RAZVOJ IGRE S LWJGL BIBLIOTEKOM

Gaćina, Filip

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

RAZVOJ IGRE S LWJGL BIBLIOTEKOM

Filip Gaćina

Zagreb, veljača 2019.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada".

U Zagrebu, 26.02.2019.

Predgovor

Želio bih se zahvaliti svojim bližnjima na beskonačnoj količini strpljenja, te svom mentoru Aleksandru Radovanu za svu pomoć pri pisanju ovog rada.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Svaka grana razvoja programskih rješenja je obilježena najpopularnijim tehnologijama za razvoj specifičnih rješenja. Iako su najpopularnije opcije obično pravilan izbor, puno se može naučiti istraživanjem onih manje popularnih opcija.

Cilj ovog završnog rada je istražiti jednu od tih manje popularnih opcija, za razvoj računalnih igara. Ovaj rad će okvirno demonstrirati razvoj računalnih igara s Java programskim jezikom uz pomoć LWJGL (*LightWeight Java Game Library*) biblioteke.

Praktični dio ovog rada je potpuno funkcionalna 3D igra koja služi kao demo, te demonstrira jednostavnu implementaciju elemenata kao što su: 3D grafika, osvjetljenje, animiranje i sl.

Ključne riječi: Java, LWJGL, 3D

Summary

Each branch of software development is marked by the most popular development stack used to develop specific type of software. Using the most popular stack is usually the right decision, however, there is a great value to be found in exploring the lesser known options.

Goal of this final thesis is to explore one of those lesser known options, for game development. This thesis will demonstrate game development using Java programming language along with LWJGL (*LightWeight Java Game Library*).

Practical part of this thesis is contained within a fully functional 3D game. Game is used to demonstrate simple implementation of essential game elements such as: 3D graphics, lighting, animation, ... etc.

Keywords: Java, LWJGL, 3D

Sadržaj

1. Uvod

Inicijalno je ovaj završni rad trebao biti samo proširenje biblioteke za razvoj igara koja je napisana za vrijeme kolegija Java 2. No, vrijednost takvog rada bi bila minimalna jer su svi glavni koncepti već pokriveni u projektima napisanim za taj kolegij.

Spomenuta biblioteka je radila s JavaFX okvirom koji je dosta ograničen s perspektive razvoja igara, te kao takav nije prigodan za ozbiljniji projekt.

Prirodni napredak u smjeru korištenja Java programskog jezika za programiranje igara je LWJGL biblioteka, koja omogućava Java programerima da koriste izvorne (engl. *native*) biblioteke za grafiku (OpenGL, Vulkan), zvuk (OpenAL) i sl.

Radi demonstracije napravljena je jednostavna 3D igra koja će pokriti najbitnije koncepte za razvoj 3D igara. Svaki koncept je okvirno objašnjen te je arhitektura same igre objašnjena u detalje.

Rad počinje s pregledom trenutnog stanja tehnologija za razvoj igara. Zatim se čitatelja upoznaje s LWJGL bibliotekom, nakon čega slijedi analiza i opis arhitekture praktičnog dijela ovog rada.

2. Pregled standardnih tehnologija za izradu igara

Ovo poglavlje se kratko dotiče nekih od najpoznatijih tehnologija za izradu video igara na tržištu.

2.1. C++ programski jezik

C++ je programski jezik dizajniran za generalnu upotrebu u razvoju efikasnih programskih rješenja. C++ je također poznat kao "C s klasama" iz razloga što C++ uvodi koncept objektno-orijentiranog programiranja u C programski jezik.

Glavni razlog popularnosti C++ programskog jezika u razvoju igara je činjenica da programer ima gotovo neograničenu kontrolu nad memorijom. Radi razine slobode koju programer ima, moguće je ostvariti optimizacije koje su drugim programskim jezicima izvan dometa.[1]

Podatkovno-orijentirani dizajn (engl. *data-oriented design*) je primjer cijelog stila programiranja koji se bazira na efikasnoj manipulaciji procesorske pred-memorije (engl. CPU *cache*) koja je drastično brža od RAM (*Random Access Memory*) memorije. Ovakav stil je gotovo nemoguće primijeniti u jezicima kao Java gdje nad memorijom vlada *garbage collector*.

C++ se koristi s postojećim okruženjima kao Unreal Engine i Unity, te kao samostalan jezik za razvoj *game enginea*. Glavni nedostatak C++ programskog jezika s perspektive razvoja igara je kompleksnost koja s godinama postaje sve veća i veća. Novi alati se fokusiraju na objektno-orijentirano programiranje te kao takvi ne služe velikoj svrsi programerima koji koriste podatkovno-orijentirani pristup.

Iz tog razloga je nastao programski jezik Jai koji je trenutno u razvoju od strane Jonathan Blowa. Jai se fokusira na podatkovno-orijentirani pristup te slobodu kontrole memorije koju ima C programski jezik. Kao takav Jai bi mogao biti sljedeći popularni jezik za razvoj video igara.[2]

[1] https://en.wikipedia.org/wiki/C%2B%2B
[2] https://github.com/BSVino/JaiPrimer/blob/master/JaiPrimer.md

2.2. Unity i Unreal Engine

Unity i Unreal *engine* su primjer gotovih okruženja za razvoj igara. Kao takvi omogućavaju programerima da se fokusiraju na samu igru, te samim time sačuvaju veliku količinu vremena koju bi u protivnom potrošili na razvoj sustava za iscrtavanje grafike, reprodukciju zvuka, *import* modela i sl. To u isto vrijeme znači da se gubi veliki dio kontrole nad tehničkim aspektom igre, te je potrebno naučiti specifičnosti samog okruženja da bi se efektivno mogla napraviti igra.

Oba *game enginea* pružaju API (*Application Programming Interface*) kojeg programeri koriste za razvijanje sustava za njihovu igru. Unity primarno koristi C# za skriptiranje, dok Unreal koristi C++. Također oba okruženja podržavaju vizualno skriptiranje, što je pogodno za brzu izradu prototipa, te ljude koji nisu iskusni programeri.

Oba sustava podržavaju razvoj igara za mnoštvo različitih platformi, kao što su: Windows, Linux, Android, PlayStation 4, Nintendo Switch i Oculus Rift. Iako je bitno napomenuti da Unity ima podršku za više platformi, kao što su razne SmartTV i AR (*Augmented Reality*) platforme.[3]

Do nedavno je razlika između ova dva okruženja bila drastična, te je opće mišljenje bilo da je Unreal bolji *engine* s perspektive grafike i performansi koje je moguće ostvariti, dok je Unity bio besplatna i lakša alternativa.[4] No, Unity je prošao kroz razna poboljšanja, a najnovija poboljšanja su u smjeru boljih performansi. Pod vodstvom poznatog *game engine* programera Mike Actona, Unity tim je razvio *burst compiler* koji pretvara IL/.NET *bytecode* u visoko optimizirani kod, koristeći LLVM (*Low Level Virtual Machine*).[5]

Koristeći oba razvojna okruženja lako se mogu ostvariti optimalne performanse za gotovo svaku igru koju razvojni tim može zamisliti.

Unity ima besplatnu i pro verziju, dok je Unreal besplatan te se naplata izvršava tek kada igra izađe na tržište. Bitno je napomenuti da oba okruženja također imaju svoj sustav tržišta gdje je moguće kupiti razne 3D modele, zvučne efekte, sustave za izgradnju *levela* i slično.

[3] https://en.wikipedia.org/wiki/Unity_(game_engine)
[4] https://en.wikipedia.org/wiki/Unreal_Engine
[5] http://infalliblecode.com/unity-burst-compiler/

3. Java programski jezik

Java je objektno orijentirani programski jezik za generalnu upotrebu. Java kod (*.java*) se pretvara u Java *bytecode* (*.class*), a sama aplikacija se pokreće na Java virtualnoj mašini. Java aplikacija se tako može uspješno pokretati na različitim platformama bez potrebe za dodatnom prilagodbom koda.

Java *bytecode* nije napisan u strojnom jeziku pa iz tog razloga uz *interpreter* postoji JIT (*Just-In-Time*) *compiler*. Zadatak JIT *compilera* je pretvoriti Java *bytecode* u strojni jezik, koji se izvršava drastično brže nego interpretirani kod.[6]

3.1. Java za razvoj igara

Glavni razlog zašto ljudi izbjegavaju Javu kad je u pitanju razvoj igara su navodne loše performanse. Dio toga se može okriviti na *garbage collector* koji preuzima kontrolu nad memorijom, provjerene iznimke (engl. *checked exceptions*) koje je nemoguće ignorirati te činjenicu da se aplikacija pokreće na virtualnoj mašini.

No, performanse ne ovise samo o tehnologiji. Problem je dijelom i u objektno orijentiranom dizajnu, ili točnije krivoj upotrebi objektno orijentiranog dizajna. Primjerice, pretjeravanje s apstrakcijom i fleksibilnosti koda, što rezultira u povećanoj količini metoda koje je potrebno izvršiti za obavljanje određenog zadatka.

Java generalno nema problema s performansama kod razvoja aplikacija koje ovise o unutarnjem *event loopu.* Takve aplikacije čekaju korisnikov unos, te odgovaraju na isti. Dok igre ne čekaju korisnika, te je potrebno konstantno i konzistentno simulirati svijet igre bez obzira na to da li je igrač aktivan ili ne. Količina vremena dostupna za potrebne operacije je 16 milisekundi, u slučaju igre u kojoj se ekran osvježava 60 puta u sekundi.

Za bolje performanse, potrebno je smanjiti broj potrebnih alokacija memorije, izbjeći korištenje iznimki te minimizirati količinu metoda koju je potrebno izvršiti u dijelu koda kritičnom za performanse igre. No, uz sve optimizacije ciljane performanse su nedostižne radi činjenice da CPU odrađuje sav posao. Stoga je potrebno zadatak iscrtavanja prebaciti na GPU (*Graphics Processing Unit*).

---

[6] https://en.wikipedia.org/wiki/Java_(programming_language)

4. LWJGL biblioteka

Nekoliko mjeseci nakon što je izašla Java verzija 1.4, stvorena je prva verzija LWJGL (*LightWeight Java Game Library*) biblioteke. Java verzija 1.4 je omogućila pristup memoriji izvan JVM *heapa,* te rad s izvornim kodom. To znači da je moguće alocirati veliku količinu memorije koju ne kontrolira *garbage collector*, te je moguće komunicirati s izvornim bibliotekama za grafiku, zvuk i sl.

LWJGL biblioteka je *wrapper* oko poznatih izvornih biblioteka koje se koriste za razvoj igara. Bitno je napomenuti da LWJGL biblioteka ne pruža korisniku gotove implementacije sustava igre, već samo pristup standardnim tehnologijama koje se koriste za razvoj sustava igre. LWJGL biblioteka pristupa izvornim bibliotekama koristeći JNI (*Java Native Interface*).[7]

4.1. JNI sučelje

Java Native Interface (JNI) je sučelje koje dozvoljava da kod koji se pokreće unutar Java virtualne mašine komunicira s aplikacijama i bibliotekama koje su napisane u drugim programskim jezicima, kao što su C i C++.

JNI sučelje je namijenjeno da se koristi u trenutcima kada određeni zahtjev nije moguće zadovoljiti korištenjem samo Java programskog jezika. Na primjer: kod potrebe za korištenjem specifičnog *hardwarea* ili za poboljšanje performansi zahtjevnih procesa.

Most između tehnologija se ostvaruje pomoću ključne riječi `native` koja označava da je metodu potrebno implementirati dijelu aplikacije koji je napisan na izvornom kodu. Te pomoću statične metode `System.loadLibrary` pomoću koje se biblioteka može učitati s datotečnog sistema u memoriju.

JNI ne specificira ograničenja implementacije Java virtualne mašine. Što znači da će Java aplikacija koja koristi izvorni kod ili aplikacije, raditi na svim Java virtualnim mašinama koje podržavaju JNI.[8]

[7] https://en.wikipedia.org/wiki/Lightweight_Java_Game_Library
[8] https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html#wp725

4.2. LWJGL 3.2.1 biblioteka

Ovaj projekt koristi LWJGL 3.2.1. Glavna razlika između ove verzije i prijašnjih verzija je modularnost. Biblioteka je podijeljena na module, gdje je svaki modul *wrapper* oko jedne izvorne biblioteke. Prednost modularnosti je u tome što prilikom razvoja igre, nije potrebno uvesti *wrappere* za izvorne biblioteke koje se ne koriste.

LWJGL 3 biblioteka je dizajnirana na način da se *wrapperi* koriste na isti način kao i izvorne biblioteke, osim u slučajevima kada razlika u jeziku to ne dozvoljava. Primjerice, umjesto *pointera* se koriste varijable tipa *long*, u koje se zatim sprema memorijska lokacija određenog objekta.

Ovaj projekt koristi module za sljedeće izvorne biblioteke: OpenGL, GLFW, OpenAL i Assimp.

4.2.1. OpenGL biblioteka

Open Graphics Library (OpenGL) je specifikacija koja definira API za iscrtavanje 2D i 3D vektorske grafike, a koristi se za komunikaciju s GPU-om, s ciljem ostvarivanja efikasnijeg iscrtavanja.

Radi toga postoji mnoštvo različitih implementacija OpenGL-a, koje su obično razvijene od strane proizvođača grafičkih kartica. Tako se eventualne greške određene implementacije mogu razriješiti instaliranjem najnovijih *drivera* za grafičku karticu.

Svaki GPU podržava specifične verzije OpenGL-a. Igra koja je praktični dio ovog projekta koristi OpenGL 3.3, te je igru moguće bez problema pokrenuti na sustavu koju za GPU ima, primjerice, Intel HD Graphics 4000. Jer ta verzija integriranog grafičkog procesora podržava OpenGL 4.0. Ovo je uspješno testirano na računalu koje se nalazi u knjižnici Visokog Učilišta Algebra.

