УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Александар Д. Ранковић

ИМПЛЕМЕНТАЦИЈА 2D ПОГОНА ЗА РАЗВОЈ ВИДЕО ИГАРА

мастер рад

Београд, 2022.

Ментор:

др Александар Картељ, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Предраг Јаничић, редовни професор
Универзитет у Београду, Математички факултет

др Иван Чукић, доцент
Универзитет у Београду, Математички факултет

Датум одбране:

Хвала Јану Черникову, Анђелки Миловановић, Илији Ранковићу и Александри Никшић

Наслов мастер рада: Имплементација 2D погона за развој видео игара

Резиме: Циљ овог рада је имплементирање 2D погона за видео игре (енг. *2D game engine*) у програмском језику C++. То укључује развој и повезивање различитих система попут „апликације", слојева „апликације", система за 2D исцртавање, система ентитета и компоненти и других мање битних система. У циљу демонстрације функционалности погона реализује се једна конкретна видео игра у њему и демонстрира се начин њеног коришћења. У раду је приказан детаљни осврт на функционалности погона за видео игре које могу да се развију у будућности као надоградња на овај пројекат.

Кључне речи: Видео игре, погон за видео игре

Садржај

Глава 1

Увод

Овај рад приказује шта је све потребно да се направи 2D погон за видео игре. На почетку рада је истражено питање шта је 2D погон за видео игре, како су настали први погони и шта су неке од функционалности које их одликују. Затим се приказују најбитније информације везане за имплементацију предложеног погона. Након тога се представља видео игра која је направљена коришћењем њега. То има два циља, један је тестирање предложеног погона, а други циљ је уопштени приказ развоја видео игара.

Мотивација за овај рад је била жеља да се истражи колико је захтевна имплементација 2D погона за видео игре и које области су најбитније за ту имплементацију. Стандардни погони за видео игре најчешће имају могућности развијања и 2D и 3D видео игара. 2D погон представља поједностављену верзију погона који може да развија искључиво 2D видео игре, то га чини приступачнијим за разумевање. Овај рад може да се користи као увод у дисциплину развоја видео игара. Рад покрива већину информација потребних почетницима који желе тиме да се баве.

Глава 2

Погони за видео игре

2.1 Историјат видео игара

Видео игре се јављају у најразличитијим облицима. Већина игара (попут игре шах на пример) поседује интерактивне изазове, прецизно дефинисана правила и циљ којем играч тежи. За видео игре је специфично да кроз интеракцију са корисником и аудио-визуелном имерзијом дозвољавају играчу да реагује на ситуације у виртуалном окружењу [1].

Видео игра *Spacewar*^{TM} коју су развили Стив Расел и студенти са Масачусетског технолошког института 1961. године се сматра за први покушај стварања нове индустрије видео игара [1].

Након тога већина видео игара је прављена за аркадне машине (енг. *arcade machine*) или кућне рачунаре. *Pac-Man*® и *Donkey Kong*® су најпознатије игре из тог периода. Оне су уздигле индустрију видео игара на виши ниво. *Nintendo* 1985. године избацује на тржиште нови систем *Nintendo Entertainment System*® - *NES*® и видео игру за њега *Super Mario Bros*® која је постала најпродаванија видео игра у то време. То је започело такмичење конзола за превласт на тржишту [1].

У исто време су видео игре за кућне рачунаре биле популарне. Пуцачка игра (енг. *First-person shooter* - *FPS*) *Doom* пролази комерционално успешно 1993. године, а заједница модификатора (енг. *mod community*) је креирала сопствене нивое за њу и делила их преко интернета што је помогло да игра буде још успешнија [1].

Слика 2.1: *Pong* једна од првих видео игара (горе), модерна видео игра *Red Dead Redemption 2 (доле)*

Видео игра *Quake* популаризује играње видео игара преко интернета са другим играчима 1996. године. Тај тренд је довео до видео игре *World of Warcraft*® (објављена 2004. године) која је најпознатија масовна вишекорисничка мрежна игра играња улога (*Massive Multiplayer Online Role Playing Game - MMORPG*). Такође су се развиле разне врсте *MOBA* (*Massive Online*

Battle Arena) и *Battle Royale* игара које тренутно доминирају индустријом [1].

Слика 2.1 упоређује једну од првих видео игара са модерном видео игром. Сваке године индустрија видео игара се побољшава. Тешко је предвидети како ће видео игре изгледати кроз 50 година.

2.2 Појам погон за видео игре

Израз погон за видео игре (енг. *Game Engine*) је настао средином деведесетих због пуцачких игара попут видео игре *Doom* коју је развио *id Software*. Архитектура игре *Doom* је јасно раздвојила најбитније софтверске системе (систем за исцртавање графике, систем за детекцију колизија и аудио систем) од текстура, осмишљених нивоа и правила игре. То раздвајање је помогло програмерима да праве нове игре тако што додају нова оружија, карактере, сценарија, а притом језгро програма (погон) остаје непромењено. Програмери су временом почели да продају право на коришћење погона за игре које су развили. У данашње време програмери често купују право на коришћење погона за игре јер је профитабилније у односу на креирање целог погона [5].

Погон за видео игре програмерима дозвољава да се фокусирају на писање саме логике видео игре, користећи ефикасне алате и функционалности погона на које би иначе потрошили доста времена.

2.3 Преглед значајнијих погона за видео игре

Постоје јавно доступни (енг. *Open Source*) погони за видео игре, и приватни (енг. *In-house*) које користе само компаније које су их развиле. Јавно доступни погони за видео игре могу и не морају да буду бесплатни за коришћење.

Unity погон за видео игре је издат 2005. године за *Mac* уређаје, а данас је један од најпознатијих погона за видео игре. *Unity* поседује напредно 3D графичко исцртавање и бесплатно извози игре за мобилне уређаје. Најпознатије игре направљене коришћењем њега су: *Among Us, Fall Guys: Ultimate Knockout, Untitled Goose Game, Cuphead* и *Pokemon Go* [2].

Unreal engine (Epic Games) је направљен 1998. године. Назван је по првој видео игри која је развијена његовим коришћењем. *Unreal engine 4* издат 2012. године је тренутно један од најкоришћенијих јавних погона за видео игре. Бесплатно може да се користи за учење и пројекте чија је зарада

испод одређене границе. Најпознатије игре направљене коришћењем њега су: *ARK: Survival Evolved, Fortnite, PlayerUnknown's Battlegrounds, Sea of Thieves, Batman: Arkham City* и *Star Wars Jedi: Fallen Order* [2].

GameMaker Studio (YoYo Games) направљен 1999. године је погон за видео игре предвиђен за коришћење од стране почетника, због тога што не захтева познавање програмских језика. Најчешће се користи за прављење 2D игара, али има и 3D функционалности. Најпознатија игра направљена коришћењем њега је *Undertale* [2].

Велике компаније често имају своје приватне погоне за развој видео игара. Тиму програмера могу да буду потребне и године да би се лиценцирани погон опремио за прављење конкретне игре (компаније често имају франшизу игара по којој су познате). Великим компанијама је често ефикасније да надограде раније верзије постојећих погона или да изграде читав погон.

Frostbite је погон за видео игре који је развијен за више платформи попут *Microsoft Windows* и конзоле новијих генерација. Првобитно је био коришћен за серијал видео игара *Battlefield*, касније је његова употреба проширена и на друге пуцачке игре, као и на игре које припадају разним жанровима. *Frostbite* је до сада коришћен ексклузивно за видео игре које издаје компанија *Electronic Arts*.

Creation Engine је 3D погон за видео игре који је створио *Bethesda* студио за видео игре, на основу *Gamebryo* погона. Уз помоћ њега су направљене многе игре играња улога (енг. *role-playing games*) попут *The Elder Scrolls V: Skyrim, Fallout 4, Fallout 76*.

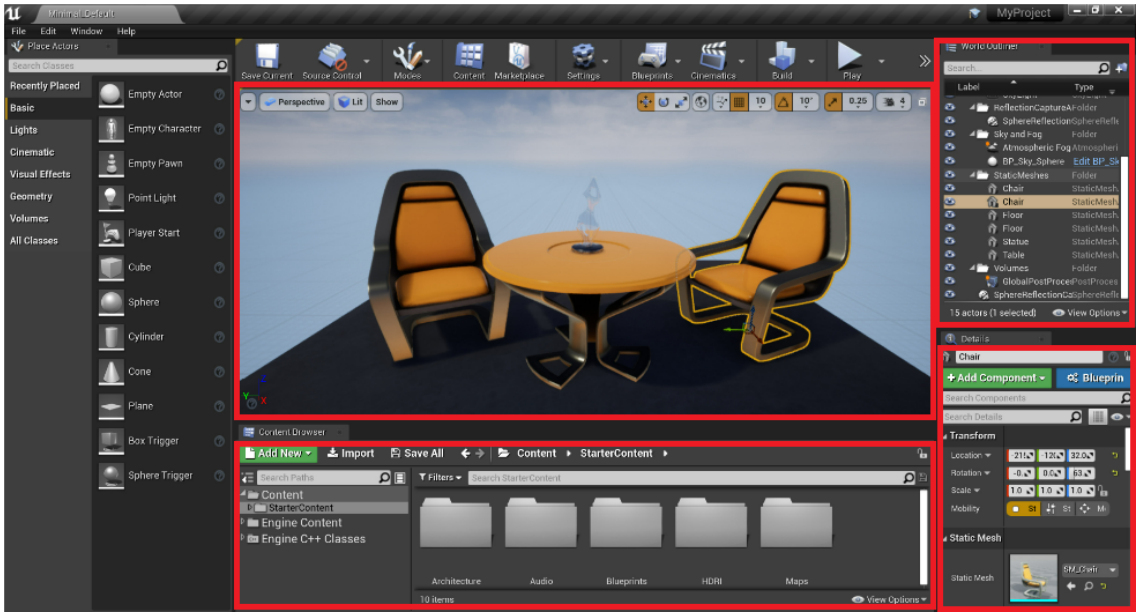
IW погон за видео игре је развио студио *Infinity Ward* за *Call of Duty* серијал видео игара.

2.4 Елементи погона за видео игре

Велики део елемената који сачињавају погоне је обрађен у оквиру пројекта и биће речи о њима у глави 3. Ово поглавље представља осврт на делове погона који су битни за развој видео игара, али нису имплементирани у оквиру пројекта. *Unreal engine* поседује поменуте елементе, а у наставку ће опис њихове имплементације послужити за разумевање тих елемената на практичним примерима.

