

ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - ВАРНА
ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ



Катедра „Компютърни науки и технологии“

ДИПЛОМНА РАБОТА

Тема:

Проектиране и разработка на 2D видеоигра

Изготвил: Николай Иванов Иванов

Специалност: Компютърни системи и технологии

Факултетен номер: 17621301

ТУ Варна, 2021 г.

Ръководител: доц. д-р инж. Вл. Николов/

Съдържание

1. Постановка на дипломното задание. Цели и задачи на разработката.....	4
1.1 Въведение	4
1.2 Постановка на дипломното задание.....	4
1.3 Цели и задачи на разработката.	4
2. Обзор на използваните програмни средства и технологии.....	5
2.1 Избор на технологии.....	5
2.2 Python	5
2.2.1 Цели и способности.....	5
2.2.2 Софтуерна архитектура	6
2.3 Pygame.....	6
2.3.1 Софтуерна архитектура	6
2.3.2 Поддържани платформи	7
2.2.3 Кратка история	7
2.4 SQLite	7
2.4.1 Цели и способности.....	8
2.4.2 Архитектура	8
2.4.3 Кратка история	8
2.5 Photoshop.....	8
2.2.1 Цели и способности.....	9
2.2.2 Кратка история	9
3. Описание на приложението.....	10
3.1 Проектиране на системата.....	13
3.1.1 Actors (player and enemy)	13
3.1.2 Navigation.....	18
3.1.3 Combat.....	23
3.1.4 Items	25
3.1.5 Inventory.....	27
3.1.6 Buy & Sell	31
3.1.7 Player Progression	35
3.1.8 Game menus	39

3.1.9 Options	48
3.2 Описание на GUI.....	49
3.2.1 Navigation grid.....	49
3.2.2 Enemy party	51
3.2.3 Бутони	52
3.2.4 Pop up	54
3.3 Описание на използваните алгоритми.	55
3.4 Описание на базата от данни	57
3.5 Тестване и резултати.....	65
3.5.1 - A* и dijkstra's algorithm.....	65
3.5.2 Python vs C vs Java	66
4. Ръководство за използване	70
4.1 Начален екран	70
4.2 Нова игра	71
4.3 Зареждане на запаметени игри.....	72
4.3 Опции	73
4.4 Екран за навигация	74
4.5 Екран за Сражение	75
4.6 Екран за града	76
4.7 Екран за търговия.....	77
4.8 Екран за инвентара	78
4.9 Екран за прогресия на играча	79
5. Изводи и заключение	81
5.1 Процес на разработката.....	81
5.2 Бъдеще и надграждане на приложението.....	82

1. Постановка на дипломното задание.

Цели и задачи на разработката.

1.1 Въведение

Световната индустрия за видео игри е мулти милиарден пазар, който често действа огромен каталист за директно или косвено развитие и иновация в компютърните науки. Директен пример за това е интеграцията на Cuda ядра във видео картите и навлизането на RTX технологии в индустрията на видео игрите. Това развитие е довело до огромни скокове в развитието на компютърната сигурност, транспорта, медицината, развлечението, и десетки други индустрии.

Видео игрите могат да служат като развлечение, така и за демонстрация на концепции в компютърните науки и математиката.

1.2 Постановка на дипломното задание

Предмет на дипломната работа е проектирането на двуизмерна Role Playing видео игра, направена в стила на ранните видео игри от 90-те години на миналия век.

1.3 Цели и задачи на разработката.

Една от основните задачи на дипломната работа демонстрация на алгоритъм за оптимална навигация на двуизмерно пространство. Тази задача е осъществена чрез имплементацията на така наречен – „информиран алгоритъм за търсене“.

2. Обзор на използваните програмни средства и технологии.

2.1 Избор на технологии

Текущо разработваната видео игра се базира на Python и неговите библиотеки. За компютърната графика и визуализация се използва Python библиотеката Pygame. За съхраняване на информацията и прогреса на играча е използвана client-side базата данни- SQLite. Текстурите са преработени на Photoshop, Free use assets и sprites.

2.2 Python



Python е интерпретативен обектно ориентиран език за програмиране от високо ниво с общо предназначение.

2.2.1 Цели и способности

Python може да се използва за разработката на уеб, десктоп или мобилни приложения. Апликации разработени с езика python, притежават по-висока степен на четимост, което ги прави по-удобни за поддръжка и дебъгване. Той предлага добра структура и поддръжка за разработка на големи приложения. За разлика от много други езици за програмиране, Python предоставя по-голяма компактност и четимост. Това е така, понеже:

- Наличните сложни типове данни позволяват изразяването на сложни действия с един-единствен оператор;
- Групирането на изразите се извършва чрез отстъп, вместо чрез начални и крайни скоби или някакви ключови думи (друг език, използващ такъв начин на подредба, е Haskell);
- Не са необходими декларации на променливи или аргументи;
- Python съдържа прости конструкции, характерни за функционалния стил на програмиране, които му придават допълнителна гъвкавост.

2.2.2 Софтуерна архитектура

2.2.2.1 Интерпритатор

Поради факта, че Python е интерпретативен език, се спестява значително време за разработка, тъй като не са необходими компилиране и свързване за тестването на дадено приложение.

2.2.2.2 Модули

В езика са включени голям набор от стандартни модули, които да се използват като основа на програмите. Съществуват и вградени модули, които обезпечават такива събития и елементи като файлов вход/изход (I/O), различни си стемни функции, sockets, програмни интерфейси към GUI-библиотеки. Всеки модул на Python се компилира преди изпълнение до код за съответната виртуална машина. Този код се записва за повторна употреба като .рус файл.

2.2.2.3 Кратка история

Програмния език се заражда се в края на 1980-те години, като реалното осъществяване започва през декември 1989г. от Гуидо ван Росум в CWI. Python имал за цел да се превърне в наследник на ABC, който да бъде способен да обработва изключения и да е съвместим с операционната система Amoeba.

2.3 Pygame



Pygame е библиотека за графична визуализация, предназначена за разработката на видео игри, на програмния език Python. Тя включва модули за компютърна графика и звук.

2.3.1 Софтуерна архитектура - Pygame използва SDL(Simple DirectMedia Layer) библиотеката за реализацията на real-time game development, без да се алага да се ползват механики от езици като С и неговите производни. Това работи при положение, че по сложните функции за компутация могат да

бъдат абстрактнати с логика от сомото приложение, това позволява използването на език като Python за разработката на видео игри.

SDL библиотеката, на която се базира Pygame, позволява използването на векторна математика, дитекция на колизий, мениджмънт на двуизмерни анимаций, поддръжка на MIDI и камери.

2.3.2 Поддържани платформи - Pygame се поддържа: на

- Linux (pygame идва инсталирана на пвечето линкус дистрибуции),
- Windows (95, 98, ME, 2000, XP, Vista, 64-bit Windows, etc),
- BeOS,
- MacOS,
- Mac OS X,
- FreeBSD,
- NetBSD,
- OpenBSD,
- BSD/OS,
- Solaris,
- IRIX
- QNX.

2.2.3 Кратка история - Pygame е оригинално написана от Пийт Шиннерс, през 2000 година, с целта да замени PySDL, след като разработката ѝ спира. Библиотеката бързо се превръща в най-популярната библиотека за разработка на видео игри в python.

2.4 SQLite



SQLite OpenSource, Client Side, система за управление на релационни бази от данни, поддържаща стандарта SQL.

2.4.1 Цели и способности – Целта на SQLite е цялата база да бъде съсредоточена в един-единствен файл. Това я прави база данни без сървърен процес, особено подходяща за използване в мобилни устройства, таблети и софтуер, където е невъзможно поддържането на сървърен процес. Поради без сървърния си дизайн, програми използващи SQLite се нуждаят от по-малко конфигурация, в сравнение с достъпа до базата от данни се контролира от самия .db файл

2.4.2 Архитектура - Реализирана е като библиотека към приложенията, а не като самостоятелно работеща програма. За разлика от client-server database management systems, SQLite няма самостоятелен процес, с който апликацията комуникира. Вместо това SQLite библиотеката е импортирана в самата програма. Информацията за базата от данни се запазва на един, cross-platform файл. Записа до този файл става последователно, тоест 2 апликаций не могат да записват по едно и също време. Ченето от този файл може да осъществи от 2 или повече апликаций едновременно.

2.4.3 Кратка история - SQLite е разработена от Дуейн Ричард Хип през 2000, докато е работил за General Dynamics. Целта на SQLite е била да позволи работата на програми, без да се инсталира database management system или database administrator. Хип базира синтаксиса и семантиката на програмата, върху PostgreSQL 6.5

2.5 Photoshop



Photoshop е професионална комерсиална програма за обработка на растерна графика от софтуерната компания Adobe.

2.2.1 Цели и способности – Photoshop позволява интерактивна редакция на сканирани и цифрово заснети графични материали в реално време чрез набор от инструменти. Photoshop е програма за визуална обработка на снимки и картини, за създаване на графики, скици, карти и други изображения. Тази програма се използва предимно за обработка и създаване на висококачествени изображения с висока резолюция, обикновено посредством скениране на много дигитални слоеве. Photoshop използва тонални и цветни инструменти за обработка и работи с модели за цвят, които описват цветовете числово (както повечето програми от тази категория). Има различни методи на описване на цветовете числово.

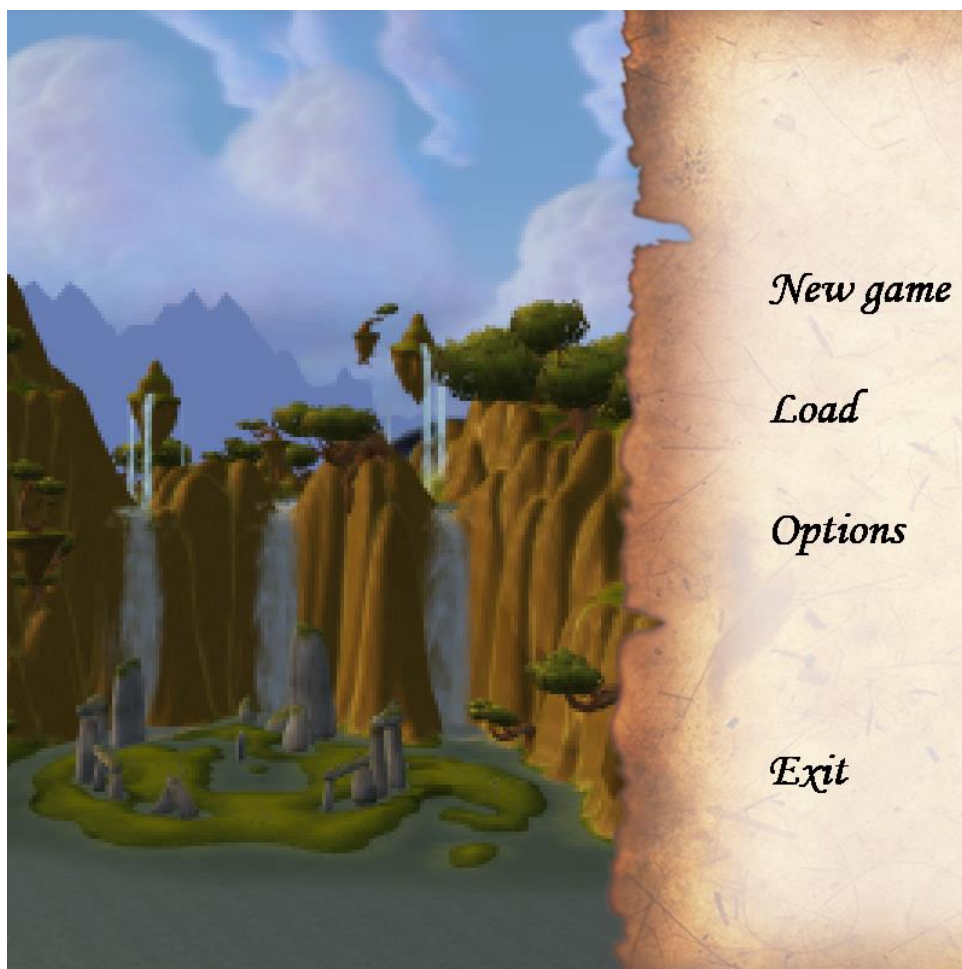
2.2.2 Кратка история - През 1987 Томас Кнол, докторант в Мичиганския университет, започва да пише програма на своя Macintosh Plus компютър за изобразяване на картини в сивата гама на монохромен дисплей. Тази програма, наречена Display, заинтригува брат му Джон Кнол, работник в Industrial Light & Magic, който предлага на Томас да я превърне в цялостна програма за редакция на образи. Томас си взима шестмесечна почивка от проучванията си през 1988 г., за да работи заедно с брат си върху програмата, която е преименувана на ImagePro. В това време Джон пътува до Силициевата долина и представя програмата на инженери от Apple Computer Inc. и на Ръсел Браун, художествен директор в Adobe. И двете демонстрации са успешни и Adobe решават да закупят разрешителното за производство през септември 1988 г. Докато Джон работи по пългини в Калифорния, Томас остава в Ann Arbor да пише програмен код. Версия 1.0 на Photoshop е пусната през 1990 г.

3. Описание на приложението

Целта на играта е да се съберат всички 50 фрагмента от короната на царя, като се избягват противниците по картата.

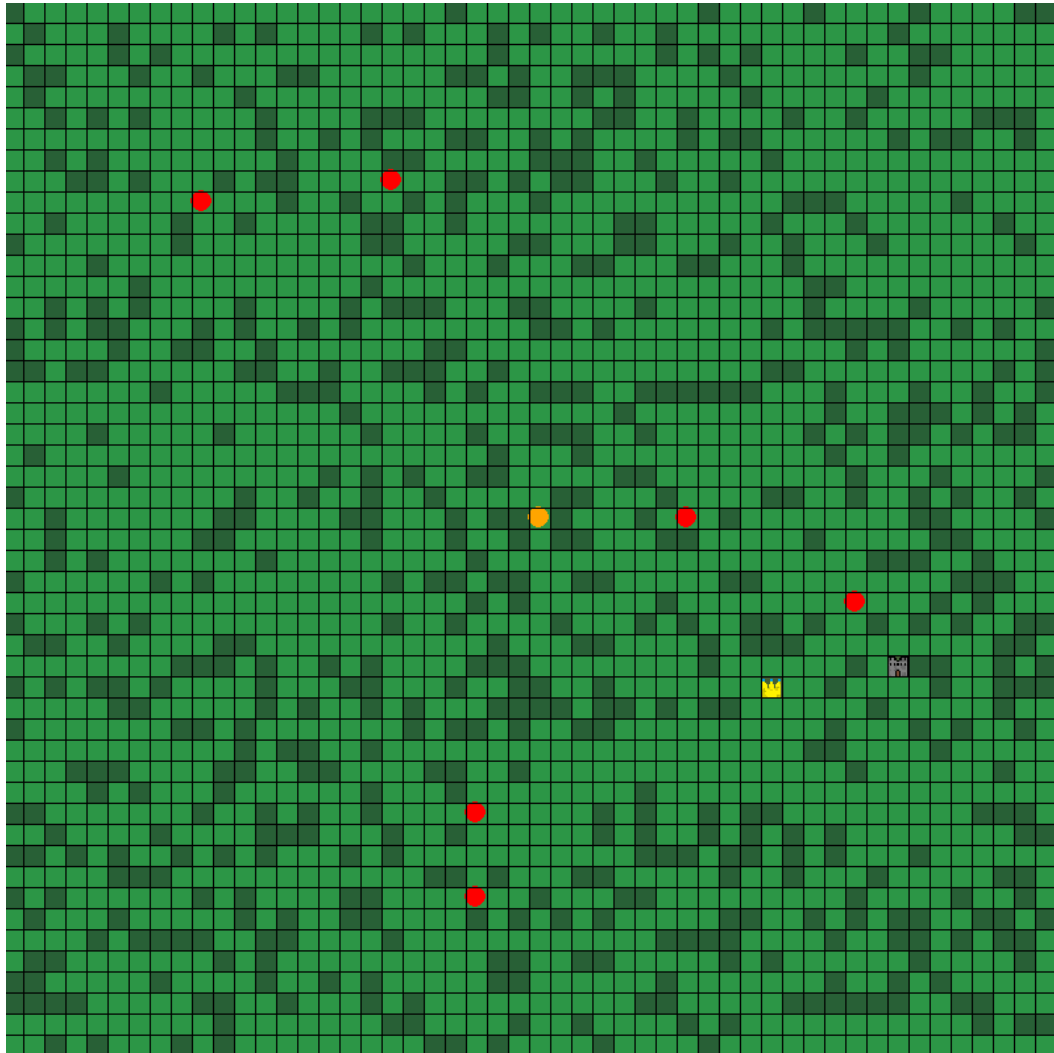
Играта се състои от 3 главни типа екран: Менюта, Карта и Екран за сражение:

- Меню – Менютата позволяват взаимодействието между различните компоненти на играта. Бутоните са обикновено центрирани или имат дясно изместване.



