Универзитет у Београду
Електротехнички факултет

ДИПЛОМСКИ РАД

ИНТЕЛИГЕНТНА ВИЗУЕЛНА СИМУЛАЦИЈА ВИРТУЕЛНОГ ИГРАЧА ШАХА

Ментор
проф. др Бошко Николић

Студент
Предраг Митровић 2015/0608

Београд, септембар 2019.

Садржај

1. Увод

Рачунари су способни да играју шах. Први покушај да се аутоматизује игра шаха се догодио 1914. године помоћу машине коју је конструисао Торес Кеведо која је била способна да изведе мат помоћу две фигуре, краља и топа, против противничког краља.

У другој половини 20. века, људи су почели да се баве програмирањем рачунарских играча за шах са циљем да направе играча који би био непобедив.

Током времена, заједница људи која се интересује за развој сличних решења се проширила. Истраживања су јавно објављивана и била доступна јавности. Хардвер је напредовао, али без обзира на то и даље не постоји непобедива машина из разлога што је стабло игре исувише комплексно и садржи приближно 10^{120} чворова.

Микропроцесори су постајали све јефтинији и моћнији и почела је продаја рачунара специјализованих за шах. Програмери који су се бавили развојем софтвера за такве рачунаре су чували своје алгоритме како би задржали предност над конкуренцијом.

Прва машина која је успела да победи у шаховском мечу против човека који је светски шампион био је рачунар *Deep Blue* који је развијен од стране научника са института Карнеги Мелон, а касније је пројекат финансиран од стране америчке компаније *IBM*. Био је састављен од 50 рачунара и направљен је специјално за ову намену. Победио је Гарија Каспарова, велемајстора шаха и једног од најбољих играча свих времена. Било је потребно конструисати машину која евалуира 200 милиона потеза у секунди да би се то догодило.

Са развојем интернета, аматери у програмирању ботова за шах су почели да размењују информације и да деле код јавно. Настала су два програма (*Fruit* и *Stockfish*) која су развила међусобни ривалитет и после одређеног времена превазишли су комерцијалне програме. Били су *open source* типа што им је омогућило да на њима ради велика група људи која је радила на унапређењу и новим идејама. Велики број волонтера и ентузијаста за шах донирао је процесорску моћ како би се на њима извршавали тестови и како би пројекат напредовао.

Данас постоји огроман број виртуелних играча за шах. Постоје турнири за ботове који се организују како би се утврдила ранг листа.

Један од најпопуларнијих статистичких метода за мерење вештине ботова се ослања на турнире где бот игра против различитих ботова огроман број мечева са истим временским ограничењима. Закључак о перформансама се изводи на основу броја победа, нерешених исхода и пораза. Потенцијални проблем је што у овом случају коначна оцена зависи од снаге противника против којих су играни мечеви. Оцена се изражава у поенима и након тога се формира коначан пласман.

Софтверски пакет за персонални рачунар који је настао као резултат овог дипломског рада се састоји из две целине. Прву целину представља виртуелни играч који анализира стања шаховске табле, генерише могуће потезе и врши одабир потеза који је евалуирао као најбољи. Поред тога, реализован је и графички кориснички интерфејс који омогућава кориснику да игра против рачунара, да посматра игру између два бота и да игра против другог човека. Програм на тај начин даје кориснику прилику да напредује као играч.

Решење је имплементирано у програмском језику C# уз коришћење мултиплатформске машине за игру под називом *Unity.*

Неки од разлога који чине програмирање виртуелних играча интересантним су коришћење различитих структура података, алгоритама сортирања, алгоритама претраге и разних других метода при имплементацији решења. Током рада се продубљује знање програмирања у изабраном програмском језику јер се тежи писању ефикасног и „чистог" кода.

У поглављу 2 биће размотрени неки од приступа које данашњи програмери примењују при решавању проблема који се обрађује. Биће описане основне софтверске компоненте који су неопходни за рад виртуелног играча за шах.

Опис рада система је написан како би се разјаснили функционални захтеви апликације и приказао рад система. Корисник ће након читања поглавља 3 бити у стању да покрене апликацију и да је користи као игру или симулатор игре између два бота.

У поглављу 4 ће бити дискутовани детаљи имплементације решења. Читалац ће се упознати са начином репрезентације табле и фигура. Биће описано како се *FEN string* конвертује у таблу, како се генерише хеш вредност стања табле, како функционише генерација потеза и на који начин је тестирана њена исправност. Биће објашњено коју хеуристику бот користи за евалуацију потеза. Биће описан *Negamax* алгоритам претраге са алфа-бета одсецањем и како оптимизовати поредак генерисаних потеза како би се максимално убрзао рад алгоритма. Приказана је имплементација визуелног дела апликације, начине интеракције бота и човека.

У закључку ће бити прокоментарисана постигнућа дипломског рада, недостаци и начини за њихово отклањање и предлози за надоградњу система.

2. Преглед постојећих решења

Шах је игра „савршене информације” из разлога што су оба играча свесни целокупног стања игре у сваком тренутку. Посматрајући таблу, они могу закључити које фигуре су у игри и где се налазе. Игре као што су дама, тавла и го спадају у исту категорију, док се рецимо покер разликује из ралога што није могуће видети које карте у руци држи противник.

Већина техника које су коришћене за решавање проблема се могу применити на различите игре „савршене информације”, иако детаљи могу варирати од игре до игре. Исти алгоритам претраживања се може применити на било коју од њих, док генерација потеза и евалуација стања зависи од правила игре.

Да би бот успешно играо шах, неопходно је реализовати различите софтверске компоненте:

- Меморијска репрезентација шаховске табле како би у сваком тренутку могао да приступи информацијама о стању у ком се налази
- Правила генерације легалних потеза како би поштовао правила игре и уверио се да противник не покушава да прекрши неко од правила
- Интелигентан начин избора наредног потеза
- Интерфејс који омогућава интеракцију са противником

Програмери који су зачетници развоја виртуелних играча за шах су имали велико ограничење у пројектовању решења зато што је у то време меморија била изузетно лимитирана. Неки од програма су користили само 8 *kB* меморије. Због тога су најједноставније репрезентације табле биле најефикасније. Типична табла је имплементирана као низ од 64 елемента где је сваки елемент представљао једно поље на табли. Сваки елемент је био величине 1 *B* и празном пољу је додељивана вредност 0, белом пешаку вредност 1 итд. У данашње време већина машина су 64-битне па су постале актуелне репрезентације табле под називом *Bitboards* где једна 64-битна реч садржи информације о једном аспекту стања игре као што је рецимо распоред пешака на табли где сваки од 64 бита показује да ли је поље заузето. Овакве репрезентације су ефикасне зато што је потребно често вршити логичке операције трају само један процесорски циклус.

Правила игре одређују које потезе може да повуче играч који је на реду. У неким играма је једноставно одредити легалне потезе. Рецимо, у игри икс-окс празно поље означава легалан потез. У шаху, правила су компликованија. Свака фигура има своја правила кретања, није дозвољено да након потеза једног од играча његов краљ остане у шаху, начин промоције фигура, специфични потези као што су рокада или *En Passant*.

Потребно је омогућити рачунару да на што бољи начин врши одабир потеза. Најбољи начин избора између два потеза јесте да се сагледају њихове последице (да се претраже потези који следе иза њих и да се резултати упореде). Како би се што мање смањила могућност грешке, бот претпоставља да ће противник изабрати најбољи потез по себе, односно најлошији по бота. Ово је главни принцип *Minimax* алгоритма претраге који је основа великог броја постојећих решења.

Нажалост, *Minimax* алгоритам има поприлично велику сложеност $O(b^n)$ где је b фактор гранања, односно број легалих потеза у просеку, а n је дубина претраге, односно

број потеза који се посматрају унапред. Неки од најуспешнијих алгоритама који представљају побољшања *Minimax*-а су *AlphaBeta*, *NegaScout* и *MTD(f)* који покушавају да минимизују број чворова које је потребно обићи у стаблу игре.

Још један велики проблем при претрази јесте *„horizon effect”* који је првобитно описао Ханс Берлинер. Претпоставимо да програм претражи све чворове до дубине 4 и закључи да на дубини 4 постоји потез где ће изгубити краљицу. Програм у том случају може одлучити да брани краљицу другим фигурама и одложи губитак краљице још неколико потеза да би је на крају ипак изгубио. Са тачке гледишта бота краљица је спашена јер није довољно дубоко претражио стабло игре да би схватио да ће она свакако бити изгубљена. На крају је изгубио и краљицу и остале фигуре којом ју је бранио и сада се налази у још лошијој позицији. Постоје разне технике које су развијене како би се избегао овај ефекат од којих је најпознатија *Quiescence* претрага.

Бот мора бити у стању да евалуира стања игре, односно да буде у стању да закључи да ли је позиција повољна или неповољна по њега. Осмишљање добре функције евалуације може бити изузетно тежак посао. Грешке при евалуацији могу довести до губитка партије из разлога што врхунски играчи могу да искористе минималну предност као што је рецимо један пешак више у односу на противника.

Програми користе велике хеш табеле фиксне величине како би складиштиле информације о позицијама које су већ евалуиране у некој од претходних претрага. Примери података који се могу чувати су:

- Дубина на којој је позиција претражена
- Вредност позиције која је евалуирана
- Да ли је вредност враћена из методе у току претраге тачан резултат или доња или горња граница
- Најбољи потез за задату позицију

Недостатак меморије онемогућава да све позиције до којих се дошло у току претраге буду ускладиштене. Зато се смишљају технике којим се чувају позиције од већег значаја, а бришу се оне мање битне. Уколико је позиција већ претражена у неком тренутку и присутна је у хеш табели и појави се поново у претрази могуће је искористити информације на два следећа начина:

- Уколико претходна претрага има довољну дубину ускладиштене информације се могу искористити и избећи поновна претрага.
- Уколико претходна претрага нема довољну дубину, али је најбољи потез доступан, потребно је покушати са претрагом стања до којег води тај потез јер постоји могућност да је то и даље најбољи потез.

Итеративно продубљивање је један од најпопуларнијих начина претраге. Уместо претраге по дубини до задатог нивоа, боље је прво претражити дубину 1, па дубину 2 и тако даље све док није достигнута жељена дубина или је време истекло. При свакој итерацији хеш табеле се могу напунити подацима о најбољим потезима што се може искористити у наредној итерацији за избор потеза који се прво претражују што доводи до великог смањења у броју чворова које треба обићи.