Графички кориснички интерфејс

Погони за видео игре *Unity* и *Unreal engine* имају графички кориснички интерфејс који помаже програмерима и људима мање стручним за програмирање (дизајнери, уметници итд.) да лакше развијају видео игре. Графички кориснички интерфејс у погонима за видео игре се обично назива *Editor* (на пример *Unreal Editor*). Најбитнији елементи графичког корисничког интерфејса су: Хијерархија елемената видео игре (енг. *Scene hierarchy*), Прозор за едитовање света (енг. *World Editor*), Претраживач садржаја (енг. *Content Browser*) и Панел са карактеристикама (енг. *Details panel*). На слици 2.2 је приказано како изгледају ти елементи за *Unreal editor*.



Слика 2.2: Приказ Графичког корисничког интерфијса за *Unreal engine* - *Unreal editor*

Хијерархија елемената видео игре је панел који представља све ентитете који постоје у видео игри (на слици 2.2 се налази у горњем десном углу). Такође приказује релације између ентитета. Постоје родитељ-дете односи између ентитета, на пример у аутомобилу каросерија би била родитељ ентитет а точкови деца ентитети. Овај панел такође служи за додавање и брисање ентитета и друге начине манипулисања њима.

Прозор за едитовање света је панел који у себи садржи приказ света у коме се налази видео игра, на слици 2.2 је позициониран у средини екрана. Има своју камеру (*Editor camera*) која може да се помера преко миша и тастатуре и тако позиционира где је потребна за рад на различитим деловима нивоа. Ентитети могу да се превуку са неког другог панела у овај прозор или на неки други начин направе. Ако се налазе у прозору могу да се померају, повећавају, смањују, ротирају и на било који други начин мењају. Манипулисање ентитетом у прозору је омогућено уз помоћ *Gizmo* система, када се кликне на ентитет појави се координанти систем у центру ентитета који може да се користи за наведене акције над њим. Осим нормалног изгледа 3D света видео игре, може да се измени пројекција камере да показује ортогоналну пројекцију света са свих страна (од горе, са лева итд.).

Претраживач садржаја је панел који представља систем фолдера и фајлова који су везани за видео игру која се прави, на слици 2.2 се налази испод прозора за едитовање света. Ресурси попут текстура, материјала, фонтова, икона, аудио фајлова, скрипти и други се могу наћи у претраживачу садржаја. Из њега ресурси могу да се превуку на ентитете на које желимо да их поставимо. Модерни погони за видео игре пружају велики број ресурса који помажу корисницима да уче или стварају једноставније видео игре, и они могу да се нађу у претраживачу садржаја.

Панел са карактеристикама приказује детаље о ентитету који је тренутно селектован, на слици 2.2 се налази у доњем десном углу. Све компоненте које Ентитет поседује су приказане у овом панелу. Приказ сваке компоненте је другачији, зато што се све компоненте разликују. Овај панел приказује компоненте али такође дозвољава да се оне мењају. На пример *Transform Component* приказује тренутни положај ентитета у координатном систему преко x, y и z координата, али такође дозвољава промену тих координата ако корисник жели да помери ентитет на неко друго место. Исто важи за величину и ротацију ентитета која се такође чува у *Transform Component*.

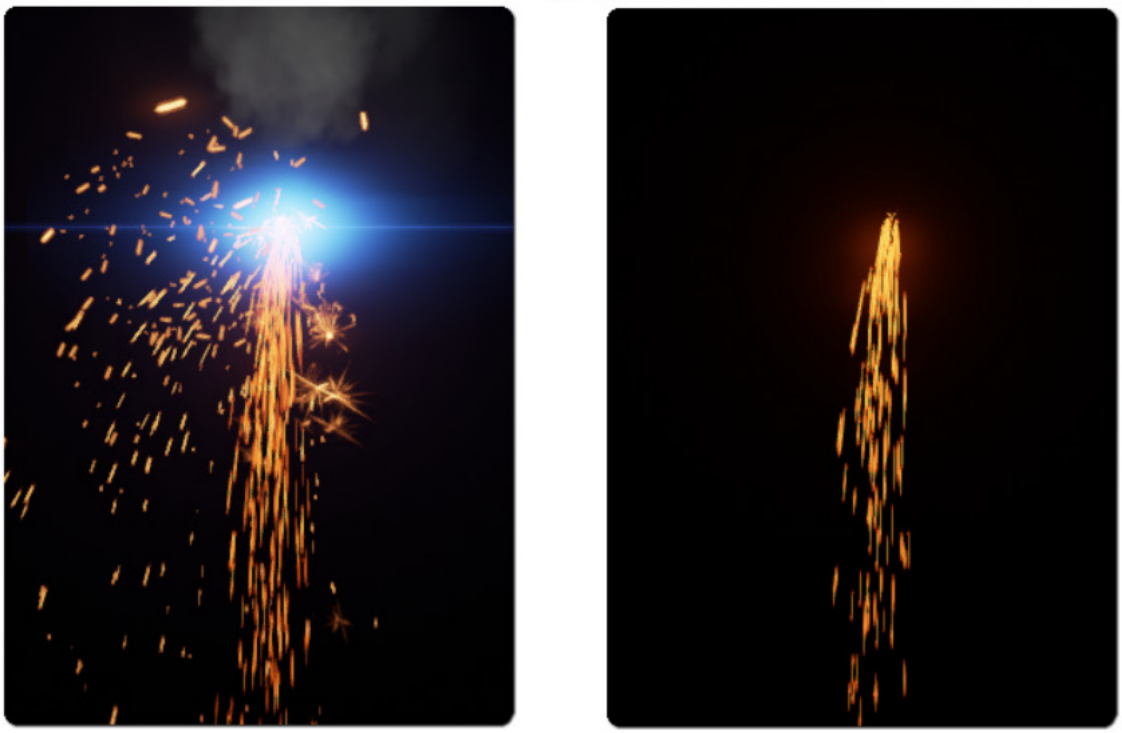
Ентитети обично имају *Mesh Component* која представља изглед ентитета, његову текстуру итд. Из претраживача садржаја може да се превуче облик тела (коцка, сфера итд.) и стави на поље које даје изглед ентитету, исто важи за текстуру.

Скрипте

Најбитнија врста компоненти која може да се дода ентитету је компонента са скриптом. Скрипта је обично блок кôда који диктира понашање ентитета коме припада. Програмски језик скрипте може бити исти језик којим је написан погон, али то није правило. Погон за видео игре *Godot* је написан у програмском језику *C++*, његове скрипте могу бити у том језику али такође подржава *C#* и *GDScript* скрипте. Број различитих понашања која могу бити направљена са скриптама је бесконачан. Пример таквог понашања би био да се главном карактеру пропише начин реаговања на унос корисника. Таокође може да се постави скрипта ентитету који представља врата која служи за отварање тих врата када се главни карактер довољно приближи.

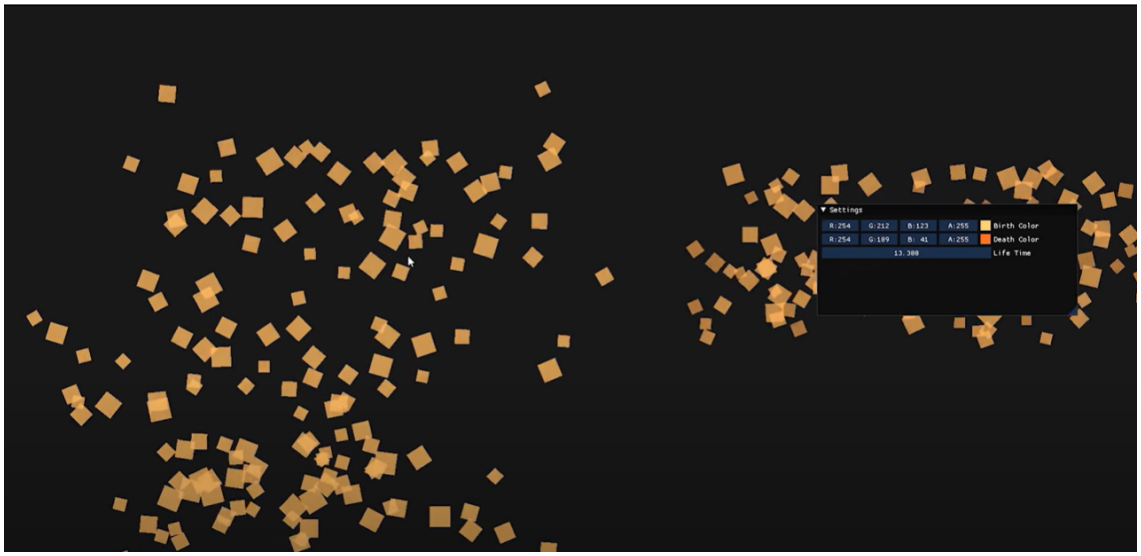
Систем честица (енг. *Particle system*)

Систем честица је потребан за развој било које модерне видео игре. Насумично генерисање ситних честица које могу да представљају било шта (ватру, дим, струју, листове итд.) даје видео игри динамичност. Параметри који могу да се подешавају за честице су количина честица, брзина стварања честица, правци у којима се честице померају итд. Сваки овај параметар треба у себи да садржи одређену насумичност иначе честице не изгледају довољно „жив". Обично користимо честице тако што их додамо на неки ентитет, честице се стварају на том ентитету и прате његово кретање. На пример, црвеној сфери се додају честице ватре да би се направила ватрена кугла. На слици 2.3 је представљено генерисање варница системом честица у 3D видео игри.



