Debagovanje i profajliranje GLSL šejder programa

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Marko Spasić, 1014/2020
mi16165@alas.matf.bg.ac.rs

10. avgust 2021.

Sažetak

Debagovanje i profajliranje programa koji se izvršava na jednom jezgru procesora sa jasno definisanom arhitekturom skupa instrukcija je teško samo po sebi. Kada program koji se izvršava na procesoru komunicira sa grafičkom karticom i izdaje naredbe za crtanje objekata koji se renderuju tako što se na grafičkoj kartici pokreću drugi programi, koji se izvršavaju na čipu drugačije arhitekture skupa instrukcija, onda se debagovanje i profajliranje tih programa na grafičkoj kartici dodatno komplikuje. Srećom, postoje tehnike i alati koji olakšavaju ovaj posao. Tehnike debagovanja predstavljene u ovom radu specifične su za OpenGL, dok se predstavljeni alati mogu koristiti i za DirectX i Vulkan.

Sadržaj

1 Uvod

Rad opisuje tehnike i alate koji mogu biti od pomoći u debagovanju i profajliranju programa koji se izvršavaju na grafičkoj kartici.

Podeljen je u tri poglavlja:

- Glavni problemi debagovanja i profajliranja GLSL programa
- OpenGL i GLSL tehnike
- Alati

U poglavlju Glavni problemi debagovanja i profajliranja GLSL programa je opisan proces pisanja, prevođenja i izvršavanja programa pisanih u GLSL jeziku, poznatih kao šejderi kao uvod u probleme koji nastaju ovakim dizajnom grafičke protočne obrade.

Nakon toga u poglavlju OpenGL i GLSL tehnike opisane su tehnike debagovanja GLSL [1] programa bez pomoćnih alata korišćenjem funkcionalnosti OpenGL-a i GLSL-a. Ovde se rad bavi funkcionalnostima koje su ugrađene u sam GLSL i OpenGL. Konkretno: korišćenje ugrađenih GLSL promenljivih i funkcija, OpenGL funkcija za debagovanje i tehnike vizuelnog debagovanja.

Poglavlje Alati je pregled spoljnih alata koji mogu biti od pomoći prilikom debagovanja i profajliranja GLSL programa. Kao što za programe pisane u C++ jeziku postoji mnoštvo debagera i profajlera isto tako postoji što komercijalnih proizvoda zatvorenog koda, tako i alata otvorenog koda za debagovanje i profajliranje programa pisanih u GLSL jeziku.

Cilj rada je da čitaoca upozna sa tehnikama i alatima za debagovanje i profajliranje šejder programa tako da kada se nađe u situaciji da šejder program ne proizvodi željenje rezultate, na raspolaganju ima mnoštvo načina za rešavanje problema.

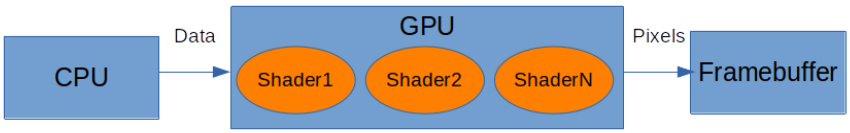
2 Glavni problemi debagovanja i profajliranja GLSL programa

Današnji računari sadrže dva čip sličnog nivoa kompleksnosti: procesor (eng. *Central Porcessing Unit (CPU)*) i grafičku karticu (eng. Graphics Processing Unit (GPU)). Procesor i grafička kartica nemaju istu arhitekturu skupa instrukcija (eng. *Instruction Set Arhitecture (ISA)*) i instrukcije na jednom čipu se izvršavaju nezavisno od instrukcija na drugom čipu.

Na slici 1 ilustrovan je tok podataka od procesora do ekrana. Procesor šalje podatke iz glavne memorije na grafičku karticu. Zatim postavlja OpenGL kontekst pozivima odgovarajućih funkcija i izdaje naredbu za crtanje objekta šejder programima koji se nalaze na grafičkoj kartici. Ti šejder programi obrade prosleđene podatke i popune boje piksela u frejmbaferu koji se nakon toga šalje na uređaj za prikaz slike.

Dodatna poteškoća je to što se za programiranje grafičkih kartica preko OpenGL skupa funkcija koristi još jedan programski jezik sintakse nalik na C jeziku, GLSL. Na to, trenutno na tržištu postoje tri velika vendora grafičkih kartica: NVidia, AMD i Intel[16]. Za svaki od ta tri vendora podržana su četiri različita aplikaciono programska interfejsa (eng. *Application Programming Interface (API)*) u C jeziku: Direct3D (za Windows), Metal (Apple), Vulkan (Windows, Linux), OpenGL (Windows, Linux, Apple[1]). Zato što svaki vendor grafičkih kartica implementira specifikaciju