- Карта – Картата за навигация представлява шахматна дъска с размери 50x50. На тази дъска има изобразени: Играч, противници, препятствия, градове и целта до която играча трябва да достигне. Контрола над играча се извършва посредством бутоните:

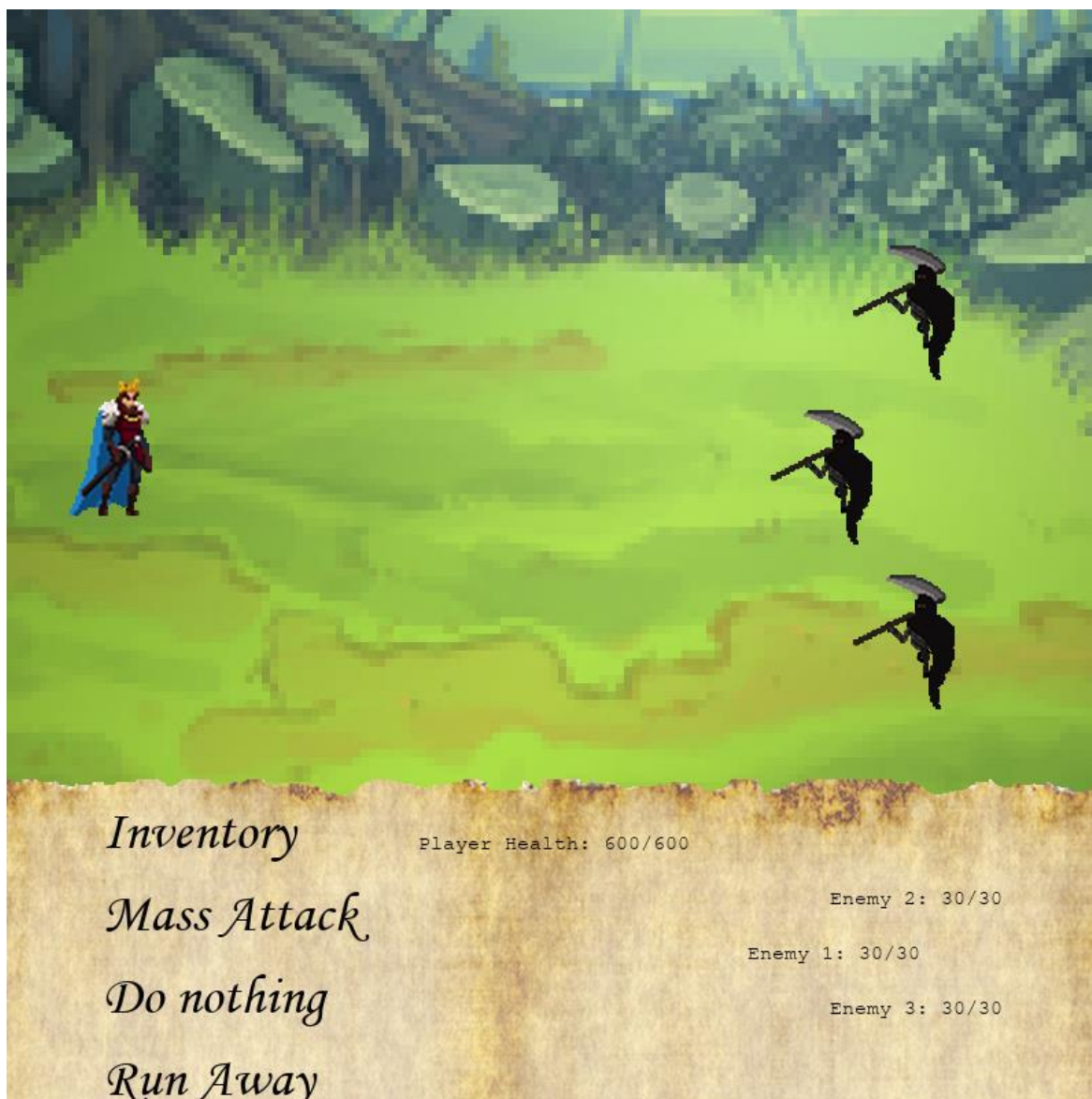
1. “W” - придвижва играча един блок нагоре
2. “A” - придвижва играча един блок наляво
3. “S” - придвижва играча един блок надолу
4. “D” - придвижва играча един блок надясно
5. “[SPACE]” – дава ход на противника
6. “M” – Отваря меню от където играча може да запамети играта си
7. “P” – Отваря екран показващ прогресията на играча.
8. “I” – Отваря меню с инвентара на играча.



- Екран за сражение: Играча влиза в сражение с противници (1 - 5 противника). От този екран играча има следните опции:
 1. **Атака на противник** – Играча може да атакува противник, като кликне върху избрания опонент. Не всяка атака е успешна.

Успеха на атаката се изчислява според екипировката, релевантните умения и нивото на играча и противника. При атака с оръжие за ръкопашен бой, противника има шанс да отвърне на удара преди да е дошъл неговия ход.

2. “Inventory” – Чрез инвентар бутона играча може:
 - Да консумира отвара за живот, която възстановява жизнените му точки по време на сражение.
 - Да смени екипировката си по време на сражение
 - Да изтрие не нужна екипировката от инвентара си
3. “Mass Attack” – Играча изпълнява масова атака, която поражда всички противници на полето, чийто успех се базира върху нивото на играча.
4. “Do nothing” – Играча пропуска хода си
5. “Run Away” – Играча се опитва да избяга от битка. Успеха на това действие се базира върху **DEXTERITY** атрибута на играча.



3.1 Проектиране на системата.

3.1.1 Actors (player and enemy)

Actors са Player и Enemy обектите които извършват действия на екран за навигация и екрана за сражение.

При стартиране на играта се създава Dummy играч, с лимитирана функционалност, който се използва само за навигация на началното меню. Ако играча реши да възстанови запаметена игра от предишна сесия, се извиква член функция - **“loadFromFile(self,file)”**.

Фикция **loadFromFile** променя всичките член променливи на играча, на тези от заредената игра.

Ако играча пожелае да започне нова игра, Dummy играч обекта довършва създаването си.

Player и Enemy обектите имат следните общи функции:

- **Hit_chance(self)** – Определя шанса за успешен удар от страна на атакуващия Actor. Функцията връща **integer** стойност, която определя успеха на атаката.
Тази стойност генерално се определя по следния начин – (произволно число от 1 до 20) + ниво на Actor + (**DEXTERITY** (за ръкопашно оръжие) || **PERCEPTION** (за далекобойно оръжие)) = успех на атаката.
Ако генерираното число е 20. Функцията връща “Critical hit”, което удвоява пораженията за удара и го прави невъзможен за избягване или блокиране.

```
def Hit_chance(self):#genirates a hit chance to be pitted against enemy's dodge and block chance
    hitchanse=randint(1,21)
    if self.equippedW != None and isinstance(self.equippedW, Items.R_weapons):#used only if player
uses a ranged weapon
        return hitchanse+self.XPlevel+ceil(self.stats["PER"]/2)
    else:
        if self.equippedW != None and self.equippedW.damage_type=='P' and hitchanse+5>=20:# used if
player has a pearsing weapon, they have a crit bonus
            hitchanse+=5
        if hitchanse == 20:#the chance to genirate a crit chach is 1/20
            return 1000
        return hitchanse+self.XPlevel+ceil(self.stats["DEX"]/2)#returns the value
```

код на функцията

- **Dodege_chance(self)** - Определя шанса за успешно избягване на удар от страна на защитаващия Actor. Функцията връща **integer** стойност, която определя успеха за избягване на удар.
Тази стойност генерално се определя по следния начин – (произволно число от 0 до 5) + ниво на Actor + **DEXTERITY** атрибута на Actor + умението на Actor с бронята която използва = успех за избягване на удар

```

def Dodge_chance(self):#s kakvo si oblechen,desxterity i level i se sravnqva sus hitchance na enemito
#genirates a dodge chance the be pitted against enemie's hit chance
if self.equippedA == None:
    hitchanse=randint(7,12)+self.stats["DEX"]
elif self.equippedA.armor_type == 'L' or self.equippedA.armor_type == 'C':
    hitchanse=randint(5,10)+self.stats["DEX"]+randint(0,self.skills["Light armour"]//6)
elif self.equippedA.armor_type == 'M':
    hitchanse=randint(3,7)+self.stats["DEX"]+randint(0,self.skills["Medium armour"]//8)
elif self.equippedA.armor_type == 'H':
    hitchanse=randint(0,5)+self.stats["DEX"]+randint(0,self.skills["Heavy armour"]//10)
return (hitchanse+self.XPlevel)

```

код на функцията

Block_chance(self) - Определя шанса за успешно блокиране на удар от страна на защитаващия Actor. Функцията връща **integer** стойност, която определя успеха за блокиране на удар.

Тази стойност генерално се определя по следния начин – (произволно число от 0 до 5) + **Blocking** умението на Actor = успех за избягване на удар

```

def Block_chance(self):
    if isinstance(self.equippedOH, Items.C_shield):#if player has a shield this is used
        return
        randint(self.equippedOH.defence//2,self.equippedOH.defence)+randint(0,self.skills["Blocking"]//8)

    else: #if player has no shiled this is used
        return randint(0,5)+randint(0,self.skills["Blocking"]//10)

```

- **genirate_damage(self)** – След като атаката е минала успешно се генерират нанесените щети върху поразения Actor. Функцията връща **integer** стойност, която определя нанесените щети за оръжието в дясната ръка на Actor. Тази стойност генерално се определя по следния начин – (произволно число от 1 до 3) + силата на оръжието + умението на Actor с използвания тип оръжие.

```

def genirate_damage(self): #genirates damage and ups a skill if lucky
    skillDamage=0# genirates bonus weapon skill damage
    #for 1 hand
    if self.equippedW != None and self.equippedW.hands == 1:
        if isinstance(self.equippedW, Items.M_weapons):
            skillDamage=self.skills["One handed"]//10
        else:
            skillDamage=self.skills["Throwing"]//10
    #for 2 hands
    elif self.equippedW != None and self.equippedW.hands == 2:
        if isinstance(self.equippedW, Items.M_weapons):
            skillDamage=self.skills["Two handed"]//10
        else:
            skillDamage=self.skills["Shooting"]//10
    #for off hand
    if self.equippedOH != None:
        if isinstance(self.equippedOH, Items.M_weapons):
            skillDamage=self.skills["One handed"]//10
        elif isinstance(self.equippedOH, Items.R_weapons):
            skillDamage=self.skills["Throwing"]//10

    if self.equippedW==None:#if you have nothing equiped you return this value
        return randint(1,4)+self.stats["STR"]
    else: return randint(1,3)+randint(1,self.equippedW.damage)+self.stats["STR"]+skillDamage

```

- **genirate_damage_offhand(self)** – След като атаката е минала успешно се генерират нанесените щети върху поражения Actor. Функцията връща **integer** стойност, която определя нанесените щети за оръжието в лявата ръка на Actor. Тази стойност генерално се определя по следния начин – (произволно число от 1 до 3) + силата на оръжието + умението на Actor с използвания тип оръжие.

```

def genirate_damage(self): #genirates damage and ups a skill if lucky

    skillDamage=0# genirates bonus weapon skill damage
    #for 1 hand
    if self.equippedW != None and self.equippedW.hands == 1:
        if isinstance(self.equippedW, Items.M_weapons):
            skillDamage=self.skills["One handed"]//10
        else:
            skillDamage=self.skills["Throwing"]//10

```



```

    #for 2 hands
elif self.equippedW != None and self.equippedW.hands == 2:
    if isinstance(self.equippedW, Items.M_weapons):
        skillDamage=self.skills["Two handed"]//10
    else:
        skillDamage=self.skills["Shooting"]//10

    #for off hand
if self.equippedOH != None:
    if isinstance(self.equippedOH, Items.M_weapons):
        skillDamage=self.skills["One handed"]//10
    elif isinstance(self.equippedOH, Items.R_weapons):
        skillDamage=self.skills["Throwing"]//10

if self.equippedW==None:#if you have nothing equiped you return this value
    return randint(1,4)+self.stats["STR"]
else: return randint(1,3)+randint(1,self.equippedW.damage)+self.stats["STR"]+skillDamage

```

- **take_damage(self, enemy, crit = None)** - След като атаката е минала успешно се генерират нанесените щети върху поразения Actor. Функцията намаля живота на Actor според генерираните щети от **genirate_damage(self) & genirate_damage_offhand(self)**. Според типа броня която защитаващия Actor носи и типа оръжие което атакуващия Actor използва, Щетите се умножават по 1.25 или 0.75. Зависимостите са следните:
 Blunt weapon > Heavy Armor = 1.25
 Slashing weapon < Heavy Armor = 0.75.
 Slashing weapon > Light Armor = 1.25

Ако **crit** параметъра е различен от **None**, нанесените щети се удвояват.

```

def take_damage(self, enemy, crit = None):#when the player takes damage this is used
    damage=enemy.genirate_damage()#enemy genirates damage
    off_damage=enemy.genirate_damage_offhand()#genirates off hand damage if any
    if crit != None:#if the damege is critical the damage is multiplied by 2(determinade by enemy hit
chance)
        damage*=2
    if self.equippedA != None and (enemy.equippedW!=None and enemy.equippedOH!=None):
        #dierrmins if there is a bonus damage based on the armour that player is wearing
        if self.equippedA.armour_type == 'H' and enemy.equippedW.damage_type == 'B':
            damage*=1.25

```

```

elif self.equippedA. armour_type == 'H' and enemy.equippedW.damage_type == 'S':
    damage*=0.75
elif self.equippedA. armour_type == 'L' and enemy.equippedW.damage_type == 'S':
    damage*=1.25
if off_damage != 0:
    if self.equippedA. armour_type == 'H' and enemy.equippedOH.damage_type == 'B':
        off_damage*=1.25
    elif self.equippedA. armour_type == 'H' and enemy.equippedOH.damage_type == 'S':
        off_damage*=0.75
    elif self.equippedA. armour_type == 'L' and enemy.equippedOH.damage_type == 'S':
        off_damage*=1.25
    #armour absorbs some of the damage
if (damage+off_damage)-self.equippedA.defence//2 <=0:
    print(self.equippedA.name + " absorbes all the damage!")
else:
    self.curHealth-=(damage+off_damage)-self.equippedA.defence//2

else: self.curHealth-=(damage+off_damage)*2

#ima shans da uvelichi nqkoi stat
increase=randint(0,10)
if increase == 0 and self.equippedA != None and self.equippedA. armour_type=='L':
    self.skills["Light armour"]+=1

elif increase == 0 and self.equippedA != None and self.equippedA. armour_type=='M':
    self.skills["Medium armour"]+=1

elif increase == 0 and self.equippedA != None and self.equippedA. armour_type=='H':
    self.skills["Heavy armour"]+=1

if self.equippedOH!= None and isinstance(self.equippedOH, Items.C_shield):
    self.skills["Blocking"]+=1

```

3.1.2 Navigation

MapNavigation.py е отговорен за работата на екрана за навигация. В Main функцията за навигация се генерира двуизмерен list от Node обекти които съдържат информация за терена и в тях им играч, противник или са свободни.

Последователността на изпълнение в главния цикъл е следната:

проверка дали играча е стигнал до ниво 50 (победил в играта) -> player.reset()
-> създаване на карта -> влизане в евент цикъл който редува ходовете на играча и противника.

Обработка на входните данни се осъществява в главния цикъл.