Слика 2.3: Приказ 3D честица које се користе за видео игре

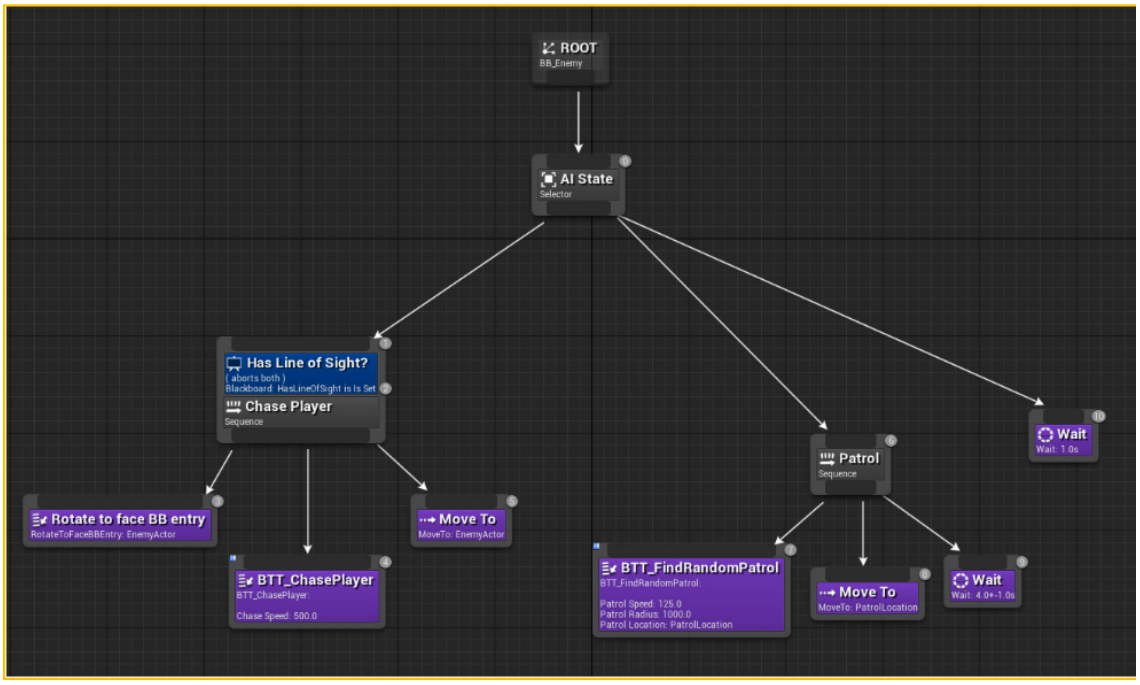
На слици 2.4 је представљено генерисање честица (наранџасти квадрати) у 2D видео игри. Честице се генеришу на локацији миша у моменту када се притисне леви тастер.



Слика 2.4: Приказ 2D честица које се користе за видео игре

Систем за вештачку интелигенцију

У видео играма вештачка интелигенција се обично користи када се праве непријатељи. Непријатељи треба да се померају, бране, нападају и слично, а за то им је потребна нека врста „размишљања". Погон за видео игре треба да олакша програмерима стварање вештачке интелигенције. *Unreal engine* поседује систем за вештачку интелигенцију, заснован на дрвету понашања (енг. *Behavior tree*), који ће бити детаљније описан у наставку.



Слика 2.5: Приказ дрвета понашања које се користи за *Unreal engine* систем вештачке интелигенције

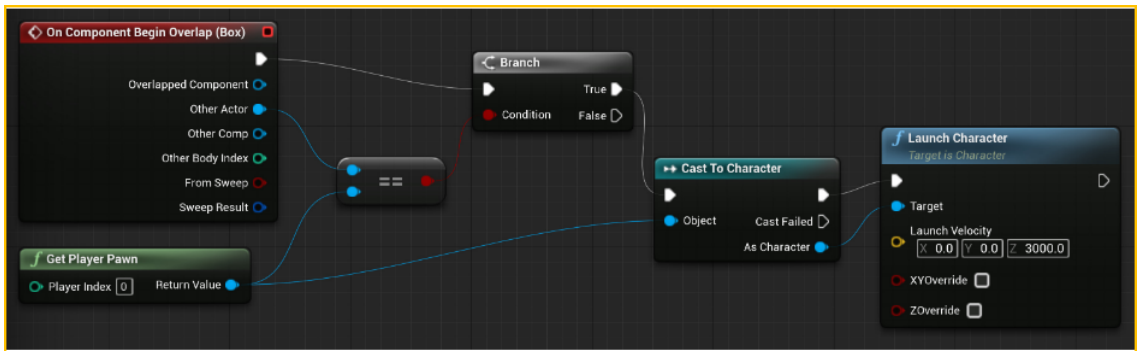
Да би неким ентитетом управљала вештачка интелигенција постоји неколико корака али најбитнији је конструкција дрвета понашања. Дрво понашања има чворове и гране, увек се извршава од горе на доле и слева на десно. Постоје три врсте чворова композита (енг. *Composites*), ови чворови диктирају како се извршавају под-дрва повезана са њима. Бирач (енг. *Selector*) извршава редом свако под-дрво слева на десно док не наиђе на једно које може да изврши до краја, ако једно под-дрво успе да се изврши онда је и селектор успео да се изврши. Секвенцер (енг. *Sequence*) извршава под-дрва слева на десно, ако једно извршавање не успе секвенцер прекида извршавање и пропагира се као неуспех. Једноставна паралела (енг. *Simple Parallel*) у

исто време извршава један главни задатак (енг. *Task*) и једно позадинско дрво извршавања. На основу неких подешавања када се изврши главни задатак позадинско дрво може да се или абортира или изврши до краја. Осим композита постоје и задаци, постоје већ направљени задаци којима само треба проследити параметре, а постоје и задаци које дефинише корисник.

На слици 2.5 је приказано једно дрво извршавања које се додељује непријатељу. Основне функције овог дрвета су трчање за главним карактером (ако је у видном пољу), патролирање у супротном и чекање ако ниједна акција од претходне две није могућа. *Unreal engine* обезбеђује доделу „чула" вештачким интелигенцијама са којима могу да „виде" или „чују" главног карактера. *Unreal engine* такође поседује навигациони квадар (енг. *NavMeshBoundsVolume*), то је ентитет који помаже вештачким интелигенцијама да се крећу кроз простор не ударајући у препреке.

Систем шаблона (енг. *blueprints*)

Систем шаблона служи истој сврси као и скрипте, али шаблони могу да се користе без велике количине искуства са програмирањем. На слици 2.6 се види приказ једног шаблона који је део система шаблона имплементираног у оквиру *Unreal* погона. у наставку ће бити детаљније описан начин коришћења тог шаблонског система.



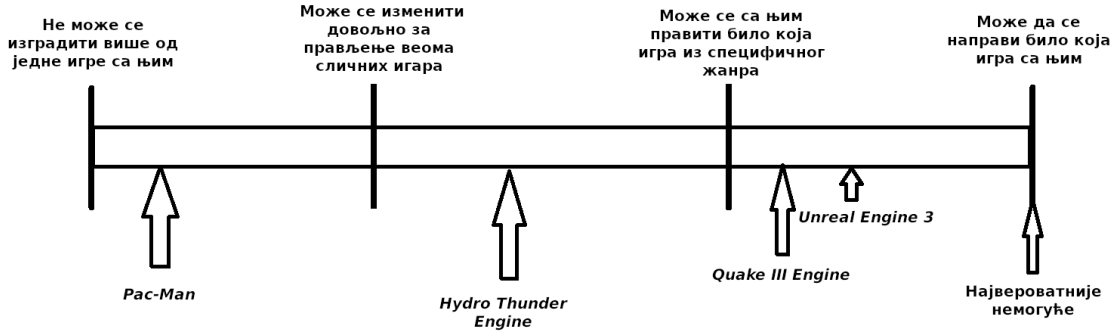
Слика 2.6: Приказ једног шаблона, који служи да главног карактера лансира увис

Шаблон се састоји из чворова (енг. *Nodes*) и веза између њих. Он је, једноставно речено, неповезан граф који може да има велики број почетних чворова. Чвор шаблона може бити догађај (енг. *Event*) и они обично представљају почетни чвор. Уобичајени догађаји су: почетак видео игре, почетак

преклапања са другим ентитетом, догађаји направљени од стране корисника итд. Постоје аритметички чворови (==, +, - итд.) који могу да се користе за разна израчунавања са променљивама. Променљиве су везане за ентитет који садржи шаблон и представљене су стандардним типовима података (*int, float, bool* итд.), могу бити и колекције података (*Array, map* итд.). Чворови такође могу бити понуђене функције (функције за дохватање променљивих, за постављање вредности променљивама, за дохватање глобалних објеката видео игре итд.) или функције које корисник сам прави. Изрази који контролишу ток (*if statement, for loop* итд.) програма могу бити представљени чвором, користе се на исти начин као и у програмским језицима.

Шаблони су корисни када је потребно нешто да се тестира брзо, корисни су почетницима и људима који се не баве професионално програмирањем. Али у озбиљном развоју видео игара програмери се не ослањају често на њих, зато што писање сопственог кôда обично даје боље перформансе.

2.5 Недостаци погона за видео игре



Слика 2.7: Спектар могућности поновног коришћења погона за видео игре [5]

Тешко је повући линију између погона за видео игре и саме игре. У једној видео игри систем за исцртавање може да зна како да нацрта краву, док је за неку другу крава унета као податак независан од погона. Обично архитек-

тура заснована на подацима одликује софтвер који није део погона. Целине програма које могу да се користе за прављење многих игара се сматрају делом погона. Делови изграђени специфично за конкретну видео игру (нека понашања ликова, исцртавање специфичних геометријских облика) се тешко користе за креирање неке друге видео игре [5].

Слика 2.7 представља спектар на који могу да се распореде погони за видео игре на основу потенцијала за поновно коришћење. У идеалном случају би постојао погон који може да направи било коју врсту игре, али он још увек постоји само у теорији. Велики број погона је написан за једну специфичну игру, други су написани са одређеним жанром видео игара у виду и за конкретну платформу. Чак и генерални погони направљени за велики број платформи обично буду ефикасни само за мали број жанрова видео игара. Показало се да што је погон општији, мање је ефикасан за прављење конкретне видео игре на конкретној платформи. Тај феномен се јавља јер конструкција ефикасног програма захтева специјализацију на основу претпоставки о начинима коришћења тог софтвера и платформи за коју се тај програм прави [5].

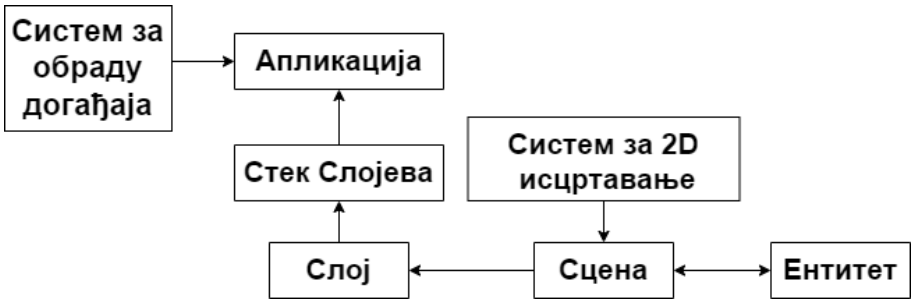
Глава 3

Архитектура предложеног алата

Предложени погон за видео игре, назван погон *Stormlight*, је написан у програмском језику *C++*. За његово писање је коришћено *Visual Studio* окружење за развој у оквиру оперативног система *Windows*. Од великог значаја за развој погона биле су следеће библиотеке: *spdlog*, *glm*, *ImGui*, *entt*, *Box2D* и *HazelAudio*, о којима ће бити речи у наставку. Комплетна имплементација погона *Stormlight* се може наћи на [11].