Неке од техника које доводе до убрзања претраге су:

1. Прво обићи најбоље потезе који се чувају у хеш табели, уколико постоје.
2. Обићи потезе које доводе до елиминације противничке фигуре.
3. Обићи потезе који су били успешни у *sibling* чворовима.
4. Сортирати потезе по томе колико су били успешни раније у истом стаблу игре.

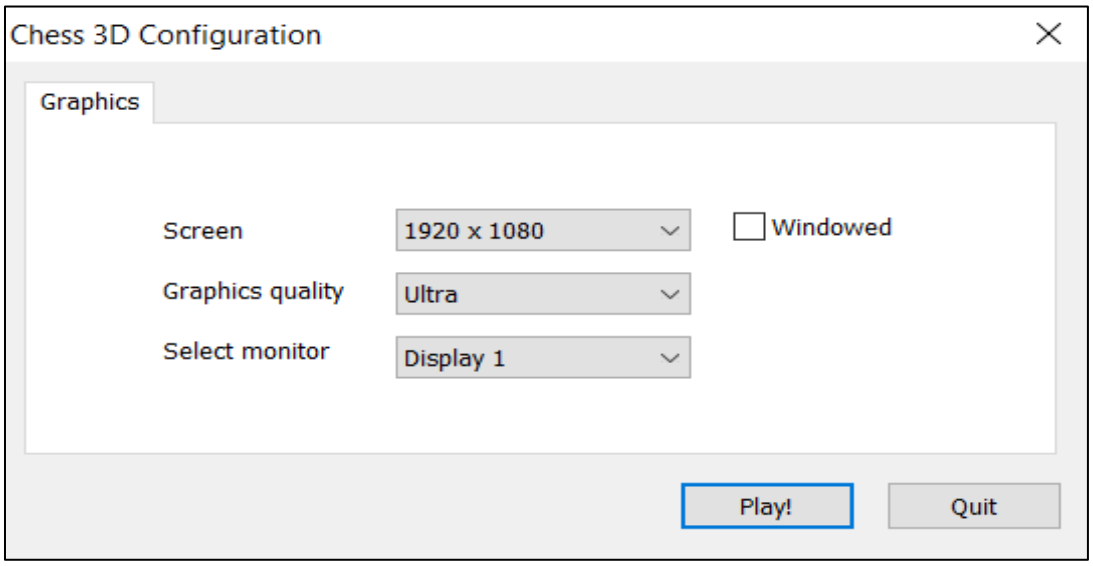
Шаховски програми често имају базу података коју називају књиге отварања. Уместо претраге почетних потеза, користе доступну литературу у вези различитих отварања која пружа виши квалитет игре. Још један разлог за коришћење ове технике јесте што пружа разноврсност у игри јер се потези из књиге обично бирају на случајан начин док су претраге обично потпуно детерминистичке. База се може користити све док противник одиграва потезе који постоје у бази података па су уобичајена отварања ускладиштена у бази до много веће дубине него неуобичајена. Када програм установи да се потез не налази у бази, почиње да извршава своје стандардне алгоритме претраге.

Све информације у бази отварања су намењене како би се игра довела у што повољнијој позицији у средњи део игре. Обично се тада ботови за шах покажу најуспешније. Базе података је могуће креирати на два начина:

1. Ручно израђене базе који израђују стручњаци који анализирају противника против којег бот треба да одигра следећи меч. Тада се врши одабир отварања за који се процењује да је тежак и опасан за противника. Још једна варијанта јесте да се аутоматски генерише књига отварања, а да се затимураде додатна подешавања ручно.
2. Друга опција је да се изабере одређен број партија на које је у великој мери утицало отварање (на пример само партије велемајстора). Затим се до одређене дубине чувају све позиције у књизи отварања. На овај начин ће у бази бити запамћене и неке озбиљне грешке, али се боту може ставити до знања да престане да игра по књизи уколико се позиција појавила мање од задатог броја пута у бази.

3. Опис рада система

Покретањем апликације кориснику ће се отворити прозор са слике 1. У том прозору је кориснику пружена могућност да изабере резолуцију екрана, да ли жели да програм буде приказан преко целог екрана, квалитет графике и монитор на којем ће се апликација отворити, уколико их има више.



Слика 1 – конфигурација апликације

Слика 2 – главни мени апликације

Кликом на дугме *Play!* графички кориснички интерфејс се покреће и приказује кориснику главни мени игре. Мени је приказан на слици 2.

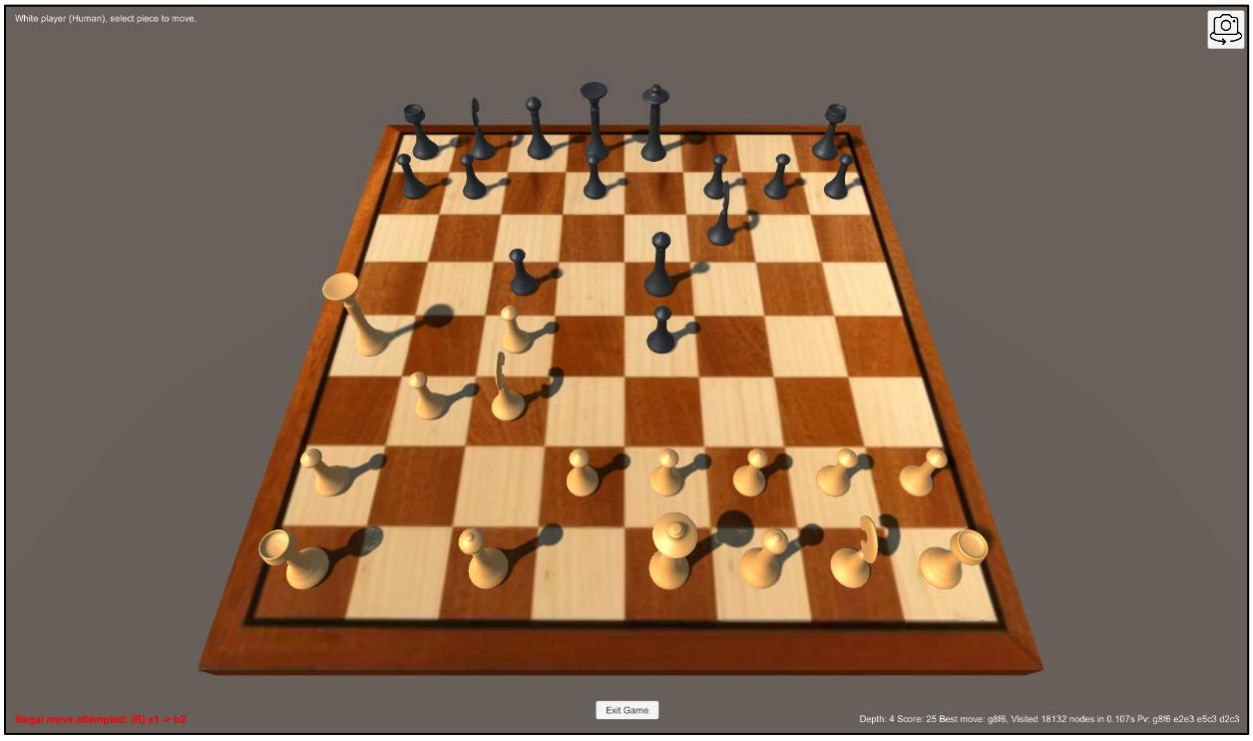
Решење омогућава три варијанте игре:

- Човек против човека
- Човек против виртуелног играча
- Визуелну симулацију игре између два виртуелна играча

Помоћу падајућих менија могуће је одабрати једну од три наведене варијанте тако што корисник врши избор играча са белим фигурама и избор играча са црним фигурама. Уколико је одабран виртуелни играч, неопходно је одабрати и максималну дубину претраге чворова у стаблу игре која је подразумевано 3, а може бити и 4 или 5.

Могуће је и започети игру која не почиње стандардном почетном позицијом тако што се у поље са лабелом *FEN* унесе позиција изражена у *FEN* нотацији која ће бити детаљније описана у поглављу 4.5. Уколико се поље остави празно, игра ће бити започета почетном позицијом која је дефинисана правилима шаха.

Нова игра се започиње кликом на дугме *Start* и приказује се сцена са слике 3, а кликом на дугме *Exit* се напушта апликација.

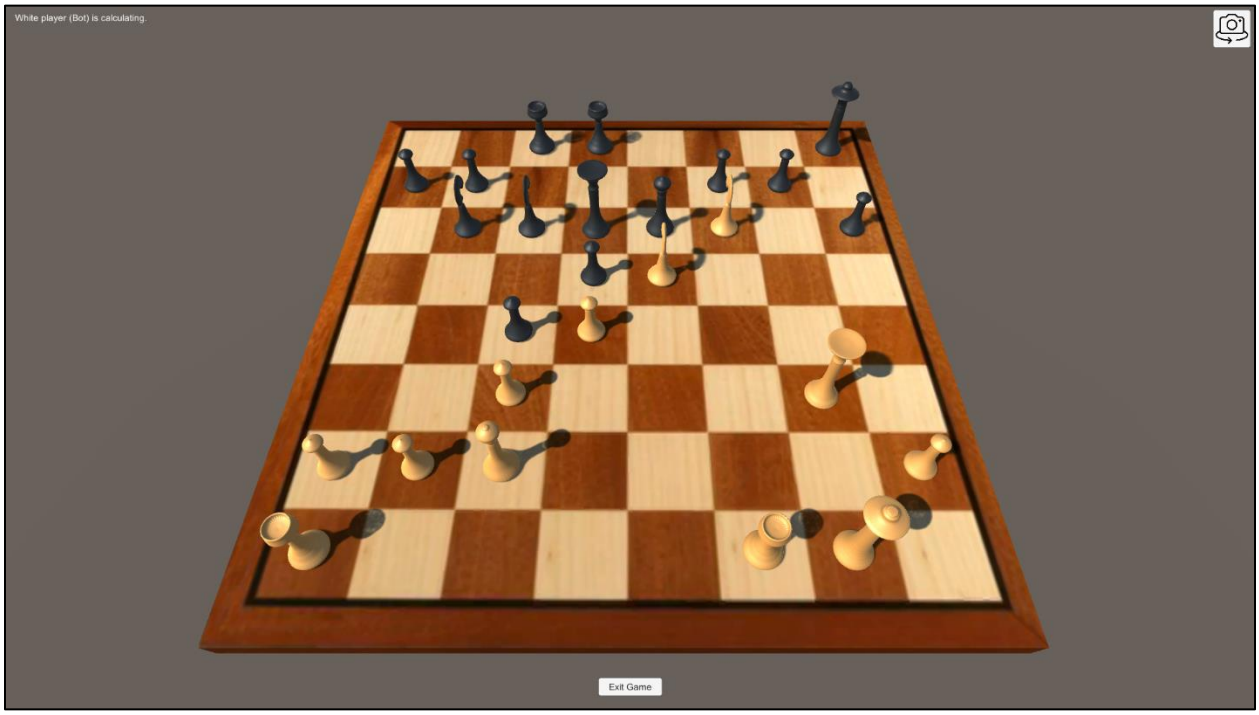


Слика 3 – сцена игре

У горњем левом углу се кориснику приказује који је следећи корак који је потребно да направи како би одиграо потез. У доњем левом углу се црвеним словима кориснику наглашава да је покушао да изведе нелегалан потез уколико је дошло до такве ситуације. У гоњем десном углу постоји дугме које омогућава кориснику да ротира камеру (поглед ка табли) за 180 степени. У доњем десном углу се приказују информације до којих је виртуелни