Обработваните входи се делят на входи за навигация и такива за достъп до менюта.

```
if event.key == pygame.K_w
    if player.worldmap[player.X_cord][player.Y_cord-1].terrain != "1" and
player.worldmap[player.X_cord][player.Y_cord-1].occupied != "2" and
player.movement_point>0 and player.Y_cord-1>=0:
        player.worldmap[player.X_cord][player.Y_cord].occupied="0"
        player.Y_cord-=1
        player.worldmap[player.X_cord][player.Y_cord].occupied="1"
        player.movement_point-=1
        if player.worldmap[player.X_cord][player.Y_cord].terrain ==
"3":
            screens.cityMenu(WIN, player)
        elif
player.worldmap[player.X_cord][player.Y_cord].terrain=="2":
            player.levelStage+=1
        return
```

Фрагмент от кода отговорен за обработката входи за навигация

```
if event.key == pygame.K_m:
    screens.menuNavMap(WIN, player)
```

Фрагмент от кода отговорен за обработката входи за достъп до менюта.

MapNavigation.py е съставен от следните функции

- **make_grid(player)** - Функцията е отговорна за генерирането на карта. For цикъл популира двумерен лист със свободни пространства и препятствия. След което се задават стартови позиции за играча и противниците. В метода за генерация има вградени проверки, които не позволяват играч или противник да се генерира върху препятствие или заето пространство. В края на работата си, функцията връща двумерен лист който бива присвоен от player обекта.
- **h(p1, p2)** – Херистика функцията се използва при изчисляване на оптималния маршрут до дестинацията

```
def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)
```

- **algorithm(draw, grid, start, end, enemy, player, screen)** – изчислява оптималния маршрут от стартовата позиция до играча. Когато алгоритъмът изчисли този маршрут, го праща на **reconstruct_path**.

```
def algorithm(draw, grid, start, end, enemy, player, screen):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(), end.get_pos())
    open_set_hash = {start}
    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
        current = open_set.get()[2]
        open_set_hash.remove(current)
        if current == end:
            reconstruct_path(came_from, end, draw, grid, enemy, player, screen)
            end.make_end()
            return True
        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1
            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(),
end.get_pos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()
        if current != start:
            current.make_closed()
    return False
```

- **reconstruct_path(came_from, current, draw, grid, enemy, player, screen)** – След като най-краткия маршрут е изчислен от algorithm функцията. Той бива предаден на reconstruct_path, който от своя страна премества противника по вече изчисления оптимален маршрут и ако критериите за сражение са изпълнени, започва битка.

```
def reconstruct_path(came_from, current, draw, grid, enemy, player, screen):
    lastpos=[]
    while current in came_from:
        current = came_from[current]
        current.make_path()
        lastpos.append(current) #putq do igracha
    for i in grid:
        for j in i:
            if j.occupied=="2":
                try:
                    if grid[lastpos[-2].row][lastpos[-2].col].occupied != "2":
                        grid[lastpos[-2].row][lastpos[-2].col].occupied="2"
                        enemy.startnode=grid[lastpos[-2].row][lastpos[-2].col]
                        grid[lastpos[-1].row][lastpos[-1].col].occupied="0" #mahane na starata poziciq

                except IndexError:
                    if combat.stratcombat(screen, player, enemy.enemyObjsList) == False:
                        enemy.startnode=None
                        grid[lastpos[0].row][lastpos[0].col].occupied="0"
                    pass
```

- **draw(win, player)** – рисува на екран всеки Node от картата на играча. Функцията итерира през двумерния лист и извиква член функция на Node обекта - **draw(win)**

```
def draw(win, player):
    win.fill(Nodeclass.WHITE)
    for row in player.worldmap:
        for node in row:
            node.draw(win)

draw_grid(win, player)
pygame.display.update()
```

- **draw_grid(win, player)** – след като всеки Node е визуализиран, сива решетка се рисува на екрана.

```
def draw_grid(win, player):
    gap = player.optionSettings.WIDTH // player.optionSettings.DIMENSIONS
    for i in range(player.optionSettings.DIMENSIONS):
        pygame.draw.line(win, Nodeclass.BLACK, (0, i * gap), (player.optionSettings.WIDTH, i * gap))
    for j in range(player.optionSettings.DIMENSIONS):
        pygame.draw.line(win, Nodeclass.BLACK, (j * gap, 0), (j * gap, player.optionSettings.WIDTH))
```

- **checkForValidroute (draw, grid, start, end, enemy, player, screen)** – Използва се за проверка, дали играча може да достигне до крайната дестинация, успешно.

```
def checkForValidroute(draw, grid, start, end, enemy, player, screen):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(), end.get_pos())

    open_set_hash = {start}

    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
        current = open_set.get()[2]
        open_set_hash.remove(current)
        if current == end:
            end.make_end()
            return True
```

```

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1
            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(),
end.get_pos())

            if neighbor not in open_set_hash:
                count += 1
                open_set.put((f_score[neighbor], count, neighbor))
                open_set_hash.add(neighbor)
                neighbor.make_open()

        if current != start:
            current.make_closed()

    return False

```

3.1.3 Combat

Combat.py е отговорен за работата на Екран за сражение. След като **stratcombat(screen,player, ,enemies)** функцията се извика от **MapNavigation**. Евевнт цикъл следи за да команди на играча, ако командата е „атака“, след като приключи, дава ход на враговете. Този цикъл може да бъде прекъснат по 3 начина:

- Победа на играча
 - Загубва на играча
 - Играча успява да избяга от битка
- **stratcombat(screen,player,stage,enemies)** – това е main функцията на **Combat.py**. Функцията приема к ато аргументи:
 1. screen – ектана (Pygame обекта, който се използва за display)
 2. player – player обекта, който се използва за интеракция с противниците
 3. enemies – EnemyParty обект който съдържа всички противници за текущата битка
 - **attack(obj1, obj2)** – функцията започва атака от obj1 към obj2. За целта се извикват член функциите:
 - Hit_chance()
 - Dodge_chance()

- Block_chance()

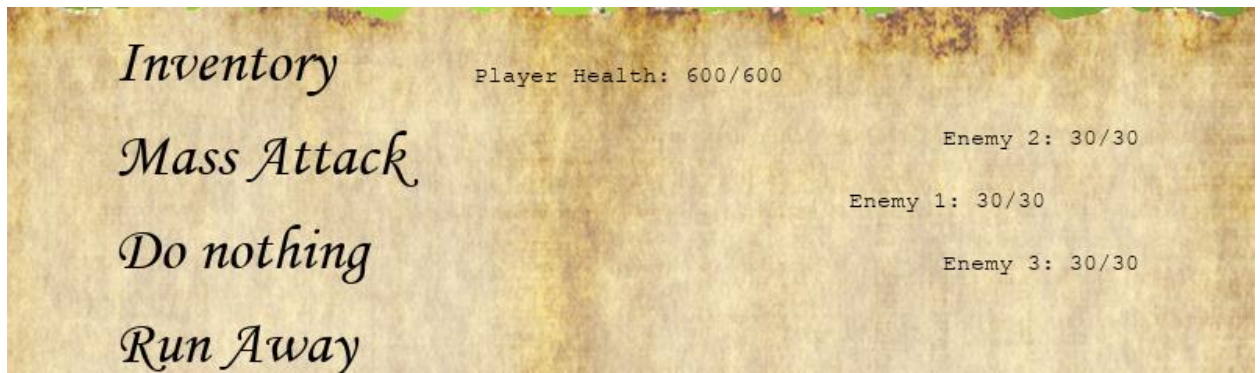
Които съответно генерират числа, решаващи дали атаката е успешна

```
def attack(obj1, obj2):
    hitchance=obj1.Hit_chance()
    dodgechance=obj2.Dodge_chance()
    blockchance=obj2.Block_chance()
    if hitchance>=dodgechance:
        if hitchance>=blockchance:
            obj2.take_damage(obj1)
            print("Hit!")
        else: print("Block!")
    else: print("Miss!")
```

- **specialAttack(obj1, obj2)** - obj1 изпълнява масова атака към obj2, която поразява всички противници на полето, чийто успех се базира върху нивото на играча.

```
def specialAttack(obj1, obj2):
    obj2.curHealth-=randint(0,3)+obj1.XPlevel
```

- **drawBattleMenu(screen,player,enemies)** – Създава overlay върху картата. Този overlay съдържа жизнените точки на играча и противниците. Освен жизнените точки, този слой позволява на играча да достъпи до бутоните :
 - „Inventory” - предоставя достъп до инвентара на играча
 - „Mass Attack – изпълнява атака която поразява всички противници
 - „Do nothing” – пропускане на ход
 - „Run Away” – опит за напускане на сражение



- **deathscreen(player, screen)** – Ако жизнените точки на играча станат 0, Играча влиза deathscreen функцията, която му позволява да започне нова игра или да върне запаметена игра

3.1.4 Items

Items.py се използва за дефиниране и създаване на артикули. Файла съдържа класове за създаване на Item Обекти. Класовете са следните:

- **Item** – Parent class които всички обекти в файла наследяват

```
class item():  
  
    def __init__(self,ID,name,value,weight):  
        self.ID=ID  
        self.name=name  
        self.value=value  
        self.weight=weight  
  
    def __str__(self):  
        return self.name  
  
    def returnName(self):  
        return self.name
```

- **M_weapons** – клас за създаване на обекти от типа ръкопашни оръжия

```
class M_weapons(item):  
    #false - blunt   true - pearce 3 slash  
    def __init__(self,ID,name,damage_type,value,weight,damage,hands,atributes=None):  
        super().__init__(ID,name,value,weight)  
        self.damage_type=damage_type  
        self.damage=damage
```

```

        self.hands=hands
        self.attributes=attributes

    def __str__(self):
        return " Gold {} | DMG:{} | TYPE:{} | {}".format(self.value, self.damage, self.damage_type,
self.name)

```

- R_weapons - клас за създаване на обекти от типа далекобойни оръжия

```

class R_weapons(item):
    ismagic=False #F normal, True staff
    def __init__(self,ID,name,damage_type,value,weight,damage,hands,accuracy):
        super().__init__(ID,name,value,weight)
        self.damage_type=damage_type
        self.damage=damage
        self.hands=hands
        self.accuracy=accuracy

    def __str__(self):
        return " Gold {} | DMG:{} | TYPE:{} | {}".format(self.value, self.damage, self.damage_type,
self.name)

```

- C_armour – клас за създаване на обекти от типа брони

```

class C_armour(item):
    #ligh - L medium - M Heavy - H
    def __init__(self,ID,name,value,weight,armour_type,defence,attributes=None):
        super().__init__(ID,name,value,weight)
        self.defence=defence
        self.armour_type=armour_type
        self.attributes=attributes

    def __str__(self):
        return " Gold {} | DEF:{} | TYPE:{} | {}".format(self.value, self.defence, self.armour_type, self.name)

```

- C_shield – клас за създаване на обекти от типа щитове

```

class C_shield(item):
    hands=1

```

```

def __init__(self,ID,name,value,weight,defence,size):
    super().__init__(ID,name,value,weight)
    self.defence=defence
    self.size=size

def __str__(self):
    return " Gold {} | DEF:{} | TYPE:Shield | {}".format(self.value, self.defence, self.name)

```

- **C_consumable** – клас за създаване на обекти от типа консумативи

```

class C_consumable(item):
    ComType='F' #F - food D - Drink S- stamina H Health
    def __init__(self,ID,name,value,weight,ComType,modifier):
        super().__init__(ID,name,value,weight)
        self.ComType=ComType
        self.modifier=modifier#by how much a stat is going to be increased

    def __str__(self):
        return " Gold {} | DEF: +{} | TYPE:{} | {}".format(self.value, self.modifier, self.ComType, self.name)

    def use(self,user):
        if self.ComType=='F':
            user.curHealth+=self.modifier
        elif self.ComType=='D':
            user.curEnergy+=self.modifier
        elif self.ComType=='H':
            user.curHealth+=self.modifier
            print("healthmof", self.modifier )
        elif self.ComType=='E':
            user.curEnergy+=self.modifier

```

3.1.5 Inventory

Inventory.py е отговорен за интеграцията на играча със собствения му инвентар. **Inventory_manage(screen, player)** приема като аргументи екрана за дисплей и играча. Функцията може да бъде достъпена от екрана за навигация, екран за сражение и City менюто.

Дисплей на инвентара

При стартиране на главната функция и промяна в инвентара на играча, стойностите от инвентара на играча биват мапнати върху 10 бутона. Ако играча има повече от 10 артикула в инвентара си, може да използва scroll “up” & “down” бутоните.

```
list10=[button1,button2,button3,button4,button5,button6,button7,button8,button9,button10]
index=0
for i in player.Inventory:
    if index<=9:
        list10[index].text_setter(str(index)+" "+i.__str__())
        index+=1
lowerLimit=0
upperLimit=10
select=""
```

фрагмент от кода за мапване върху бутони

Смяна на екипировката

При стартиране на главната функция и промяна в екипировката, текущата екипировка на играча е мапната на 3 бутона

- Body
- Right hand
- Left hand

При кликване върху един от съответните бутони, стойността на бутона се връща към дефалутната си стойност и екипировката се връща обратно в инвентара на играча.

```
buttonEquippedA = Button.button(250, 300 ,100 ,70, player.equippedA.__str__(), "red",screen , None)
buttonEquippedRH = Button.button(100, 350 ,100 ,70, player.equippedW.__str__(),"red",screen , None)
buttonEquippedLH = Button.button(400, 350 ,100 ,70, player.equippedOH.__str__(),"red",screen , None)
```

фрагмент от кода

Начина по който се избира типа на интеракцията с артулите се извършва чрез 3 бутона

- **Use** – След като играча избере артикул от менюто в дясно и натисне бутона “Use”. Избрания артикул ще бъде избран като активен. За осъществяването се използва функция **Inventory_manage_selectObj (player,buttonlist,buttonNumber):**

```
def Inventory_manage_selectObj(player,buttonlist,buttonNumber):  
    for i in player.Inventory:  
        if i.__str__() == buttonlist[buttonNumber].text_getter()[2:]:  
            print(i)  
            return i  
    return ""
```

кода на функцията

името на бутона се предава като аргумент за функцията и се извършва проверка за наличност в инвентара на играча.

Ако артикула е открит в инвентара на играча, той бива избран за активан. Код за това генерално изглежда по следния начин.

```
if buttonUse.draw_button():  
    print(select, type(select))  
    if isinstance(select,Items.C_armour):  
        if player.equippedA!=None:  
            player.Inventory.append(player.equippedA)  
            player.equippedA=select  
            del player.Inventory[player.Inventory.index(select)]  
        else:  
            player.equippedA=select  
            del player.Inventory[player.Inventory.index(select)]
```

кода на функцията – за активиране на артикул

- **Delete** - След като играча избере артикул от менюто в дясно и натисне бутона “Delete”. Избрания артикул ще бъде Изтрит от инвентара на

играча. Преди да се случи това, играча трябва да потвърди действието си с Pop up проверка.



pop up потвърждение

```
if buttonDelete.draw_button():  
    if select!="":  
        if Button.Popup("save",screen) == "Y": # ArithmeticErrorscreen  
            del player.Inventory[player.Inventory.index(select)]  
            screen.fill((0,0,0))
```

код за изтриване на атрикул

функцията приема като параметри играч обекта, листа с бутоните на екрана и бутона на селектирания артикул.

- **Exit** – извежда играча обратно в екрана от където е дошъл



3.1.6 Buy & Sell

Market.py — отговаря за купуването и продаването на артикули от страна на играча **marketa(screen, player)** приема като аргументи екрана за дисплей и играча. Функцията може да бъде достъпена са само от City Менюто когато играча избере Market.



Дисплей на инвентара, на продавача

При стартиране на главната функция стойностите от артикули в играта биват мапнати върху 10 бутона. Артикулите в в играта са повече от 10, за това може да използва scroll “up” & “down” бутоните.

```
list10=[button1,button2,button3,button4,button5,button6,button7,button8,button9,button10]
```

```
available=[]  
available.extend(Items.Mweapons)  
available.extend(Items.Armour)  
available.extend(Items.patitions)  
available.extend(Items.Shield)
```

```
lowerLimit=0  
upperLimit=10  
select=""
```



```

index=0
flag="Buy"
for i in available[lowerLimit:upperLimit]:
    if index<=9:
        list10[index].text_setter(str(index)+" "+i.__str__())
        index+=1

```

фрагмент от кода за мапване върху бутони

Продаване:

По default, опцията за продаване е избрана. Когато е избрана, на дисплей са артикулите които могат да бъдат закупени. Когато играча избере артикул който иска да закупи, трябва да избере бутона confirm.

