Математички факултет
Универзитет у Београду

Мастер рад

Архитектура система за имплементацију игара заснована на компонентама

студент: Урош Јовановић
ментор: проф. др. Владимир Филиповић

Београд, 2017

Ментор:

проф. др. Владимир Филиповић

Математички факултет

Универзитет у Београду

Чланови комисије:

проф. др. Саша Малков

Математички факултет

Универзитет у Београду

проф. др. Филип Марић

Математички факултет

Универзитет у Београду

Датум одбране:

_____

Садржај

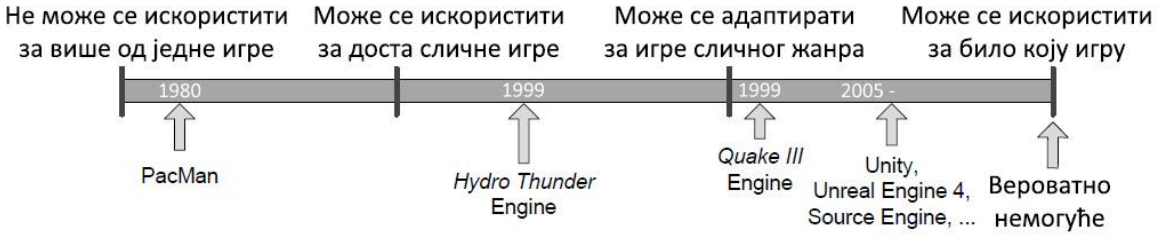
1. Увод

Систем за имплементацију игара (енг. *"Game Engine"*, у наставку скраћено СИИ) представља развојно окружење за израду игара које је производ вишеструких дисциплина из области рачунарства са јаким акцентом на флексибилност, проширивост, робусност и перформансе. Развојем технологија, оваква окружења су превазишла домене прављења игара и данас имају примену у разним областима попут комплексне визуализације, тренинга, медицинске или војне симулације у реалном времену. Нека од популарних развојних окружења данас укључују *Unreal*, *Unity* і *CryEngine*.

Предмет овог рада биће основни појмови и концепти система за имплементацију игара. Биће проучен и концепт програмирања заснованог на компонентама, предности и мане овог приступа у односу на приступ ослоњен на модел наслеђивања. Надаље, биће описани системи, алгоритми, структуре и пројектни узорци који се користе за развој система за имплементацију игара заснованог на компонентама.

2. Историја система за имплементацију игара

Видео игре (у наставку само *игре*), као врста дигиталне забаве, јавиле су се још раних 50-их година са појавом првих рачунара. То су биле јако једноставне интерактивне графичке апликације. У овом стадијуму разлика између игре и система за имплементацију игара је била скоро непостојећа. Употребна вредност кода који је написан за једну игру била је јако мала уколико би програмер хтео да направи неку другу игру. Међутим, како је брз развој хардвера и софтвера допринео богаћењу садржаја и обима игара, јављала се све већа потреба за општијим, поново употребљивим решењима. Термин *Game Engine* јавља се тек почетком 90-их година с појавом комплекснијих игара попут *Doom*-а (акциона игра пуцања из првог лица) који је развила компанија *id Software*. Ово је била једна од првих игара где су јасно биле изоловане кључне компоненте попут 3Д графичког и аудио подсистема као и подсистема за детекцију колизије између објеката. Оваква сепарација опште логике игре од њених специфичних детаља омогућила је лиценцирање технологије другим студијима као и развој „модинг“ заједнице (енг. “*modding community*”) која је једноставном заменом графичких и звучних ресурса могла да модификује или у потпуности измени постојећу игру.



Иако се може размишљати о СИИ као софтверу који, у општем случају, може да генерише и репродукује било коју игру, у пракси то често није случај. Различити жанрови игара доносе специфичне скупове проблема па већина СИИ-а се развија оријентисано ка конкретном жанру. Примери жанрова игара и техничких захтева које носе:

- Пуцање из првог лица (енг. “*First Person Shooters*”)
    - Ефикасан приказ великих отворених и затворених простора
    - Детаљни модели и анимације оружја и ликова
    - Аутентичан звук оружја и окружења
    - Брзе и поуздане реакције на контроле играча
    - Добар систем вештачке интелигенције за неигриве ликове
    - Могућност играња преко интернета за до 64 играча
    - Игре: *Quake, Doom, Unreal Tournament, Counter-Strike, Call of Duty*
    - СИИ: *Unreal Engine, Source Engine, IW Engine*

- Игре из трећег лица и платформске игре (енг. "*Third-Person Games, Platformers"*)
    - Покретни делови окружења (платформе, степенице, конопци итд.)
    - Загонетке као део окружења
    - Детаљни модели и анимације играча (у 3Д окружењу)
    - Камера из трећег лица која прати играча и контролише се путем улазних уређаја попут миша или џојстика. У 3Д окружењу је посебно изазовно реализовати овакву камеру тако да не пролази кроз моделе окружења и осталих објеката на сцени приликом ротације око играча.
    - Игре: *Super Mario, Tomb Raider, Gears of War*
    - СИИ: *Unreal Engine, Foundation*

- Борилачке игре (енг. "*Fighting Games"*)
    - Богат скуп борилачких анимација
    - Прецизна детекција колизије између играча
    - Прецизан систем контрола који је у могућности да детектује вишеструке комбинације тастера
    - Активна публика око ринга
    - Модерне борилачке игре често имплементирају и анимације одеће бораца базиране на физици као и специјалне ефекте попут зноја
    - Игре: *Mortal Kombat, Street Fighter, Tekken, Fight Night*
    - СИИ: *Unreal Engine, Enforcer Engine*

- Тркачке игре (енг. "*Racing Games"*)
    - Детаљни модели и анимације возила
    - Кретање возила базирано на физици
    - Разни трикови како би се окружење приказивало што ефикасније јер се највећи графички акценат ставља на возила и стазе
    - Стазе се углавном деле на дводимензионе регионе који се називају и „секције". Овакве структуре података омогућавају лакше детектовање видљивих елемената на сцени, побољшавају вештачку интелигенцију неигривих ликова при примени алгоритама проналаска најбржег пута и генерално решавају многе друге техничке проблеме
    - Игре: *Need for Speed, Gran Turismo*

- СИИ: *GT, Frostbite*

- Стратегије у реалном времену (енг. "*Real Time Strategies*")
  - Већи акценат на броју елемената на сцени него на квалитету модела
  - Удаљена камера која обухвата већи део сцене
  - Дизајн нивоа игра велику улогу у квалитету игре
  - Динамичке измене нивоа (играч често прави нове базе, јединице итд.)
  - Интеракција са играчем се заснива на кликовима по мапи и менијима који су увек доступни на екрану
  - Модерне стратегије омогућавају апсолутну контролу камере што додатно компликује логику исцртавања модела и окружења
  - Игре: *Starcraft, Command & Conquer, Age of Empires, Warhammer*
  - СИИ: *SAGE*

- Масивне онлајн игре (енг. "*Massive Online Games – MMOG*")
  - Подршка за стотине хиљада играча путем интернета
  - Комбинација са било којим другим жанром
  - Игре: *World of Warcraft*
  - СИИ: обично развијен интерно, од стране произвођача (енг. "*in-house*")

3. Постојећи системи за имплементацију игара

3.1. Quake

Првом правом 3Д игром пуцања из првог лица често се сматра *Castle Wolfenstein 3D* (1992). Направљена од стране *id Software*, ова игра је усмерила индустрију у новом и узбудљивом правцу. *Id Software* је даље наставио са играма попут *Doom, Quake, Quake II* и *Quake III*. Сви СИИ иза ових наслова су доста слични по архитектури и генерално се називају *Quake* фамилијом СИИ. Ова технологија се на даље користила у многим играма и многи модерни СИИ имају корене дубоко у *Quake* фамилији (нпр. *Valve*-ов *Source Engine*). Изворни код *Quake* и *Quake 2* игре написан у програмском језику C, је данас јавно доступан и, иако доста застарео, пружа одличан увид у то како функционишу професионални СИИ [1].

3.2. Unreal

Компанија *Epic Games* остварила је свој пробој на сцену са игром *Unreal* 1998. Од тог момента *Unreal Engine* је постао озбиљан конкурент *Quake*-у у жанру игара пуцања из првог лица. *Unreal Engine* 2 и 3 (2002-2004) донели су велик број нових функционалности и побољшања. Иако не савршен, овај СИИ је изузетно адаптиван и конфигурабилан јер се (уз одређени тип лиценце) доставља заједно са комплетним изворним кодом што омогућава програмерима да из њега извуку максимум за специфичну апликацију. Неприкосновен списак функционалности као и интуитиван графички едитор и добар комерцијални план, утврдио је место овог СИИ у индустрији. Велики број корисника условио је и развој велике онлајн заједнице као и обимне и детаљне документације. Године 2012, објављена је четврта итерација СИИ-а, која је донела бројна побољшања на пољу графичких могућности, корисничке интеракције и продуктивности. Данас, *Unreal* технологија је лиценцирана и коришћена у многим играма различитих жанрова, у филмској индустрији као и у различитим симулацијама забавног карактера или у сврхе тренинга. Поред велике популарности, *Epic Games* је опет затресао играчку индустрију када је 2015. године објављено да ће *Unreal Engine* 4 бити доступан бесплатно уз комплетан изворни код [2].

3.3. CryEngine

Развијен од стране немачке компаније *Crytek*, *CryEngine* је био главни алат иза игре *Far Cry* (2004). Од тог момента, *CryEngine* је непрекидно у развоју и коришћен је за бројне игре (углавном пуцање из првог лица) укључујући популаран серијал *Crysis*. Оно што издваја *CryEngine* од конкуренције је технологија, специфично, његов графички подсистем који спада у сам врхунац у индустрији и нуди скоро фотореалистичан приказ уз добро искоришћење хардвера, перформансе и

скалабилност. Током 2016, у маниру осталих професионалних СИИ, *CryEngine* је постао доступан свима по „Плати колико желиш“ (енг. “*Pay what you want”*) систему уз јавно доступан изворни код [3].

3.4. Unity

За разлику од већине заступљених СИИ, *Unity* није настао из потребе за прављењем специфичне игре већ је од самог зачетка конципиран као генерално софтверско решење за генерисање интерактивних симулација. Прва верзија објављена је јуна 2005. године међутим тек 2012. је *Unity* доживео велики скок у популарности и прихваћености од стране индустрије (поготово од стране независних студија) и од тада је интересовање у константном порасту. Оно што издваја *Unity* од конкуренције је изузетно интуитиван едитор, могућност кодирања на више програмских језика укључујући и C#, могућност развоја на *Windows* и *OSX* оперативним системима као и могућност генерисања за сваку релевантну платформу. Додатно, *Unity* је СИИ базиран на компонентама, а као такав је од великог интереса за овај рад и у наставку ће многе паралеле бити повлачене управо са овим системом [4].

3.5. Остали

Поред набројаних, постоји још доста бесплатних и комерцијално доступних СИИ, као и многи приватни, интерно развијени алати који студији користе унутар својих компанија. Неки од њих су:

- *Source* – Приватно власништво, *Valve*
- *Frostbite* – Приватно власништво, DICE EA
- *Ogre3D* – Отворен код
- *Irrlicht* – Отворен код

4. Преглед архитектуре и основни концепти

4.1. Дијаграм архитектуре

4.2. Слојеви архитектуре

- *Хардвер*. Представља циљни персонални рачунар, мобилни уређај или играчку конзолу на којој ће се игра покретати. Многе савремене игре подржавају више платформи, што представља додатни изазов при развоју.
- *Драјвери / ОС*. Софтвер ниског нивоа који представља „заштитни слој“ између компоненти СИИ и самог хардвера. Игра је обично само један од процеса којима управља оперативни систем и као таква комуникацију са хардвером обавља искључиво путем механизама пружених од стране ОС, односно драјвера.
- *Софтвер трећих лица (енг. “3rd party software”).* Већина СИИ се ослања на неки вид већ готових софтверских решења. Типичан пример су графичке библиотеке попут *OpenGL* / *Direct3D* или библиотеке за физичке прорачуне попут *nVidia PhysX.*

- *Базни подсистеми.* У ову категорију спадају основни модули попут математичких библиотека, система за управљање меморијом, структура података, алгоритама итд.
- *Управљање ресурсима.* Сваки СИИ има неки вид управљача ресурсима (енг. *"Resource Manager"*). Ресурси представљају било који тип улазног податка за СИИ, попут аудио датотеке, слике, текстура, 3Д модела, текста итд. Неки СИИ имају комплексне, централизоване управљаче реурсима док други, једноставнији, тај посао делегирају програмерима игре.
- *Подсистеми вишег нивоа.* Подсистеми попут логике, графике или физике су изграђени над претходно наведеним слојевима. Овим радом биће описани неки од најчешће имплементираних подсистема вишег нивоа, уз практичне примере имплементације у компонентно базираним СИИ.

4.3. Главна петља

Приликом извршавања игре, подсистеми вишег нивоа морају бити обрађивани у одређеним временским интервалима и у одређеном редоследу. Најједноставнији вид такве обраде је ажурирање свих подсистема игре у једној *главној петљи*.

```
while (!playerInput.quit)
{
        updateGameLogic();
        updatePhysics();
        updateAnimation();

        render();
}
```

Међутим, у реалним имплементацијама, подсистеми се обично ажурирају у различитим временским интервалима. На пример, подсистем анимације захтева ажурирање 30 или 60 пута у секунди и мора бити у синхронизацији за петљом ажурирања графичког подсистема. С друге стране, подсистем за физичке прорачуне је потребно ажурирати свега десетак пута у секунди. Подсистеми попут вештачке интелигенције можда могу бити ажурирани само једном у секунди, не нужно у синхронизацији са графичким подсистемом итд.

4.4. Мека симулација у реалном времену

У рачунарским наукама, игре се често описују као *меке симулације у реалном времену* (енг. "*soft real-time simulation*"). Терминологија потиче од чињенице да грешке апроксимације и прекорачења одређених временских интервала који су наметнути не резултују катастрофалним исходима. Прескакање једног циклуса физичког прорачуна или лош одзив и брзина ажурирања у игри ће можда збунити или иритирати играча,

али дефинитивно неће угрозити његов живот. Супротно томе, погрешни или закаснели прорачуни у *чврстим симулацијама у реалном времену*, попут система у авијатици или нуклеарним електранама могу имати фаталне последице.

Симулирање реалног (или измишљеног) света (или делова света) се у играма реализује одређеним математичким апроксимацијама. Паметним коришћењем апркоксимација и упрошћавања, развијалац игре може доћи до резултата који се практично не могу разликовати од реалног.

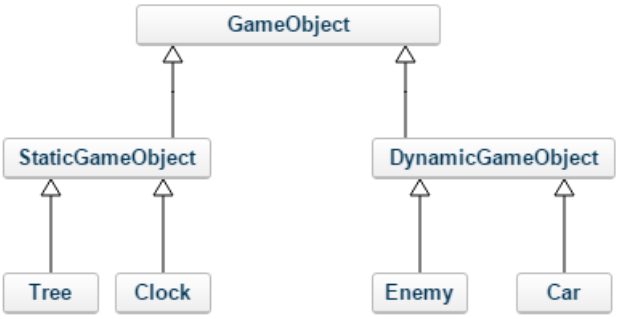
Игре се такође класификују и као *интерактивне привремене симулације* јер се ток симулације не одвија континуално већ зависи од уноса корисника и догађаја који се одвијају у свету игре.

5. Логички подсистем

У срцу сваког СИИ налази се подсистем који коме је делегирана логика игре. Логика игре најпре описује стање и понашање учесника (објеката, елемената) игре, било да њима управља играч или рачунар. Стање учесника могу бити информације попут количине преостале енергије играча, док понашање може бити начин на који играч реагује након што га непријатељ погоди. Са техничке стране, логички подсистем треба да омогући механизме помоћу којих развијаоци игара (не нужно програмери) могу лако додавати, ажурирати или уклањати типове објеката у игри. Додатно, потребно је реализовати начине дефинисања понашања објеката (обично скриптама) који су интуитивни и не захтевају познавање целокупног СИИ. Поновна употребљивост логике једне игре у некој другој у многоме зависи од нивоа модуларности и флексибилности логичког подсистема. У наставку ће бити изложено неколико примарних начина имплементације, уз акценат на компонентно базирани приступ.

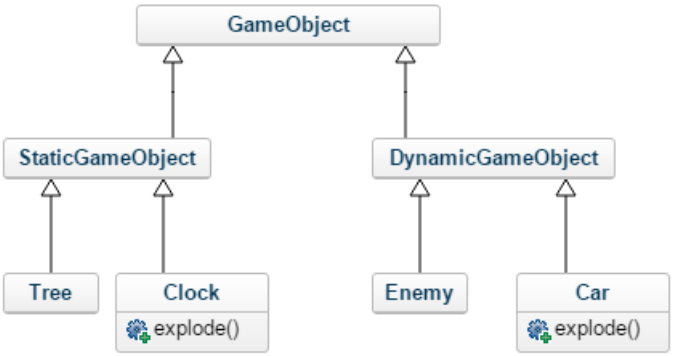
5.1. Модел наслеђивања и практични проблеми

При учењу или развоју једноставних игара многи програмери ће започети код са неким видом базног типа из којег ће даље бити изведени сви типови у игри. Овакав приступ је интуитиван и природан свакоме ко познаје основе објектно оријентисаног програмирања и био је доста заступљен при развоју игара 90-их. Међутим, сложеније игре захтевају сложеније хијерархије које су неугодне за мењање, одржавање и разумевање. Концепт ће бити размотрен на следећем примеру.



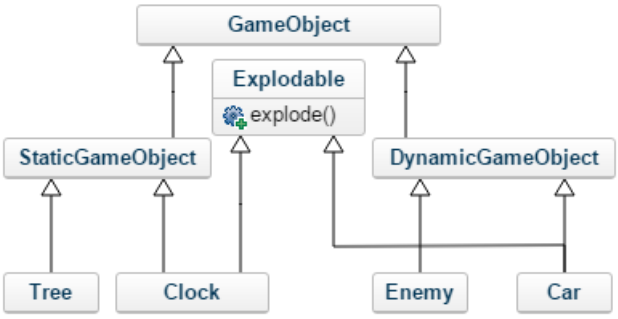
Слика 1 - Пример хијерархије типова у игри

Може се догодити да дизајнери игре одлуче да *Car* и *Clock* могу да експлодирају. Програмер дефинише методу *explode()* коју ова два типа треба да имплементирају. Једна варијанта је да оба типа имају одвојене имплементације тог метода. Из очигледних разлога, овакво решење није оптимално.



Слика 2 – Метода *explode()* је имплементирана у обе класе што доводи до дуплираног кода

Друга варијанта је да типови *Car* и *Clock* наследе методу из типа *Explodable*. Ово решење је ограничено на језике који подржавају вишеструко наслеђивање или интерфејсе. Додатно наслеђивање даље компликује хирерархију и прегледност јер, да би неко разумео тип *Car*, мора да буде упознат са типовима *Explodable* и *DynamicGameObject*.



Слика 3 – Метода *explode()* је апстрахована у базну класу

Надаље, нека су тестови показали да опција *explode()* уводи занимљиву динамику у игру и дизајнери сада одлучују да та опција треба да постоји свуда. Програмер покушава да „подигне“ методу у хијерархији и стиже до базног типа *GameObject*. Међутим, можда није прихватљиво нити смислено да сви типови (тренутни или будући) имају функционалност *explode()*. Тест играчи пријављују грешку јер могу да униште објекат типа *Bed*, који је једино место где напредак игре може да се сачува. Ово је могуће средити појединим „прљавим“ програмерским триковима попут *превазилажења* (енг. “*overriding*”) методе празном имплементацијом у језицима где је тако нешто подржано.

Додатно, оваква структура није превише модуларна. Како сваки тип дефинише своје понашање, логика игре меша се са логиком подсистема самог СИИ. Примера ради, типови могу имплементирати методу *draw()* која исцртава објекат на екрану. Таква метода мора носити логику за исцртавање која је део графичког подсистема. Оваква спрегнутост онемогућава да типови који представљају логику игре буду у потпуности издвојени у слој података, већ се морају налазити заједно са подсистемима

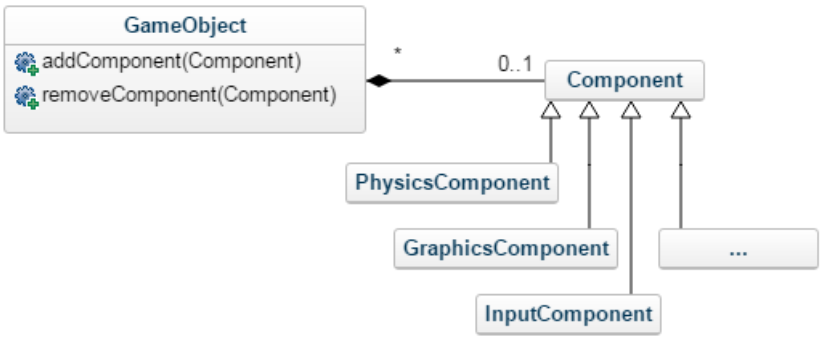
од којих зависе. То значи да свака измена, додавање или уклањање типова захтева поновно (у језицима где је то неопходно) превођење изворног кода система и такве измене може радити искључиво софтверски инжењер који је упознат са целокупним системом.

Важно је напоменути да хијерархијска структура типова није нужно лоше решење, напротив, за једноставне игре је и препоручива јер у таквим играма хијерархије типова обично нису дубоке, нити у њиховом развоју учествује велики број људи.

- Добре стране система базираног на наслеђивању:
    - Једноставно за разумевање;
    - Може се развити брзо од нуле;
    - Статичка типизираност;
    - Низак утицај на перформансе
- Лоше стране система базираног на наслеђивању:
    - Недовољно флексибилно, хијерархију типова је тешко променити након што је формирана;
    - Тешко за одржавање, поготово ако је хијерархија дубока;
    - Објекти често имају понашања која су им непотребна или немају смисла

5.2. Систем базиран на компонентама

У овом систему, хијерархија типова је сажета у један базни објекат игре који агрегира компоненте. Компонента дефинише понашање објекта. Како понашање више није непосредан део објекта, он постаје доста флексибилнији, једноставнији и лакши за одржавање. Систем заснован на компонентама омогућава дефинисање нових типова објеката у игри једноставним додавањем / одузимањем компоненти (понашања). Динамичка природа компоненти, омогућује да се оне дефинишу у слоју података и могу се развијати и одржавати независно од осталих подсистема те тај посао није више нужно везан за програмера у строгом смислу.



Слика 4 – Пример класног дијаграма система компоненти

Битна ствар при разумевању компонентно базираног система је да се више не ради у објектно оријентисаној парадигми. Објекти игре више нису дефинисани наслеђивањем, већ је њихов тип дефинисан искључиво компонентама које садрже. Такви типови су често описани датотекама (које садрже кôд компоненти) и инстанцирају се користећи неки вид метода пројектног узорка *фабрика* (енг. *"factory"*).

Како се објекат састоји од једне или више компоненти које дефинишу понашање, поставља се питање где се чувају својства тј. стање објекта? Својства се могу чувати у самом објекту или у компонентама објекта. Оба решења имају своје предности и мане али у оба случаја носилац својстава мора имати неки вид интерфејса ка њима.

Уколико се својства чувају у објекту, она морају имати своје *приступне методе* (енг. *"getters / setters"*). Компоненте могу да приступе својствима једино преко објекта тако да референца ка објекту којем припадају мора постојати унутар самих компоненти. Оваква референца је свакако потребна због уопштене комуникације међу компонентама. Због динамичке природе система компоненти, објекат не зна унапред која ће сва својства имати. То може довести до колизије у именима уколико две различите компоненте захтевају исто својство али различитог типа. Проблеми ове природе се, у зависности од програмског језика, обично решавају одговарајућим структурама података.

