* obaviještava ogradu unutar polja *inFlightFences* za trenutni okvir *currentFrame* da nastavi odvijanje programa. Potrebno je svaku sliku unutar lanca zamjenjivanja pratiti kako ne bi došlo do uzimanja slike koja je već „u letu“. Ovo se postiže drugim poljem ograda *imagesInFlight*, čija se ograda trenutne slike u obradi indeksa *imageIndex* postavlja da blokira program dok ne postane slobodna. Trenutna slika „u letu“ se označava da je korištena izjednačavanjem ograde koja ju predstavlja s ogradom *inFlightFences* trenutnog okvira *currentFrame*. Funkcija *vkResetFences* postavlja danu ogradu na zadanu, nesignaliziranu vrijednost. Na samom kraju funkcije se brojač trenutnog okvira *currentFrame* pomiče za jedan. Kada iduća vrijednost brojača bude jednaka maksimalnom broju okvira „u letu“, tada se pomoću modulo operatora % vraća na vrijednost nula, to jest prvi okvir. (Slika 3.9.6.)

Na slici 3.9.7. prikazana je provjera valjanosti lanca zamjenjivanja. Ova provjera se obavlja nakon poziva funkcija *vkAcquireNextImageKHR* (tada radi samo prvu provjeru prikazanu unutar *if* funkcije sa slike) i *vkQueuePresentKHR* unutar funkcije *drawFrame* tako da provjerava povratne vrijednosti istih, pomoću varijable *result*. Ukoliko rezultat *result* javi da je lanac zamjenjivanja *OUT_OF_DATE* ili *SUBOPTIMAL*, ili ako dođe do promjene veličine dimenzija okvira, tada se poziva funkcija *recreateSwapChain* koja briše postojeći lanac te kreira novi s odgovarajućim parametrima. Ovo će se dogoditi kada dođe do promjene dimenzija prozora aplikacije, na primjer uzrokovane korisnikovom promjenom.

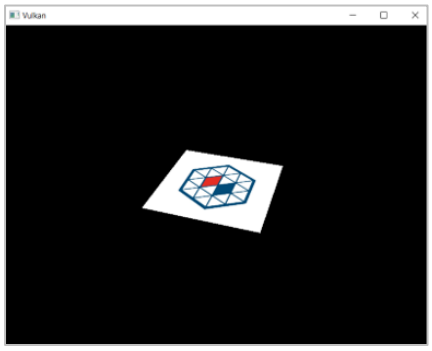
```
if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR || framebufferResized) {
    framebufferResized = false;
    recreateSwapChain();
}
else if (result != VK_SUCCESS) {
    throw std::runtime_error("failed to present swap chain image!");
}
```

Slika 3.9.7. Provjera valjanosti lanca zamjenjivanja

4. REZULTATI

Uz dane podatke o točkama i indeksima prikazanih na slici 4.1, pokretanje programa iscrtava kvadrat u prozoru kao što je prikazano na slici 4.2.

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}},
};

const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0,
};
```



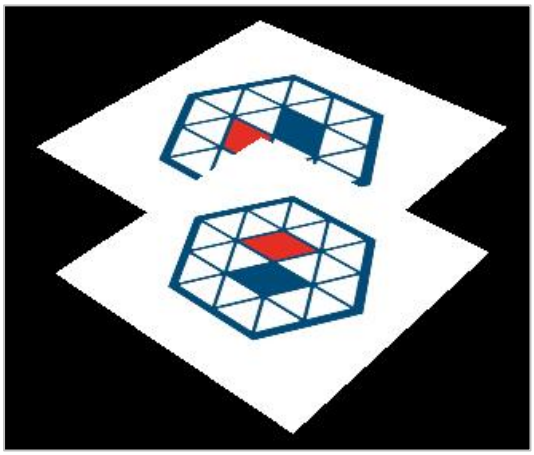
Slika 4.1. Točke i indeksi za iscrtavanje kvadrata u 3D prostoru

Slika 4.2. Iscrtani kvadrat

4.1. Prijelaz na 3D koordinate

Dodavanjem treće, Z koordinate unutar vrijednosti vektora *vertices* dolazi do pogreške. Na slici 4.1.2. novounešena donja ploha se nalazi iznad gornje plohe. Razlog ove greške je jednostavan. Nove točke koje su dodane u polje točaka se unutar njega nalaze nakon onih koje definiraju gornju plohu. Postoji 2 načina za rješavanje ovoga problema. Jedan je sortiranje poziva iscrtavanja po dubini, od „iza“ prema „naprijed“. Drugi je testiranjem dubine pomoću dodatnih podataka unutar kojih se pamti dubina svake točke. Drugi način je puno češće korišten.

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}},

    {{-0.5f, -0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, -0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, -0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};

const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0,
    4, 5, 6, 6, 7, 4
};
```