3.1 Структурални општи преглед система

Погони за видео игре могу бити сачињени од великог броја система и не постоји опште правило које говори шта све један погон мора да садржи у себи. Погон *Stormlight* садржи системе који су неопходни за изградњу 2D видео игара и најбитнији међу њима се виде на слици 3.1.



Слика 3.1: Приказ архитектуре погона *Stormlight*

Апликација (енг. *Application*) представља улаз у програм и садржи *петљу извршавања* која је активна током целог рада програма и ажурира системе

погона *Stormlight*. Слојеви апликације (енг. *Layers*) постоје да би корисник могао да бира редослед исцртавања делова видео игре на екрану. Током развоја видео игара некада је потребно да одређене информације увек буду видљиве. Такве информације би сместили у највиши слој и оне би биле исцртане последње.

Сцена (енг. *Scene*) у себи садржи ентитете (енг. *Entity*) то су: карактер кога контролише играч, непријатељи, препреке и други. Сцена такође има функције за рад са ентитетима (креирање ентитета, уништавање ентитета, провера колизије међу њима). Ентитети се приказују уз помоћ система за 2D исцртавање (енг. *Renderer*). Систем за 2D исцртавање омогућава исцртавање правоугаоника и могу бити обојени или обложени текстуром (учитаном сликом).

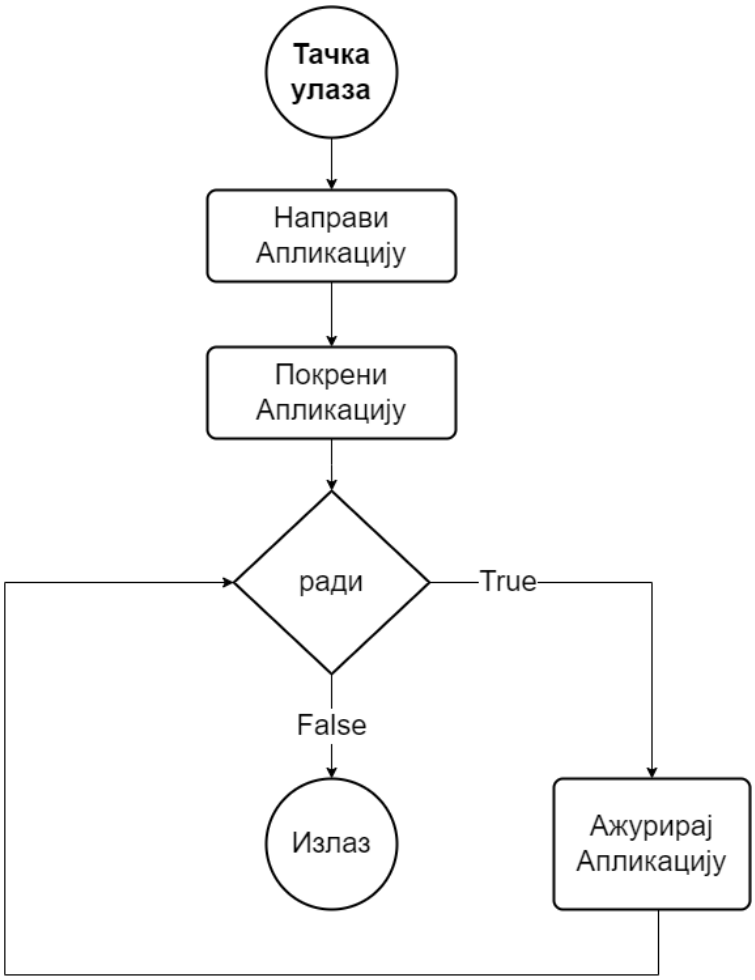
3.2 Опис општег понашања система

На слици 3.2 је приказано опште понашање погона *Stormlight*. Први корак је прављење апликације, а затим се покреће *извршна петља* у њој. Апликација се ажурира све док се не промени параметар који контролише извршавање петље. Током ажурирања апликације извршавају се аутоматска ажурирања система попут обраде колизија, ажурирање ентитета, исцртавање објеката на екрану и многих других, као и реаговање на унос корисника преко система за обраду догађаја.

3.3 Апликација

Видео игра која се надограђује на погон *Stormlight* мора у себи да садржи класу која наслеђује базну класу *Application*. Та класа спаја све елемете погона и успоставља везе међу њима.

Класа *Application*, приказана на слици 3.3, се ослања на уникат (енг. *Singleton*) образац, а поље *s_instance* чува објекат класе *Application*. На основу имплементације уникат шаблона може постојати само један објекат класе *Application*. Осим тог поља класа *Application* чува прозор апликације, *ImGui* слој, стек слојева, као и поље типа *bool* које чува информацију о томе да ли је апликација завршила са радом или не.



Слика 3.2: Приказ понашања погона *Stormlight*

Application
-s_Instance Application* -m_Window : std::unique_ptr<Window> -m_ImGuiLayer: ImGuiLayer* -m_LayerStack : LayerStack -m_Running : bool
+Run() : void +OnEvent(Event& e) : void +PushLayer(Layer* layer) : void

Слика 3.3: Структура класе *Application*

Метод *Run()* у себи садржи *петљу извршавања*, он се позива на почетку рада програма и извршава се до његовог краја. У сваком проласку кроз петљу рачуна се *Timestep* а то је време потребно да се изврши један пролаз. *Timestep* се пропагира свим функцијама за ажурирање система, оне га користе за

разна израчунавања, а посебно је битан у случајевима када апликација услед спољашњих фактора успори са радом. *Timestep* обезбеђује да чак и кад апликација успори са радом, објекти видео игре не успоре своје кретање.

Метод *Run()* такође пролази кроз стек слојева и ажурира сваки понаособ, а ажурирањем прозора апликација проверава да ли се десио неки догађај (притиснут тастер, померен миш, промењена величина прозора итд.) и ако јесте реагује на њега.

3.4 Слојеви апликације

Слојеви представљају једну одвојену целину апликације која врши исцртавање на екрану, обрађује догађаје и ажурира део система. Чувају се у стеку (енг. *Stack*) зато што је добро имати јасан редослед извршавања апликације. То је потребно у случајевима када неке информације попут укупног броја бодова, менија (енг. *menu*) када је укључен итд. увек треба да буду исписане на екрану. Таква информација се ставља у највиши слој (енг. *Overlay Layer*), због тога што се ажурирање и исцртавање слојева врши од најнижег слоја до највишег, добијамо ефекат да је најнижи слој исцртан први, а највиши последњи.

Обрада *улазних догађаја* (који настају као реакција на улазне уређаје) се врши у обрнутом редоследу. Највиши слој први реагује на догађај и у зависности од догађаја врши обраду или га игнорише чиме се догађај пропагира до наредног слоја који се налази испод тренутног. Тај редослед обраде се користи због ситуације када је у видео игри укључен мени и корисник мишем кликне на дугме на менију. Потребно је да се тај догађај обради у највишем слоју где се налази мени. У супротном може да се догоди да у неком нижем слоју тај догађај буде обрађен (на пример карактер кога контролише играч испали стрелу) и тиме корисник не добије жељени ефекат.

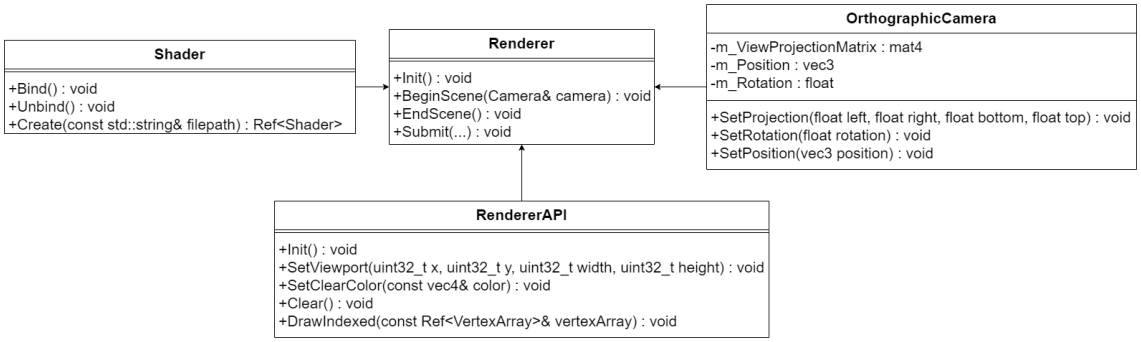
Класа *LayerStack* је омотач око низа који у себи садржи показиваче на објекте класе *Layer* и тај низ представља стек слојева. *LayerStack* садржи функције за манипулисање над тим стеком које додају или бришу *Layer* са стека. Класа *Application* садржи инстанцу класе *LayerStack*.

Класе *Layer* има виртуалне функције *OnAttach()*, *OnUpdate()*, *OnDetach()*, *OnEvent()* и *OnImGuiRender()*. У току развоја видео игре стварају се потребни слојеви, који наслеђују класу *Layer* дефинисану у погону, и преписују

њене виртуалне функције својим конкретним.

OnAttach() има сличну сврху као конструктор, али се позива сваки пут када се слој дода у стек слојева. *OnDetach()* се позива сваки пут када се избаци слој са стека слојева. *OnUpdate()* може имати разне намене у зависности која је сврха конкретног слоја и у тој функцији може да се обавља исцртавање на екрану, ажурирају битни системи видео игре итд. У *унутрашњој петљи* апликације пролази се кроз стек слојева и позива се *OnUpdate()* функција за сваки слој. *OnEvent()* функција обрађује догађаје прослеђене слоју. У *унутрашњој петљи* се пролази кроз стек слојева у обрнутом редоследу и за сваки слој се позива та функција. *OnImGuiRender()* функција ажурира графички кориснички интерфејс видео игре.