Избирането на артикул се осъществява със следната функция Trade_selectObj(buttonlist,buttonNumber)

```

def Trade_selectObj(buttonlist,buttonNumber):
    available=[]
    available.extend(Items.Mweapons)
    available.extend(Items.Armour)
    available.extend(Items.potions)
    available.extend(Items.Shield)
    for i in available:
        if i.__str__() == buttonlist[buttonNumber].text_getter()[2:]:
            print(i)
            return i
    return ""

```

фрагмент от кода за купуване

Ако разполага с нужните средства, транзакцията ще мине и артикула ще бъде добавен в инвентара на играча. Цената на артикула зависи от **Diplomacy** уменията на играча.

```

elif flag == "Buy":
    if player.money>=available[available.index(select)].value-player.stats["DIP"]:
        player.money-=available[available.index(select)].value-player.stats["DIP"]
        player.Inventory.append(available[available.index(select)])

```

```
else:  
    print("insufficient funds")
```

кода отговорен за таксуването на играча

продаване на артикули

след избиране на бутона “Sell”. Лентата с артикули се мапва с тези Артикули от инвентара на играча.

```
if flag == "Sell":  
    player.money+=player.Inventory[player.Inventory.index(select)].value//2  
    del player.Inventory[player.Inventory.index(select)]  
    index=0  
    printBG(screen, player)  
    for i in player.Inventory[lowerLimit:upperLimit]:  
        if index<9:  
            list10[index].text_setter(str(index)+" "+i.__str__())  
            index+=1
```

кода отговорен за смяната на режима на работа

Когато играча избере артикул който иска да продаде, трябва да избере бутона confirm. Когато бутона е избран, играча трябва да потвърди действието си с Pop up проверка.

Селектирания артикул се селектира чрез, преизползваната функция за селекция от **inventory.py**

Inventory_manage_selectObj(player,buttonlist,buttonNumber)

```
def Inventory_manage_selectObj(player,buttonlist,buttonNumber):  
    for i in player.Inventory:  
        if i.__str__() == buttonlist[buttonNumber].text_getter()[2:]:  
            print(i)  
            return i  
    return ""
```

кода отговорен за селекция на артикула

функцията приема като параметри играч обекта, листа с бутоните на екрана и бутона на селектирания артикул.



3.1.7 Player Progression

Progression.py е отговорна за на играча. Функцията, `progress(screen, player)`, приема като аргументи екрана за дисплей и играча. Функцията може да бъде достъпена а City Менюто и когато играча натисне бутон „р“ екран за навигация.

Атрибути

След влизане в Progression менюто, играча получава информация за четирите си главни атрибута:

- **Strength** – Strength атрибута е отговорен за силата на атаките, на играча. Колкото по-висок е атрибута, толкова по силни са атаките на играча
- **Endurance** - Endurance атрибута е отговорен за жизнените точки на играча и устойчивостта му на атаки. Колкото по висок е този атрибут, толкова по устойчив е играча на атаки.
- **Diplomacy** - Diplomacy атрибута е отговорен за успеха на играча при купуване и продаване на артикули. Колкото по висок е този атрибут, толкова по-добри цени за играча има на пазара.
- **Dexterity** - Dexterity атрибута е отговорен за успеха на играча при избягване на удари от противници и точност на атаките му. Колкото по висок е този атрибут, толкова по голям е шанса на играча за успешна атака и избягване на атака от опонента.

Над четирите атрибута са показани точките които играча може да използва да повиши тези атрибути. Това става със следния фрагмент от кода.

```
elif StrenghtP.draw_button():
    if player.ability_points > 0:
        player.stats["STR"]+=1
        player.ability_points-=1
        screen.fill((0,0,0))

elif StrenghtM.draw_button():
    if player.stats["STR"] > 0:
        player.stats["STR"]-=1
        player.ability_points+=1
        screen.fill((0,0,0))
```

кода отговорен за поквичаване и намаляне на атрибутите

Умения

Играча има умения, които диктуват успеха му при използване на определен тип броня или оръжия. Колкото е по-голямо умението на играча с типа оръжие, толкова по-голям е шанса на играча да изпълни успешна атака или да се защити от атака.

- One handed – оръжия за една ръга
- Two handed – оръжия за две ръгце
- Throwing – оръжия за хвърляне
- Shooting – оръжия за стрелба
- Blocking – щитове за блокиране
- Heavy armour – тежки брони
- Medium armour – средни брони
- Light armour – леки брони

```
screen.blit(player.optionSettings.myfont.render("One handed -  
"+str(player.skills["One handed"]), 1, (255,255,255)), (550, 150))  
    screen.blit(player.optionSettings.myfont.render("Two handed -  
"+str(player.skills["Two handed"]), 1, (255,255,255)), (550, 190))  
    screen.blit(player.optionSettings.myfont.render("Throwing -  
"+str(player.skills["Throwing"]), 1, (255,255,255)), (550, 230))  
    screen.blit(player.optionSettings.myfont.render("Shooting -  
"+str(player.skills["Shooting"]), 1, (255,255,255)), (550, 270))  
    screen.blit(player.optionSettings.myfont.render("Blocking -  
"+str(player.skills["Blocking"]), 1, (255,255,255)), (550, 310))  
    screen.blit(player.optionSettings.myfont.render("Heavy armour -  
"+str(player.skills["Heavy armour"]), 1, (255,255,255)), (550, 350))  
    screen.blit(player.optionSettings.myfont.render("Medium armour -  
"+str(player.skills["Medium armour"]), 1, (255,255,255)), (550, 390))  
    screen.blit(player.optionSettings.myfont.render("Light armour -  
"+str(player.skills["Light armour"]), 1, (255,255,255)), (550, 430))
```

Фрагмент от кода, отговорен за визуализацията на уменията

Над атрибутите и уменията са разположени нивото на играча и точките опит нужни за вдигане на нивото. Точките опит се придобиват когато играча победи в сражение.

Кода за тяхната визуализация е следния:

```

screen.blit(player.optionSettings.myfont.render("Player XP:
"+str(player.XP), 1, (255,255,255)), (350, 100))

screen.blit(player.optionSettings.myfont.render(" /
"+str(player.XPcaps[player.XPlevel-1]), 1, (255,255,255)), (420, 100))

screen.blit(player.optionSettings.myfont.render("Player level:
"+str(player.XPlevel), 1, (255,255,255)), (500, 100))

```

Фрагмент от кода, отговорен за визуализацията на опита, на играча



3.1.8 Game menus

Повечето функции за изобразяване на менюта се съдържат в screens.py с изключение на главния екран и екрана за сражение, които се намират в Menu.py и Combat.py.

Структура на менютата

Повечето менюта съдържат следната структура:

```
def mainmenu(screen, player):

    if player.optionSettings.art == True:
        screen.blit(player.optionSettings.BGs[4], (0,0))
    newGameButton = Button.button(600, 200 ,150 ,40,"New game", "red", screen,
pg.image.load('buttons/newg1.png').convert_alpha())
    LoadButton = Button.button(600, 300 ,150 ,40,"Load", "red", screen,
pg.image.load('buttons/loadg.png').convert_alpha())
    OptionsButton = Button.button(600, 400 ,150 ,40,"Options", "red", screen,
pg.image.load('buttons/Options.png').convert_alpha())
    CreditsButton = Button.button(600, 500 ,150 ,40,"Credits", "red", screen,
pg.image.load('buttons/credits.png').convert_alpha())
    exitButton = Button.button(600, 600 ,150 ,40,"Exit", "red", screen,
pg.image.load('buttons/Exit.png').convert_alpha())
    run = True
    while run:
        for event in pg.event.get():
            if event.type == pg.QUIT:
                run = False
            elif newGameButton.draw_button():
                screens.charcreation(screen, player)
                # MapNavigation.main(screen, player)
                if player.optionSettings.art == True:
                    screen.blit(player.optionSettings.BGs[4], (0,0))
            elif LoadButton.draw_button():

                while CRUD.CRUD(screen, player, 1):
                    MapNavigation.main(screen, player)
                if player.optionSettings.art == True:
                    screen.blit(player.optionSettings.BGs[4], (0,0))

            elif OptionsButton.draw_button():
                screens.manage_settings(screen, player.optionSettings)
                screen.fill((0,0,0))
                if player.optionSettings.art == True:
                    screen.blit(player.optionSettings.BGs[4], (0,0))
```

```
elif CreditsButton.draw_button():
    screens.creditscreen(screen)
    screen.fill((0,0,0))
    if player.optionSettings.art == True:
        screen.blit(player.optionSettings.BGs[4], (0,0))

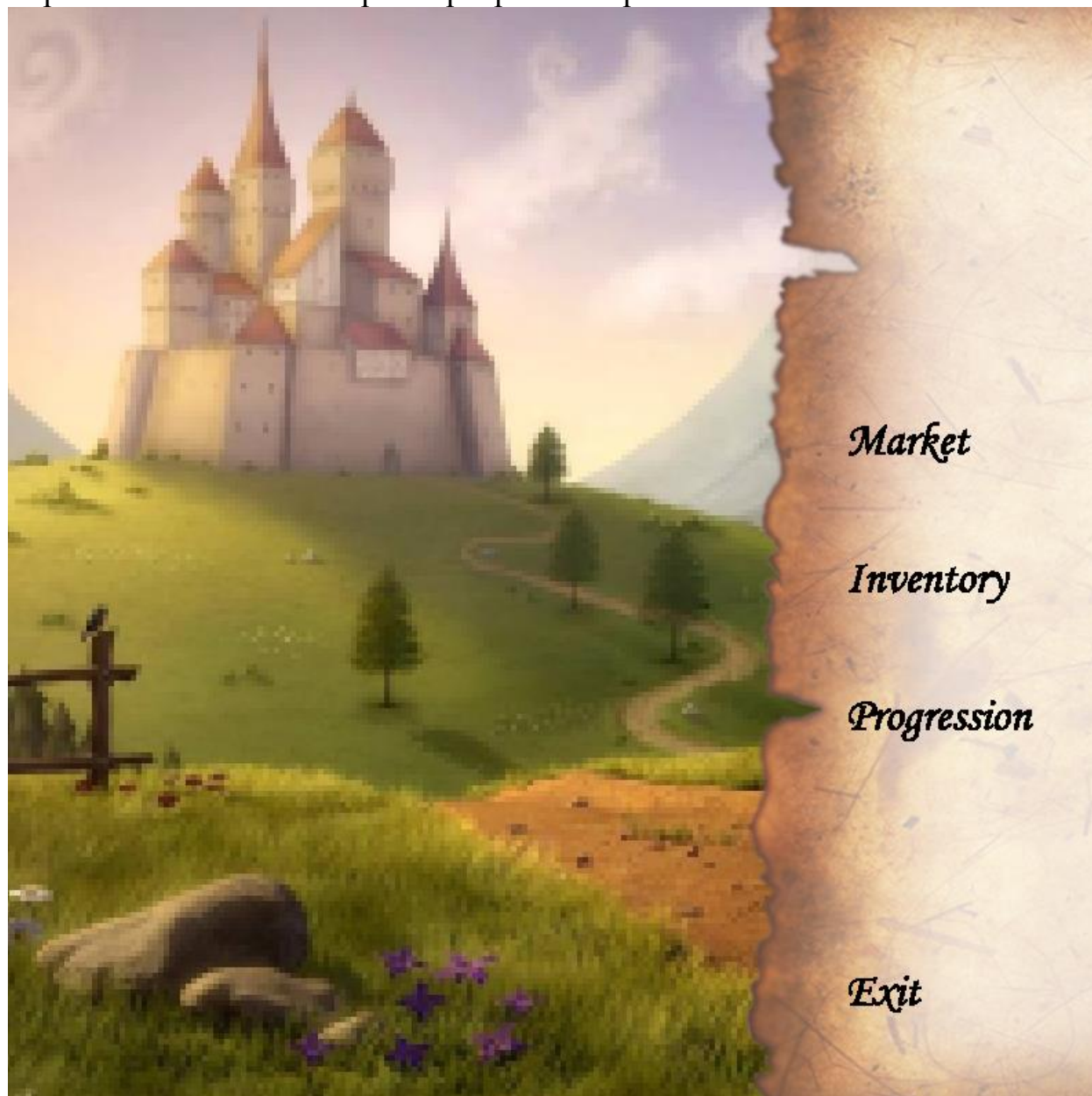
elif exitButton.draw_button():
    run = False
pg.display.update()
```

Бутоните се създават посредством Button класа. Следва цикъл който изобразява бутоните на екрана и обработва изникналите “Events”.



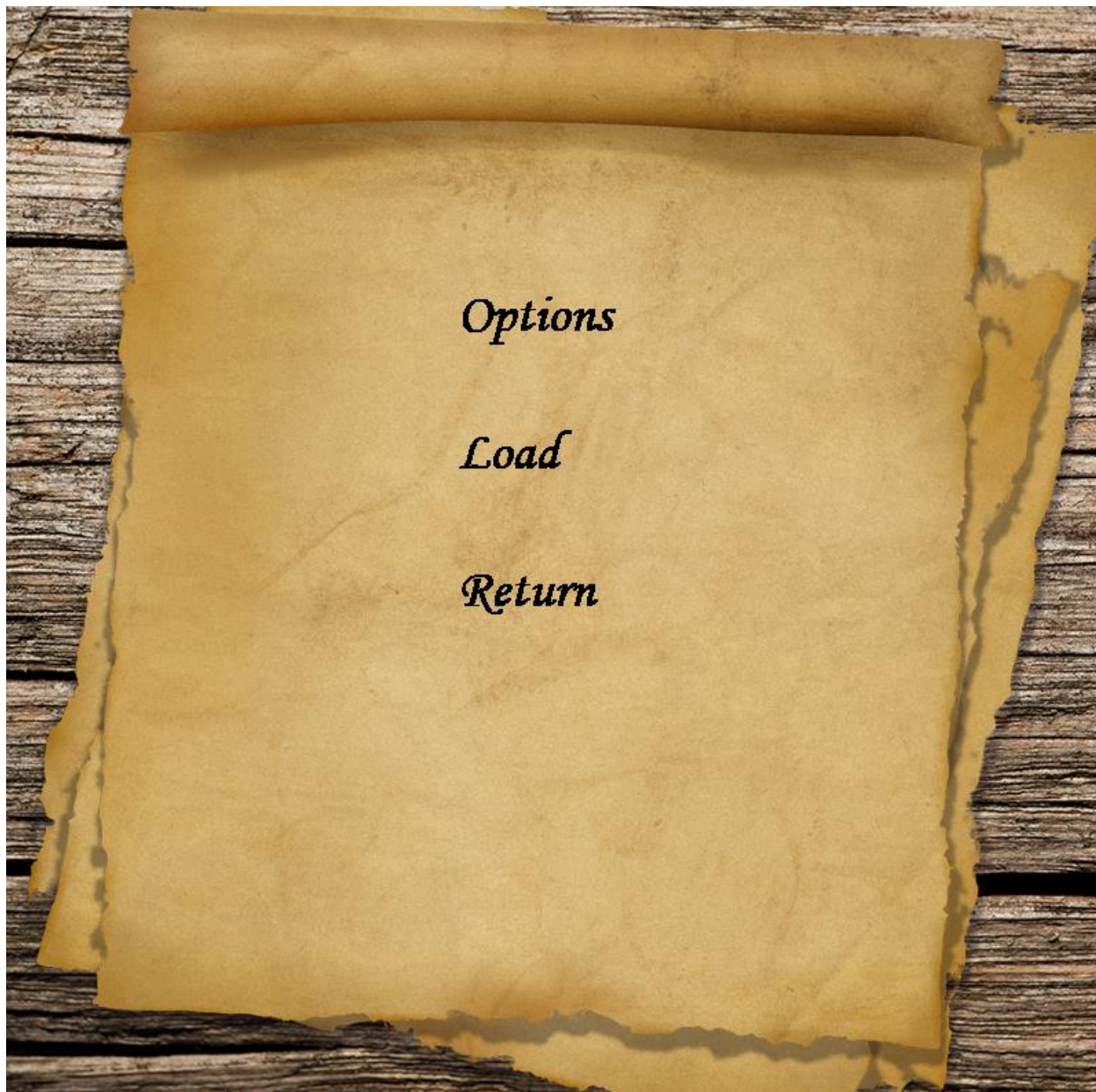
3.1.8.1 Старт меню

Чрез функция **mainmenu(screen, player)**, в Menu.py, приема като аргументи, екрана на който ще се визуализира информацията и играч обекта. Екрана се извиква само при стартиране на приложението.