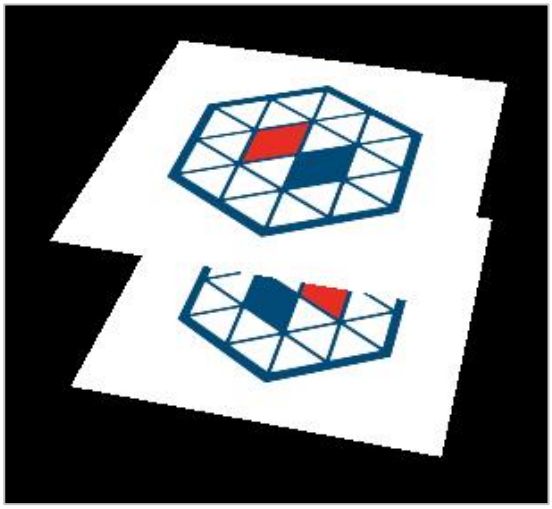
Slika 4.1.1. Dodavanje treće koordinate

Slika 4.1.2. Prikaz ploha

4.2. Dodavanje dubinskih podataka

(engl. *Depth buffering*)

Svakoj točki pridodaje se dodatni podatak njene dubine. Vulkan obilježava dubinu decimalnim vrijednostima u rasponu od 0.0 do 1.0, za razliku od OpenGL koji obilježava u rasponu od -1.0 do 1.0. Iz ovoga razloga je potrebno definirati *GLM_FORCE_DEPTH_ZERO_TO_ONE* na početku programa. Potrebna je definicija samo jedne slike i pogleda slike za spremanje dubinske slike, jer se u jednom trenutku odvija samo jedna operacija iscrtavanja. Rezolucija slike je jednaka onoj kod opsega zamjena. Svaki put kada je fragment kreiran tijekom postupka rasterizacije, test dubine će provjeriti ako je novi fragment bliže nego prethodni. Ako nije, onda se odbacuje, ali ako je, tada se njegova vrijednost dubine zapisuje u objekt dubinske slike. Krajnji rezultat pokretanja programa nakon dodavanja dubinskh podataka prikazan je na slici 4.2.1.



Slika 4.2.1. Prikaz ploha nakon dodavanja dubinskih podataka

4.3. Učitavanje gotovog 3D modela

Podaci 3D modela mogu biti spremljeni unutar raznih podatkovnih tipova, kao što su *wavefront (obj)*, *collada* (*dae*) i *STL*. Od navedenih, za svrhu ovoga rada učitati će se *wavefront* datoteka pomoću biblioteke *tinyobjloader* koju je moguće preuzeti na linku [10]. Potrebno je ponoviti korak II iz poglavlja „2. Postavljanje okruženja“ sa putanjom do vanjske „*master*“ mape.

```
tinyobj::attrib_t attrib;
std::vector<tinyobj::shape_t> shapes;
std::vector<tinyobj::material_t> materials;
std::string warn, err;

if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &warn, &err, MODEL_PATH.c_str())) {
    throw std::runtime_error(warn + err);
}
```

Slika 4.3.1. Učitavanje podataka iz vanjske *wavefront* datoteke

Funkcija *LoadObj* učitava podatke iz datoteke putanje *MODEL_PATH* te ih sprema u varijable *attrib, shapes, materials, warn* i *err*. (Slika 4.3.1.)

```
std::unordered_map<Vertex, uint32_t> uniqueVertices{};
for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex{};
        vertex.pos = {
            attrib.vertices[(int64_t)3 * index.vertex_index + 0],
            attrib.vertices[(int64_t)3 * index.vertex_index + 1],
            attrib.vertices[(int64_t)3 * index.vertex_index + 2]
        };
        vertex.texCoord = {
            attrib.texcoords[(int64_t)2 * index.texcoord_index + 0],
            1.0f - attrib.texcoords[(int64_t)2 * index.texcoord_index + 1]
        };
        vertex.color = { 1.0f, 1.0f, 1.0f };
        if (uniqueVertices.count(vertex) == 0) {
            uniqueVertices[vertex] = static_cast<uint32_t>(vertices.size());
            vertices.push_back(vertex);
        }
        indices.push_back(uniqueVertices[vertex]);
    }
}
```