3.5 2D систем за исцртавање

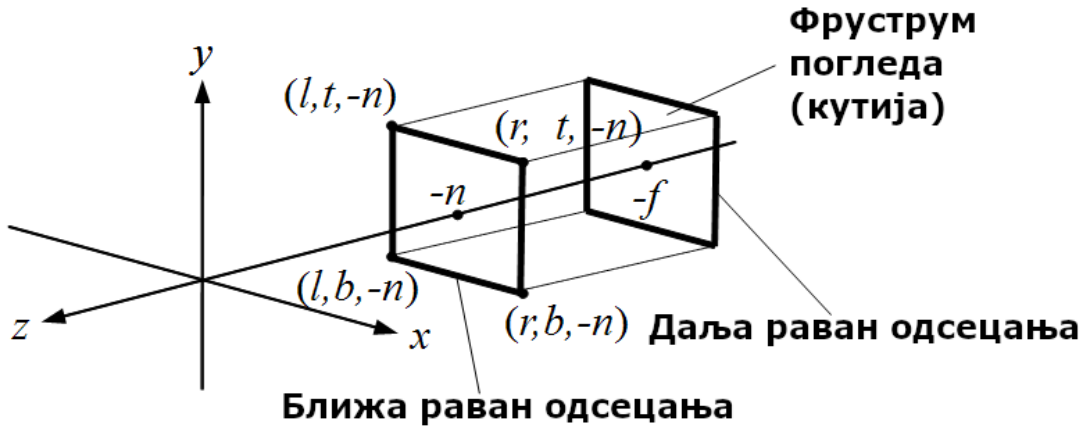


Слика 3.4: Поједностављени приказ 2D система за исцртавање

Систем за исцртавање је потребан сваком погону за видео игре и он у сваком тренутку ефикасно исцртава различите објекате на екрану. Да би то постигао *Renderer* се ослања на графичку картицу рачунара која садржи хардвер посвећен исцртавању као и способност паралелног извршавања великог броја операција одједном (сваком пикселу на екрану треба да се одреди боја, паралелизација тог посла повећава ефикасност система за исцртавање).

На слици 3.4 се виде најбитније класе које су део 2D система за исцртавање. Систем за исцртавање је комплексан и да би се разумео процес исцртавања прво мора да се објасни појам синтетичке камере и шта је шејдер (енг. *Shader*).

Синтетичка камера



Слика 3.5: Фруструм погледа ортографске камере, оригинална слика је преведена [7]

Синтетичка камера одређује шта ће бити приказано на екрану и због тога што је погон *Stormlight* намењен искључиво за видео игре у 2D, користи се ортогонална пројекција (објекти који се налазе даље од камере неће бити умањени). На слици 3.5 су приказане главне одлике ортографске камере. Њен фруструм је квадар одређен са две равни одсецања и тачкама које дефинишу величину видног поља. На слици се камера налази у координатном почетку а правац њеног погледа је z-оса, док јој је смер гледања негативан у односу на смер z-осе. Оно што се налази у простору између камере и ближе равни одсецања (енг. *near clipping plane*), обележене са *-n* на слици, неће бити исцртано. Аналогно све иза даље равни одсецања (енг. *far clipping plane*), обележене са *-f* на слици, такође неће бити исцртано. Тачке *left (l), right (r), bottom (b), top (t)* ограничавају видно поље и одређују његову дужину и висину.

Класа *OrthographicCamera* представља синтетичку камеру као и методе за њено коришћење, њена структура се види на слици 3.4. Поља *m_ Position* и *m_ Rotation* служе за позиционирање камере у простору, а матрица која носи битне информације о камери се чува у пољу *m_ ViewProjectionMatrix* (то поље је потребно систему за исцртавање да креира слику у видном пољу).

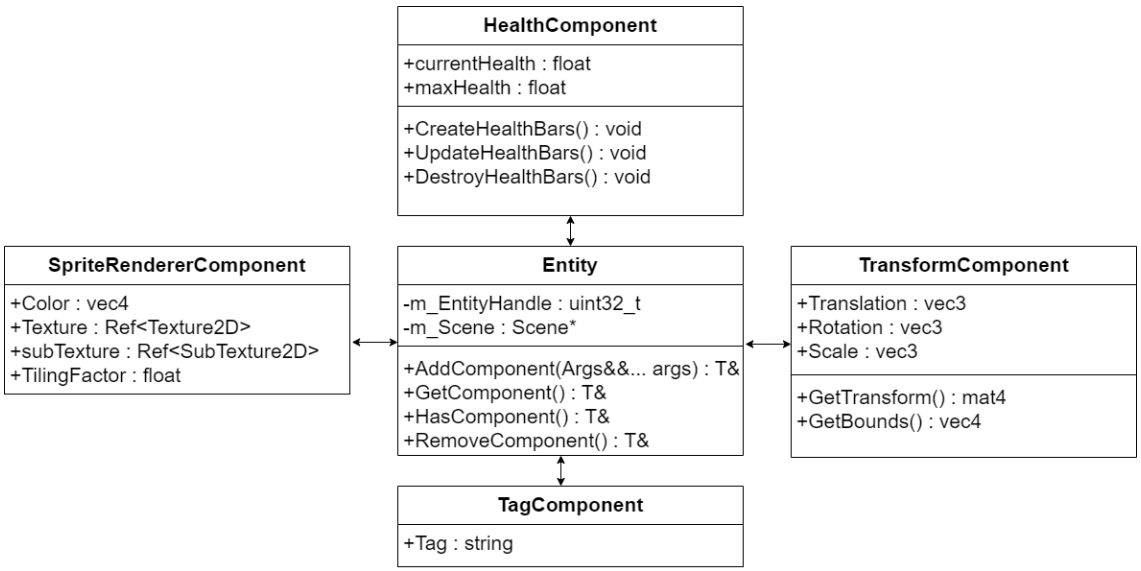
Шејдери (енг. *Graphics Shaders*)

Шејдер је блок кôда који се извршава на графичкој картици (уместо да се извршава на процесору). *Vertex шејдер* и *Fragment шејдер* су две врсте *шејдера* који се користе када се позове исцртавање. *Vertex шејдер* се позива за свако теме (енг. *Vertex*) полигона који се исцртава (ако се црта троугао, *Vertex шејдер* би био позван три пута - по једном за свако теме). Главна функција *Vertex шејдера* је утврђивање где се на екрану налазе темена полигона који се исцртава. *Fragment шејдер* ће се позвати по један пут за сваки пиксел који припада унутрашњости полигона који се исцртава. Његов главни задатак је да утврди које боје задати пиксел треба да буде. *Vertex шејдер* означи границе полигона, а *Fragment шејдер* обоји његову унутрашњост. *Fragment шејдер* се позива огроман број пута због чега увек треба водити рачуна какве операције се извршавају у њему, неке ствари је боље обрадити у *Vertex шејдеру* и тако обрађене их проследити *Fragment шејдеру*. Када се позове исцртавање *Vertex шејдеру* се шаље матрица пројекције главне камере дате сцене и темена полигона који се исцртава. Да би се утврдио положај полигона у односу на камеру, вектор који представља теме се множи матрицом пројекције главне камере. *Fragment шејдер* може да обезбеди „налепљивање" текстуре на полигон које се исцртава.

Позив исцртавања

За 2D видео игре је потребно да систем за исцртавање буде способан да исцрта правугаоник у боји или са „налепљеном" текстуром. Када се позове исцртавање систему је потребно да зна: које *Шејдере* треба да повеже (енг. *Bind*), низ темена које представљају правугаоник који ће бити исцртан, матрицу трансформације тог правоугаоника (она представља његов положај, ротацију и величину), као и матрицу пројекције камере. *Vertex Шејдеру* се проследе те две матрице које он множи са сваким теменом правоугаоника понаособ да би добио релевантне позиције темена. *Fragment шејдеру* може да се проследи текстура и координате које дефинишу који део текстуре се „лепи" на правоугаоник или само боју правоугаоника одређену са четири броја (обично се користи *RGBA* формат за боје, прва три броја представљају колико црвене, зелене и плаве боје учествује у резултујућој боји, а последњи број одређује величину алфа параметра који утиче на транспарентност).

3.6 Систем ентитета и компоненти



Слика 3.6: Пример једног ентитета са својим компонентама

Мотивација за систем ентитета и компоненти

Видео игре су врста програма који садрже велики број објеката (ентитета, агената итд.) попут непријатеља, препрека, замки, карактера кога контролише играч. Сваки од тих објеката има одређена својства попут способности да испушта звук, скрипту понашања, текстуру и тако даље.

У развоју видео игара често је потребно исцртати све објекте који поседују текстуру. За решење тог проблема био би потребан низ свих објеката кроз који се пролази *for* петљом. За сваки објекат се проверава да ли поседује текстуру и ако поседује исцртава се.

Овакав приступ има више проблема: сви објекти у низу морају или да буду исте класе, или да буду изведени из исте класе што није практично јер може да постоји велики број различитих објеката са специфичним својствима (нека својства могу и да се преклапају, па би било доста понављања или јако заплетена хијерархија наслеђивања). Други проблем представља ефикасност извршавања. Ако се пролази кроз низ свих објеката петљом сувишно се проверавају објекти који немају текстуру, они нису потребни у датом сценарију. Ако укупно објеката има милион, а само сто њих поседује текстуру, очигледно

је да претходно решење ради превише непотребног посла сваки пут када се исцртавају искључиво објекти са текстуром. Већина погона за видео игре садржи неку врсту система ентитета и компоненти (енг. *Entity Component System*) како би решили представљене проблеме.

Предности система ентитета и компоненти

Једно предложено решење би било да се сва својства представе преко структура које наслеђују базну класу компонента, а класа ентитет (која представља објекат) у себи садржи низ показивача на компоненте (која год својства тај конкретни ентитет има похрањена су у том низу). Тиме се елиминишу комплексне хијерархије и понављања кôда, али проблем ефикасности још увек није решен. У новом решењу када се пролази петљом кроз све ентитете, за сваки од њих се проверава да ли у низу компоненти има компоненту за текстуру, дакле и даље се проверавају ентитети који немају компоненту за текстуру.

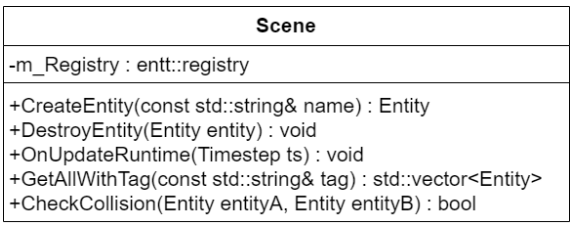
Зато систем ентитета и компоненти групише компоненте заједно (на пример у низове). У већини случајева информација ком ентитету нека компонента припада није битна, у описаном случају шаље се компонента за текстуру 2D систему за исцртавање, он је обради и исцрта (осим текстуре била би му потребна и матрица трансформације, то може да се реши тако што се групише више различитих компоненти у један низ). Систему за исцртавање није битно ком ентитету припада текстура коју је исцртао. Ако је потребно да се огласе сви ентитети који имају аудио компоненту, *for* петља би прошла кроз све аудио компоненте и пустила звук сачуван у њима, у овом сценарију такође ентитет коме аудио компонента припада није пресудна информација. Оба проблема су решена, сада *for* петља пролази искључиво кроз ентитете (специфичније кроз тражене компоненте тих ентитета) који су релевантни. У систему ентитета и компоненти, ентитет је представљен само као број, а компоненте које припадају истом ентитету су повезане преко тог броја.