Уколико се својства чувају у компонентама, тада компонента мора омогућити приступне методе за својства. Да би компонента приступила својствима објекта, она мора знати која компонента чува та својства. Мана оваквог приступа је спрегнутост међу компонентама која отежава одржавање кода као и чињеница да компоненте морају да претпоставе да су неке друге компоненте присутне што не мора увек бити тачно због динамичке природе додавања / одузимања.

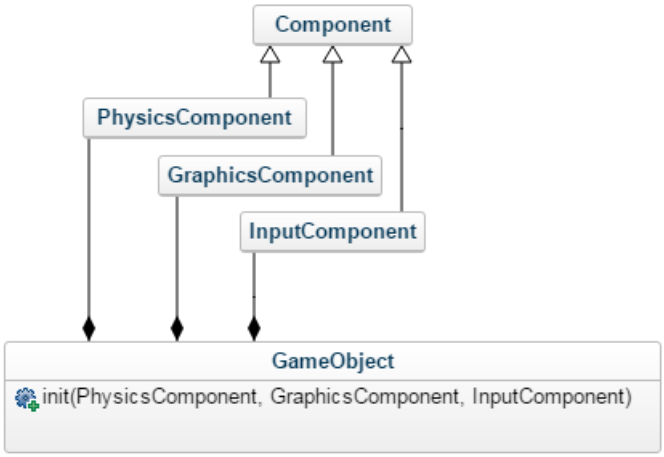
- Добре стране система заснованог на компонентама:
  - Скалабилност по броју различитих типова објеката и броју различитих компоненти (понашања) придружених тим објектима.
  - Типови се могу дефинисати у слоју података.

- Модуларност. Компоненте се могу искористити у више игара.
- Могућност прављења библиотеке компоненти, што може драстично убрзати развој будућих игара.

- Лоше стране система заснованог на компонентама:
  - Прављење оваквог система захтева доста времена и труда.
  - Динамичка типизираност.
  - Компоненте су лаке за одржавање, али било каква измена у API-у (енг. "*Application Programming Interface*") није флексибилна јер би утицала на све компоненте.

5.3. Хибридни приступ

Хибридни приступ представља компромис између два претходно описана решења. Уместо листе компоненти, објекат садржи статички дефинисане интерфејсе ка апстрактним компонентама, а конкретна понашања су изведена и достављају се објекту путем конструктора. Различити типови се праве наслеђивањем или директно преко конструктора.



Слика 5 - Пример класног дијаграма хибридног приступа

Проблем оваквог решења је што се губи доста флексибилности у односу на „чисто" компонентни приступ, јер постоји само пар предефинисаних типова понашања која се могу мењати. Без обзира на то, предефинисани типови понашања су дати интерфејсом те се ипак могу конфигурисати динамички што је флексибилније него чисто хијерархијски приступ. Овакво решење је добро за игре мале до умерене величине, где постоји план за поновно коришћење направљених компоненти.

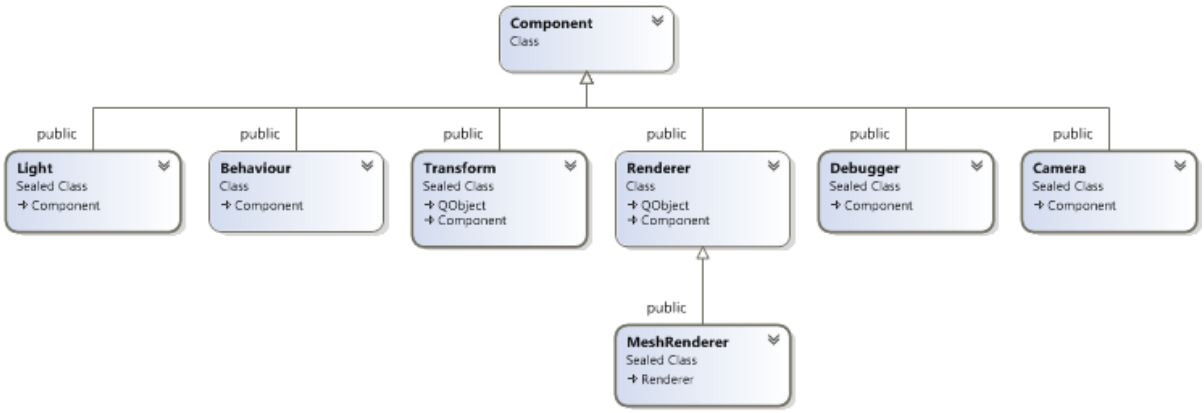
- Добре стране хибридног система:
  - Статичка типизираност.
  - Могућност промене појединих типова понашања.
  - Флексибилност компоненти без комплексности имплементирања компонентно базираног система.
- Лоше стране хибридног система:
  - Слично као и за хијерархијски приступ.

5.4. Имплементација

У наставку ће бити представљени поједностављени (у сврхе читљивости) примери имплементације компонентног система из СИИ који представља део овог рада. Кôд је написан у програмском језику C++, али се сам концепт може имплементирати помоћу било ког објектно-оријентисаног програмског језика.

5.4.1. Класа *Component*

Класа *Component* представља апстрактну базну класу свих компоненти у СИИ и имплементира, претходно описан, приступ базиран на компонентама.



Слика 6 - Дијаграм хијарархије типова компоненти у имплементираном СИИ

```
class Component /* Bazna klasa iz koje nasleđuju sve komponente */
{
private:
	GameObject* _gameObject; /* Pokazivač na vlasnika komponente */

protected:
	explicit Component(GameObject* gameObject);
	virtual ~Component();

public:
	static void destroy(Component* component);
	GameObject* gameObject() const;
};
```

Behaviour је кључна компонента из које даље изводе све компоненте које дефинишу специфична *понашања* објекта за конкретну игру. При ажурирању логике, главна петља игре позива методу *update()* над сваком компонентом *Behaviour* која је тренутно активна на сцени у сврхе напредовања симулације. Важно је напоменути да објекат игре није у обавези да поседује ову нити било коју другу компоненту.

```
class Behaviour : public Component
{
protected:
	explicit Behaviour(GameObject* gameObject);
	virtual ~Behaviour() override;

public:
	virtual void startUp();
	virtual void update();
	virtual void shutDown();
};
```

5.4.2. Класа *GameObject*

Поједностављена верзија класе *GameObject* из имплементираног СИИ дата је следећим примером:

```
class GameObject
{
private:
	QString _name;
	QList<Component*> _components;

protected:
	~GameObject();

public:
	static void destroy(GameObject* gameObject);

	GameObject();
	GameObject(const QString& name);

	const QString& getName() const;
	void setName(const QString& name);

	const QList<Component*> getComponents() const;
	const Component* getComponent(const QString& name) const;
	const void addComponent(Component* component) const;
	const void removeComponent(const QString& name) const;
}
```

У овој, наизглед једноставној, имплементацији постоји пар кључних проблема. Првенствено, метода *getComponent()* враћа показивач на базну класу, што захтева експлицитну конверзију типа (енг. "*explicit cast*") и провере у време извршавања које могу бити небезбедне и компликују код. Додатно, упит о компоненти по њеном имену може бити спор због поређења стрингова и захтева неки вид мапирања (име, компонента) што повећава величину објекта у меморији. Коначно, класа *GameObject* није запечаћена (енг. "*sealed*"), што, супротно ономе што се жели постићи, охрабрује програмера да прави нове типове наслеђивањем. Побољшана верзија класе *GameObject* дата је следећим примером:

```
class GameObject final
{
private:
	QString _name;
	QList<Component*> _components;

protected:
	~GameObject();

public:
	static void destroy(GameObject* gameObject);

	GameObject();
	GameObject(const QString& name);

	const QString& getName() const;
	void setName(const QString& name);

	const QList<Component*> getComponents() const;
	template <typename T> T* getComponent() const;
	template <typename T> T* addComponent() const;
	template <typename T> void removeComponent() const;
}
```

Метода *getComponent()* је сада *шаблон функција* (енг. *"function template"*) без аргумената. Недостатак аргумента оправдан је експлицитним навођењем типа компоненте који се жели дохватити док се истовремено избегава накнадна експлицитна конверзија типа повратне вредности. Међутим, у тренутку писања рада, C++ стандард не подржава никакве рестрикције над типом T, те он може бити било који валидан тип укључујући и уграђене типове попут *int* или *double*. Овакво понашање је могуће елиминистати проверама типа у време превођења (статичким проверама) [5]:

```
template <typename T>
T* getComponent() const
{
	/*
	   Statička provera tipa T.
	   Ukoliko tip nije nasleđen iz Component klase,
	   prevođenje se obustavlja.
	*/
	static_assert(std::is_base_of<Component, T>::value,
	"Tip mora biti izveden iz Component klase!");

	for (auto component : getComponents())
		if (auto t = dynamic_cast<T*>(component))
			return t;
	return nullptr;
}
```

Аналогно се могу имплементирати и методе *addComponent()* и *removeComponent()*, а уколико је дозвољено да један објекат може имати више компоненти истог типа метода *getComponent()* се може модификовати да враћа колекцију показивача. Имати у виду да оператор *dynamic_cast* може бити спор, те у одређеним ситуацијама, често дохватање компоненти може имати негативан утицај на перформансе. Додатни механизми, попут *кеширања* (енг. *"caching"*) података, могу убрзати овакве операције уз додатни меморијски трошак.

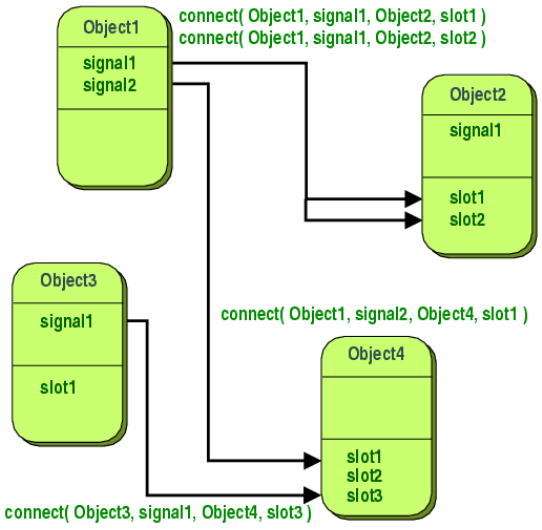
5.4.3. Комуникација

Често је потребно да компоненте комуницирају између себе или са системом. На пример, компонента може бити заинтересована за догађаје попут колизије, померања или промене својстава објекта. Уобичајени начин комуникације је путем *порука* (енг. *"messaging"*). Компонента може емитовати или примити поруку.

```
/*
  Igrač je udario u zid.
  Šalje se poruka o događaju.
*/
message("Collision");

void onMessage(const QString& msg)
{
	if (msg == "Collision")
	{
		/*Reakcija na koliziju...*/
	}
}
```

Конкретна имплементација у раду се ослања на систем *сигнала* и *слотова* као део Qt библиотеке [6]. Сигнал се емитује при догађају од интереса, попут промене својства објекта. Слотови су функције које се позивају као одговор на специфичан сигнал. Систем сигнала и слотова је реализован посебним механизмом библиотеке под називом *"Meta-Object System - MOS"* [7]. MOS генерише додатне (енг. "*meta*") информације приликом превођења типова који наслеђују класу *QObject*.



Слика 7 - Qt слотови и сигнали

```
Component* GameObject::addComponent(Component* component)
{
	/*
		Ukoliko je komponenta tipa Transform,
		želimo da budemo obavešteni kada se ona promeni.
	*/
	if (auto transform = dynamic_cast<Transform*>(component))
		connect(transform, SIGNAL(changed(Transform*)),
			this, SLOT(onTransformChanged(Transform*)));
	_components.push_back(component);
	/*
		Obaveštavamo sve zainteresovane
		da je nova komponenta dodata.
	*/
	emit componentAdded(this, component);

	return component;
}
```

Слика 8 - Исечак кода из имплементације методе *addComponent()*

Размена порука између компоненти производи додатни утрошак времена (у већ доста ограниченом временском интервалу за ажурирање логике), те је треба максимално ограничити.

5.4.4. Ажурирање

Како би се реакције играча и опште стање симулације пропагирали кроз систем, потребно је ажурирати логику игре у сваком циклусу главне петље пре него што графички подсистем генерише слику тренутне сцене на екрану. Поједностављена петља система у имплементираном СИИ дата је следећим кодом:

```
bool Viewport::event(QEvent* e)
{
	/* Od interesa je samo zahtev za ponovno iscrtavanje */
	if (e->type() == QEvent::UpdateRequest)
	{
		double elapsedSeconds = updateTimer.nsecsElapsed() / 10e9;
		updateTimer.restart();

		/* Ažuriranje logike igre */
		if (auto project = ProjectManager::instance()->activeProject())
			if (auto scene = project->getActiveScene())
				scene->update(elapsedSeconds);

		/* Iscrtavanje na ekran */
		render();

		/* Zahtev za ponovno iscrtavanje kako bi se petlja nastavila */
		QCoreApplication::postEvent(this, new QEvent(QEvent::UpdateRequest));

		return true;
	}
	/* Ostali događaji se propagiraju baznoj klasi QWindow */
	return QWindow::event(e);
}
```

Слика 9 - Klasa *Viewport* opisuje grafičku kontrolu u kojoj se igra iscrtava i prikazuje igraču [8]

Ажурирање понашања објеката врши се кроз ажурирање сцене позивом методе *update()*. Даље, сцена позива методе *update()* над свим компонентама *Behaviour* својих објеката (уколико их објекат садржи). Ради убрзања и једноставности итерације, компоненте понашања организоване су одвојено од колекције објеката којима припадају.

```
void Scene::update(double deltaTime) const
{
	for (const auto& behaviour : _behaviours)
		behaviour->update(deltaTime);
}
```

Како једна компонента може приступити било ком објекту или компоненти на сцени, може се десити да се позивом ажурирања једне компоненте стање сцене драстично измени. На пример, позив методе ажурирања објекта карактера играча утиче на то да пет непријатељских објеката буде уништено (а самим тим и све њихове компоненте) док уједно ствара нови објекат који представља награду за играча. С тога је потребно обратити додатну пажњу јер колекција кроз коју се итерира потенцијално може да се промени током итерације.

Аргумент *deltaTime* представља протекло време од последњег позива методе *update()*. Овај податак је од велике важности уколико се жели елиминисати ефекат „убрзања“ игре у зависности од јачине хардвера на којој се игра извршава. Примера ради, потребно је да се лопта у игри помера константном брзином у произвољном правцу. Једноставна компонента која би реализовала такво понашање дата је следећим кодом:

```
void BallBehaviour::update(double deltaTime)
{
	QVector3D position = gameObject()->transform()->getPosition();
	gameObject()->transform()->setPosition(position + QVector3D(1, 0, 0));
}
```

Уколико се ажурирање извршава на сваких 10 милисекунди, лопта ће се померати 100 јединица у правцу позитивне X осе сваке секунде. Међутим, на споријој машини где се ажурирање, због недостатка ресурса, не може позивати чешће од 20 милисекунди, иста лопта ће се у једној секунди померити само 50 јединица. Додатно, време између два ажурирања може доста да варира и на истом хардверу у зависности од расположивих ресурса те понашање лопте може бити још више непредвидиво. Решење овог проблема је скалирање помераја са *deltaTime* вредношћу. Уколико је време ажурирања брзо, вредност *deltaTime* ће бити мања, те ће се лопта померати за краће растојање. Уколико је време ажурирања споро, вредност *deltaTime* ће бити пропорционално већа, те ће се лопта померати за дуже растојање и брзина кретања ће остати иста невезано од брзине ажурирања.

```
gameObject()->transform()->setPosition(position + QVector3D(1, 0, 0) * deltaTime);
```

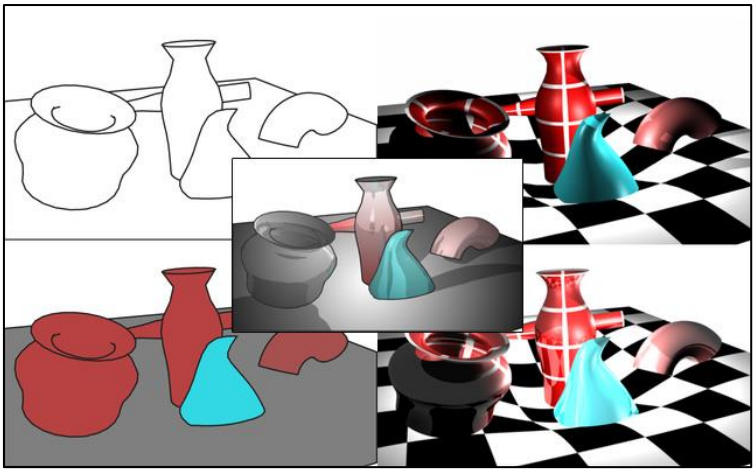
6. Графички подсистем

Прва асоцијација на појам видео игре обично јесу импресивни визуелни ефекти и анимације. Добра графика доприноси бољем доживљају игре и омогућава играчу да се боље „утопи“ у виртуелни свет. Исцртавање у реалном времену (енг. “*Real-time rendering*”) је комплексна и широка област рачунарства и детаљно излагање на поменуту тему било би ван обима овог рада. У овом поглављу ће бити описане основе рачунарске графике на мало апстрактнијем нивоу као и пример употребе компоненти у графичком подсистему.

Исцртавање у реалном времену у суштини се своди на следеће кораке:

- Описати *виртуелну сцену*, обично неким 3Д површинама задатим у математичкој форми.
- Позиционирати и ротирати *виртуелну камеру* како би се добио жељени поглед. Камера је обично замишљена као идеална *фокална тачка* у простору испред које се налази замишљена правоугаона површ која прима светлост, обично на јако малој удаљености. Улога камере је, слично као у реалном свету, да конвертује регистровану светлост у нешто што се може интерпретирати на уређају за приказ.
- Дефинисати један или више *виртуелних извора светлости*. Ови извори пружају светлосне зраке који се одбијају од 3Д површи и окружења на сцени, а које сензор виртуелне камере региструје.
- Описати *визуелна својства* површина на сцени. Ово дефинише на који начин ће светлост реаговати са површином.
- За сваки пиксел обухваћен правоугаоном површи камере, израчунати боју и интензитет тог пиксела на основу светлосних зрака који конвергирају ка фокалној тачки камере кроз тај пиксел. Овај поступак је познат и као решавање проблема сенчења (енг. “*shading problem*”).

Многе технологије се могу користити за извршавање претходно описаних корака. Циљ је обично неки вид *фотореализма*, али неке игре циљају и на друге стилове, попут стила цртаног филма (енг. “*cartoon*”) или стила водених боја (енг. “*watercolor*”). Када је стил дефинисан, софтверски инжењери и уметници се максимално труде да тај стил буде и испоштован подешавајући визуелна својства и оптимизујући конкретан алгоритам сенчења. Технологије такође варирају и у зависности од тога какав ниво фотореализма се жели постићи. Напредни алгоритми сенчења, са скоро фотореалистичним моделима осветљења обично се не могу приказати у реалном времену, док друге, апроксимативне методе могу, али на уштрб визуелног квалитета.



Слика 10 - Пример различитих визуелних стилова једне виртуелне сцене

Графички подсистем би требало да буде у стању да описане кораке извршава 30, 50 или чак 60 пута у секунди. То значи да је максимално време за извршавање 33.3 милисекунде (у случају 30 слика у секунди). У реалним условима, ово време је доста краће, с обзиром да у задатом временском оквиру СИИ мора да изврши и остале послове, укључујући и вештачку интелигенцију, динамичку детекцију колизије или комплексне анимације карактера. Поређења ради, у филмској индустрији, једна рачунарски генерисана слика може захтевати од неколико минута до неколико сати, тако да графички подсистеми модерних СИИ свакако представљају импресивно достигнуће.

6.1. Сцена

Сцена се, у контексту реалног света, састоји од објеката у простору. Неки објекти су провидни (стакло), а неки не (картонска кутија), док неки могу бити и делимично провидни (енг. "*translucent*"). За разлику од потпуно провидних, где светлост пролази скоро неометано, делимично провидни објекти имају таква својства да се зраци преламају пролазећи кроз њих што као резултат даје замућену слику онога што се налази иза објекта из угла посматрача. Гледано из контекста виртуелне сцене, за приказ непровидних објеката, интересантна је само њихова површина али не и објекти које заклањају. У случају провидних и делимично провидних објеката, потребне су додатне информације о унутрашњој структури и својствима објекта како би се израчунало одбијање светлости. Многи графички подсистеми не врше тако комплексне рачуне. За симулирање провидности, користи се једноставно нумеричко својство звано *алфа* (енг. "*alpha*"), које означава колико је нека површ видљива (обично вредност од 0 до 1). Ово може узроковати поједине аномалије у приказу ако би се објекти који су иза провидног објекта исцртали после њега, али постоје начини да се такве ситуације умање тако да добијени резултати буду довољно добри. Исцртавање објеката попут ватре, дима или облака своди се на исцртавање честица (енг. "*particles*") које су у

суштини мале правоугаоне површине одређене провидности. Стога, слободно се може рећи да се графички подсистем у СИИ махом бави исцртавањем површи.

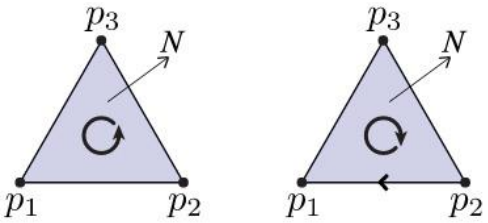
6.1.1. Троуглови

Апроксимација објеката површима, традиционално се врши повезаном мрежом троуглова (енг. "*triangle mesh*") која се моделује у специјализованим 3Д програмима попут *3D Studio Max, Blender* или *Z-Brush*.

Зашто троуглови?

- Троугао је најједноставнији полигон.
- Троугао је увек у равни (планаран).
- Троугао остаје троугао и након примене већине афиних трансформација или пројекција.
- Скоро сав модеран графички хардвер је направљен да ефикасно растеризује троуглове.

У рачунарској графици, троугао је описан са три тачке – темена троугла (енг. "*vertex*") $\boldsymbol{p_1}$, $\boldsymbol{p_2}$, $\boldsymbol{p_3}$ и нормалом на раван у коме троугао лежи $\boldsymbol{N}$. При моделовању објеката, посебну пажњу треба обратити на оријентисаност троуглова. Графички API-ји нижег нивоа (*OpenGL*, *Direct3D*) имају могућност дефинисања која страна троугла се сматра предњом односно задњом, на основу редоследа тачака (у смеру или супротно смеру казаљке на сату). Битно је да сви модели на сцени буду конзистентни по том питању, у супротном може доћи до аномалија при исцртавању. Поменути API-ји такође омогућују и искључивање исцртавања полеђине троуглова што може допринети перформансама и побољшати приказ провидних елемената.



Слика 11 - Различите оријентације троугла дефинисаног тачкама p1, p2, p3 и нормалом N

Координате тачака троуглова модела обично су изражене релативно неком локалном координатном систему који се означава као простор модела (енг. "*model space*"). Он може бити дефинисан у односу на центроид модела или неку другу тачку. Битно је да локални координатни систем буде конзистентан за све моделе у игри. Додатно, један геометријски модел се може искористити за више објеката у игри који су на различитим позицијама, ротирани или скалирани. На пример, модел једне уличне светиљке може бити постављен на сваких десет метара дуж пута. Овај поступак зове се инстанцирање (енг. "*instancing*") а геометрија која се користи више пута назива се инстанцирана геометрија.