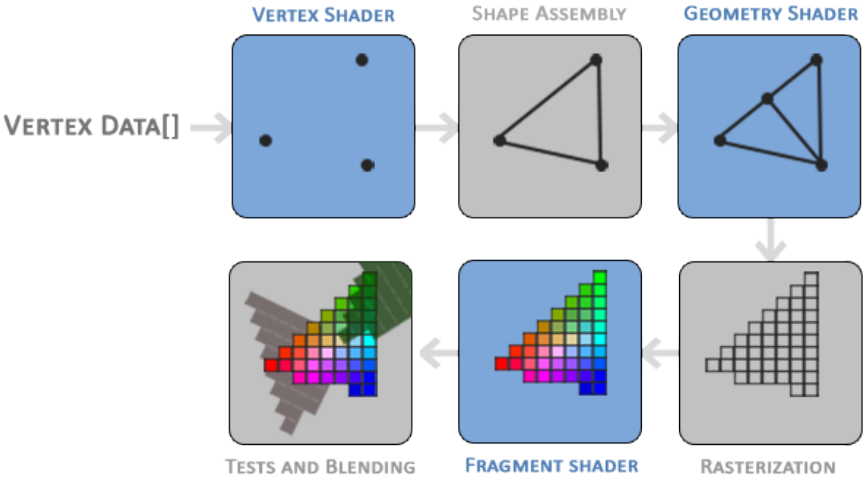
[1] Apple drops opengl support news



Slika 1: Apstraktna reprezentacija toka podataka od procesora do ekrana pri izdavanju komandi za crtanje

API-ja, ponašanje može da varira od platforme do platforme.[2]

OpenGL model ima fiksnu grafičku protočnu obradu, koja ima programabilne delove. Delovi grafičke protočne obrade koji su na slici 2 obeleženi



Slika 2: Apstraktna reprezentacija grafičke protočne obrade OpenGL-a[13]

plavom su programabilni. Programer piše kod za vrteks šejder, geometrijski šejder i fragment šejder. Ta tri izvorna koda se zatim tokom izvršavanja glavnog programa kompajliraju pozivima OpenGL funkcija i linkuju u jedan šejder program. Tako dobijeni šejder program se nalazi na grafičkoj kartici i može se koristiti za crtanje nekog objekta.

Svaka od ovih stavki otežava pravljenje alata i debagovanje šejder programa. Vendor implementira specifikaciju API-ja i nije moguće generisati kod za grafičku karticu kao što je moguće generisati kod za Intel x86-64 arhitekturu nego je neophodno koristiti API. Ponašanje može varirati od platforme do platforme u zavisnosti od načina korišćenja API-ja. Program koji se izvršava na grafičkoj kartici (šejder) kompajlira sama aplikacija. Kao rezultat se ne dobije generisan kod, kao kada se kompajlira C program na Intel x86-64 arhitekturu, već glavna aplikacija dobije neki identifikator koji služi kao referenca na šejder program koji se nalazi na

[2]Kada bi svaki vendor imao svoj API programiranje grafičkih kartica bi bilo jednostavnije

grafičkoj kartici u formatu koji odgovara vendoru te grafičke kartice.

Uprkos svim ovim poteškoćama, postoje tehnike i alati koji programerima grafičkih kartica koji rade sa OpenGL API-jem (i drugim) olakašavaju proces debagovanja i profajliranja.

3 OpenGL i GLSL tehnike

OpenGL specifikacija definiše nekoliko funkcija za rad sa greškama. Glavna aplikacija poziva ove OpenGL funkcije kako bi dobila informacije da li je u nekom trenutku u grafičkoj protočnoj obradi došlo do greške i ako jeste koja od predefinisanih grešaka se dogodila. Pozivi ovih funkcija se pišu tako da budu prevedeni samo u debag modu, jer njihovo pozivanje može uticati na performanse ako se koriste često u glavnoj petlji renderovanja.

Pored tih funkcija postoje predefinisane promenljive GLSL-a koje se po specifikaciji koriste u tačno određenu svrhu u samoj grafičkoj protočnoj obradi. Te ugrađene promenljive se mogu iskoristiti za vizuelno debagovanje. Vizuelno debagovanje je za grafiku ono što bi printf debagovanje bilo u većini aplikacija. Radi tako što se u zavisnosti od stanja ostalih promenljivih u šejder programu određena predefinisana GLSL promenljiva postavi na neku ekstremnu vrednost. Ta postavljena ekstremna vrednost će onda proizvesti neki veoma uočljiv vizuelni efekat na objektu čije crtanje se debaguje i to će biti indikacija o mestu greške u šejder programu koji crta taj objekat.

Od funkcija za debagovanje koje se nalaze u OpenGL-u u daljem radu navedena su dva skupa funkcija. Stari glGetError[2], koji se koristi u slučaju da na datom sistemu nema verzija OpenGL Core 4.3 i novi skup funkcija dostupan od verzije OpenGL Core 4.3[7] koji proizvodi preciznije opise i tačne lokacije grešaka.

Od predefinisanih promenljivih GLSL-a obično se koriste gl_FragCoord u kombinaciji sa izlaznom bojom iz fragment šejdera jer je tako najlakše uslovno obojiti neki fragment kako bi eventualna greška u izračunavanjima bila uočljiva.

3.1 glGetError

Debagovanje OpenGL API-ja korišćenjem glGetError funkcije je jednostavno za korišćenje, ali ne i preterano korisno van najbazičnijih situacija. OpenGL API je apstrakcija nad automatom stanja. Pozivi OpenGL funkcija mogu da postave ili koriste neko stanje. Crtanje objekata pomoću OpenGL-a se obično odvija tako što se podaci potrebni za crtanje objekta pošalju na grafičku karticu i postave odgovarajuća OpenGL stanja, koja se zajedno nazivaju OpenGL kontekst. Zatim se pozove funkcija koja prethodno postavljena stanja koristi da nacrta objekat.