3.1.8.2 Градско меню

Чрез функция **cityMenu(screen, player)** в screens.py, приема като аргументи, екрана на който ще се визуализира информацията и играч обекта. Екрана се извиква само при от екран за навигация, когато играча влезе в град.



3.1.8.3 Генерично меню за навигацията

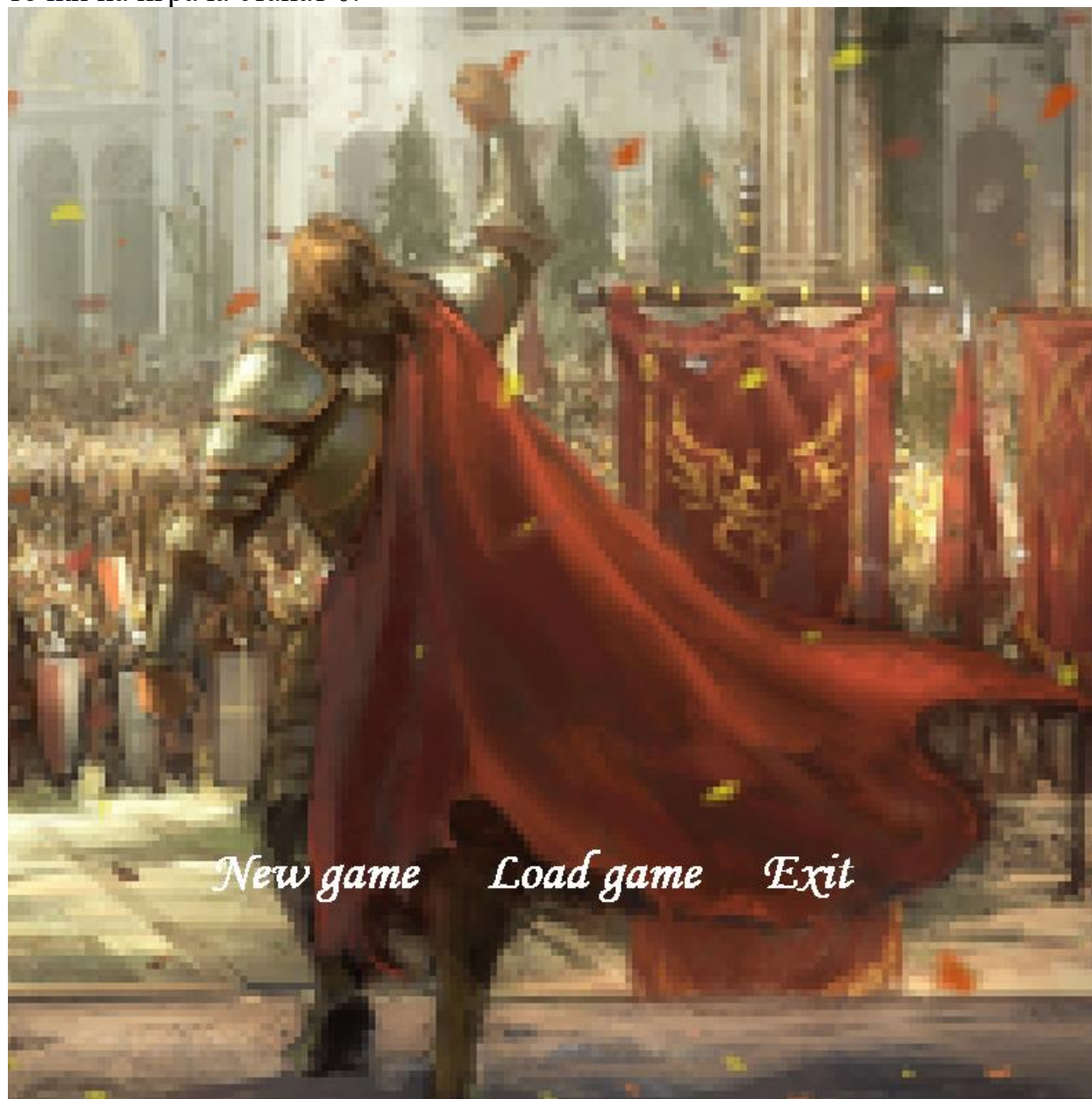
Чрез функция **menuNavMap(screen, player)** в screens.py, приема като аргументи, екрана на който ще се визуализира информацията и играч обекта . Менюто може да се достъпи само от екран за навигация, когато играча натисне бутон „М.“



3.1.8.4 Меню за загубена игра

Чрез функция **deathscreen(screen, player)** в Combat.py, приема като аргументи, екрана на който ще се визуализира информацията и играч обекта.

Менюто може да се достъпи само от екрана за сражение, когато жизнените точки на играча станат 0.



3.1.8.5 Меню за спечелена игра

Чрез функция **winCondition(player, screen)** в `screens.py`, приема като аргументи, екрана на който ще се визуализира информацията и играч обекта. Менюто може да се достъпи само от екрана за навигация, когато играча достигне до петдесето ниво.



3.1.8.6 Меню за конфигурация на опциите

Чрез функция **manage_settings(screen, OptionsObj)** в `screens.py`, приема като аргументи, екрана на който ще се визуализира информацията и обект, който съдържа всички опции. Менюто може да се достъпи от екрана за навигация и старт менюто.



Чрез функция **charcreation(screen, player)** в `screens.py`, приема като аргументи, екрана на който ще се визуализира информацията и обект, който съдържа всички опции. Менюто може да се достъпи само от старт менюто.



Чрез функция **CRUD(screen, player, access)** в **CRUD.py**, приема като аргументи, екрана на който ще се визуализира информацията, обект от типа **player** и нивото за достъп. Нивото за достъп определя кои от стандартните **CRUD** (**create remove update delete**) функционалности ще са достъпни до играча. **CRUD** Менюто може да бъде достъпено от старт менюто или менюто от картата за навигация.

Възможните стойност за достъпа са следните

Ниво на достъп	Create	Update	Remove	Delete
----------------	--------	--------	--------	--------

1	Не	Да	Да	Да
2	Да	Да	Да	Да

3.1.9 Options

При започване стартиране на играта се създава Gameoptions обект, чиито стойности се вземат от файл наречен “ini.txt”, какъвто е стандарта в индустрията. Всеки път когато опциите се обновят, от менюто, файла се презаписва с новите опции.

Ако този файл не съществува, ще бъде създаден с функция **createfile(self)**

```
def createfile(self):
    with open('ini.txt', 'w') as f:
        f.write(str(self.art)+"/"+str(self.volume)+"/"+str(self.sound)+"/"+str(self.difficulty))
```

функция за създаване на файл при нужда

инициализацията на Gameoptions обета става със следната функция

```
def loadfile(self):
    try:
        with open('ini.txt') as f:
            fileContent=f.readlines()
            holdoptions=fileContent[0].split("/")
            print(type(holdoptions[0]))
            if holdoptions[0] == "True":
                self.art = True
            else: self.art = False
            # self.volume = int(holdoptions[1])
            # self.sound = int(holdoptions[2])
            self.difficulty = holdoptions[3]
    except:
        with open('ini.txt', 'w') as f:
            f.write(str(self.art)+"/"+str(self.volume)+"/"+str(self.sound)+"/"+str(self.difficulty))
```


3.2 Описание на GUI.

GUI (Graphical User Interface) е реализиран чрез предоставените функции от Pygame и Button класа, създаден за тази апликация. Първоначалния екран се създава от следния фрагмент от кода:

```
pg.init()
clock = pg.time.Clock()
fps = 60
WIN = pg.display.set_mode((gameSettings.WIDTH, gameSettings.WIDTH))
```

Генерално, структурата на кода формира от безкраен цикъл който следи за евенти(движение на мишката, натискане на клавиш от клавиатурата или мишката). В самия цикъл се визуализират бутоните създадени извън цикъла от клас button. В края на цикъла има функция която освежава екрана.

кадрова честота

Екрана има дефалтна стойност на опресняване 30 fps (frames per second). Тази стойност може да бъде променена с чрез следния параметър:

```
pygame.time.Clock().tick(60)
```

Blit функция

Blit функцията се използва за визуализацията на Background и текст на екрана. Функцията има следната структура – [Pygame екран обект].blit((Път до избраното изображение или текст за визуализация), (X и Y координати))

3.2.1 Navigation grid

Nodeclass.py съдържа Node класът. Този клас се използва за реализацията на навигационна карта. навигационната карта се състои от 2500 node обекта. Всеки от които има главни атрибута определящи генерацията – terrain и occupied.

Terrain е статичен слой, той не може да се променя. Слой служи за визуализация на терен като свободни пространства, препятствия, замъци и финалната дестинация.

Таблицата за състоянията на terrain е следната:

0	1	2	3
Свободно	Препятствие	Замък	Цял

Occupied е динамичен слой който може да се променя всеки ход. Слой служи за визуализация на играча и противниците.

Таблицата за състоянията на occupied е следната:

0	1	2
Празно пространство	Играч	Противник

Конструктор за Node – приема като параметри:

- row – Определя позицията на обекта по оста X
- col - Определя позицията на обекта по оста Y
- width – изпасва се определяне на позицията на обекта при рисуване на картата за навигационна
- total_rows – изпасва се определяне на позицията на обекта при рисуване на картата за навигационна

```
def __init__(self, row, col, width, total_rows):
    self.terrain = "0"
    self.occupied = "0"
    self.neighbors = []
    self.row = row
    self.col = col
    self.color = WHITE # relace with navigation
    self.x = row * width
    self.y = col * width
    self.width = width
    self.total_rows = total_rows
```

- Draw – приема само един параметър, който е екрана за визуализация. Когато функцията бъде извикана, прави проверка какви член променливи я съставят и визуализира съответната кутия на карта.

```
def draw(self, win):
    if self.terrain == "1":
        pygame.draw.rect(win, "#286036", (self.x, self.y, self.width, self.width))
```

```

elif self.terrain == "2":
    pygame.Surface.blit( win,pygame.image.load("sprites/crown.png").convert_alpha(), (self.x, self.y,
self.width, self.width))
elif self.terrain == "0":
    pygame.draw.rect(win, "#2c9646", (self.x, self.y, self.width, self.width))
elif self.terrain == "3":
    pygame.Surface.blit( win,pygame.image.load("sprites/castle.png").convert_alpha(), (self.x, self.y,
self.width, self.width))
#draw actors
if self.occupied == "1":
    pygame.draw.circle(win, ORANGE, (self.x+7, self.y+7), self.width/2)
elif self.occupied == "2":
    pygame.draw.circle(win, RED, (self.x+7, self.y+7), self.width/2)

```

update_neighbors – Всеки node трябва да знае типа терен на своите 4 съседни за да бъде успешен алгоритъма за навигация. Това се осъществява със функцията **update_neighbors**.

```

def update_neighbors(self, grid): #4testing
    self.neighbors = []
    if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].terrain == "1": # DOWN
        self.neighbors.append(grid[self.row + 1][self.col])

    if self.row > 0 and not grid[self.row - 1][self.col].terrain == "1": # UP
        self.neighbors.append(grid[self.row - 1][self.col])

    if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].terrain == "1": # RIGHT
        self.neighbors.append(grid[self.row][self.col + 1])

    if self.col > 0 and not grid[self.row][self.col - 1].terrain == "1": # LEFT
        self.neighbors.append(grid[self.row][self.col - 1])

```

3.2.2 Enemy party

Обекти от типа EnemyParty се използват за населяването на картата навигация с противници. Обекта съдържа от 0-5 в зависимост от нивото на трудност на играта.

```

class EnemyParty :
    pass
    def __init__(self, X, Y, partyDifficulty):

```

```
self.Y_cord=X
self.X_cord=Y
self.enemyObjsList=[]
self.startnode=0
difrng=0
if partyDifficulty=="Easy":
    difrng=randint(1,4)
elif partyDifficulty=="Medium":
    difrng=randint(1,5)
else: difrng=randint(1,6)

for i in range(difrng):
    self.enemyObjsList.append(Enemies.Enemy("enemy1",5))
```

3.2.3 Бутони

За създаването на бутони се използва класът `button` който се намира в `Button.py`. След като бъде създадена инстанция на класа `button`, бутона може да се визуализира на екрана чрез член функцията **`draw_button()`**:

Конструктор

Конструктора приема като параметри:

`X` – определя позицията по `X` координата - (int)

`Y` – определя позицията по `Y` координата - (int)

`Width` – определя ширината на бутона - (int)

`Height` - определя височината на бутона - (int)

`Text` – задава текст на бутона – (str)

`Color` – определя цвета на бутона – (str)

`screen` – Pygame екран обекта – (Pygame obj)

`Image` – приема пътя до изображение на компютъра (str)

Ако има зададено изображение на `image` параметъра параметрите `text`, `color`, `width` и `height` биват игнорирани и на тяхно място се използва зададеното изображение.

Инициализатор на обекта

```
def __init__(self, x, y, width, height, text, color, screen, image):
    self.x = x
    self.y = y
    self.width = width
    self.height = height
    self.text = text
    self.color = color
    self.screen=screen
    self.image=image
```

Визуализация на екран

Зависимост дали е избрано изображение, функцията създава правоъгълник с зададените параметри от конструктора или изрисува избраното изображение в посочените координати.

```
def draw_button(self):

    global clicked
    action = False
    pos = pg.mouse.get_pos()
    button_rect = pg.Rect(self.x, self.y, self.width, self.height)
    if button_rect.collidepoint(pos):
        if pg.mouse.get_pressed()[0] == 1:
            clicked = True
        elif pg.mouse.get_pressed()[0] == 0 and clicked == True:
            clicked = False
            action = True
    else:
        if self.image == None:
            pg.draw.rect(self.screen, pg.Color(self.color), button_rect, border_radius=5)
            font = pg.font.SysFont('comicsans', 30)
            text = font.render(self.text, True, (255, 255, 255))

            self.screen.blit(text, (self.x + 30, self.y + 10))
        else:
            self.screen.blit(self.image, (self.x, self.y))
```

```
return action
```

Setter & Getter

Setter `color_setter(self,color)`, `text_setter(self,text)` , `text_getter(self)` са стандартни функции използвани за промяна и връщане на стойността, на член променлива.

- `text_setter` - променя текста на бутона
- `text_getter` – връща текста на бутона
- `color_setter` – променя цвета на бутона

използват се при `market.py` и `inventory.py`

```
def text_setter(self,text):
    self.text=text

    def text_getter(self):
        return self.text

    def color_setter(self,color):
        self.color=color
```

3.2.4 Pop up

Popup() не е член функция на класа, но се намира в `Button.py`. Функцията приема като аргументи екрана за дисплей и играча. Предназначението ѝ е да служи като двойна проверка, дали играча иска да направи невъзвратимо действие. Използва се от **`market.py`**, **`CRUD.py`** и **`inventory.py`**.

```
def Popup(caller, screen):
    buttonBG = button(300, 300 ,250 ,150, "", "brown",screen, None)
    buttonYes = button(350, 350 ,50 ,50, "Yes", "red",screen, None)
    buttonNo = button(450, 350 ,50 ,50, "No", "red",screen, None)
    buttonBG.draw_button()
    pg.display.update()
    while True:
```

```
for event in pg.event.get():
    pg.display.update()
    if event.type == pg.QUIT:
        pg.quit()
    elif buttonYes.draw_button():
        print("test2")
        return "Y"
    elif buttonNo.draw_button():
        print("test3")
        return "N"
```

3.3 Описание на използваните алгоритми.

Какво е алгоритъм

В математиката и компютърните науки алгоритъмът е крайна последователност от добре дефинирани, компютърно изпълними инструкции, обикновено за решаване на клас специфични задачи или за извършване на изчисления.

Информиран алгоритъм

Информираният алгоритъм за търсене е по -полезен за голямо пространство за търсене. Информираният алгоритъм за търсене използва идеята за евристика, затова се нарича още евристично търсене. Тази функция приема текущото състояние на агента като свой вход и дава оценка на това колко близо агент е от целта. Евристичният метод обаче не винаги може да даде най -доброто решение, но гарантира, че ще намери добро решение в разумни срокове.

Не информиран алгоритъм

Алгоритмите за неинформираното търсене са клас алгоритми за търсене с общо предназначение, които работят по груба сила. Неинформираните алгоритми за търсене нямат допълнителна информация за състоянието или пространството за търсене, освен как да пресичат графа, така че се нарича още търсене на сляпо.