6.1.2. Матрице

Једна од најкоришћенијих структура у рачунарској графици јесу *матрице*. Матрице су дводимензиони, *nxm* низови елемената. У контексту развоја игара, матрице нуде практичан начин представљања линеарних трансформација попут ротације, скалирања или транслирања. Овакве матрице се зову *матрице трансформације* [9]. Множењем две матрице трансформације добија се нова матрица која носи информацију о *обе* трансформације, тако да се низ операција ротација, транслирања и скалирања увек може, коначно, чувати у једној матрици. С обзиром да множење матрица није *комутативно*, редослед извршавања трансформација је битан. Ротација вектора у 3Д простору за угао $\boldsymbol{\varphi}$ описана је 3x3 матрицом скалара:

$$\begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Међутим, транслирање вектора у 3Д простору за вредности $\boldsymbol{t_x, t_y, t_z}$ захтева 4x4 матрицу скалара,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

с тога се матрице трансформације обично памте у овом формату. Како би се множењем матрице добио транслирани вектор потребно је проширити полазни вектор $\boldsymbol{r}$ још једном координатом $\boldsymbol{w = 1}$. За овакав вектор се каже да је записан у *хомогеним координатама*.

$$r + t = [r_x \quad r_y \quad r_z \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

6.1.3. Позиционирање

Да би се модел коректно позиционирао у виртуелној сцени односно простору игре или *свету* игре (енг. "*world space*"), мора бити дефинисана *матрица пресликавања* из простора модела у простор света:

$$M_{M\to W} = \begin{bmatrix} (RS)_{M\to W} & 0 \\ t_W & 1 \end{bmatrix}$$

где $(RS)_{M\to W}$ означава 3x3 матрицу ротације и скалирања из простора модела у простор света а t_W означава вектор транслације изражен у координатама простора света. Уколико су јединични локални базни вектори модела изражени у координатама простора света, матрица се може записати као:

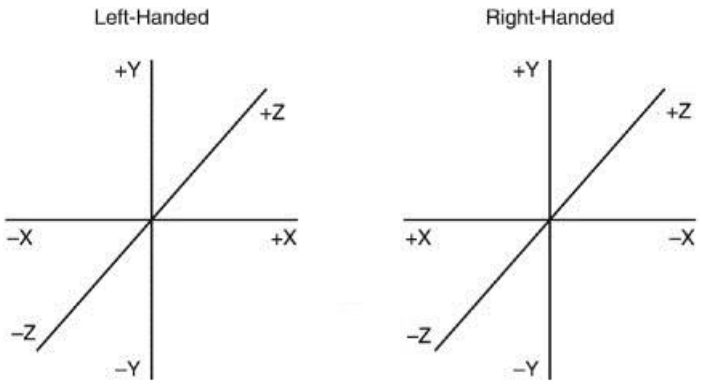
$$M_{M\to W} = \begin{bmatrix} i_m & 0 \\ j_m & 0 \\ k_m & 0 \\ t_W & 1 \end{bmatrix}$$

где су i_m, j_m и k_m јединични локални базни вектори. Формула по којој графички подсистем рачуна крајњи положај је:

$$v_w = v_m M_{M\to W}$$

Поједини стационарни објекти, чија је геометрија јединствена и не инстанцира се, могу олакшати поступак графичком подсистему јер никакве трансформације нису потребне у том случају.

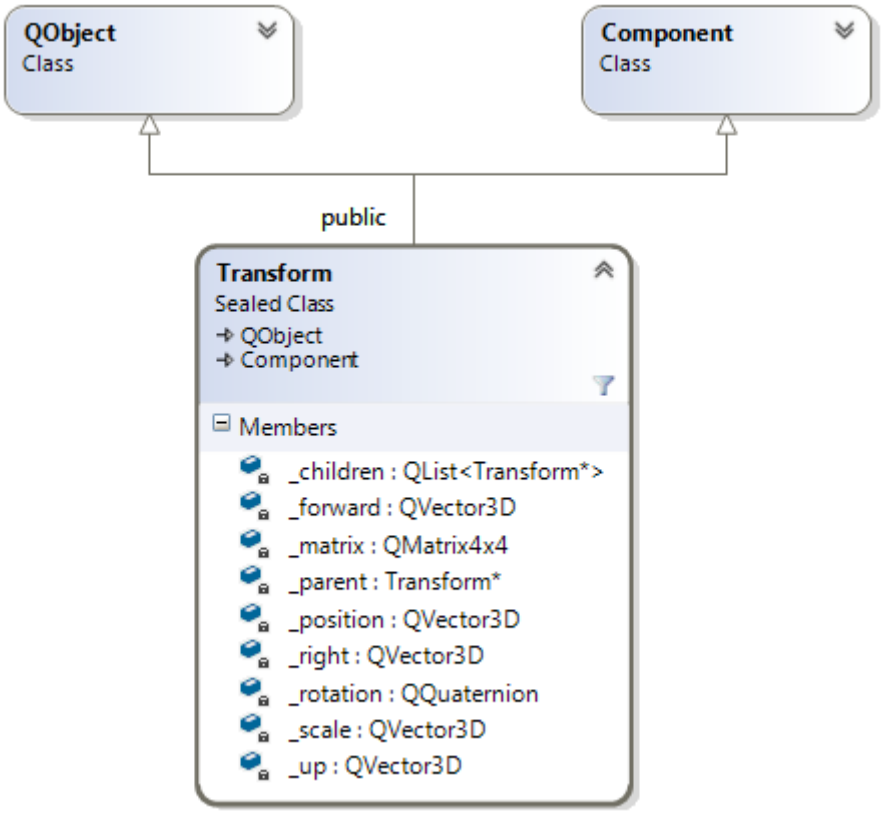
Координатни систем простора света у играма обично је дефинисан тако да Y оса буде „горе“. Остаје још да се дефинише која оса је „лево“ (или „десно“) а која „право“. Координатни систем леве руке (енг. “*left handed*”) има за „лево“ негативну X осу док је „право“ позитивна Z оса, док координатни систем десне руке (енг. “*right handed*”) има позитивну X осу за „лево“. У првом случају позитивна ротација је дефинисана у смеру казаљке на сату око осе ротације, док је у другом смер супротно од казаљке на сату.

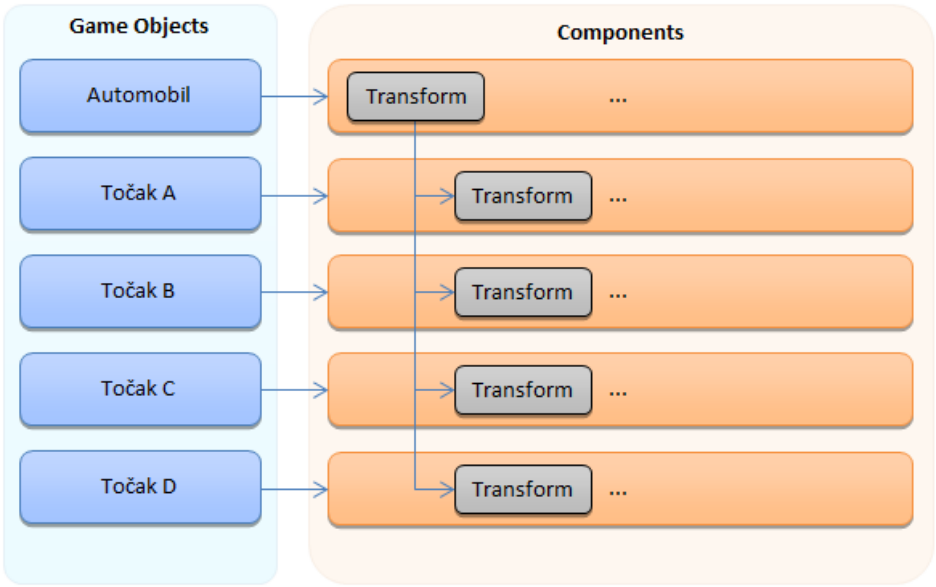


Слика 12 - Координатни систем леве и десне руке

6.1.4. Класа *Transform*

Као што је објашњено у претходном поглављу, у систему базираном на компонентама објекат игре не садржи никакву логику и његово понашање и стање су дефинисани скупом компоненти које садржи. Међу најбитнијим својствима објекта свакако јесу његова позиција, ротација и скалирање у простору. У имплементацији СИИ-а као дела овог рада, та својства су смештена у компоненту *Transform* која је дизајнирана по узору на истоимену компоненту из *Unity* СИИ. Поред набројаних својстава, ова компонента омогућава и увезивање више објеката у хијерархију уланчавањем *Transform*-а. Тако да, на пример, *точак* може бити дете-објекат *аутомобила*. Точак тада има локална својства која су релативна родитељу, као и глобална својства која су релативна простору света.





Како *Transform* не садржи геометрију, да би се објекат приказао на сцени потребно му је додати посебну компоненту типа *Renderer*. Ова компонента садржи све што је потребно графичком подсистему за цртање, између осталог и референцу ка геометрији која дефинише графичку репрезентацију објекта. Сва геометрија је центрирана у координатни почетак простора света, те се трансформација из простора

модела своди на множење матрицом трансформације самог објекта која се може добити из *Transform* компоненте.

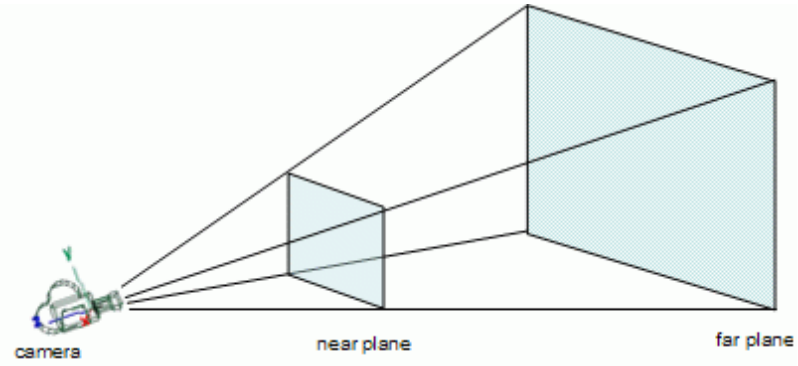
```
void MeshRenderer::render()
{
	auto renderManager = RenderingManager::instance();
	auto transform = gameObject()->transform()->getMatrix();

	/* Aktivacija matrice transformacije */
	renderManager->pushTransform(transform);
	/* Postavljanje materijala objekta */
	renderManager->bindMaterial(getConstMaterial());
	/* Iscrtavanje */
	renderManager->draw(_mesh);
	/* Deaktivacija matrice transformacije */
	renderManager->popTransform();
}
```

6.2. Камера

6.2.1. Простор погледа

Виртуелна камера представља доста упрошћенију верзију реалне камере или људског ока. Дефинисана је идеалном фокалном тачком испред које на малој раздаљини „лебди“ правоугаона површ која опонаша сензор светлости. Камера дефинише свој локални координатни систем означен као *простор погледа* (енг. “*view space*”). Центар овог координатног система представља управо фокалну тачку камере, док су осе дефинисане тако да „право“ увек указује у смеру погледа камере.



Слика 13 - Виртуелна камера

Позиција и ротација камере у простору дефинисани су матрицом пресликавања из простора погледа у простор света на исти начин као што се модели пресликавају из простора модела. Како би се мрежа троуглова исцртала, потребно је из простора света пресликати тачке у простор погледа. Ово се може добити инвертовањем матрице из простора погледа у простор света:

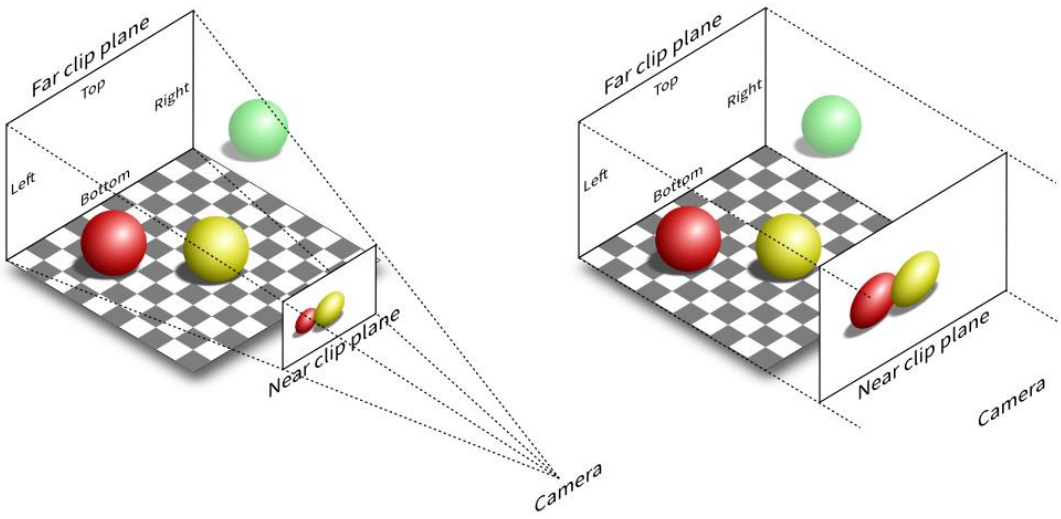
$$M_{W \to V} = (M_{V \to W})^{-1} = M_{view}$$

Пресликавање из модела у поглед се обично прерачунава унапред како графички подсистем не би морао да врши множење свих матрица при сваком исцртавању. Овако добијена матрица се у *OpenGL* API-у зове модел-поглед (енг. "*model-view*") матрица:

$$M_{M\to V} = M_{M\to V} M_{W\to V} = M_{model-view}$$

6.2.2. Пројекција

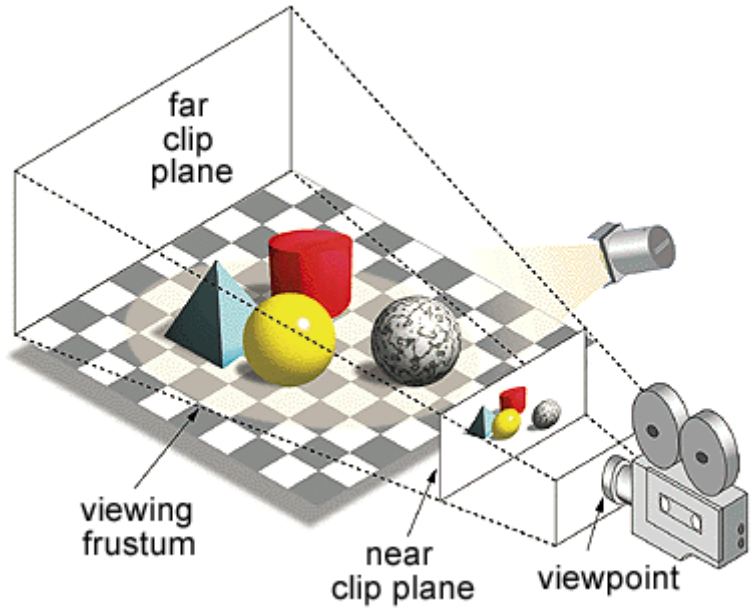
Како би се описана 3Д сцена исцртала на 2Д површ, користи се специјалан тип трансформације под називом *пројекција*. Постоје разни типови пројекција, али једна од најкоришћенијих је *перспективна*. Перспективна пројекција опонаша како реална камера и/или људско око доживљава објекте у простору тако што им се величина смањује у зависности од тога колико су одаљени од камере. *Ортогонална* пројекција, са друге стране, чува дужине и користи се углавном за планске погледе (са врха, са страна) а у контексту игара често се користи за 2Д мените и приказ информација преко 3Д сцене (енг. *"heads up display" – HUD*).



Слика 14 - Перспективна (лево) и ортогонална (десно) пројекција

6.2.3. Опсег погледа

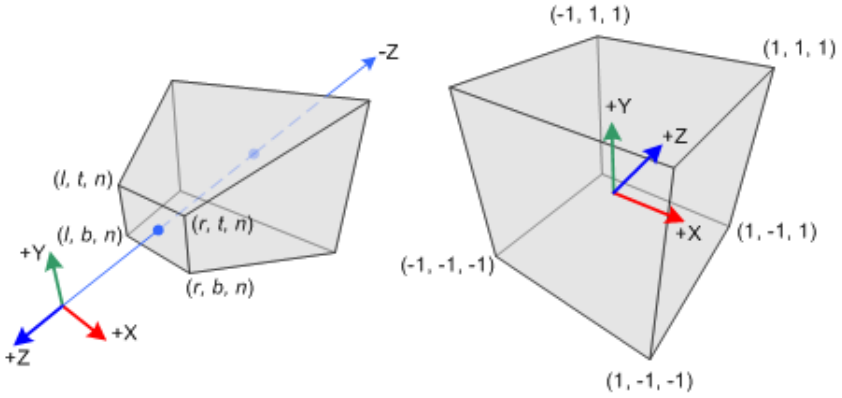
Регион простора који камера може да „види" назива се опсег погледа или *фрустум* (енг. "*view volume, frustum*"). Опсег погледа је дефинисан са шест равни. Предња раван представља претходно поменуту површ испред фокалне тачке (енг. "*near plane*"), четири равни около представљају ивице виртуелног екрана. Шеста раван (енг. "*far plane*"), служи као оптимизација да се објекти много удаљени од камере одсецају (енг. "*clipping*") при исцртавању. Додатно, пружа горњу границу за дубину пиксела о чему ће бити речи мало касније. При коришћењу перспективне пројекције, облик запремине погледа је зарубљена пирамида док је квадар облик добијен ортогоналном пројекцијом.



Слика 15 – Опсег погледа (фрустум) је дефинисан са шест равни

6.2.4. Простор одсецања

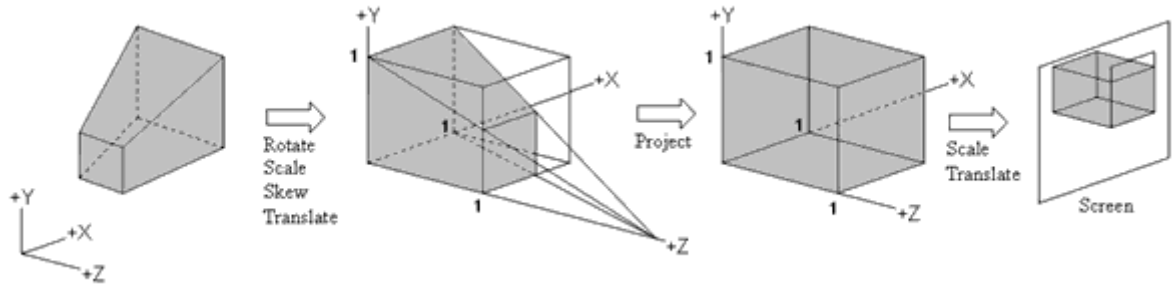
Након примене пројекције, тачке из простора погледа трансформишу се у нови простор под називом *хомогени простор одсецања* (енг. "*homogenous clip space*"). Овај тродимензиони простор је само изобличена верзија простора погледа. Сврха простора одсецања је да претвори простор погледа у јединствену запремину која је независна од врсте коришћене пројекције, као и од аспекта екрана и резолуције у којој ће се сцена исцртавати. Опсег погледа у простору одсецања је квадар који се по X, Y и Z осама пружа од -1 до 1. Изузетак постоји у *DirectX* API-у где се по Z оси квадар пружа од 0 до 1. Име „простор одсецања" долази од чињенице да су равни запремине погледа поравнате са осама (енг. "*axis aligned*"), те је јако једноставно одсећи (енг. "*clip*") троуглове који су ван региона.



Слика 16 - Простор погледа (лево) и хомогени простор одсецања (десно)

Сада се, коначно, може дати опис простора екрана. Простор екрана је дводимензиони координатни систем где су дужине изражене у пикселима а осе су

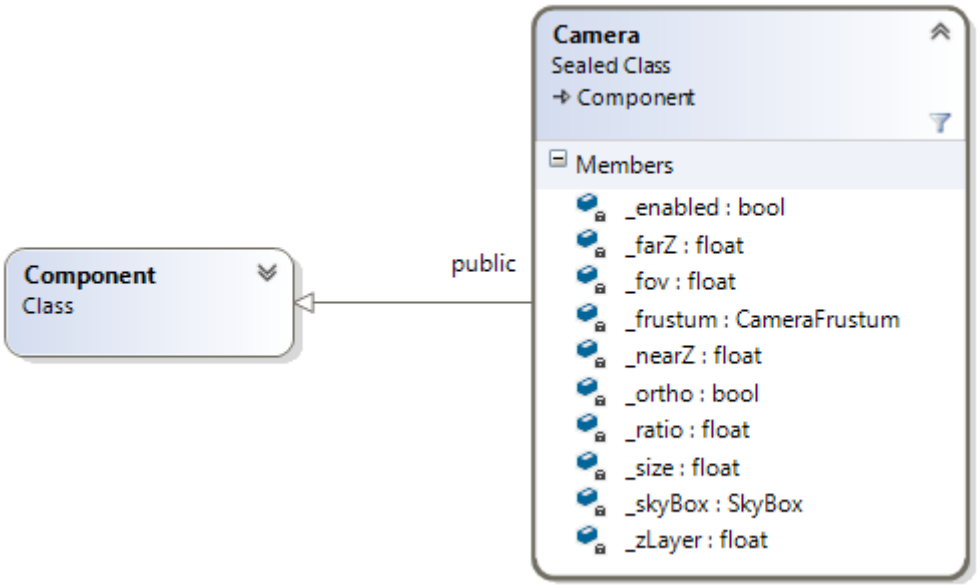
усмерене тако да се X оса пружа са леве на десну страну, док се Y пружа из горњег левог угла на доле. Разлог за обрнуту Y осу је историјско-техничке природе јер су CRT монитори исцртавали слику одозго на доле. Однос ширине и висине екрана назива се аспект екрана (енг. "*aspect ratio*"). Најкоришћенији аспекти данас јесу 4:3 (традиционална телевизија) и 16:9 (филмови). Троуглови из простора одсецања се исцртавају у простору екрана једноставним померањем и скалирањем из јединичног квадра по X и Y оси (Z се занемари). Овај процес је познат и као мапирање екрана (енг. "*screen mapping*").



Слика 17 - Процес мапирања сцене на екран

6.2.5. Класа *Camera*

Из претходног је јасно да су за одређивање камере потребне информације попут позиције, равни запремине погледа и матрице пројекције. При развоју једноставнијих СИИ, интуитивно би било имплементирати камеру као засебан тип објекта, евентуално делити део логике који се тиче позиционирања и ротације са објектом игре. У компонентно базираном систему, камеру треба посматрати као ништа друго до нови скуп својстава и понашања која се могу описати компонентом – *Camera*. Позиције и ротације нема потребе чувати додатно јер се о томе већ стара *Transform* компонента. Својство *position* дефинише фокалну тачку док својство *forward* усмерава камеру у жељеном правцу. Равни које дефинишу опсег погледа се обично не наводе експлицитно већ се *near* и *far* задају позитивним бројевима који одређују удаљености равни од фокалне тачке дуж смера гледања камере. Ове бројеве треба пажљиво изабрати јер превелики распони или постављање *near* равни преблизу фокалне тачке могу изазвати аномалије при исцртавању. Додатни параметри за запремину погледа су зависни од типа матрице пројекције. Перспективна пројекција захтева аспект екрана и угао гледања (енг. "*field of view – FOV*"), док ортогонална пројекција захтева дефинисање четири ивице екрана. Када су обезбеђени сви подаци, они се достављају графичком API-у како би се дефинисао поглед.



Слика 18 - Класни дијаграм *Camera* компоненте из имплементације

Додатна оптимизација при ажурирању и исцртавању сцене се може остварити ако се све компоненте *Camera* накачене на објекте чувају у одвојеном низу ради бржег приступа.