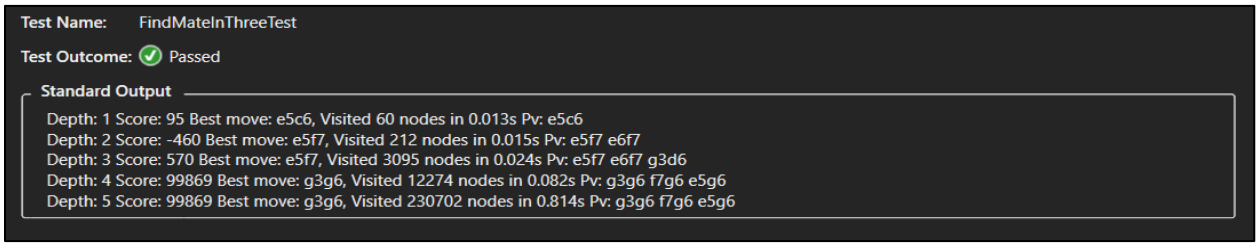
играч дошао у претходној претрази, уколико је виртуелни играч барем један од актера игре. Приказује се дубина претраге, одигран потез и његова процењена вредност, број евалуираних чворова и време утрошено на претрагу и *Principal Variation* линија која ће бити детаљније описана у поглављу 4.11. Кликом на дугме *Exit Game* напушта се тренутна игра и следи повратак у главни мени.

Размотримо пример тока партије која има стање описано следећим *string*-ом у *FEN* нотацији: „*2rr3k/pp3pp1/1nnqbN1p/3pN3/2pP4/2P3Q1/PPB4P/R4RK1 w - -*”. Стање је визеулно представљено на слици 4. Ситуација је интересантна из разлога што бели играч има прилику да матира црног у три потеза (два, уколико не урачунамо и међупотез црног играча). Бели играч је бот који врши претрагу до дубине 5, а црни играч је човек.



Слика 4 – Ситуација где постоји мат у два потеза

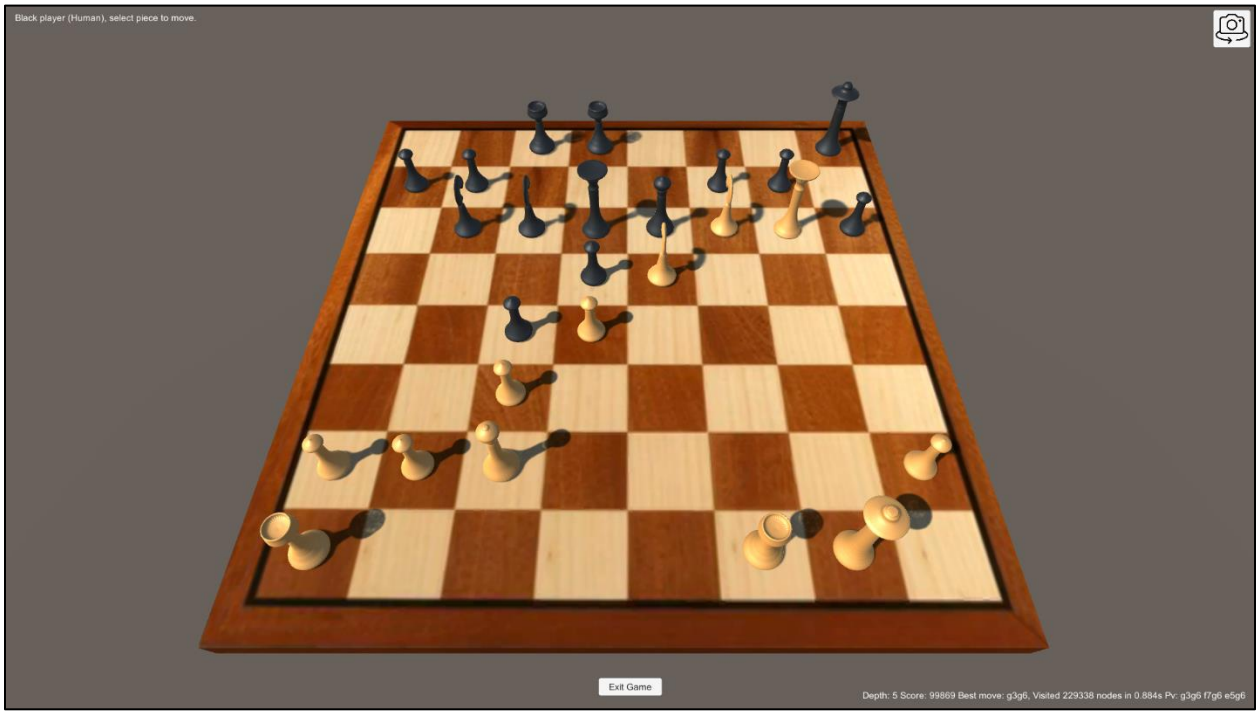
На потезу је бели играч и врши претрагу чији се резултати могу видети на слици 5.



Слика 5 – Резултати претраге у стању са слике 4

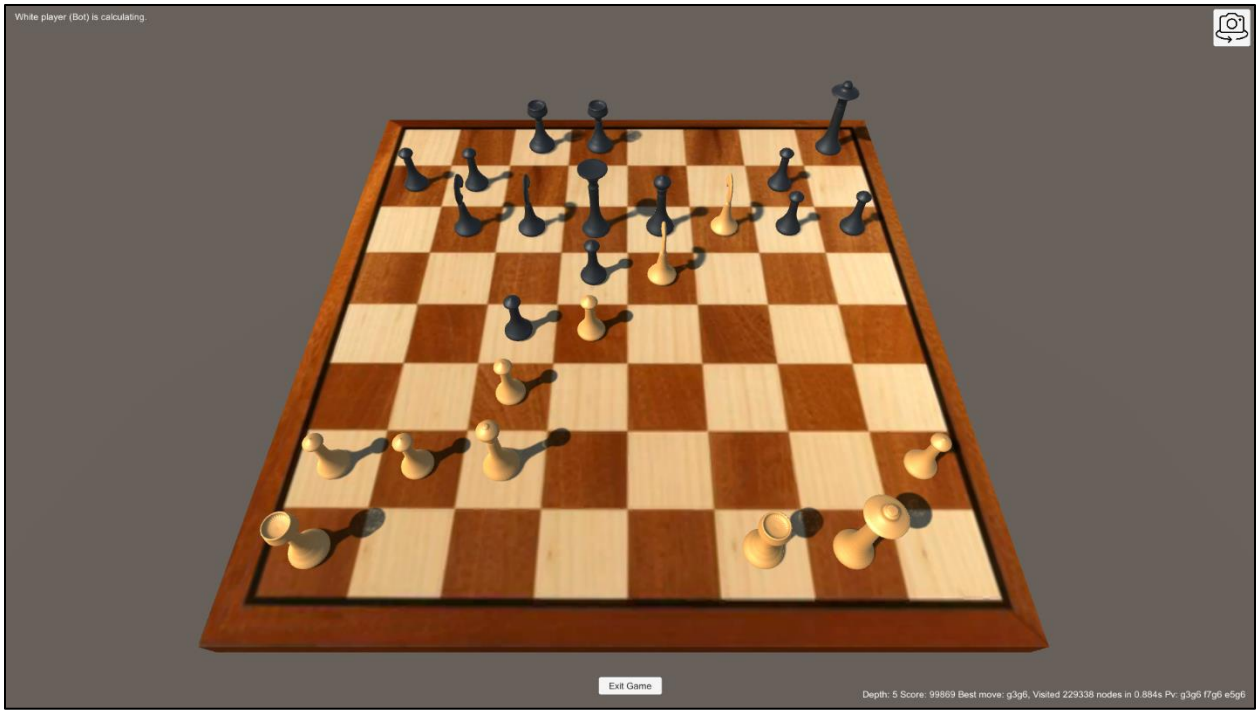
Већ на дубини 4, виртуелни играч је пронашао секвенцу потеза која где води до стања у ком побеђује. Анализом резултата претраге се може доћи до закључка да је задата дубина при претрази веома битан фактор који утиче на то колико ће добре одлуке доносити бот.

Бот помера белу краљицу са *g3* на *g6*, што се може видети на слици 6, На први поглед ово изгледа као веома лош потез јер је логично да ће након тога црни играч појести краљицу пешаком који стоји на *f7*.



