

# ДИПЛОМНА РАБОТА

Тема: Софтуерно обезпечаване на развойна гейминг конзола, базирана върху  
arduino и raspberry PI

Дипломант:

*Момчил Ангелов*

Научен ръководител:

маг. инж. Росен Витанов

С О Ф И Я

2 0 1 6





# **Увод**

В света на видео игрите конзолите заемат все по-голямо място. Те имат редица преимущества:

- По-лесен механизъм за участие на повече от един играч при липса на интернет: свързване на допълнителен контролер към конзолата. При персоналния компютър трябва да се разполага с допълнителен настолен компютър или лаптоп;
- Спрямо тежестта на игрите: компютърът е по-бавен в сравнение с конзолата поради бързо променящите се изисквания на игрите, което довежда до проблеми при изпълнението на най-новите игри при по-стари компютърни конфигурации;
- По-евтини: един мощен компютър (за времето си, той става относително слаб по-бързо от конзолата) струва около 1500 лева, докато конзолата - 500lv;
- Един компютър се амортизира спрямо игрите по-бързо от конзолите, и се нуждае от постоянни подобрения на хардуера, докато конзолата- не се нуждае от това;
- Повече внимание от страна на софтуерни разработчици – една конзолна игра се пише да работи върху една конзола, хардуер, докато при компютъра се пише за много различни конфигурации. Поради тази причина са нужни повече бъг фиксове – затова все повече и повече заглавия излизат първо за конзола, и след това – за компютър, поради причини като пиратство и липса на интерес;

В глобален мащаб има тенденцията хората да използват компютъра си само за работа. Телефонът и таблетът вече изместват компютъра. Устройствата стават с все по-малки размери. Хората предпочитат да играят на телефона си или на

конзолата, отколкото на компютъра. Най-новите тенденции са в така наречени „wearable“ технологии – например smart watch, smart glasses, smart wrist и други. Компютрите вече не се използват за забавление, а за работа – изместени от таблети, телефони и конзоли.

Всичко това става благодарение на иновации в света на електронните елементи, които позволяват на все по-малка площ да се слагат все повече транзистори, увеличавайки производителността и отваряйки нов свят за разработчиците. Минусът е, че вече можем да носим технологиите навсякъде – докато преди можеше да излезеш навън и да няма техника по и около теб, то вече не е така – телефона/таблета/четеца на книги/умния часовник са навсякъде, което в известна степен намаля физическия социален контакт, който според хората е незаменим.

Ето така се роди идеята да бъде създадена гейминг конзолата, начин хората да се забавляват заедно на едно място.

# 1. Първа Глава – Обзор на съществуващите развойни игрови системи както и на embedded системи за бързо прототипиране

## 1.1. Arduboy

Arduboy е игрова конзола, с големината на кредитна карта (фиг. 1.1.1.). Какво е общото с нашия проект:



Фигура 1.1.1. Игрова конзола Arduboy

може да се програмират игри, или да се изтеглят от сървър. Какви са разликите: качването на игри става чрез компютър, а не през Интернет директно от някакъв сървър . Няма никаква комуникация с външния свят, така че няма поддръжка на multiplayer. Хардуерно статичен – не могат да се добавят допълнителни хардуерни модули. Има големина на кредитна карта

(90/45/12 mm)<sup>[1]</sup>.

Програмиране на Arduboy става чрез C++ файлове в които седят действителните спрайтове, хедър файл в който седят прототипите им, и .ino файл, в който е кода на самата игра. Друг минус е, че игрите са малки по обем заради ограничната памет

(2,5 КБ RAM памет и 1 КБ EEPROM)<sup>[2]</sup>.

## 1.2. GAMBY

GAMBY е игрова конзола, която предлага възможност не само да си правим



*Фигура 1.2.1. Игрова конзола Gamby*

*Фигура 1.2.2. Gamby игрално поле*

сами игри, но и да поддържаме различен от стоковия хардуер, тоест можем да подменяме елементи с такива, каквите имаме. Минус отново е нуждата от компютър за инсталација на игри, както и липсата на multiplayer<sup>[3]</sup>.

В GAMBY, както и в Arduboy, трябва да създаваме огромни масиви за видео памет (Показано на фигура 1.2.2. , взет от примерния код, качен на github акаунта на GAMBY)<sup>[4]</sup>.

Всички тези огромни масиви са в основния код на програмата, а не както при Arduboy – в отделен файл. Това прави четенето и писането на код на GAMBY по-трудно спрямо на Arduboy. Основната библиотека се казва GAMBY, и в нея има

начини за манипулация на спрайтове, рисуване на спрайтове, както и режими на работа (графичен или текстови). Текстовият се дели на нормален (бял фон и черни букви), обратен (черен фон, бели букви), зачертан и подчертан. Графичният имплементира самите спрайтове и техния цвят и поведение.

### 1.3. Fuzebox

Fuzebox е игрова конзола, която поддържа двама играчи. Няма вграден

екран, затова се връзва чрез RCA към экран (пр. телевизор). Не поддържа връзка с Интернет,. отново трябва да ползваме компютър за качване на игри. Има изискване инсталацирането на игри да стане в първите 3 секунди от стартирането на игровата конзола. Друг минус е че



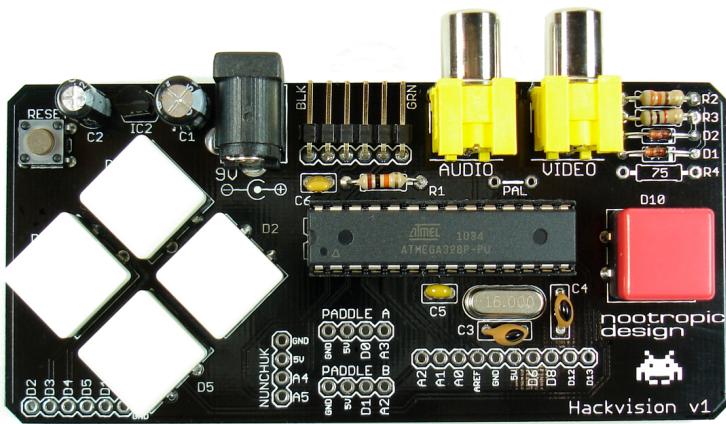
1.3.1. Игрова конзола Fuzebox

поддържа само стокови контролери, няма това хакване като при GAMBY<sup>[5]</sup>.

Самите игри са на C, а не .ino файлове. Библиотеката е направена да поддържа аудио и видео, цветовете са от 256 комбинации (1 байт в разпределение: 3 червени – 3 зелени – 2 сини бита). Резолюцията е 360x244 пиксела, в 3 различни варианта – полета, спрайтове и полета, и битмап. Предоставя възможност за до 32 спрайта на полето<sup>[6]</sup>. Поддържа контролери от NES семейството, включително и NES мишка.

## 1.4. Hackvision

Hackvision поддържа връзка към екран чрез RCA, свалянето на игри чрез USB to TTL кабел, писането на игри и драйвери за използването на собствен



#### *1.4.1. Игрова конзола Hackvision*

хардуер. Минусът е в това, че пак трудно се качват игри, нужен е компютър, няма собствен дисплей, и не поддържа *multiplayer*<sup>[7]</sup>.

Паметта с която разполага (SRAM) е 2КБ, като библиотеката за писане по экрана използва 1.5КБ от нея при 128x96 пиксела, и 1632 байта при резолюция 136x96 пиксела. Трябва да

се използват методи за пестене на паметта, като съхранение на символни низове, използвани в изобразяването им по екрана, чрез ключовата дума за ардуино PROGMEM. Използване на възможно най-малки типове за съхранение на данните (което може да доведе до преливане или преобръщане на стойностите запазени там), и писане на код който вика в функции в дълбочина над 10-20 функции. Работи с 2 основни библиотеки – TVOut и Controller, като TVOut предава данни на екрана, а Controller – управлява контролера<sup>[8]</sup>. Конструктора на Controller е функция, приемаща като параметри номерата на пиновете на бутоните, откъдето да взима информация за текущото състояние на контролера. Свързани са чрез pull-up резистор, като при натискане на бутон се установява ниво нула на съответния пин.

## 1.5. DIY Gamer Kit

DIY Gamer Kit е игрова конзола, която не поддържа multiplayer. Липса на модулност на хардуера – използва вграден дисплей и не може да се върже към

екран. Поддържа само една игра, която е предварително заредена в паметта чрез компютър. Има гравна, с цел да не го изпуснеш и повредиш<sup>[9]</sup>.

Поради цялата си статичност и малък екран, библиотеката която го управлява е относително малка, в сравнение с предишните. Дава възможност

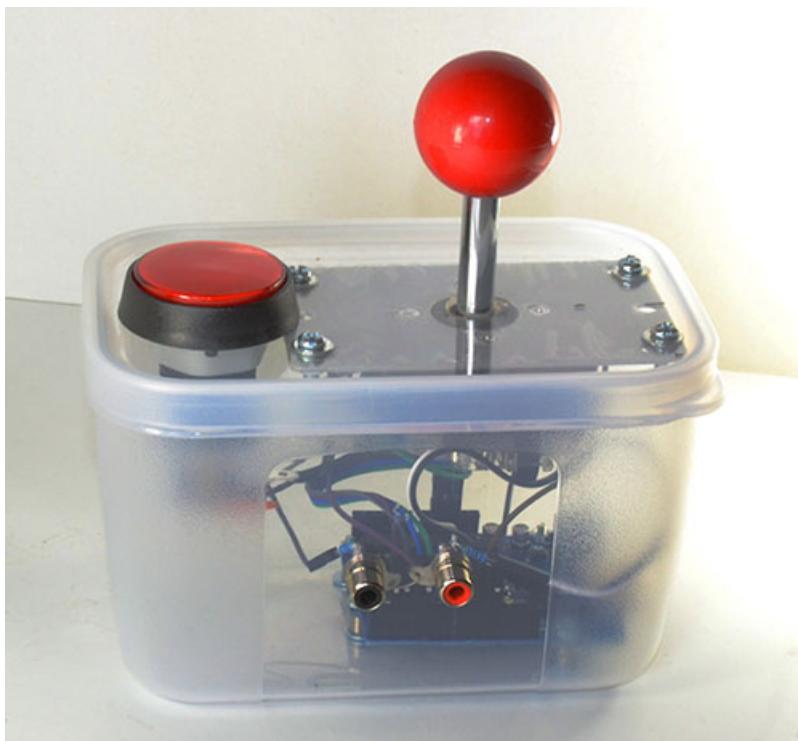


Фигура 1.5.1. Игрова конзола DIY Gamer Kit

за музика към играта, рисуване на спрайтове, писане на букви по екрана чрез предварително предифинирани битмапове, пазене на данни в енергонезависима памет<sup>[10]</sup>. Използва арduino uno, което го прави по-слабо от предните конзоли, повечето от които използват по-мощното ATMEGA644. Поради ограничение на паметта функцията която изписва резултата на текущата игра дава лимит от максимум 2 цифри.

## 1.6. Retro Gamebox

Retro Gamebox е игрова конзола, която няма собствен еcran и се връзва чрез RCA към еcran. Не поддържа multiplayer, нито повече от една игра едновременно.



Фигура 1.6.1. Игрова конзола Retro Gamebox

За да се качи игра е нужен компютър. Не поддържа различен от стоковия хардуер, както и multiplayer.

От софтуерна гледна точка, използва библиотеката за писане по еcran, използвана и от Hackvision. Това го прави възможно без особен проблем да пише по екрана на телевизора. Ако телевизора е старо CRT с PAL, то трябва да заземим D12 пина на ардуиното

за да му кажем, че ще работим с PAL кодировка. На по-новите телевизори това не е нужно, защото поддържат и PAL и NTSC кодировка<sup>[13]</sup>.

## 1.7. Gamebuino

Gamebuino е игрова конзола, която дава огромна свобода на програмиста. Идва с вграден дисплей, спекер и възможност за разширяване на хардуера. Може

да се пишат игри на него,

но трябва да се качват  
през [компютър](#).

Поддържа връзка с друго  
Gamebuino, което му дава  
минимален *multiplayer*<sup>[11]</sup>.

Връзката с другата  
конзола става чрез  
RX/TX. Библиотеката на  
Gamebuino е най-  
богатата от всички



Фигура 1.7.1. Игрова конзола Gamebuino

разгледани досега игрови конзоли. Включва странично осветление на конзолата, показване на информация за батерията, рисуване и писане по еcran, издаване на звук, поддръжка на спрайтове и битмапове. При създаване на игрови свят, той се запазва в двумерен масив, правещ го твърде голям за RAM паметта, затова се запаметява във flash паметта на микроконтролера. Има различни функции като чертане на кръг, запълването му, линии и задаване на цветов, както и пускане на клавиатура. Цветовете на екрана са само черни и бели, резолюцията е 160x144 пиксела<sup>[12]</sup>.

## 1.8. Заключение

Минусът на всички тези устройства, е нуждата от компютър за подмяната на игрите. Друг минус е, че не всички поддържат различен от фабричния хардуер, както и липсата на multiplayer и вграден экран. Това ги прави зависими от еcran или компютър.

Предложената конзола преодолява тези недостатъци – освен един минимален setup при пускането на конзолата за първи път, няма да е нужно никаква друга хардуерна намеса. Ще има вграден еcran, както и механизъм за инсталлиране и подмяна на игри. Също ще може да поддържа до 4 играча, както и Multiplayer чрез безжични комуникация между конзоли, позволяваща да се играе с приятел, който не е при нас.

Повечето игрови конзоли, които разглеждахме не поддържат реална абстракция на хардуера, не поддържат адекватен Multiplayer, не поддържат 3D еcran, не поддържат сваляне на игри директно от конзолата. Това прави всичко много зависимо от компютър, и не е самостоятелно, както се предполага от една истинска игровая конзола.