6.3. Осветљење

6.3.1. Особине светлости

Видљиво светло је део спектра електромагнетног зрачења, што значи да истовремено испољава особине таласа и честица. Боја светлости одређена је њеним *интензитетом* I и *таласном дужином* λ (или *фреквенцијом* $f = v / \lambda$, где је v брзина светлости кроз одређену средину). Таласне дужине видљиве светлости су у опсегу од 780nm до 380nm, односно фреквенција од 400THz до 790THz. Светлост може имати разне комплексне интеракције са материјом. Њено понашање је дефинисано делом медијумом кроз који путује, а делом и површи које, технички речено, представљају интерфејс између два типа медијума (вода и ваздух). Упркос комплексности, светлост заправо може да дејствује на четири начина:

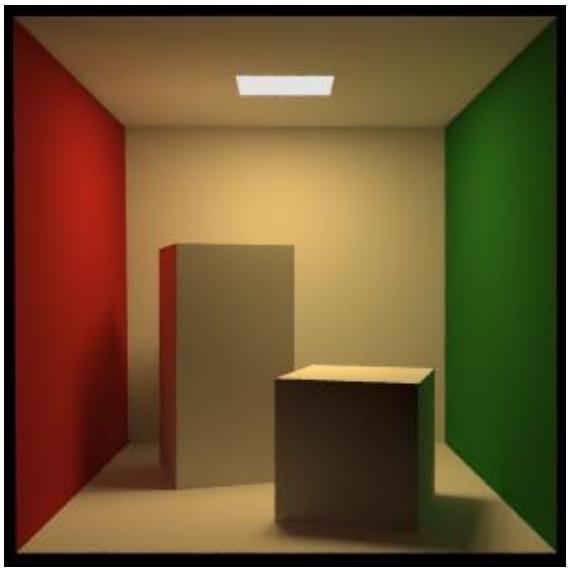
- Може бити *апсорбована*.
- Може бити *рефлектована*.
- Може бити пропуштена кроз неки објекат, где може доћи до *рефракције* (кривљења).
- Може бити пропуштена кроз јако уске отворе, где може доћи до *дифракције*.

Већина СИИ који теже фотореализму имплементира прва три понашања. Дифракција светлости обично нема видљив утицај на сцену те се занемарује у рачуну и

у исцртавању. Апсорпција светлости од стране објеката је оно што нам даје перцепцију боје. На пример, ако површина објекта апсорбује све таласне дужине осим 550nm, објекат ће се приказати као зелен. Потпуна апсорпција приказује објекат као црн, док потпуно рефлектовање даје утисак беле боје. Рефлектовање светлости може бити *дифузно* (енг. "*diffuse*"), тј. униформно у свим правцима, *одразно* (енг. "*specular*") уколико је одбијена светлост концентрисана у уски сноп или *анизотропно* (енг. "*anisotropic*") уколико начин одбијања светлости зависи од угла гледања. Рефракција представља закривљење светлости при проласку кроз неки објекат које зависи од материјала унутрашњости објекта као и од конкретне таласне дужине светлости. Класичан пример ове појаве је пропуштање светлости кроз троугаону призму где се, при проласку, светлост разлаже на компоненте различитих таласних дужине (боје дуге).

6.3.2. Модел осветљења

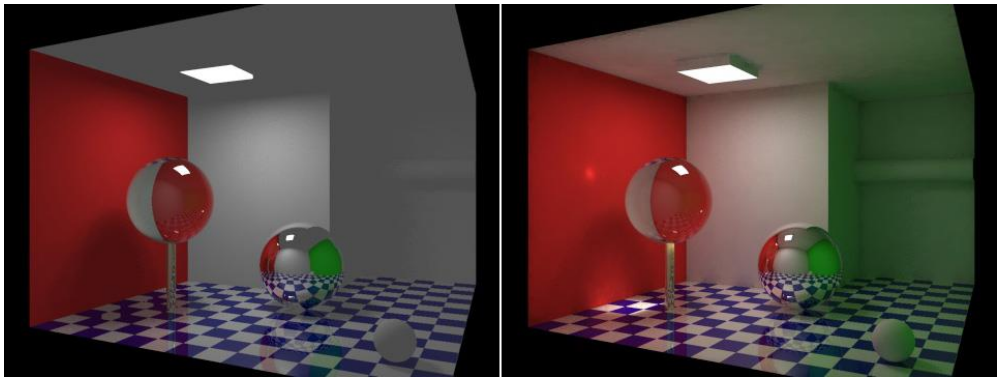
Добар модел осветљења је кључан за сваки иоле озбиљан графички подсистем. Без њега ће и најдетаљнији 3Д модели и материјали изгледати једнолично и вештачки. Супротно томе, и најједноставнија сцена може да изгледа фотореалистично ако је добро осветљена.



Слика 19 - Корнелова кутија (на слици) је уобичајени тест веродостојности рачунарски генерисане слике поређењем са реалном фотографијом исте сцене [10]

Постоје бројни математички модели за израчунавање интеракције светлосних извора са површинама и атмосфером под називом *транспортни модели светлости* (енг. "*light transport model*") или једноставно – модели осветљења. Најједноставнији вид модела осветљења је такозвани *локални* модел (енг. "*local illumination*"). При рачуну осветљености неког објекта, локални модел узима само утицај директних извора светлости (често и само оних који су најближи). Логично, овакве технике су прве коришћене при исцртавању у реалном времену. *Глобални* модели (енг. "*global*

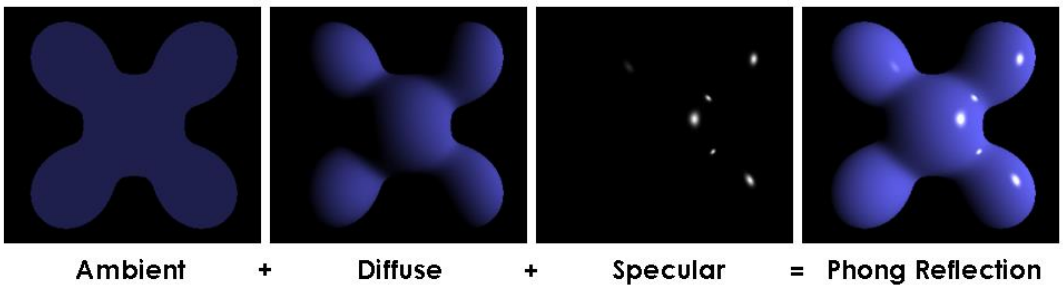
illumination") осветљења у обзир узимају и многе друге факторе, попут утицаја одбијене светлости једног објекта на осветљеност другог, или интензивно одбијање светлости о површину воде или сјајног метала. Овакав вид осветљења зове се индиректно осветљење (енг. "*indirect illumination"*). Глобални модели су напреднија тема која неће бити размотрена овим радом, али како је ова техника интензивно коришћена у данашњој индустрији игара, постоји велики број радова и алгоритама који се баве том проблематиком.



Слика 20 - Локални (лево) и глобални (десно) модел осветљења

Најчешће коришћен локални модел осветљења у графичком подсистему СИИ-а је Фонгов (енг. "*Phong"*) модел рефлексије. Светлост одбијена од површи се рачуна као сума три компоненте:

- *амбијентална* (енг. "*ambient"*) компонента је груба апроксимација индиректног осветљења. Ово спречава да се неосветљени делови објекта исцртавају црно, што се ретко дешава у природи и изгледа вештачки.
- *дифузна* (енг. *"diffuse"*) компонента покрива униформно одбијену светлост од површине. Ово је добра апроксимација реалног понашања светлости.
- *одразна* (енг. "*specular"*) компонента апроксимира одсјај који се често види код материјала попут метала или углачане пластике. Одраз је видљив када се угао гледања поравна са одбијеним зрацима из извора светлости.



Слика 21 - Фонгов модел осветљења

Параметри потребни за рачун Фонговог модела су:

- Вектор смера гледања $\boldsymbol{V} = [\boldsymbol{V_x}\ \boldsymbol{V_y}\ \boldsymbol{V_z}]$ који иде од тачке одбијања светлости до фокалне тачке камере. Другим речима, негирани вектор погледа камере („напред“).
- Интензитет амбијенталне компоненте изражен у RGB вредностима $\boldsymbol{A} = [\boldsymbol{A_R}\ \boldsymbol{A_G}\ \boldsymbol{A_B}]$.
- Нормала површи у тачки где се зрак светлости одбија $\boldsymbol{N} = [\boldsymbol{N_x}\ \boldsymbol{N_y}\ \boldsymbol{N_z}]$.
- Рефлектујућа својства површи изражена преко:
  - амбијенталне рефлексије $\boldsymbol{k_A}$
  - дифузне рефлексије $\boldsymbol{k_D}$
  - одразне рефлексије $\boldsymbol{k_S}$
  - експонента одраза $\boldsymbol{\alpha}$ који дефинише „сјајност“ (енг. “*shininess”* или “*glossiness”*)
- и, за сваки извор светлости $\boldsymbol{i}$:
  - интензитет и боја светла $\boldsymbol{C_i} = [\boldsymbol{C_{R_i}}\ \boldsymbol{C_{G_i}}\ \boldsymbol{C_{B_i}}]$
  - вектор смера $\boldsymbol{L_i}$ од тачке преламања назад до извора светлости

Коначно, Фонгов модел рачуна укупну одбијену светлост $\boldsymbol{I}$ по формули:

$$I = k_A A + \sum_i [k_D(N \cdot L_i) + k_S(R_i \cdot V)^{\alpha}]C_i$$

где је $\boldsymbol{R_i}$ вектор одбијене светлости дуж $\boldsymbol{L_i}$ у односу на нормалу $\boldsymbol{N}$. Овај вектор се може лако израчунати уз мало векторске аритметике. Сваки вектор се може представити као сума тангентних и нормалних компоненти. Вектор смера извора светлости $\boldsymbol{L_i}$ се према томе може написати као:

$$L = L_T + L_N$$

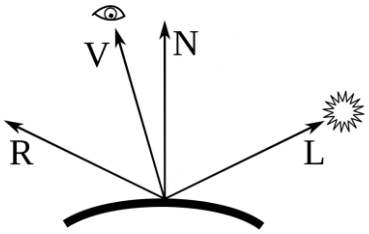
Знамо да скаларни производ $\boldsymbol{N \cdot L}$ представља пројекцију $\boldsymbol{L}$ нормале на површину. Тако да је компонента $\boldsymbol{L_N}$ једнака:

$$L_N = (N \cdot L\ )N$$

Рефлектовани вектор $\boldsymbol{R}$ има исту нормалну а супротну тангентну компоненту у односу на $\boldsymbol{L}$ $\boldsymbol{(-L_T)}$. Рачунамо $\boldsymbol{R}$ по формули:

$$\begin{aligned} R &= L_N - L_T \\ &= L_N - (L - L_N) \\ &= 2L_N - L; \end{aligned}$$

$$\boldsymbol{R = 2(N \cdot L)N - L}$$



Слика 22 - Вектори коришћени при рачуну осветљења у Фонговом моделу

6.3.3. Извор светлости

Поред дефинисања модела интеракције светлости са објектима, потребно је и дефинисати моделе конкретних извора светлости. Као и обично, у исцртавању у реалном времену, апроксимирају се услови из стварног света неком од следећих метода:

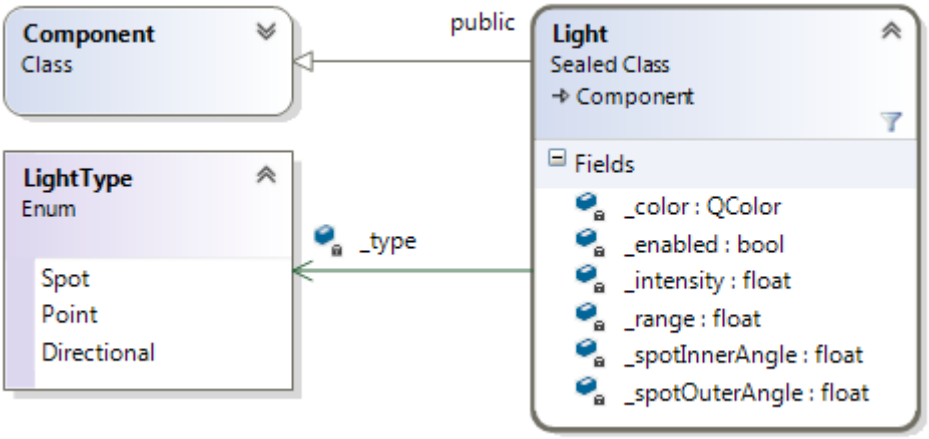
- Статичко осветљење
    - Најбржи вид осветљења је свакако онај који се не рачуна уопште. Идеја је да се Фонгова метода примени једном на сцену и да се добијене информације о осветљењу потом „утисну“ у геометрију (путем атрибута, о чему ће бити речи касније). Ово ће радити брзо, али динамички објекти, очигледно, неће добити добро осветљење како се буду померали кроз сцену. Додатно, објекти често имају налепљене текстуре које су поновљене (енг. “*tiling*”) више пута, те припајање информација о осветљењу не би било практично.
- Амбијентално осветљење (енг. “*ambient light*”)
    - Овај вид извора светлости одговара амбијенталној компоненти Фонг модела. Дакле, у потпуности је одређен бојом и интензитетом датим вектором $\boldsymbol{A}$ и множи се са амбијенталним својством саме површи $\boldsymbol{k_A}$. Амбијентално осветљење је неусмерено и не зависи од угла гледања. Могуће је да интензитет и боја буду променљиви у зависности од региона у простору.
- Усмерено осветљење (енг. “*directional light*”)
    - Усмерено осветљење представља извор светлости који је од осветљене површи „бесконачно“ удаљен (попут сунца). Сви зраци су паралелни а сам извор нема конкретну позицију у простору, те је овај модел у суштини дефинисан само бојом (односно интензитетом) $\boldsymbol{C}$ и смером $\boldsymbol{L}$.
- Тачкасто осветљење (енг. “*point (omni) light*”)
    - Тачкасто осветљење има јединствен положај у простору и емитује светлост равномерно у свим правцима. Обично се рачуна да интензитет

осветљења опада пропорционално квадрату растојања површи од тачкастог извора. Поред положаја $\boldsymbol{P}$ и боје (односно интензитета) $\boldsymbol{C}$, дефинише се и максимални полупречник $\boldsymbol{r_{max}}$. Ван ове границе, интензитет извора је нула. Ово може бити искоришћено као додатна оптимизација, јер графички подсистем не мора да узима у обзир тачкасто осветљење ако се објекат не налази у његовим границама.

- Фокусирано тачкасто осветљење (енг. "*spot light"*)
    - Овај модел има све особине тачкастог осветљења, уз додатно понашање да је емитована светлост ограничена купом у дефинисаном правцу извора. У пракси су то често две купе, како би се постигао лепши ефекат опадања интензитета светлости. Дакле, фокусирано тачкасто осветљење је дефинисано положајем $\boldsymbol{P}$, бојом (односно интензитетом) $\boldsymbol{C}$, смером $\boldsymbol{L}$, полупречником $\boldsymbol{r_{max}}$, као и са додатна два угла који описују унутрашњу и спољашњу купу θ_{min} и θ_{max}.
- Површинско осветљење (енг. "*area light"*)
    - Сви претходно описани извори представљају идеалне тачке у простору. У реалном свету извор светлости скоро увек има неку површину што доводи до вишеструких делова сенке (лат. *umbra* и *penumbra*). Обично се овакви модели не дефинишу експлицитно већ се симулирају разним триковима попут бацања вишеструке сенке.
- Емисиони објекти (енг. "*emissive objects"*)
    - Неки објекти на сцени могу сами бити извори светлости. На пример, неонски знак, батеријска лампа, кристална кугла. Ефекат емитовања светлости се може постићи посебним типом материјала чија су боја и интензитет константни и не зависе од околних извора светлости. Додатно, за објекат се може везати конкретно светло које ће симулирати његово сопствено зрачење.

6.3.4. Класа *Light*

Слично као код камере, и за изворе светлости се дефинише посебна компонента *Light*. Ова компонента треба да носи претходно наведена својства попут боје, интензитета, полупречника (за тачкаста осветљења) итд. Информација о типу осветљења је свакако потребна да би се избегло постојање по једне компоненте за сваки тип. На основу типа, графички подсистем одлучује на који начин ће се осветљење израчунати.



Слика 23 - Класни дијаграм *Light* компоненте из имплементације

6.4. Материјали

6.4.1. Атрибути

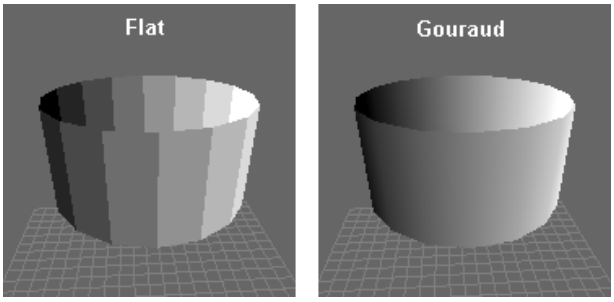
Најједноставнији начин описивања визуелних својстава површи је да се наведу на нивоу дискретизованих тачака у мрежи троуглова. Вертекси су погодно решење за складиштење таквих својстава у ком случају се они зову *атрибути вертекса*. Типични скупови атрибута су углавном (али не и искључиво):

- Вектор положаја ($p_i = [p_{i_x}\ p_{i_y}\ p_{i_z}]$). Представља 3Д координате i-тог вертекса, обично изражене у простору модела.
- Вектор нормале ($n_i = [n_{i_x}\ n_{i_y}\ n_{i_z}]$). Представља јединични вектор нормале површи на позицији i-тог вертекса. Нормале се најчешће користе у рачуну осветљења на нивоу вертекса (енг. "*per-vertex*").
- Тангентни ($t_i = [t_{i_x}\ t_{i_y}\ t_{i_z}]$) и битангентни ($b_i = [b_{i_x}\ b_{i_y}\ b_{i_z}]$) вектори. Ова два вектора су нормална један на други и на вектор нормале n_i. Заједно n_i, t_i и b_i чине осе тангентног простора. Овај простор се користи за разна израчунавања на нивоу пиксела (енг. *"per-pixel"*).
- Дифузна боја ($d_i = [d_{i_R}\ d_{i_G}\ d_{i_B}\ d_{i_A}]$). Одређује дифузну компоненту материјала површи на i-том вертексу изражену у RGB систему. Додатно, садржи алфа компоненту (провидност).
- Одразна боја ($s_i = [s_{i_R}\ s_{i_G}\ s_{i_B}\ s_{i_A}]$). Одређује одразну компоненту материјала површи на i-том вертексу.
- Координате текстуре ($u_{i_j} = [u_{i_j}\ v_{i_j}]$). Омогућују да се дводимензиона (у неким случајевима и тродимензиона) слика „омота" око 3Д модела. Координате текстуре (u и v) одређују локацију i-тог вертекса на слици.

Троугао може бити мапиран са више текстура, тако да може постојати више оваквих атрибута (отуда подознака *j*).

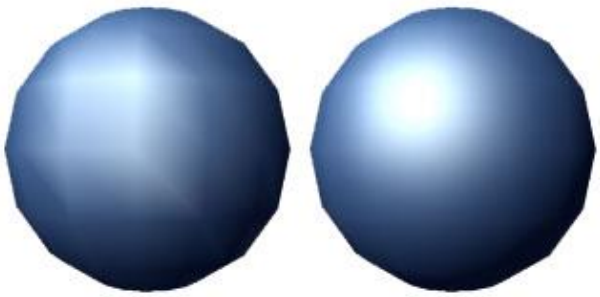
- Скининг (енг. "*skinning*") тежине (k_{ij}, w_{ij}). При скелетној анимацији (описано у поглављу 9), вертекси модела су придружени одређеним „зглобовима" (означено са *k*). Вертекс може бити под утицајем више зглобова у ком случају се крајњи положај рачуна пондерисањем. Тежина којом зглоб утиче на положај вертекса је означена са *w*. У општем случају, на вертекс *i* може утицати више зглобова *j*, од којих је сваки означен паром $[k_{i_j}\ w_{i_j}]$.

Атрибути над вертексима троугла покривају само одређени дисктретизовани подскуп тачака модела. Оно што је стварно од интереса су визуелна својства унутрашњости троуглова, односно оног дела троугла који се види „кроз пикселе" екрана. Другим речима, потребно је одредити атрибуте на нивоу пиксела, не на нивоу вертекса. Једноставан начин да се овако нешто постигне је да се линеарно интерполирају атрибути на нивоу вертекса. Интерполирање атрибута боје вертекса зове се *Гуро сенчење* (енг. "*Gouraoud shading*").



Слика 24 - Стандардни (лево) и Гуро (десно) тип сенчења

Већ је истакнуто да осветљеност у великој мери зависи од нормале површи у тачки додира са светлосним зраком. Уколико се у рачуну користе само нормале вертекса (енг. "*per-vertex lighting*") могу се добити непривлачни резултати. Визуелно лепше (али рачунски интензивније) решење је да се осветљење рачуна на нивоу пиксела (енг. "*per-pixel lighting*"). У овом случају се осветљеност рачуна на основу интерполираног атрибута нормале вертекса у тачки обухваћеној пикселом. Слично као боја и нормале, сви остали атрибути вертекса могу бити интерполирани.



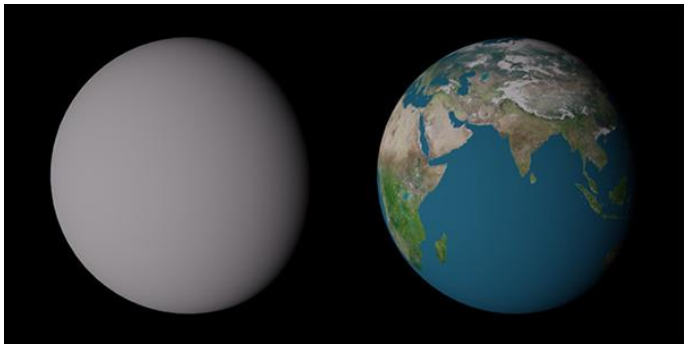
Слика 25 - *Per-vertex* (лево) и *per-pixel* (десно) осветљење

6.4.2. Текстуре

Често није довољно дефинисати само боје на нивоу атрибута и надати се да ће интерполација дати добре резултате. У пракси, већина објеката на сцени своју боју добија из бит-мапираних слика под називом *текстуре*. Текстура не мора нужно садржати само информације о боји, нити се мора пројектовати на мрежу троуглова већ се може користити као упоредна табела (енг. "*lookup table*"). Делови текстуре ће у наставку бити означени као *тексели* како би се разликовали од пиксела на екрану.

Уобичајене резолуције текстура су степени двојке (256x256, 512x512, 1024x1024 итд.). Неки графички хардвери подржавају само такав формат, док неки немају никаква ограничења за резолуцију или величину док год текстура може да стане у меморију. Као што је поменуто, текстуре могу бити носиоци информације о боји (дифузна, одразна) али и о нормалама (тексел чува јединични вектор у RGB формату), о провидности (тексел чува алфа вредност), о окружењу (садржи слику окружења што може бити искоришћено за рефлексије) и многе друге. Текстура се, у општем случају, може користити за складиштење било каквих информација које су од значаја при рачуну осветљења (упоредне табеле).

Координате текстуре (u, v) дефинишу како се 2Д мапа пројектује на 3Д мрежу троуглова. Дводимензиони координатни систем текстура назива се *простор текстура* (енг. "*texture space*"). Пар координата (u, v) увек иде од (0, 0) у доњем левом до (1,1) у горњем десном углу. Овако нормализоване координате су независне од резолуције текстуре. Дакле, мапирање текстуре се своди на придруживање пара координата (u_i, v_i) сваком вертексу i.