A * алгоритъм за навигация –

A * е информиран алгоритъм за обхождане на графики и търсене на път, който често се използва в много области на компютърните науки поради своята универсалност, оптималност и оптимална ефективност.

Принцип на работа

Търси най-добро първо търсене, което означава, че е формулиран като претеглени графики: започвайки от конкретен начален възел на графика, той има за цел да намери път към зададения възел на целта с най-малкия цена (най -малко изминато разстояние, най -кратко време и т.н.). Той прави това, като поддържа дърво от пътища, произхождащи от началния възел, и разширява тези пътища, докато не бъде изпълнен критерият за прекратяване.

При всяка итерация на основния си цикъл, A^* трябва да определи кои от своите пътища да удължи. Това се прави въз основа на цената на пътя и оценка на разходите, необходими за удължаване на пътя по целия път до целта. По -конкретно, A^* избира пътя, който минимизира

$$f(n) = g(n) + h(n)$$

където n е следващият възел по пътя, $g(n)$ е цената на пътя от началния възел до n , а $h(n)$ е евристична функция, която оценява цената на най -евтиния път от n до целта. A^* завършва, когато пътят, който избере да удължи, е път от началото до целта или ако няма пътища, отговарящи на условията за разширяване. Евристичната функция е специфична за проблема. Ако евристичната функция е допустима, което означава, че тя никога не надценява действителните разходи за достигане до целта, A^* гарантирано ще върне пътя с най-ниска цена от началото до целта.

(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)	(5, 0)	(6, 0)	(7, 0)	(8, 0)	(9, 0)
0	0	0	0	0	0	0	0	0	0
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)	(6, 1)	(7, 1)	(8, 1)	(9, 1)
0	0	0	0	0	0	0	0	0	0
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)	(5, 2)	(6, 2)	(7, 2)	(8, 2)	(9, 2)
0	0	0	0	0	0	0	0	0	0
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)	(5, 3)	(6, 3)	(7, 3)	(8, 3)	(9, 3)
0	0	0	0	1	0	0	0	0	0
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)	(5, 4)	(6, 4)	(7, 4)	(8, 4)	(9, 4)
0	0	0	0	1	0	0	0	0	0
(0, 5)	(1, 5)	(2, 5)	(3, 5)	(4, 5)	(5, 5)	(6, 5)	(7, 5)	(8, 5)	(9, 5)
0	0	0	0	1	0	0	0	0	0
(0, 6)	(1, 6)	(2, 6)	(3, 6)	(4, 6)	(5, 6)	(6, 6)	(7, 6)	(8, 6)	(9, 6)
0	0	0	0	1	0	0	0	0	0
(0, 7)	(1, 7)	(2, 7)	(3, 7)	(4, 7)	(5, 7)	(6, 7)	(7, 7)	(8, 7)	(9, 7)
0	0	0	0	0	0	0	0	0	0
(0, 8)	(1, 8)	(2, 8)	(3, 8)	(4, 8)	(5, 8)	(6, 8)	(7, 8)	(8, 8)	(9, 8)
0	0	0	0	0	0	0	0	0	0
(0, 9)	(1, 9)	(2, 9)	(3, 9)	(4, 9)	(5, 9)	(6, 9)	(7, 9)	(8, 9)	(9, 9)
0	0	0	0	0	0	0	0	0	0

Недостатъци:

Един голям недостатък е неговата сравнително ниска продуктивност и скаларност, тъй като съхранява всички генерирани възли в паметта. По този начин, в практическите системи за маршрутизиране на пътуване, той обикновено се превъзхожда от алгоритми, които могат предварително да обработват графиката, за да постигнат по-добра производителност.

Недостатъци:

A* първоначално е проектиран за намиране на пътища с най-ниска цена, когато цената на един път е сумата от неговите разходи, но е показано, че A* може да се използва за намиране на оптимални пътища за всеки проблем, отговарящ на условията на алгебра за разходи.

3.4 Описание на базата от данни

CRUD (create update remove delete) функционалността е реализирана чрез SQLite 3. Код за манипулация на базата от данни се намира в **Database.py**. А код който позволява интеракцията на играча с базата от данни се намираща се в **CRUD.py**.

Цялата SQLite таблица се запазва във файла **save.db**

Създаване на таблица

Функцията се намира в **Database.py**. Използва се за създаване на таблица със следните полета:

Име на параметъра	Тип на данните
savename	TEXT
race	TEXT
name	TEXT
lvlst	INT
mnt_pts	INT
max_eng	INT
max_hlt	INT
cut_hlt	INT
level	INT
off_hand	TEXT
main_hand	TEXT
armour	TEXT
stats	TEXT
skill	TEXT
abp	INT
money	INT
XP	INT
inv	TEXT
map	TEXT
date	TEXT

Функция за създаване на таблицата

```
def create_table():
    connection = sqlite3.connect('save.db')
    c = connection.cursor()
    c.execute("""CREATE TABLE IF NOT EXISTS saves (savename TEXT, race TEXT, name TEXT, lvlst INT,
mnt_pts INT,max_eng INT, max_hlt INT, cut_hlt INT,level INT, off_hand TEXT, main_hand TEXT, armour
TEXT, stats TEXT, skill TEXT, abp INT, money INT, XP INT, inv TEXT, map TEXT, date TEXT)""")
    connection.commit()
    connection.close()
```

Функция

Визуализиране на всички данни

```
def view():
    conn=sqlite3.connect("save.db")
    cur=conn.cursor()
    cur.execute("SELECT savename FROM saves")
    rows=cur.fetchall()
    conn.close()
    return rows
```

Добавяне на нов ред данни в таблицата

```
def add_row(savename, race, name, spec, mnt_pts, max_eng, max_hlt, cut_hlt, level, off_hand,
main_hand, armour, stats, skill, abp, money, XP, inv, expmap, date):
    conn=sqlite3.connect("save.db")
    cur=conn.cursor()
    cur.execute("INSERT INTO saves VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?), (savename, race, name,
spec, mnt_pts, max_eng, max_hlt, cut_hlt, level, off_hand, main_hand, armour, stats, skill, abp, money,
XP, inv, expmap, date))
    conn.commit()
    conn.close()
```

Функция

Връщане на данни от таблицата

```
def load(a):
    conn=sqlite3.connect("save.db")
    cur=conn.cursor()
    cur.execute("SELECT * FROM saves WHERE savename==?",(a,))
    rows=cur.fetchall()
    conn.close()
    return rows
```

Функция

Изтриване на ред данни от таблицата

```
def delete(a):
```

```
conn=sqlite3.connect("save.db")
cur=conn.cursor()
cur.execute("DELETE FROM saves WHERE savename=?", (a,))
conn.commit()
conn.close()
```

Функция

Компресиране

Стойности като инвентар и умения на играча и картата за навигация са трудни за стандартна обработка от базата данни. За това се изпасват функции за компресия, които превръщат комплексните данни типове в лесни за менажиране стрингове. Функциите за компресия са следните и се намират в **player.py** файла

Компресиране на обекти от типа Stats и Skills

```
def compressDictValues(self,toBeComp):
    compressed=""
    for i in list(toBeComp.values()):
        compressed+="/"+str(i)
    return compressed
```

Функция

Компресиране на обекти от типа Items

```
def compressInventory(self):
    compressed=""
    for i in self.Inventory:
        compressed+="/"+str(i.name)
    return compressed
```

Функция

Компресиране на картата за навигация

```
def compressWorldmap(self):
    compressed=""
```

```
for i in self.worldmap:
    for j in i:
        compressed+= j.terrain + j.occupied + "/"
return compressed
```

Decompression

След връщане от базата данни, информацията е компресирана и неизползваема за програмата. Преди да бъде изпасвана, информацията трябва да бъде декомпресирана. Функциите за декомпресия са следните и се намират в **player.py** файла

Декомпресия на обекти от типа Items

```
NamesInventory=file[0][17].split("/")
NamesInventory=NamesInventory[1:]
self.Inventory=[]
for i in NamesInventory:
    for j in Items.listOfAllItems:
        if i==j.name:
            self.Inventory.append(j)
```

фрагмент от кода

Декомпресия на обекти от типа Stats и Skills

```
statsHolder=file[0][12].split("/")
statsHolder=statsHolder[1:]

skillHolder=file[0][13].split("/")
skillHolder=skillHolder[1:]

for i,j in zip(statsHolder,self.stats):
    self.stats[j]=int(i)

for i,j in zip(skillHolder,self.skills):
    self.skills[j]=int(i)
```

фрагмент от кода

Декомпресия на картата за навигация

```

mapHolder = file[0][18].split("/")
grid = []
self.enemiesOnMap=[]
gap = self.optionSettings.WIDTH // self.optionSettings.DIMENSIONS
count=0
for i in range(self.optionSettings.DIMENSIONS):

    grid.append([])
    for j in range(self.optionSettings.DIMENSIONS):
        node = Nodeclass.Node(i, j, gap, self.optionSettings.DIMENSIONS)

        node.terrain=mapHolder[count][0]
        node.occupied=mapHolder[count][1]
        if node.occupied == "1":
            print(node.row-1)
            self.X_cord=node.row
            self.Y_cord=node.col

        elif node.occupied == "2":
            EP=enemyParty.EnemyParty(randint(1, self.optionSettings.DIMENSIONS-1), randint(0,
self.optionSettings.DIMENSIONS-1),1)
            EP.startnode=node
            self.enemiesOnMap.append(EP)

        count+=1

    grid[i].append(node)
self.worldmap = grid

```

фрагмент от кода

CRUD Функция

CRUD предоставя функционалността на играча да взаимодейства с базата от данни. Позволените действия са:

- Запаметяване на игра
- Зареждане на игра
- Изтриване на игра

Структура на главния цикъл

Променливата, „**crud_choise**“, определя по какъв начин играча ще взаимодейства с базата от данни и променя графичния интерфейс да рефлектира избраната функционалност.

Стойност на примамливата	Извършено действие
“save”	Запаметяване на игра
“load”	Зареждане на игра
“delete”	Изтриване на игра

Кода за определяне на стойността за „crud chose“ е следния

```
while True:
    for event in pg.event.get():

        if event.type == pg.QUIT:
            pg.quit()
        elif access==2 and save.draw_button():
            printBG(player,screen)
            crud_choise="save"

        elif load.draw_button():
            printBG(player,screen)
            crud_choise="load"

        elif delete.draw_button():
            printBG(player,screen)
            crud_choise="delete"

        elif backtomemu.draw_button():
            screen.fill((0,0,0))

        return False
```

фрагмент от кода

Запаметяване на игра

Функцията проверява дали има вече записан файл в следната позиция, ако има извиква Pop up меню, за конфурмация дали искате да презапишете нова игра на мястото на вече съществуващата. Ако позицията е свободна играча може да запише играта директно.

```
if crud_choise=="save":
    if save1.draw_button():
        if DB.load("1") != None and len(DB.load("1")) > 0:
            if Button.Popup("save", screen) == "Y":
```

```

        savesOBJ[0].text_setter("1")
        loadOBJ[0].text_setter("1")
        deleteOBJ[0].text_setter("1")
        DB.delete("1")
        CRUD_sendToDB(player,"1")
        saves[0]="1"
        loads[0]="1"
        dels[0]="1"

    else:
        savesOBJ[0].text_setter("1")
        loadOBJ[0].text_setter("1")
        deleteOBJ[0].text_setter("1")
        DB.delete("1")
        CRUD_sendToDB(player,"1")
        saves[0]="1"
        loads[0]="1"
        dels[0]="1"
    printBG(player,screen)

```

фрагмент от код

Зареждане на игра

Прави проверка за валидност и зарежда играта запаметената игра с избраното ID

```

elif crud_choise=="load":
    if load1.draw_button():
        try:
            player.loadFromFile(DB.load("1"))
        except:
            print("no such file")
            screen.fill((0,0,0))
            return True

```

фрагмент от код

Изтриване на игра

Извиква Pop up меню, за конфурмация дали искате да изтриете запаметената игра.

```

elif crud_choise=="delete":

    if delete1.draw_button():

```



```

if Button.Popup("delete", screen) == "Y":
    DB.delete("1")
    saves[0]="1 save"
    loads[0]="1 load"
    dels[0]="1 delete"

    savesOBJ[0].text_setter("1 save")
    loadOBJ[0].text_setter("1 load")
    deleteOBJ[0].text_setter("1 delete")

    printBG(player,screen)

```

фрагмент от код

Изпращане на данни в таблицата

```

def CRUD_sendToDB(player,savename):

DB.add_row(savename,player.race,player.name,player.levelStage,player.max_movement_points,player.
maxEnergy,player.maxHealth,player.curHealth,player.XPlevel, player.equippedOH.name,
player.equippedW.name,player.equippedA.name,player.compressDictValues(player.stats),
player.compressDictValues(player.skills),player.ability_points,player.money,player.XP,player.compressIn
ventory(),player.compressWorldmap(),datetime.now().strftime("%d.%m.%Y/%R"))

```

фрагмент от код

3.5 Тестване и резултати.

3.5.1 - A* и dijkstra's algorithm

При създаването на апликацията, избора на алгоритъм за навигация беше между „A*“ и „dijkstra's algorithm“. Двата алгоритъма имат своите предимства и недостатъци. След проучване и обективни тестове следните резултати излизат на лице:

Постановка за теста



A* cost **364**, visited: **150**

dijkstra cost **338**, visited: **377**

Cost – оптималността на избрания маршрут

Visited – броя на клетките които са били посетени, преди да бъде избран оптимален маршрут.

От данните става ясно, че в повечето случаи ,алгоритъма на **dijkstra** е по-точен в намирането на най-оптималния маршрут до целта. Но за сметка на това, алгоритми пъти по бавен от A*. Поради факта, че алгоритъма на **dijkstra** е не информиран алгоритъм за търсене, който няма допълнителна информация за състоянието или пространството за търсене.

Алгоритъма A*, макар по-ниската си точност, е пъти по-бърз от алгоритъма на **dijkstra**. Коего прави избора за алгоритъм, следния – За целта на апликацията кое е по-важно, точност или скорост.

При положение, че в най-тежък случай, алгоритъма ще се изпълнява 50 пъти върху граф с 2500 точки, A* се превръща в фаворит за задачата.

3.5.2 Python vs C vs Java

Постановка за теста:

Задачата е Матрично умножение, използвайки и трите езика. Матриците са с размер 2048 x 2048 (т.е. 8 589 934 592 операции за умножение и събиране всяка) Всяка клетка на матрицата е запълнена със случайна стойност между 0 и 1,0. Задачата е изпълнена 5 пъти и средния резултат сумиран.