## **2. Втора Глава – Функционални изисквания към програмния продукт и развойната среда. Аргументация за избор на програмна среда.**

### **2.1. Функционални изисквания към програмния продукт и развойната среда**

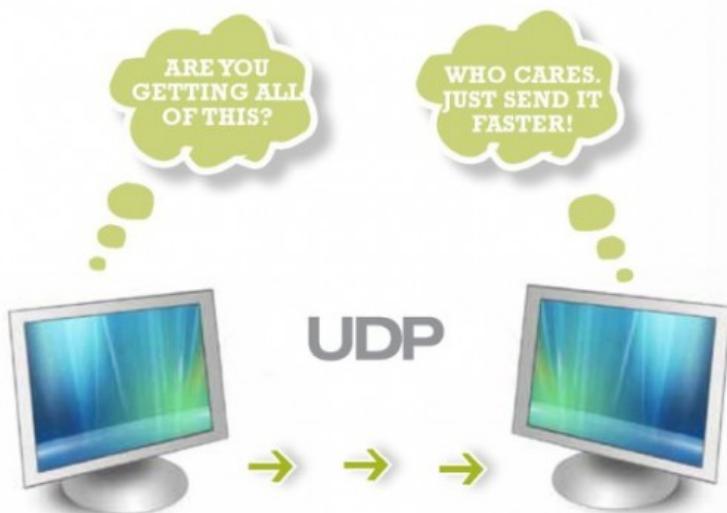
Изискванията към софтуерното обезпечение на игрова конзола са:

- Възможността за писане на игри за нея;
- Комуникация между конзола и сървър;
- Комуникация между конзола и контролер;

За да поддържаме повече от един играч, както и възможност да се изтеглят игри от отделен сървър, то ни е нужна комуникация. Тази комуникация се случва по дадени правила, описани в протокол. Протоколи са например TCP, UDP, IP, HTTP, LMP, L2CAP, SDP<sup>[18]</sup>. Първите четири, TCP, UDP, IP и HTTP се използва при интернет комуникацията. HTTP предава данни по мрежата, с помоща на TCP или UDP, които са едно ниво над IP<sup>[15][16][17]</sup>. LMP, L2CAP и SDP са протоколи, използвани за предаване на данни чрез Bluetooth технологията. LMP се използва за правене и контролиране на връзка между две устройства, чрез L2CAP мултиплексираме множество връзки между две устройства, а чрез SDP откриване други Bluetooth устройства<sup>[14]</sup>.

## Комуникация между контролер и конзола

Протоколите за управление на комуникацията между микроконтролерите ще бъде UDP. Той ще се ползва за комуникацията между конзолата и контролера, с цел по-голямо бързодействие и по-бързо пращане на сигналите. UDP обаче има проблеми като ненадеждност за получаване на всички данни, пращани от конзолата към контролера и обратно (както е показано във фигура 2.1.1., UDP не се интересува от данните които получава) , липса на проверка за грешки, липса на потвърждение за получени пакети. Минусът на UDP идва от самия протокол – докато в TCP чрез този „handshake“ езикът Python може да отвори порт и конзолата да си „говори“ с всеки един контролер на отделен порт, то в UDP имплементацията на python не се предоставя тази възможност – цялата информация към конзолата от контролерите се приема от един порт, което изисква нуждата от синхронизация



Фиг. 2.1.1. UDP протокол

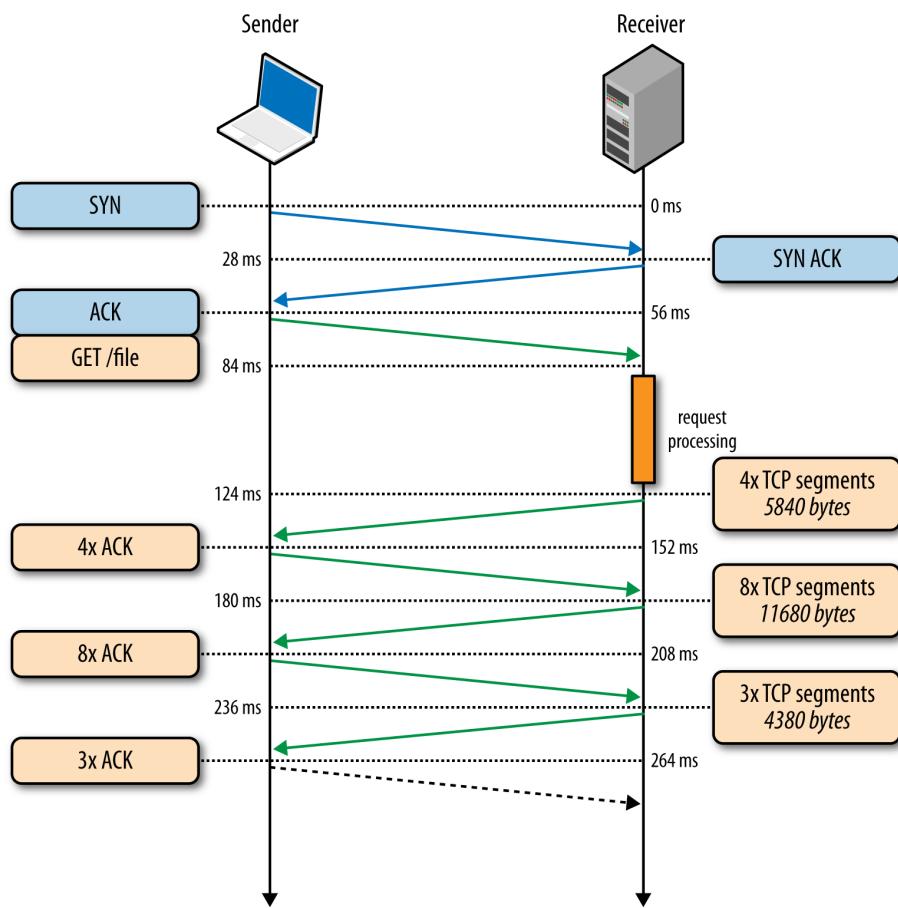
и изчакване. UDP се използва като игрови протокол, когато данните трябва да се пращат в реално време, защото няма проверки за коректно пратени пакети, или забавяне заради „handshake“. Този протокол също така се ползва и в стриймването на видеота, пращане на информация от

метеорологични станции, и навсякъде, където ни е нужен бърз отговор на заявката, която сме пратили.

## Комуникация сървър – конзола

Комуникацията между сървъра и конзолата ще се изразява в това да се

пращат игрови файлове на конзолата от сървъра. Поради нуждата от правилно подреждане на файловете, пратени от сървъра към конзолата, протокола който ще се ползва е TCP. Той ще се погрижи файловете да са правилно подредени и пълни, което в случая с UDP не е възможно без допълнително разширение на протокола от страна на програмиста. Също така и скоростта не е от първостепенно значение, защото спрямо големината на файла, закъснението от „handshake“ и отговорите за получен файл ще са малки в сравнение с прашането на самия файл(на фигура 2.1.2. е показан принципа на действие на протокола).



Фиг. 2.1.2. Работа на TCP протокол за комуникация

Както се забелязва, TCP протокола има голямо количество проверки за това дали информацията се е изпратила. Друг проблем на протокола за използване в комуникацията между контролера и конзолата е първоначалния „handshake“. Това ръкостискане е един от механизмите за гарантиране на правилното предаване на

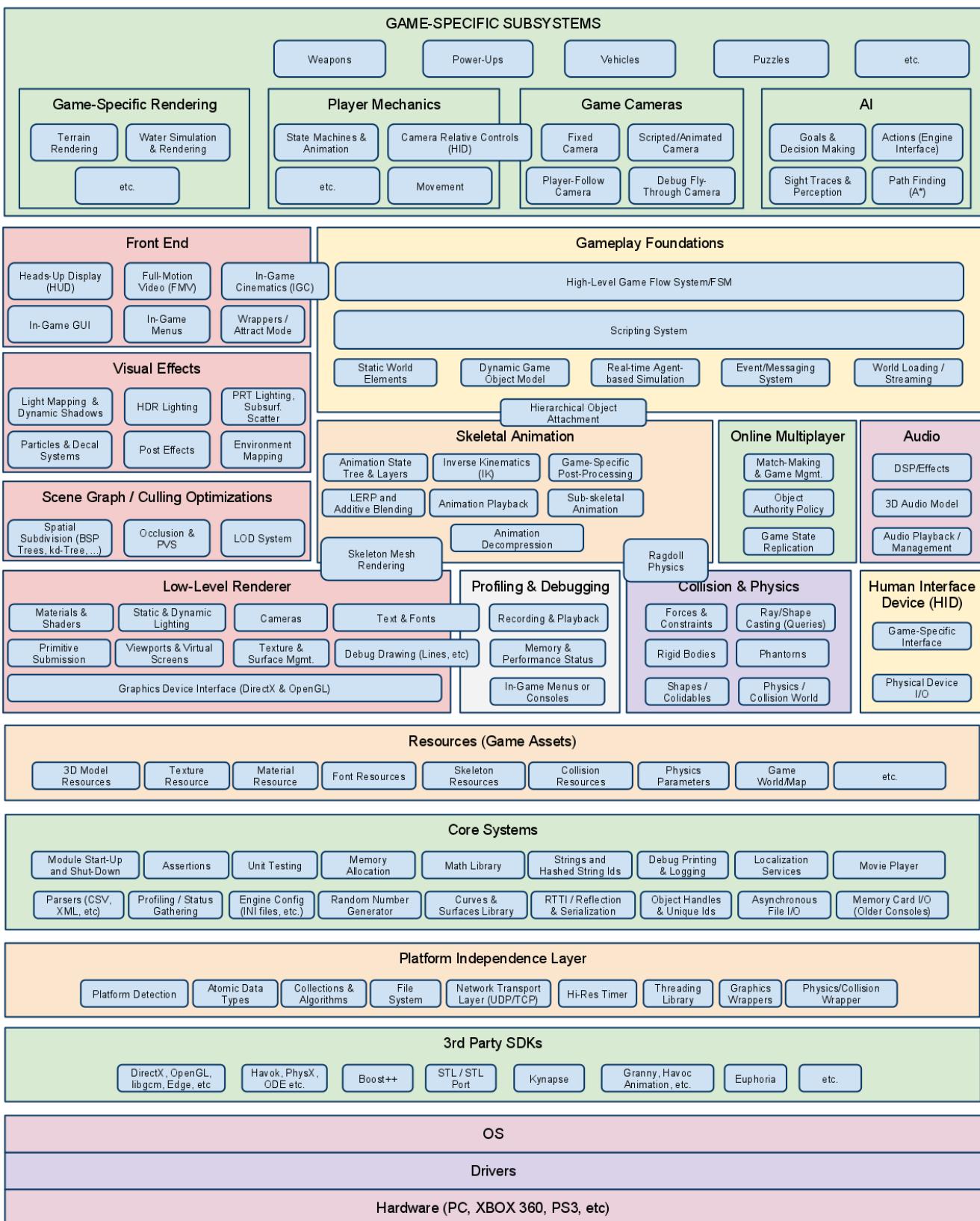
данни, но също и забавя процеса на комуникация, в зависимост от скоростта на мрежата.

## Проектиране на игри

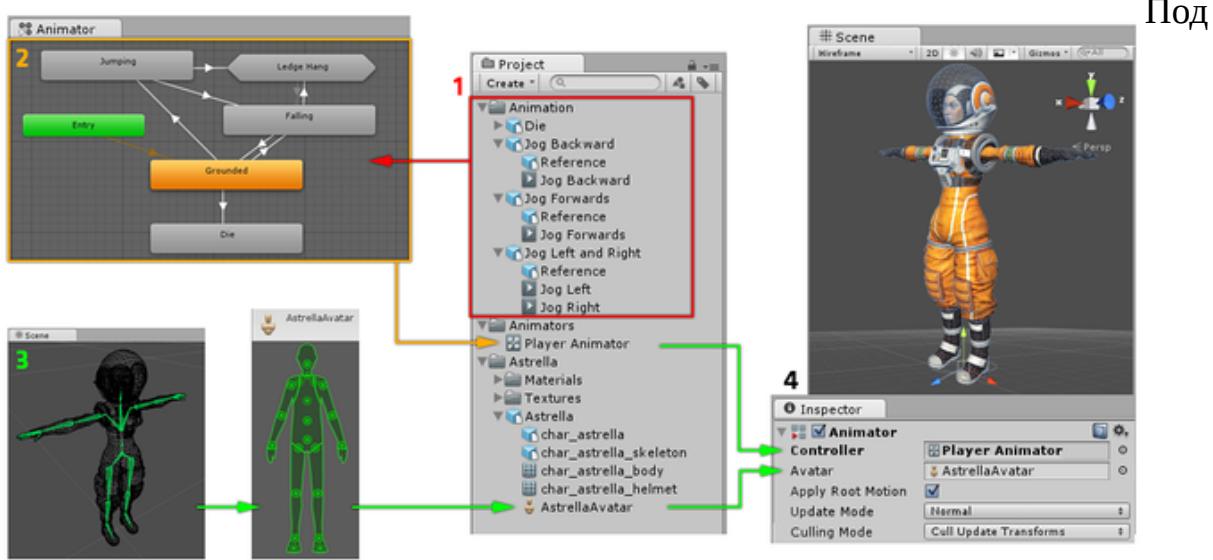
За по-бързо и лесно проектиране на игри, програмистите са направили така наречения „game engine“. Чрез този game engine, човека който иска да направи игра, може да го направи много по-бързо, отколкото ако почне от нулата. Този game engine му предоставя готов код с едни от най-използваните функции, за да не се съсредоточат усилията на програмиста в това какво е специфично за дадената игра и игрова логика, а не общото – то вече е написано и готово. В модерните компютърни системи, game engine-ите са огромни и сложни, както се вижда на фигура 2.1.1. <sup>[19]</sup> Те трябва да имат:

- Механики, които играча използва;
- Изкуствен интелект;
- Рендерване на полета и фонове;
- Избор на различни начини за визуализиране на полето чрез позицията и типа на камерата;
- Front End – в това се включва графичния интерфейс, менютата, филми вътре в играта;
- Основа на самата игра – скрипторска система, статични световни обекти, динамични игрови обекти, известяване при събития, зареждане и стриймване на игровата вселена;
- Визуални ефекти – светлини, сенки, частици, анимации и ефекти, мапване на околнния свят;
- Възможност за много играчи – уеднаквяване на игралното поле за всички играчи, система за свързване на играчи и точкова система за тяхното класиране, персонализиране на обектите (кой кой обект притежава);

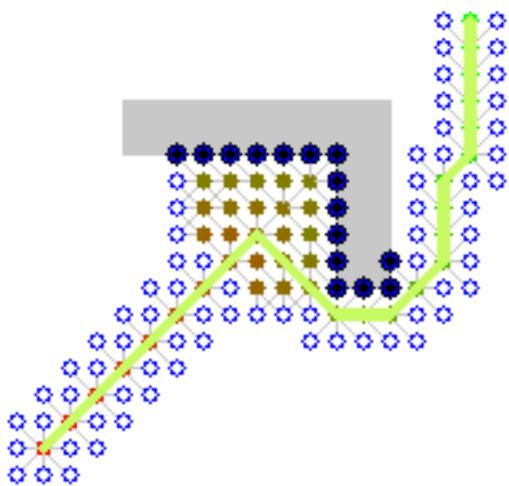
- Аудио система – 3D аудио модел, плейбек на аудио, управление на аудио, аудио ефекти;
- Контролер – физическото устройство, чрез което потребителя навигира из игровата конзола;
- Сблъскаване и физика на обектите – сили и ограничения, фантоми, форми които могат да се блъскат, граници на Вселената;
- Ресурси – 3D модели, текстури, материали, шрифтове, основи (скелети), физически параметри, игровата Вселена;
- Основни системи – модули за стартиране и изключване, unit testing, заделяне на памет, математическа библиотека, принтиране на информация при дебъгване, плейър за визуализиране на филми, асинхронна работа с файлове, работа с карти с памет, парсъри, инициализиращи файлове;
- Основни системи, предоставени от операционна система (platform independant) – основни типове данни, контейнери и алгоритми, файлова система, начин за пренос на данни (TCP/UDP), библиотека за работа с нишки;



Фигура 2.1.1. Основа на модерен game engine



*Фиг. 2.1.2. Свързване на модел, анимация и логика на анимациите в Unity3d*  
 механики, използвани от играта, се има предвид движение на обекта, управляем от играта, анимациите му, движение на камерата. На всеки игрови обект съответства модел, и анимация. Под анимация може да се разбира и поведение на самия обект, като движението му в пространството. Анимация е общо название, което се използва като „отговор на обекта при някакво събитие“ - движение, удар, стоеене на едно място, катерене и т.н. (на фигура 2.1.2. може да се види имплементацията на тези механизми от Unity game engine<sup>[20]</sup>).



2.1.4. Краен резултат на A\* алгоритма за намиране на път

Под изкуствен интелект се разбира възможността играта да играе срещу играчи, контролирани от компютър (NPC – non player character). Той е много по-различен от реалния изкуствен интелект, с който се занимават учени от различни области на науката<sup>[21][22]</sup>. В игровия изкуствен интелект се използват ограничения, заложени от самата игра. Поради тези могат да се програмират по-лесно. Не е нужно и обратната

връзка, тоест изкуствения интелект да се „учи“ от действията на играта, както е например при невронните мрежи.

Проблемът на игровия изкуствен интелект е, че даже и да е по-лесен от реалния такъв, то на него му трябва огромно количество процесорна мощ за вземане на по-сложни решения. Основните действия, които трябва да прави един игрови изкуствен интелект е да разбира каква е неговата цел и да взима адекватни решения спрямо това (дали да тръгне нагоре или надолу при посрещане на топката при играта Pong например – фигура 2.1.3.), да вижда светът около себе си и спрямо това да взима адекватни решения, и да намира най-бързия път до дадена точка А(фигура 2.1.4.)<sup>[23]</sup>.

Под рендеринг на полета и фонове се има предвид рендерването на самата игра. Това е специфично за всяка една игра (всяка игра си има собствени ресурси, модели и материали на обектите). Този раздел също включва и рендерване на симулацията на течности (примерно вода).

Под front end се има предвид всичко което вижда потребителя, без да може да го контролира. Това са статичните части на играта – сцени и менюта, филми (фигура 2.1.5.) и вградени графични интерфейси (фигура 2.1.6.).



Фигура 2.1.5. Сцена от играта WoW

Основи на геймплея – съдържа няколко части:

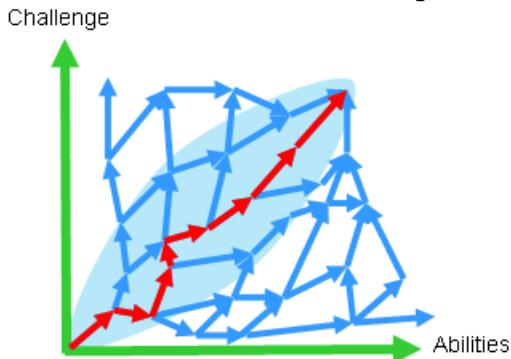


Figure 7 Active Flow Adjustment through Choices

#### 2.1.7. Пример за Active Game Flow

High Level Game flow system – това е един вид FSM (finite state machine). Можем да разгледаме играта като последователност от решения, и всяко решение е някакъв state, състояние на играта<sup>[24]</sup>. Това е нагледно показано в абстракцията на игра, фигура 2.1.7. Съществуват 2 вида game flow –

активен и пасивен. Активния е когато играча прави избор, и така може да нагласи дали играта да стане по-лесна или по-трудна, като по този начин сам регулира



GUI Textures

Фигура 2.1.6. Вградени графични интерфейси

трудността. В пасивния game flow, играта минава по статичен път, който не се променя, и така играта става еднообразна, скучна, и без стойност при повторянето. Не се разбира нищо ново, не се открива нова история и т.н. Друг минус е, че ако

играта е твърде сложна или твърде лесна, потребителя не може да нагласи играта спрямо своите възможности, както при активния game flow.

- Скриптова система – под скриптова система се има предвид свързването на всички компоненти на game engine. Така например могат да се свържат частта която дефинира обекти (животни или други), с частта която ги използва, без те да са на едно място. Така в самия game engine нямаме дефиницията на тези обекти, те ни се предоставят от отделно място, което прави имплементацията на нов обект много по-лесно. Начина, по който се представят обекти в играта при тази система, е чрез тяхното дефиниране, след което тази дефиниция се парсва от парсер и се вкарва в game engine<sup>[25]</sup>.

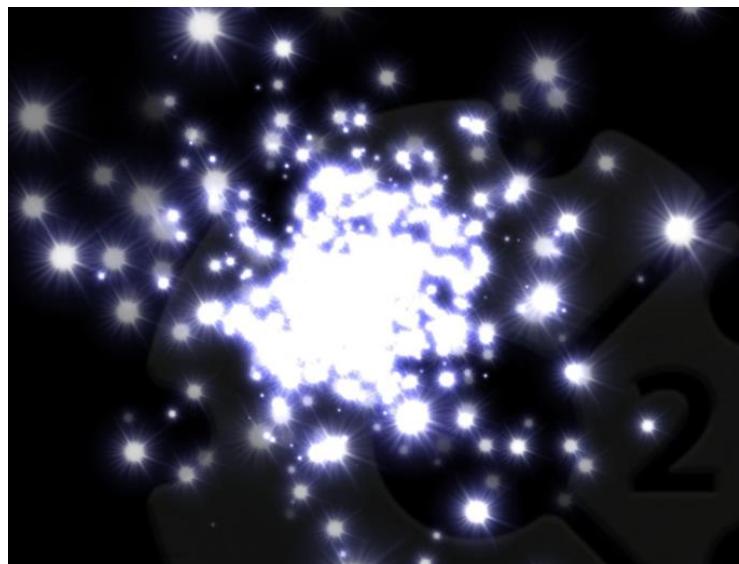
- Статични глобални елементи – под това се има предвид например градове, сгради, или някакъв друг вид структура – неща които просто ги има, не можеш да



Фигура 2.1.8. Структура в играта WoW ги счупиш или да ги сглобиш/построиш (фигура 2.1.8.).

- Динамични игрови обекти – под динамични игрови обекти се разбираят играчите, животните в една игра, всичко което се променя – в змия това са самата змия и ябълката.
- Event/Messaging system – под това се разбира система за известявания. Например при сблъсък между два обекта, се известява потребителя, или се случва някакво следствие – смъртта на героя, победа в играта или друго.

Визуални ефекти – визуалните ефекти се правят чрез множество малки спрайтове, 3D модели и други графични обекти. Основните феномени, които се правят чрез



Фигура 2.1.9. Визуален ефект на construct 2 game engine

визуални ефекти, са огън, експлозии, пушек, течаща вода, искри, падащи листа, облаци, прах, метеоритни опашки, звезди, галактики. Но визуалните ефекти не са само природни явления – те са също и абстрактни явления, като магии, остатъчни светлини и други. Инструменти за създаване на визуални ефекти са Cinema 4D, Lightwave и Houdini.

Възможност за много играчи – тази категория от game engine се разделя на 3 подчасти:

- Уеднаквяване на игралната Вселена – уеднаквяването на игралната Вселена става чрез качване на всички данни на един общ сървър, и от него клиентите взимат данните. Това е нужно, за да може коректно да се извикват функциите или

методите (в зависимост от имплементацията) при интеракция на два или повече спрайта (обекта).

- Персонализиране на обектите – всеки играч притежава дадени обекти.

Това е нужно, за да не може един играч да контролира обектите на друг играч без неговото позволение.

- Свързване на играчи по някаква точкова система (свързвани на играчи с близки стойности на точките) – това е нужно, за да може в игра, където хората са един срещу друг, те да са по-равностойни.

Аудио плейър – аудиото е толкова важно в една игра, колкото и графиката. Но за жалост получава много по-малко внимание от графиката, затова и има големи вариации в това колко е развит аудио плейъра на различните game engine. За DirectX платформите, Microsoft използват XACT аудио инструменти, а SCEA – Scream 3D аудио engine<sup>[26]</sup>.

Контролер – чрез контролера, потребителя комуникира с game engine и конзолата. Той е физическо устройство, което праща данни в някакъв вид, способен да бъде интерпретиран от game engine. Контролера освен физическо устройство, може и да е някакъв интерфейс, софтуерна имплементация на контролите<sup>[26]</sup>.

Сблъскаване и физика на обектите – това е физическият engine на игровия такъв.

Тази категория се разделя на три части:

- Сили и лимити – в тази част на game engine, програмиста трябва да дефинира константи като силата с която гравитацията влияе на обектите, какви други сили да влияят (магнетизъм, триене), както и ограничения (колко силно да влияят тези сили).

- Фантоми и обекти, които не могат да бъдат бълснати – game engine трябва да поддържа обекти, които никога не могат да бъдат преместени – планини, камъни, някои дървета и структури. Другите обекти, които трябва да бъдат поддържани, са тези, които могат да преминават през всичко, без да бъдат ефектирани. Чрез тях се

правят камера ъгли, които да могат да видят целия игрови свят и да се използват за дебъгване.

- Обекти които могат да бъдат бълснати – всички обекти, които могат да бъдат бълснати, могат да бъдат вместени в един голям масив, и чрез техните координати може да се проследи кой обект в кой се е бълснал.

Ресурси – делят се на пет части:

- 3D модели – това са моделите, които game engine рендерва в игровата Вселена.

Техните параметри могат да се описват в някакъв файл, който game engine може да интерпретира и рендерне. Това най-често става в XML формат.

- Текстури – текстурите са елементите, с които покриваме моделите. Текстурата може да е цвят, но също може и да е материал – дърво, коса, мрамор и други.

- Шрифтове – шрифтовете представляват форматиране на текста, който ще бъде рендернат от game engine.

- Физически параметри – физическите параметри са ускорението на земното притегляне (при работа с гравитация), коефициента на триене и други

- Игровата Вселена – в този ресурс са записани данните за игровата Вселена – къде какво има.

Основни системи

- Модули за стартиране и изключване на game engine

- Тестване на модулите на game engine – чрез тестването на модули, програмиста може да разбере коя част от game engine не работи коректно

- Заделяне на памет – чрез ефективно заделяне на памет, game engine може да работи бързо и без да е много ресурсоемка

- Принтиране на дебъг информация – чрез информация за дебъгване, програмиста може да разбере по-лесно къде се е счупил кода

- Парсери – чрез парсерите, game engine може да извлича данни от различни структури от данни за своите модели, текстури и шрифтове

- Асинхронно четене на данни – заедно с парсерите, извличат данни от файлове

- Четене на данни от SD карта – за съхранение на настоящо състояние

Основни системи, предоставени от операционната система

- Транспорт на данни по мрежата чрез UDP и TCP – чрез тези протоколи, game engine може да се синхронизира със сървъра, и така да поддържа множество играчи.

- Библиотека за работа с нишки – чрез библиотеката за работа с нишки на операционната система, game engine може много по-ефективно да използва хардуерните ресурси

## 2.2. Аргументация за избор на програмна среда

Конзолата е базирана върху Raspberry PI 2 Model B, което има четири ядрен 900MHz процесор, с 1GB RAM памет и 3.5mm аудио/видео жак. От тази кратка хардуерна характеристика на микроконтролера се вижда, че той е достатъчно мощен да може да върви операционна система на него. Операционната система е Raspbian, която е базирана върху Debian, което и дава богат набор от вградени функционалности. А върху Raspbian ние пишем на Python. Чрез Python ние можем лесно да достъпим всички функционалности на операционна система чрез вградените пакети.