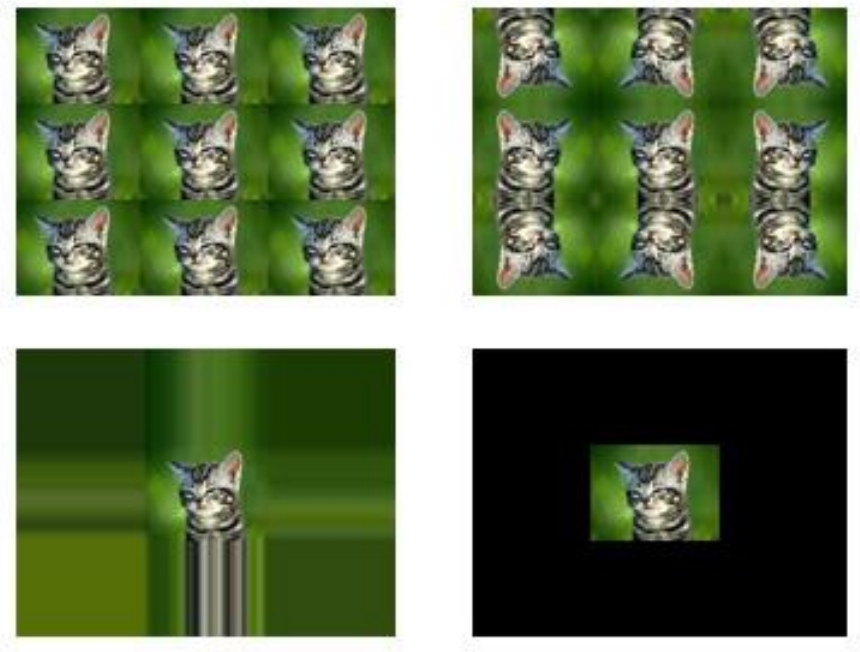


Слика 26 - Текстура планете Земље мапирана на сферни модел

Уколико координате текстура изађу из опсега [0, 1] графички хардвер може да адресира вредност користећи неке од предефинисаних подешавања које корисник задаје:

- *Repeat*. У овом режиму текстура се понавља по свим правцима. Другим речима координате (ju, kv) су еквивалентне координатама (u, v) где су j и k произвољни цели бројеви.
- *Mirror*. Понаша се исто као *Repeat* с тим што се текстура обрће по v оси за сваки непарни умножак u и по u оси за сваки непарни умножак v.

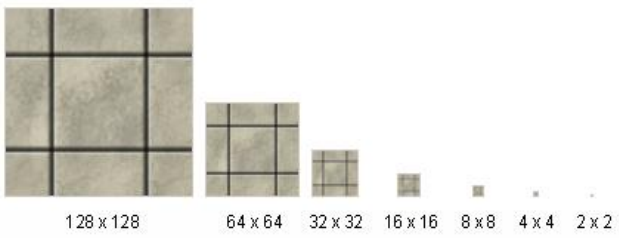
- *Clamp*. Тексели око ивице текстура се једноставно понављају за све координате ван опсега.
- *Border color*. У овом режиму, кориснички дефинисана боја се узима за све координате ван опсега.



Слика 27 - Редом (одозго ка доле, с лева на десно) ***Repeat, Mirror, Clamp, Border***

6.4.3. Минимапирање

Текстура много високе резолуције може бити пресликана на објекат који је далеко од камере. Не само да је то непотребан временски утрошак, већ долази и до визуелних аномалија јер је однос тексела по пикселу (густина тексела) много велик те долази до „борбе“ за боју у пикселима удаљеног објекта. У ту сврху је уведена техника минимапирања (енг. “*mipmapping*”). За сваку текстуру генерише се низ мањих битмапа, где је свака нова представљења половином резолуције претходника. Графички хардвер потом бира одговарајући ниво резолуције на основу удаљености троугла од камере како би одржао однос тексела и пиксела близак јединици.



Слика 28 - Пример ***mipmapping***-а

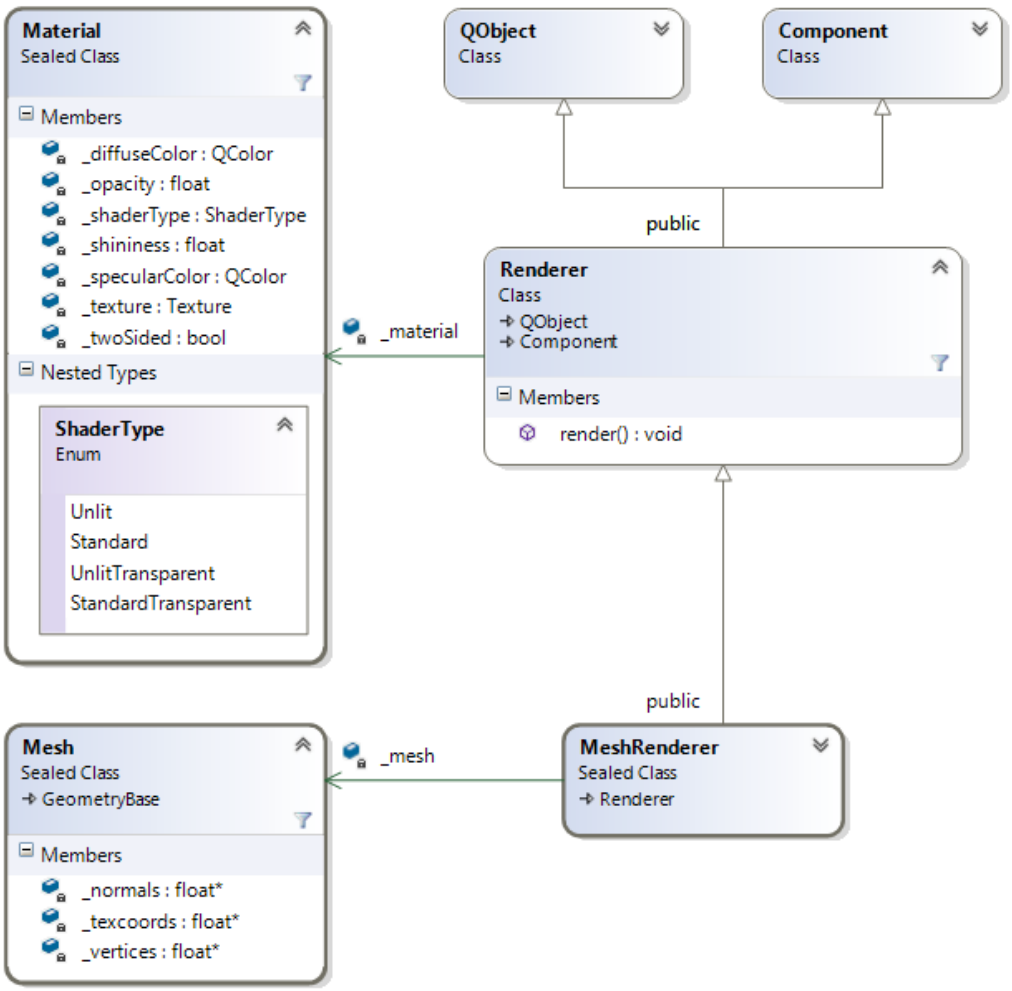
При исцртавању једног пиксела текстурисаног троугла, графички хардвер узоркује (енг. "*sampling*") текстуру гледајући где тај пиксел „пада" у простору текстура. Како пресликавање између пиксела и тексела обично није један-на-један, може се десити да центар пиксела буде на граници између два или више тексела. У том случају графички хардвер примењује неку од предефинисаних техника које корисник задаје:

- *Nearest neighbor.* Тексел који је најближи центру пиксела бива одабран.
- *Bilinear.* Четири окружујућа тексела су узоркована и резултујућа боја је пондерисана сума њихових боја где су тежине одређене удаљеношћу од центра пиксела.
- *Trilinear. Bilinear* приступ се примењује на два најближа мип нивоа (уколико је мипмапинг омогућен).
- *Anisotropic.* Мана претходна два приступа је што су узорковани блокови 2x2 тексела правоугаоног облика и то је довољно добро када се текстурисана површ посматра чеоно. Међутим, уколико се површ гледа под углом *Anisotropic* је боље решење јер узоркује текселе трапезоидним регионом где је трапезоид дефинисан углом гледања површи.

Под *материјалом* се могу сматрати сва визуелна својства објекта. То укључује све дефинисане текстуре, рутине за сенчење, параметре за рутине као и сваки вид упутства за исцртавање који се доставља графичком хардверу. Иако, технички гледано, атрибути вертекса не спадају у материјал они су нераздвојни део модела (мреже троуглова) и као такви се не могу занемарити. Често се у литератури среће појам *mesh-material* као скуп свих информација које су нам потребне да исцртамо објекат.

6.4.4. Класа *Renderer*

У имплементацији СИИ приложеној уз рад, сва својства за исцртавање објекта налазе се у *Renderer* фамилији компоненти. Као базни тип, *Renderer* дефинише својство *Material*. Изведени тип *MeshRenderer* представља специјализовану компоненту која чува референцу ка мрежи троуглова. Додатно, може постојати и *ParticleRenderer*, који би своју геометрију чувао у другачијем формату. *Material* садржи већину описаних визуелних својстава који се достављају графичком подсистему. Атрибути вертекса који се користе су положај, нормала и координате текстуре.

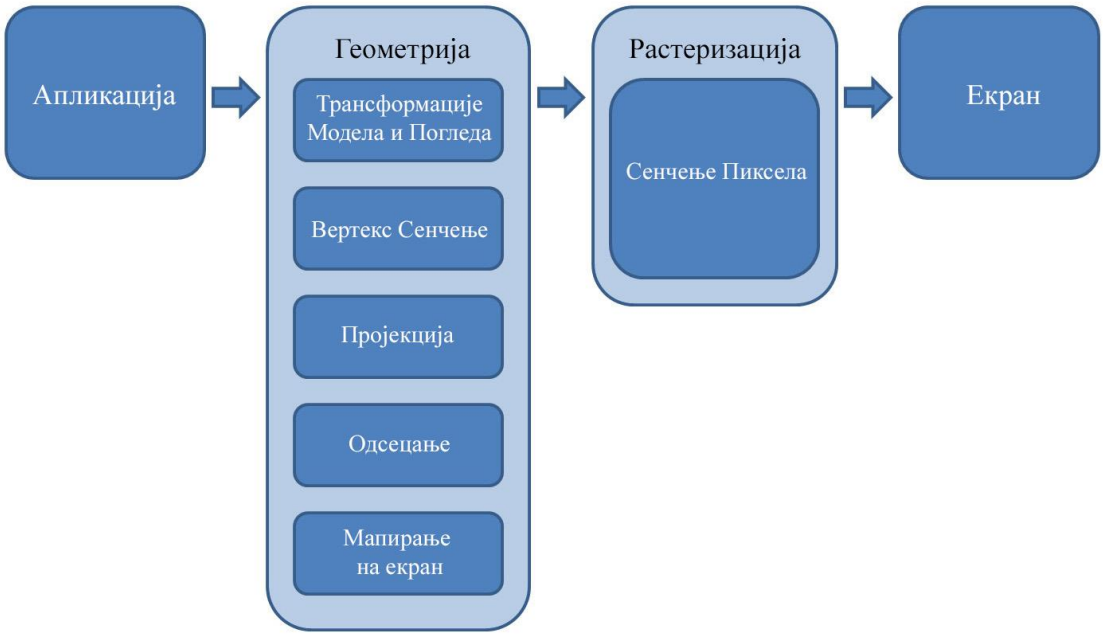


Слика 29 - Класни дијаграм компоненти из графичког подсистема имплементације

6.5. Графички ток

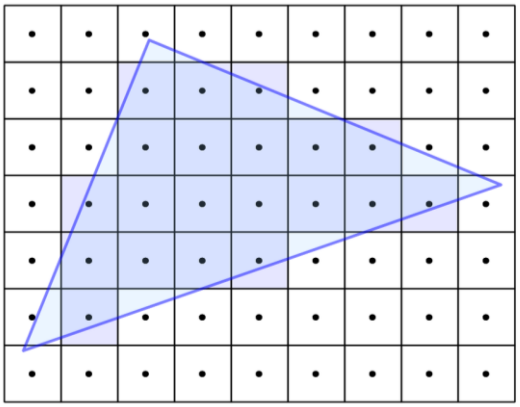
Термин *ток* (енг. "*pipeline*"), у техничком смислу, означава уланчан низ јединица (блокова) које треба да изврше посао поделом одговорности између себе, непрекидно обрађујући улазни ток података и пропагирајући га у наредни блок. Када један блок пошаље свој излаз даље, он може да прими нови скуп података за обраду. Управо највећи бенефит тока лежи у одређеном нивоу паралелизма. Поједини блокови могу извршавати задатке паралелно и унутар себе.

У контексту рачунарске графике, ток представља конкретну хардверско-софтверску имплементацију претходно изложених концепата. На слици је дат дијаграм уобичајеног тока за исцртавање у реалном времену. Прве две колоне су, суштински, већ описане у претходним поглављима.



6.5.1. Растеризација троугла

Да би се троугао исцртао на екрану, потребно је попунити пикселе које он покрива. Овај процес је познат као *растеризација* (енг. "*rasterization*"). Током растеризације површина троугла се дели на мале регионе који одговарају једном пикселу на екрану. Ови региони се зову *фрагменти*. Да би се фрагмент уопште узео у обзир при рачуну боје пиксела, он мора проћи низ тестова. Један од таквих тестова јесте тест дубине фрагмента (енг. "*depth testing*").



Слика 30 - Растеризација троугла, плави квадрати означавају фрагменте

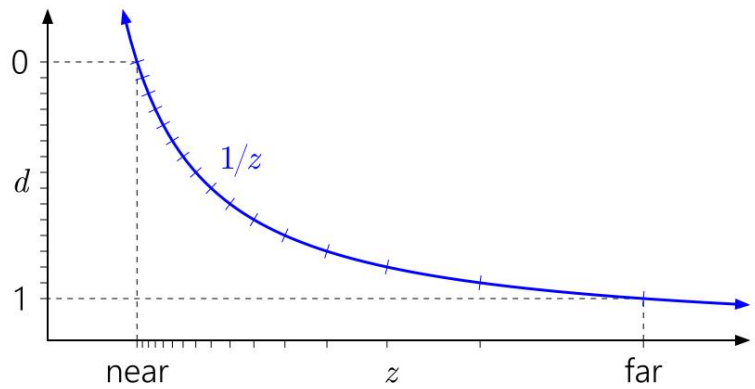
6.5.2. Тест дубине

При исцртавању два троугла која се преклапају, битно је осигурати да се троугао ближи камери исцрта последњи. Исцртавање од назад ка напред (енг. "*back to front*") је задовољавајуће решење докле год се троуглови не пресецају (овај метод је познат као *алгоритам сликара* [11]). Како би се овај проблем превазишао, потребан је

механизам који не зависи од редоследа исцртавања троуглова. У те сврхе, графички ток користи технику баферовања дубине (енг. "*depth buffering*" или *"z-buffering"*). Бафер дубине обично садржи 16- или 24-битне децималне бројеве у покретном зарезу који чувају информацију о дубини за конкретан пиксел у баферу слике (енг. "*frame buffer*"). Сваки фрагмент има z координату која одређује његову „дубину у екрану". Ова координата се добија интерполирањем дубина самих вертекса троугла. Уколико надолазећи фрагмент има дубину мању од оне која је тренутно у баферу слике („ближи" је екрану) његова дубина се преписује преко постојеће, у супротном, не пролази тест и његова даља обрада се обуставља.

6.5.3. z-fighting

При исцртавању паралелних површина које су јако близу једна другој, важно је да графички подсистем може да разликује њихове дубине у односу на поглед. Нажалост, величина бафера је ограничена (обично 24 бита) те је и прецизност дубина које може да чува ограничена. Ово може довести до графичких аномалија где би се фрагменти два троугла „борили" (отуда и назив) око дубине због немогућности бафера да разликује њихове вредности. Додатно отежавајућа чињеница је да бафер дубине нема униформно дистрибуирану прецизност за цео опсег запремине погледа (конкретно растојање између *near* и *far* равни). Прецизност се смањује са растојањем објеката од камере, те је *z-fighting* још израженији за далеке објекте. До ове појаве долази због начина на који се дубина рачуна у простору одсецања где се вредности деле са z координатом из простора погледа. Из графика криве $1/z$ може се видети да се разлике у растојањима удаљених објеката у простору погледа преводе у доста мали опсег вредности у простору одсецања.



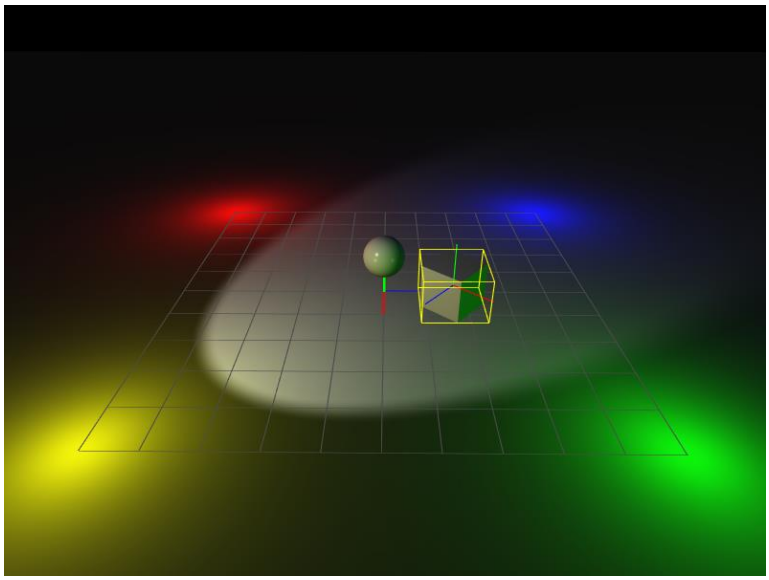
Слика 31 - Расподела вредности дубине фрагмента d [0,1] у односу на растојање z

Једно од решења је користити z вредности директно из простора погледа јер су оне линеарно распоређене између *near* и *far* равни камере. Згодно, управо ова вредност се појављује као w компонента хомогеног простора одсецања. У пракси, наведена техника је нешто мало сложенија и зове се *w-buffering* [12]. Алтернативно, постоји још пар техника за делимично или скоро потпуно превазилажење овог проблема као што су *reversed z* [13] или *infinite projection matrix* [14].

6.5.4. Рутине за сенчење

На самом почетку, графички хардвер је нудио ток фиксне функционалности (енг. "*fixed functionality pipeline*"), тј. графички програмер је имао јако малу контролу (обично само конфигурисање) над током исцртавања. Временом, делови тока су постајали све више програмабилни и програмери су могли да утичу на начин извршавања пишући своје рутине за сенчење (енг. "*shaders*"). Сваки модеран графички хардвер подржава следеће типове програмабилних рутина за сенчење:

- Вертекс сенчење:
  - Улаз је вертекс чије су позиције и нормале изражени у простору модела или у простору света. Излаз је осветљен вертекс, изражен у хомогеном простору одсецања.
- Геометријско сенчење:
  - Улаз су примитиве попут тачке, линије или троугла изражене са n вертекса (n = 1, 2, 3) уз до n контролних тачака. Излаз су нула или више примитива, не нужно исти број као и улазних. Примера ради, могу се елиминисати неки непотребни троуглови.
- Сенчење пиксела:
  - Улаз је фрагмент троугла чији су атрибути интерполирани између три вертекса којима је дефинисан. Излаз је боја која је израчуната за тај пиксел уколико је фрагмент прошао све тестове. Сенчење пиксела је у могућности и да одбаци фрагмент из било ког разлога и у том случају не даје никакав излаз.



Слика 32 - Тест сцена за рутине за сенчење и осветљење коришћена у имплементацији СИИ

7. Подсистем за физичке прорачуне

До сада су биле описане технике које се махом тичу визуелног утиска који објекат даје у игри. Међутим, у већини случајева, игра се не своди на шетње парком или посматрање заласка сунца већ укључује велику меру интерактивности и динамике између играча и објеката као и између самих објеката. Примера ради, играч може шутирати лопту у спортским симулацијама или уништавати окружење у играма пуцања. Како објекти могу да раде само оно што им је „речено“, програмери улажу велики труд да би развили што реалистичнији динамички модел у *подсистему за физичке прорачуне* (енг. “*physics engine*”). Физика је, у реалности, огромна област и под „физиком“ у играма се обично сматра мањи подскуп понашања од којих су, у већини случајева, два најчешће имплементирана:

- Детекција колизије (енг. “*collision detection*”)
- Динамика крутих тела (енг. “*rigid body dynamics*”)

Важно је напоменути да се динамика крутих тела ослања на детекцију колизије док обрнуто не важи. Стога, физички подсистем може подржавати и само детекцију колизије док би се динамика симулирала неким другим путем (нпр. анимацијом).

7.1. Детекција колизије

Скоро свака игра захтева неки вид провере да ли се две мреже троуглова пресецају или додирују. Једноставно решење би било вршити за свака два троугла проверу за обе мреже, међутим временска сложеност таквог приступа је неприхватљива за апликацију у реалном времену. У општем случају, поступак детектовања колизије између објеката на сцени се своди на три фазе:

- Широка фаза (енг. “*broad phase*”). У овој фази задатак је искористити додатне просторне структуре података како би се брзо елиминисали објекти који никако не могу бити у колизији.
- Средња фаза (енг. “*mid phase*”). Када су одређени потенцијални кандидати, проверавамо да ли се њихове габаритне апроксимације преклапају. Габаритне апроксимације су углавном примитиве попут квадра или сфере које обухватају конкретну геометрију. Ако оваква провера не врати позитиван резултат, може се са сигурношћу тврдити да колизија не постоји.
- Уска фаза (енг. “*narrow phase*”). У овој фази се извршава алгоритам провере пресецања на нивоу конкретне геометрије модела објекта.

7.1.1. Облици и апроксимације облика

Један од најважнијих концепата у детекцији колизије је разликовање *конвексних* и *не-конвексних* (*конкавних*) облика. Особина конвексности је важна јер детекција колизије између облика који имају то својство доста једноставнија и бржа него код конкавних облика (конкретни алгоритми ће бити наведени касније). Поједини СИИ уопште не подржавају провере колизије над конкавном геометријом (или је процес у многоме ограничен, нпр. само провере са геометријом терена), те се њихов облик мора апроксимирати једним или више конвексних облика. Уобичајено се користе неке од следећих структура:

- Гранична сфера (енг. "*Bounding Sphere*")
    - Најједноставнији вид апроксимације. Дефинише се вектором позиције и скаларом који представља полупречник сфере (ове информације се могу уписати и у четвородимензиони вектор). Провера преклапања је брза и структура не мора у потпуности да се прерачунава при афиним трансформацијама објекта.
- Гранична кутија поравната са осама (енг. "*Axis Aligned Bounding Box – AABB*")
    - AABB је квадар чије су стране паралелне са осама одређеног координатног система. Дефинисан је са само две тачке где једна означава минимум (мин) а друга максимум (макс) вредности по свим осама. Главна предност ове структуре је брза провера преклапања са другим AABB-ом. Главна мана је што мора остати поравната са осама како би очувала своја својства. То значи да се приликом ротације објекта, струкура мора прерачунати. Чак и ако је геометрија објекта облика квадра, AABB може дати јако лошу апроксимацију.
- Оријентисана гранична кутија (енг. "*Oriented Bounding Box – OBB*")
    - OBB представља AABB са могућношћу ротације у сопственом координатном систему. Обично се задаје са полу-димензијама (полу-висина, полу-ширина, полу-дужина) и трансформацијом која позиционира и оријентише квадар. OBB представља доста бољу апроксимацију за произвољно ротиране објекте док уједно има једноставну репрезентацију.
- Произвољни конвексни облици
    - Многи СИИ дозвољавају и апроксимацију габарита објекта произвољним конвексним облицима. Системи попут *Unity*-а нуде и могућност генерисања *конвексног омотача* [15] од конкавног облика који се потом може искористити за апроксимацију.

- Сложени облици
    - Ова техника је најчешће коришћена у пракси. Уместо једним, објекат се апроксимира вишеструким примитивним, конвексним облицима. Детектор тада извршава провере колизије за сваки од облика, што је обично доста брже него провера са оригиналним обликом, поготово ако је конкаван.