Кода използван за теста е следния

C - код

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define n 2048

double A[n][n];
double B[n][n];
double C[n][n];
int main() {
    //populate the matrices with random values between 0.0 and 1.0
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = (double) rand() / (double) RAND_MAX;
            B[i][j] = (double) rand() / (double) RAND_MAX;
            C[i][j] = 0;
        }
    }
    struct timespec start, end;
    double time_spent;
    //matrix multiplication
    clock_gettime(CLOCK_REALTIME, &start);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    clock_gettime(CLOCK_REALTIME, &end);
    time_spent = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1000000000.0;
    printf("Elapsed time in seconds: %f \n", time_spent);
    return 0;
}
```

Java - код

```
import java.util.Random;
```

```

public class MatrixMultiplication {
    static int n = 2048;
    static double[][] A = new double[n][n];
    static double[][] B = new double[n][n];
    static double[][] C = new double[n][n];

    public static void main(String[] args) {
        //populate the matrices with random values between 0.0 and 1.0
        Random r = new Random();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = r.nextDouble();
                B[i][j] = r.nextDouble();
                C[i][j] = 0;
            }
        }
        long start = System.nanoTime();
        //matrix multiplication
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        long stop = System.nanoTime();
        double timeDiff = (stop - start) * 1e-9;
        System.out.println("Elapsed time in seconds: " + timeDiff);
    }
}

```

Python - код

```

import random
import time

n = 2048
#populate the matrices with random values between 0.0 and 1.0
A = [[random.random() for row in range(n)] for col in range(n)]
B = [[random.random() for row in range(n)] for col in range(n)]
C = [[0 for row in range(n)] for col in range(n)]

start = time.time()
#matrix multiplication
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]

```