Starije verzije OpenGL-a koristile su *fixed function pipeline.* Većina funkcionalnosti je bila sakrivena u samoj biblioteci, te programeri nisu imali dovoljno kontrole nad iscrtavanjem. Počevši od verzije 3.2 fiksni set funkcija za iscrtavanje u potpunosti je zamijenjen *shader* programima koji su dali programerima veliku količinu slobode i kontrole nad iscrtavanjem.[9]

[9] https://en.wikipedia.org/wiki/OpenGL

4.2.2. GLFW biblioteka

Graphics Library FrameWork (GLFW) je pomoćna biblioteka koja se koristi uz OpenGL, OpenGL ES ili Vulkan biblioteke, te pruža jednostavan način za kreiranje OpenGL konteksta, te komunikaciju s korisničkom periferijom za unos kao što tipkovnica i miš.

U igri napisanoj za ovaj projekt, igrač može koristiti tipkovnicu ili *joystick*. A *joystick* kontrole igre su prilagođene X360 *joysticku*.[10]

4.2.3. OpenAL biblioteka

Open Audio Library (Open AL) je API za reprodukciju zvuka. Pomoću ove biblioteke moguće je zvuk pozicionirati u 3D prostoru tako da igrač dobije dojam da zvuk dolazi iz određenog smjera. Te je moguće modulirati zvuk da bi se postigli željeni efekti koji odgovara atmosferi same igre, okruženju u kojem se igrač nalazi i sl.[11]

4.2.4. Assimp biblioteka

Open Asset Import Library (Assimp) je biblioteka za uvoz raznih formata 3D modela, a novije verzije Assimp biblioteke podržavaju i izvoz podataka.[12]

U ovom projektu Assimp se koristi za uvoz skeletnih animacija (engl. *skeletal animation*).

[10] https://en.wikipedia.org/wiki/GLFW
[11] https://en.wikipedia.org/wiki/OpenAL
[12] https://en.wikipedia.org/wiki/Open_Asset_Import_Library

5. Praktični dio rada: 3D igra

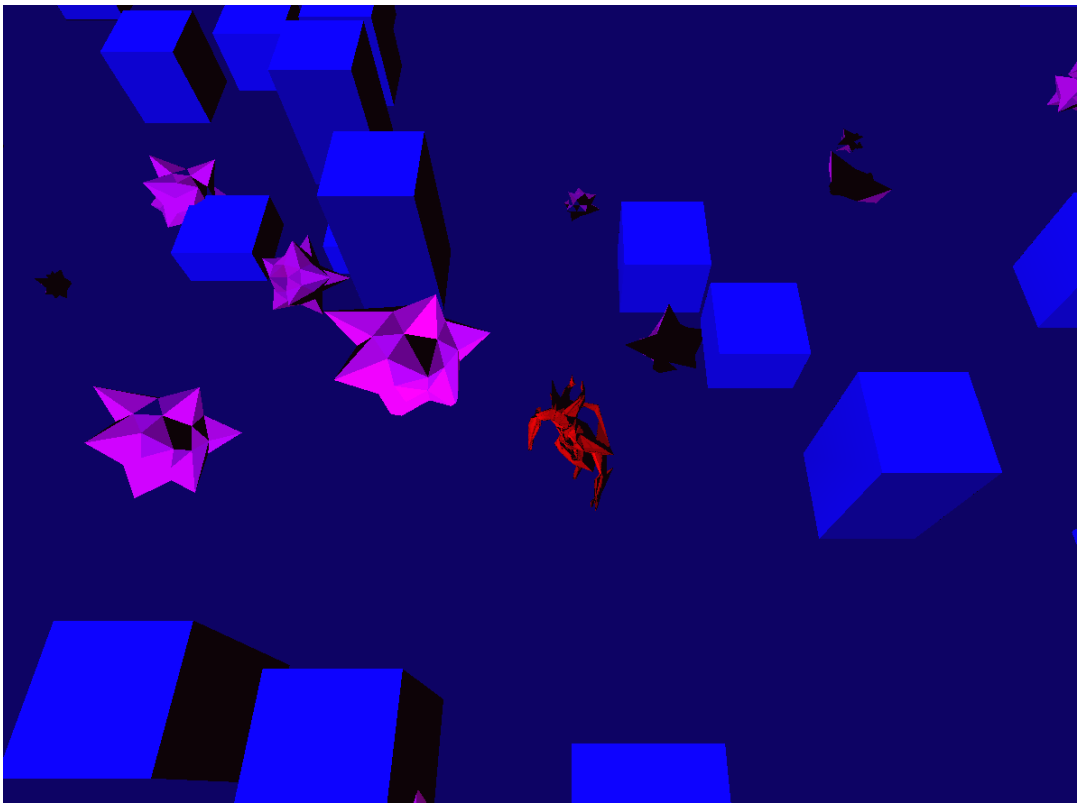
Ovo poglavlje se dotiče arhitekture i implementacije praktičnog dijela ovog završnog rada. Glavni cilj praktičnog dijela je demonstracija tehničkih aspekata u radu s LWJGL bibliotekom. Za potrebe demonstracije kreirana je 3D igra.

Decipherer je igra u kojoj igrač preuzima kontrolu nad zatvorenikom, te se bori za svoju slobodu. Igra se sastoji od proceduralno generiranih nivoa koji su međusobno odvojeni. Igrač će napredovati do sljedećeg nivoa tek kada su eliminirani svi neprijatelji na trenutnoj razini.

Igrač se može micati u svim smjerovima te u isto vrijeme izvršavati napade u smjeru prema kojem je trenutno okrenut. Svaka smrt igrača šalje na prvu razinu, te se cijeli niz nivoa ponovo generira.

Igra se može igrati pomoću miša i tipkovnice ili *joysticka*, te se vrlo lako može zamijeniti između tih opcija.

Slika 5.1 predstavlja *screenshot* iz igre Decipherer.



Slika 5.1. *Decipherer*

5.1. GLFW Prozor

Prozor služi kao kontekst za iscrtavanje 3D grafike, što ga čini osnovom ovog projekta. Stvaranje prozora je ostvareno pomoću GLFW biblioteke. Klasa `Window` sadržava stvaranje prozora i inicijalizaciju OpenGL konteksta u metodi `initialize` iz programskog isječka 5.1.

```
public long initialize()
{
    glfwSetErrorCallback(GLFWErrorCallback.createPrint(System.err));
    glfwInit();

    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
    glfwWindowHint(GLFW_DOUBLEBUFFER, GLFW_TRUE);

    long hMonitor = glfwGetPrimaryMonitor();
    GLFWVidMode primaryVidMode = glfwGetVideoMode(hMonitor);

    if (fullscreen)
    {
        width = primaryVidMode.width();
        height = primaryVidMode.height();
        hWindow = glfwCreateWindow(width, height, title, hMonitor, NULL);
    }
    else
    {
        hWindow = glfwCreateWindow(width, height, title, NULL, NULL);
        glfwSetWindowPos
        (
            hWindow,
            (primaryVidMode.width() - width) >> 1,
            (primaryVidMode.height() - height) >> 1
        );
    }

    glfwMakeContextCurrent(hWindow);
    glfwSwapInterval(0);

    GL.createCapabilities();
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    glFrontFace(GL_CW);
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_FRAMEBUFFER_SRGB);

    Renderer.initialize(width, height, fov, near, far);

    glfwShowWindow(hWindow);

    return hWindow;
}
```

Kod 5.1. Kreiranje prozora i OpenGL konteksta

Prije nego je moguće koristiti GLFW biblioteku, potrebno ju je inicijalizirati, a to se vrši pozivom `glfwInit` metode. Nakon toga je potrebno postaviti parametre za stvaranje prozora pozivom `glfwWindowHint` metode.

Prozor u ovom projektu je fiksne veličine te za iscrtavanje koristi *double buffering*. Što znači da se korisniku prikazuje slika iz jednog međuspremnika, dok grafički procesor procesira sljedeću sliku u drugom međuspremniku. Kada je sljedeća slika spremna za iscrtavanje, međuspremnici zamjenjuju uloge.

Prozor se kreira pozivom metode `glfwCreateWindow` koja vraća *handler* tipa `long`, kojim se može referencirati novo kreirani prozor. Nakon što je stvoren prozor, potrebno je postaviti OpenGL kontekst pozivom `glfwMakeContextCurrent` metode.

Da bi se omogućila kontrola nad brzinom iscrtavanja, potrebno je isključiti VSync. To se može ostvariti pozivom `glfwSwapInterval(0)`. Ovo je bitno iz razloga što će se u protivnom igra iscrtavati brzinom koja je definirana *hardwareom*, obično 60 FPS-a (*Frames Per Second*).

Nakon toga slijedi postavljanje OpenGL postavki koje će biti objašnjene u poglavlju koje se bavi iscrtavanjem. Na kraju, da bi korisnik mogao vidjeti prozor potrebno je pozvati metodu `glfwShowWindow` s odgovarajućim *handlerom*.

Bitno je još napomenuti da klasa `Window` podatke kao što su širina i visina prozora prima u konstruktoru, a sama inicijalizacija je odvojena u metodu. Iz razloga što će vlasnik OpenGL konteksta biti nit koja izvrši kod za inicijalizaciju. Time je omogućeno da se Window konfigurira u glavnoj niti, a inicijalizira u niti igre.

Pri zatvaranju programa potrebno je osloboditi svu memoriju rezerviranu za GLFW biblioteku i prozor, to se vrši u metodi `terminate` iz programskog isječka 5.2.

```
public void terminate()
{
    glfwFreeCallbacks(hWindow);
    glfwDestroyWindow(hWindow);

    glfwTerminate();
}
```

Kod 5.2. Oslobađanje memorije prozora

5.2. Glavna petlja

Prozor kreiran pomoću `Window` klase biti će vidljiv svega nekoliko milisekundi. Da bi prozor ostao aktivan, potrebno je redovito procesirati sve evente na tom prozoru. Osim toga, za ostvarenje funkcije igre, potrebno je procesirati korisnički unos, izračunavati trenutno stanje igre te iscrtavati grafički prikaz stanja igre na ekran. Radi toga je potrebna glavna petlja, definirana u programskom isječku 5.3, koja će konstantno izvršavati sve potrebne operacije.

```
@Override
public void run()
{
    game.initialize();

    previousFrameStart = nanoTime();
    currentFrameStart = 0;
    unprocessedTime = 0;
    totalElapsed = 0;

    do
    {
        currentFrameStart = nanoTime();
        elapsedBetweenFrames = currentFrameStart - previousFrameStart;
        previousFrameStart = currentFrameStart;
        unprocessedTime += elapsedBetweenFrames;
        totalElapsed += elapsedBetweenFrames;

        game._elapsedTime = totalElapsed;
        game._activeStage.input();

        while (unprocessedTime >= NS_UPDATE)
        {
            game._activeStage.update(NS_UPDATE);
            unprocessedTime -= NS_UPDATE;
        }

        game._activeStage.render(elapsedBetweenFrames);
        game.swap();

        while (nanoTime() - currentFrameStart < NS_FRAME)
        {
            try { Thread.sleep(1); }
            catch (InterruptedException ex)
            { System.err.println(ex.getMessage()); }
        }
    }
    while(game._shouldRun);

    game.terminate();
}
```

Kod 5.3. Implementacija glavne petlje

Stare igre su bile napisane s točnom informacijom na kakvom procesoru će se igra pokretati. Tako su programeri mogli točno odrediti koliko će vremena biti potrebno za procesiranje igre u svakom trenutku. U današnjem dobu, radi količine različitih platformi i *hardwarea,* praktički je nemoguće znati detalje *hardwarea* na kojem će se igra pokretati.

Rješenje za ovaj problem je glavna petlja koja ne ovisi o *hardwareu* na kojem se pokreće.

Klasa `Sequencer` sadržava implementaciju glavne petlje, koja koristi trenutno vrijeme `System.nanoTime()` za kalkulacije. Ova implementacija također osigurava da se igra ažurira u fiksnim intervalima, što je bitno radi predvidljivosti izvršavanja igre. Ukoliko postoji višak vremena na kraju ciklusa, nit će „prespavati" to vrijeme. Ovime se smanjuje opterećenje nad procesorom.

No, bitno je imati na umu da ovisno o platformi, postoje razne nepreciznosti. Primjerice, poziv metode `Thread.sleep(N)` nikada ne znači da će nit „odspavati" `N` milisekundi, već to znači da će „odspavati" minimalno `N` milisekundi. Vrijeme odstupanja ovisi o načinu na koji platforma upravlja nitima.

Još je bitno napomenuti da se u konstruktoru također definira učestalost iscrtavanja i ažuriranja stanja igre. Pa je tako, primjerice, moguće namjestiti da se igra ažurira 32 puta u sekundi, a iscrtava 64 puta u sekundi.

Da bi se omogućila veza između koda igre i glavne petlje, klasa `Sequencer` u konstruktoru prima instancu klase `Game`, ta instanca definira igru koja će se izvršavati u glavnoj petlji.

Nakon što su svi potrebni parametri konfigurirani, sve što je potrebno napraviti je izraditi novu nit, te joj kao `Runnable` proslijediti instancu klase `Sequencer`. Ovaj postupak se radi u glavnoj metodi ovog projekta. Ovime se stvara nit igre koja će ujedno i inicijalizirati Window, te samim time postati vlasnik OpenGL konteksta.

5.3. Struktura igre

Kao što je već spomenuto u prijašnjem poglavlju, klasa `Sequencer` u svojoj `run` metodi izvršava operacije nad instancom klase `Game`. Ovim se omogućava da promjena implementacije igre ne zahtijeva promjenu glavne petlje.

```
public abstract class Game
{
    protected boolean _shouldRun;
    protected long _elapsedTime;
    protected Stage _activeStage;

    protected abstract void initialize();
    protected abstract void swap();
    protected abstract void terminate();
}
```

Kod 5.4. Klasa Game

Klasa `Game` iz programskog isječka 5.4, definira osnovno sučelje potrebno za pokretanje različitih igara u sklopu glavne petlje.