Stari način za debagovanje, pomoću glGetError funkcije, radi tako što OpenGL postavlja greške koje su se javile tokom izvršavanja na stek grešaka. Taj stek se nalazi unutar OpenGL konteksta. Pozivom funkcije glGetError skida se poslednja greška sa ovog steka i vraća korisniku. Debagovanje se onda radi tako što se pre poziva OpenGL funkcije očisti ceo stek grešaka uzastopnim pozivom glGetError dok funkcija ne vrati GL_NO_ERROR. Zatim se pozove OpenGL funkcija i nakon nje pozove glGetError. Ukoliko glGetError vrati grešku znači da je baš u prethodnom pozivu OpenGL funkcije došlo do greške koju je vratio glGetError.

Da bi se ovaj proces ubrzao obično se definiše makro GLCALL koji kao argument uzima ceo poziv OpenGL funkcije. Taj makro pre samog poziva očisti OpenGL stek grešaka, zatim pozove prosleđenu OpenGL funkciju i ukoliko nakon poziva te funkcije glGetError vrati vrednost koja nije GL_NO_ERROR, ovaj makro zaustavlja izvršavanje programa i na izlazu za greške ispisuje poruku u kojoj liniji i fajlu je došlo do greške. Potpisi ovih funkcija i definicija makroa dati su u primeru 1.

```
1 void clearAllOpenGlErrors();
2 const char* openGLErrorToString(GLenum error);
3 bool wasPreviousOpenGLCallSuccessful(const char* file, int line,
      const char* call);
4 #define BREAK_IF_FALSE(x) if (!(x)) __builtin_trap()
5 #define ASSERT(x, msg) do { std::cerr << msg << '\n';
      BREAK_IF_FALSE(false); } while(0)
6 #define GLCALL(x) clearAllOpenGlErrors(); x; BREAK_IF_FALSE(
      wasPreviousOpenGLCallSuccessful(__FILE__, __LINE__, #x));
```

Primer 1: Definicija GLCALL makroa za debagovanje

Ovaj makro se onda koristi kao u primeru 2. Ukoliko je došlo do greške u pozivu funkcije glBindBuffer onda će se izvršavanje zaustaviti baš na toj liniji. Ovo pruža informaciju gde se greška dogodila, ali ne i zbog čega je do greške došlo.

```
1 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
      GL_STATIC_DRAW);
2 GLCALL(glBindBuffer(GL_BUFFER_BINDING, EBO)); // GRESKA. Trebalo
      bi GL_ELEMENT_ARRAY_BUFFER
3 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
      GL_STATIC_DRAW);
```

Primer 2: Primer korišćenja GLCALL makroa

Takođe, glGetError ima siromašan skup vrednosti grešaka koje može da vrati. Sve vrednosti koje glGetError može da vrati su:

- GL_NO_ERROR - Nema greške
- GL_INVALID_ENUM - Pogrešna vrednost argumenta tipa enum
- GL_INVALID_VALUE - Nevalidna vrednost argumenta
- GL_INVALID_OPERATION - Nevalidna operacija
- GL_STACK_OVERFLOW - Prekoračenje steka
- GL_STACK_UNDERFLOW - Potkoračenje steka
- GL_OUT_OF_MEMORY - Nedovoljno memorije
- GL_INVALID_FRAMEBUFFER_OPERATION - Nevalidna operacija nad frejmbaferom

Sve ove greške se odnose samo na trenutnu funkciju i ne govore ništa o širem kontekstu u kome se funkcija nalazi. Greške koje se najteže pronalaze su one koje su se desile zbog kombinacije poziva OpenGL funkcija, ne poziva samo jedne. Zbog toga je u verziji OpenGL Core 4.3 definisan bolji sistem za debagovanje.

3.2 Debug izlaz

Od verzije OpenGL Core 4.3 uveden je novi skup funkcija koje korisniku daju detaljnije informacije o greškama. Pored informacije o greškama u nekim slučajevima daju i predlog kako bi greška mogla da se ukloni. Ovo je moguće jer je verzija OpenGL Core 4.3 uvela debag kontekst. Radi tako što se OpenGL-u prosledi funkcija koju će OpenGL pozvati ukoliko dođe do određene greške. Potpis ove funkcije dat je u primeru 3.

```
void APIENTRY glDebugOutput(GLenum source, GLenum type, unsigned
    int id, GLenum severity,
GLsizei length, const char *message, const void *userParam);
```

Primer 3: Prototip glDebugOutput callback funkcije

Sa ovim parametrima može se kreirati kolbek (eng. *callback*) funkcija koja će dati detaljan opis greške do koje je došlo. Ova funkcija će biti pozvana kada do greške dođe. Time se uklanja ručno ubacivanje makroa.

Poruke koje se dobiju od OpenGL se onda mogu i filtrirati korišćenjem funkcije glDebugMessageControl kao u primeru 4. Nakon ovog poziva OpenGL će u debag izlaz slati samo poruke o greškama na nivou izvornog koda, koje su greške visokog prioriteta.

```
glDebugMessageControl(GL_DEBUG_SOURCE_API, GL_DEBUG_TYPE_ERROR,
    GL_DEBUG_SEVERITY_HIGH,0, nullptr, GL_TRUE);
```

Primer 4: Primer glDebugMessageControl

Još jedan veoma jednostavan način za debagovanje je da se u glDebugOutput funkciji koja se prosleđuje kao kolbek OpenGL API-ju, postavi tačka zaustavljanja (eng. *breakpoint*). Tako će, u slučaju greške, izvršavanje stati na toj funkciji, a u debageru na steku poziva će se nalazite sve funkcije koje su pozvane do tog trenutka. Iz tog steka poziva je onda moguće videti celokupno stanje u kome se nalazila aplikacija i OpenGL kontekst kada je do greške došlo što znatno olakšava njeno uklanjanje.