Slika 4.3.2. Spremanje učitanih podataka u klasne varijable

Za svaki poligon 3D modela *shape* se prolazi kroz svaku njegovu točku. Podaci svake točke, kao što su pozicija, koordinate teksture i boja točke spremaju se u *Vertex* varijablu. Ova varijabla se zatim pridodaje klasnoj varijabli, vektoru *vertices* ako već ne sadrži identičnu točku. Ovim korakom se uklanjaju duplicirani podaci točaka. Na kraju se pomoćnom varijablom *uniqueVertices* pridodaju unikatne točke klasnoj varijabli, vektoru *indices*. Ovaj vektor sprema indekse točaka koji se kasnije koriste tijekom snimanja naredbi za iscrtavanje. (Slika 4.3.2.)

Na slici 4.3.3. prikazan je jedan iscrtani učitani gotovi model.

Slika 4.3.3. Učitani gotovi model

4.4. Dodavanje izvora svjetlosti

Učitani model izgledati će realističnije nakon što se doda izvor svjetlosti u njegov prostor. Ovo je moguće ostvariti dodavanjem množitelja kod izračuna boje fragmenta *out_color* unutar *fragment* shadera, koji će ovisno o kutu upada svjetla na plohe modela mijenjati boju prikaza teksture.

```
const vec3 DIRECTION_TO_LIGHT = normalize(vec3(1.0,7.0,3.0));

void main() {
    vec3 normalWorldSpace = normalize(mat3(ubo.model) * inNormal);
    float lightIntensity = max(dot(normalWorldSpace, DIRECTION_TO_LIGHT), 0);
```

Slika 4.4.1. Računanje množitelja boje unutar *vertex* shadera

Ovaj množitelj *lightIntensity* izračunat je unutar *vertex* shadera postupkom prikazanim na slici 4.4.1, a zatim postavljen kao izlaz kako bi bio prenesen *fragment* shaderu. Vrijednost te varijable jednaka je vektorskom umnošku vektora upada zraka svjetlosti *DIRECTION_TO_LIGHT* i vektora normale pojedine točke *inNormal* pomnožene s *ubo.model* kako bi se dobio trenutni smjer vektora normale točke unutar aktivnog 3D prostora. Ako je rezultat ovog vektorskog umnoška manji od nule, tada se uzima 0 kao rezultat. Smjer zraka svjetlosti je jednostavno postavljen na određenu vrijednost unutar *vertex* shadera tako da odgovara prikazu gotovog modela prikazanog na slici 4.4.4. Ova vrijednost bi se mogla postaviti unutar koda te unijeti u shader kao još jedan ulaz. Na ovaj način bi se mogla simulirati dinamična sunčeva svjetlost tijekom dana.

Osim izmjena shadera, potrebno je dodati varijablu vektora normale u strukturu *Vertex* te učitati istu iz datoteke gotovog modela unutar funkcije *loadModel*. (Slika 4.4.2.)

```
vertex.normal = {
    attrib.normals[(int64_t) 3 * index.normal_index + 0],
    attrib.normals[(int64_t) 3 * index.normal_index + 1],
    attrib.normals[(int64_t) 3 * index.normal_index + 2]
};
```

Slika 4.4.2. Dodavanje učitavanja vektora normale u funkciju *loadModel*

Potrebno je i povezati varijablu normale unutar shadera s onom u kodu dodatnom *VkVertexInputAttributeDescription* strukturom prilikom kreiranja grafičkog cjevovoda. (Slika 4.4.3.)

```
attributeDescriptions[3].binding = 0;
attributeDescriptions[3].location = 3;
attributeDescriptions[3].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[3].offset = offsetof(Vertex, normal);
```

Slika 4.4.3. Dodavanje poveznice sa shaderom za vektor normale

Na slici 4.4.4. prikazan je učitani gotovi model s dodanim izvorom svjetlosti unutar 3D prostora kojemu pripada. Može se primjetiti kako zrake upadaju s desne strane slike, prema dolje.