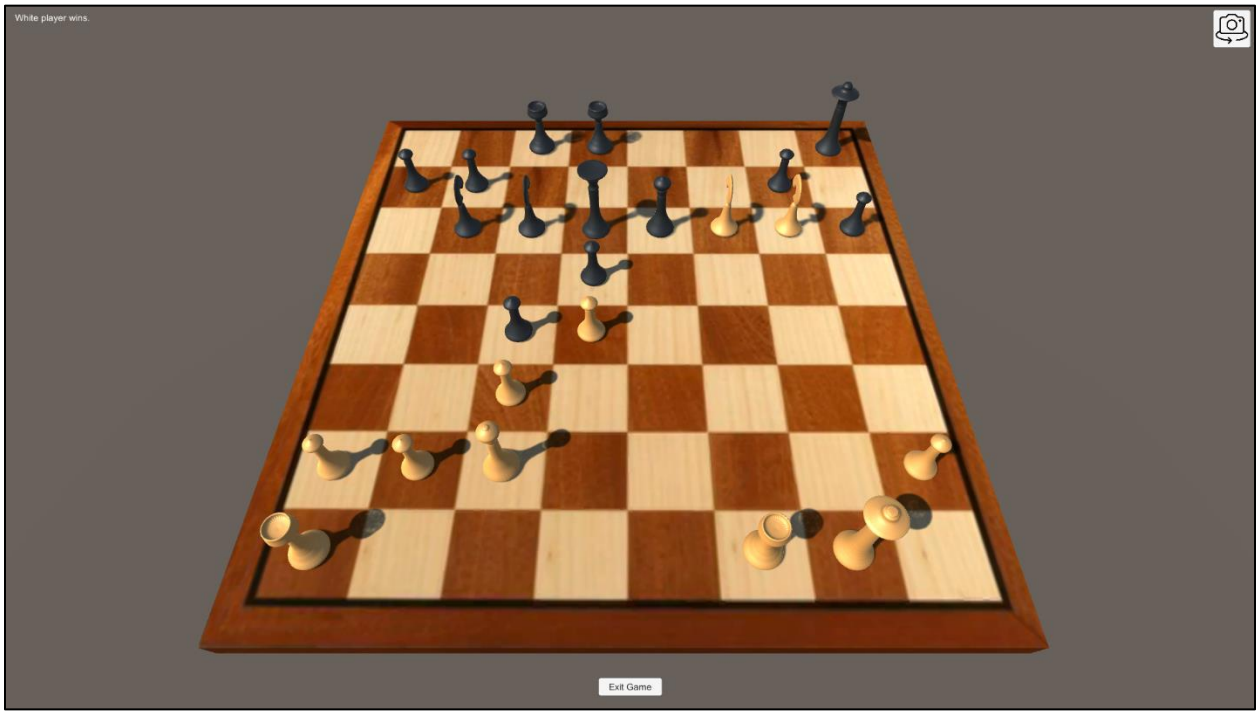
Слика 6 – Стање након потеза *g3g6*

Црни играч пешаком који стоји на *f7* једе белу краљицу и помера се на *g6* и ово делује као одличан потез јер пешак узима краљицу. Стање табле након овог потеза се може видети на слици 7.



Слика 7 – Стање након потеза *f7g6*

У следећем кораку бели играч скакачем који стоји на *e5* узима пешака са *g6*. Задаје шах-мат и игра је завршена што се може видети на слици 8. Исписује се порука у горњем левом углу да је бели играч победник, табла и фигуре више не реагују на догађаје миша и кориснику преостаје да се врати у главни мени кликом на дугме *Exit Game*.

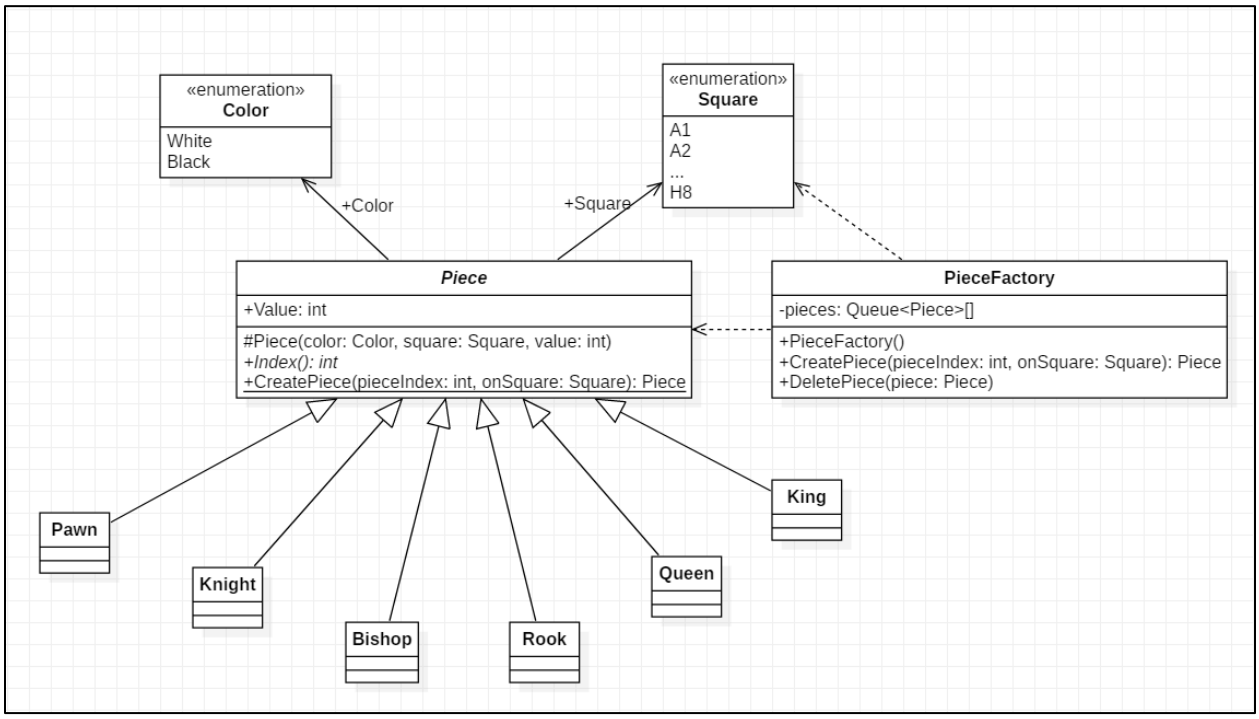


Слика 8 – Стање након потеза *e5g6*

4. Детаљи имплементације решења

4.1 Фигуре

Апстрактна класа *Piece* садржи основне информације о фигури као што су боја фигуре, поље на ком се тренутно налази и вредност фигуре. Сви типови шаховских фигура су изведени из ове класе и позивају заштићени конструктор надкласе при креирању. Приликом генерације потеза долази до хиперпродукције објеката фигура. Проблем је превазиђен помоћу класе *PieceFactory* која представља фабрику помоћу које се креирају фигуре. Фабрика садржи низ од 12 редова (6 типова фигура беле и црне боје) и приликом позивања методе *CreatePiece* долази до провере да ли већ постоји претходно креиран објекат типа *Piece* који је доступан. Ако је доступан, уклања се из реда. У супротном, креира се нов објекат и враћа као резултат методе. Позивањем методе *DeletePiece* прослеђени објекат се враћа у ред. Решење представља имплементацију пројектног узорка *Object pool pattern* и примењује се како би се побољшале перформансе програма.



Слика 9 – *UML* дијаграм класа који описује фигуре

4.2 Табла битова

Bitboards, називане и као сетови битова или мапе битова представљају један од начина репрезентације шаховске табле. Представљене су помоћу 64-битне целобројне вредности где сваки бит даје информацију о одређеном пољу. Битским операцијама је могуће на брз начин добити одговор на питање да ли одређено поље садржи одређену фигуру. Још једна добра страна је што заузимају малу количину меморије у односу на количину информација коју пружају. Погодне су и за конкурентно програмирање јер операције над различитим пољима могу да се извршавају паралелно што убрзава евалуацију. У решењу су искоришћене за праћење поља на којима се налазе пешаци.

4.3 Репрезентација шаховске табле

Шаховска табла се састоји од 64 поља где је свако поље једнозначно одређено колоном и редом у ком се налази. Иако би било могуће направити низ од 64 елемента где би сваки елемент представљао једно поље, а вредност елемента представљала фигуру која се налази на том пољу, табла је проширена. Низ заправо има 120 елемената и визуелно је представљен на слици 5. Поља означена сивом бојом не припадају табли и вредност тих елемената низа је вредност референце на *singleton* класе *OffLimits.* Поља на којима нема фигура имају вредност *null,* а поља на којима их има имају вредност референце на конкретне фигуре које им припадају. Два реда граничних вредности на врху и дну табле су неопходна како би се осигурало да чак и потенцијална места скока скакача дају као резултат валидне индексе низа у границама од 0 до 119. На овај начин је генерација потеза поједностављена о чему ће више речи бити у поглављу 4.7.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119

	a	b	c	d	e	f	g	h
1	21	22	23	24	25	26	27	28
2	31	32	33	34	35	36	37	38
3	41	42	43	44	45	46	47	48
4	51	52	53	54	55	56	57	58
5	61	62	63	64	65	66	67	68
6	71	72	73	74	75	76	77	78
7	81	82	83	84	85	86	87	88
8	91	92	93	94	95	96	97	98

Слика 10 – Приказ шаховске табле

4.4 Структура шаховске табле

Класа *Board* садржи неопходне информације о стању табле:

```
public class Board
{
        private PieceFactory pieceFactory;

        public Piece[] Pieces { get; set; }
        public Bitboard[] Pawns { get; set; }
        public Square[] Kings { get; set; }
        public PieceList PieceList { get; set; }
        public int[] Material { get; set; }

        public Color OnTurn { get; set; }
        public int FiftyMove { get; set; }
        public Square EnPassant { get; set; }
        public int CastlePerm { get; set; }

        public int HistoryPly { get; set; }
        public int Ply { get; set; }
        public ulong StateKey { get; set; }

        public LinkedList<UndoMove> History { get; set; }
        public PvTable PvTable { get; set; }
        public List<Move> PvMoves { get; set; }

        ...
}
```

Води се рачуна о позицији свих фигура на табли и додатно пешака и краљева ради поједностављења алгоритама. *Material* представља суму вредности фигура и прати се за оба играча. *OnTurn* чува информацију о томе који играч је на потезу. *FiftyMove* је бројач који је уведен да би се испоштовало правило о педесет потеза. До завршетка пројекта предвиђени бројач није искоришћен што је могуће учинити у наредној верзији апликације. *EnPassant* је поље које се памти у наредном потезу након што играч направи почетни потез пешаком два поља унапред. Правило је имплементирано већ у овој верзији као и правило рокаде где *CastlePerm* представља информацију о томе да ли је варијанта рокаде дозвољена. *HistoryPly* је бројач потеза од почетка игре, а *Ply* бројач потеза током претраге (дубина претраге). *StateKey* је 64-битни хеш који служи као кључ у хеш табели *PvTable* о којој ће бити више речи у поглављу 4.11. *PvMoves* је секвенца најбољих потеза који су пронађени у току претходне претраге. *History* је листа потеза одиграних од почетка партије и служи да би било могуће враћати се у претходна стања што је неопходно при претрази потеза.