Za rad sa OpenGL API-jem korišćenje debag izlaza je preporučljivo za novije verzije. Stari način debagovanja korišćenjem glGetError poziva se ne preporučuje, osim za stare verzije OpenGL-a.

3.3 Predefinisane GLSL promenljive

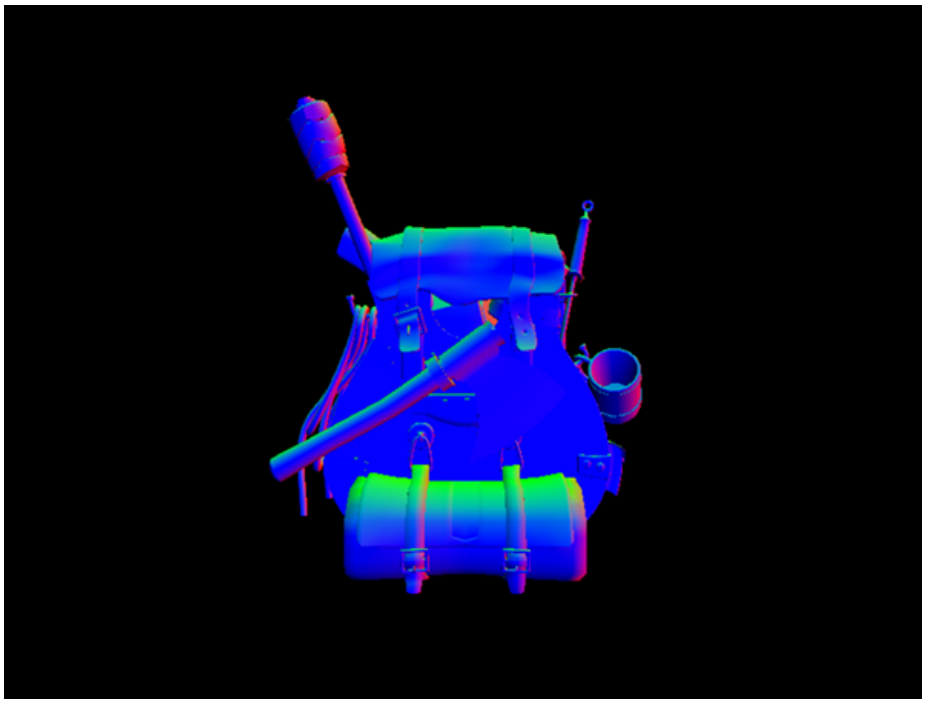
GLSL specifikacija definiše nekoliko promenljivih koje su rezervisane i koriste se u tačno određene svrhe. Transformacije koje GLSL radi nad ovim promenljivima i opseg u kojem su vidljive su takođe precizno definisani. Ukoliko spoljni alati kao što su RenderDOC [5] ili NVidia Nsight[8] nisu dostupni, onda se ove promenljive mogu koristiti kao neki vid printf debagovanja.

Jedna od tih promenljivih je izlazna promenljiva iz fragment šejdera koja određuje boju fragmenta. U primeru 6 to je promenljiva FragColor. Sve vrednosti iz vrteks šejdera se proslede fragment šejderu, a onda u zavisnosti od greške koja se traži, boja fragmenta se postavi na neku jasno uočljivu ekstremnu vrednost ili na vrednost koju je potrebno vizuelno prikazati.

```
#version 330 core
out vec4 FragColor;
in vec3 Normal;
[...]

void main()
{
[...]
FragColor.rgb = Normal;
FragColor.a = 1.0f;
}
```

Primer 5: Primer glDebugMessageControl



Slika 3: Bojenje fragmenata vrednostima normala u svrhu debagovanja[14]

Konkretno, u primeru 6 debaguju se vrednosti normala vrteksa. Pa se boja fragmenta postavi da bude normala. Rezultat je da se objekat oboji vrednostima normala kao na slici 3.

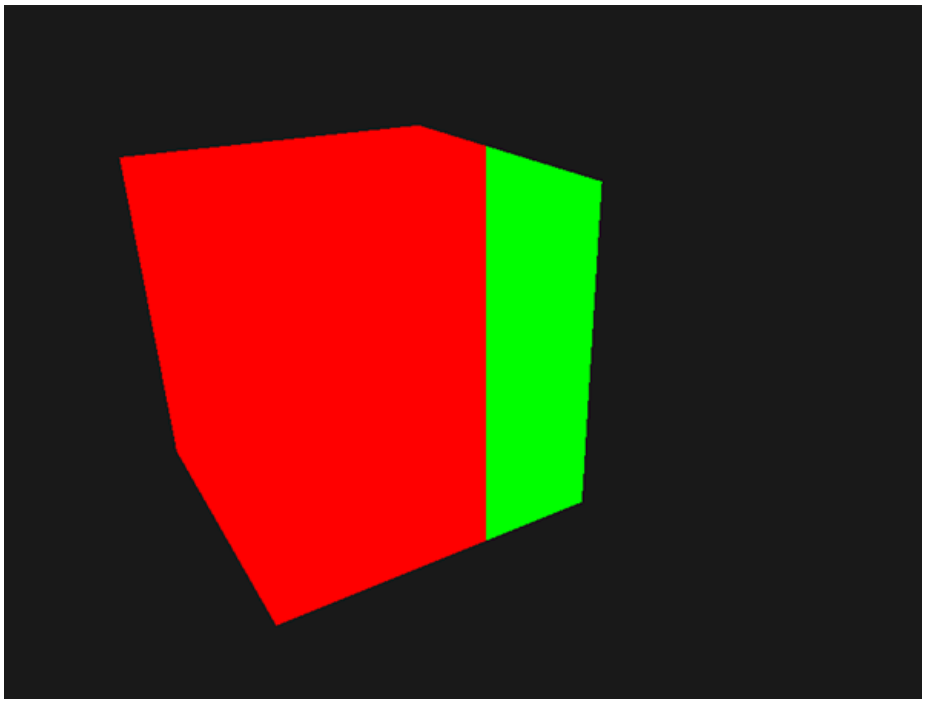
Druga ugrađena promenljiva koja može biti od koristi je gl_PointSize. To je promenljiva koja definiše veličinu tačke na ekranu. Onda kada je potrebno jasno vizuelno pratiti gde se nalazi svaki od temena trougla objekta koji se crta, povećanjem vrednosti gl_PointSize će i temena zauzimati više fragmenata na ekranu i samim tim biti lakše uočljiva.