Metoda `initialize` se izvršava jednom pri pokretanju niti igre, te kao takva služi za pokretanje podsustava igre.

Metoda `swap` se izvršava jednom na kraju svakog ciklusa glavne petlje. Namjena ove metode je izvršavanje finalnih kalkulacija, primjerice, postavljanje vrijednosti `_shouldRun` varijable, čije se stanje provjerava neposredno poslije izvršavanja ove metode.

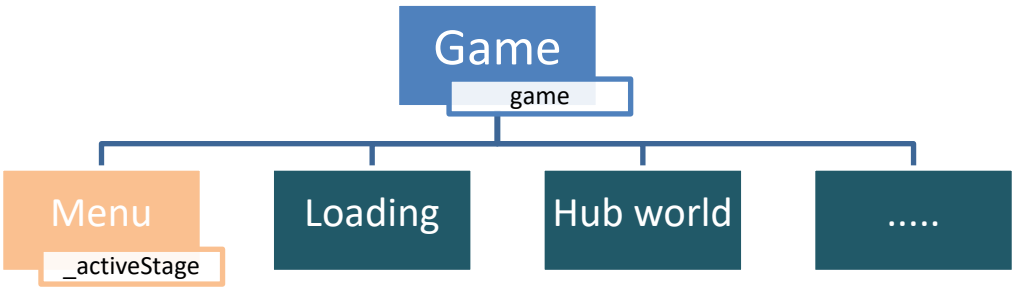
Metoda `terminate` se izvršava jednom pri završetku izvršavanja igre, te kao takva služi primarno za oslobađanje rezervirane memorije.

Prilikom izrade ovog projekta, prve verzije klase Game su se sastojale od 6 metoda. U trenutnoj verziji projekta 3 metode su postale podatkovni članovi, odnosno varijable. Glavni razlog za to je činjenica da su konkretne implementacija klase Game te metode obično implementirale kao *getter*, odnosno *setter* metode.

Varijabla `_shouldRun` daje glavnoj petlji do znanja da li je potrebno nastaviti izvršavanje igre, a stanje iste se provjerava na kraju svakog ciklusa glavne petlje. Trenutno ukupno vrijeme izvršavanja izraženo u nanosekundama pohranjuje se u varijablu `_elapsedTime`.

Posljednja varijabla je `_activeStage`, ova varijabla predstavlja trenutni dio igre koji se izvršava. Igre su često podijeljene u više dijelova, primjerice, u većini igara se može razlikovati ekran za učitavanje (engl. *loading screen*), glavni meni (engl. *main menu*) i sl.

Ti odvojeni dijelovi često imaju resurse koji se ne koriste nigdje drugdje, te logiku koja je specifična samo za taj dio igre. Primjerice, ekran za učitavanje često ne procesira korisnički unos, te ima sliku koja ispunjava ekran dok igrač čeka.



Slika 5.2. Osnovna struktura igre

Klasa `Stage` pruža sučelje za implementaciju specifičnog dijela igre, koje ima svoju logiku i resurse. Resursi instance `Stage` klase se mogu učitati odjednom ili korak po korak. Razlog za postojanje druge opcije je činjenica da ako se ne koristi odvojena nit za učitavanje resursa, prva opcija će uzrokovati zastoj. Druga opcija omogućava da se u svakom ciklusu glavne petlje učita samo maleni dio resursa, što znači da je količina posla u jednom ciklusu puno manja, što znači da neće doći do zastoja.

U svakom trenutku može biti aktivna samo jedna instanca klase `Stage`. Instanca `Stage` klase se smatra aktivnom ukoliko su svi resursi za tu instancu učitani i referenca za istu se nalazi u varijabli `_activeStage`.

Ovim klasama kreirana je osnovna struktura igre na slici 5.2. Iako ovakav sustav nije bio potreban za ostvarenje igre *Decipherer*, radi jako malog broja resursa i logike. Sustav olakšava dodavanje novih stvari u igru, te služi kao dobra podloga za buduće zahtjevnije projekte. U igri Decipherer postoje `LoadingStage` i `LevelStage`. Glavni zadatak `LoadingStage` instance je učitavanje drugih instanci klase `Stage`. Iz ovog razloga resursi `LoadingStage` instance su uvijek učitani. Svaki zahtjev za promjenom aktivnog dijela igre prolazi kroz `LoadingStage` koji priprema zahtijevanu instancu u metodi `update` iz programskog isječka 5.5.

```
@Override
public void update(long delta)
{
    if (requested != null)
    {
        if (requested.loadPart())
        {
            _loadComplete = true;
           requested.loadComplete();
        }
    }
}
```

Kod 5.5. Učitavanje novog *stagea*

Slika 5.3. Izgled ekrana prilikom učitavanja

Tijekom učitavanja, `LoadingStage` iscrtava rotirajući igračev model, kao što se vidi na slici 5.3. Bitno je još napomenuti da `LoadingStage` ne oslobađa resurse i ne postavlja trenutno aktivnu instancu u `_activeStage`. Taj proces se odvija u `swap` metodi konkretne implementacije klase `Game`:

```
@Override
protected void swap()
{
    if (!loadingStage._active)
    {
        swap = _activeStage.pollSwaps();

        if (swap != -1)
        {
            _activeStage.unload();
            loadingStage.setStage(stages[swap]);

            _activeStage = loadingStage;
            loadingStage._active = true;
        }
    }
    else if (loadingStage._loadComplete)
    {
        _activeStage = stages[swap];
        loadingStage._active = false;
    }

    _shouldRun = !glfwWindowShouldClose(hWindow);
}
```

Kod 5.6. Zamjena aktivnog *stagea*

Zahtjev za promjenom *stagea* se dobiva kao rezultat metode `pollSwaps` klase `Stage`. Rezultat je tipa `int` te predstavlja ID željenog *stagea*. Ukoliko je ta vrijednost različita od -1, resursi za trenutni *stage* se oslobađaju, te se zahtjev za učitavanje novog *stagea* šalje `LoadingStage` instanci, koja se tada aktivira, te ostaje aktivna sve dok učitavanje željenog *stagea* nije gotovo. Ova metoda također ažurira vrijednost `_shouldRun` varijable putem `glfwWindowShouldClose` metode, koja će provjeriti stanje *close flaga* za dani prozor.

5.4. Korisnički unos

Korisnički unos je *esencijalni* dio svake igre jer interaktivnost igre ovisi o korisničkom unosu. Ovaj aspekt je također implementiran pomoću GLFW biblioteke.

Za potrebe demonstracije igra *Decipherer* može se igrati pomoću miša i tipkovnice ili *joysticka*. Prvi korak u ostvarenju korisničkog unosa je skupljanje stanja promatrane periferije za unos. Nakon toga je potrebno iste kombinirati u korisne informacije koje se mogu koristiti za upravljanje. Pri svemu tome je potrebno imati na umu da različiti dijelovi igre mogu koristiti različite kontrole, te da bi trebalo omogućiti promjenu izvora korisničkog unosa u danom trenutku.

5.4.1. Korisnički unos pomoću tipkovnice

Za potrebe ovog projekta potrebno je dohvatiti stanje tipki na tipkovnici. Da bi se dohvatilo stanje za određenu tipku, potreban je ID te tipke te *handler* za prozor. Ti parametri su potrebni za poziv metode `glfwGetKey`:

```
int keyE = glfwGetKey(hWindow, GLFW_KEY_E);
```

Povratna vrijednost specificira u kojem se stanju nalazi tražena tipka. Vrijednost može biti jedna od sljedećih konstanti:

- `GLFW_PRESS`
- `GLFW_REPEAT`
- `GLFW_RELEASE`
- `GLFW_UNKNOWN`

`GLFW_UNKNOWN` se koristi u slučaju kada GLFW nema token koji predstavlja traženi ključ.

Osim stanja tipki GLFW biblioteka prati i evente znakova, što se može koristiti za unos teksta.

5.4.2. Korisnički unos pomoću miša

Za potrebe ovog projekta potrebno je dohvatiti stanje lijeve i desne tipke miša, te poziciju kursora. Za dohvaćanje stanja tipki miša koristi se metoda `glfwGetMouseButton` koja prima *handler* za prozor, te ID tipke koju je potrebno pregledati. Povratna vrijednost može biti `GLFW_PRESS` ili `GLFW_RELEASE`.

```
int left = glfwGetMouseButton(hWindow, GLFW_MOUSE_BUTTON_LEFT);
int right = glfwGetMouseButton(hWindow, GLFW_MOUSE_BUTTON_RIGHT);
```

Pozicija kursora se može dohvatiti pomoću `glfwGetCursorPos` metode. Ova metoda demonstrira jednu od situacija gdje je razlika u korištenju GLFW biblioteke s Java programskim jezikom poprilično izražena. Radi činjenice da Java programski jezik ne podržava *pointere*, metoda će vrijednosti x i y pozicije spremiti u `DoubleBuffer` objekt ili statično polje, ovisno o *overloadu* metode koji se koristi.

```
double[] cursorX = new double[1];
double[] cursorY = new double[1];

glfwGetCursorPos(hWindow, cursorX, cursorY);
```

Bitno je još napomenuti da je pozicija kursora relativna gornjem lijevom kutu prozora. GLFW također prati evente *scrollanja*, te omogućuje prilagodbu načina na koji se kursor prikazuje u prozoru. U ovom projektu kursor je skriven, što je ostvareno pozivom sljedeće metode:

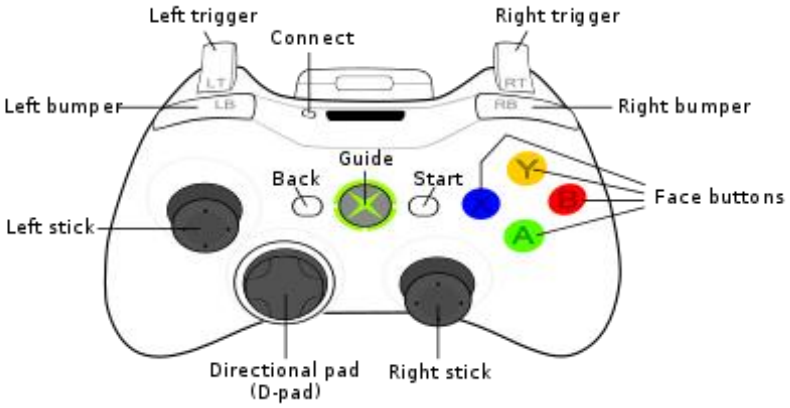
```
glfwSetInputMode(hWindow, GLFW_CURSOR, GLFW_CURSOR_HIDDEN);
```

5.4.3. Korisnički unos pomoću joysticka

Kada je GLFW inicijaliziran, svakom detektiranom *joysticku* je pridružen indeks kojeg će zadržati dok god je spojen. Indeksi počinju s vrijednosti `GLFW_JOYSTICK_1` i završavaju s vrijednosti `GLFW_JOYSTICK_LAST`.

GLFW podržava do 16 *joysticka* u isto vrijeme, a prisutnost *joysticka* na određenom indeksu može se provjeriti korištenjem sljedeće metode:

```
int present = glfwJoystickPresent(GLFW_JOYSTICK_1);
```



[13]

Slika 5.4. XBOX 360 kontroler

Stanje gumba se može dohvatiti pomoću `glfwGetJoystickButtons` metode, koja vraća stanje gumba za dani *joystick* indeks. Vrijednost stanja gumba može biti `GLFW_PRESS` ili `GLFW_RELEASE`.

```
ByteBuffer buttons = glfwGetJoystickButtons(GLFW_JOYSTICK_1);
```

Sljedeći bitan dio korisničkog unosa pomoću *joysticka* čine osi (engl. *axes*). Na XBOX 360 kontroleru pod osi spadaju: *left/right trigger* i *left/right stick* (Slika 4). Stanje osi se može dohvatiti korištenjem `glfwGetJoystickAxes` metode, koja vraća stanje osi za dani *joystick* indeks.

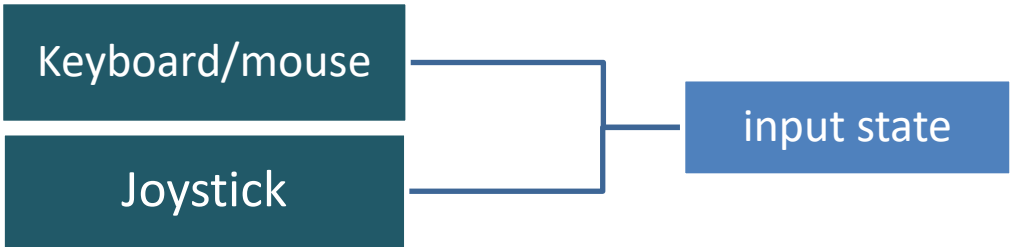
```
FloatBuffer axes = glfwGetJoystickAxes(GLFW_JOYSTICK_1);
```

Vrijednost stanja osi je broj od -1.0 do 1.0. Prilikom čitanja vrijednosti osi bitno je uzeti u obzir činjenicu da ovisno o kontroleru vrijednosti će imati nepreciznosti. Primjerice, za XBOX 360 kontroler je vrijednosti od -0.2 do 0.2 je potrebno ignorirati, jer se u tom stanju može naći os koja je u stanju mirovanja.

5.4.4. Kombiniranje korisničkog unosa

Različiti dijelovi igre mogu koristiti različite kontrole, također je moguće mijenjati izvor unosa. Iz tog razloga klasa za procesiranje korisničkog unosa je specifična za svaki dio igre. Za ovaj projekt, klasa koja procesira korisnički unos koristi *flag* koji definira koji tip unosa se u tom trenutku koristi. Ovisno o tipu unosa, skupljaju se vrijednosti iz pravilnih izvora, te se kombiniraju u varijable koje su dostupne izvana. Na ovakav način je također moguće ostvariti kontrole koje se aktiviraju pritiskom na više gumbova u isto vrijeme.