4.5 Форсајт-Едвардс нотација

Објекат типа *Board* креира се тако што се проследи *string* у Форсајт-Едвардс нотацији. *FEN* је стандардна нотација за опис конкретне шаховске позиције. Сврха јој је да омогући покретање игре из позиције која се разликује од стандардне описане правилима шаха. *FEN* запис састоји се од низа *ASCII* карактера у једној линији и садржи 6 поља. Сепаратор поља је *space* карактер.

Поља нотације су редом:

1. Распоред фигура из перспективе белог играча. Редови су описани редом од 8. до 1. У оквиру сваког реда садржај сваког поља је описан колоном од почев од a до h. Свака фигура је једнозначно одређена првим карактером имена фигуре на енглеском језику. Беле фигуре се означавају карактерима „*PNBRQK*”, а црне фигуре малим карактерима на исти начин. Празна поља се означавају цифрама од 1 до 8 (број празних поља), а „/” раздваја редове.
2. Боја играча на потезу. „*W*” значи да бели игра следећи, а „*B*” црни.
3. Дозвола за рокаду. Поље има вредност једног или више од карактера „*KQkq*”, у зависности од тога који играч сме да изврши малу или велику рокаду. Уколико ни једној страни није дозвољена рокада поље има вредност „-”.
4. Уколико је у претходном потезу пешак направио почетни потез два поља унапред, памти се *En passant* поље које представља поље иза пешака. У супротном, поље има вредност „-”.
5. Бројач полупотеза. Број који говори колико је полупотеза прошло од последњег уклањања фигуре са табле или помераја пешака. Користи се да би се одредило да ли је могуће да играч затражи нерешених исход користећи правило о педесет потеза.
6. Бројач потеза. Број који почиње са 1 и инкрементира се након потеза црног играча.

Пример:

1. Почетна позиција:
   rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
2. Бели помера пешака са *e2* на *e4*:
   rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1
3. Црни помера пешака са *c7* на *c5*:
   rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR w KQkq c6 0 2
4. Бели помера скакача са *g1* на *f3*:
   rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2

4.6 Хеш генератор

Због постојања класе *PvTable* која је практично једна хеш мапа у којој је кључ стање табле, а вредност претходно пронађен најбољи потез, било је неопходно на неки начин разликовати стања табле. Рад програма се заснива на једном објекту типа *Board* над којем се извршавају и поништавају потези.

Хеш генератор се иницијализује при позиву статичког конструктора класе *Board* на следећи начин:

```
private static void InitHashGenerator()
{
	pieceKeys = new ulong[Constants.PieceTypeCount,
	Constants.BoardSquareCount];

	for (int i = 0; i < Constants.PieceTypeCount; ++i)
	{
		for (int j = 0; j < Constants.BoardSquareCount; ++j)
		{
			pieceKeys[i, j] = Get64BitRandom();
		}
	}

	sideKey = Get64BitRandom();
	castleKeys = new ulong[16];

	for (int i = 0; i < 16; ++i)
	{
		castleKeys[i] = Get64BitRandom();
	}
}
```

Све вредности су 64-битни неозначени цели бројеви. Коначан хеш се добија тако што се на сва поља која утичу на кључ редом примени битска операција ексклузивно или. Следећа поља утичу на кључ стања:

- *pieceKeys* се индексира типом фигуре и индексом поља на ком се налази и користи се за сваку фигуру на табли, а индекс 0 служи за *En passant* поље уколико постоји
- *sideKey* се користи уколико је играч на потезу бели
- *castleKeys* се користи за дозволе рокаде које су представљене са 4 бита, те имају вредности од 0 до 15

Ипак, теоретски је могуће да се вредности два хеша поклопе те је при анализи улаза у *PvTable* потребно проверити да ли је вредност припада тренутном стању или не. О томе ће бити више речи у поглављу 4.11.

4.7 Генерација потеза

Класа *Move* садржи информације о потезу. Како се при претрази генерише огроман број потеза пожељно је да потез има што мање заузеће меморије. Потез је реализован као целобројна вредност где се битским операцијама долази до информација од значаја. Састоји се од поља са ког се фигура помера, поља на које се фигура помера, поједене фигуре уколико је постојала и фигуре у коју је извршена промоција ако је потез био промоција пешака. Прати се и да ли је потез био рокада, *En passant* или први потез пешака што је битно због ажурирања *EnPassant* поља табле. Поменуте информације заузимају следеће битове:

```
/*
       0000 0000 0000 0000 0000 0111 1111 -> FromSq
       0000 0000 0000 0011 1111 1000 0000 -> ToSq
       0000 0000 0011 1100 0000 0000 0000 -> CapturedPiece
       0000 0000 0100 0000 0000 0000 0000 -> IsEnPassant
       0000 0000 1000 0000 0000 0000 0000 -> IsPawnStart
       0000 1111 0000 0000 0000 0000 0000 -> PromotedPiece
       0001 0000 0000 0000 0000 0000 0000 -> IsCastle
*/
```

Позивом методе `GenerateAllMoves()` генеришу се сви псеудолегални потези. Потез је легалан уколико не оставља краља у шаху што се проверава тако што се потез повуче позивом методе позивом методе `DoMove(Move move)` која након што ажурира неопходна поља табле обави позив `IsSquareAttacked(Kings[(int)OnTurn], (Color)((int)OnTurn ^ 1))`. Уколико позив врати вредност *true* аутоматски се све промене поништавају, уколико не, потребно је додатно позвати методу `UndoMove()`.

Псеудолегални потези се генеришу уз помоћ дводимензионалног статичког низа који дефинише смерове кретања фигура:

```
private static readonly int[,] PieceDirection = {
       { -9, -11, 11, 9 }, // Bishop
       { -1, -10,  1, 10 }, // Rook
       { -1, -10,  1, 10, -9, -11, 11, 9 }, // Queen
       { -8, -19,  -21, -12, 8, 19, 21, 12 }, // Knight
       { -1, -10,  1, 10, -9, -11, 11, 9 }, // King
};
```

Низ се индексира типом фигуре и бројем итерације петље која генерише потезе. Пешаци се крећу на посебан начин и њихов начин кретања се не налази у овом низу. Постоји подела на клизајуће фигуре у које спадају топ, ловац и краљица и на неклизајуће фигуре у које спадају скакач и краљ. Самим тим се једна вредност из низа за клизајуће фигуре користи све док се не стигне до граничне вредности или друге фигуре.

4.8 Тестирање генерације потеза

Пре него што се дође до евалуације и претраге потеза од кључног значаја је тестирати генератор потеза. Уколико алгоритам није у потпуности исправан може доћи до проблема при претрази где нека од подстабала игре неће постојати. Тиме противник може да стекне одређену предност над ботом.

Решење је тестирано помоћу *Perft* функције која обилази стабло игре и броји листове стабла на одређеној дубини за које постоје већ утврђене очекиване вредности. Може се користити и као показатељ брзине генератора потеза.

Тест приказан на слици 11 је обухватио шест карактеристичних позиција, за сваку од њих пребројао листове на дубини 4 и упоредио са очекиваним вредностима. Након успешног тестирања може се са великом сигурношћу рећи да генератор потеза ради потпуно тачно због тога што су за тестирање изабране специфичне позиције које су у прошлости задавале доста проблема људима који су се бавили развојем виртуелних играча за шах.

```
[TestMethod()]
public void PerftTest()
{
    /* Position FEN, MoveCount at depth 4
     * https://www.chessprogramming.org/Perft_Results
     */
    Dictionary<string, int> positionMoveCountPairs = new Dictionary<string, int>()
    {
        { "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1", 197281 },
        { "r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - ", 4085603 },
        { "8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -", 43238 },
        { "r2q1rk1/pP1p2pp/Q4n2/bbp1p3/Np6/1B3NBn/pPPP1PPP/R3K2R b KQ - 0 1", 422333 },
        { "r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1", 422333 },
        { "rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8", 2103487 },
        { "r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w - - 0 10", 3894594 }
    };

    Perft perft = new Perft();
    foreach (var entry in positionMoveCountPairs)
    {
        Board b = new Board(entry.Key);
        Assert.AreEqual(entry.Value, perft.CountLeafNodes(b, 4));
    }
}
```

Слика 11 – *Perft* тестирање

4.9 Евалуација потеза

Евалуација подразумева хеуристичку функцију која рачуна релативну вредност позиције, односно шансу за победу. У пракси је веома тешко изралунати тачну вредност позиције па је неопходно направити довољно добру апроксимацију. Почетници у шаху временом уче да процењују вредност потеза тако што искуством стичу слику о томе колико је која фигура вредна.