Promenljiva gl_FragCoord čuva poziciju fragmenta na ekranu u svojim X i Y koordinatama, a u Z koordinati njegovu dubinu na sceni. Može se koristiti za uslovno crtanje samo dela ekrana, ili samo fragmenata koji su na određenoj dubini. U primeru 6 ukoliko se fragment nalazi na levoj polovini ekrana boji se crvenom, u suprotnom boji se zelenom.

```
void main()
{
if(gl_FragCoord.x < 400)
  FragColor = vec4(1.0, 0.0, 0.0, 1.0);
else
  FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Primer 6: Primer glDebugMessageControl

Ovo može biti od koristi ako je potrebno posmatrati efekte nad fragmentima na nekoj određenoj udaljenosti od kamere jer je obično teško uočiti detalje efekata i debagovati ih na malom broju fragmenata koji se nalaze daleko ukoliko je ekran pun ostalih fragmenata.



Slika 4: Bojenje fragmenata vrednostima normala u svrhu debagovanja[15]

4 Alati

U ovom poglavlju predstavljeni su najpopularniji alati za debagovanje i profaljiranje koji se, pored OpenGL-a, mogu koristiti i za druge API-je kao što su: DX11, DX12 i Vulkan.

Ovo su alati opšte namene koji u sebi imaju mnoštvo funkcionalnosti. Neki od navedenih alata zahtevaju izmenu izvornog koda aplikacije kako bi mogli da funkcionišu. Na primer, da bi RenderDoc mogao lakše da pronađe teksturu, potrebno je anotirati je sa nekim imenom. Takođe, imaju mogućnost da zabeleže kompletno stanje u kome se nalazi grafička

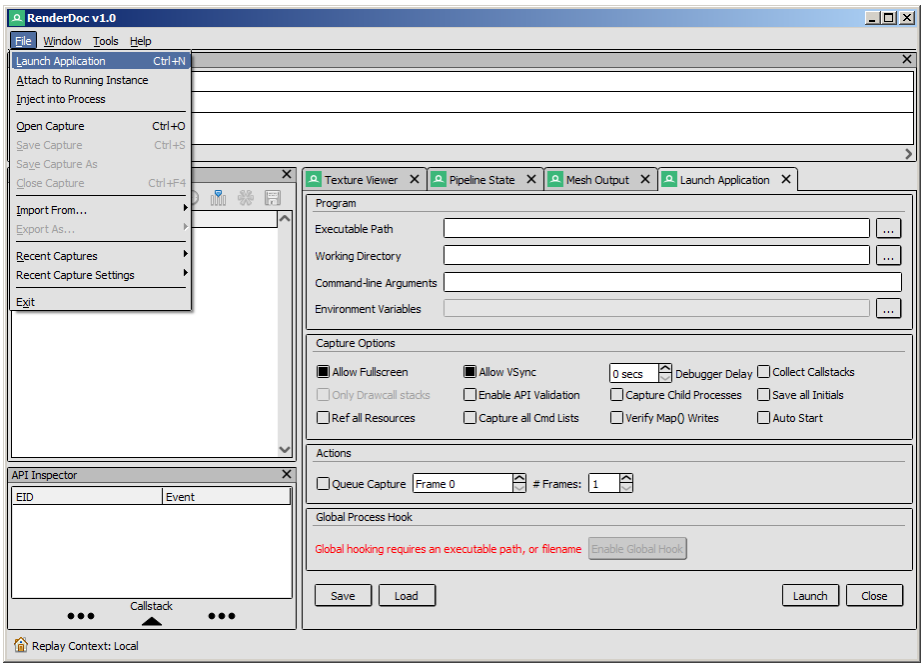
kartica u datom frejmu i tako ubrzaju proces pronalaženja greške u samim šejder programima koji se izvršavaju na grafičkoj kartici.

4.1 RenderDOC

RenderDOC je besplatan debager pod MIT licencom za debagovanje grafike koji omogućava lak i detaljan pregled celokupnog stanja grafičke kartice u nekom frejmu. Podržava operativne sisteme: Windows 7-10, Linux, Android, Stadia i Nintendo Svič[3].

Potpuno je otvorenog koda i celokupan razvoj se odvija preko githaba (eng. em GitHub). RenderDOC u sebi sadrži Python okruženje što omogućava programabilno debagovanje korišćenjem Python programskog jezika tako da je moguće napisati skriptu koja će izvršavati program dok se ne stigne do frejma koji sadrži u sebi stanje koje python skripta treba da detektuje.

RenderDOC je dobro dokumentovan i intuitivan za korišćenje. Radi tako što se pokrene zajedno sa aplikacijom i zatim se RenderDOC proces 'nakači' na proces aplikacije koja se debaguje kao na slici 5. Nakon toga RenderDOC će pokupiti sve potrebne informacije sa grafičke kartice i omogućiti debagovanje pokrenutog programa.



Slika 5: Početni prozor RenderDOC alata[5]

Najčešće korišćene funkcionalnosti RenderDOC alata su:

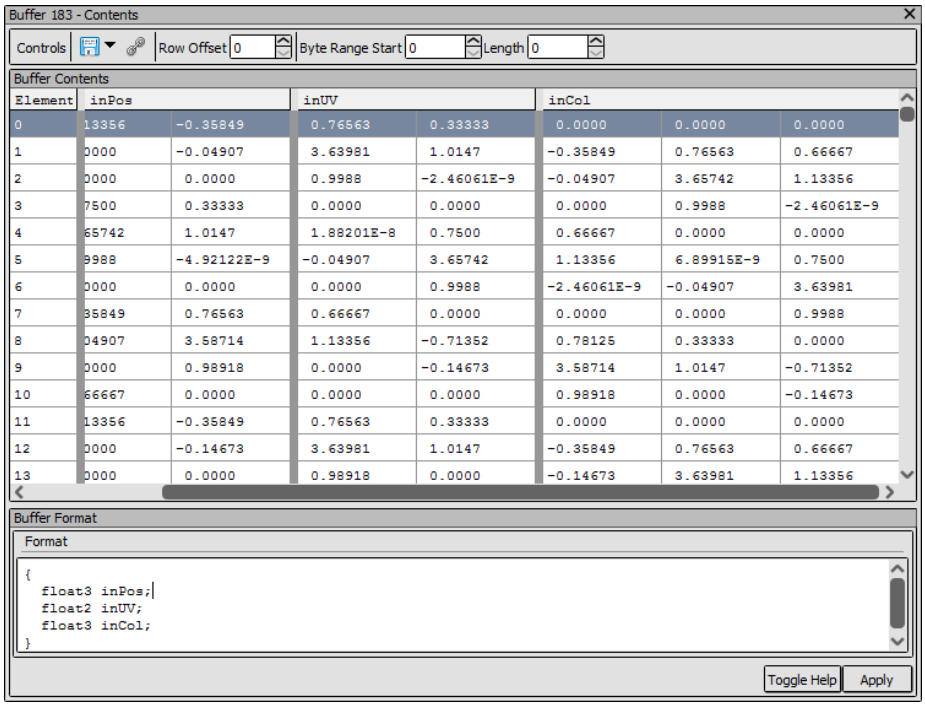
- Pregled celokupnog stanja grafičke kartice u željenom frejmu
- Korak po korak debagovanje šejdera
- Prikaz mapiranih tekstura