Slika 4.4.4 Učitani gotovi model sa izvorom svjetlosti

Ovaj način generiranja svjetlosti prikazuje 3D model puno realnijim, ali nije u potpunosti točan. Na tamnoj stranici modela sanduka je vidljivo da su izbočene površine osvijetljene, iako ne bi trebale biti. Ovo bi se moglo ispraviti pomoću dodatne dubinske mape, ali za svjetlosni izvor. One površine modela koje svjetlosni izvor „ne vidi“ bi bile označene za iscrtavanje sjene.

4.5. Oslobađanje zauzete memorije

Kako Vulkan zahtjeva od programera da bude eksplicitan u svemu što radi u kodu, tako je potrebno u trenutku zatvaranja aplikacije ručno obrisati svu zauzetu memoriju. Svaki Vulkan objekt koji se kreira mora biti eksplicitno uništen pripadajućom funkcijom kada više nije potreban, baš kao što je u C++ jeziku potrebno pozvati funkciju *delete* nakon prestanka korištenja alocirane memorije.

Na slici 4.5.1. prikazana je funkcija *cleanup* koja se poziva nakon prekida glavne petlje. Unutar funkcije *cleanup* pozivaju se funkcije za oslobađanje zauzete memorije potrebnih objekata za funkcioniranje aplikacije. Ove funkcije je potrebno pozvati obrnutim redoslijedom od poziva za kreiranje istih objekata jer na ovaj način neće doći do grešaka. Na primjer, objekt logičkog uređaja *device* treba biti obrisan nakon što se obrišu svi objekti povezani s njime.

```
void VulkanApplication::cleanup() {
    cleanupSwapChain();
    vkDestroySampler(device, textureSampler, nullptr);
    vkDestroyImageView(device, textureImageView, nullptr);
    vkDestroyImage(device, textureImage, nullptr);
    vkFreeMemory(device, textureImageMemory, nullptr);
    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);
    vkDestroyBuffer(device, indexBuffer, nullptr);
    vkFreeMemory(device, indexBufferMemory, nullptr);
    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);
    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
        vkDestroyFence(device, inFlightFences[i], nullptr);
    }
    vkDestroyCommandPool(device, commandPool, nullptr);
    vkDestroyDevice(device, nullptr);
    if (enableValidationLayers) {
        DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
    }
    vkDestroySurfaceKHR(instance, surface, nullptr);
    vkDestroyInstance(instance, nullptr);
    glfwDestroyWindow(window);
    glfwTerminate();
}
```

Slika 4.5.1. Oslobađanje zauzete memorije

Na slici 4.5.2. prikazana je funkcija *cleanupSwapChain* koja se osim prilikom zatvaranja programa poziva i kada je potrebno ponovo kreirati lanac zamjenjivanja.

```
void VulkanApplication::cleanupSwapChain() {
    vkDestroyImageView(device, depthImageView, nullptr);
    vkDestroyImage(device, depthImage, nullptr);
    vkFreeMemory(device, depthImageMemory, nullptr);
    for (auto framebuffer : swapChainFramebuffers) {
        vkDestroyFramebuffer(device, framebuffer, nullptr);
    }
    vkFreeCommandBuffers(device, commandPool,
        static_cast<uint32_t>(commandBuffers.size()), commandBuffers.data());
    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    vkDestroyRenderPass(device, renderPass, nullptr);
    for (auto imageView : swapChainImageViews) {
        vkDestroyImageView(device, imageView, nullptr);
    }
    vkDestroySwapchainKHR(device, swapChain, nullptr);
    for (size_t i = 0; i < swapChainImages.size(); i++) {
        vkDestroyBuffer(device, uniformBuffers[i], nullptr);
        vkFreeMemory(device, uniformBuffersMemory[i], nullptr);
    }
    vkDestroyDescriptorPool(device, descriptorPool, nullptr);
}
```