В: Какво е Python?

О: Python е обектно-ориентиран, бърз и лесен език, с който може да се направи всичко. Той е general-purpose, тоест няма ясно ограничена област, в която да се ползва (за разлика от примерно MatLab, който е изчислителен софтуер. Чрез него не можем да управляваме микроконтролер).

Python предлага преизползване на код в така наречените „модули“ (това е еквивалента на gems в Ruby и „библиотеки“ в PHP).

### **2.2.1. Модул за работа със сокети**

Този модул ни дава достъп до ниско ниво мрежово програмиране, а по-точно – сокетите. Сокетът е единия край на двупосочна връзка между две програми в мрежата. Той е обвързан с порт, за да може да се предават данни през TCP или UDP протокола. Този модул ни предоставя следните елементи:

#### **Изключения**

socket.error – това е изключение, вдигнато от грешка при адресация. Примерно при използване на функцията gethostbyaddr('IP символен низ'), то ако не се намери host с това IP, ще се вдигне това изключение. Друг пример е при използване на функцията gethostbyname\_ex(hostname) и няма наличен такъв host.

socket.timeout – Това е изключение, което се връща когато на сокета му изтече времето за живот, зададено от програмиста. Задаването става чрез функцията settimeout() или setdefaulttimeout(), и символния низ който връща е винаги „timed out“

#### **Константи**

Константи от типа AF\_\* се използват за адресното семейство. AF (address family) може да бъде един от следните 3 вида - \_UNIX, \_INET, \_INET6. \_UNIX е адресно семейство, което основно се използва за между процесна комуникация на една машина. Чрез \_INET адресното семейство се комуникира между различни мрежи по IPv4, а чрез \_INET6 – по IPv6.

Константи от типа SOCK\_\* се използват за самия тип на сокета. SOCK\_STREAM се говори за стрийм от сокети. Той използва TCP протокола, което му дава надеждност, последователност на данните и минимален риск от корумпиране на данните. Но това идва с цена – по-бавно пренасяне на данните поради голям overhead, и по-бавно възстановяване при загуба на пакет.

SOCK\_DGRAM работи по UDP протокола, което го прави удобен за приложения от типа на VoIP или гейминг протоколи. Данните се пренасят възможно най-бързо по мрежата, с никакви вградени начини за препращане на данните, без да се знае дали връзката е приключила, или не. SOCK\_RAW е за пращане на чисти данни, пращат се символни низове на дадения порт. Това използва telnet например, и се използва за прости приложения и дебъгване. SOMAXCONN е константа, която казва на сокета колко да е дълга опашката от чакащи да предадат данни през него.

## Функции

Функциите предоставени от този модул ни предоставят лесен и безболезнен начин за работа със сокети.

`socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)` - създава сокет със специфицираното адресно семейство, тип на сокета, протоколния номер и номера на файловия дескриптор. По подразбиране адресното семейство е AF\_INET, а типа на сокета – SOCK\_STREAM, което предполага ползване на TCP протокола. Аргумента proto се използва при адресно семейство AF\_CAN, и протокола е или CAN\_RAW или CAN\_BCM. Ако аргумента fileno е специфициран, то всички други аргументи се пропускат, и се създава сокет със специфицирания файлов дескриптор

`socket.create_connection(address[,timeout[, source_address]])` –

Прави връзка с TCP service, който слуша адреса, който е даден(адреса е тупъл от двойката хост, порт). Ако хост не е от цифри, то ще се търсят всички адреси с този hostname през AF\_INET и AF\_INET6, и след това ще се пробва да направи връзка с тях, докато успее. Може да се зададе аргумента timeout, който ще даде параметъра timeout на сокета. Ако се даде аргумента source\_address, то параметъра на сокета source\_address ще е равен на този аргумент. Така можем да кажем откъде идва този сокет. Ако е празно, операционната система ще се погрижи за този параметър. Source\_address е във формата на тупъл (хост, порт).

socket.bind(adress) – Присвоява на сокета данни за това откъде да слуша/праща данни. Аргумента зависи от това какво адресно семейство сме избрали за сокета

socket.listen(int) – Чрез тази функция ние правим сокета да работи в режим на сървър. Аргумента казва с колко връзки ще се свърже дадения сокет.

socket.close() - Затваря сокета и освобождава ресурсите, заети от него (Порт или файлов дескриптор)

socket.detach() - Затваря сокета и връща файловия дескриптор.

socket.recv(bufsize[, flags]) - Приема данни от другия край на връзката, bufsize на брой.

socket.send(bytes[,flags]) - Праща данни към другия край на връзката

socket.sendfile(file, offset=0, count=None) – file е файлов обект, отворен в двоичен режим, offset е от кой байт да започнем да четем, count е колко байта да пратим. Нормално ще чете до EOF. При липса на функция на операционната система os.sendfile, то се ползва socket.send(). Сокета трябва да е в режим на предаване на стриймове (SOCK\_STREAM). socket.accept() -

прави връзка между два сокета. Сокета върху който е викнат трябва вече да е инициализиран чрез `socket.create()` и `socket.bind()`. Връща 2 аргумента – единия е нов сокет с адрес автоматично инициализиран от kernela на операционната система, а другия аргумент – адреса на другия край на връзката (сокета с който „говорим“). Чрез `socket.accept()` и `socket.listen()` ние можем да направим един елементарен сървър(фигура 2.2.1.1. и 2.2.1.2.).

```
import socket
import sys

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_adress = ('localhost', 12345)
sock.bind(server_adress)
sock.listen(1)

while True:
    connection, adress = sock.accept()
    try:
        while True:
            data = connection.recv(16).decode()

            if data:
                print(data)
                print("Sending back data")
                connection.sendall(data.encode())

            else:
                break
    finally:
        connection.close()
```

Фиг. 2.2.1.1. Сървър част

```
import socket
import sys
import time
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_adress = ('localhost', 12345)
sock.connect(server_adress)

try:
    message = b'This is a test code. Lorem Ipsum'
    sock.send(message)
    data = sock.recv(16).decode()
    if data:
        print(data)

finally:
    sock.close()
```

Фиг. 2.2.1.2. Клиентска част

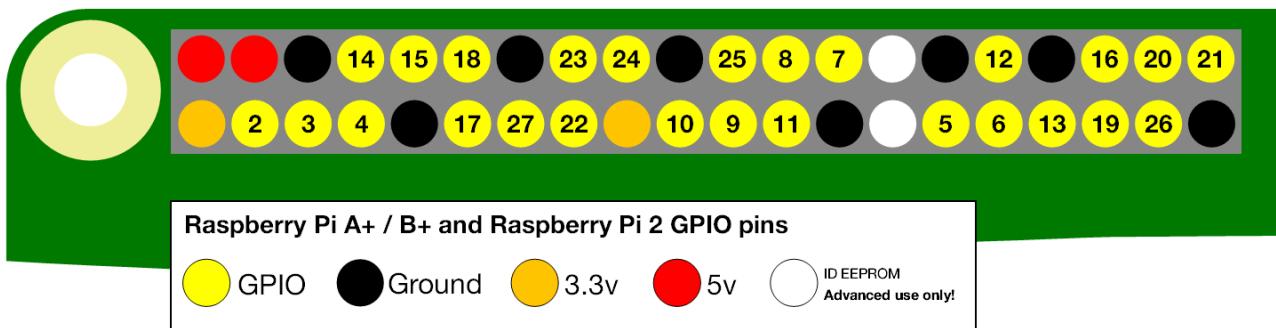
## 2.2.2. Модул за работа с операционната система os

Този модул ни дава достъп до операционната система – работа с файлове, достъп до командния интерпретатор, синхронизация между нишки или работа със системни функции като `urandom(n)`, връщаща символен низ с големина n байта за криптографски цели.

Тази библиотека осидурява достъп до командния интерпретатор на операционната система, с цел дебъгване при нужда или стартиране на друг скрипт от нашия скрипт, например за пускане на игра от основния скрипт. Функцията която прави тази операция е `system` на модула, и се вика с параметър символен низ,

който изпълнява ролята на команда в интерпретатора.

### 2.2.3. Модул за работа с входно/изходните пинове GPIO



Фиг. 2.2.3.1. GPIO пиновете на RPI 2

GPIO пиновете (general purpose input output) се използват за комуникация между Raspberry PI 2 и хардуера, прикачен към него. Чрез тях може да се сложат допълнителни модули:

- NFC (near field communication) четец, позволяващ работа с NFC устройства, -

Друг начин за ползване на тези пинове е за захранване на някакъв мотор

(примерно на количка или вертолет/куадкоптер),

-светване на светодиод който да индицира текущото състояние на системата,

- Управляване сензори – температура, вибрация, влажност, осветеност, налягане и изключване на системата в случай на прегряване с цел запазване целостта и, както и допълнителна интерактивност с потребителя.

#### Функции предоставени от модула

GPIO.setmode(MACRO) – аргументите на тази функция могат да бъдат или GPIO.BOARD или GPIO.BCM. Това показва по какъв начин ще броим пиновете – ако използваме GPIO.BOARD като аргумент, ние ги броим спрямо самата платка – тоест 1,2,3,4 от ляво надясно от горе надолу, а ако използваме GPIO.BCM, ги броим по номерата им, описани по-горе.

`GPIO.setup([pins], MACRO, initial=None)` – тази функция настройва посочените пинове като изход или вход. Ако не е даден лист, а само една стойност, то тя ще бъде настроена. MACRO може да бъде или `GPIO.IN`, или `GPIO.OUT` (респективно за вход и изход). Initial е незадължителен аргумент, който посочва началното състояние на пина, ако е в режим на изход. Може да е `GPIO.HIGH` или `GPIO.LOW`. Ако е в режим на четене, можем софтуерно да му сложим `pull_up` или `pull_down` резистор. Това става чрез `pull_up_down = GPIO.PUD_UP` `pull_up` резистор или `GPIO.PUD_DOWN` за `pull_down` резистор.

`GPIO.cleanup(pin = None)` – Изчиства пиновете. Препоръчително е винаги да се ползва винаги, за да не си повредим по някакъв начин RPI или най-малкото да не ни дава предупреждения интерпретатора при излизане от програмата за некоректност. Аргумента който подава на функцията може да изчисти само посочения пин. Ако не го дадем, изчиства всички пинове. Чрез подаване на лист от пинове, може да изчистим повече от един наведнъж.

`GPIO.wait_for_edge(pin, MACRO, timeout=None)` – Функцията блокира изпълнението на скрипта докато не се случи или изтичането на времето, зададено от `timeout`, или слушаване на някакъв евент. Ако `timeout` не е зададен, то ще се чака до появяване на евента. Какъв ще е този евент зависи от MACRO – ако `MACRO = GPIO.RISING` то когато стойността от този пин стане от 0 на 1 ще се продължи изпълнението. Аналогично при `GPIO.FALLING` при смяна на състоянието от 1 на 0, а при `GPIO.BOTH` – при каквато идея смяна на състоянието.

`GPIO.add_event_detect(pin, MACRO)` – MACRO може да бъде `GPIO.` [`RISING`, `FALLING`, `BOTH`], и аналогично като при `wait_for_edge` казва кога ще се случи евента. Разликата е че при слушаване на евента, `wait_for_edge` продължава нормалното изпълнение на програмата, докато `add_event_detect` създава нишка, която изпълнява `callback` функциите, зададени за този пин от `GPIO.add_event_callback()`. Това го прави полезен при асинхронно изпълнение на

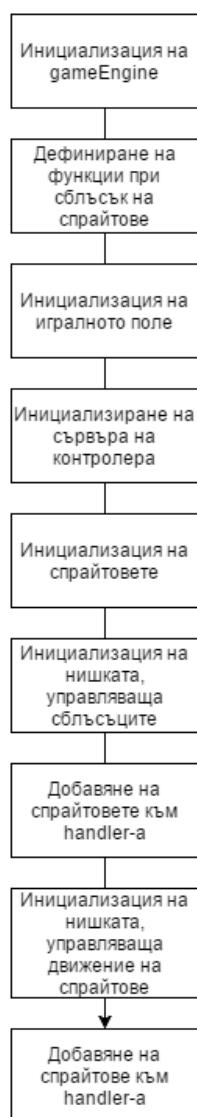
скрипта (той върши работа, и при натискане на бутона му обръща внимание, а не да чака за натискането му).

`GPIO.add_event_callback(пин, function bouncetime = None)` – Function е функцията, която да изпълним при засичане на евент на пин, като този евент се вика от функцията `GPIO.add_event_detect`. При зададен аргумент `bouncetime`, функцията ще бъде извикана след `bouncetime` милисекунди. Това е за да се предотврати софтуерно така нареченото „подскачане“ на бутоните.