```
end = time.time()
print("Elapsed time in seconds %0.6f" % (end-start))
```

Резултат от теста

Език	Време за изпълнение (секунди)
Python	2821,10
Java	84,61
C	49,96

От резултата става ясно, че Java кодът е 1,69 пъти по-бавен от C, докато Python кодът е 56 пъти по-бавен.

Заклучение

Python е сравнително бавен, защото C и Java се компилират, а Python от друга страна се интерпретира. Компиляторът трансформира кода в машинен код, всички наведнъж. Компиляторът, от друга страна, трябва да чете, интерпретира и изпълнява всеки ред код и да актуализира състоянието на машината.

Когато програма се компилира в машинен код, процесорът може да я изпълни директно. Докато при интерпретаторът, самият интерпретатор изпълнява програмата. Това прави Python много гъвкав. Python жертва от производителността си, за да осигури по-голяма гъвкавост, четливост и компактност на код, което го прави по-подходящ за разработката на такъв тип апликация.

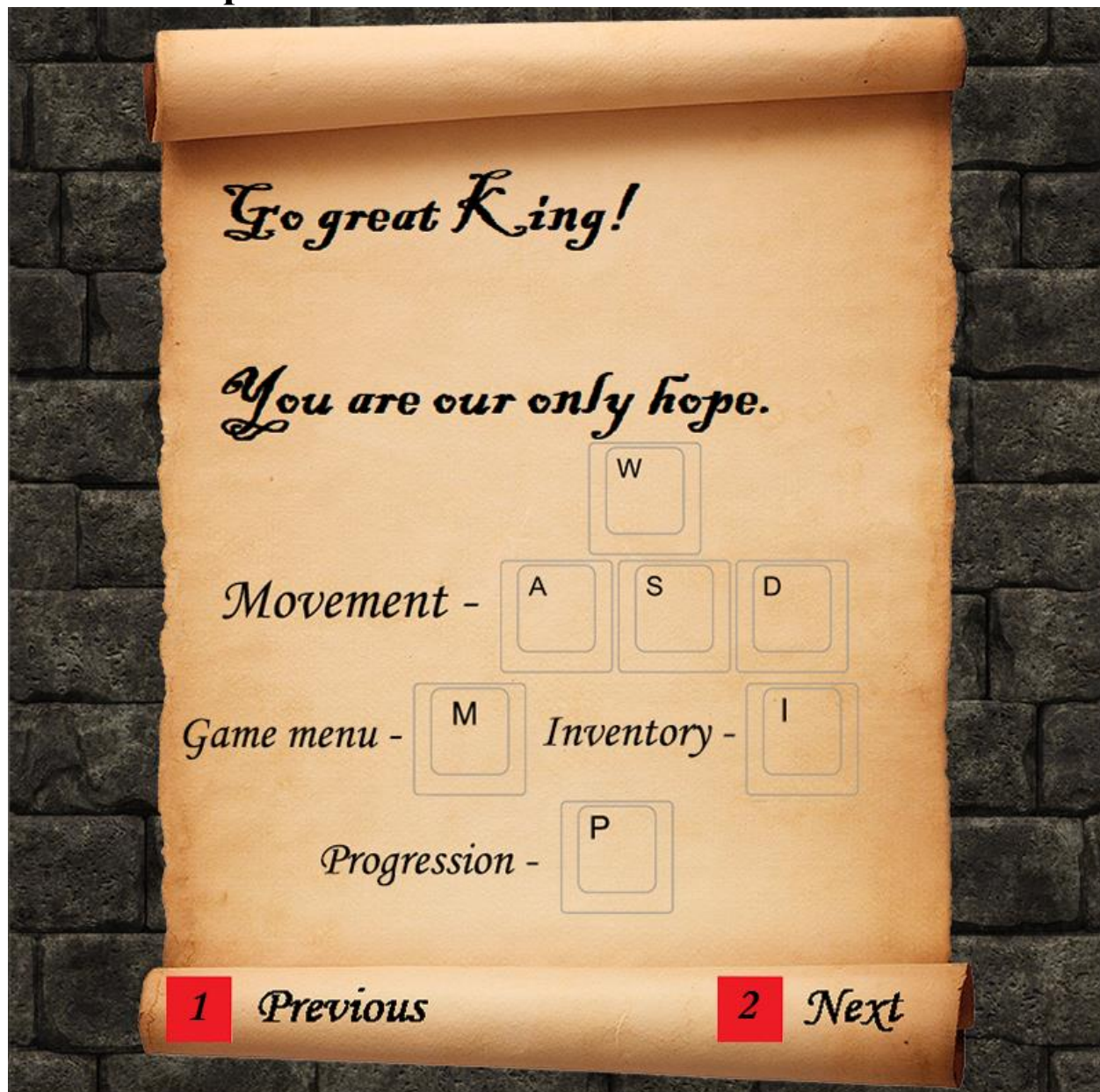
4. РЪКОВОДСТВО ЗА ИЗПОЛЗВАНЕ

4.1 Начален екран



1. **New game** – започване на нова игра
2. **Load** – зареждане или изтриване на запаметена игра
3. **Options** - Конфигурация на нивото на трудност и графиките на приложението
4. **Exit** – прекратява играта и затваря прозореца

4.2 Нова игра



1. **Previous** - Връща потребителя обратно в началния екран
2. **Next** - Продължава със създаването на новата игра

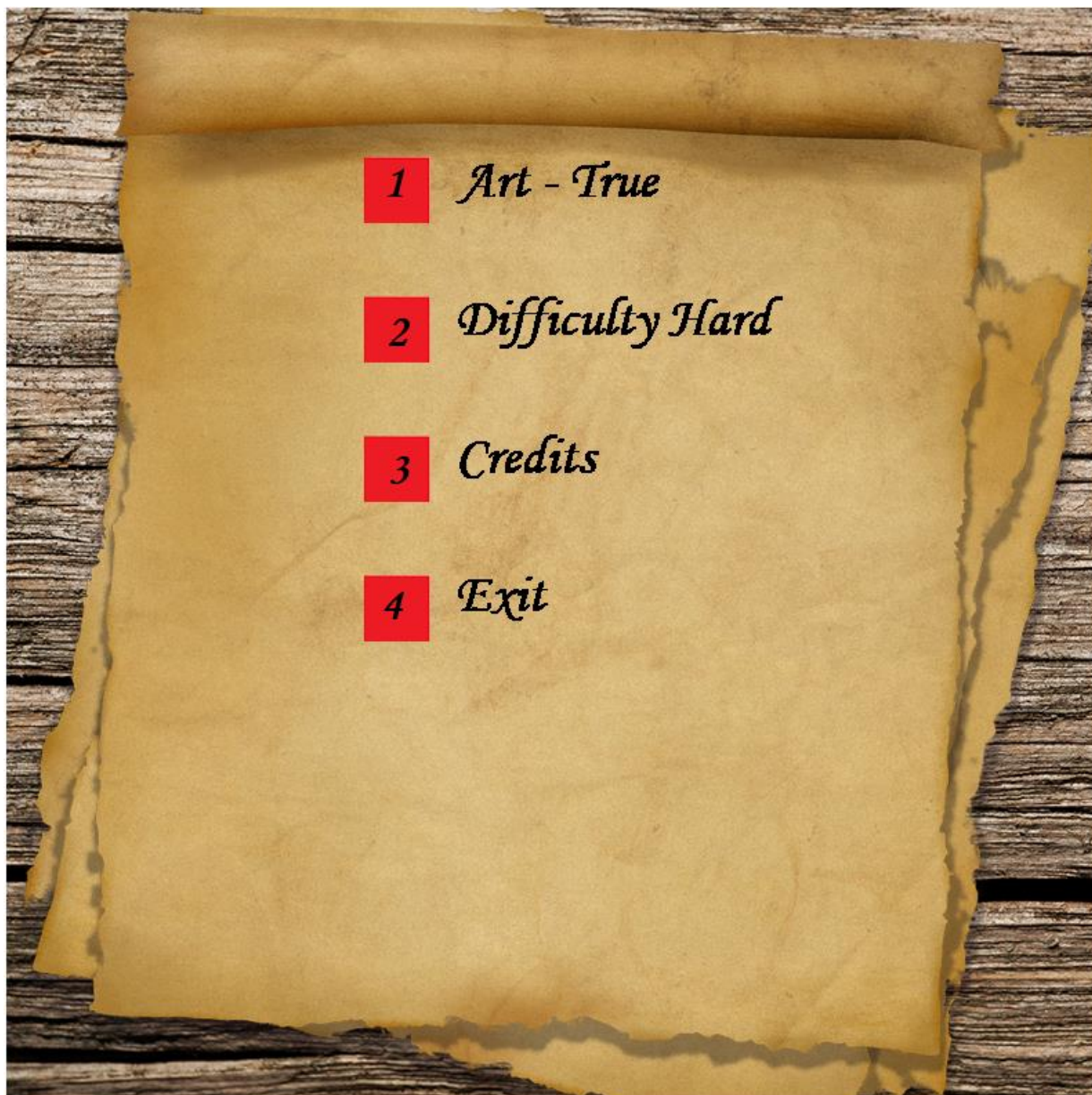
4.3 Зареждане на запаметени игри



1. **Save** – Задава екрана в режим запамятаване
2. **Load** – Задава екрана в режим зареждане
3. **Delete** – Задава екрана в режим изтриване
4. **Exit** – връща потребителя обратно в менюто от където е дошъл

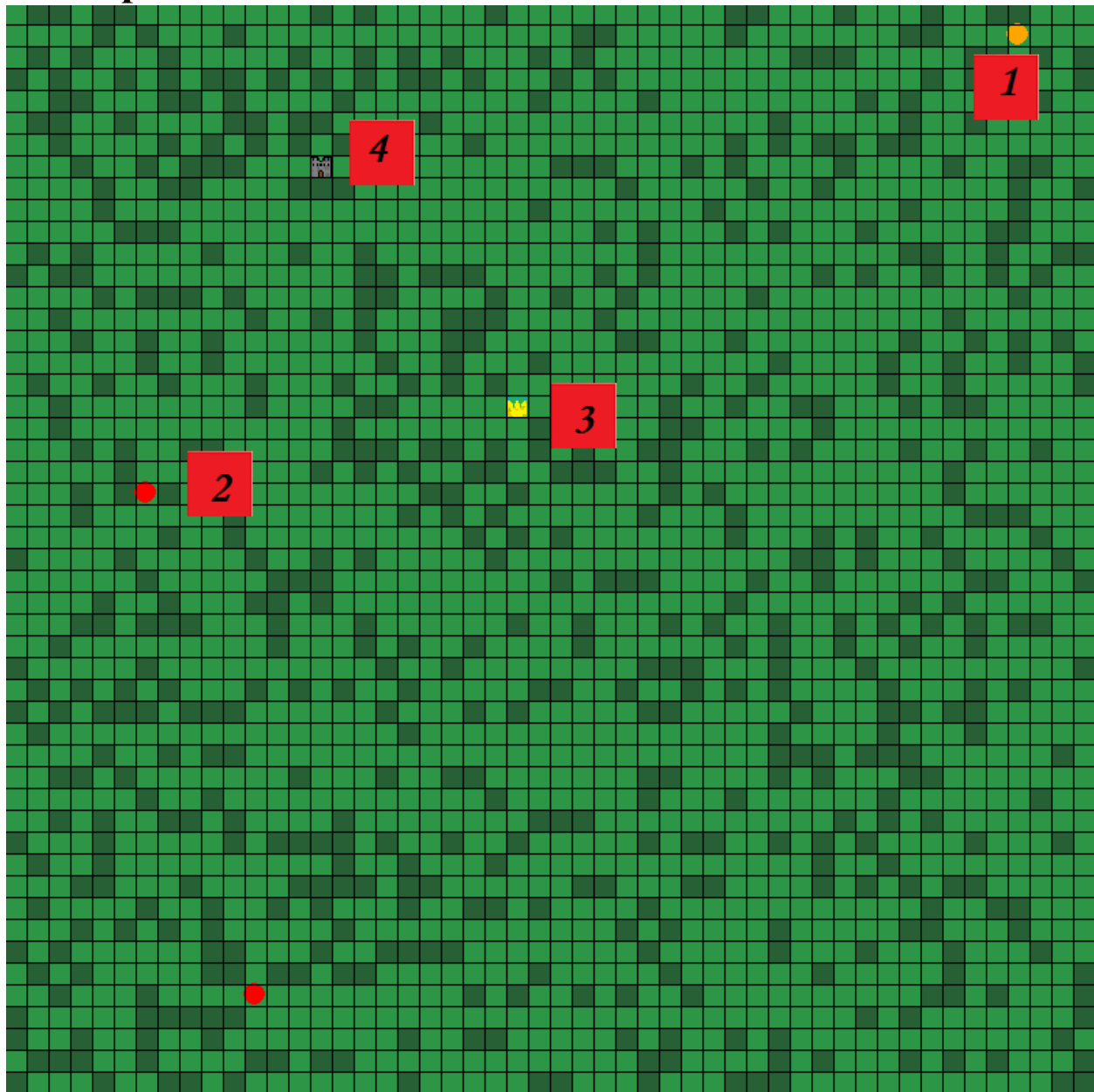
5. **1 save** -според затаения режим всеки от 9-те бутона може да запамятава, зарежда и трие игра







4.3 Опции



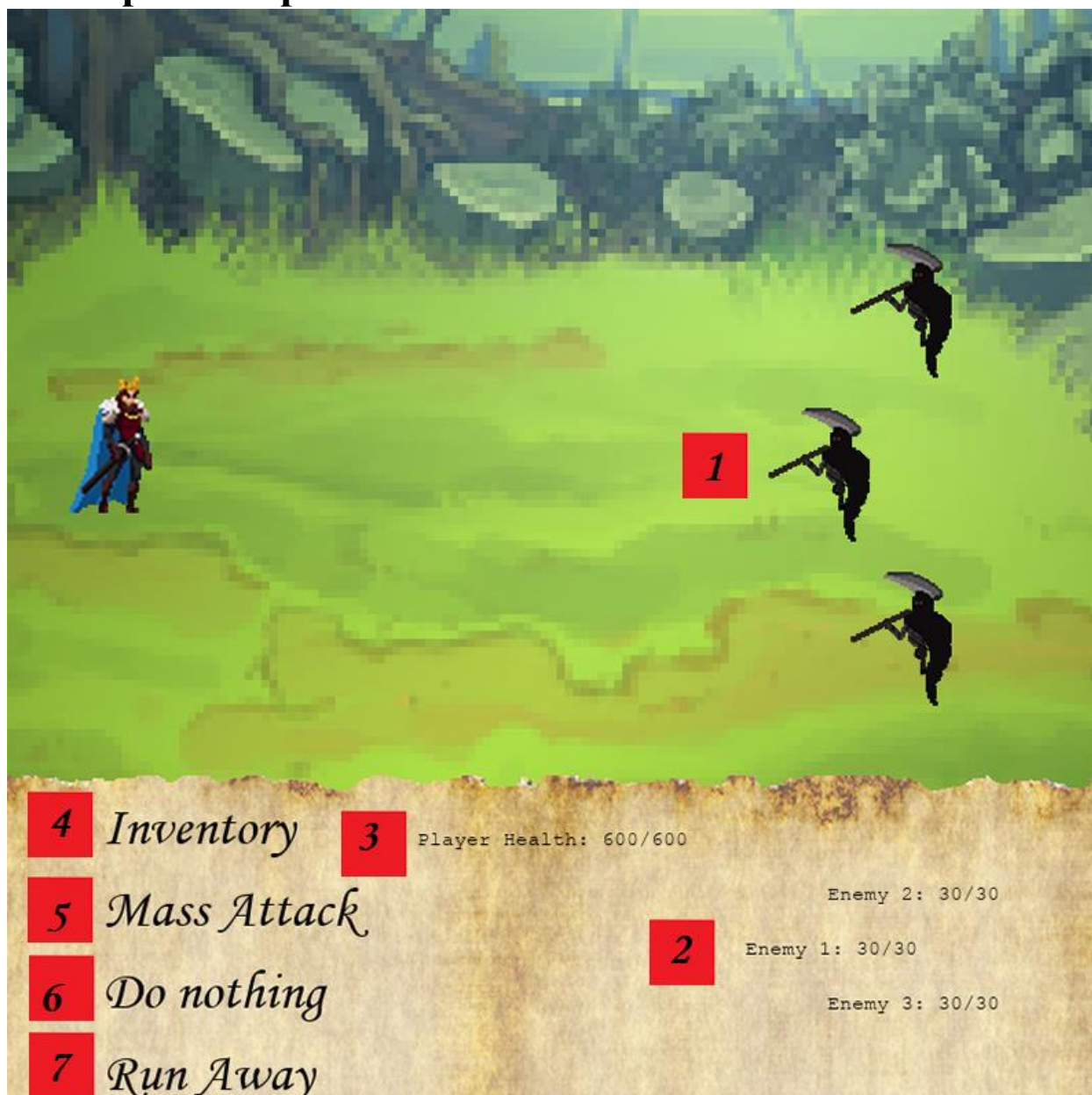
1. **Art** – изключва графичния интерфейс на приложението
2. **Difficulty** – променя сложността на играта ,на една от 3 стойности – **easy/medium/hard**
3. **Credits** – разработка по приложението
4. **Exit** - връща потребителя обратно в менюто от където е дошъл

4.4 Екран за навигация



-		Свободно пространство
-		препятствие
1		играч
2		противник
3		Цел
4		Запък

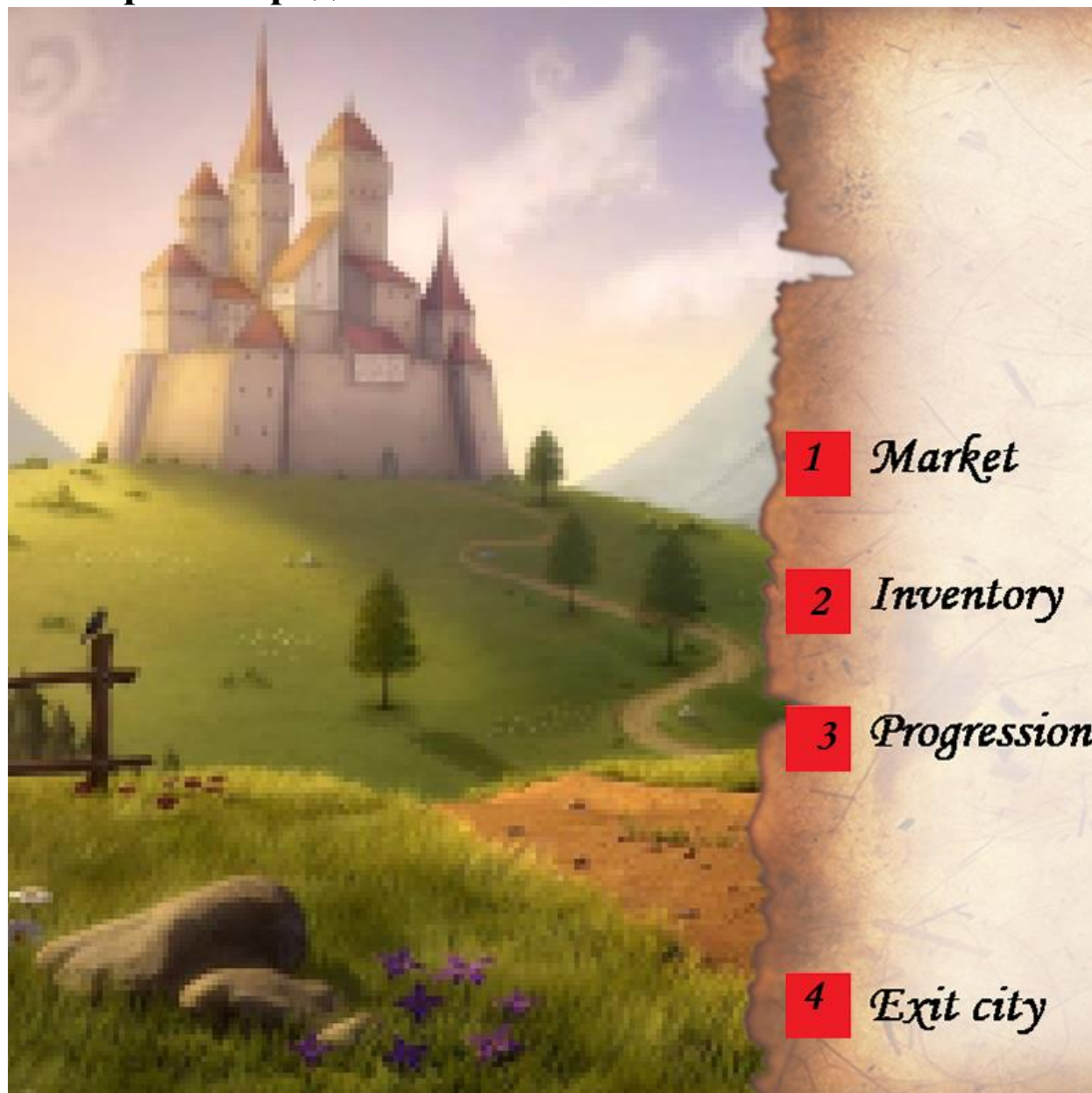
4.5 Екран за Сражение



1. **Противник** – при натискане с мишка играча извършва атака срещу селектирания противник.
2. **Жизнени точки на противника** – при успешна, атака точките на противника намаляват
3. **Жизнени точки на играча** - при успешна, атака на противника точките на играча намаляват
4. **Inventory** – играчът достъпва своя инвентар
5. **Mass attack** – играча извършва атака, която поразява всички опоненти

6. **Do nothing** – Пропускане на хода
7. **Run away** – Опит за напускане на битка

4.6 Екран за града



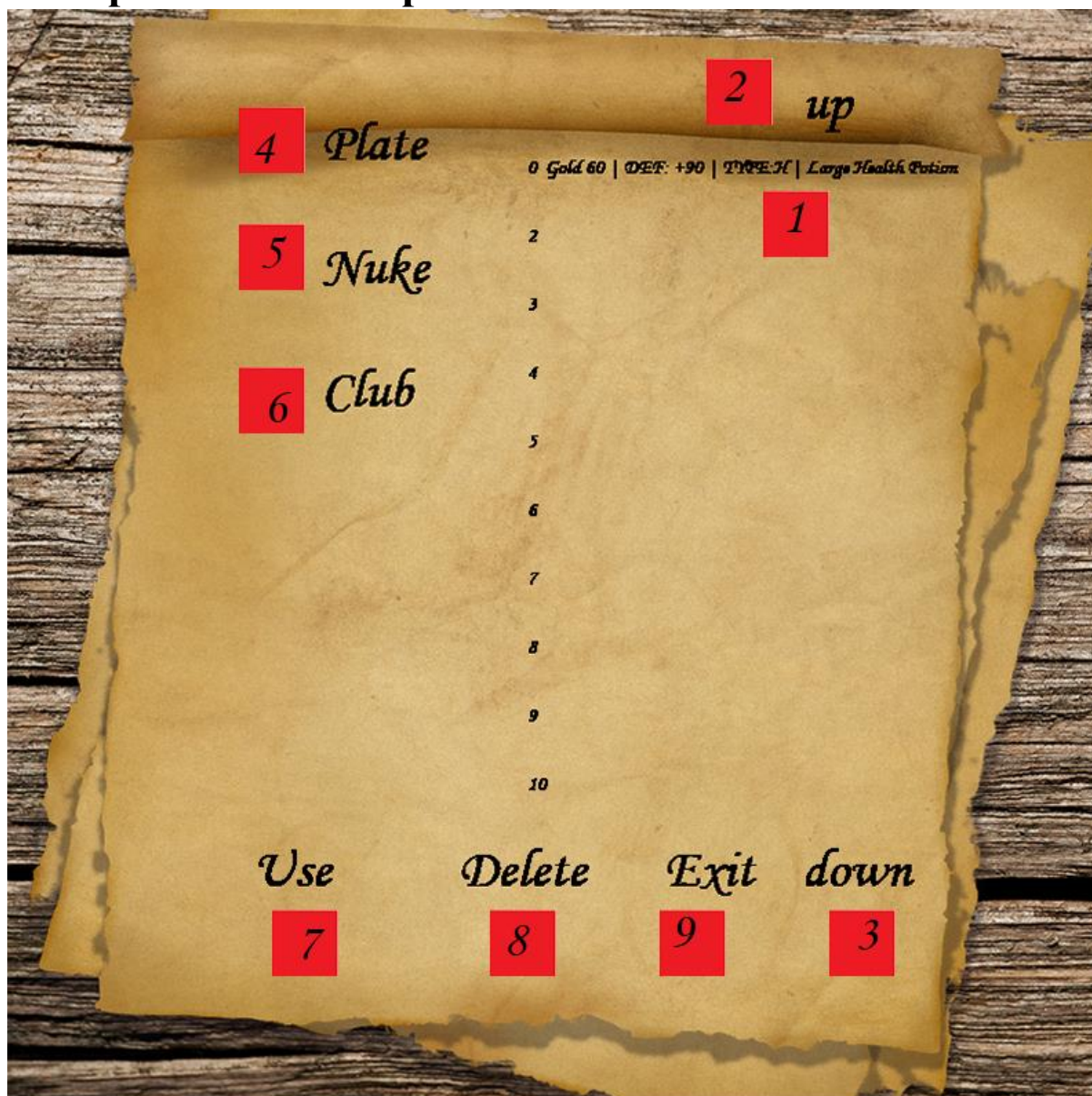
1. **Market** – отваря екрана за търговия
2. **Inventory** – играчът достъпва своя инвентар
3. **Progression** – визуализира прогресията на играчът
4. **Exit city** - връща потребителя обратно в менюто от където е дошъл

4.7 Екран за търговия



1. **Buy** - Задава екрана в режим купуване
2. **Sell** - Задава екрана в режим продаване
3. **Пари на играча** – показва с какви средства разполага играчът
4. **Инвентар на търговеца или играча** – в зависимост дали режимът на екрана е купуване или продаване, показва инвентара на играча или продавача.
5. **Up** – ако инвентара е повече от 10 артикула, превърта списъка нагоре
6. **Down**- ако инвентара е повече от 10 артикула, превърта списъка надолу
7. **Confirm** – потвърждава селектирана транзакция в зависимост дали екрана е в режим купуване или продаване/
8. **Exit** - връща потребителя обратно в менюто от където е дошъл

4.8 Екран за инвентара



1. **Инвенвентар** на играча – Визуализира инвентара на играча
2. **Up** - ако инвентара е повече от 10 артикула, превърта списъка нагоре
3. **Down** -ако инвентара е повече от 10 артикула, превърта списъка надолу
4. **Броня** – показва каква броня използва играча. Ако бъде кликнат с мишка, артикула ще бъде преместен в инвентара
5. **Оръжие за дясна ръка** – показва какво оръжие използва играча за дясната си ръка. Ако бъде кликнат с мишка, артикула ще бъде преместен в инвентара
6. **Оръжие за лява ръка**

7. **Use** – след като артикул бъде избран от инвентара. Артикула се използва от играча.
8. **Delete** - след като артикул бъде избран от инвентара. Артикула се изтрива от инвентара
9. **Exit** - връща потребителя обратно в менюто от където е дошъл

4.9 Екран за прогресия на играча



1. **Ability points** – показва колко свободни Ability points има играчът. Излизват се за повишаване на 4-те атрибута (**STR/END/DIP/DEX**)
2. **Stage level** – показва нивото на прогресия, на играта
3. **Player XP** – показва нивото на опит, на играча. Когато опита мине границата, играча качва ниво.
4. **Player level** - показва нивото на играча. Когато играча качи ниво, получава 1 **Ability point**
5. **Strength** - показва силата на играча. STR определя силата на ударите
6. **Endurance** – показва издръжливостта на играча. END определя издръжливости на удари.
7. **Diplomacy** - показва дипломацията на играча. DIP определя цените на пазара
8. **Dexterity** - показва силата на играча. DEX определя шанса играча да избегне и нанесе атака.
9. **Умения** - показва индивидуалните умения на играча
10. **Exit** - връща потребителя обратно в менюто от където е дошъл

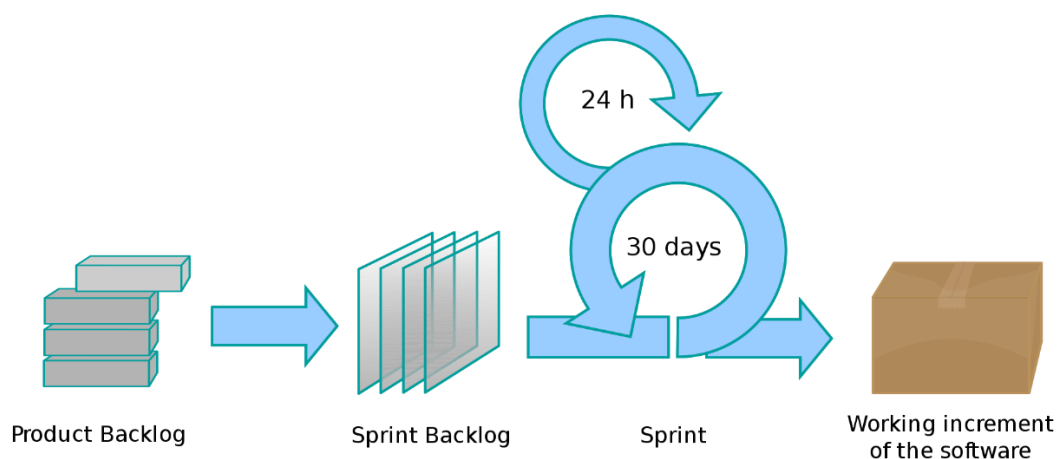
5. Изводи и заключение

Разработката на видео игра, без помощта на вече съществуващи инструменти или frameworks е възможен, макар и труден процес. За създаване на 2D role playing игра на чист Python трябваше да се проектира и разработи:

- Система за сражение
- Генерация на карта за навигация
- Алгоритъм за навигация на противниците
- Система прогресия на играча
- Система за търгуване
- Система за запаметяване и зареждане на игри
- Графични асети

5.1 Процес на разработката

За разработката на приложението се използва SCRUM методология за разпределяне и организиране на работа. SCRUM е итеративна, инкрементална рамка за управление на проекти, в софтуерно инженерство. Процесът се състои от отделни итерации, наречени спринтове. Спринтовете могат да имат продължителност от една седмица до четири седмици. В края на всеки спринт екипът разполага с работеща версия на продукта, която включва всички готови задачи от backlog-a.



На първо началните задачи се дава времева естимация, за колко време може да бъде изпълнена и имплементирана. Според времето и ресурсите с които

разполага разработчика на приложението може по-точно да приоритизира задачите от backlog-a.

5.2 Бъдеще и надграждане на приложението

Потенциални авенюта за развитие са:

- **Добавяне на изкуствен интелект при навигацията на противниците** – Чрез python библиотеката SKlearn, може да се добри основен изкуствен интелект при взимането на решения, на противниците. В сегашната итерация на играта, противниците просто преследват играча. В бъдеще противниците могат да предсказват на къде ще се придвижи играча.
- **Анимиране на графичните асети** – Могат да се наемат уменията на графичен дизайнер, който да създаде анимирани асети, специфично за апликацията.
- **Добавяне на звук** – Ругаме включва основни функции за интеграцията на звуци в играта. Въпреки това, звуковия дизайн е комплексна тема, коят
- **Разширяване на подходите за водене на сражение -**
- **Добавяне на повече типове терен -** В сегашната итерация на играта, има само 4 типа терен, Вода, пясък, лед. Всеки от които ще има различни свойства.
- **Добавяне на мисии и задачи за играча** – В градовете могат да се добавят места където играча може да говори с NPC(None player character), което да му възложи задача за изпълнение.
- **Добавяне система за създаване на артикули** – В градовете може да бъде добавена работилница, където играча може да създава артикули по-силни от тези които може да купи от града.
- **Добавяне на различни типове архетип за играча** – когато играча започне нова игра, има опцията да избере архетип който определя какви оръжия и специални сили може да използва.
- **Банкиране** – В градовете играча ще има опцията да тегли, влага и тегли пари от банка. Всеки ден (игрален ден) вложените пари в банката се увеличават с 1%.
- **Мини игри -** В градовете може да се добави таверна, където играча може да играе карти срещу компютъра и да залага пари.

Използвана литература:

- [1]. Python - практическо програмиране / 2015 – издателство Асеновци
- [2]. Hands-On Data Structures and Algorithms with Python - Second Edition - Dr. Basant Agarwal , Benjamin Baka
- [3]. Wikipedia – Python
- [4]. Wikipedia – A* search algorithm
- [5]. Wikipedia – SQLite
- [6]. Wikipedia - Pygame