7.1.2. Детекција колизије и аналитичка геометрија

Систем за детекцију колизије користи *аналитичку геометрију* (математички опис тродимензионих површи и запремина) како би рачунски открио пресеке између облика. У овој секцији, укратко ће бити описани основни концепти и примери употребе аналитичке геометрије као и алгоритам за детекцију колизије између произвољних конвексних *полиедара* [16].

- Тачка и сфера
    - Може се одредити да ли се тачка налази унутар сфере једноставном провером да ли је вектор растојања од центра сфере до тачке мањи (или једнак) од полупречника сфере:

$$s = c - p;$$
$$if\ |s| \leq r, then \text{ тачка } p \text{ је унутар сфере}$$

- Сфера и сфера
    - Одређивање да ли се две сфере пресецају је подједнако лако као и провера да ли је тачка у сфери. Формира се вектор растојања s између центра две сфере. Ако је растојање мање или једнако од збира полупречника обе сфере, постоји пресек, иначе не.

$$s = c_1 - c_2;$$
$$if\ |s| \leq (r_1 + r_2), then \text{ сфере се пресецају}$$

Додатно, може се избећи рачун квадратног корена при рачуну дужине вектора $\boldsymbol{s}$ тако што квадрирамо цео израз.

$$s = c_1 - c_2;$$
$$|s| = \sqrt{s \cdot s};$$
$$if\ |s|^2 \leq (r_1 + r_2)^2, then \text{ сфере се пресецају}$$

- AABB и AABB
    - Да би тестирали пресек две граничне кутије поравнате са осама (A и B), довољно је да тестирамо минимум и максимум оба дуж сваке осе појединачно. Дуж x осе имамо два интервала $[x^A_{min}, x^A_{max}]$ и $[x^B_{min}, x^B_{max}]$, аналогно томе, дефинишу се интервали по y и z осама.

Уколико се интервали преклапају по *све три осе*, два AABB-а се пресецају, иначе не.

- Теорема о оси раздвајања
    - Многи системи за детекцију колизије се у великој мери ослањају на *теорему о оси раздвајања* (енг. "*The Separating Axis Theorem*" [17]). Теорема каже да, ако не постоји оса за коју се пројекције конвексних облика не преклапају, онда се облици пресецају. Иста тврдња не важи за конкавне облике. Пројекцију конвексног облика на осу се може означити затвореним интервалом $[c_{min}, c_{max}]$. Да би се детектовао пресек између два облика A и B, пролази се кроз све потенцијалне осе раздвајања и проверава да ли се интервали пројекције $[c^A_{min},\ c^A_{max}]$ и $[c^B_{min},\ c^B_{max}]$ не преклапају. Математички гледано, интервали се не преклапају ако важи $c^A_{max} < c^B_{min}$ или $c^B_{max} < c^B_{min}$. Ако се интервали не преклапају, онда је нађена оса раздвајања те се облици не пресецају. За неке специфичне облике, унапред се зна које осе треба проверити. Један од таквих примера је управо претходно описан пресек између AABB-ова који је у суштини специјалан случај ове теореме.
- ГЈК алгоритам
    - Постоји ефикасан алгоритам за детекцију пресека произвољних конвексних полигона у 2Д (односно полиедара у 3Д). Алгоритам се зове ГЈК а назив је добио по иницијалима својих аутора E.G. **G**ilbert, D.W. **J**ohnson и S.S. **K**eerthi са Универзитета у Мичигену [18]. Овај алгоритам се ослања на геометријску операцију звану *разлика Минковског* (енг. "*Minkowski difference*" [19]) која, у суштини, представља скуп тачака добијен одузимањем свих тачака (пар по пар) облика B од облика A, означен са $\{(A_i - B_i)\}$. Корисно својство разлике Минковског је да садржи тачку (0, 0) уколико се облици пресецају. Ово је логично с обзиром да, уколико пресек постоји, облици морају имати бар једну заједничку тачку. Одузимањем ових тачака добија се управо (0, 0). ГЈК је итеративни алгоритам који у свакој итерацији покушава да формира *симплекс* (у 3Д простору симплекс је тетраедар) који обухвата координатни почетак, усмеравањем претраге по тачкама из скупа разлике Минковског. Битно је напоменути да су само спољне тачке од интереса, као и да тачке не морају бити претходно израчунате већ се рачунају „у ходу". Одличан увид у то како овај алгоритам функционише дат је на *Molly Rocket* вебсајту у видео објашњењу под називом "*Implementing GJK*" [20].

7.1.3. Детекција колизије за објекте у покрету

До сада је разматрано само детектовање колизије за непомичне (статичне) објекте. Међутим, уколико се објекти померају (што се неретко дешава у играма), то уводи додатан ниво комплексности. При померању објеката, утисак кретања је обично симулиран дискретизованим корацима и коришћењем неког вида интерполације зарад утиска флуидности.

Најједноставније решење за детекцију колизије објеката у покрету је третирати их као непокретне у кључним тачкама и извршити стандардни статички модел детекције колизије. Ово решење је врло често имплементирано, али доводи до проблема уколико се мала тела крећу много брзо. У тим ситуацијама може се десити да је растојање које је тело прешло између два дискретна корака доста дуже од ширине тела тако да, уколико се неки други објекат нашао између, колизија неће бити детектована. Описани проблем је познат као *проблем проласка метка кроз папир* (енг. "*bullet through paper problem*") или проблем *проласка кроз тунел* (енг. "*tunneling problem*").

Један од начина да се заобиђе овај проблем је да се колизија не рачуна са обликом конкретног покретног тела, већ са обликом који је добијен померањем тела дуж неког правца (енг. *"swept shapes"*). Проблем са овим решењем је то што померање неког конвексног облика дуж произвољне криве (поготово ако се облик и ротира) не даје увек конвексан облик. Додатно, линеарна интерполација између дискретизованих тачака није најбоље решење апроксимације кривудаве путање нити објекта који се ротира у покрету, те то додатно компликује рачун.

Други начин заобиласка проблема тунеловања је тзв. континуална детекција колизије (енг. "*Continuous Collision Detection – CCD*"). Циљ овог вида детекције је проналазак најранијег времена судара (енг. "*time of impact* – TOI") између два објекта у покрету у датом временском интервалу. CCD алгоритми су итеративне природе. Своде се на одржавање позиције и оријентације објекта из претходног дискретног корака и тренутног. Информације се могу користити за одвојену линеарну интерполацију позиције и ротације у било ком временском интервалу између два дискретна корака. Алгоритам тада тражи најраније време судара дуж путање кретања објекта. За претрагу се може користити више алгоритама, укључујући и Брајан Миртичов (енг. "*Brian Mirtich*") метод *конзервативног напредовања* (енг. "*conservative advancement*").

7.1.4. Упити о колизији

Још једна одговорност система за детекцију колизије је да буде у могућности да пружи одговор на захтеве попут:

- Ако метак путује из оружја играча у правцу и смеру у коме је уперено оружје, који објекат ће прво погодити?
- Да ли се возило може померити од тачке А до тачке Б без сударања?
- Пронађи све непријатеље око играча у одређеном радијусу.

У ове сврхе се користе *упити о колизији* (енг. "*collision queries*"). Разлика између упита и регуларне детекције колизије је што су ентитети (објекти) коришћени у упитима привремени, немају визуелну репрезентацију и ни на који начин не утичу на стање на сцени. Најчешћи тип упита је *пробни упит* (енг. "*cast, probe, trace*") и то конкретно:

- Линијски пробни упит (енг. "*ray cast*")
    - Најједноставнији вид пробног упита. Дефинише се усмерена дуж задата почетном и крајњом тачком или полуправа задата почетном тачком и смером, мада многи системи интерно не подржавају бесконачне (полу)праве те крајња тачка свакако мора постојати. Тачку p на дужи дефинисаној са p_0 и p_1 можемо описати параметарском једначином по t:

$$p(t) = p_0 + td, \qquad t \in [0,1]$$

где важи $p_0 = p(0)$ и $p_1 = p(1)$. Систем детекције колизије за овакав упит враћа (уколико постоји) први објекат у простору који се пресеца са задатом дужи (гледајући од смера почетне тачке). Неки системи могу да врате и све објекте који се пресецају са задатом дужи. Додатно, као резултат се може вратити тачка пресека дефинисана експлицитно (координатама) или параметром дужи t. Линијски пробни упити имају велику примену у играма. На пример, може се брзо тестирати да ли играч А види играча Б, за детекцију погодака из неког оружја (који објекат је метак погодио) или ради провере да ли ће карактер контролисан вештачком интелигенцијом ударити у зид итд.

- Пробни упит обликом (енг. "*shape cast*")
    - Други, чести случај пробног упита је питање да ли произвољан конвексни облик пресеца неки објекат на дужини d у задатом правцу. Слично као и код линијског упита, дефинише се почетна тачка p_0, дужина d као и тип, димензије и оријентација облика за који треба да се изврши упит. Облик се „пушта" дуж дефинисаног правца и зауставља при детекцији колизије. Главна разлика у односу на линијски упит је та што у овом случају систем детекције колизије

мора да има могућност враћања више детектованих пресека. Без обзира на одабир облика за упит, скоро је увек могуће дотаћи више тачака (једног или више објеката) одједном. Детектовани пресеци се могу вратити као списак објеката заједно са списком тачака (координатама или параметарски, слично као код линијског упита). Најчешће коришћен облик за овакве типове упита је сфера у ком случају се техника зове пробни упит сфером (енг. "*sphere casting*"). Примене у играма су вишеструке. На пример, провера да ли се играч може неометано померити дуж неког правца или спречавање камере да пролази кроз околну геометрију.

- Фантоми
  - Специјалан случај пробног упита обликом су фантоми (енг. "*phantoms*"). Разлика је у томе што фантоми немају дефинисану дужину кретања (d је нула), већ се упит врши „у месту". У овом случају, додатне оптимизације се могу извршити од стране система за детекцију колизије. Фантоми могу да одговоре на хипотетичко питање „Са киме би био у колизији да сам прави објекат?".
- Остали упити
  - Поједини системи подржавају и упите попут *упита најближе тачке* (енг. "*closest point query*") који за дати објекат враћа скуп најближих тачака околних објеката у простору колизије.

7.1.5. Додатне оптимизације

Детекција колизије је захтеван посао за CPU из два разлога:

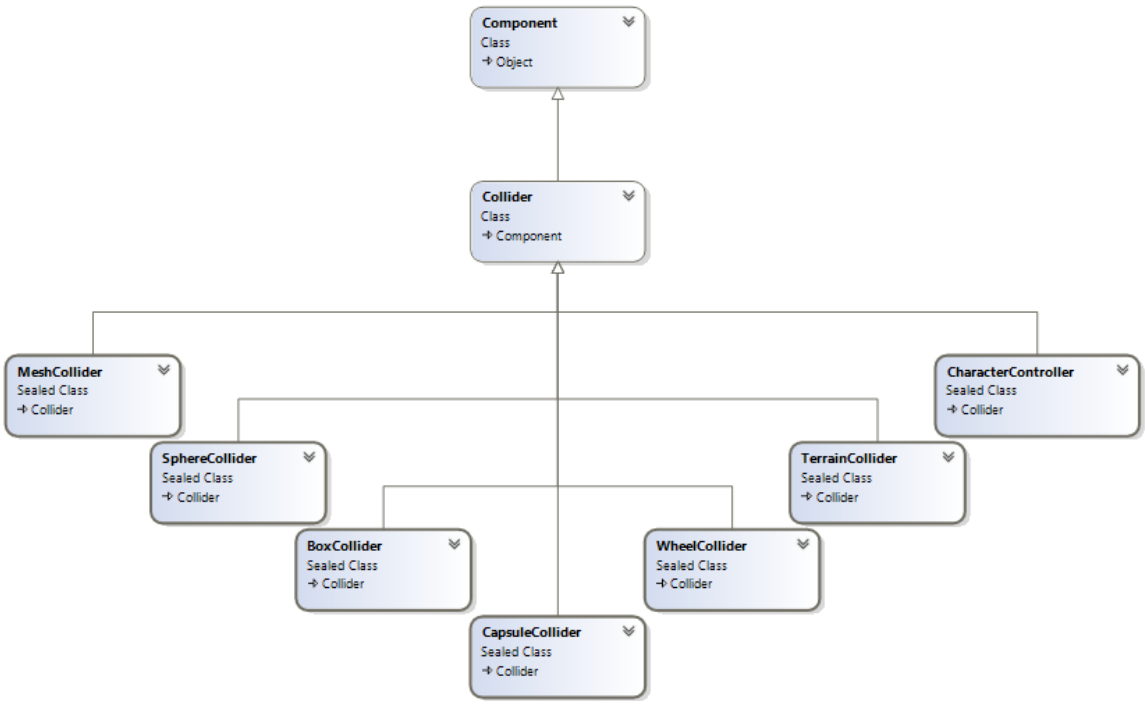
1. Прорачуни за упит о пресеку облика нису тривијални.
2. Већина сцена у играма садржи велики број објеката тако да број потребних тестова може драстично да порасте.

Алгоритам грубе силе за тестирање колизије између n објеката би захтевао проверу сваког пара, те би временска сложеност била $O(n^2)$. Срећом, постоје много ефикасније технике које се користе у пракси попут просторне поделе (енг. "*spatial partitioning*"), хијерархије граничних запремина (енг. "*bounding volume hierarchies*") и слично.

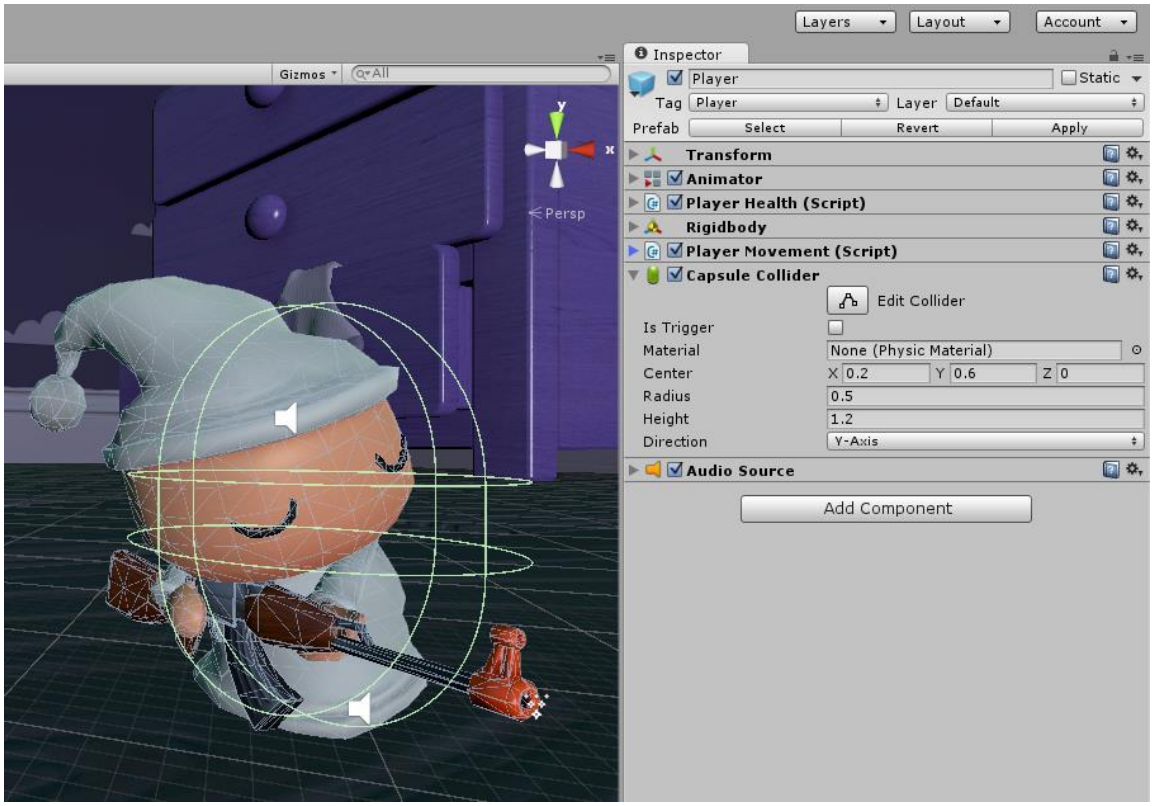
Уобичајени начин просторне поделе је дрволика структура која дели простор на ситније делове, почевши од корена који обухвата габарите целе сцене. Уколико се два објекта у потпуности налазе у различитим листовима, то је сигуран знак да провера између њих није потребна (широка фаза). Једна од таквих структура је *октално дрво* (енг. "*octree*") [21].

7.1.6. Имплементација

Као пример употребе компоненти у детекцији колизије, посматра се систем имплементиран у *Unity*-у. *Unity* дефинише конвексне апроксимације објекта путем фамилије компоненти *Collider* [22]. Ово је заправо базни тип из кога изводе многи други попут *BoxCollider*, *SphereCollider, CapsuleCollider* или *MeshCollider* компоненте. Геометрија ових компоненти је *конвексна* и невидљива у игри јер служи искључиво подсистему за физичке прорачуне. Један објекат може имати више *Collider*-а. Приликом детектовања колизије са другим *Collider*-ом, компонента шаље поруку *OnCollisionEnter* на коју је могуће реаговати одговарајућим понашањем (које програмер дефинише). Слично, *OnCollisionStay* и *OnCollisionExit* поруке су емитоване док контакт траје, односно када је контакт раскинут. Уколико је од интереса само информација да ли колизија постоји, али није пожељно да систем спречи објекте да пролазе један кроз други, користиће се својство *IsTrigger*. *Collider* који је конфигурисан као *Trigger* не понаша се као чврсто тело и дозвољава пролазак других *Collider*-а, притом емитујући одговарајуће поруке *OnTriggerEnter*, *OnTriggerStay* и *OnTriggerExit*.



Слика 33 - *Collider* фамилија типова у Unity СИИ.



Слика 34 - Модел играча са додељеним *CapsuleCollider*-ом. Габарити модела су апроксимирани капсулом ради ефикаснијег рачуна колизије.

```
void OnTriggerEnter (Collider other)
{
	/* Ukoliko collider pripada igraču,
	   igrač je u opsegu za napad */
	if(other.gameObject == player)
		playerInRange = true;
}

void OnTriggerExit (Collider other)
{
	/* Ukoliko collider pripada igraču,
	   igrač je van opsega za napad */
	if(other.gameObject == player)
		playerInRange = false;
}

void Update ()
{
	if(playerInRange)
	{
		/* Ukoliko je igrač u opsegu,
		   izvršiti napad */
		Attack ();
	}
}
```

Слика 35 – Пример кода за реакцију на детекцију колизије између модела непријатеља и играча.

7.2. Динамика крутих тела

Многи СИИ дефинишу посебан тип подсистема за физичке прорачуне који се бави искључиво прорачуном *динамике.* У физици, динамика се бави изучавањем кретања тела под дејством сила. Конкретно, од интереса је поддисциплина под називом *класична динамика крутих тела* што намеће два битна ограничења симулације динамике:

- Објекти симулације подлежу Њутновом (класичном) закону механике.
- Објекти симулације су савршено крута тела тј. њихов облик је константан у времену. Ово ограничење знатно поједностављује рачун динамике чврстих тела и уклапа се добро са претпоставкама наметнутих у подсистему за рачун колизије.

Подсистем за рачун динамике такође осигурава да кретање објеката симулације подлеже разним ограничењима. Једно од најчешћих ограничења је немогућност продора између објеката, већ се подсистем труди да што реалније симулира реакције при судару у складу за законима физике.

Крута тела која чине симулацију често нису идентична са објектима који представљају визуелну репрезентацију сцене. Један визуелни објекат (попут карактера) може имати више додељених крутих тела у симулацији динамике, слично као што може имати више конвексних апроксимација у детекцији колизије. Подсистем за детекцију колизије и динамике често деле податке који, заједно, дефинишу *физичку сцену* или *физички модел.* [23]

8. Аудио подсистем

Као што је одговорност графичког подсистема да играчу понуди најбољи визуелни доживљај, може се рећи да је одговорност аудио подсистема да играчу понуди најбољи звучни доживљај. Звук је јако битан део игара, поготово у жанровима где је велики носицал атмосфере (хорор, игре преживљавања, пуцање из првог лица). Теоријска позадина звука неће бити део овог поглавља, већ ће акценат бити стављен на архитектуру и техничке детаље из угла имплементације овог подсистема у СИИ.

8.1. Одговорности аудио подсистема

Било која игра у 3Д простору захтева адекватан 3Д аудио подсистем. Циљ је обезбедити играчу највећи могући квалитет звучног доживљаја који одговара ситуацији приказаној на сцени. Улаз аудио подсистема су обично вишеструки *3Д звуци* (кораци, оружја, амбијентални звуци итд.) које потом треба обрадити у вишеканални (енг. "*multichannel*") излаз. Идеално, излаз би требао да буде у формату подржаном од стране напреднијег вида озвучења (попут 5.1 и 7.1 система) али исто тако мора да буде прилагођен и за играче са скромнијим хардвером попут стандардних звучника или слушалица. Такође, аудио подсистем би требало да репродукује све додатне звуке који не потичу из виртуелне сцене (ефекти менија, музика, нарација итд). Овакви звуци се обично означавају као *2Д звуци* и миксују се накнадно, заједно са обрађеним 3Д звуцима.

Главни задаци које аудио подсистем треба да испуни су:

- *Синтеза звука* (енг. "*sound synthesis*") представља процес генерисања звучних сигнала који одговарају догађајима у игри. Звуци могу бити већ снимљени узорци или процедурално генерисани у време извршавања.
- *Просторни распоред* (енг. "*spatialization*") ствара илузију да 3Д звуци долазе из неке специфичне тачке у простору релативно позицији посматрача (камере). Овакав ефекат се постиже контролом јачине (амплитуде) звучних таласа и то на два начина:
    - *Слабљење на основу растојања* (енг. "*distance-based attenuation*") контролише укупну јачину звука ради постизања ефекта радијалне удаљености од слушаоца.
    - *Померај* (енг. *"pan"*) контролише релативну јачину звука за сваки од звучника како би дочарао осећај смера извора звука.
- *Акустучно моделовање* (енг. "*acoustical modeling*") појачава дозу реализма симулирајући одбијања звука и акустику простора кроз који се звук простире, узимајући у обзир објекте који делимично или потпуно блокирају

пут звука од извора до слушаоца. Поједини системи урачунавају и ефекте атмосферске апсропције (енг. "*atmospheric absorption*").

- *Доплеров ефекат* (енг. "*Doppler shifting*") такође може бити узет у обзир за релативни померај између извора звука и слушаоца.
- *Миксовање* (енг. *"mixing"*) представља процес контролисања релативне јачине између свих 2Д и 3Д звукова. Диктиран је уједно физиком и естетичким изборима инжињера звука.

8.2. Аудио сцена

Да би се звук репродуковао од стране аудио подсистема, потребно је прво описати сцену коју је потребно интерпретирати. Аудио сцена се обично састоји од следећих елемената:

- *3Д извори звука* (енг. "*3D sound source*"). Сваки 3Д звук се састоји од *монофоног* звучног сигнала емитованог из одређене позиције. Додатно, потребно је описати и *брзину, емисиони шаблон* (у коју страну се звук креће) и *опсег* (до ког растојања је могуће примити сигнал) звука.
- *Слушалац* (енг. "*listener*"). Слушалац представља „виртуелни микрофон" дефинисан *позицијом, брзином* и *оријентацијом*.
- *Модел окружења* (енг. "*environment model*"). Модел описује *геометрију* и *својства* објеката на сцени, и/или описује *акустична својства* простора кроз који се звук простире.