[13] http://compat.cemu.info/wiki/File:360_controller.svg



Slika 5.5. Kombiniranje korisničkih unosa

Ovime se rješava problem potrebe za različitim tipovima unosa. Primjerice, kombiniranje kontrola za pokretanje igrača u različitim smjerovima može se implementirati na sljedeći način:

```
switch (_inputMode)
{
    case INM_KEYBOARD:
        _UP = glfwGetKey(hWindow, GLFW_KEY_W) == GLFW_PRESS;
        _LEFT = glfwGetKey(hWindow, GLFW_KEY_A) == GLFW_PRESS;
        _DOWN = glfwGetKey(hWindow, GLFW_KEY_S) == GLFW_PRESS;
        _RIGHT = glfwGetKey(hWindow, GLFW_KEY_D) == GLFW_PRESS;
        break;
    case INM_JOYSTICK:
        buttons = glfwGetJoystickButtons(GLFW_JOYSTICK_1);
        lastButtonID = 1;
        while (buttons.hasRemaining())
        {
            lastButtonValue = buttons.get();
            switch (lastButtonID)
            {
                case BTN_UP:
                    _UP = lastButtonValue == GLFW_PRESS;
                    break;
                case BTN_LEFT:
                    _DOWN = lastButtonValue == GLFW_PRESS;
                    break;
                case BTN_DOWN:
                    _LEFT = lastButtonValue == GLFW_PRESS;
                    break;
                case BTN_RIGHT:
                    _RIGHT = lastButtonValue == GLFW_PRESS;
                    break;
            }
            lastButtonID++;
        }
}
```

Kod 5.7. Kombiniranje kontrola za pokretanje igrača

Stanje ovih kontrola se može pročitati izvana, te klasa odgovorna za procesiranje unosa može odraditi potrebne operacije, ovisno o snimljenom stanju.

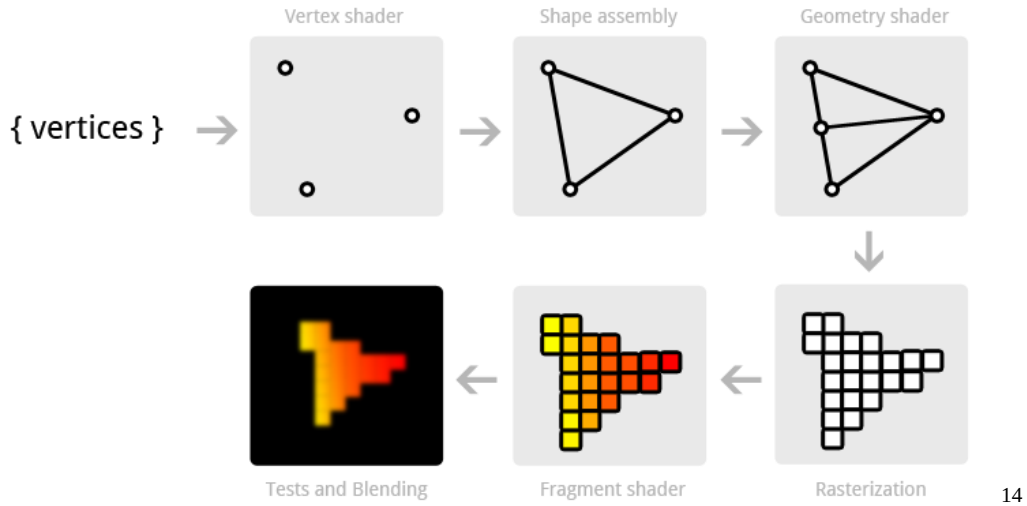
5.5. 3D grafika

Iscrtavanje grafičkog prikaza stanja igre na ekran je najbitniji aspekt svake video igre. Ljudi puno lakše procesiraju vizualne informacije, što grafiku igre čini jako bitnim načinom komunikacije s igračem.

Grafika se generalno može procesirati na dva načina, pomoću procesora za generalnu uporabu ili pomoću grafičkog procesora. Iako je procesor za generalnu uporabu više nego dovoljan za manje igre, za vizualno kompleksnije igre to nije dovoljno. Glavni problem je u tome što generalni procesor također obavlja poslove izračunavanja stanja igre, učitavanja datoteka, i sl. Ovaj projekt za iscrtavanje koristi grafički procesor, pomoću OpenGL biblioteke.

5.5.1. Proces iscrtavanja grafičkih elemenata

Prije OpenGL verzije 2.0 programeri su za iscrtavanje koristili fiksni set funkcija, što je bilo jako ograničavajuće. Verzija 2.0 je uvela koncept *shader* programa putem *vertex* i *fragment shadera* koji su zamijenili dio fiksnog seta funkcija. Na to se u verziji 3.0 nadogradilo u obliku *geometry shadera*, a verzija 4.0 je uvela *tasselation shadere*. *Shader* programi omogućavaju programerima slobodu u određivanju kako bi se određeni grafički element trebao iscrtati.



Slika 5.6. Procesiranje grafičkih elemenata

[14] https://open.gl/drawing

Proces iscrtavanja 3D grafike je set transformacija čiji je cilj proizvesti 2D sliku koja se prikazuje na ekranu (Slika 5.6).

Da bi se određeni grafički element mogao iscrtati, potrebno je prvo grafičkom procesoru poslati listu točaka koje čine taj element. Ovisno o tipu grafičkog elementa, točka može biti u 2D ili 3D prostoru.

Prvi korak u procesiranju je pozicioniranje točki u OpenGL koordinatni sustav, to se vrši u *vertex shaderu*. Nakon toga dolazi *geometry shader,* koji spaja sve relativne točke, te generira primitivne poligone. Grafički procesori koriste trokute, te ih kombiniraju u kompleksnije poligone.

Sljedeći dio je proces *rasteringa,* u kojem se izbaci geometrija koja nije unutar vidljivog raspona te se trokuti generirani u *geometry shaderu* transformiraju u fragmente. Fragmente zatim procesira *fragment shader,* koji svakom fragmentu dodjeljuje boju te generira piksele koji se zatim zapisuju u međuspremnik za iscrtavanje.

5.5.2. Shader program

Shader program je program koji se pokreće na grafičkom procesoru. *Shader* programi napisani za potrebe ovog projekta sastoje se od *vertex shadera* i *fragment shadera*. To znači da *geometry shader* nije specificiran, ali to ne predstavlja problem jer *geometry shader* predstavlja opcionalni *shader* program. Proces generiranja poligona će se odraditi po predefiniranom načinu.

Shader programi su neovisni te međusobno komuniciraju samo preko ulaznih i izlaznih vrijednosti. Bitno je napomenuti da programi nemaju kontrole nad tijekom grafičkog procesiranja, već su samo specificiraju kako će se određeni korak procesiranja odraditi. Bitno je još napomenuti da se *vertex shader* izvršava jednom za svaku točku, dok se *fragment shader* izvršava jednom za svaki piksel.

Shader programi se pišu pomoću GLSL (*OpenGL Shading Language*) programskog jezika. Jezik se fokusira na pružanje funkcionalnosti koje se koriste u izračunima vektorske grafike.

Programi napisani pomoću GLSL programskog jezika ne moraju imati specifičnu ekstenziju, no za potrebe *syntax highlighta* u ovom projektu se koristi ekstenzija „.glsl.“[15]

[15] https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)

```
#version 330

layout (location = 0) in vec3 i_positions;
out vec3 vs_colour;
uniform float u_time;

void main(void)
{
   float red = fract(u_time);
   float green = 1.0 - fract(u_time);
   float blue = abs(red - green);

   vs_colour = vec3(red, green, blue);
   gl_Position = vec4(i_positions, 1.0);
}
```

Kod 5.8. Primjer *vertex shadera*

Na primjeru u programskom isječku 5.8 mogu se vidjeti ključni dijelovi *shader* programa. Prvo se specificira verzija GLSL-a koja se koristi, u ovom slučaju to je GLSL 330, odnosno OpenGL verzija 3.3. Sve verzije GLSL-a mogu se vidjeti u tablici 5.1.

Tablica 5.1. GLSL verzije

GLSL	**110**	**120**	**130**	**140**	**150**	**330**	**400**	**410**	**420**	**430**
OpenGL	2.0	2.1	3.0	3.1	3.2	3.3	4.0	4.1	4.2	4.3

Shaderi komuniciraju međusobno putem ulaznih i izlaznih vrijednosti. Ulazne vrijednosti ispred imaju ključnu riječ `in`, dok izlazne vrijednosti ispred imaju ključnu riječ `out`. Kako je *vertex shader* uvijek prvi, ulazne vrijednosti prima od aplikacije. Način na koji aplikacija prosljeđuje vrijednosti *vertex shaderu* je pomoću VAO (*Vertex Array Object*) objekta. Ovaj koncept će biti detaljnije objašnjen u poglavlju koje se tiče učitavanja 3D modela. Za sada je dovoljno znati da će aplikacija proslijediti listu pozicija u obliku polja. Svaka pozicija se pohranjuje u varijabli tipa `vec3`, koja predstavlja trodimenzionalni vektor. Primjer ulazne strukture je prikazan u tablici 5.2.

Tablica 5.2. Struktura polja pozicija

x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	...	x_n	y_n	z_n

Globalne varijable se definiraju pomoću ključne riječi `uniform`, vrijednost ovih varijabli se ne mijenja za svako izvršavanje *shader* programa.

Glavni dio *shader* programa je funkcija `main` koja predstavlja prvu funkciju koja će biti pozvana prilikom svakog izvršavanja *shader* programa.

GLSL ima razne ugrađene varijable, od kojih je za potrebe ovog projekta bitna samo varijabla `gl_Position`, koja predstavlja rezultat izvršavanja *vertex shadera*. Tip ove varijable je `vec4` pa je stoga potrebno pozicije pretvoriti u `vec4` varijablu.

5.5.3. Učitavanje shader programa

U ovom projektu, *shader* programi se učitavaju pomoću klase `ShaderLoader` koja pomoću metode `load` vrši učitavanje, kao što se vidi u programskom isječku 5.9.

```
public static int load(String vsFilepath, String fsFilepath, byte key)
{
    int programID = glCreateProgram();

    try
    {
        String vertexSource = new String
        (
            Files.readAllBytes(Paths.get(vsFilepath)),
            UTF_8
        );

        String fragmentSource = new String
        (
            Files.readAllBytes(Paths.get(fsFilepath)),
            UTF_8
        );

        int vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertexShaderID, vertexSource);
        glCompileShader(vertexShaderID);
        glAttachShader(programID, vertexShaderID);

        int fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragmentShaderID, fragmentSource);
        glCompileShader(fragmentShaderID);
        glAttachShader(programID, fragmentShaderID);

        glLinkProgram(programID);

        glDetachShader(programID, vertexShaderID);
        glDetachShader(programID, fragmentShaderID);

        glDeleteShader(vertexShaderID);
        glDeleteShader(fragmentShaderID);

        SHADER_MAP.put(key, programID);
    }
    catch (IOException ex) { programID = -1; }

    return programID;
}
```

Kod 5.9. Učitavanje *shader* programa

Prvi korak pri učitavanju *shader* programa je stvaranje programa pomoću metode `glCreateProgram` koja će vratiti ID s kojim se može referencirati novo kreirani *shader* program. Novi program je prazan, te mu je potrebno pridružiti *vertex shader* i *fragment shader*. Kod za oba *shadera* se učitava s datotečnog sustava, *compajlira* i pridružuje *shader* programu. Nakon toga se izvršava *linkanje* čime nastaje *executable shader* program, kojeg se referencira korištenjem ID vrijednosti `programID`. Poslije *linkanja* učitani se *vertex shader* i *fragment shader* mogu pobrisati iz memorije.

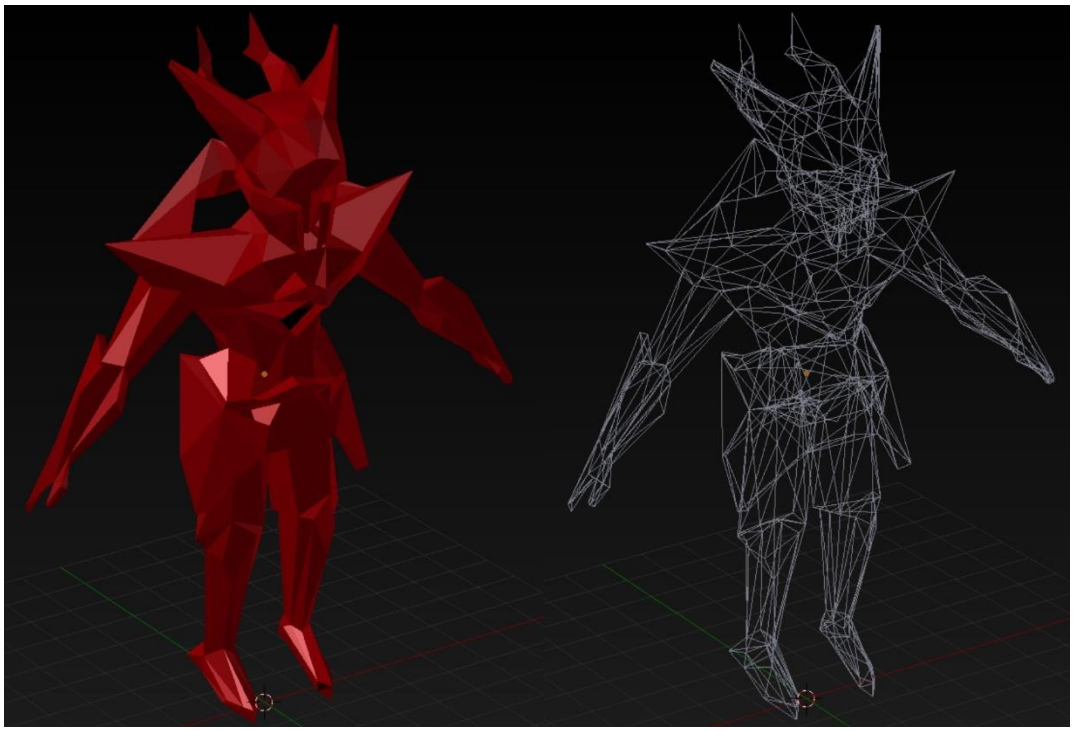
`ShaderLoader` klasa sprema reference za svaki aktivni *shader* program u *hash mapu* `SHADER_MAP` kako bi se olakšalo brisanje *shader* programa iz memorije pri zatvaranju aplikacije. Metode za brisanje dio su `ShaderLoader` klase, a definirane su na sljedeći način:

```
public static void unload(byte key)
{
    if (SHADER_MAP.containsKey(key))
        glDeleteProgram(SHADER_MAP.remove(key));
}

public static void unload(byte[] keys)
{
    for (byte key : keys)
    {
        if (SHADER_MAP.containsKey(key))
            glDeleteProgram(SHADER_MAP.remove(key));
    }
}

public static void unloadAll()
{
    Set<Byte> keys = SHADER_MAP.keySet();
    for (byte key : keys)
    {
        if (SHADER_MAP.containsKey(key))
            glDeleteProgram(SHADER_MAP.remove(key));
    }
}
```