Структура система ентитета и компоненти

У погону *Stormlight* се користи библиотека *entt* [9] као основа за систем ентитета и компоненти. Као надоградњу за функционалности те библиотеке у решењу су написане две класе *Scene* и *Entity*. Класа *Entity* представља

омотач око ентитета (*entt::entity*) који је дефинисан у библиотеци *entt*. Тамо је ентитет обичан тип података *uint32_t*, класа-омотач *Entity* осим поља које садржи тај једнозначни идентификатор такође садржи показивач на објекат класе *Scene*. На слици 3.6 је приказана структура класе *Entity*, као и пример неколико компоненти које могу да јој припадају. Класа *Entity* садржи шаблон функције (енг. *template function*) за контролисање компоненти.

Компоненте су једноставно дефинисане као структуре које садрже нека поља и понашања. Најбитније компоненте су: *TransformComponent* која садржи информације о положају, ротацији и величини Ентитета и метод којим се рачуна матрица трансформације, *SpriteRendererComponent* без које ентитет не може да буде исцртан на екрану.



Слика 3.7: Структура класе *Scene*

Класа *Scene* служи за груписања ентитета и њихово обрађивање. На слици 3.7 видимо структуру класе *Scene*. Поље *m_Registry* је структура која у себи чува све ентитете и компоненте. Та структура има методе за креирање нових ентитета, брисање ентитета као и за додавање, брисање и дохватање компоненти везаних за специфични ентитет (омотачи за ове методе су дефинисани у класи *Entity*).

Метода *OnUpdateRuntime()* проналази тренутну главну камеру (сцена понекад садржи више камера, али само једна може да буде главна) и исцртава све ентитете који имају *SpriteRendererComponent*-у из угла те камере. Метода *GetAllWithTag()* проналази све ентитете који имају прослеђени *tag* (*string* који представља име ентитета) и враћа их у низу. У видео игри овај метод се користи ако је потребно да се провери да ли је дошло до колизије између испаљеног пројектила и непријатеља. Позивањем ове методе се добија низ сачињен од свих непријатеља у игри, а онда се за сваког проверава да ли је у колизији са пројектилом. Провера колизије се врши методом *CheckCollision()* која на основу позиција, величине и ротације два ентитета утврђује да ли је дошло до преклапања међу њима.

3.7 Осврт на коришћене библиотеке

У пројекат су укључене многе библиотеке. Обично би неке од функционалности које оне пружају биле имплементиране у оквиру погона за видео игре али користе се зато што доста убрзавају процес развоја и углавном пружају добре перформансе. Коришћене библиотеке, осим библиотеке *entt* о којој је већ било речи, ће бити продискутоване у овом поглављу. Поменуте библиотеке су: *spdlog*, *glm*, *ImGui*, *Box2D* и *HazelAudio*.

Библиотека *spdlog*

Библиотека *spdlog* [4] омогућава логовање информација што је веома битно у фази развоја. Пружа неколико различитих врста порука које могу бити исписане: *trace*, *info*, *warn*, *error* и *critical*. Свака од ових порука се исписује у различитој боји што омогућује програмеру да лако разазна која врста поруке се исписује.

Библиотека *glm*

Сваки погон за видео игре има библиотеку за математику. Ова библиотека пружа основне операције са матрицама и векторима које су потребне сваком погону. *OpenGL Mathematics (GLM)* [6] је биоблиотека за математику имплементирана у програмском језику *C++* намењена за коришћење уз *OpenGL* језик за *шејдере* (*GLSL*). Користи се у шејдерима за множење матрица, у ортографској камери за креирање матрице пројекције, у *TransformComponent* за стварање матрице трансформације и у још много делова кôда.

Библиотека *ImGui*

ImGui [8] је библиотека која у себи садржи функционалности потребне за изградњу графичког корисничког интерфејса (енг. *Graphical User Interface - GUI*). Током развоја видео игре ова библиотека се користи за приказивање одређених информација на екрану које су потребне за дебаговање. При развоју видео игре може да се користи и за креирање интерактивних менија и приказивање порука на екрану.

Библиотека *Box2D*

Библиотека *Box2D* [3] симулира физику крутих тела за видео игре. Ако је потребно да се тело креће на природан начин користи се ова библиотека. На пример ако је потребно да се једно тело одбије о друго или да одскочи. Ова библиотека омогућује да свет направљен у игри постане више интерактиван. Може да се постави гравитација на почетку игре која утиче на сва тела која нису статична (не могу да се померају). У погону *Stormlight* се користи за проверу да ли је дошло до колизије између два ентитета. Библиотека *Box2D* је написана у програмском језику *C++*.

Библиотека *HazelAudio*

Библиотека *HazelAudio* [10] пружа функције за манипулацију аудио фајлова (подржава екстензије *.ogg* и *.mp3*). Аудио фајлови могу да се пусте, зауставе, паузирају и ставе у бесконачну петљу. Библиотека такође дозвољава утишавање и појачавање аудио фајлова. Класа *AudioManager* користи ову библиотеку да учита и сачува све аудио фајлове потребне за видео игру.

Глава 4

Развој видео игре

Након изградње погона за видео игре развијена је једноставна видео игра (*демо*) као илустрација способности погона. Имплементација видео игре се може наћи на [11].

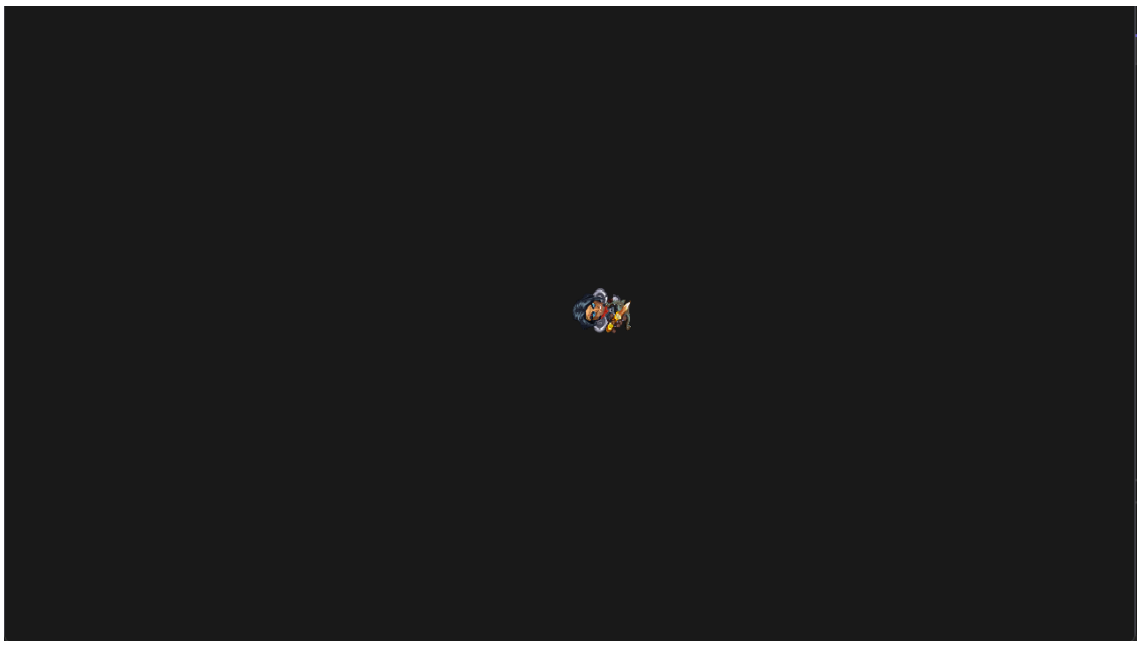
Процес креирања видео игре започиње са фазом планирања: који тип 2D видео игре се прави? Како изгледа карактер кога играч контролише? Које способности има? Која је његова релација са другим елементима у видео игри? Да ли има непријатеље или решава загонетке? Ако има непријатеље, које су њихове одлике? Шта је циљ играча? Коју причу има видео игра?

Ово су само нека од многих почетних питања која се намећу у процесу планирања нове видео игре. Такође се истражује тржиште, то јест прикаупљају се подаци о актуелним видео играма, трендовима у индустрији видео игара, интересовањима играча и предвиђањима о правцима развоја индустрије. За конкретну видео игру која је развијена већина тих ствари није била потребна зато што се ради о једноставној видео игри која испитује функционалности погона.

Да би се започело прављење видео игре погон треба да буде укључен као библиотека у нови пројекат који представља видео игру. Праве се класе које наслеђују базне класе *Application* и *Layer*. На тај начин се покрећу основни системи погона за видео игру (систем за исцртавање, главна *петља извршавања* итд.). Након тога креира се инстанца класе *Scene*, која може да се попуни са ентитетима. Први ентитет који се додаје на сцену је камера. То је ентитет коме је додељен *CameraComponent*, та компонента садржи инстанцу синтетичке камере и још неке помоћне информације.

Следећи корак је прављење карактера кога контролише играч, такозвани главни карактер. Класа *Player* наслеђује *Entity*, главни карактер је ентитет коме могу да се додају компоненте, осим тога има доста додатних функционалности (способности, оружја, анимације). На слици 4.1 се види играч који има текстуру, додату преко *SpriteRendererComponent*, али још увек нема способност померања и анимације.

Кретање се остварује преко *TransformComponent*. Њено поље *Translation* чува *x, y* и *z* координате ентитета. У зависности од притиснутог тастера (*w, a, s, d*) одређује се правац кретања ентитета. На *x* и *y* координате поља *Translation* се додаје одређена вредност докле год су одговарајући тастери притиснути. Додата вредност зависи од правца, тренутне брзине главног карактера и вредности *ts* која може да варира у зависности од тога колико треба систему да прође једном кроз главну *петљу извршавања*.