Позиције извора и слушаоца се користе при рачуну *слабљења на основу растојања*. *Емисиони шаблони* извора звука такође улазе у овај рачун. *Оријентација* слушаоца служи за рачун *угаоне позиције* звука који даље утиче на *померај* – релативан однос јачине међу звучницима у 5.1, односно 7.1 системима. Релативна *брзина* извора и слушаоца се битна за *Доплеров ефекат*. Коначно, *модел окружења* служи за прорачун акустике простора који се ослушкује и утицаја делимичне или поптуне блокаде од стране објеката у простору. За потребе испитивања који објекти су на путу од извора до слушаоца, може се, на пример, користити линијски упит о колизији.

8.3. Архитектура аудио подсистема

8.3.1. Преглед архитектуре

Попут већине рачунарских система, и аудио подсистем се састоји из слојевито организованих хардверских и софтверских компонената.

- *Хардвер* представља основу структуре, пружајући минималан скуп електронских кола која могу да да спроведу дигитални, односно аналогни излаз рачунара или играчке конзоле на пар слушалица, ТВ или кућни биоскоп. Аудио хардвер такође може убрзати софтверски слој пружајући кодеке, миксере, ефекте или синтисајзере у облику чипова. Овакав хардвер се често зове и *звучна картица* с обзиром да се у рачунарима реализује екстерним, плаг-ин (енг. "*plug-in*") уређајима (картицама).
- На рачунару, хардвер је обично енкапсулиран *драјверима* (енг. "*driver*"). Ово омогућује оперативном систему подршку за разне звучне карте и произвођаче карата.
- Додатно, и конзоле и рачунари обично енкапсулирају драјвере и хардвер у API ниског нивоа које ослобађа програмера главобоља познавања конкретног модела звучне карте.
- *3Д аудио подсистем* је изграђен над наведеним слојевима.

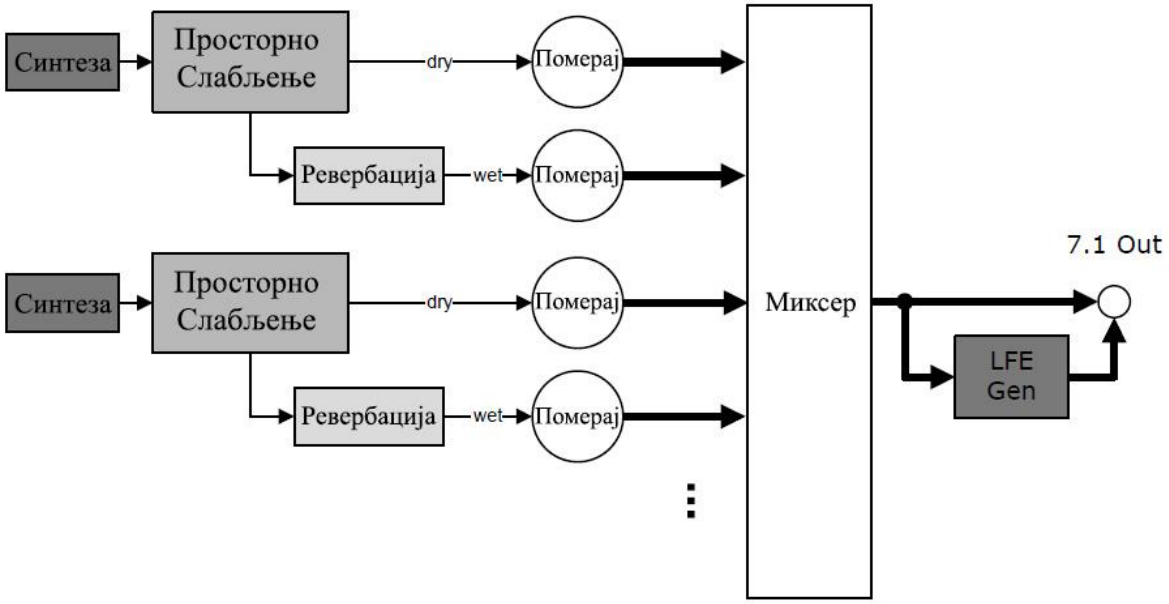
Скуп функционалности доступан програмеру од стране хардверско/софтверског аудио стека је представљен кроз модел *вишеканалне конзоле за миксовање* (енг. "*multi-channel mixing console*"). Задатак овако дефинисаног система је да прима вишеструке звучне сигнале као улаз, миксује их, подешава релативне јачине звука или додаје ефекте. Коначно, излазни сигнал се шаље на адекватан аудио уређај како би играч добио утисак 3Д звука.

8.3.2. Ток процесирања звука

Процес генерисања 3Д звука састоји се од неколико дискретизованих корака:

- За сваки 3Д звук, „чист“ (енг. “*dry*”) дигитални сигнал мора бити синтетизован (енг. “*synthesized*”).
- Ефекат слабљења са растојањем се примењује на сигнал како би се створио утисак удаљености од слушаоца, као и ревербациони (енг. “*reverb*”) ефекти. Ово ствара утисак *просторности* звука на слушаоца. Излаз је „процесирани“ (енг. “*wet*”) звучни сигнал.
- На чист и процесиран сигнал се, појединачно, примењује *померај* на један или више звучника, како би се створила коначна „слика“ сваког сигнала у тродимензионом простору.
- Померени вишеканални сигнали свих 3Д звукова се *миксују* у један вишеканални сигнал који се шаље у паралелну банку дигитално-аналогних конвертера (енг. “*Digital-to-Analog Converter – DAC*”) и појачала за излаз у аналогне звучнике или се шаље директно у дигитални излаз попут HDMI или S/PDIF.

Описани процес може се сматрати *током*. Како у пракси виртуелна сцена игре може садржати много звучних извора, вишеструки токови процесирања звука могу бити истовремено активни. Из овог разлога, у употреби је и назив *граф процесирања звука*. Процесирање звука заправо и представља граф међусобно увезаних компоненти који као излаз даје финални миксован, померен звучни сигнал.



Слика 36 - Ток процесирања звука у аудио подсистему

8.4. Управљање ресурсима

8.4.1. Аудио клипови

Најатомичнији облик звучног ресурса је *клип* (енг. "*clip*") – јединични дигитални ресурс са локалном временском линијом (енг. "*local timeline*"). Клипови су познати и као *звучни бафери* (енг. "*sound buffer*") зато што, у суштини, служе као складиште звучног податка. Аудио клип може да чува монофоне звучне податке (уобичајено за 3Д звуке) као и вишеканални звучни сигнал (уобичајено за 2Д звуке или стерео 3Д звуке) у било ком формату звучне датотеке подржане од стране СИИ.

8.4.2. Колекције звукова

Колекције звукова (енг. "*sound cues*") представљају више груписаних клипова уз мета информације (енг. "*metadata*") о томе како и када клипови требају бити репродуковани. Ово је уобичајен начин репродукције звука у СИИ. Дизајнери звука могу дефинисати колекције ван СИИ, тзв. *офлајн* (енг. "*offline*") алатима и предати их програмерима који потом могу репродуковати звукове као одговор на одређене догађаје у игри без бриге о детаљима енкапсулираним у колекцијама. Систем који прихвата колекције обично нуди и одређени API за њихову репродукцију. Уз колекцију се углавном достављају и информације попут типа звучног сигнала (2Д или 3Д), позиције и брзине звучног извора итд. Као резултат API враћа одређени вид показивача (не нужно показивача на меморијску адресу) како би програмер могао да контролише ток репродукције или обустави исту. Уобичајена употреба колекција звукова су поновљени клипови уз додатни, „завршни" звук. Примера ради, то може бити звук рафалне паљбе из ватреног оружја који се завршава одјеком (или падањем чаура на под).

8.4.3. Банке звукова

Подистем репродукције звука управља великом количином ресурса. Због меморијске сложености, није практично чувати све ресурсе у меморији. Такође, није оптимално ни гранулисати ресурсе на резолуцију клипова. Банке звукова (енг. "*sound banks*") складиште више клипова или колекција које се одједном учитавају у меморију. На пример, банка звукова за ниво А није потребна у нивоу Б и може се уклонити из меморије, и обрнуто. Поједине банке, попут оних за главног играча, остају константно у меморији јер су често потребне. Релокација банки у меморији може довести до *фрагментације*, те неки системи не дозвољавају ову операцију.

8.4.4. Стриминг

Поједини звучни ресурси могу бити превелики или предуги да би се чували у меморији. Важно је приметити да је, у било ком тренутку времена, потребан само један део клипа за репродукцију. Коришћењем цикличног бафера, део клипа се може превремено учитати (*баферовати*) и репродуковати док се истовремено бафер пуни следећим блоком бајтова. Докле год има података у баферу, звук ће бити репродукован неометано.

8.5. Миксовање

Репродуковање свих звукова из свих извора на сцени, са или без ефеката, у већини случајева неће довести до оптималног аудио искуства. Разлика између врхунског аудио доживљаја и *какофоније* је процес *миксовања* звука. У наставку ће укратко бити описане уобичајене технике миксовања у аудио подсистемима.

8.5.1. Груписање

Најосновнији вид миксовања је подешавање релативних јачина звукова. На пример, кораци би требало да се чују тише него звуци оружја. У акционим моментима, музика би требало да дође до изражаја док је дијалог примаран када се ситуација смири. Дакле, јавља се потреба да се одређене *групе* звукова динамички контролишу у зависности од ситуације. Софтверска имплементација не мора нужно бити груписање, већ се звуковима једноставно може доделити етикета којој групи припадају (музика, ефекат, оружје итд.). Аудио подсистем би, у том случају, требао да пружи начине контролисања јачине звука за сваку од група.

8.5.2. Привремено утишавање

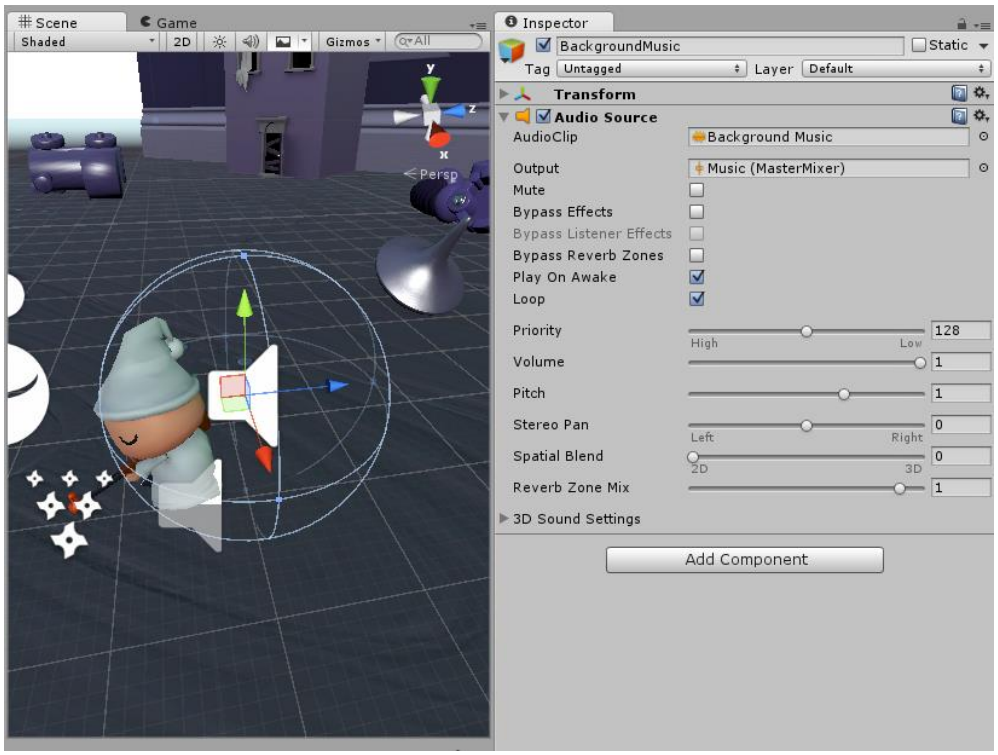
Привремено утишавање (енг. "*ducking*") је техника која омогућава да се амплитуда појединих звучних сигнала смањи како би други сигнали дошли до изражаја. На пример, при дијалогу између ликова потребно је смањити позадинску буку. Привремено утишавање се може окинути на више начина. Присуство неког звучног сигнала може аутоматски утишати један или више типова других сигнала. Окидање може бити и мануелно, у потпуности испрограмирано за специфичну ситуацију. Технике попут подешавања приоритета звуцима или колекцијама у сврхе контролисања јачине звука, такође су у употреби.

8.5.3. Ограничавање броја инстанци

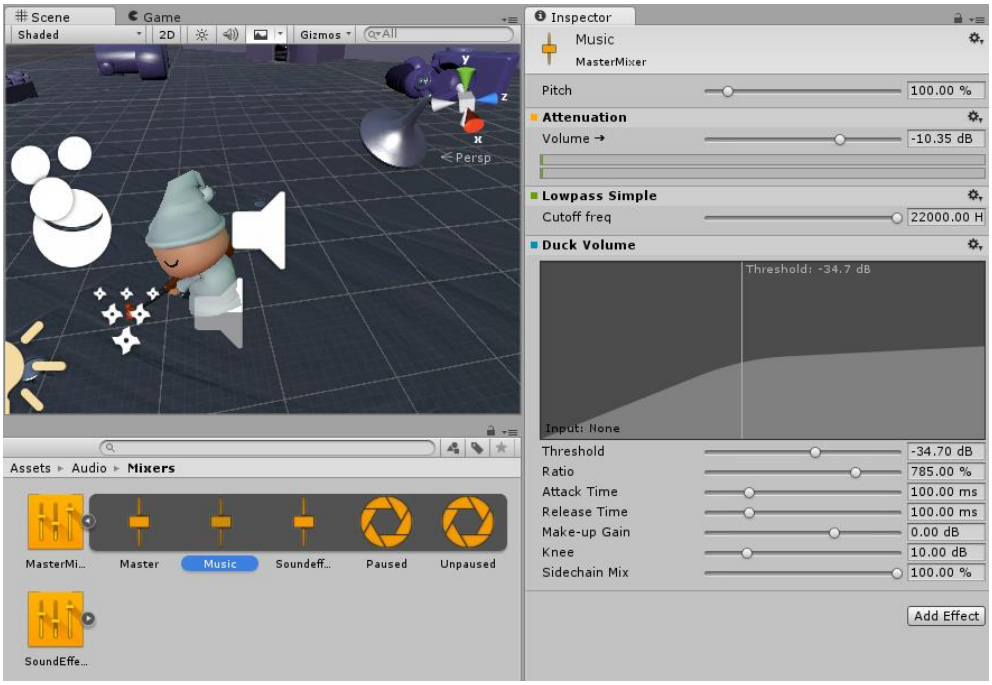
Често долази до ситуације да се више звукова репродукује у исто време. Уколико, на пример, 20 непријатеља пуца из својих оружја, репродуковање 20 понављајућих клипова пуцња истовремено доводи до какофоније. Као решење описаног проблема, користи се техника *ограничавања броја инстанци* (енг. "*instance limiting*") активних звучних сигнала. У претходном примеру, довољно би било користити две или три истовремене репродукције звука како би се дочарала вишеструка паљба из различитих праваца. Фиксиран број активних звукова може бити узрокован и хардвером (ограничења звучне карте) или меморијским, односно процесорским ограничењима софтвера. Додатно, ограничавање броја инстанци се може применити различито на различите групе звукова. На пример, највише пет оружја може да се чује или највише три лика могу да причају истовремено.

8.6. Пример имплементације

Подсистем репродукције звука у СИИ *Unity* описује аудио сцену компонентама *AudioSource* и *AudioListener* [24]. Да би аудио сцена функционисала коректно највише један *AudioListener* може бити активан на сцени (обично као компонента главне камере). Ова компонента не садржи никаква додатна својства. Компонента *AudioSource*, са друге стране, обилује подешавањима и дефинише који звучни клип ће бити репродукован, који ефекти ће бити примењени на сигнал, приоритет, померај и већину претходно описаних својстава звучних извора.



Слика 37 - Компонента *AudioSource* додељена "празном" објекту који представља позадинску музику у игри



Слика 38 - Панел за миксовање звука као део *AudioSource* компоненте

9. Подсистем анимација

Већина модерних 3Д игара је базирана на ликовима, обично у форми човека или човеколиких бића. Ликови представљају јединствен проблем јер би њихово кретање требало да буде флуидно и природно. Ово поставља нови скуп изазова у односу на анимације крутих тела попут аутомобила, лопте или робота. Задатак уношења „живота“ у ликове кроз реалистичне анимације припада компоненти СИИ под називом *подсистем анимација.*

9.1. Врсте анимације ликова

9.1.1. Спрајт анимација

Традиционалне анимације цртане руком су постизане брзим листањем слика како би се постигла илузија кретања. Целулоидна анимација (или скраћено *цел анимација*) је посебна техника где се само динамички делови цртају поново на целулоиду и постављају преко статичке позадине. Дигитални еквивалент ове технике је спрајт (енг. “*sprite’*) анимација. Индивидуалне сличице (енг. “*frames’*) се често слажу тако да се анимација може пуштати у петљи, наизглед, непрекидно (енг. “*looping animation’*). Пример такве анимације је анимација ходања.

Слика 39 - Пример спрајт анимације ходања лика

9.1.2. Крута хијерархијска анимација

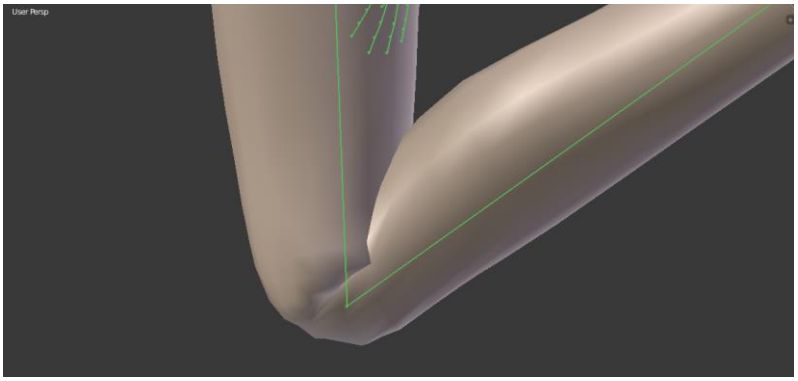
Појавом 3Д игара, спрајт анимације су постале застарео концепт. Међутим, ова техника се и даље користи за апроксимирање једноставних и/или удаљених објеката. Једна од најранијих техника развијених за анимирање 3Д модела је крута хијерархијска анимација (енг. “*rigid hierarchy animation’*). Модел карактера се дефинише хијерархијом крутих делова. Типичан пример овакве структуре је модел човека или човеколиког бића:

- Pelvis
  - Torso
    - UpperRightArm
      - LowerRightArm
        - RightHand
    - UpperLeftArm

- UpperLeftArm
  - LeftHand
- Head

- UpperRightLeg
  - LowerRightLeg
    - RightFoot
- UpperLeftLeg
  - UpperLeftLeg
    - LeftFoot

Приликом помераја десне надлактице, десна подлактица и шака се такође померају. Овакав вид анимације даје јако добре резултате за роботе или машине чији су делови заиста крути, али је визуелно проблематичан за симулацију бића сачињених од коже, костију и меких ткива. Проблем се јавља при савијању у зглобовима где би се, на пример, троуглови надлактице и подлактице одвојили или пресецали међусобно.

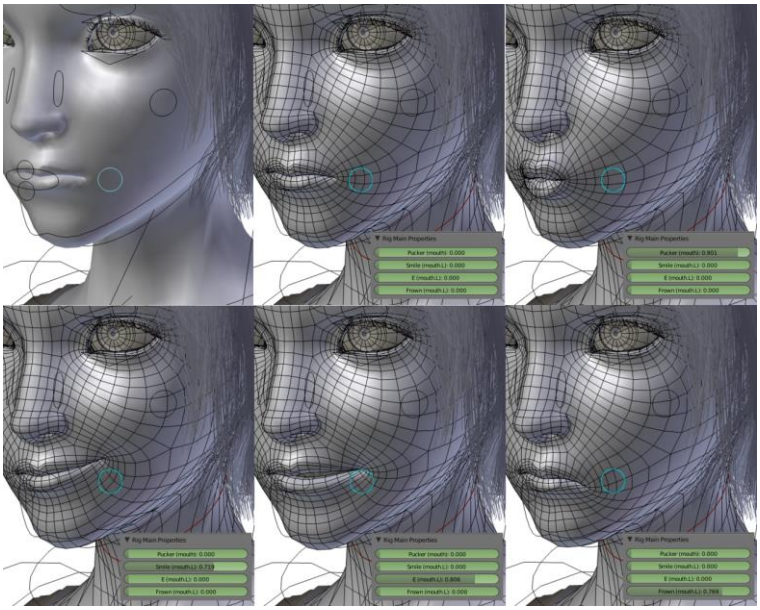


Слика 40 - Приликом савијања у зглобовима, може доћи до неприродних ситуација

9.1.3. Вертекс анимација

Недостатак природности круте анимације може се надоместити анимирањем појединачних вертекса модела. Овакав приступ може реализовати било коју замисливу деформацију модела (уз одговарајући ниво теселације) али количина података која би се морала чувати за сваки вертекс чини технику непрактичном за игре у реалном времену.

Варијација технике подразумева дефинисање неколико кључних поза померањем вертекса модела од стране аниматора. Подсистем анимација, потом, анимира прелазе између дефинисаних кључних поза. За прелазе се, углавном, користи линеарна интерполација (енг. "*linear interpolation – LERP*"). Овакав вид анимације је уобичајен за изразе лица карактера због комплексности и нивоа детаља који мора бити дочаран (у изразима људског лица учествује преко 50 мишића).

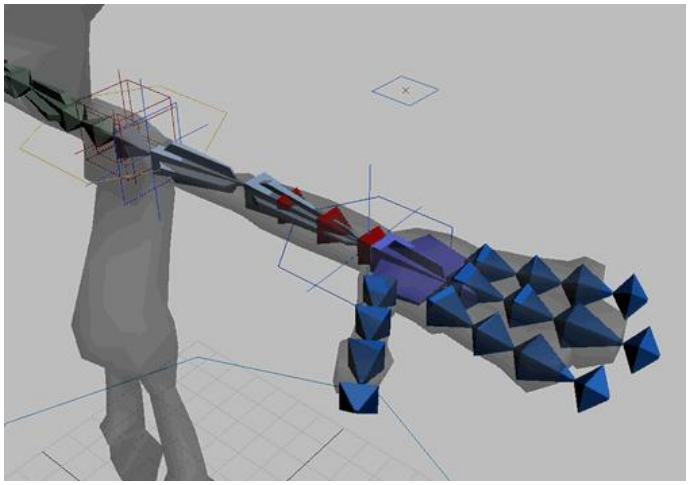


Слика 41 - Анимација лица карактера коришћењем четири кључне позе

9.1.4. Скелетална анимација

Напредовањем хардвера, развијане су и напредније технике анимације. Скелетална анимација комбинује предности које доноси вертекс анимација у погледу анимирања индивидуалних троуглова и меморијску ефикасност круте анимације. Симулације тканине и коже/ткива су честе употребе ове технике (отуда и енглески назив „*skinned animation*”).

Скелет модела карактера је дефинисан виртуелним костима које су хијерархијски организоване. Ове компоненте нису видљиве у самој анимацији. Скелет је „пресвучен“ континуалном мрежом троуглова а вертекси су анимирани заједно са костима. Један вертекс може бити под утицајем једног или више зглобова са одређеном тежином, као што је већ описано раније у делу о графичком подсистему.