Виртуелни играчи такође највише узимају у обзир разлику сума вредности сопствених и противничких фигура. То је први корак при евалуацији. У решењу су следеће вредности узете као показатељ вредности фигуре:

- Пешак – 100 поена
- Скакач – 325 поена
- Ловац – 325 поена
- Топ – 550 поена
- Краљица – 1000 поена
- Краљ – 50000 поена

Побољшање основне евалуације које је примењено јесте *Piece-Square Tables.* Креирањем низова од 64 елемента за сваки тип фигуре, могуће је евалуирати колико је добра позиција фигуре на табли. Нека решења имплементирају и више табела ове врсте које користе у зависности од тога да ли је игра тек започела или се ближи крају.

```
private static readonly int[] PawnPositionValue = {
0  ,  0  ,  0  ,  0   ,  0   ,  0   ,  0   ,  0  ,
10 ,  10 ,  0  ,  -10 ,  -10 ,  0   ,  10  ,  10 ,
5  ,  0  ,  0  ,  5   ,  5   ,  0   ,  0   ,  5  ,
0  ,  0  ,  10 ,  20  ,  20  ,  10  ,  0   ,  0  ,
5  ,  5  ,  5  ,  10  ,  10  ,  5   ,  5   ,  5  ,
10 ,  10 ,  10 ,  20  ,  20  ,  10  ,  10  ,  10 ,
20 ,  20 ,  20 ,  30  ,  30  ,  20  ,  20  ,  20 ,
0  ,  0  ,  0  ,  0   ,  0   ,  0   ,  0   ,  0
};
private static readonly int[] KnightPositionValue = {
0  ,  -10 ,  0   ,  0   ,  0   ,  0   ,  -10 ,  0  ,
0  ,  0   ,  0   ,  5   ,  5   ,  0   ,  0   ,  0  ,
0  ,  0   ,  10  ,  10  ,  10  ,  10  ,  0   ,  0  ,
0  ,  0   ,  10  ,  20  ,  20  ,  10  ,  5   ,  0  ,
5  ,  10  ,  15  ,  20  ,  20  ,  15  ,  10  ,  5  ,
5  ,  10  ,  10  ,  20  ,  20  ,  10  ,  10  ,  5  ,
0  ,  0   ,  5   ,  10  ,  10  ,  5   ,  0   ,  0  ,
0  ,  0   ,  0   ,  0   ,  0   ,  0   ,  0   ,  0
};
```

```
private static readonly int[] BishopPositionValue = {
0  ,  0   ,  -10 ,  0   ,  0   ,  -10 ,  0   ,  0  ,
0  ,  0   ,  0   ,  10  ,  10  ,  0   ,  0   ,  0  ,
0  ,  0   ,  10  ,  15  ,  15  ,  10  ,  0   ,  0  ,
0  ,  10  ,  15  ,  20  ,  20  ,  15  ,  10  ,  0  ,
0  ,  10  ,  15  ,  20  ,  20  ,  15  ,  10  ,  0  ,
0  ,  0   ,  10  ,  15  ,  15  ,  10  ,  0   ,  0  ,
0  ,  0   ,  0   ,  10  ,  10  ,  0   ,  0   ,  0  ,
0  ,  0   ,  0   ,  0   ,  0   ,  0   ,  0   ,  0
};
private static readonly int[] RookPositionValue = {
0  ,  0   ,  5   ,  10  ,  10  ,  5  ,  0  ,  0  ,
0  ,  0   ,  5   ,  10  ,  10  ,  5  ,  0  ,  0  ,
0  ,  0   ,  5   ,  10  ,  10  ,  5  ,  0  ,  0  ,
0  ,  0   ,  5   ,  10  ,  10  ,  5  ,  0  ,  0  ,
0  ,  0   ,  5   ,  10  ,  10  ,  5  ,  0  ,  0  ,
0  ,  0   ,  5   ,  10  ,  10  ,  5  ,  0  ,  0  ,
25 ,  25  ,  25  ,  25  ,  25  ,  25 ,  25 ,  25 ,
0  ,  0   ,  5   ,  10  ,  10  ,  5  ,  0  ,  0
};
```

Слика 12 – Табеле *Piece-Square*

На овај начин, пешаци се мотивишу да крену напред и да се промовишу ако је могуће. Скакач, ловац, топ и краљица су највреднији када су у средини табле јер имају велики радијус кретања. За краљицу и ловца је коришћена иста табла. За краља се користе посебне технике које нису обрађиване у решењу.

4.10 Претрага потеза

Алгоритам претраге који је имплементиран је *Negamax* алгоритам са алфа-бета одсецањем. *Negamax* је варијанта *Minimax* алгоритма која се уместо две одвојене методе за *Min* играча и *Max* играча ослања на негиран резултат на основу математичке релације: *max(a, b) = -min(-a, -b)*

Замислимо да је бели играч на потезу и да тражимо потезе до дубине 2 (разматрамо све потезе белог играча и све одговоре црног на тај потез). Рецимо да одаберемо потез број 1 и након разматрања свих одговора црног играча дођемо до закључка да је вредност тог потеза 0. Ако кренемо да разматрамо потез број 2 и дођемо до првог контра потеза црног играча који за резултат има да бели губи топа, можемо са сигурношћу да игноришемо све остале потезе црног играча на потез 2 зато што знамо да је за белог потез 1 бољи. Не занима нас колико је тачно потез 2 лошији јер је анализа потеза 1 дала доњу границу и бели играч сигурно неће бирати потез лошији од тог. Ситуација постаје компликованија уколико претрагу вршимо до дубине 3 или више зато што тада оба играча могу да праве изборе који утичу на стабло игре. Тада је неопходно пратити и доњу и горњу границу које се називају алфа и бета. Доња граница је битна због тога што ако је потез лош не желимо да га разматрамо, али исто тако и горња граница због тога што ако је потез исувише добар, противник нам неће дозволити да га одиграмо јер је претходно имао потез који је могао да одигра како би избегао ову ситуацију. Доња граница једног играча је горња граница другог.

Смањење броја чворова које пружа алфа-бета одсецање може бити значајно. Ако стандардно *Minimax* стабло има *x* чворова, добро написан програм који примењује алфа-бета одсецање може обићи број чворова који је близу квадратном корену из *x*. Број чворова који ће бити евалуиран зависи од тога колико је чворова одсечено, што зависи од тога колико је добар поредак чворова у стаблу игре. Уколико се одмах претражи најбољи потез, већина осталих чворова ће бити елиминисана. Проблем је што је најбољи потез непознат, иначе не би било ни потребе за претрагом, па је неопходно служити се разним апроксимацијама како би број одсечених чворова био што већи.

Имплементирана је хеуристика са слике 13 која се назива *Most Valuable Victim – Least Valuable Aggressor*. У оквиру такозваног циклуса тражења жртве, потребно је прво претражити потезе где је потенцијална жртва једна од нападнутих противничких фигура у опадајућем поретку вредности, дакле краљица, топ, ловац, скакач и на крају пешак. Након што је највреднија жртва пронађена тражи се најслабија фигура која је напада, дакле у обрнутом редоследу од наведеног.

```
private static readonly int[] VictimScore =
{
    0, 100, 200, 300, 400, 500, 600, 100, 200, 300, 400, 500, 600
};
public static readonly int[,] MvvLvaScores = new int[Constants.PieceTypeCount, Constants.PieceTypeCount];

private static void InitMvvLva()
{
    for (int vic = 1; vic < Constants.PieceTypeCount; ++vic)
    {
        for (int att = 1; att < Constants.PieceTypeCount; ++att)
        {
            MvvLvaScores[vic, att] = VictimScore[vic] + 6 - (VictimScore[att] / 100);
        }
    }
}
```

Слика 13 – Иницијализација *MVV-LVA* хеуристике

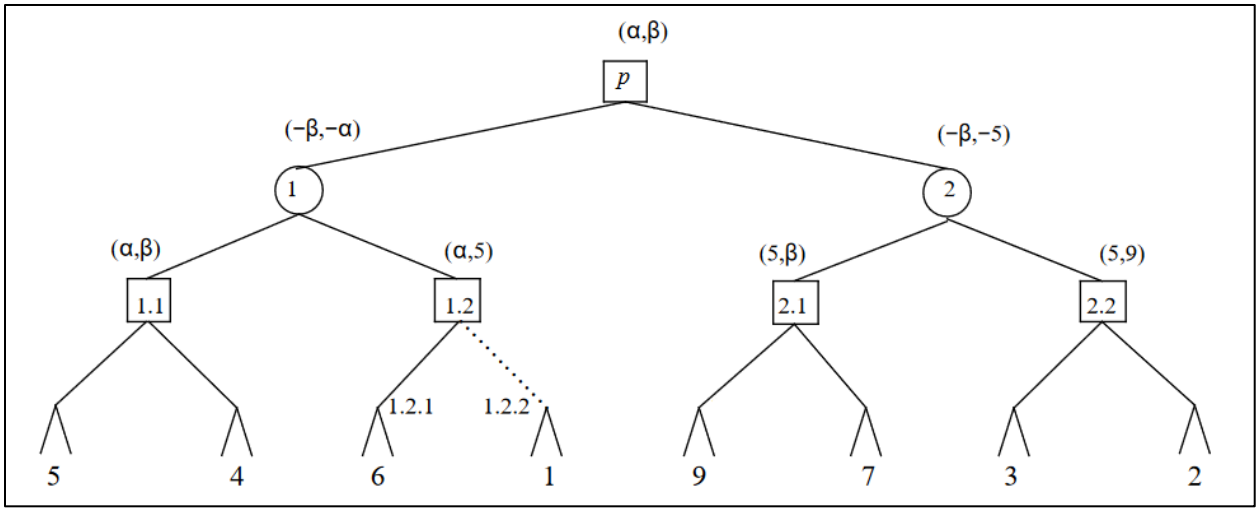
При претрази коришћена је техинка итеративног продубљивања која је популарна стратегија у претрагама по дубини и погодна је у ситуацијама када постоји временски лимит. Функционише тако што програм почиње са претрагом на следећој дубини, затим се враћа на почетак, инкрементира дубину и претражује до новодобијене дубине. Процес се понавља све док време за претрагу није истрошено. У случају незавршене претраге, постоји *Principal Variation* линија о којој ће бити више речи у поглављу 4.11. Она је изузетно битна при одређивању поретка потеза који се претражују. Уколико се прво претраже *PV* потези, а затим по *MVV-LVA* хеуристици програм је показао да се стабло игре значајно смањује, у неким случајевима и до 50%.

```
for (int currentDepth = 1; currentDepth <= info.DepthLimit; ++currentDepth) // iterative deepening
{
    bestScore = AlphaBeta(-Constants.Infinity, Constants.Infinity, currentDepth, info);

    GetPvLine(currentDepth);
    bestMove = PvMoves.Count > 0 ? PvMoves[0] : null;

    if (currentDepth == info.DepthLimit)...
}
```

Слика 14 – Итеративно продубљивање

Део Negamax алгоритма са алфа-бета одсецањем који је позиван током итеративног продубљивања:

```
MoveList moveList = GenerateAllMoves();
for (int i = 0; i < moveList.Count; ++i){
        Move move = moveList.PickNextMove(i);
        if (!DoMove(move)) continue;
        ++legalMovesCount;
        score = -AlphaBeta(-beta, -alpha, depth - 1, info);
        UndoMove();
        if (score >= beta) return beta;
        if (score > alpha) {
                alpha = score;
                bestMove = move;
        }
}
```