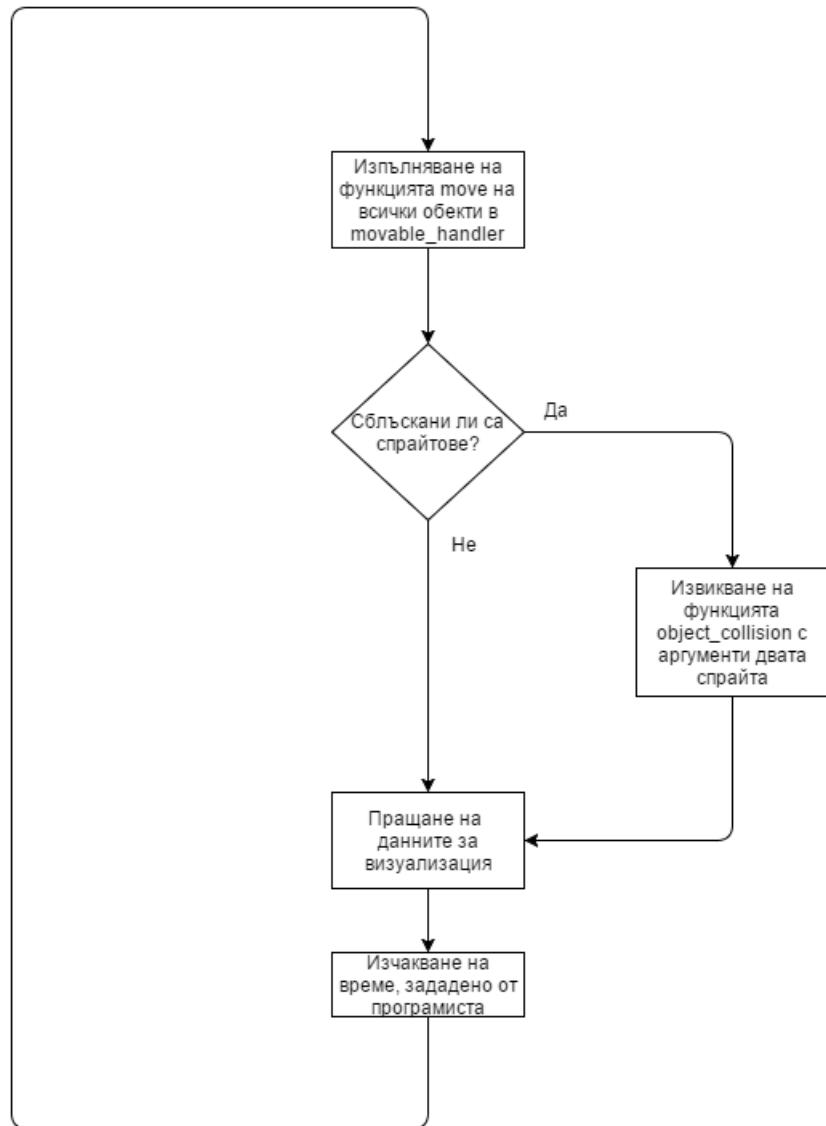
`GPIO.remove_event_detect(пин)` – Премахва следенето за евент на този пин.

# 3. Трета Глава – блокови схеми на управляващия софтуер. Софтуерен модел на игрите. Описание на библиотеката за проектиране на игри. Описание на ROM-овете.

## 3.1. Блокови схеми на управляващия софтуер



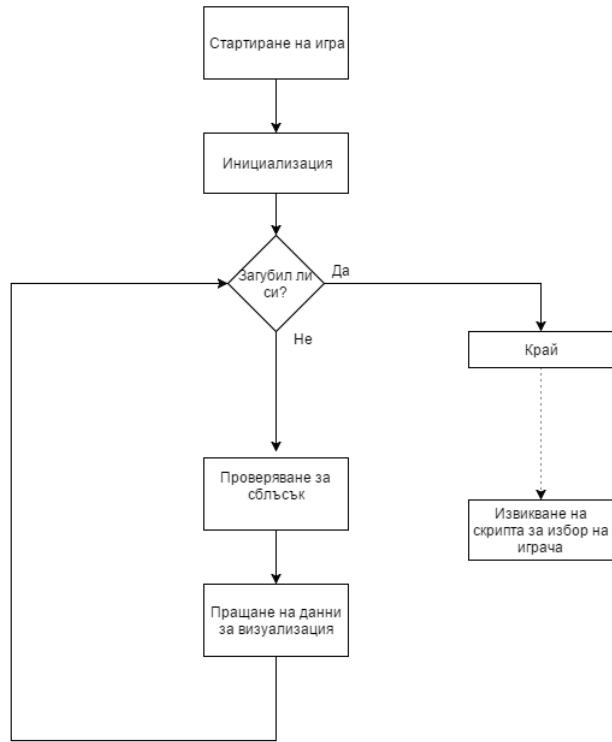
Фигура 3.1.1.  
Инициализация



Фигура 3.1.2. Блокова схема на сблъсъка на спрайтове



*Фигура 3.1.3. Пращане на данните за визуализация*



*Фигура 3.1.4. Основен Workflow на game engine*

## 3.2. Софтуерен модел на игрите

Проектирането на игри се изразява в това да се направи пристапа библиотека за помагане на програмистите в писането на игри за конзолата. Тази библиотека, както и самите скриптове на игрите, ще са написани на Python. Библиотеката трябва да поддържа базова работа със спрайтове (движение, интеракция, проектиране), както и базова работа с някакъв вид игрово поле. Други полезни функции на тази библиотека ще е слагане на спрайта на произволно място на полето, без да се засече с друг спрайт, обръщане на спрайта наляво или надясно, слагане на спрайт,

на който една от страните му е залепена за край на игралното поле (игралното поле е двумерен(или тримерен) масив), с което може да се създадат врати или тръби. Модела на игрите е те да са с прост дизайн, с цел по-лесно и адекватно представяне на „алтернативни“ визуализиращи устройства като 2D матрица или 3D куб, създадени от фотодиоди. Поради тази причина, на най-ниско ниво всичко се опира до работа с масиви, докато накрая цялата информация от игрите се състои в това една 2D или 3D матрица, която се подава на визуализиращ контролер, чрез който да управляеме визуализиращото устройство. Самата логика на игрите зависи от самия програмист, който създава играта – тя може да е проста като змия, или тетрис, но може и да се сложна като DnD или Dungeon Crawler. Чрез нашата библиотека ние помагаме на програмиста да пише игрите чрез предефинирани класове и функции, които да му помогнат в това начинание. При разработка на игри с нашата конзола, програмиста (или любителя) трябва първоначално да си избере дали да ползва 2D или 3D игрална среда. Това зависи основно от това каква игра иска да направи – ако иска да направи приключенска игра, то 2D би му свършило повече работа от 3D (така наречения top-down view universe, където играча вижда игровата вселена от горе). След това идват въпросите за самата логика на играта – ще има ли стрелба, какво се случва при сблъсък на 2 обекта, какви са тези обекти, организацията на вселената.

Дефиниране на вселена става по представения долу начин – в момента

обекта pg се състои от обект, кой в

```
pg = gameEngine.PlayGround2D()
```

Фиг. 3.2.1. Създаване на двуизмерно  
игрално поле 8x8

същността си е двумерен масив. Като

аргументи може да се подаде колко голям  
да е този масив. Като аргумент може и да

се подаде с какво да е инициализиран този масив, като по подразбиране това са нули.

След това идва дефинирането на спрайтове. Дефинирането им става чрез използването на двумерен масив, като всеки подмасив съдържа 3 стойности –

```
mylist = [
    #y,x,val[x][y]
    [0,0,8],
    [0,1,8],
    [0,2,8]
]
```

Фиг 3.2.2. Създаване на спрайт, с 3 координати и стойности на координатите 8

Y координатата, X координатата и стойността при тези координати. Ако използваме 3D поле за игра, то към масивите се добавя и 4ти аргумент, който се явява Z координатата. Също така трябва и да декларираме какъв е

спрайта като тип (enemy, friend, wall, tile) с цел при сблъскване с него, да разберем какво да правим. Но спрайта, даден ни по подразбиране не е особено полезен. На него трябва да бъде дефинирано функцията move(). Функцията move() е функция, която извикваме на всеки n секунди, специфицирано от програмиста.

Това се постига чрез използването на нишка за викането на тази функция, и след това използването на time.sleep(n) в безкраен цикъл. При създаването си, всички обекти на които трябва да се вика тази функция се слагат в листа от обекти object\_movable\_list. Програмиста също трябва да дефинира и как се мърда обекта

```
def move_all_objects(object_movable_list):
    for _object in object_movable_list:
        _object.move()
```

Фиг. 3.2.3. Реализация на функцията move\_all\_objects

при натискане на бутоните на клавиатурата. В нормалната библиотека са дефинирани мърдането на спрайтовете по

подразбиране (при натискане наляво, целия спрайт се мърда наляво). Ако се иска различно от това поведение, то програмиста трябва да предефинира функциите.

Това става по следния начин – създаване на функциите, с които искаме да променим държанието по подразбиране. След това присвояваме функциите на функциите, които искаме да сменим. По подразбиране има само 4 функции, запазени за движение, но програмиста може да дефинира колкото иска (само да бъде адекватно спрямо броя бутона/ управляващи единици). Управлението на натискането на бутоните става чрез нишка, която следи какво е натиснато и вика

функцията, дефинирана от програмиста. показаната имплементация работи с вход от стандартния такъв, с цел дебъгване на програмата. След дефиниране на функцията move() на спрайта, както и на функциите свързани с управление от потребителя, трябва да се дефинира какво се случва при сблъскване 2 спрайта. Това се случва по тази последователност – първо се вижда дали спрайтовете са се докоснали, и ако да – какъв им е типа. Спрямо типът, програмиста може да дефинира какво се случва, и да го приложи. Чрез различните стойности на полето, програмиста може да мени в какъв цвят ще светят дадените обекти. Примерно ако полето е със стойност X тя ще отговаря на някакъв цвят. Пращането на данни става чрез командата send() на игралното поле. Визуализацията на игралното поле става чрез командата draw().

## **3.3. Описание на библиотеката за проектиране на игри**

Целта на библиотеката е да подпомогне програмиста в писането на игра за нашата конзола. Библиотеката съдържа някои базови функции, както и основни класове заедно с техните методи.

### **3.3.1. Описание на функциите**

#### **Функции**

Функциите дефинирани в модула са няколко:

`get_random_number(min, max)` – връща произволно целоочислено число между `min`(включително) и `max`(изключено).

`up_button_pressed(sprite)` – вика функция `goUp` на класа `sprite`. Това е по подразбиране, и може лесно и е желателно да се промени. Аналогично другите 3 функции са `left_button_pressed`, `right_button_pressed` и `down_button_pressed`.

#### **Класове**

Класовете са по-голямата част от библиотеката. Класовете които са включени в библиотеката са:

## **play\_ground\_2D**

### **Методи**

Конструктор - `__init__(self, y=8, x=8, default = 0)` – създава двумерен масив със стойности `default` и големина по ордината `y` и `x` по абсцисата.

`draw()` - не приема аргументи. Праща цялото поле на контролера, контролиращ рисуването.

`draw_centered(spriteRevolver, visionX=8, visionY=8)` – Тази функция рисува спрайта `spriteRevolver` в центъра на поле с големина `visionX` и `visionY`. Това поле се явява част от цялото игрално поле. Ако не сме близо до стена, игралното поле е с отстояние `visionX/2` от ляво и отдясно на спрайта, и `visionY/2` отстояние отгоре и отдолу. Ако стигнем до някоя стена, противоположната страна ще го компенсира (ако се намираме в горен ляв ъгъл, ще има 8 единици надясно и надолу и 0 нагоре и наляво). Праща към контролера игрално поле с големина `visionX + дебелината на спрайта (най-десния индекс – май-левия индекс)` и `visionY + височина на спрайта (най-долния индекс – най-горния индекс)`.

`clear_this(Sprite)` – изтрива `sprite` от играта. Това се случва като се изтрие `Sprite.shape` от игралното поле и се премахне от `object_bumpable_list` и от `object_movable_list`.

`clear()` - изтрива цялото игрално поле. Вика отново конструктора.

`draw_sprite_on_random_wall(Sprite)` – чертае спрайта на произволно място, като една част от спрайта е закачен за стена. Полезно е при искането да се създаде стена или врата.

## **Член-променливи**

Класът съдържа 3 член променливи – size\_of\_playfield, default и playfield. Първата е масив който съдържа 2 стойности – големината на игралното поле по X и по Y. Default е целочислено число, което съдържа стойността на всяка една клетка на полето. Playfield е двумерен масив, който съдържа самото поле.

## **movable\_handler()**

Този клас се грижи за мърдането на всички обекти, които трябва да се мърдат (movable).

### **Методи**

конструктор – инициализира се с празен масив за съдържанието на всички movable обекти, както и с инициализиране на родителския клас threading.Thread, тъй като ще ползваме този клас като нишка.

add\_object – добавя обект към movable handler.

delete\_object – изтрива обект от movable handler.

run – на всеки n секунди вика функцията move на всички обекти в него.

### **Член-променливи**

data – масив от обекти, на които трябва да се викне функцията move.

pg – игралното поле, на което са разположени всички спрайтове.

sleep\_timer – таймерът за това колко често да се вика нишката.

## **bumpable\_handler()**

Логиката на този клас е същата като на movable\_handler(), обаче вместо да държи всички обекти които се мърдат, той държи всички обекти, с които можем да се сблъскаме.

### **Методи**

Методите, както и член-променливите са аналогични на горния клас.

add\_object, delete\_object – манипулация с масива от обекти.

run – праща данни на контролера, отговорен за визуализация на екрана на всеки n секунди.

compare\_objects – при сблъскване на два обекта, те отиват в тази функция и биват манипулирани, както е зададено от програмиста.

### **Член-променливи**

Променливите са аналогични на горния клас.

data - масив от всички обекти, които трябва да бъдат материални и да може да се сблъскват.

pg – игралното поле, което ще се ползва от всички спрайтове.

timer – времето, през което данните ще се пращат за визуализация.

## **sprite()**

Спрайта е основата на цялата система. То представлява всичко което потребителя на конзолата ще види. Всичко ще се извършва посредством тях – мърдането, бълскането, всичко ще използва спрайтовете.