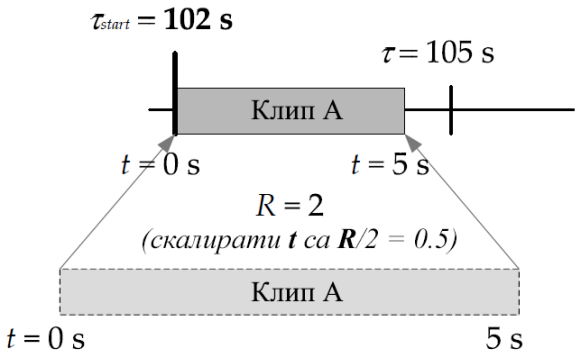
Слика 42 - Пример скелетног модела у софтверу *3D Studio Max*

9.2. Анимациони клипови

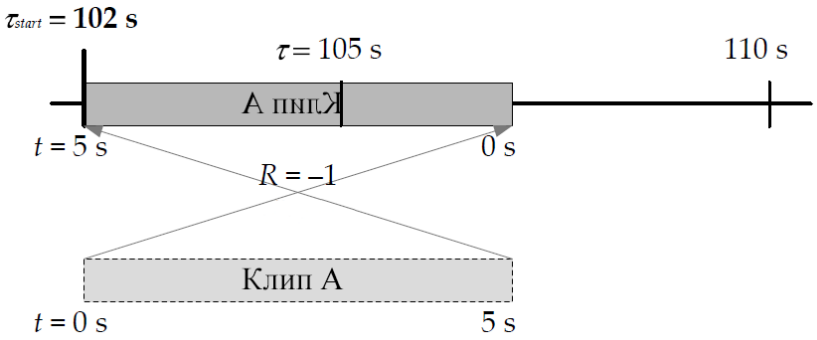
Анимација у играма је категорија за себе јер обично не прати исти шаблон попут анимације на филму. Ток анимације обично није предвидив и зависи од акција самог играча. Ово важи чак и за неигриве ликове, који су контролисани од стране рачунара, но ипак и њихове акције у многоме зависе од потеза играча. С тога, дуге, континуалне анимације обично нису практичан концепт у играма. Понашање лика је дефинисано серијом краћих *анимационих клипова* (у наставку ће се термини *анимација* и *анимациони клип* користити равноправно), попут ходања, трчања или скока. Једини изузетак овог правила су неинтерактивне секвенце (енг. "*In-game Cinematic*") чија је примарна сврха приповедање приче саме игре. Додатно, све чешће је присутан и хибрид оба типа анимације где играч мора брзо да одреагује на захтев (обично притиском неког тастера) како би се секвенца наставила (енг. *"Quick-time Event"*).

9.2.1. Локално и глобално време

Сваки клип се може представити својим *локалним временом*, обично представљен децималним бројем који означава временски индекс изражен у *секундама*, *фрејмовима* или *нормализованим веменским јединицама* (у ком случају се индекс зове *фаза*). На почетку анимације локални временски индекс ***t*** углавном има вредност 0. Репродукција анимације унапред се врши напредовањем свих клипова по локалној временској линији. Уколико се користи нејединична брзина репродукције ***R***, потребно је скалирати временски индекс овом вредношћу. Вредност параметра ***R*** може бити и негативна, што резултује репродукцијом анимације уназад.



Слика 43 - Репродукција анимације дуплом брзином (*R = 2*) одговара скалирању локалне временске линије *t* фактором 0.5



Слика 44- Репродукција анимације уназад одговара скалирању локалне временске линије *t* негативним фактором

Аналогно, сваки клип се може представити и *глобалним временом*. Глобално време игре почиње да тече приликом неког иницијалног догађаја, попут учитавања нивоа или стварања играча у простору игре. Идеја је мапирати почетак анимације за фиксну тачку на глобалној временској линији, која ће бити означена глобалним временским индексом ***T***, обично изражен у секундама. Локално време клипа се лако може израчунати формулом:

$$t = \left(clamp\left([(T - T_{start})R] \middle|_{0}^{NT} \right) \right) mod\ T$$

где је:

- T_{start} временски индекс почетка анимације изражен у глобалном времену
- R брзина репродукције
- N број понављања анимације

9.2.2. Синхронизација

Како клипови изражени локалним временом углавном почињу од временског индекса t = 0, да би се два клипа синхронизовала, потребно их је репродуковати истовремено. Ово, наизглед тривијално, решење често није оствариво у пракси. Проблем лежи у томе што је клипове могуће репродуковати из незавнисних подсистема СИИ. Примера ради, играч, из подсистема логике, може иницирати ударац ка противнику у ком моменту ће се репродуковати анимација ударца. Реакција противника, контролисана од стране подсистема В.И., може доћи тек у наредном циклусу ажурирања у ком случају долази до кашњења од једног фрејма. Обрнут проблем је такође могућ.

Синхронизација анимација преко глобалног времена решава многе аспекте наведеног проблема. Уколико се глобална времена играча поклапају, анимације ће бити репродуковане синхронизовано. Чак и ако то није случај, могуће је дефинисати одвојену, *мастер* временску линију по којој се анимације свих играча морају синхронизовати.

9.2.3. Комбиновање анимација

Често предефинисани скуп анимационих клипова није довољан да покрије комплетан опсег покрета и варијација покрета неког лика. Техника *комбиновања анимација* (енг. *"Animation Blending"*) пружа могућност две или више позе, резултујући потпуно новим скупом анимација. На пример, анимација ходања рањеног противника се може комбиновати са анимацијом нормалног хода како би се добили различити степени повреде. Технике комбиновања такође имају велику примену у анимацијама израза лица и/или говора ликова.

9.2.4. Комбиновање линеарном интерполацијом

Потребно је добити позу лика (дефинисаног скелетом са N зглобова) која представља комбинацију две позе дефинисане редом са $P_A^{skel} = \{(P_A)_j\}\Big|_{j=0}^{N-1}$ и $P_B^{skel} = \{(P_B)_j\}\Big|_{j=0}^{N-1}$. Ово се може остварити *линеарним интерполирањем* (енг. *"Linear Interpolation – LERP"*) сваког зглоба између два екстрема дефинисаних позама A и B:

$$(P_{LERP})_j = LERP\big((P_A)_j, (P_B)_j, \beta\big) = (1-\beta)(P_A)_j + \beta(P_B)_j$$

Интерполирана поза целог скелета је скуп свих интерполираних зглобова:

$$P_{LERP}^{skel} = \{(P_{LERP})_j\}\Big|_{j=0}^{N-1}$$

У наведеним једначинама параметар β проценат (или фактор) комбиновања. Вредност $\beta = 0$ означава да ће коначна поза бити идентична пози A, док $\beta = 1$ означава да ће коначна поза бити идетнична пози B. Све вредности параметра β између 0 и 1 резултују комбинованом позом између A и B.

Интепролација позиција зглобова подразумева интерполације 4x4 матрица. Међутим, директна линеарна интерполација матрица није практична. Ово је један од разлога зашто се позиције зглобова представљају SQT (енг. *"Scale, Quaternion, Translation"*) форматом. Како су све три компоненте вектори, интерполација се своди на интерполацију вектора.

9.3. Архитектура подсистема анимација

9.3.1. Преглед архитектуре

Већина подсистема анимација се генерално састоји из три слоја:

- Ток анимације (енг. *"Animation pipeline"*). За сваки анимирани објекат у игри, ток узима један или више анимационих клипова и миксује их како би се добила *локална поза* скелета. Додатно, рачуна се и *глобална поза*

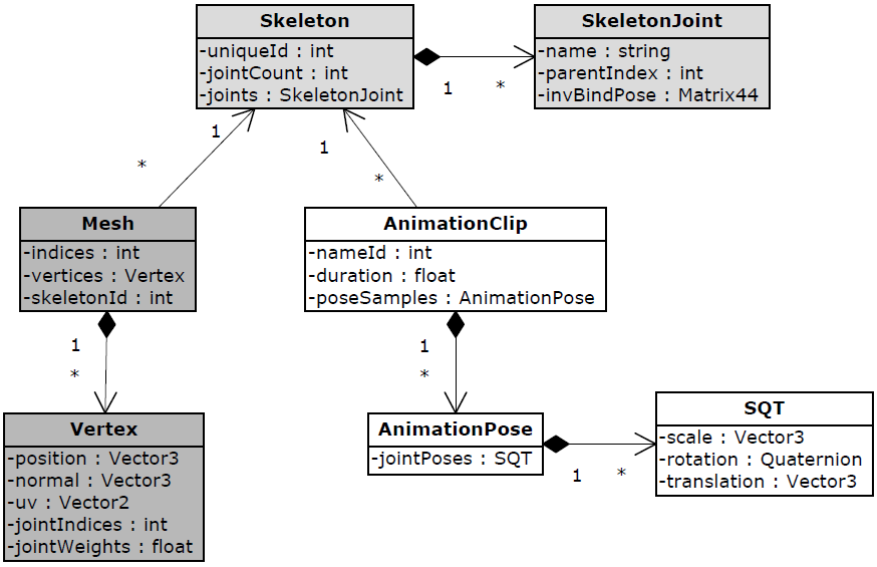
скелета и као палета *скининг матрица* (енг. *"skinning matrix"*) за коришћење од стране графичког система при исцртавању објекта. Ток такође открива могућност за додатна процесирања попут инверзне кинематике, физике тела у слободном паду (енг. "*ragdoll"*) и других типова *процедуралних анимација*.

- Машина стања акција (енг. *"Action state machine – ASM"*). Акције ликова (стој, ходање, трчање, скакање итд.) се најбоље представљају коначном машином стања. Подсистем ASM је део скоро сваког озбиљнијег СИИ и пружа стањем вођен интерфејс који омогућава глатке прелазе између различитих стања. Додатно, већина подсистема анимација омогућава да различити делови модела ликова прелазе у различита стања истовремено. На пример, лик може нишанити и пуцати док уједно трчи. Овакво решење је могуће реализовати вишеструким машинама стања које управљају анимацијама једног лика кроз више *слојева стања* (енг. *"state layers"*).
- Контролери анимације (енг. *"Animation controllers"*). Понашања игривих и неигривих ликова су обично контролисана од стране система вишег нивоа под називом *контолери анимације*. Више контролера може управљати једним ликом. На пример, један контролер може управљати понашањем док је лик у заклону, док други управља понашањем приликом пуцања и/или кретања. Контролери анимације енкапсулирају логику управљања анимацијама како се не би мешала са остатком подсистема логике игре.

9.3.2. Структуре података

Попут већине подсистема СИИ, и овде је потребно направити јасну разлику између *дељених ресурса* и *ресурса везаних за конкретну инстанцу*. Сваки индивидуални лик у игри има своје структуре података али слични ликови често и деле податке између себе. Дељени подаци обично укључују неке од следећих ставки:

- *Скелет*. Скелет описује хијерархију компоненти и позу модела.
- *„Кожа"* (енг. *"Skinned mesh"*). Мрежа троуглова која може бити везана за један или више скелета. Сваки вертекс модела садржи индекс(е) придруженог зглоба и информацију у којој мери (тежини) тај зглоб утиче на позицију вертекса.
- *Анимациони клипови*. Више стотина анимационих клипова може бити направљено за конкретан скелет. Те анимације могу бити целовите (покривају комплетно „тело") или парцијалне.



Слика 45 - Пример структуре података за анимацију дат УМЛ дијаграмом

Творци игара се обично труде да минимизују број коришћених скелета, јер сваки нови скелет захтева комплетан сет нових анимационих клипова. Идеја је искористити исти скелет за различите моделе ликове који су слични по анатомији. Напретком технологије анимирања, могуће је и искористити скуп анимационих клипова дизајнираних за један скелет на неком другом скелету. Овај концепт је познат као *техника премапирања* (енг. *"Retargeting technique"*). Уколико су два скелета морфолошки идентична, техника премапирања се своди да ажурирање зглобних индекса за вертексе модела. У супротном, проблем постаје доста комплекснији [25].

У већини игара, могуће је да се више инстанци једног лика појављује истовремено. Свака инстанца тог лика захтева свој приватни скуп података о анимацији. Наравно, не постоји универзално решење организације ових података али генерално сваки подсистем анимација прати следеће информације:

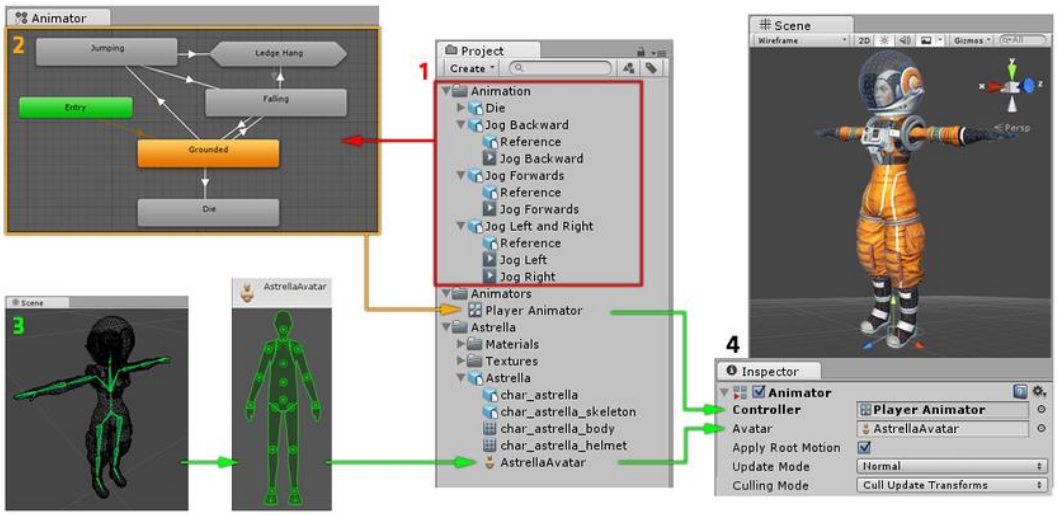
- *Стање клипа*. За сваки клип прати се његово *локално или глобално време* из ког се рачуна тренутно поза лика. Додатно, *брзина репродукције* клипа се узима у обзир.
- *Комбиновање анимација* (енг. *"Animation Blending"*). Комбиновање анимација дефинише на који начин се анимациони клипови спајају заједно у датом тренутку на временској линији. Степен утицаја сваког клипа на коначну позу лика је дат *тежинама комбиновања* (енг. *"blend weight"*). Две примарне методе комбиновања анимација су:
  - *Пондерисано комбиновање* (енг. "*Flat weighted average blend*"). Потребно је одржавати списак свих активних клипова позитивне тежине за конкретан скелет у датом тренутку. Коначна поза је одређена пондерисаном тежином свих клипова.
  - *Стабло комбиновања* (енг. "*Blend trees*"). Дрволика структура одражава *синтаксно стабло* где су операнди клипови а операције

су различити типови линеарне интерполације који се извршавају над њима. Класичан пример је тзв. бинарна линеарна интерполација (енг. *"Binary LERP Blend"*) где се коначна поза лика добија интерполирањем између две различите анимације у датом тренутку. Тежине анимација одређују њихов процентуални учинак у коначној пози.

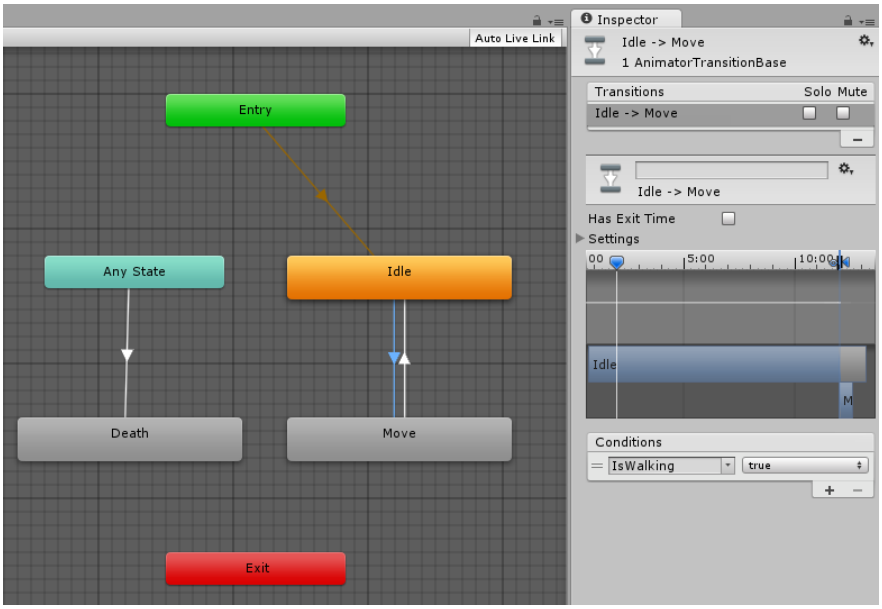
- *Локална поза*. Често имплементирано као низ SQT структура, по једна за сваки од зглобова, чувајући коначну позу лика у простору релативном родитељу.
- *Глобална поза*. Низ SQT структура или 4x4, односно 4x3 матрица, по једна за сваки од зглобова. Чува коначну позу лика у простору модела или простору света.
- *Палета матрица*. Низ 4x4, односно 4x3 матрица, по једна за сваки од зглобова. Садржи скининг матрице које су потребне графичком подсистему за исцртавање коначне позе лика.

9.4. Пример имплементације

У *Unity* СИИ објекти се анимирају посредством компоненте *Animator* [26]. Сама компонента не носи много својстава али садржи референцу на *Animator Controller* који представља машину стања за контролисање анимационих клипова, њихово комбиновање и ефекте транзиције. За човеколике ликове, Unity нуди могућност ремапирања анимације путем апстракције скелеталне структуре лика – *Avatara*. *Avatar* омогућава примену истог скупа анимација на више човеколиких ликова у игри.



Слика 46 - Преглед подсистема анимације у *Unity* СИИ




```
void Animating(float h, float v)
{
        Animator anim = GetComponent<Animator>();

        /* Provera horizontalnog/vertikalnog unosa korisnika */
        bool walking = h != 0f || v != 0f;

        /* Ukoliko unos postoji, animiramo kretanje lika */
        anim.SetBool("IsWalking", walking);
}
```

Слика 47 - Транзиција прелаза између стања *Idle* и *Move* контролисана је променљивом *IsWalking*

10. Закључак

Овим радом описани су основни концепти у архитектури софтвера за имплементацију игра. Првенствено су истакнути кључни подсистеми, попут графичког и аудио подсистема као и подсистема за израчунавање физике и анимација. Посебан акценат стављен је на архитекуру базирану на компонентама. Примери везани за подсистеме логике и графике су дати кроз имплементацију СИИ као дела овог рада, док су примери везани за подсистеме физичких прорачуна, звука и анимација дати кроз примере из *Unity*-а.

Видео игре данас имају значајан удео у индустрије забаве и као такве су у непрестаном развоју, константно померајући границе софтвера и хардвера. Импресивна, фотореалистична графика и анимације, реална физика, паметни системи вештачке интелигенције као и виртуелна реалност су неки од праваца којима се тежи. Сходно томе, и софтвер за имплементацију игара се непрекидно морају развијати и напредовати како би ишли у корак са захтевима модерне индустрије видео игара.

Као део овог рада, пружена је и имплементација једноставног 3Д СИИ базираног на компонентама у складу са описаном теоријском основом, као и по узору на постојеће системе. Кôд је написан у програмском језику C++, користећи *Qt* библиотеке (*QtCore, QtGUI*) и *OpenGL* графички API. Софтвер је развијен и тестиран на *Windows* платформи у оквиру *Visual Studio 2013* окружења али се, с обзиром на вишеплатформске могућности наведених технологија, може преводити и извршавати на *Linux* и *OSX* оперативним системима.

10.1. Наставак развоја

Имати у виду да је архитектура професионалних СИИ у већој мери сложенија од приложене имплементације. У тренутку писања рада имплементација садржи логички и графички подсистем као и подсистем за управљање уносом корисника (играча). Наставак развоја би подразумевао првенствено имплементацију остатка кључних подсистема (физика, аудио, анимација) као и апстрактнију реализацију управљања ресурсима. Напреднији концепти попут паралелног извршавања подсистема могу бити посебно занимљиви и програмерски изазовни [23]. Коначно, кориснички интерфејс за рад са СИИ је модул од велике користи који би био изграђен над базном библиотеком, носећи са собом нови скуп изазова.

11. Литература

[1] "id Software," [Online]. Available: http://www.idsoftware.com/. [Accessed 2016].

[2] "Unreal Engine," [Online]. Available: https://www.unrealengine.com/. [Accessed 2016].

[3] Crytek, "Introducing CRYENGINE V: A letter to our Community," 2016. [Online]. Available: https://www.cryengine.com/news/introducing-cryengine-v-a-letter-to-our-community. [Accessed 2016].

[4] "Unity 3D," [Online]. Available: https://unity3d.com/. [Accessed 2016].

[5] C++ Reference, "Static Assertion," [Online]. Available: http://en.cppreference.com/w/cpp/language/static_assert. [Accessed 2016].

[6] Qt Docs, "Signals and Slots," [Online]. Available: http://doc.qt.io/qt-5/signalsandslots.html. [Accessed 2016].

[7] Qt Docs, "The Meta-Object System," [Online]. Available: http://doc.qt.io/qt-5/metaobjects.html. [Accessed 2016].

[8] Qt Docs, "QWindow," [Online]. Available: http://doc.qt.io/qt-5/qwindow.html. [Accessed 2016].

[9] Wikipedia, "Transformation Matrix," [Online]. Available: https://en.wikipedia.org/wiki/Transformation_matrix.

[10] Graphics, Cornell University Program of Computer, "The Cornell Box," [Online]. Available: http://www.graphics.cornell.edu/online/box/.

[11] S. Owen, "Visible Surface Determination: Painter's Algorithm," 1998. [Online]. Available: https://www.siggraph.org/education/materials/HyperGraph/scanline/visibility/painter.htm.

[12] D. Rogers, "W-Buffering in Direct3D," [Online]. Available: http://mathinfo.univ-reims.fr/IMG/pdf/W-buffering_in_Direct3D_Rogers_.pdf.

[13] N. Reed, "Depth Precision Visualized," 2015. [Online]. Available: https://developer.nvidia.com/content/depth-precision-visualized.

[14] B. Kemen, "Maximizing Depth Buffer Range and Precision," 2012. [Online]. Available: http://outerra.blogspot.rs/2012/11/maximizing-depth-buffer-range-and.html.

[15] Wikipedia, "Convex Hull," [Online]. Available: https://en.wikipedia.org/wiki/Convex_hull. [Accessed 2016].

[16] Wikipedia, "Polyhedron," [Online]. Available: https://en.wikipedia.org/wiki/Polyhedron. [Accessed 2016].

[17] D. Eberly, "Intersection of Convex Objects: The Method of Separating Axes," 2001. [Online]. Available: https://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf. [Accessed 2016].

[18] P. Lindemann, "The Gilbert-Johnson-Keerthi Distance Algorithm," [Online]. Available: https://www.medien.ifi.lmu.de/lehre/ss10/ps/Ausarbeitung_Beispiel.pdf. [Accessed 2016].

[19] Wikipedia, "Minkowski Addition," [Online]. Available: https://en.wikipedia.org/wiki/Minkowski_addition. [Accessed 2016].

[20] Molly Rocket, "Implementing GJK," [Online]. Available: https://mollyrocket.com/849. [Accessed 2016].

[21] "Octree Partitioning Techniques," 1997. [Online]. Available: http://www.gamasutra.com/view/feature/131625/octree_partitioning_techniques.php.

[22] Unity Docs, "Colliders," [Online]. Available: https://docs.unity3d.com/Manual/CollidersOverview.html. [Accessed 2016].

[23] J. Gregory, Game Engine Architecture, Second Edition, 2015.

[24] Unity Docs, "Audio Source," [Online]. Available: https://docs.unity3d.com/Manual/class-AudioSource.html. [Accessed 2016].

[25] M. Gleicher, "Retargetting Motion to New Characters," 1999. [Online]. Available: http://davis.wpi.edu/~matt/courses/retarget/.

[26] Unity Docs, "Animator," [Online]. Available: https://docs.unity3d.com/Manual/class-Animator.html. [Accessed 2016].

[27] N. Porter, Component-based game object system, 2012.

[28] A. Sherrod, Data Structures and Algorithms for Game Developers.