- Prikaz detalja na objektu

- Uvid u celokupan stek poziva funkcija
- Izmena koda i ponovno učitavanje šejder programa

Svaka od navedenih stavki se može uraditi preko korisničkog interfejsa RenderDOC alata ili Python API-ja koji RenderDOC implementira. Omogućava automatizaciju čestih postupaka koji se izvode tokom debagovanja nekog problema, što dugoročno može uštedeti dragoceno vreme u razvoju.

Izmena koda šejdera u realnom vremenu je takođe od velike važnosti. Šejderi su programi koji crtaju objekte. Kada se pravi grafička aplikacija akcenat je na izgledu scene koja se prikazuje. Bez alata koji omogućava da se kod šejdera menja u realnom vremenu tokom izvršavanja aplikacije bilo bi neophodno implementirati sistem koji bi to omogućavao jer ponovno pokretanje aplikacije nakon svake izmene šejder koda bi oduzelo mnogo vremena. Izmena šejder koda tokom izvršavanja aplikacije radi na sličnom principu kao izmena koda aplikacije tokom debagovanja. Kada se promeni izvorni kod šejdera, RenderDOC ponovo kompajlira taj šejder program i nastavlja da crta objekat sa novo-dobijenim šejderom.

Na slici 6 je primer pogleda na vrednosti u baferima u kojima se nalaze podaci na grafičkoj kartici. Kada se podaci šalju na grafičku karticu obično se šalju kao niz vrteksa. Svaki vrteks se sastoji od više atributa kao što su: pozicija vrteksa u svetu, normale, koordinate tekstura i tangentni vektori. Objekti mogu imati na hiljade takvih vrteksa. Svaki vrteks je teme jednog trougla na modelu objekta koji se crta.



Slika 6: Prikaz sadržaja bafera koji se nalazi u memoriji grafičke kartice[6]

RenderDOC alat omogućava prikaz ovih vrednosti u korisnički definisanom formatu. Konkretno na slici 6 korisnik u donjem delu prozora zada-

je format u kome želi prikaz vrednosti bafera, a RenderDOC te vrednosti pročita iz datog bafera i prikaže u formatu pogodnim za debagovanje.

Ovde su navedene samo neke od funkcionalnosti RenderDOC alata. Jednostavan je za instalaciju i korišćenje. Ima sopstveni korisnički interfejs i ceo RenderDOC projekat ne zavisi, niti zahteva instalaciju nekog drugog programa da bi se koristio. Mnogi iskusni programeri u oblasti računarske grafike ga toplo preporučuju. [4]

4.2 Nvidia NSight

NVidia Nsight je još jedan alat opšte namena za debagovanje i profajliranje programa koji se izvršavaju na grafičkoj kartici. Može se koristiti kao program sam za sebe ili kao dodatak za Visual Studio integrisano razvojno okruženje kompanije Microsoft. Postoji i verzija koja radi kao Eclipse dodatak, ali ona se slabiji koristi.[9]

Ima tri glavne namene:

- *Nsight Compute* - Interaktivni CUDA kernel profajler
- *Nsight Graphics* - Debager i profajler frejmova grafičkih aplikacija
- *Nsight Systems* - Alat za analizu performansi na nivou celokupnog sistema