Kod 5.10. Metode za brisanje *shader* programa

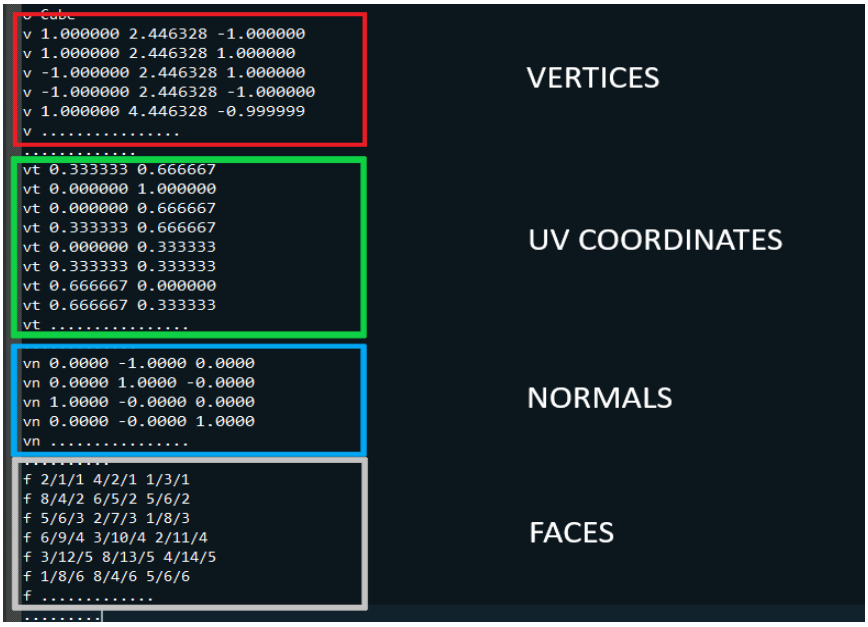
5.5.4. Učitavanje 3D modela

3D Modele čini velika količina međusobno povezanih podataka. Takvu podatkovnu strukturu je teško definirati ručno pa se stoga koriste razni alati za 3D modeliranje. Modeli napravljeni alatima za modeliranje se zatim izvoze u razne formate, koji se zatim uvoze i koriste u igrama. Za potrebe ovog projekta korišten je besplatni alat za 3D modeliranje Blender. Primjer modela može se vidjeti na slici 5.7.



Slika 5.7. Igračev model

Prilikom stvaranja modela za igru osim stila igre potrebno je imati na umu mogućnosti sustava za učitavanje i renderiranje modela. Modeli za igru Decipherer trebaju biti trianguliran, što znači da svi poligoni koji čine model moraju biti trokut. Ukoliko će model koristiti teksturu mora imati definirane UV koordinate, te normale ako reagira na osvjetljenje. Ovaj projekt koristi Wavefront (.obj) format za sve modele koji nemaju animaciju.



Slika 5.8. Format .obj datoteke

Wavefront format je jako jednostavan tekstualni format, te ga je radi toga lako učitati i parsirati. Kao što je prikazano na slici 5.8, iz Wavefront datoteke su bitna 4 tipa podataka.[16]

Osnova svakog modela su pozicije, odnosno točke u 3D prostoru (engl. *vertex positions*). Pozicije se sastoje od 3 vrijednosti koje skupa čine točku V_i s koordinatama (x_i, y_i, z_i).

Sljedeća bitna informacija su UV koordinate koje služe za mapiranje tekstura na površinu 3D modela. UV koordinate se sastoje od dvije vrijednosti koje skupa čine točku (x_i, y_i). Prilikom čitanja ovih vrijednosti bitno je imati na umu da je vrijednost y_i potrebno invertirati, iz razloga što su UV koordinate definirane u četvrtom kvadrantu Kartezijevog koordinatnog sustava.

Nakon toga slijede vrijednosti za izračun normale. Svaki trokut 3D modela ima jednu normalu, tj. jedinični vektor koji je okomit na površinu trokuta te služi za izračunavanje osvjetljenja modela.

Posljednja potrebna informacija je lista indeksa koja specificira kako je potrebno kombinirati pozicije, UV koordinate i normale da bi se uspješno složio 3D model. Bitno je napomenuti da se 3 grupe indeksa u ovom slučaju kombiniraju u jedan trokut jer je model trianguliran.

Primjerice, `f 2/1/1` specificira da pozicija na indeksu 2 koristi UV koordinate na indeksu 1 i normalu na indeksu 1. Iz vrijednosti indeksa za pozicije se također konstruira polje koje definira redoslijed procesiranja točaka 3D modela.

Pomoću liste indeksa potrebno je presložiti pročitane podatke tako da ih je moguće grupirati. Cilj je imati točku koja ima svoju poziciju, UV koordinate te vrijednost normale.

Kada su svi podaci pravilno raspoređeni, potrebno ih je spremiti u memoriju grafičke kartice. Ovo se radi pomoću VAO (*Vertex Array Object*) i VBO (*Vertex Buffer Object*) objekata.

VAO se može zamisliti kao polje, čiji je svaki element VBO objekt. VBO je polje koje služi za spremanje niza brojeva koji predstavljaju pozicije, UV koordinate, normale, boje i sl.[17]

U ovom projektu klasa `ModelLoader` je odgovorna za parsiranje „.obj“ datoteka i spremanje podataka u VAO objekt. Kreiranje VAO objekta se vrši u metodi `createVAO`, koja je definirana u programskom isječku 5.11.

[16] https://en.wikipedia.org/wiki/Wavefront
[17] https://www.khronos.org/opengl/wiki/Vertex_Specification

```
public static int[] createVAO
(
    FloatBuffer positionBuffer,
    FloatBuffer normalBuffer,
    FloatBuffer uvBuffer,
    IntBuffer indexBuffer,
    int vertexCount,
    byte key
)
{
    int[] modelData = new int[2];
    modelData[0] = vertexCount;

    int VAO = glGenVertexArrays();
    glBindVertexArray(VAO);
    modelData[1] = VAO;

    int positionsVBO = glGenBuffers();
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    glBufferData(GL_ARRAY_BUFFER, positionBuffer, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, false, 12, 0);

    int normalsVBO = glGenBuffers();
    glBindBuffer(GL_ARRAY_BUFFER, normalsVBO);
    glBufferData(GL_ARRAY_BUFFER, normalBuffer, GL_STATIC_DRAW);
    glVertexAttribPointer(1, 3, GL_FLOAT, false, 12, 0);

    int uvVBO = glGenBuffers();
    glBindBuffer(GL_ARRAY_BUFFER, uvVBO);
    glBufferData(GL_ARRAY_BUFFER, uvBuffer, GL_STATIC_DRAW);
    glVertexAttribPointer(2, 2, GL_FLOAT, false, 8, 0);

    int indicesVBO = glGenBuffers();
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indicesVBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indexBuffer, GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

    memFree(positionBuffer);
    memFree(normalBuffer);
    memFree(uvBuffer);
    memFree(indexBuffer);

    MODEL_MAP.put(key, modelData);

    return modelData;
}
```

Kod 5.11. Kreiranje VAO objekta

Podatci za 3D modele mogu zauzimati dosta mjesta pa se iz tog razloga za alokaciju koristi klasa LWJGL biblioteke `MemoryUtil` koja omogućava alokaciju memorije izvan Java *heapa*. Metoda za alociranje `memAlloc` koristi standardnu C funkciju `malloc` za alokaciju bloka memorije izvan Java *heapa*.

Ovo je jako bitno jer je veličina Java *heapa* često ograničena, te ovisi o konfiguraciji Java virtualne mašine. Za vrijeme testiranja ovog projekta učitavanje većih modela koristeći Java *heap* je rezultiralo u `OutOfMemory` greški. Svi podaci modela su radi tog razloga alocirani izvan JVM *heapa*, te se nalaze u odgovarajućim `Buffer` objektima.

VAO objekt se kreira korištenjem `glGenVertexArrays` metode koja će vratiti ID pomoću kojeg je moguće referencirati novo stvoreni VAO objekt. Prije nego je moguće dodijeliti VBO objekte, potrebno je specificirati da se trenutno koristi ovaj VAO objekt. To se vrši pomoću `glBindVertexArray` metode.

Za svaki niz podataka se zatim stvara VBO objekt pomoću `glGenBuffers` metode koja vraća ID pomoću kojeg se može referencirati novo stvoreni VBO objekt.

Svaki VBO objekt je potrebno napuniti podatcima, no prije toga je potrebno OpenGL-u dati do znanja koji se VBO objekt koristi pomoću `glBindBuffer` metode. Nakon što je specificirano koji se VBO objekt koristi pomoću metode `glBufferData` se određuje sadržaj tog objekta.

Prvi parametar `glBufferData` metode specificira tip objekta, u ovom slučaju `GL_ARRAY_BUFFER` što znači da će se u ovaj VBO objekt spremati podaci koji predstavljaju atribut točke (pozicija, UV koordinate, vrijednosti normale). Nakon toga se prosljeđuje `Buffer` s podatcima, te posljednji parametar `GL_STATIC_DRAW` koji specificira da se vrijednosti pohranjene u ovaj VBO objekt neće mijenjati. Nakon što je VBO objekt napunjen podacima, potrebno je odrediti kako će se podaci tog objekta koristiti.

Ovo se može definirati pomoću `glVertexAttribPointer` metode. Prvi parametar predstavlja lokaciju atributa u VAO objektu, pomoću te lokacije se podaci iz VBO objekta mogu dohvatiti u *shader* programu.

Sljedeći parametar definira koliko vrijednosti čini jedan podatak, primjerice, za poziciju je vrijednost ovog parametra 3. Nakon toga se specificira tip vrijednosti, te je li potrebno normalizirati vrijednosti, što je definirano `boolean` *flagom*.

Nakon toga se definira veličina jednog podatka u bajtovima, te poziciju na kojoj je potrebno početi čitati podatak u ovom objektu.

Ovaj proces je isti za podatke o poziciji, UV koordinate i normale. Jedino se za podatke o redoslijedu iscrtavanja točaka koristi drugačiji pristup. VBO objekt za podatke o redoslijedu

je tipa `GL_ELEMENT_ARRAY_BUFFER`, što znači da podaci u ovom objektu sadržavaju indekse drugog VBO objekta.

Kada su svi VBO objekti uspješno dodani u VAO objekt, oslobađa se memorija svih `Buffer` objekata pomoću `memFree` metode klase `MemoryUtil`.

Bitno je još napomenuti da klasa `ModelLoader` sprema ID VAO objekta i broj točaka za svaki učitani model u *hash mapu* `MODEL_MAP` kako bi se olakšao proces oslobađanja memorije. Metode za oslobađanje alocirane memorije su sljedeće:

```
public static void unload(byte key)
{
    if (MODEL_MAP.containsKey(key))
    {
        int VAO = MODEL_MAP.remove(key)[1];

        glDeleteBuffers(VAO);
        glDeleteVertexArrays(VAO);
    }
}

public static void unload(byte[] keys)
{
    int VAO;

    for (Byte key : keys)
    {
        if (MODEL_MAP.containsKey(key))
        {
            VAO = MODEL_MAP.remove(key)[1];

            glDeleteBuffers(VAO);
            glDeleteVertexArrays(VAO);
        }
    }
}

public static void unloadAll()
{
    Set<Byte> keys = MODEL_MAP.keySet();
    int VAO;

    for (Byte key : keys)
    {
        VAO = MODEL_MAP.remove(key)[1];

        glDeleteBuffers(VAO);
        glDeleteVertexArrays(VAO);
    }
}
```

Kod 5.12. Metode za brisanje VAO objekata

5.5.5. Učitavanje teksture

Tekstura je obična slika, koja se pomoću UV koordinata može mapirati na površinu 3D modela. Ovaj projekt za teksture koristi slike PNG (*Portable Network Graphics*) formata. Proces učitavanja teksture se vrši u `load` metodi `TextureLoader` klase (Kod 5.13).

```
public static final int load(String filepath, byte key)
{
    int textureID = glGenTextures();

    try
    {
        PNGDecoder decoder = new PNGDecoder(new FileInputStream(filepath));

        int width = decoder.getWidth();
        int height = decoder.getHeight();

        ByteBuffer data = memAlloc(4 * width * height);
        decoder.decode(data, width * 4, PNGDecoder.Format.RGBA);
        data.flip();

        glBindTexture(GL_TEXTURE_2D, textureID);
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

        glTexImage2D
        (
            GL_TEXTURE_2D,
            0,
            GL_RGBA,
            width,
            height,
            0,
            GL_RGBA,
            GL_UNSIGNED_BYTE,
            data
        );

        glGenerateMipmap(GL_TEXTURE_2D);
        TEXTURE_MAP.put(key, textureID);

        memFree(data);
    }
    catch (IOException ex) { textureID = -1; }

    return textureID;
}
```

Kod 5.13. Učitavanje nove teksture

Prvi korak pri učitavanju teksture je dekodiranje PNG formata, za ovo se koristi gotova klasa `PNGDecoder` iz samostalne biblioteke *pngdecoder-1.0.jar*.[18]

[18] https://github.com/mattdesl/slim/blob/master/slim/src/slim/texture/io/PNGDecoder.java

Nova tekstura se kreira pozivom metode `glGenTextures` koja vraća ID pomoću kojeg se može referencirati ta tekstura.