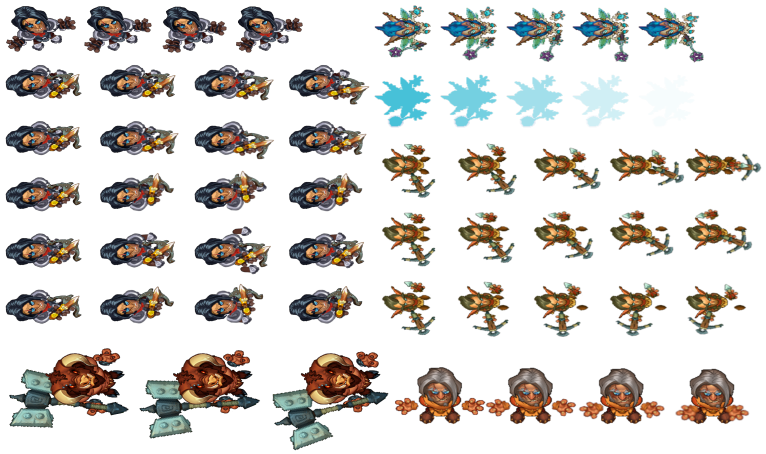


Слика 4.1: Приказ прве етапе развоја видео игре - камера и главни карактер

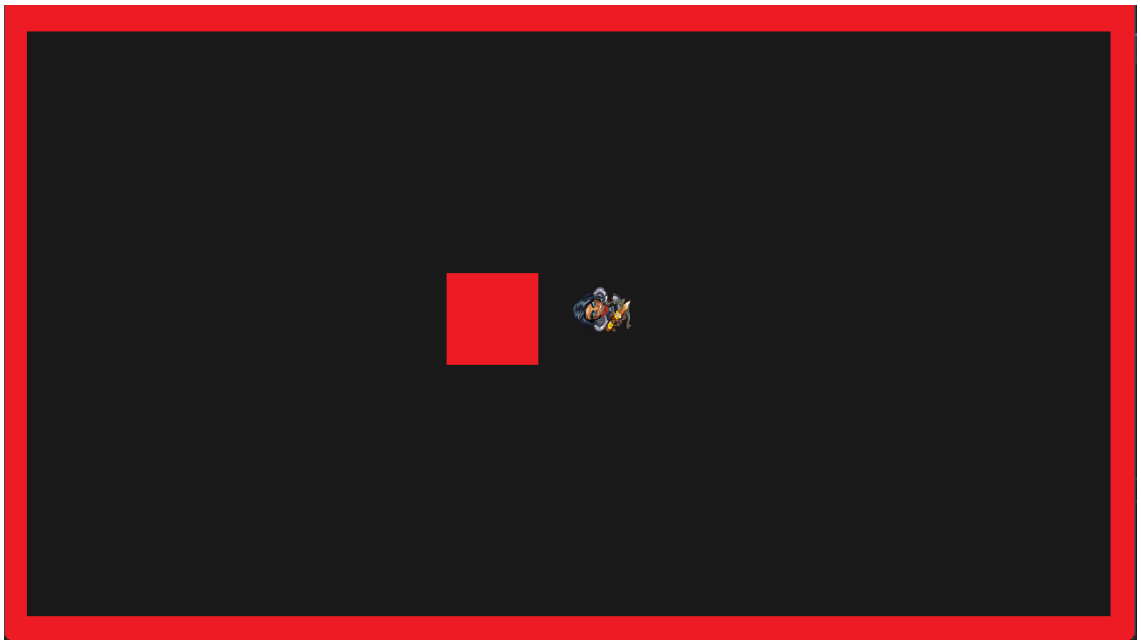
Анимације у 2D видео играма се већински остварују брзим смењивањем слика које све заједно представљају неку активност (кретање лево, кретање десно, пуцање из оружија итд.). На слици 4.2 су приказани примери анимација које се користе у видео игри. *AnimationComponent* чува информације о времену приказивања једне слике у низу (*FrameDuration*), као и низове подтекстура које представљају различите акције ентитета. Уместо да се учита

свака слика засебно, учитава се једна велика слика која садржи целу акцију. Та слика се подели на подтекстуре које се брзо смењују да би дале привид кретања.

Класа *TextureRepository* учитава све текстуре када се видео игра покрене. Текстуре и подтекстуре су дефинисане у погону за видео игре у виду класа *Texture2D* и *SubTexture2D*.



Слика 4.2: Анимације карактера у видео игри



Слика 4.3: Приказ друге етапе развоја видео игре - препреке и колизије

Због тога што главни карактер сад може да се креће, постављају се препреке на сцену. На слици 4.3 се виде додати ентитети који представљају препреке. Препреке спречавају да главни карактер напусти видно поље. Метод *OnUpdate()* класе *Player* дохвата све ентитете који су препреке (на основу функције *GetAllWithTag()* класе *Scene*) и позива метод *CheckCollision()* за конкретну препреку и главног карактера. Ако дође до колизије она се обрађује. Постоје разни начини да се спречи пролаз кроз препреке, овај конкретни ради на основу правца кретања главног карактера и његове позиције у односу на препреку. Када дође до колизије главни карактер се помера уназад правцем којим је дошао, док се не нађе ван препреке.

Слика 4.4: Приказ треће етапе развоја видео игре - позадина и оквир

На слици 4.4 је приказана додата позадина и оквир, то су обични ентитети којима је величина повећана тако да прекривају цео прозор и којима су додате одговарајуће текстуре. Направљене су нове сцене *Backgound Scene* (позадинска сцена) и *Foreground Scene* (сцена за предњи план), да би слојеви били јасно раздвојени због редоследа исцртавања. Прво се исцртава позадина, па главни део игре (главни карактер, непријатељи итд.) и на крају оквир. Када је потребно да видео игра прикаже да се главни карактер померио у неки други део нивоа промени се текстура позадине и оквира. Сцена са оквиром (*Foreground Scene*) ће у себи садржати и елементе за приказ битних

информација (енг. *Heads up display - HUD*).

Направљена је нова класа *Level* која садржи све три сцене и води рачуна о њиховој иницијализацији и ажурирању. Видео игра има само један ниво, али ова класа је осмишљена као шаблон за све потенцијалне нивое. Класа *Level* води рачуна о иницијализацији већине система у видео игри.

Слика 4.5: Приказ четврте етапе развоја видео игре - способности главног карактера

Главни карактер може да се креће али не може да се бори против непријатеља. Класа *Crossbow* (самострел) представља оружије које користи главни карактер. Самострел испаљује стреле на леви клик миша, када стреле погоде непријатеља одузму му одређену количину *енергије* (*Health*). Након испаљене стреле постоји пауза (енг. *Cooldown*) пре него што може да се испали следећа. Осим оружија главни карактер поседује способности (*Skills*). На слици 4.5 приказано је коришћење све три способности одједном. Способности имају паузу између коришћења, у дну екрана је приказано време потребно да се сачека пре него што активирана способност може опет да се искористи. Иконе способности, количина новца коју тренутно главни карактер поседује и опрема представљају делове *HUD* система. Класа *Level* чува инстанцу класе *HUDManager* која иницијализује и ажурира *HUD* систем. *HUD* система се ослања на *ImGui* функције како би приказао текст на екрану.

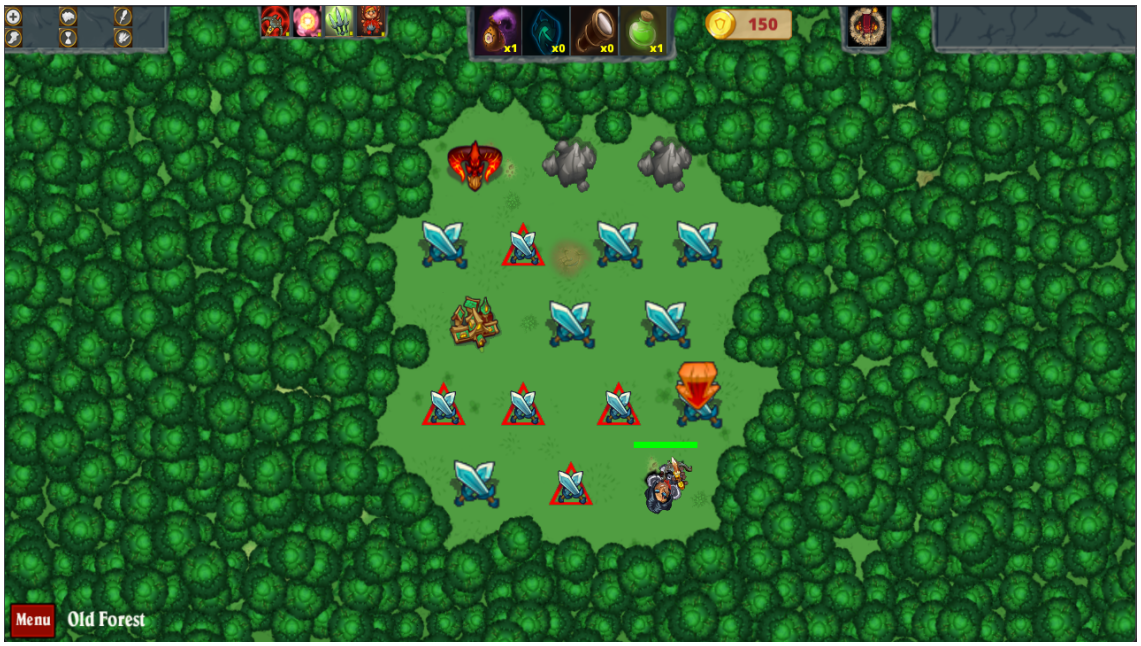
Слика 4.6: Приказ пете етапе развоја видео игре - непријатељи

Непријатељи су ентитети које главни карактер треба да уништи у циљу напредовања до краја видео игре. Постоје три врсте непријатеља, сваки поседује специфичан начин напада који главни карактер треба да избегне. Непријатеље води слаба вештачка интелигенција, заснована на низу акција које се извршавају једна за другом по истеку времена додељеног свакој акцији. Класичне акције су мировање, кретање и напад.

Класа *EnemyManager* ствара непријатеље и чува их у низу. Преко тог низа их ажурира. Непријатељи су ентитети коме је додата *EnemyComponent*. *EnemyComponent* садржи неке информације потребне непријатељима али најбитније је да свим ентитетима са том компонентом може ефикасно да се приступи по потреби. На слици 4.6 су приказане три различите врсте непријатеља који постоје у овој видео игри *Минотаур*, *Вила* и *Бандит*.

Следећи корак у прављењу видео игре је приказан на слици 4.7, то је мапа која представља ниво. На мапи се налазе сценарији које играч треба да превазиђе да би победио. Класа *Encounter* је базна класа за све сценарије и садржи у себи основна понашања сваког сценарија која се преписују у класама које наследе *Encounter*. Постоји седам врста сценарија: *застава*, *борба*, *јача борба*, *продавница*, *краљица*, *стена* и *мрачни сценарио*.