Pored pregleda celokupnog stanja grafičke kartice, steka poziva funkcija, tekstura, boja individualnih piksela i još mnoštva funkcionalnosti za debagovanje Nsight Graphics ima i profajler frejmova čiji se primer profajliranja frejma može videti na slici 7.

Nsight profajler automatski detektuje spore delove programa i prilike za optimizaciju. Sa minimalnim dodatnim uticajem na izmerene vrednosti. Takođe može da meri propusni protok (eng. *throughput*) instrukcija i memorije i broj pogodaka keš memorije.

Još jedna korisna funkcionalnost ovog alata je istorija piksela. Na slici 8 prikazan je izgled interfejsa ove funkcionalnosti. Za svaki piksel se može posmatrati koju je vrednost imao u nekom rasponu frejmova i iz kojih izvora su dolazile boje za taj piksel. Na primer, ako je za crtanje providnih objekata koji se na sceni preklapaju iskorišćena tehnika stapanja boja, preko istorije piksela će se videti iz kojih tačno tekstura i kojih koordinata svake od tih tekstura dolazi boja svakog individualnog piksela.

Nvidia Nsight je alata sa bogatijim skupom funkcionalnosti od RenderDOC debagera jer pored debagera ima i moćan profajler. Jedina mana mu je što radi samo sa NVidia grafičkim karticama [10] i to ne svim.

Koriste ga mnoge poznate kompanije u svetu grafike kao što su: Epic Games, Ubisoft i Autodesk.

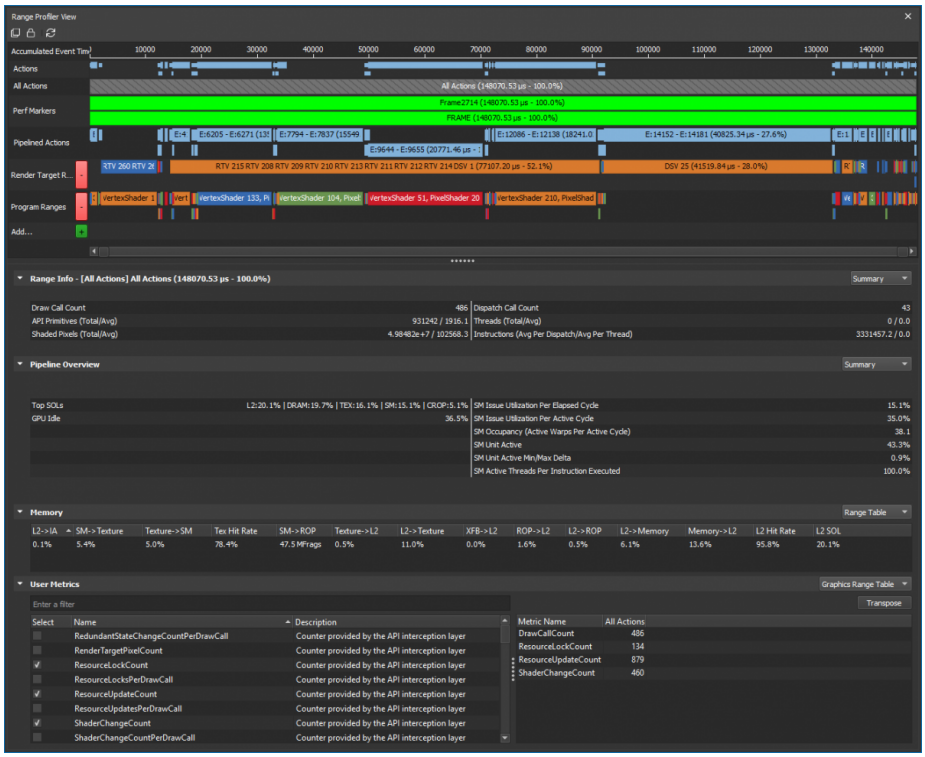
4.3 PerfWorks

PerfWorks je takođe alat kompanije Nvidia. Koristi se za analizu performansi NVIDIA grafičkih kartica. Za razliku od Nsight alata koji dolazi sa korisničkim interfejsom, PerfWorks je C++ API. [11]

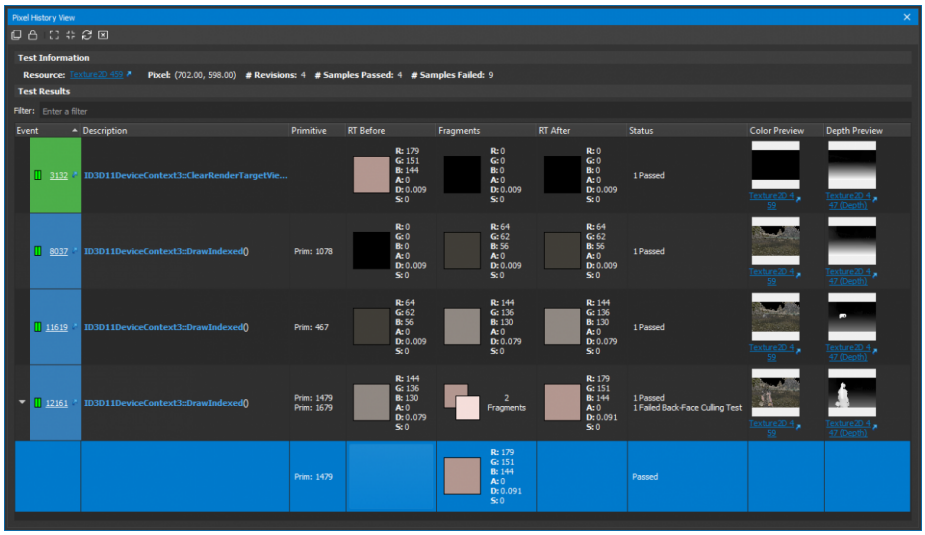
Omogućava programerima da prikupe metrike izvršavanja programa na grafičkoj kartici na veoma niskom nivou i tako prepoznaju gde se nalaze kritične tačke performansi.

PerfWorks prikuplja podatke u četiri kategorije:

- Kumulativni posao - sve što grafička kartica uradi tokom određenog broja obojenih piksela



Slika 7: NSight Graphics profajler[8]



Slika 8: NSight Graphics istorija boje piksela[8]

- Tajming (eng. *Timing*) - Prosečna brzina izračunavanja
- Aktivnost - Mesta na kojima je grafička kartica aktivna, a gde je na

čekanju

- Protok (eng. *Throughput*) - Broj izvršenih operacija

PerfWorks se koristi upravo u Visual Studio izdanju NSight alata kako bi prikupljao i prikazivao svu potrebnu metriku o stanju i radu grafičke kartice.

Ovaj alat je obično koristan ukoliko je potrebno napraviti zaseban program za profajliranje specifične aplikacije jer sve što PerfWorks radi je da stavi razne metrike koje prikuplja korisnicima na raspolaganje.

5 Zaključak

Grafičke kartice se u vreme pisanje ovog rada ne koriste više samo za grafičke aplikacije već i za izračunavanja opšte namene. Većina trenutne moći izračunavanja hardvera se nalazi na grafičkoj kartici i dobri alati za debagovanje i profajliranje ne samo šejder programa, već i programa opšte namene će biti sve potrebniji.