Nakon toga se specificira način na koji su posloženi pikseli slike, te se nakon toga vrijednost dekodirane slike sprema pozivom `glTexImage2D`.

Na kraju se generira *mipmap* za tu teksturu, *mipmap* predstavlja tehniku gdje postoji više verzija iste teksture, a svaka verzija je drugačije rezolucije. Ovo se zatim može primijeniti da se na objekte koji su više udaljeni primjenjuje tekstura manje rezolucije, dok se na bliže objekte primjenjuje tekstura više rezolucije.

Još je bitno napomenuti da je slika koja se koristi za teksturu pravilnog oblika gdje su obje strane jednake i potencija broja dva, primjerice: `32x32`, `64x64`, `128x128`, `256x256`, `512x512`, `1024x1024`.

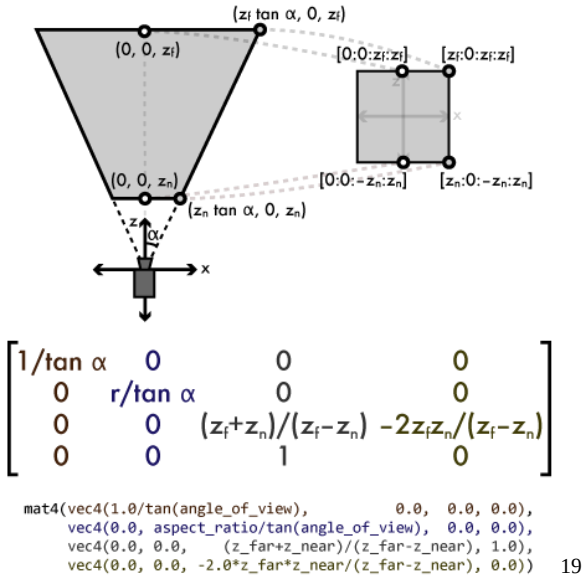
Klasa `TextureLoader` također pamti sve ID vrijednosti tekstura koje su učitane, radi olakšavanja procesa brisanja tekstura. Metode kojima se vrši brisanje tekstura su definirane u programskom isječku 5.14.

```
public static void unload(byte key)
{
    if (TEXTURE_MAP.containsKey(key))
        glDeleteTextures(TEXTURE_MAP.remove(key));
}

public static void unload(byte[] keys)
{
    for (byte key : keys)
    {
        if (TEXTURE_MAP.containsKey(key))
            glDeleteTextures(TEXTURE_MAP.remove(key));
    }
}

public static void unloadAll()
{
    Set<Byte> keys = TEXTURE_MAP.keySet();
    int VAO;

    for (byte key : keys)
    {
        glDeleteTextures(TEXTURE_MAP.remove(key));
    }
}
```

Kod 5.14. Brisanje tekstura

5.5.6. Projekcija 3D svijeta

Kada bi se model iscrtao na ekran u trenutnom stanju, rezultat bi bio daleko od očekivanog. Glavni razlog za to je činjenica da trenutno ne postoji koncept udaljenosti, te je z koordinata praktički beskorisna. Rješenje tog problema je matrica za projekciju (engl. *projection matrix*) koja se izračunava po formuli na slici 5.9.



[19]

Slika 5.9. Formula matrice za projekciju

Matrice su ovom projektu reprezentirane klasom `Matrix4f` koja sadržava i osnovne operacije kao što su operacije za množenje, postavljanje identiteta i sl. Matrica za projekciju se izračunava prilikom inicijalizacije `Renderer` klase te se pohranjuje kao konstanta jer se vrijednosti ove matrice nikada neće mijenjati. Određena se pozicija može projektirati tako da se množi s ovom matricom. Ta operacija se vrši u *vertex shaderu*, a matrica je proslijeđena kao *uniform* varijabla.

5.5.7. 3D transformacije

Projekcija osigurava da se točke 3D objekata pravilno iscrtavaju na ekran, no model će se uvijek iscrtavati na istoj poziciji i pod istim kutem. Za potrebe ovog projekta svakom modelu je moguće mijenjati veličinu, te odrediti poziciju i rotaciju. Svaki objekt koji se nalazi u igri je instanca klase `Entity` koja je definirana u programskom isječku 5.15.

[19]http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-3:-3D-transformation-and-projection.html

```
public abstract class Entity
{
    private static int counter;

    static
    {
        counter = 0;
    }

    public final int _ID;

    public Entity()
    {
        _ID = counter++;
    }

    public abstract void initialize();
    public abstract void update(long delta);
    public abstract void terminate();
}
```

Kod 5.15. Klasa Entity

Svaki sistem igre koji odrađuje neki posao specificira *interface* pomoću kojeg se ostvaruje komunikacija sa sustavom. Primjerice, *interface* `Transformable`:

```
public interface Transformable
{
    Matrix4f transformation();
}
```

Matrica transformacije je rezultat promjene veličine, promjene rotacije i promjene pozicije, u tom redoslijedu. Objekt koji je transformiran se nalazi u koordinatama svijeta (engl. *world space*).

5.5.8. Koncept kamere

Pomoću transformacija objekt u igri može se transformirati u željenu poziciji, ali kamera uvijek stoji na mjestu. Kamera je zamišljeni objekt koji predstavlja poziciju i kut iz kojeg igrač promatra scenu.

Kamera je uvijek pozicionirana na lokaciji (0, 0, 0) i to se nikada neće promijeniti. Da bi se stvorio dojam da se kamera pomiče potrebno je transformirati sve objekte svijeta u obrnutom smjeru transformacije kamere.

Primjerice, ukoliko se kamera miče u lijevo, svi objekti svijeta se trebaju micati u desno. Naravno, veličina kamere ne igra nikakvu ulogu pa se ta transformacija neće raditi.

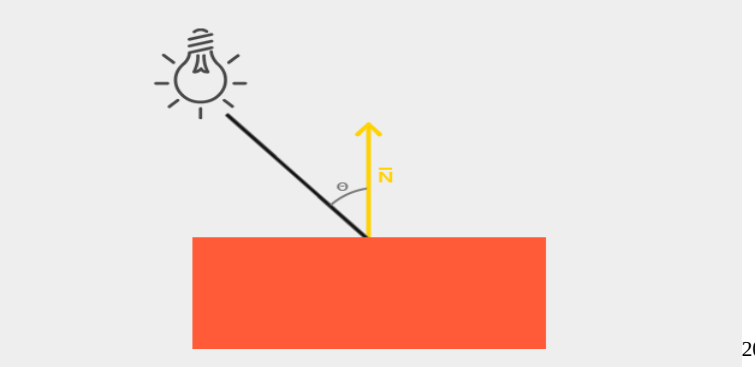
Matrica transformacije kamere se tako sastoji od translacije i rotacije u obrnutom smjeru zamišljene transformacije kamere. Ovakvu matricu nazivamo matricom pogleda (engl. *view matrix*).

5.5.9. Osvjetljenje 3D svijeta

Za potrebe ovog projekta koristi se samo jedan izvor svijetla koji obasjava sve elemente u vidljivom prostoru. Tip osvjetljenja koji se koristi je Gouraud *shading*, koji je specifičan po tome da se osvjetljene računa po točki modela, za razliku od druge poznate tehnike Phong pomoću koje se osvjetljenje računa za svaki fragment.

Radi toga je Gouraud osvjetljenje pogodno za igre s malom razinom detalja. Dok se Phong koristi za realističnije rezultate. Također Gouraud tehnika će se puno brže izvršiti radi činjenice da je potrebno izvršiti manje kalkulacija za izračun svjetla. Za izračun difuzne komponente svjetla potrebna je pozicija svjetla te vektor normale. Vektor iz trenutne točke modela prema izvoru svijetlosti s normalom zatvara kut koji se koristiti za računanje intenziteta svjetla na površini modela (Slika 5.10).

Ove kalkulacije se izvršavaju u *vertex shaderu*.



[20]

Slika 5.10. Izračun difuznog svjetla

5.5.10. Renderiranje 3D grafike

Posao renderiranja odrađuje klasa `Renderer` koja sadržava metode za renderiranje 3D modela, animiranih 3D modela, čestica i sl.

Slijedi implementacija renderiranja 3D modela s osvjetljenjem i teksturom.

[20] https://learnopengl.com/Lighting/Basic-Lighting

Klasa `Renderer` čuva *hash mapu* podataka o *shaderima* koji se koriste za renderiranje različitih tipova modela. Klasa koja predstavlja jedan *shader* je `ShaderData` klasa iz programskog isječka 5.17.

```
public class ShaderData
{
    public final byte _ID;
    public final int _SHADER_PROGRAM;
    private final HashMap<String, Integer> uniforms;

    public ShaderData(byte id, int shaderProgram, String[] uniformNames)
    {
        _ID = id;
        _SHADER_PROGRAM = shaderProgram;
        uniforms = new HashMap<>();

        for (String name : uniformNames)
        {
            uniforms.put(name, glGetUniformLocation(shaderProgram, name));
        }
    }

    public int location(String uniformName)
    {
        return uniforms.get(uniformName);
    }
}
```

Kod 5.16. Klasa ShaderData

Sve instance `ShaderData` klase za određenu sekciju igre se stvaraju prilikom učitavanja *shader* programa. Svaka instanca pamti *shader* program kojeg je potrebno koristiti, te dohvaća lokacije svih *uniform* varijabli.

```
public class ModelData
{
    public final byte _ID;
    public final int _VAO;
    public final int _VERTEX_COUNT;
    public final int[] _ATTRIB_POINTERS;

    public ModelData
    (
        byte id, int vao, int vertexCount, int apPosition,
        int apNormal, int apUV, int apBone, int apWeights
    ){
        _ID = id;
        _VAO = vao;
        _VERTEX_COUNT = vertexCount;

        _ATTRIB_POINTERS = new int[] {
            apPosition, apNormal, apUV, apBone, apWeights
        };
    }
}
```

Kod 5.17. Klasa ModelData

`Renderer` također čuva *hash mapu* podataka o modelima koji se trenutno koriste (programski isječak 5.18). Na takav način se olakšava upravljanje trenutno aktivnim modelima. Svaki učitani model je definiran ID vrijednosti VAO objekta, brojem točki te poljem lokacija atributa. Osim standardnih atributa klasa `ModelData` pamti i lokacije atributa za animiranje, te vrijednosti biti će objašnjene u poglavlju koje se bavi animacijom.

Teksture su previše jednostavne da bi se za njih izradila klasa pa se tako aktivne teksture čuvaju samo kao broj, te svaki entitet ima podatak o teksturi koju trenutno koristi.

```
private static final HashMap<Byte, ShaderData> SHADER_MAP;
private static final HashMap<Byte, ModelData> MODEL_MAP;

private static Matrix4f projectionMatrix;
private static Vector3f lightpos;

private static ShaderData currentShader;
private static ModelData currentModel;

private static Matrix4f vm;
private static Matrix4f mvm;
private static Matrix4f mvpm;
```

Kod 5.18. Članovi klase Renderer

Prije nego je moguće objasniti metodu za renderiranje potrebno se podsjetiti nekoliko detalja koji se tiču trenutne konfiguracije OpenGLa.

```
glFrontFace(GL_CW);
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```

Prilikom iscrtavanja 3D objekata cilj je smanjiti količinu poligona koje je potrebno iscrtati. Tehnika kojom se broj potrebnih poligona smanjuje naziva se culling. U ovom projektu se koristi *backface culling*, što znači da se poligoni sa stražnje strane modela neće iscrtavati.

Način na koji ova tehnika prepoznaje stranu na kojoj se poligon nalazi je pomoću smjera u kojem se poligon iscrtava. U ovom slučaju pomoću metode `glFrontFace` definirano je da se poligoni koji se nalaze na prednjoj strani modela iscrtavaju u smjeru kazaljke na satu.

Nakon što je definiran smjer, pomoću metode `glCullFace` se definira koje poligone je potrebno izostaviti iz iscrtavanja, u ovom slučaju `GL_BACK` za poligone koji se nalaze na stražnjoj strani modela.

Na kraju je potrebno uključiti culling pozivom metode `glEnable` s parametrom `GL_CULL_FACE`.