Застава је почетни сценарио на коме се налази главни карактер када се



Слика 4.7: Приказ шесте етапе развоја видео игре - мапа сценарија

појави на мапи. Сценарији који су повезани са *заставом* су доступни за померање на њих (везе између сценарија су уређене графом чији су чворови сценарији а гране везе између сценарија). На почетку сценарији су скривени, осим *заставе* и сценарија повезаних са њом, померањем се откривају нови сценарији.

Борба (на слици 4.7 представљена са два укрштена мача) је основни сценарио, када се уђе у њега генерише се неколико непријатеља које главни карактер мора да победи да би напустио сценарио. *Јача борба* (на слици 4.7 представљена са два укрштена мача преко црвеног троугла) такође генерише непријатеље, али осим њих има и додатну *замку* коју главни карактер мора да избегне. На слици 4.8 је приказана *јача борба* са *замком*.

Продавница је приказана на слици 4.9, служи за куповину опреме. Када главни играч уђе у *продавницу* инстанца класе *ItemManager* ствара трајну и потрошну опрему. *ItemManager* је задужен за реаговање на куповину опреме и на активацију потрошне опреме.

Краљица је најтежи сценарио (На слици 4.7 је представљна главом чудовишта са роговима), када се он превезиђе видео игра је готова. На слици 4.10 је приказан изглед непријатеља који представља *краљицу*.

Стена (На слици 4.7 је представљна стеном) је сценарио у који не може

Слика 4.8: Приказ једног од сценарија - *јача борба*

Слика 4.9: Приказ једног од сценарија - *продавница*

да се уђе. Може да се користи као препрека на мапи.

Мрачни сценарио настаје када се победи неки сценарио у који је главни карактер ушао. Када се главни карактер помери на *мрачни сценарио* ништа

се не догоди, зато што је већ превазиђен.

Слика 4.10: Приказ једног од сценарија - борба са *краљицом*

Класа *EncounterManager* ствара сценарије сваки пут на насумични начин, мора да постоји *продавница*, *краљица* и *застава* али су уз нека правила увек насумично распоређени. *EncounterManager* чува низ сценарија до којих главни карактер може да дође и низ сценарија који су откривени до сада. *EncounterManager* ажурира стрелицу која се појави када миша држимо изнад сценарија у који главни карактер може да уђе. *EncounterManager* памти који је тренутни сценарио у који смо ушли, да би по изласку знао где главни карактер треба да се врати.

Приказано је стварање најбитнијих елемената видео игре, у наставку развоја би се ти елементи користили за креацију нових нивоа, различитих непријатеља, других *краљица*, нове опреме, нових способности главног карактера, нова оружија за главног карактера, нови типови главног карактера итд. Неке видео игре не поседују причу, али често је она најбитнији део видео игре зато што се играчи повежу са карактерима у њој или светом који је представљен кроз причу. То би могао да представља још један правац даљег развоја видео игре.

Глава 5

Демонстрација рада креиране видео игре

Ова глава садржи кратко објашњење коришћења развијене видео игре. На слици 5.1 су стрелицама означени правци кретања и тастери који се користе за кретање. Главни карактер такође може да се креће по дијагоналама ако су притиснута два одговарајућа тастера у исто време (ако су притиснути тастери *w* и *d* на пример). Када се кликне леви тастер миша главни карактер испаљује пројектиле из самострела ка тренутној локацији миша (генерално правац и смер главног карактера су означени са вектором који полази из центра главног карактера а завршава се у тренутној локацији миша на екрану). Осим кретања и пуцања главни карактер поседује способности, прва способност се активира на тастер *Shift*, а друга на тастер *Space*.

Главни карактер се налази у помоћном сценарију када игра почне. Тај ниво служи за испорбавање способности главног карактера, када се очисте непријатељи у овом сценарију главни карактер прелази на мапу нивоа. Када је мапа приказана главни карактер не може да се помера на уобичајени начин тастерима *w, a, s, d*. Такође су блокирани напади и способности јер нису потребни док је карактер на мапи. На мапи је приказан сценарио *заставa* на коме се главни карактер налази, ако је сценарио повезан са *заставом* и он се види. Ако кликнемо левим тастером миша на видљив сценарио поред главног карактера померамо главног карактера у тај нови сценарио (тако се главни карактер помера када се налази на мапи). Померање на нови сценарио започиње одигравање тог сценариа, али такође на мапи открије нове сценаријe који су сада доступни. На слици 5.2 је приказана једна путања до

Слика 5.1: Приказ кретања главног карактера

Слика 5.2: Приказ једне путање до *краљице*

краљице. На слици је истакнуто да главни карактер може да користи потрошну опрему када се налази на мапи, тако што кликне на њу левим тастером миша. Потрошна опрема је *напитак за енергију* (енг. *health potion*), *камен*

телепортације (енг. *warp stone*), *дурбин* (енг. *spyglass*). *Напитак за енергију* се користи за додавање *енергије* главном карактеру. *Камен телепортације* омогућује један већи „скок" на било који сценарио на мапи, *дурбин* открива све неоткривене сценарије на мапи. У *продавници* главни карактер може да купи и потрошну и трајну опрему. Трајна опрема побољшава карактеристике главног играча. Карактеристике главног играча су *јачина напада* (колико *енергије* главни карактер одузима непријатељима стрелама), *јачина магије* (колико одузима *енергије* способностима), *брзина напада*, *брзина кретања*, *количина енергије* и *cooldown reduction* (што је ова карактеристика већа то главни карактер мање чека између активирања способности). Трајна опрема има разне комбинације побољшања ових карактеристика које важи до краја игре.

Глава 6

Правци даљег рада и закључак

6.1 Правци даљег рада

Погони за видео игре су често огромни и овај рад није представио све могућности погона. Одлука да буде искључиво 2D издваја *Stormlight* од већине погона који имају тенденцију да обухвате што више различитих жанрова видео игара. Будућа надоградња погона би подразумевала креирање додатних карактеристика које олакшавају развој 2D видео игара. На пример, графички кориснички интерфејс дизајниран специјално за 2D игре или систем за анимирање објеката игре уграђен у погон.

Погон *Stormlight* тренутно искључиво ради на *Windows* оперативном систему, видео игре за *Linux, macOS* и многе конзоле не би могле да се праве коришћењем њега. Даљи развој овог пројекта би обухватао додавање функционалности које оспособљавају погон за коришћење на већем броју платформи. Додавање тих функционалности неће захтевати драстичне промене у системима који већ постоје. Одређени аспекти архитектуре су изграђени са тим функционалностима у виду. У погону *Stormlight* постоји раздвајање између кôда који је писан за конкретну платформу (*Windows*) и кôда који не зависи ни од једне платформе.

Стандардни погони обично поседују поменуте системе за шаблоне, честице и вештачку интелигенцију, стога њихова реализација представља добар правац даљег развоја за погон *Stormlight*.

6.2 Закључак

Идеја овог рада је била упознавање са основама прављења погона за видео игре и видовима његовог коришћења. Погон *Stormlight* је довољно функционалан за развијање простих видео игара, али тешко да би могла да се направи профитабилна видео игра његовим коришћењем. Постоји више разлога за то: развој видео игара за различите платформе није могућ на њему, не поседује графички кориснички интерфејс што смањује број људи који би га користили и не повећава ефикасност продукције 2D видео игара. Прављење профитабилних видео игара зависи и од много других фактора (изглед видео игре, основне механике, прича), а не само од погона који се користи за њихово стварање.

Профитабилност на страну, у раду су описани најбитнији системи погона за видео игре, проблеми које решавају и начин на који су ти системи повезани. Направљена је једна кратка видео игра користећи погон која задовољавајуће тестира његове функционалности.

Проласком кроз цео процес развоја погона за видео игре, постало је јасно да би цео пројекат био једноставније и потпуније реализован када би на њему радио већи број програмера. Погони за видео игре су састављени од великог броја јасно раздвојених компоненти, из тог разлога посао изградње погона би лако могао да се раздели тимовима програмера. Уколико се неко специјализовао за прављење система за исцртавање, не мора нужно да зна како се прави систем ентитета и компоненти. Проналажење стручних људи за сваку компоненту би далеко унапредило резултујући погон. Када овакав пројекат ради једна особа постоји велики број области којима мора да се посвети, а самим тим се смањује пажња коју би требало издвојити за сваку област понаособ. Као резултат тога перформансе изграђеног погона испаштају.

Библиографија

[1] H. Chandler and R. Chandler. *Fundamentals of Game Development*. Foundations of game development. Jones & Bartlett Learning, 2011.

[2] Eleftheria Christopoulou and Stelios Xinogalos. Overview and comparative analysis of game engines for desktop and mobile devices. *International Journal of Serious Games*, 4(4), Dec. 2017.

[3] Erin Catto (erincatto). box2d. `https://github.com/erincatto/box2d`, 2016.

[4] Gabi Melman (gabime). spdlog. `https://github.com/gabime/spdlog`, 2015.

[5] Jason Gregory. *Game engine architecture*. Taylor & Francis Ltd., 1 edition, 2009.

[6] Christophe (Groovounet). glm. `https://github.com/g-truc/glm`, 2011.

[7] Martin Kraus. Orthographic view frustum. Martin Kraus, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons.

[8] omar (ocornut). imgui. `https://github.com/ocornut/imgui`, 2015.

[9] Michele Caini (skypjack). entt. `https://github.com/skypjack/entt`, 2018.

[10] Yan Chernikov (TheCherno). Hazelaudio. `https://github.com/TheCherno/HazelAudio`, 2022.

[11] Александар Ранковић. Thebrokenflame. `https://github.com/AleksandarTheBoulderRankovic/TheBrokenFlame`, 2022.

Биографија аутора

Ранковић Александар рођен је у Београду 19.01.1993.

Похађао је Основну школу „Стеван Дукић" и паралелно са њом музичку школу „Станковић" дувачки одсек – инстумент хармоника где је стекао основно музичко образовање.

Од 2008. године до 2012. године похађао је Прву Београдску гимназију, природно математички смер, након које је уписао Електротехнички факултет Универзета у Београду. После две године се исписао са тог факултета.

Од 2016. године до 2020. године студирао је на Математичком факултету Универзитета у Београду, на информатичком смеру. Дипломирао је 2020. године и уписао мастер студије на истом смеру.

У току мастер студија обавио је праксу у фирми *Mad Head Games*, током које се упознао са основама функционисања *Unreal* погона.