Слика 15 – Алфа-бета одсецање при *Negamax* алгоритму

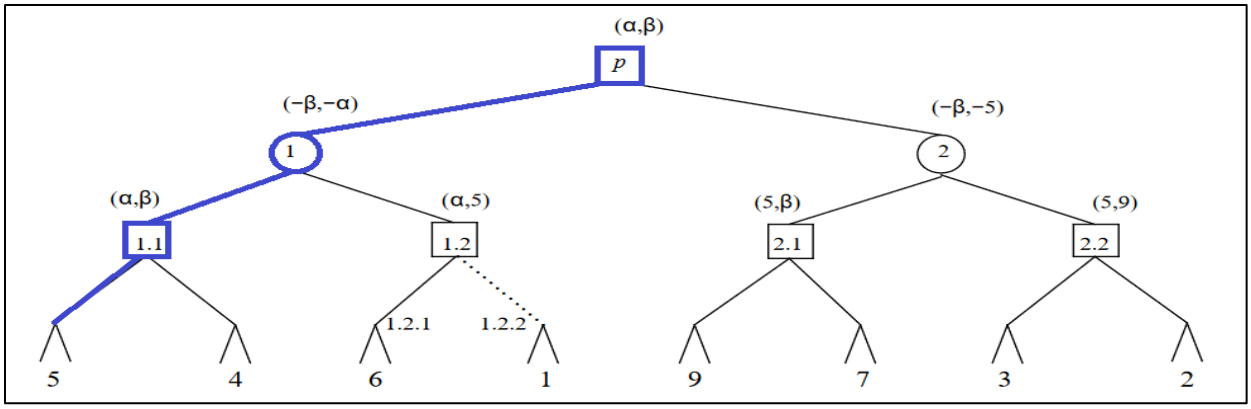
Већ је речено да је Negamax варијанта Minimax алгоритма која је једноставнија за имплементацију. На слици 15 се може видети једноставно стабло игре дубине 3 у којој су чворови обележени у нотацији која идентификује редослед обиласка чворова. Листови практично представљају корене скривених подстабла и дата је процена њихове вредности. Такође, сваки чвор поред себе има иницијалне вредности за алфа и бета које је добио при претрази. Чвор од значаја има ознаку 1.2 и има иницијалне вредности (алфа, 5). Како је вредност чвора 1.2.1 већа од 5, догодиће се одсецање и чвор 1.2.2 неће бити посећен. Циљ је да се број одсечених чворова максимизује што је могуће постићи тако што се апроксимирају чворови са потенцијално највећом вредношћу и обићу први. То је у програму постигнуто на следећи начин:

```
public Move PickNextMove(int moveIndex)
{
	int bestScore = 0, bestIndex = moveIndex;
	for (int i = moveIndex; i < Count; ++i)
	{
		if (Moves[i].Score > bestScore)
		{
			bestScore = Moves[i].Score;
			bestIndex = i;
		}
	}
	Swap(moveIndex, bestIndex);
	return Moves[moveIndex];
}
```

Ова метода представља неку врсту *Selection sort* алгоритма, што није најбољи алгоритам сортирања, али је искоришћен за решење проблема из разлога што је једноставан за имплементацију. Низови који се претражују нису велике дужине и обично дође до одсецања и пре него што се цео низ сортира. Ипак, најбоље решење би дефинитивно био *Heap sort* јер се увек тражи следећи максимум низа, што представља могуће побољшање решења.

4.11 *Principal Variation* табела

Principal Variation је секвенца потеза за које програм процењује да су најбољи и очекује да буду одиграни (чворови обојени у плаво на слици 16). Уколико се користи стратегија итеративног продубљивања изузетно је битно да се у следећој итерацији обиђу *PV* чворови претходне итерације јер се тиме повећава број чворова који ће бити одсечени при претрази. Иако није неопходно, већина имплементација виртуелних играча исписује *PV* сваког пута кад се она промени или је достигнут следећи ниво дубине јер је интересантна за анализу.



Слика 16 – Principal Variation линија

Класа *PvTable* представља имплементацију хеш табеле која као кључ има хеш вредност стања табле, а као вредност објекат класе типа *PvTableValue* која се може видети на слици 17. Вредност се састоји од најбољег пронађеног потеза за дато стање, вредности потеза, дубине на којој је пронађен потез и хеш вредности стања табле за коју је потез најбољи.

Логично питање је зашто хеш вредност стања табле постоји и као кључ и као вредност улаза. С обзиром на то да је табела ограничене величине могуће је да дође до колизије јер се за кључ користи остатак при дељењу хеш вредности стања са бројем улаза табеле, као што је приказано на слици 17. Због тога је неопходно при дохватању вредности проверити да ли добијена вредност потиче од прослеђеног кључа или од неког другог кључа који се мапира на исти улаз.

```
public bool TryGetValue(ulong StateKey, out PvTableValue value)
{
    ulong key = StateKey % MaxEntries;
    value = values[key];
    if (value == null || value.StateKey != StateKey)
    {
        value = null;
        return false;
    }
    return true;
}
```

```
public class PvTableValue
{
    public Move Move { get; set; }
    public int Score { get; set; }
    public int Depth { get; set; }
    public ulong StateKey { get; set; }
};
```

Слика 17 – Начин дохватања вредности табеле и структура вредности

4.12 Тродимензионална видео игра

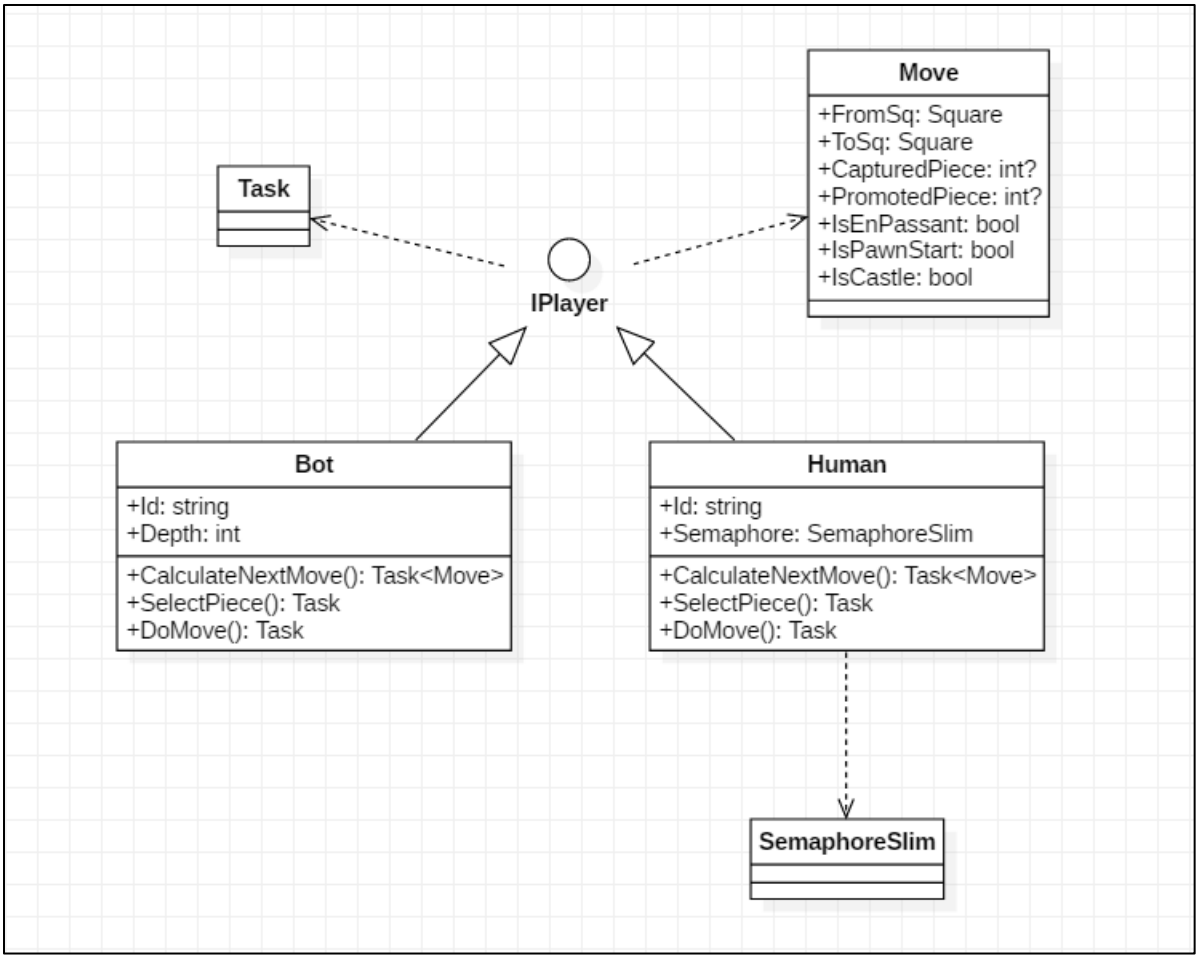
Визуелни део апликације је развијен у окружењу за развој игара *Unity* као *3D* игра за *desktop*. Омогућава корисницима да креирају апликације пружајући одличан *API* за програмски језик *C#*. Приликом израде рада, коришћени су бесплатни модели фигура и табле са *Unity* продавнице.