## Методи

Конструктор – (self, matrix = [[0,0,5]]) – приема матрица, която да е формата на спрайта. Масив от масиви му се подава, като всеки масив характеризира една точка с координати Y,X и стойността на точката.

get\_left\_most\_part\_X() - връща най-левата точка на спрайта.

get\_right\_most\_part\_X() - връща най-дясната точка на спрайта.

get\_down\_most\_part\_Y() - връща най-долната точка на спрайта.

get\_top\_most\_part\_Y() - връща най-горната точка на спрайта.

draw(pg, x=0, y=0) – рисува спрайта върху игралното поле. Ако са подадени аргументите x и y, то те ще са явят отстоянието от оригиналното положение на спрайта, дефинирано от shape. Аргументите x и y автоматично се прибавят към всички части на спрайта.

go\_left(pg, speed=1) – мърда целия спрайт в посока наляво (делта y = 0, делта x = -speed). Аргумента speed се използва за разбиране на това с колко полета да се премести спрайта. Ако се бутнем в стена, то ние не можем да продължим. Извиква функцията draw() на игралното поле (pg)

go\_right(pg, speed=1) - мърда целия спрайт в посока надясно (делта y = 0, делта x = speed). Аргумента speed се използва за разбиране на това с колко полета да се премести спрайта. Ако се бутнем в стена, то ние не можем да продължим. Извиква функцията draw() на игралното поле (pg)

go\_top(pg, speed=1) – мърда целия спрайт в посока наляво (делта y = -speed, делта x = 0). Аргумента speed се използва за разбиране на това с колко полета да се премести спрайта. Ако се бутнем в стена, то ние не можем да продължим. Извиква функцията draw() на игралното поле (pg)

go\_down(pg, speed=1) - мърда целия спрайт в посока надясно (делта y = speed, делта x = 0). Аргумента speed се използва за разбиране на това с

колко полета да се премести спрайта. Ако се бутнем в стена, то ние не можем да продължим. Извиква функцията draw() на игралното поле (pg)

is\_inside(matrix) – проверява дали спрайта се намира в двумерната матрица, която сме подали. Тази матрица може да бъде shape на друг обект, и така да разберем дали 2 спрайта се сблъскват. Ако се намира там, is\_inside връща 1. Ако не се засичат, връща 0.

move() - дефинира функцията move на спрайта, която обаче не прави нищо. Тя трябва да бъде предефинирана от програмиста. Всички обекти които трябва да се мърдат без вход от потребителя, трябва да имат тази функция, за да може movable\_handler-а да го ползва коректно.

put\_at\_random\_place(pg) – поставя спрайта на произволно място по игралното поле, което не е заето от друг спрайт.

flip(pg, direction = 0) – обръща спрайта на 90 градуса. Ако се бълска в стена, то спрайта не се обръща. Случва се чрез викане последователно на функцията find\_flip\_point, която връща най-долната лява точка на спрайта, и след това я подава като аргумент на функцията get\_new\_array, който преобръща целия спрайт, и проверява дали не се бълска в някоя стена. Ако го прави, то функцията връща 1. Ако не се бълска в нито една стена или друг спрайт, то функцията връща новия масив, с който да е заменен спрайта. Ако direction е равен на 1, то завъртането става на 270 градуса.

find\_flip\_point() - помощна функция за функцията flip. Чрез нея се намира най-долната лява точка.

get\_new\_array(self, pg) – помощна функция за функцията flip. Чрез нея се взима нов масив, който се предава като новата форма на спрайта. Ако масива е негоден, то тя връща 1.

### **Член-променливи**

shape – масив от масиви, чрез който се описва формата на спрайта, както и неговите цветоте. Масивите са във формата на Y, X, стойност на клетка [x][y].

type – определя типа на спрайта. Използва се при определяне на действието, което трябва да се предприеме при удар между два спрайта

### **catch\_keyboard\_input()**

Този клас има за цел да даде на програмиста алтернатива вместо да ползва контролер, да ползва клавиатурата си за вход на данни.

#### **Методи**

Класът има само един метод – run, тъй като се изпълнява като нишка. В този метод, класът чете за вход от потребителя, и при натискане на специално предефиниран бутон се изпълнява предефинирана функция. Чрез тази функция може да се дебъгва и тества програма ефективно и лесно.

### **Член-променливи**

Класът няма член-променливи

### **controller\_input**

Класът има за цел да прочита информация от сокет, и да я обработва. Това е входа от потребителя. Конзолата чете от специфичен сокет.

#### **Методи**

Конструктор() - не приема аргументи. По подразбиране слуша за конекции на localhost:12345

check\_if\_here(addr) – проверява дали човека, направил конекцията току-що го има в масива от масиви [“адрес“, “име“]. Ако го има, не прави нищо. Ако го няма, го записва и му дава име.

return\_user\_by\_addr(addr) – връща името, на което отговаря този адрес.

call\_function\_with\_player(data, player) – функция, която трябва да бъде предефинирана от програмиста. В нея се получават данните от комуникацията с контролера. Player е играча, от който са получени данните. Така можем да поддържаме много играчи.

### **Член-променливи**

Член-променливите на хендъръра на клиентите са две.

data – тази променлива представлява лист от листове, като във всеки лист има двойка адрес и име. Името се взима от масива names, и се слага на всеки уникален адрес.

names – масив от имена, с които да диференцираме различните играчи. По подразбиране се състои от стринговете „player1“, „player2“, „player3“, „player4“.

### **server**

Този клас приема от сокет на порт 12345 данните от контролерите. След това ги праща на класът controller\_input.

### **Методи**

Методите на този клас са само два – тъй като се изпълнява в нишка, то той има метода run, и има конструктор като всеки клас

Конструктор (names=[]) - приема като аргументи масива names, чрез който се идентифицират връзките. Чрез тези names програмиста може да разбере кой спрайт да премести или коя функция да извика при натискане на някой бутон. Конструктора също така създава сокет на порт 12345, localhost, както и инициализира нов all\_client\_handler с аргументи масива names.

run() - приема данни от сокета, записва ги в all\_client\_handler, ако за първи път среща този адрес. След това извиква функцията call\_function\_with\_player на all\_client\_handler, с аргументи пратените данни и името на игрacha.

### **Член-променливи**

ch – накратко от client\_handler – това е променлива от клас all\_client\_handler. Ней я използваме за държане на данните.

server\_address – пълния адрес на сървъра. По подразбиране той е localhost:12345.

sock – представлява сокета, от който четем данните, пратени от контролера. Той работи на адресно ниво Ipv4 и на протокол UDP.

### 3.4. Описание на ROM-овете

В.: Каква е целта на тези ROM-ове?

О.: Абстракция на хардуера. Чрез тях ние ще можем да ползваме всякакъв контролер, само като напишем един json и C-код, който ще бъде даден с цел пълно улеснение на потребителя, който иска да ползва наличните материали, а не да ползва статичен хардуер, който ще му се наложи да закупи.

```
json: {  
    Has: {  
        buttons: 4,  
        gyroscope: 1  
    },  
    Utility: {  
        pause: 10,  
        menu: 11  
    },  
    Groups: {  
        A: {  
            number_of_states: 4,  
            type: button,  
            readpins: {  
                1: None  
                2: None  
                3: None  
                4: None  
            }  
        },  
        D: {  
            number_of_states: 4,  
            type: gyroscope,  
            readpins: {  
                direction: 12,  
                power: 13  
            },  
            function: {  
                0: None  
                90: None  
                180: None  
                270: None  
            }  
        }  
    }  
}
```

Фиг. 3.4.1. Примерна абстракция

Контролера ще съдържа този JSON файл, който ще е абстракцията на хардуера, наличен на този контролер. При започване на връзката между контролера и конзолата, този JSON ще бъде променен от конзолата за да може точно да опише пращането на данни.

На фигура 3.4.1 се вижда примерна реализация на този JSON за контролер, който има 2 бутона за меню и пауза, 4 бутона за интеракция с конзолата и жироскоп. Всички данни се намират в json, който има поле Has, Utility, Groups

**HAS** – в това поле е написано какво има нашия контролер откъм хардуер.

**Utility** – полето Utility има за цел да държи това

къде се намират бутон за пауза, и бутона за меню. Съдържа 2 полета – едното е

Pause, което при натискане праща на конзолата команда, че искаме да спрем временно играта, а също и menu – което ни казва да прекратим напълно играта.

**Groups** – Groups е най-важната част от този JSON. В него се казва изрично при получаване на interrupt от някой pin, какво да правим с него. Дели се на логическата абстракция A и D – те са физически едни и същи, но можем да дефинираме че A ще е някакъв Action, действие, като стрелба или скок. D идва от Движение, и означава че ще преместим нашата фигура в някаква посока.

**Group: A** – Група A се грижи за някакъв Action. При получаване на interrupt при някаква промяна в нивото на напрежение на някой от readpin, то контролера ще изпрати низа, който е като стойност за този ключ. В конзолата се сравнява какво се праща, и оттам се обработва по предназначение

**Group: D** – Група D се грижи за движението на обекта в пространството. Разликата с бутона е в това, че се управлява от жirosкоп, а не от бутони, което прави обработването му по-трудно. Чрез полето function, ние казваме какво да се прати на конзолата, в зависимост от изчисленията, върнати от direction.

Пример: Ако direction е 37 градуса, то той се намира в диапазона [45,-45] градуса, и се укрупнява в 0 градуса. Формулата за диапазона на укрупнение е

$(2u \mp 1) \times (180 \div n)$  , където n са броя сегменти на които е разделен кръга (number\_of\_states полето), а u – номера на полето (от 0 до number\_of\_states-1).

Укропнява се до 360/u като стойност, и ако u е 0, то градусът става 0.

### **3.5. Описание на сървъра**

От сървъра на игровата конзола има възможността както да се теглят, така и да се качват игри, като първото става директно, а второто - през сайта, хостнат върху него. За сървъра е използвано XAMPP developer stack, в който се намира Apache v.2.4.4. и PHP v.5.6.12. Използването на PHP е заради относителната простота на сървъра – той само трябва да дава начин за качване на файлове на сървъра. В бъдеще ще бъде пренаписан на Django, когато започне имплементацията на комуникация между 2 конзоли през сървъра.

Структура на директорията:

- games
- lib
- not\_approved\_games
- denied
- games.csv
- index.html

В директорията games се намират всички игри, които са преценени като добри и готови. Ще се проверява дали не злоупотребяват със системата на потребителя.

В директорията lib се намират всички помощни файлове за сайта – css, js, php, картинки.

В директорията not\_approved\_games се намират всички игра, които очакват да бъдат оценени.

В директорията denied са всички игри, които след разглеждането им, не са били одобрени.

Във файла games.csv се намират имената на всички игри (така, както

потребителя на конзолата ще ги види), както и името на файла на съответната игра (Примерно snake, snake123v235nobugs.py, фиг. 3.5.5.).

Във файла index.html се намира самия HTML код на сайта. На фигура 3.5.1. и фигура 3.5.2. се вижда изглед от сайта, съответно преди и след качването на игра.



Фигура 3.5.1. Изглед преди да се качи игра



Фигура 3.5.2. Изглед след като се качи игра

За качването на игра се използва AJAX технология, написана чрез jQuery.

Чрез jQuery скрипта, файла и името му се праща на php файла „post\_game.php”, откъдето файла се премества от /tmp в not\_approved\_games. Кода за тези скриптове може да се види на фигура 3.5.4. и 3.5.3. съответно за jQuery кода и за PHP кода.

```
if (!empty($_GET['Name'])){  
    $data = $_GET['Name'];  
    $better_token = md5(uniqid(mt_rand(), true));  
    move_uploaded_file($_FILES['file']['tmp_name'], "../../../../not_approved_games/".$_FILES['file']['name'].$better_token.".py");  
    echo json_encode("1");  
}
```

Фигура 3.5.3. PHP код за сървъра

Това по-горе е кода за качване на игра. При приемане на GET HTTP заявка с параметър Name, скрипта започва да обработваме информацията. Чрез променливата better\_token, ние даваме винаги уникално име на файла, за да не се получи изтриване и презаписване на файловете.

```

var name = $('[name='gameName']).val();
var file = $('[name='gameFile']").prop("files")[0];
var extension = file.name.split(".")[1];
var size = file.size;
if (extension != "py"){
    return;
}
if (size > 1000000){
    return;
}

var fd = new FormData();
fd.append('file',file);

var _url = "libs/dynamic/php/post_game.php"?Name=name;
var createPromise = $.ajax({
    url: _url,
    data: fd,
    dataType: "JSON",
    processData: false,
    contentType: false,
    method: "POST",
}).then(function(responce){
    if (responce == "1"){
        $("#forms").fadeOut("slow");
        $("#forms").html("");
        $("#forms").html("<p class='no-top left big'>Game uploaded correctly! Waiting for our approval</p>");
        $("#forms").fadeIn("slow");
    }
}).fail(function(responce){
    console.log(responce);
});

```

Фигура 3.5.4. Javascript код за AJAX заявка към php файл

Кода на фиг. 3.5.4. представлява взимане на файл от input HTML елемент, проверяване му, и след това неговото изпращане. Файловете могат да бъдат до максимум 1Мегабайт, и да са с разширение ru (python). Обхващането на AJAX заявката в wrapper ни осигурява правилната последователност на действията. Така не извикваме then метода преди завършване на заявката.

```

hello world, helloWorld.py
Snake, snakeWithEngine.py
test script, test.py
god, ImetoNaSazdate1q.py

```

Фигура 3.5.5. Съдържанието на файла "games.csv"

## **4. Четвърта глава. Ръководство на потребителя.**

### **4.1. Стартране на игровата конзола**

За стартране на игровата конзола, потребителя първо трябва да конфигурира конзолата да се върже към някоя безжична мрежа. Това може да се направи само веднъж с помоща на USB to TTL сериен кабел. Друг начин за влизане в Raspberry PI 2 е чрез мрежов кабел от порта на конзолата до рутер. От там вече конзолата ни е вързана към интернет, и можем безпроблемно да я достъпим. Името и паролата, ако използваме ssh за достъп са име – pi, и парола – raspberry. Ако работим с windows операционната система, то ще ни е нужно и софтуер за емулиране на терминал, като Putty. Друго нещо което ще ни е нужно при USB to TTL метода е USB-to-Serial Comm Port, с цел да може да се предават сигналите от USB до серийния интерфейс на RPI2.