```
public static void renderTexturedShaded
(Matrix4f transformation, byte shader, byte model, int texture)
{
    vm = camera.viewMatrix();
    mvm = transformation.mul(vm);
    mvpm = transformation.mul(vm.mul(projectionMatrix));

    currentModel = MODEL_MAP.get(model);
    currentShader = SHADER_MAP.get(shader);

    glUseProgram(currentShader._SHADER_PROGRAM);

    try (MemoryStack stack = stackPush())
    {
        FloatBuffer mvpmBuffer = stack.mallocFloat(16);
        mvpmBuffer.put(mvpm._value);
        mvpmBuffer.flip();

        glUniformMatrix4fv(
            currentShader.location("u_model_view_projection"),
            true, mvpmBuffer
        );

        FloatBuffer mvmBuffer = stack.mallocFloat(16);
        mvmBuffer.put(mvm._value);
        mvmBuffer.flip();

        glUniformMatrix4fv(
            currentShader.location("u_model_view"), true, mvmBuffer
        );

        FloatBuffer lightposBuffer = stack.mallocFloat(3);
        lightposBuffer.put(lightpos._value);
        lightposBuffer.flip();

        glUniform3fv(
            currentShader.location("u_light_position"),
            lightposBuffer
        );
        glUniform1i(currentShader.location("u_sampler"), 0);
    }

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);

    glBindVertexArray(currentModel._VAO);

    for (int ap : currentModel._ATTRIB_POINTERS)
        if (ap != -1) glEnableVertexAttribArray(ap);

    glDrawElements(
        GL_TRIANGLES, currentModel._VERTEX_COUNT, GL_UNSIGNED_INT, 0
    );

    for (int ap : currentModel._ATTRIB_POINTERS)
        if (ap != 1) glDisableVertexAttribArray(ap);
}
```

Kod 5.19. Metoda za iscrtavanje 3D modela s teksturom i osvjetljenjem

Prvi korak u iscrtavanju modela (programski isječak 5.20) je izračun matrica koje će se koristiti za transformaciju pozicija. Matrice se mogu kombinirati tako da se međusobno množe, a ovime se smanjuje količina podataka koju je potrebno poslati *shader* programu.

Nakon toga se dohvaćaju mapirani podaci za *shader* program i model. Metoda `glUseProgram` specificira koji je *shader* program trenutno u uporabi.

Kada je odabran *shader* program potrebno je postaviti vrijednosti svih *uniform* varijabli koje taj *shader* koristi. Za alokaciju potrebne memorije koristi se `MemoryStack` klasa LWJGL biblioteke. `Memory Stack` klasa je namijenjena za alokaciju manje količine memorije. `Matrix4f` klasa čuva vrijednosti matrice u polju od 16 elemenata koje se naziva `_value`, te je javno dostupno. Ta vrijednost se zatim pohranjuje u `FloatBuffer` objekt.

Koristeći metodu `glUniformMatrix4fv` *shader* programu se prosljeđuje matrica. Ta metoda prima tri parametra. Prvi parametar specificira poziciju uniformne varijable. Drugi parametar je *flag* koji specificira način na koji je matrica posložena u polje, u ovom slučaju vrijednost `true` znači da matrica sprema u polje red po red. Treći parametar je vrijednost matrice.

Pomoću metode `glUniform3fv` *shader* programu se prosljeđuje trodimenzionalni vektor. Ova metoda prima lokaciju *uniform* varijable i vrijednost vektora.

Za mapiranje teksture na objekt, koristi se uniformna varijabla *sampler* koja reprezentira jednu teksturu. Postoje različiti tipovi *samplera*, a za potrebe ovog projekta koristi se `sampler2D`, odnosno *sampler* za 2D teksturu.

Sampler se prosljeđuje korištenjem `glUniform1i` metode, koja prima lokaciju uniformne varijable te indeks. Nakon toga se indeks aktivne teksture definira pomoću metode `glActiveTexture`. Podaci o teksturi se zatim vežu za aktivni indeks korištenjem metode `glBindTexture` koja prima tip teksture i ID teksture.

Nakon toga potrebno je specificirati koji VAO objekt će se koristiti za iscrtavanje, pomoću metode `glBindVertexArray`. Te se specificiraju aktivni atributi modela korištenjem metode `glEnableVertexAtrribArray`.

Na kraju se model iscrtava pozivom metode `glDrawElements`. Prvi parametar specificira da se koriste trokuti za konstrukciju poligona, sljedeći parametar specificira broj točaka u modelu. Nakon toga se specificira tip indeksa koji se koriste za određivanje redoslijeda iscrtavanja te opcionalni *pointer* na polje koje sadrži indekse.

```
#version 330

layout (location = 0) in vec3 i_position;
layout (location = 1) in vec3 i_normal;
layout (location = 2) in vec2 i_texcoord;

out vec2 vs_texcoord;
out float vs_brightness;

uniform mat4 u_model_view_projection;
uniform mat4 u_model_view;
uniform vec3 u_light_position;

void main(void)
{
   vec3 world_position = (u_model_view * vec4(i_position, 1.0)).xyz;
   vec3 world_normal = (u_model_view * vec4(i_normal, 0.0)).xyz;
   vec3 light_vector = normalize(u_light_position - world_position);

   float diffuse = max(dot(world_normal, light_vector), 0.04);

   vs_texcoord = i_texcoord;
   vs_brightness = diffuse;

   gl_Position = u_model_view_projection * vec4(i_position, 1.0);
}
```

Kod 5.20. *Vertex shader* za procesiranje 3D modela s teksturom i osvjetljenjem

Vertex shader definiran u programskom isječku 5.21, transformira pozicije modela pomoću proslijeđenih matrica te izvršava kalkulaciju difuzne svjetlosti za svaku točku.

```
#version 330

in vec2 vs_texcoord;
in float vs_brightness;

out vec4 o_colour;
uniform sampler2D u_sampler;

const float c_shading_levels = 4.0;
const float c_rmin = 0.004;
const float c_gmin = 0.001;
const float c_bmin = 0.002;

void main(void)
{
   vec4 result = vs_brightness * texture(u_sampler, vs_texcoord);
   result.x = (floor(result.x * c_shading_levels) / c_shading_levels) + c_rmin;
   result.y = (floor(result.y * c_shading_levels) / c_shading_levels) + c_gmin;
   result.z = (floor(result.z * c_shading_levels) / c_shading_levels) + c_bmin;

   o_colour = result;
}
```

Kod 5.21. *Fragment shader* za procesirane 3D modela s teksturom i osvjetljenjem

Fragment shader definiran u programskom isječku 5.22, dohvaća boju piksela za svaki fragment te je množi s difuznom komponentom svjetlosti koja je izračunata u *vertex shaderu*. Konačno se nad tom bojom vrši *cel shading*, što je tehnika kojom se smanjuje razina detalja.

Kod 5.22. Rezultat renderiranja

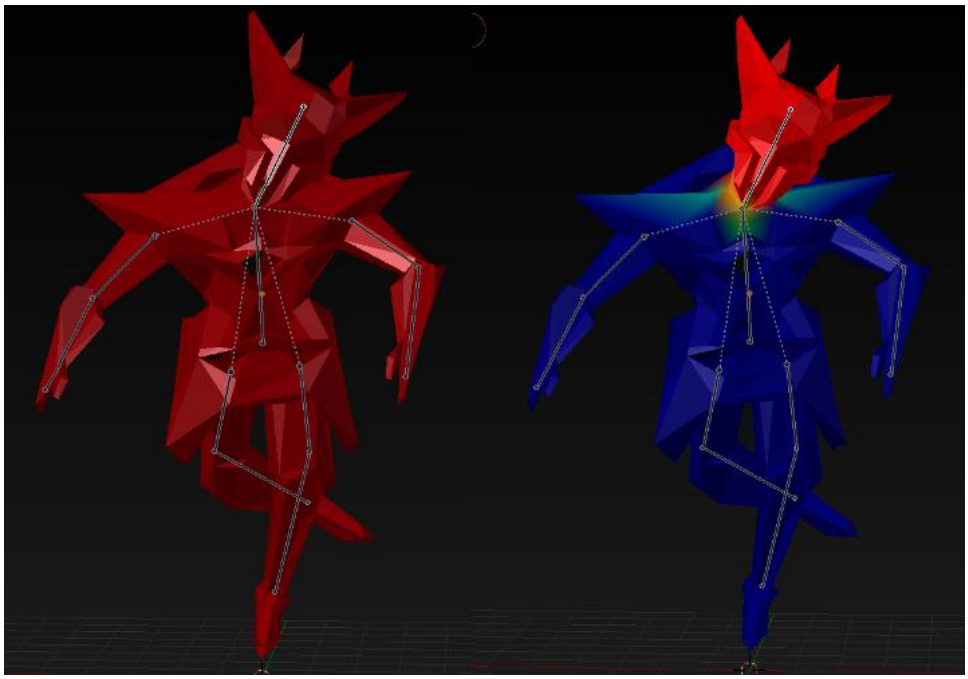
Bitno je napomenuti da se model u slici 5.23 nalazi u pozi, jer se isti model koristi za animacije.

5.5.11. Animacija 3D modela

Animacije su ključni dio igre jer pomoću njih se može ostvariti mnoštvo zadovoljavajućih efekata. Za potrebe ovog projekta koristi se skeletna animacija (engl. *skeletal animation*).

Skeletnu animaciju čini sustav kostiju koje su međusobno povezane. Veze između kostiju su tipa roditelj-dijete. Svaka se kost može transformirati, a dijete transformirane kosti se transformira skupa s roditeljskom kosti.

Svaka kost također transformira određeni set pozicija koje čine 3D model. Intenzitet utjecaja transformacije kosti na određenu poziciju se definira pomoću *vertex weight* vrijednosti. Vizualizacija utjecaja kosti može se vidjeti na slici 5.11.



Slika 5.11. Kostur i prikaz utjecaja kosti glave na okolne pozicije

Za učitavanje animacija koristi se Assimp biblioteka. Sve animacije se izvoze u Collada (.dae) formatu. Što je format koji se može koristiti za izvoz i uvoz čitavih scena. Da bi se uspješno učitao model kojeg je moguće animirati, potrebno je prvo učitati osnovne podatke o modelu kao što su pozicije, UV koordinate i normale. Nakon toga se učitavaju podaci o kostima i utjecaju kostiju na pozicije 3D modela, te se konstruira kostur koji ima jednu izvorišnu kost. Ovaj proces vrši klasa `AnimationLoader` u metodi `loadAnimatedComponent` iz programskog isječka 5.24.

```
public static final AnimatedModel loadAnimatedComponent
(String file, String rootNodeName)
{
    AIScene scene = Assimp.aiImportFile
    (
        file,
        aiProcess_Triangulate |
        aiProcess_FlipUVs |
        aiProcess_LimitBoneWeights
    );

    BaseModel model = loadAnimatedFile(file, rootNodeName, scene);
    SkeletalNode rootNode = getNodeHierarchy(scene, rootNodeName);
    AnimatedModel component = getAnimatedComponent(scene, model, rootNode);

    scene.free();

    return component;
}
```

Kod 5.23. Čitanje animacija pomoću Assimp biblioteke

`BaseModel` predstavlja osnovni model koji se može učitati iz Wavefront datoteke, s dodatkom indeksa kosti koje utječu na svaku poziciju. `SkeletalNode` predstavlja jednu kost koja skupa s drugim kostima čini kostur, a kombinacija kostura i osnovnog modela je klasa `AnimatedModel`.

U ovom projektu sve se Assimp klase potrebne za animiranje modela pretvaraju u zamjenske klase. Razlog ovome je činjenica da je potrebno otpustiti memoriju koju je Assimp rezervirao za pohranu informacija iz datoteke.

```
public class SkeletalNode
{
    public final String _NAME;
    public final int _CHILDREN_COUNT;
    public final Matrix4f _TRANSFORMATION;
    public final SkeletalNode[] _CHILDREN;

    private SkeletalNode(String name, Matrix4f transformation, int childCount)
    {
        _NAME = name;
        _TRANSFORMATION = transformation;
        _CHILDREN_COUNT = childCount;
        _CHILDREN = new SkeletalNode[childCount];
    }

    public static SkeletalNode fromAINode(AINode node)
    {
        SkeletalNode current = new SkeletalNode
        (
            node.mName().dataString(),
            Matrix4f.fromAINodeTransform(node.mTransformation()),
            node.mNumChildren()
        );

        for (int i = 0; i < current._CHILDREN_COUNT; i++)
        {
            current._CHILDREN[i] = fromAINode
            (
                AINode.create(node.mChildren().get(i))
            );
        }

        return current;
    }
}
```

Kod 5.24. Klasa SkeletalNode

Prva zamjenska klasa je klasa `SkeletalNode` iz programskog isječka 5.25, koja se koristi kao zamjena za `AINode` klasu. Jedan `SkeletalNode` predstavlja kost, te sadrži naziv, transformaciju i listu djece. Bitno je napomenuti da je za uspješno konstruiranje kostura potrebno metodi `fromAINode` proslijediti glavnu izvorišnu kost.

```
public class SkeletalVectorKey
{
    public final float _STAMP;
    public final Vector3f _VALUE;

    private SkeletalVectorKey(float stamp, Vector3f value)
    {
        _STAMP = stamp;
        _VALUE = value;
    }

    public static SkeletalVectorKey fromAIVectorKey(AIVectorKey key)
    {
        SkeletalVectorKey current = new SkeletalVectorKey
        (
            (float) key.mTime(),
            Vector3f.fromAIPositionKey(key.mValue())
        );

        return current;
    }
}
```

Kod 5.25. Klasa SkeletalVectorKey

Klasa `SkeletalVectorKey` iz programskog isječka 5.26 predstavlja vektorsku transformaciju u određenom vremenu, te služi kao zamjena za klasu `AIVectorKey`. Transformacije koje se spremaju u instance ove klase predstavljaju linearne pomake kostiju.

```
public class SkeletalQuaternionKey
{
    public final float _STAMP;
    public final Quaternion _VALUE;

    private SkeletalQuaternionKey(float stamp, Quaternion value)
    {
        _STAMP = stamp;
        _VALUE = value;
    }

    public static SkeletalQuaternionKey fromAIQuatKey(AIQuatKey key)
    {
        SkeletalQuaternionKey current = new SkeletalQuaternionKey
        (
            (float) key.mTime(),
            Quaternion.fromAIRotationKey(key.mValue())
        );

        return current;
    }
}
```

Kod 5.26. Klasa SkeletalQuaternionKey

Klasa `SkeletalQuaternionKey` iz programskog isječka 5.27 predstavlja rotaciju u određenom vremenu, te služi kao zamjena za klasu `AIQuatKey`.