Креиране су две сцене: главни мени и игра. У сцени игре постоји неколико објеката игре и које размењују податке међусобно током времена:

- *EventManager* – реагује на догађаје које корисник производи кликом на лево дугме миша
- *GameController* – води рачуна о току игре, позива методе *GameUiController*-а, *Spawner*-а и *Visualizer*-а и обезбеђује да корисничко искуство буде на адекватном нивоу
- *GameUiController* – ажурира текстуална обавештења и реагује на захтев за ротацију камере
- *Spawner* – алоцира и деалоцира објекте игре
- *Visualizer* – задужен за визуелне ефекте

У игри је коришћен принцип асинхроног програмирања помоћу класе *Task* и кључних речи *async-await*. У главној петљи игре програм чека да се заврше асинхрони послови који су започети како би наставио са извршавањем. То омоућава да се игра не замрзне, већ дозвољава интеракцију са апликацијом. Уколико је на потезу бот корисник мора да сачека да бот направи потез, у супротном чека се корисник да изабере фигуру и поље на које ће је померити. Игра је завршена уколико играч који је на реду нема могућих потеза. Главна петља игре изгледа овако:

```
async void StartGame()
{
	while (true)
	{
		OnTurn = Players[(int)Board.OnTurn];
		if (NoPossibleMoves()) break;

		Move move = await OnTurn.CalculateNextMove();

		if (OnTurn is Bot)
		{
			Bot bot = OnTurn as Bot;
			UiController.ShowSearchInfoText(bot.LastSearchResult);
			SelectPiece((int)move.FromSq);
			DoMove((int)move.ToSq);
		}
		else
		{
			await OnTurn.SelectPiece();
			if (SelectedPiece == null) continue;
			await OnTurn.DoMove();
		}
	}
	EndGame();
}
```



Слика 18 – *UML* дијаграм класа који описује играче

Сваки тип играча реализује интерфејс *IPlayer*. Класа *Bot* у методи *CalculateNextMove* врши претрагу што се види на слици 19 и тада су кориснику онемогућене контроле, док класа *Human* враћа *dummy* вредност потеза јер се очекује да изабере фигуру и поље на које помера фигуру. Након претраге, потез се аутоматски извршава.

```
public Task<Move> CalculateNextMove()
{
    var task = Task.Run(() =>
    {
        SearchInfo searchInfo = new SearchInfo
        {
            DepthLimit = Depth
        };
        LastSearchResult = GameController.Board.SearchPosition(searchInfo);
        return Game
    });
    GameController.UiController.ShowInputInfoText(Id + " (Bot) is calculating.");
    return task;
}
```

string Bot.LastSearchResult { get; set; }

Слика 19 – Имплементација методе *CalculateNextMove* у класи *Bot*

За разлику од класе *Bot*, класа Human користи семафор како би сачекала на акцију корисника што се види на слици 20.

```
public Task SelectPiece()
{
    var task = Task.Run(() => { Semaphore.Wait(); });
    GameController.UiController.ShowInputInfoText(Id + " (Human), select piece to move.");
    return task;
}

public Task DoMove()
{
    var task = Task.Run(() => { Semaphore.Wait(); });
    GameController.UiController.ShowInputInfoText(Id + " (Human), select square to move.");
    return task;
}
```

Слика 20 – Имплементација методе *SelectPiece* и *DoMove* у класи *Human*

EventHandler прихвата акције и шаље их *GameController*-у који ослобађа семафор након што обради захтев, погледати слику 21.

```
private void ReleaseHumanSemaphore()
{
    Human human = OnTurn as Human;
    if (human != null)
    {
        if (human.Semaphore.CurrentCount == 0) human.Semaphore.Release();
    }
}
```

Слика 21 – Сигнализирање семафора у класи *GameController*

4.13 *Dependency Injection*

Објекти игре који припадају сцени у *Unity*-ју се уништавају приликом промене активне сцене. У решењу је било неопходно дохватити све параметре које корисник унесе у главном менију и сачувати их тако да буду доступни и након активације следеће сцене. Такође, било је потребно креирати објекат типа *Board* и проследити га сцени игре.

У софтверском инжењерству постоји техника која се зове *Dependency Injection.* Помоћу ње се омогућава да један објекат пружи све неопходне зависности другом објекту. Ова техника је једна форма шире технике која се назива инверзија контроле. Клијент делегира одговорност креирања објеката другом објекту који се назива *Injector*. Он је задужен да креира све потребне сервисе и ослобађа клијента одговорности да познаје начин креирања објеката већ је потребно само да познаје интерфејс који му је потребан да би им приступио.

Креирана је класа *InjectionContext* која у себи садржи *Dictionary<string, object>* у који се умећу све зависности које су живе све време рада апликације. Поседује методе које умећу и разрешавају зависности.

Након тога је креирана статичка класа *SceneLoading* која садржи статички објекат типа *InjectionContext* и унутрашњу статичку класу под називом *Parameters* у којој се дефинишу називи зависности који су неопходни за дохватање одговарајућих објеката.

У класи *MenuUiController* се након клика на *Start Game* иницијализује објекат типа Board и позива се следећа метода како би *GameController* могао да му приступи:

```
SceneLoading.Context.Inject(SceneLoading.Parameters.Board, board);
```

Све што *GameController* треба да уради како разрешио зависност је:

```
Board = SceneLoading.Context.Resolve<Board>(SceneLoading.Parameters.Board);
```

На овај начин се раздвајају одговорности креирања и коришћења објеката и повећава читљивост кода.

5. Закључак

Циљ дипломског рада био је развој виртуелног играча за шах који би био достојан противник играчима који су аматери или почетници. Тренутно евалуира приближно 400 хиљада потеза у секунди што је задовољавајуће за такву врсту игре.

Развијена *3D* игра може служити и за разоноду и као систем за вежбање шаха где корисник осим тога што размишља о потезима које прави, може да види резултате претраге бота, да размисли о добијеним резултатима и на тај начин подиже свој ниво игре. Корисник може вршити и симулацију игре два бота и посматрати какве одлуке доносе.

Како би се евалуирао ниво игре бота неопходно је креирати протокол за комуникацију као што је *Universal Chess Protocol*. Након тога би бот могао да комуницира са графичким корисничким интерфејсима који подржавају исти протокол, учествује у такмичењу против осталих ботова и добије коначну оцену.

У раду су разматрана постојећа решења која су у великој мери утицале на развој система. Након тога је дат опис рада система који омогућава корисницима да се упознају са системом и његовим функционалностима. Приказан је и ток једне партије како би се приказао начин размишљања бота у пракси. У детаљима имплементације је приказано како систем функционише изнутра ради бољег разумевања рада апликације. Дискутовано је о најбитнијим компонентама система као што су репрезентација шаховске табле, генерација и евалуација потеза и претрага потеза. Приказан је и начин функционисања визуелног дела апликације.

У наставку ће бити размотрени неки од недостатака система и могућности за њихово отклањање.

Рачунарски играч је функционалан, али поседује недостатке од којих је главни ефекат хоризонта што је први проблем који би требало да се отклони током надоградње система. Заустављање претраге након што се дође до жељене дубине може довести до погрешних закључака. Проблем би могао да се отклони *Quiescence* претрагом. Идеја би била се креира метода у класи *Board* слична *GenerateAllMoves()*. Назовимо је *GenerateAllCaptures()* и њена функција би била да генерише потезе помоћу којих долази до елиминације противничке фигуре. Након тога било би потребно креирати *Quiescence()* методу која би се позивала уместо *EvaluatePosition()* на последњем нивоу претраге. Псеудокод би изгледао овако:

```
int Quiescence(int alpha, int beta) {
    int stand_pat = Evaluate();
    if (stand_pat >= beta) return beta;
    if (alpha < stand_pat) alpha = stand_pat;

    until( every_capture_has_been_examined )  {
        MakeCapture();
        score = -Quiescence( -beta, -alpha );
        UndoMove();
        if(score >= beta) return beta;
        if(score > alpha) alpha = score;
    }
    return alpha;
}
```

Како бисмо избегли претраживање свих могућих елиминација фигура, на почетку евалуирамо стање и на тај начин одређујемо доњу границу резултата. Уколико је доња граница већа од бета, можемо вратити бета вредност. У супротном, претрага се наставља како би се установило да ли неки од потеза може повећати алфу.

Правило *Threefold repetition,* које каже да уколико се иста позиција догоди три пута играч може да затражи да меч буде завршен нерешеним резултатом, није имплементирано. Могућа надоградња би била да се у спрези са видео игром кориснику омогући да уместо дубине претраге, унесе време доступно за претрагу.

Што се тиче визуелног дела симулације, потребно је додати више визуелних ефеката, на пример да се кориснику означе дозвољена поља на које може да помери изабрану фигуру. Тренутно је проблематично и то што се померај фигуре од стране виртуелног играча обави у јако кратком временском интервалу па кориснику може бити нејасно који је потез направио бот. Требало би омогућити и снимање игре како би се могла наставити касније. Промоција фигуре се аутоматски ради у краљицу, што је коректно, али није потпуно исправно. Корисник би требало да има могућност да изабере да ли жели да промовише фигуру у топа, скакача, ловца или краљицу.

С обзиром на то да *Unity* омогућава развој апликација за више платформи, требало би омогућити да игрица функционише на различитим платформама као што су *iOS, Android, Windows Phone, PS4*…

Могуће је развити и неки вид *online* игре у варијанти човек против човека и организовати такмичарске мечеве како би корисници усавршавали свој ниво игре. Тада би игра могла и да се монетизује приказивањем реклама.

Након наведених измена, могућности за надоградњу виртуелног играча су неограничене. Постоје разни алгоритми који се могу применити како би се побољшале перформансе програма и подигао ниво игре бота. *DeepMind,* компанија која се бави истраживањем вештачке интелигенције, развила је програм 2017. године који је за 24 сата учења победио *Stockfish,* светског шампиона у категорији виртуелних играча шаха. Неуралне мреже су изузетно погодне за решавање проблема ове врсте. Могао би се применити и неки од алгоритама вештачке интелигенције као што је генетички алгоритам.

6. Литература

1. Репозиторијум информација о програмирању виртуелних играча шаха, линк: https://www.chessprogramming.org/
2. Документација за *Unity engine*, линк: https://docs.unity3d.com/Manual/
3. Докментација за *Microsoft .NET*, линк*: https://docs.microsoft.com/en-us/dotnet/*
4. Блог о процесу развоја игре *PicoCheckmate*, линк: https://krystman.itch.io/pico-checkmate
5. Тони Марсланд, „Рачунарски шах и претрага”, линк: http://webdocs.cs.ualberta.ca/~tony/RecentPapers/encyc.mac-1991.pdf
6. Стјуарт Расел, Питер Норвиг, „Вештачка интелигенција – Савремени приступ”, треће издање, 2011.
7. Денис Бреукер, „Меморија против претраге”, линк: http://www.dennisbreuker.nl/thesis/index.html
8. Ернст Хајнц, „Скалабилна претрага у рачунарском шаху”, 1999.
9. Фриц Реул, „Нове архитектуре у рачунарском шаху”, линк: https://pure.uvt.nl/ws/portalfiles/portal/1098572/Proefschrift_Fritz_Reul_170609.pdf
10. Дејвид Леви, Монти Њуборн, „Како рачунари играју шах”, 1990.
11. Алекс Бел, „Машина игра шах?”, 1978.
12. Тони Марсланд, Џонатан Шефер, „Рачунари, шах и когницијa”, 1990.
13. Стеф Лујитен, „Како написати шаховски програм у 99 корака”, линк: http://web.archive.org/web/20120621100214/http://www.sluijten.com/winglet/