#### **USB to TTL**

Кабелът който се ползва, трябва да е стандартен USB кабел от едната страна, и 4 женски входа от другата. Четирите входа са със следните цветове – червен, зелен, бял и черен.

Червения се използва за захранване на конзолата. Това обаче няма да е нужно, затова този кабел го игнорираме. Ако потребителя иска да го ползва вместо да използва микро-USB директно в микро-USB порта на конзолата, то той може да използва този кабел. За да го направи, потребителя трябва да постави накрайника на червения кабел върху означения пин.

Зеления кабел е задължителен, и се използва на RXD. RXD означава,

че от него микроконтролера ще чете. Вкарва се в означеният със зелено пин.

Белия кабел е задължителен, и се използва за TXD. TXD означава, че на този порт микроконтролера ще предава информация. Вкарва се в означеният с бяло пин.

Черния кабел е задължителен, и се използва за заземяване на връзката. Задължителен е, дори ако не сме използвали червения кабел за захранване. Вкарва се в означения с черно пин.

За този метод ще са ни нужни специални драйвери за USB to TTL комуникацията.

Те се

### [PL2303 Windows Driver Download](#)

Download File: [PL2303\\_Prolific\\_DriverInstaller\\_v1.12.0.zip](#)

**Windows Driver Installer Setup Program**

**(For PL2303 HXA, XA, HXD, EA, RA, SA, TA, TB versions)**

Installer version & Build date: **1.12.0** (2015-10-07)

Windows XP (32 & 64-bit) WDM WHQL Driver: **v2.1.51.238 (10/22/2013)**

- Windows XP Certified WHQL Driver
- [Windows Certification Report](#)
- Compatible with Windows 2000SP4 & Server2003

Windows Vista/7/8/8.1/10 (32 & 64-bit) WDF WHQL Driver: **v3.6.81.357 (09/04/2015)**

- Windows 10 Certified WHQL Driver
- Windows Vista, 7, 8, 8.1 Certified WHQL Driver
- [Windows Certification Report](#)
- Compatible with Windows Server2008, 2008R2, 2012, 2012R2
- Driver can auto-download via Windows Update (Windows 7, 8, 8.1, 10)

Installer Language Support: English (default), Chinese (Traditional and Simplified), Japanese

For Prolific USB VID\_067B&PID\_2303 and PID\_2304 Only

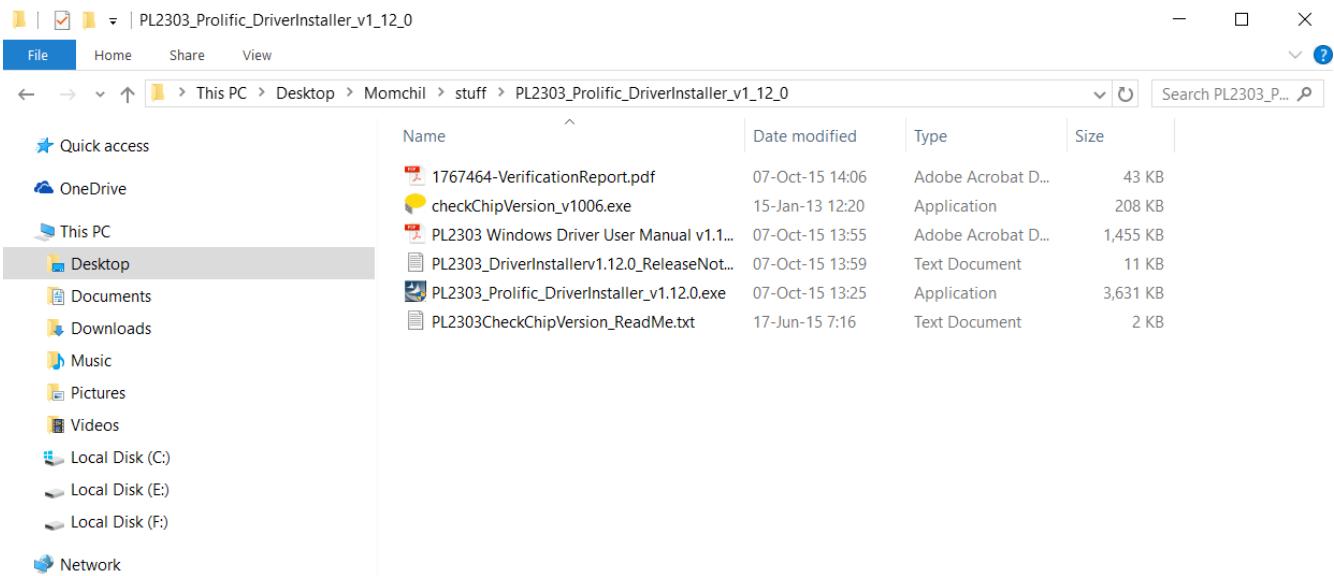
Includes Certification Report, User Manual, Driver Release Notes & CheckChipVersion Tool

Installer supports silent install (add "/s" parameter when running program)

*Фиг. 4.1.1. Изглед на сайта, от който се тегли zip файла*

свалят от [http://www.prolific.com.tw/US>ShowProduct.aspx?p\\_id=225&pcid=41](http://www.prolific.com.tw/US>ShowProduct.aspx?p_id=225&pcid=41) и след това се

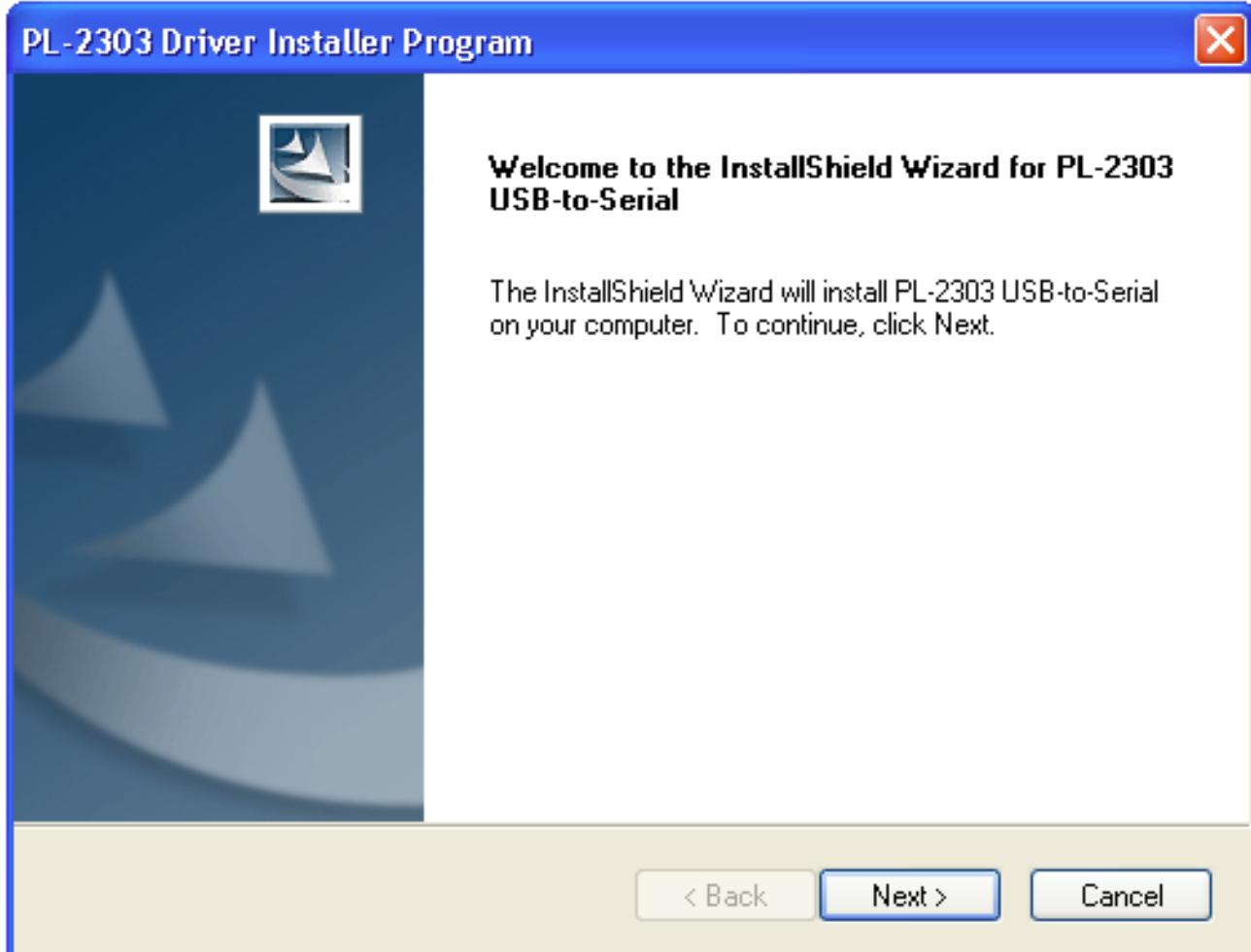
следва инсталатора. При последване на линка, той ще ни отведе до страница със съдържание, подобно на това описано във фигура 4.1.1.



#### 4.1.2. Съдържание на zip файл

При натискане на Download File, на машината ще бъде свален zip файл. При отварянето му, ние ще видим папка със съдържание, като това на фигура 4.1.2. От тук, два пъти натискаме PL2303\_Profilific\_DriverInstaller\_v.1.12.0.exe. Ако операционната система ни каже, че този файл може да повреди компютъра, ние все пак го пускаме. След пускането на wizard-а, следвайте инструкциите.

Изображенията по-долу ще ви помогнат с това.



Фиг. 4.1.3. Стартуране на Wizard-а

## Found New Hardware Wizard

Please wait while the wizard installs the software...



Prolific USB-to-Serial Comm Port



< Back

Next >

Cancel

Фиг. 4.1.4. Инсталация на драйверите

### Found New Hardware Wizard



### Completing the Found New Hardware Wizard

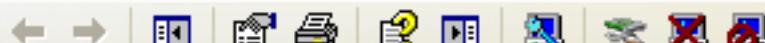
The wizard has finished installing the software for:



Prolific USB-to-Serial Comm Port

### Device Manager

File Action View Help



- SIMON-8E5F5D4EA
  - + Batteries
  - + com0com - serial port emulators
  - + Computer
  - + Disk drives
  - + Display adapters
  - + DVD/CD-ROM drives
  - + Human Interface Devices
  - + IDE ATA/ATAPI controllers
  - + Jungo
  - + Keyboards
  - + Mice and other pointing devices
  - + Monitors
  - + Network adapters
  - Ports (COM & LPT)
    - + Printer Port (LPT1)
    - + Prolific USB-to-Serial Comm Port (COM7)
  - + Sound, video and game controllers
  - + System devices
  - + Universal Serial Bus controllers

Фиг. 4.1.5

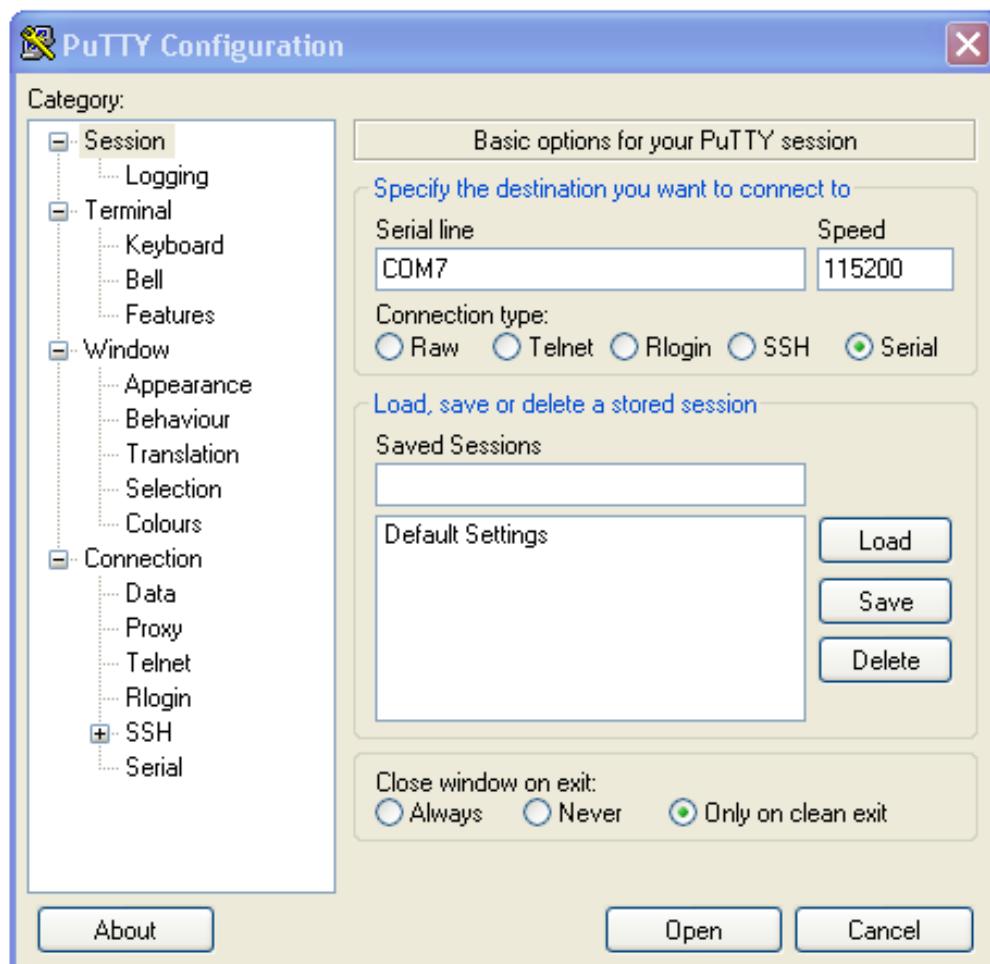
При

Illustration 1: Фиг. 4.1.6. Изглед на DEVICE MANAGER при правилна инсталация

правилна инсталация на драйверите, в DEVICE MANAGER ще се появи с кой COM порт да си говорим.