Slika 4.5.2. Oslobađanje memorije lanca zamjenjivanja

5. ZAKLJUČAK

Kako bi se iscrtao jednostavan 3D model (npr. model kocke) na zaslonu unutar Vulkan okruženja, potrebno je napisati puno više linija koda nego unutar *OpenGL* sučelja. Također je potrebno eksplicitno rukovati memorijom. Ovo može biti odbojno za programera, ali upoznavanje sa Vulkan sučeljem se svakako isplati. Starija sučelja sa svakim novim ažuriranjima postaju sve kompliciranija jer imaju potrebu održati kompatibilnost sa njihovim starijim verzijama. Također previše ovise o programskoj podršci proizvođača grafičkih kartica. Vulkan je budućnost grafičkih sučelja niske razine zbog brojnih prednosti koje nudi. Svaka operacija unutar Vulkan okruženja se eksplicitno obavlja što znatno poboljšava razumljivost koda i pomaže u pronalaženju grešaka. Komunikacija s grafičkim uređajima je direktna te je sučelje jedinstveno za sve vrste uređaja. Postižu se veće brzine zbog mogućnosti dijeljenja poslova na više procesorskih jezgri te mogućnosti onemogućavanja provjera grešaka. Nakon što se postave ključni čimbenici Vulkana za iscrtavanje, ostatak kodiranja, kao što je izrada novih shadera, dijeli mnogo sličnosti kao ono unutar *OpenGL* grafičkog okruženja.

U ovome radu objašnjeni su osnovni čimbenici potrebni za iscrtavanje slika 3D modela unutar jednoga prozora radi upoznavanja s grafičkim sučeljem. Kao nastavak ovoga rada mogla bi se dodati mogućnost učitavanja više od jednoga modela i pomicanja kamere korisničkim unosom u svrhu izrade interaktivnog, virtualnog svijeta. Osim toga vrijedilo bi posjetiti naprednije postavke Vulkan čimbenika, kao što su deriviranje grafičkog cjevovoda iz drugoga, isprobati druge vrste primitivne topologije (određenih strukturom *inputAssembly* prilikom definiranja grafičkog cjevovoda) te usporediti iskorištenost memorije različitih topologija, dodati geometrijske shadere, podijeliti poslove na više od jednog grafičkog uređaja i još mnogo toga.

6. LITERATURA

[1] Download VulkanSDK, LunarG, dostupno na: https://vulkan.lunarg.com/ [28.2.2021.]

[2] Vulkan specification, The Khronos Group Inc, dostupno na: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html [23.8.2021.]

[3] A. Overvoorde, Vulkan tutorial, dostupno na: https://vulkan-tutorial.com/ [23.8.2021.]

[4] GLFW Vulkan support reference, dostupno na: https://www.glfw.org/docs/latest/group__vulkan.html [24.8.2021.]

[5] C++ optional, dostupno na: https://en.cppreference.com/w/cpp/utility/optional [29.7.2021.]

[6] C++ vector, dostupno na: https://en.cppreference.com/w/cpp/container/vector [13.3.2021.]

[7] Render pipeline, dostupno na: https://vkguide.dev/docs/chapter-2/vulkan_render_pipeline/ [30.7.2021.]

[8] Matrices, Opengl-tutorials, dostupno na: https://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/ [30.7.2021.]

[9] S. Barrett, *stb_image* library, dostupno na: https://github.com/nothings/stb [23.5.2021.]

[10] S. Fujita, *tinyobjloader* library, dostupno na: https://github.com/tinyobjloader/tinyobjloader, [30.6.2021.]

7. SAŽETAK

Vulkan je grafičko aplikacijsko programsko sučelje (API) nove generacije kojemu je cilj riješiti temeljne probleme s kojima se druga sučelja susreću. Sučelje je koje funkcionira na bilo kojem uređaju koje ima grafičke mogućnosti. Postupak pripremanja grafičkog cjevovoda za iscrtavanje slika 3D modela je nešto dulji od sučelja prošlih generacija, ali nudi mnoge dodatne mogućnosti za poboljšanje rada aplikacija. Poboljšanjem abstrakcija objekata i njihove kontrole, Vulkan pruža programerima mogućnost jasnog specificiranja njihovih namjera, a time i postizanje boljih performansi.

Ključne riječi: 3D model, grafički cjevovod, Vulkan

8. ABSTRACT

Vulkan is the next generation graphics application programming interface (API) with a purpose to solve core issues other APIs cannot solve. It is an API that can function on any type of a device with graphics capabilities. The preparation of graphics pipeline for rendering images of 3D models is somewhat longer than by using previous interfaces, but it offers additional posibilities for the improvement of application's workflow. With improvement of object abstraction and control, Vulkan offers programmers the possibility to clearly specify their intentions and achieve the better performance with it.

Keywords: 3D model, graphics pipeline, Vulkan