Kako bude rasla potrebna za više izračunavanja na grafičkim karticama tako će i alati za rad postajati sve bolji. Bogata lista već napisani aplikacija koje koriste OpenGL govori da je bez obzira na sve poteškoće u programiranju grafičkih kartica moguće napraviti stabilne aplikacije koje korisnicima pružaju sate uživanje bez većih bagova. [12].

Za manje aplikacije i projekte skup funkcija sa kojim dolazi OpenGL API mogu biti sasvim dovoljne za uklanjanje većine čestih grešaka. Dok za bilo koju aplikaciju koja se pravi u komercijalne svrhe, neophodno je koristiti neki od alata koji pomažu proces debagovanja i profajliranja.

Za Windows platformu NVidia Nsight dodatak za Visual Studio je odličan izbor. Za ostale platforme, pogotovu za Android, RenderDOC je zbog svoje dostupnosti i lakoće korišćenja u tom slučaju ipak bolji.

Literatura

[1] The Khronos Group. GLSL specification, 2019. on-line at: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[2] The Khronos Group. OpenGL documentation - glGetError, 2019. on-line at: https://www.khronos.org/registry/OpenGL-Refpages/es2.0/xhtml/glGetError.xml.

[3] Baldur Karlsson. RenderDOC , 2020. on-line at: https://renderdoc.org/.

[4] Baldur Karlsson. RenderDOC home page, 2020. on-line at: https://renderdoc.org/.

[5] Baldur Karlsson. RenderDOC quickstart, 2020. on-line at: https://renderdoc.org/docs/getting_started/quick_start.html.

[6] Baldur Karlsson. RenderDOC view object details, 2020. on-line at: https://renderdoc.org/docs/how/how_object_details.html.

[7] Khronos. OpenGL Core Profile 4.3, 2013. on-line at: https://www.khronos.org/registry/OpenGL/specs/gl/glspec43.core.pdf.

[8] NVidia. NSight Graphics, 2020. on-line at: https://developer.nvidia.com/nsight-graphics.

[9] Nvidia. Nvidia Nsight , 2020. on-line at: https://developer.nvidia.com/nsight-visual-studio-edition.

[10] NVidia. NVidia supported GPUs , 2020. on-line at: https://developer.nvidia.com/nsight-visual-studio-edition-supported-gpus-full-list.

[11] NVidia. PerfWorks home page, 2020. on-line at: https://developer.nvidia.com/perfworks.

[12] PCgamingwiki. List of opengl games , 2020. on-line at: https://www.pcgamingwiki.com/wiki/List_of_OpenGL_games.

[13] Joey De Vries. Abstract representation of OpenGL pipeline, 2018. on-line at: https://learnopengl.com/Getting-started/Hello-Triangle.

[14] Joey De Vries. LearnOpenGL debug, 2018. on-line at: https://learnopengl.com/In-Practice/Debugging.

[15] Joey De Vries. LearnOpenGL - Advanced Data , 2020. on-line at: https://learnopengl.com/Advanced-OpenGL/Advanced-Data.

[16] WCCFtech. GPU market share, 2017. on-line at: https://wccftech.com/nvidia-amd-intel-gpu-market-share-q3-2017/.