```
public class SkeletalNodeAnim
{
    public final String _NODE_NAME;
    public final int _POSITION_COUNT;
    public final int _ROTATION_COUNT;
    public final SkeletalVectorKey[] _POSITION_KEYS;
    public final SkeletalQuaternionKey[] _ROTATION_KEYS;

    private SkeletalNodeAnim
    (String nodeName, int positionCount, int rotationCount)
    {
        _NODE_NAME = nodeName;
        _POSITION_COUNT = positionCount;
        _ROTATION_COUNT = rotationCount;
        _POSITION_KEYS = new SkeletalVectorKey[positionCount];
        _ROTATION_KEYS = new SkeletalQuaternionKey[rotationCount];
    }

    public static SkeletalNodeAnim fromAINodeAnim(AINodeAnim node)
    {
        SkeletalNodeAnim current = new SkeletalNodeAnim
        (
            node.mNodeName().dataString(),
            node.mNumPositionKeys(),
            node.mNumRotationKeys()
        );

        for (int i = 0; i < current._POSITION_COUNT; i++)
        {
            current._POSITION_KEYS[i] = SkeletalVectorKey.fromAIVectorKey
            (
                node.mPositionKeys().get(i)
            );
        }

        for (int i = 0; i < current._ROTATION_COUNT; i++)
        {
            current._ROTATION_KEYS[i] = SkeletalQuaternionKey.fromAIQuatKey
            (
                node.mRotationKeys().get(i)
            );
        }

        return current;
    }
}
```

Kod 5.27. Klasa SkeletalNodeAnim

Klasa `SkeletalNodeAnim` iz programskog isječka 5.28 predstavlja animaciju jedne kosti, te služi kao zamjena za klasu `AINodeAnim`. Animacija kosti sastoji se od niza pozicijskih i rotacijskih transformacija koje su pohranjene u instance klase `SekeltalVectorKey` i `SkeletalQuaternionKey`.

Ime kosti `_NODE_NAME` se koristi za referenciranje kosti koju je potrebno transformirati.

```
public class SkeletalAnimation
{
    public final float _DURATION;
    public final float _TICKS_PER_SECOND;
    public final int _NODE_ANIM_COUNT;
    public final SkeletalNodeAnim[] _NODE_ANIMATIONS;

    private SkeletalAnimation
    (float duration, float ticksPerSecond, int animationCount)
    {
        _DURATION = duration;
        _TICKS_PER_SECOND = ticksPerSecond;
        _NODE_ANIM_COUNT = animationCount;
        _NODE_ANIMATIONS = new SkeletalNodeAnim[animationCount];
    }

    public static SkeletalAnimation fromAIAnimation(AIAnimation anim)
    {
        SkeletalAnimation current = new SkeletalAnimation
        (
            (float) anim.mDuration(),
            (float) anim.mTicksPerSecond(),
            anim.mNumChannels()
        );

        for (int i = 0; i < current._NODE_ANIM_COUNT; i++)
        {
            current._NODE_ANIMATIONS[i] = SkeletalNodeAnim.fromAINodeAnim
            (
                AINodeAnim.create(anim.mChannels().get(i))
            );
        }

        return current;
    }
}
```

Kod 5.28. Klasa SkeletalAnimation

Klasa `SkeletalAnimation` iz programskog isječka 5.29 predstavlja jednu animaciju, te se koristi kao zamjena za `AIAnimation` klasu. Animacija modela je skup animacija kosti, odnosno niz transformacija kosti kroz vrijeme.

S ovim podacima animacija modela se vrši tako što se u danom trenutku rekurzivno prolazi kroz sve kosti te se ažurira stanje transformacijske matrice. Da bi se vršila transformacija potrebno je znati stanje prošle i sljedeće transformacije, te ovisno o vremenu interpolirati vrijednosti pozicije i rotacije.

Kada je izračunato interpolirano stanje, za svaku se kost stanje transformacijske matrice sprema u uniformo polje koje se šalje *shader* programu iz programskog isječka 5.30.

```
#version 330
#define MAX_BONES 16

layout (location = 0) in vec3 i_position;
layout (location = 1) in vec3 i_normal;
layout (location = 2) in vec2 i_texcoord;
layout (location = 3) in ivec4 i_bone_ids;
layout (location = 4) in vec4 i_bone_weights;

out vec2 vs_texcoord;
out float vs_brightness;

uniform mat4 u_model_view;
uniform mat4 u_model_view_projection;
uniform mat4 u_bone_transforms[MAX_BONES];
uniform vec3 u_light_position;

const float c_shading_levels = 8.0;

void main(void)
{
   mat4 bone_transform = u_bone_transforms[i_bone_ids.x] * i_bone_weights.x +
             u_bone_transforms[i_bone_ids.y] * i_bone_weights.y +
             u_bone_transforms[i_bone_ids.z] * i_bone_weights.z +
             u_bone_transforms[i_bone_ids.w] * i_bone_weights.w;

   vec4 animated_position = bone_transform * vec4(i_position, 1.0);
   vec4 animated_normal = bone_transform * vec4(i_normal, 1.0);
   vec3 animated_world_position = (animated_position * u_model_view).xyz;
   vec3 animated_world_normal =  (animated_normal * u_model_view).xyz;

   vec3 light_vector = normalize(u_light_position - animated_world_position);
   float diffuse = max(dot(animated_world_normal, light_vector), 0.04);

   vs_texcoord = i_texcoord;
   vs_brightness = diffuse;

   gl_Position = u_model_view_projection * animated_position;
}
```

Kod 5.29. *Vertex shader* za animirani model

Na svaku poziciju mogu djelovati maksimalno 4 različite kosti, ukupna vrijednost utjecaja svih kosti na određenu poziciju je jednaka 1.

Ukupna animacijska transformacija određene točke je jednaka sumi transformacija kostiju koje utječu na tu poziciju koje su pomnožene sa snagom utjecaja.

Nakon što je izračunata animacijska transformacija, potrebno je uvažiti transformaciju modela u prostoru. Te se finalna transformacija pozicije dobiva tako što se pomnoži animacijska transformacija s *model view* matricom, koja predstavlja kombinaciju transformaciju kamere i 3D modela u prostoru. Ista se kalkulacija provodi za normalu, jer bi u protivnom izračun osvjetljenja bio baziran na modelu koji se nalazi u inicijalnoj pozi.

5.6. Zvuk igre

Bitan aspekt svake igre je i zvuk. U ovom projektu zvuk se reproducira pomoću OpenAL biblioteke.

Glavni elementi OpenAL biblioteke su međuspremnici zvuka (engl. *sound buffers*), izvori zvuka (engl. *sound sources*) i slušatelji zvuka (engl. *sound listeners*).

Međuspremnici pohranjuju podatke o zvuku, ovdje je bitno napomenuti da OpenAL biblioteka radi s PCM (*Pulse Coded Modulation*) formatom, što ujedno znači da je svaki drugi tip ulazne zvučne datoteke potrebno pretvoriti u PCM.

Izvori zvuka predstavljaju poziciju u 3D prostoru koja je izvorište zvuka. Za potrebe ovog projekta svi zvučni efekti imaju izvor u točki (0, 0, 0), jer radi kuta pod kojim igrač promatra svijet nema previše smisla imati 3D zvuk.

I krajnji objekt su slušači, koji predstavljaju poziciju iz koje bi se određeni zvuk trebao čuti. Kao što je već spomenuto, ovaj projekt ne vrši izračune 3D zvuka. Klasa `SoundManager` sadržava sve metode za implementaciju zvuka. A učitavanje se vrši pomoću metode `createBuffer` definiranoj u programskom isječku 5.31.

```
public static void createBuffer(byte key, String oggFile)
{
    try (STBVorbisInfo vorbInfo = STBVorbisInfo.malloc())
    {
        int soundBuffer = alGenBuffers();
        ShortBuffer pcmData = readOgg(oggFile, SIZE, vorbInfo);

        int format;

        if (vorbInfo.channels() == 1)
            format = AL_FORMAT_MONO16;
        else
            format = AL_FORMAT_STEREO16;

        SOUNDS.put(key, soundBuffer);

        alBufferData
        (
            soundBuffer,
            format,
            pcmData,
            vorbInfo.sample_rate()
        );
    }
}
```

Kod 5.30. Učitavanje zvuka

Ova metoda uzima datoteku tipa "*.ogg*" te ju pretvara u PCM, radi zahtijeva OpenAL biblioteke. Nakon toga se provjerava broj kanala, iz čega se zna da li se radi o *mono* ili *stereo* zvuku. *Mono* zvuk koristi samo jedan zvučni kanal dok stereo koristi više. Programski isječak 5.32 definira metode koje se koriste za manipulaciju zvuka.

```
public static void initialize()
{
    alListener3f(AL_POSITION, 0, 0, 0);
}

public static void createSoundSource(byte key, boolean looping, int buffer)
{
    int soundSource = alGenSources();
    SOUND_SOURCES.put(key, soundSource);

    alSourcei(soundSource, AL_BUFFER, buffer);
    if (looping) alSourcei(soundSource, AL_LOOPING, AL_TRUE);
}

public static void gain(Byte key, float value)
{
    alSourcef(SOUND_SOURCES.get(key), AL_GAIN, value);
}

public static void playSound(Byte key)
{
    alSourcePlay(SOUND_SOURCES.get(key));
}

public static void stopSound(Byte key)
{
    alSourceStop(SOUND_SOURCES.get(key));
}
```

Kod 5.31. Metode za manipulaciju zvuka

Metoda `initialize` pozicionira *listenera* na lokaciju (0, 0, 0).

Da bi se mogao reproducirati zvuk potrebno je imati izvor, a za stvaranje izvora se koristi metoda `createSoundSource`. Metoda prima ključ pod kojim će se novi zvučni izvor zapamtiti, *flag* koji definira da li je zvuk potrebno ponavljati i ID *sound buffer* objekta u kojem se nalaze podatci zvuka.

Metoda `gain` se koristiti da bi se podesila glasnoća određenog izvora. Za pokretanje, odnosno zaustavljanje zvuka koriste se metode `playSound` i `stopSound`.

Pri zatvaranju aplikacije potrebno je pobrisati sve izvore pozivom metode `alDeleteSources`, te sve *buffer* objekte pozivom `alDeleteBuffers` metode.

Zaključak

Svaka tehnologija ima svoje prednosti i nedostatke.

U svijetu razvoja video igara vladaju razne platforme, među kojim su Unity i Unreal *engine*. Takvi napredni alati nude jako dobra gotova rješenja s aspekta tehnologije, te prepuštaju kreativni dio korisniku.

Naravno, to znači da korisnik nema potpunu kontrolu nad svim aspektima igre. To je cijena koja se plaća za brzinu razvoja koja se dobije korištenjem takvih rješenja.

Zatim postoji sasvim druga strana priče, a to je razvoj kompletnog engine sustava za igru koristeći se samo bibliotekama koje omogućavaju komunikaciju s hardwareom. U tom svijetu vlada C++ programski jezik, radi nepobjedivih performansi.

No rad s C++ programskim jezikom zna biti poprilično nezgodan, te je količina vremena koju je potrebno uložiti za razvoj *engine* sustava jako velika. To je cijena koja se plaća za potpunu kontrolu nad razvojem i mogućnost ostvarivanja najboljih performansi igre.

Negdje između ova dva primjera bi se mogao smjestiti razvoj igara pomoću LWJGL biblioteke.

Java je vrlo jednostavan programski jezik te je iz tog razloga razvoj Java programskih rješenja relativno brz. Java aplikacija može se pokretati na mnogo različitih platformi bez potrebe mijenjanja izvornog koda. Cijena te jednostavnosti i portabilnosti su performanse, što nije problem u većini slučajeva, no za razvoj igara to definira gornju granicu kompleksnosti koju igre napisane u Javi ne mogu preći.

S aspekta korištenja *nativnih* biblioteka razlike su vidljive jedino u sintaksi jezika, te nedostatku ključnih funkcionalnosti jezika kao što su *pointeri*. Svi bitni koncepti su isti radi činjenice da se koristi isti API.

Zaključno, razvoj igara pomoću LWJGL biblioteke je brži način razvoja kompletnog *engine* sustava, a glavni nedostatak predstavlja ograničenost Java platforme na kojoj se pokreće.

Popis kratica

LWJGL	*LightWeight Java Game Library*	Java biblioteka za razvoj igara
RAM	*Random Access Memory*	Memorija s nasumičnim pristupom
API	*Application Programming Interface*	Aplikacijsko programsko sučelje
AR	*Augmented Reality*	Proširena stvarnost
LLVM	*Low Level Virtual Machine*	Alat za razvoj kompajlera
JIT	*Just In Time*	Točno na vrijeme
GPU	*Graphics Processing Unit*	Grafički procesor
JNI	*Java Native Interface*	Java sučelje za izvorni kod
OpenGL	*Open Graphics Library*	Biblioteka za grafiku
GLFW	*Graphics Library FrameWork*	Okvir za razvoj grafike
OpenAL	*Open Audio Library*	Biblioteka za zvuk
Assimp	*Open Asset Import Library*	Biblioteka za uvoz raznih formata
FPS	*Frames Per Second*	Sličica u sekundi
GLSL	*OpenGL Shading Language*	Jezik za pisanje *shader* programa
VAO	*Vertex Array Object*	Mapa za spremanje atributa
VBO	*Vertex Buffer Object*	Polje za spremanje vrijednosti atributa
PNG	*Portable Network Graphics*	Slikovni format
PCM	*Pulse Coded Modulation*	Zvukovni format

Popis slika

Popis kôdova

Literatura

[1] ROBERT NYSTROM, Game Programming Patterns, Siječanj 2011

[2] Learn OpenGL, https://learnopengl.com/

[3] GLFW Documentation, https://www.glfw.org/docs/latest/

[4] LWJGL Overview, https://javadoc.lwjgl.org/overview-summary.html

ALGEBRA
VISOKO
UČILIŠTE

RAZVOJ IGRE S LWJGL BIBLIOTEKOM

Pristupnik: Filip Gaćina, 0321005012

Mentor: v. pred. Aleksander Radovan, dipl.
.