След разбирането на кой COM порт да комуникираме, стартираме putty.exe приложението.

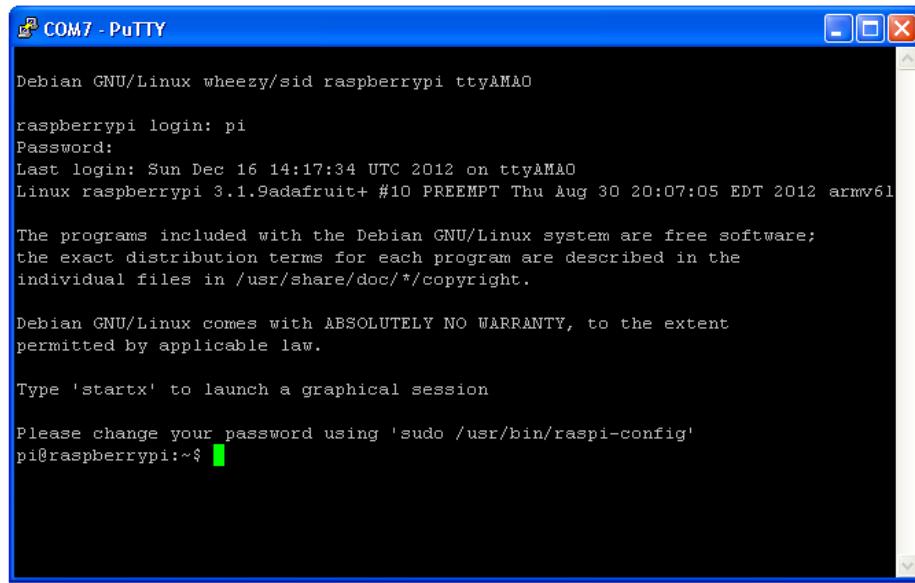
Изгледът е следния:



Фиг. 4.1.7. Изглед на програмата putty

При стартирането на тути, избираме connection type Serial, Serial Line въвеждаме COMx, където x е стойността, взета от device manager, а speed винаги е 115200.

При натискане на бутон open, ще ни излезе конзолата на RPI.



The screenshot shows a PuTTY terminal window titled "COM7 - PuTTY". The window displays a terminal session on a Raspberry Pi running Debian Wheezy. The session starts with a login prompt for "raspberrypi" user, followed by a password prompt. Below the login information, it shows the system's last login details and the kernel version. It then displays standard Debian free software distribution terms and a disclaimer about warranty. Finally, it prompts the user to change their password using the "raspi-config" tool.

```
Debian GNU/Linux wheezy/sid raspberrypi ttyAMA0\n\nraspberrypi login: pi\nPassword:\nLast login: Sun Dec 16 14:17:34 UTC 2012 on ttyAMA0\nLinux raspberrypi 3.1.9adafruit+ #10 PREEMPT Thu Aug 30 20:07:05 EDT 2012 armv6l\n\nThe programs included with the Debian GNU/Linux system are free software;\nthe exact distribution terms for each program are described in the\nindividual files in /usr/share/doc/*copyright.\n\nDebian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent\npermitted by applicable law.\n\nType 'startx' to launch a graphical session\n\nPlease change your password using 'sudo /usr/bin/raspi-config'\npi@raspberrypi:~$
```

Фиг. 4.1.1.8. Терминала на RPI

Въвеждаме като име `pi`, а като парола – `raspberry`. Вече сме вътре в конзолата.

Продължете надолу за да разберете как да конфигурирате конзолата.

## Чрез Network cable и рутер

За тази връзка са ни нужни само един рутер, кабел за мрежова връзка, както и

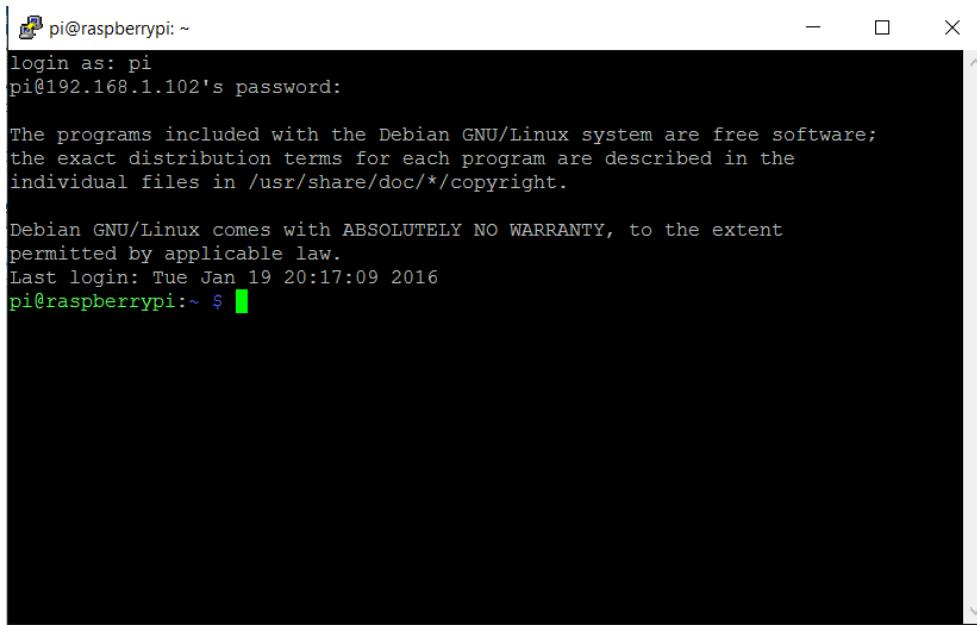


Фиг. 4.1.9. Връзка между RPI и рутер през LAN кабел



Фиг. 4.1.10. Намиране на RPI в локалната мрежа

приложение за сканиране на безжичната мрежа(фиг. 4.1.10. е безплатното приложение за андроид „network scanner”, за което не е нужен root на телефона). Първо връзваме конзолата към рутера (както е показано на фигура 4.1.9.), след това използваме приложението за сканиране на мрежата за да разберем на кое IP е микроконтролера. След това чрез Putty се връзваме към контролера през ssh. Това се случва като на полето hostname въведем локалния адрес на RPI, работим през порт 22, и сме задали начин на свързване ssh. При натискане на бутон open, ще ни излезе терминал на конзолата. Въвеждаме име rpi и парола raspberry, и така влизаме в системата на конзолата.



A screenshot of a terminal window titled "pi@raspberrypi: ~". It shows the following text:  
login as: pi  
pi@192.168.1.102's password:  
  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/\*/\*copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
Last login: Tue Jan 19 20:17:09 2016  
pi@raspberrypi:~ \$

Фиг. 4.1.12. Конзолата на RPI

След като вече имате достъп до основата на конзолата, трябва да въведем команда `sudo nano /etc/wpa_supplicant/wpa_supplicant.conf`. Това ще ни отвори терминален редактор, чрез който трябва да добавим команди, чрез които автоматично да се връзваме към безжичната мрежа.

Примерни данни може да видим на картинката вляво. След като сме променили файла за да излезем, натискаме `ctrl+X`, след това `Y` за да запазим файла, и `enter` за да излезем от него.

С този процес ние вече имаме готова конзола. Тя вече може да се върже към интернет. Автоматично пуска скрипта чрез който се избира коя игра и с колко

```
network = {  
    ssid = "my_network_name"  
    psk = "my_network_password"  
}
```

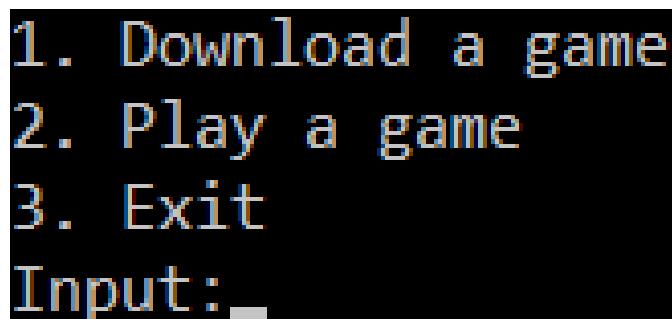
играчи да се играе. След като сме стартирали конзолата и сме и подали захранване чрез micro-USB кабела към micro-USB порта, ние трябва да стартираме контролерите за игра.

Фиг. 4.1.13. Примерни данни за вход към мрежата

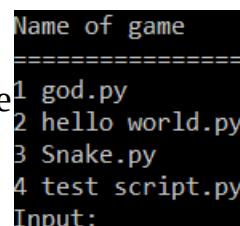
## 4.2. Стартране на игралните контроли

Стартиране на игралните контроли е значително по-лесно. При натискане на power бутона, контролера се пуска. При пускането на контролера, ще ни изведе клавиатура, чрез която да въведем името и паролата на мрежата. От страна на контролера е само това – трябва да пуснем контролера и да въведем името и паролата на мрежата. Оттам той сам ще се свърже и ще праща данните на конзолата.

При стартиране на всичко, в конзолата ще се появи текст (фигура 4.1.14.), чрез който ние ще можем да пуснем игра, както и да изтеглим игра. Също ние можем и да изключим скрипта.



Фигура 4.1.14. Изглед при стартиране на конзолата



При селектиране на опция 1, ще се

Фигура 4.1.16. Изглед при стартиране на игра

покажат всички игри, качени на игровия сървър. Оттам ние можем да изтеглим играта, и чрез опция 2 да я стартираме. На фигура 4.1.15. и 4.1.16. се вижда изгледа, който ни се открива при селектиране на съответно на опция 1 и 2 (за теглене, и за играене на игра).

```
Name of game
=====
hello world
Snake
test script
god
Input:■
```

Фигура 4.1.15. Изглед при теглене на игра

## 4.3. Наръчник на програмиста

Тази част от ръководството на потребителя е по-сложен, но е направен само за улеснение на програмистите, които ще пишат игри или софтуер върху нашата система. В тази точка ще кажем на програмиста как да пише скриптовете, които ще се явяват игрите, които ще може да играе потребителя.

### Workflow

В тази част ще се пише за това какви са основните неща, които всеки един програмист трябва да направи за да напише програмата, която иска.

Първо програмиста може да реши дали да е режим на дебъгване или в режим на продукция. Ако е в режим на дебъгване, информацията идва от клавиатура, така че не е нужно програмиста да има контролер или конзолата като цяло. Трябва му само един Python интерпретатор. Ако е в продукционен режим, то програмата ще чете входа си от UDP сокет, което прави дебъгването с една идея

по-трудно.

След това програмиста трябва да реши какво ще е неговото игрално поле – двуизмерно или триизмерно. Това става чрез викането на класа play\_ground\_2D или play\_ground\_3D за двуизмерно и съответно триизмерно игрално поле. След избирането на това с което поле ще се занимава, той трябва да избере големината на полето. Тя се подава като отделни целочислени числа, и стойността на игралното поле в началото. Тя по подразбиране е 0, а големината - 8 на 8 или 8 на 8 за 2D и 3D.

След като избере игрално поле, програмиста трябва да избере какво се случва при натискане на бутоните. Конзолата, при приемане на команди, свързва тези команди с някакво име, дефинирано от програмиста. Той може да избере дали имената да са „1“, „2“ и тн. или „Иван“, „Пешо“ и тн. Всичко е избор на програмиста да направи имената адекватни спрямо приложението. Имената са направени така, че първото име да се взима от първия контролер, който се свърже към сървъра, второто – към втория и тн. Програмиста трябва да предефинира метода call\_function\_with\_player, част от класа all\_clients\_handler. Дефинирането на вход от потребителя и прединирането на функцията става по следния начин:

```
my_server = gameEngine.server()  
my_server.ch.call_function_with_player = call_function_with_player  
my_server.start()
```

,

където call\_function\_with\_player е предефинирана от програмиста функция.

След като предефинира функциите, които се викат, той трябва да дефинира спрайтове. Те се дефинират чрез унаследяване на класът спрайт, и пренаписването на функцията move ако е нужно, ако не е нужно нещо различно от поведението на спрайта по подразбиране, то само трябва да му се зададе тип и

форма. Тип е символен низ, чрез който ще разбираме кой-кой е ударил на игралното поле, а форма – масив от масив, всеки от който съдържа три стойност – Y, X и стойността на клетката с индекс XY. Ако се говори за 3D игрално поле, то масива shape съдържа масиви с по 4 стойности – Y,X,Z, и стойността на клетката с индекс XYZ.

След дефиницията на спрайтовете, програмиста трябва да инициализира класовете movable\_handler и bumpable\_handler. Тези класове се използват за управление на спрайтовете – всеки спрайт, който трябва да се движи без вход от потребителя се слага в movable\_handler, където ще се вика неговия метод move на всеки n секунди, където n е специфициран от програмиста. bumpable\_handler има за цел да държи всички спрайтове и да гледа дали някой не се е докоснал. При докосване, се извиква функцията object\_collision с 2 аргумента – двета спрайта, които са се сблъскали. Програмиста трябва да предефинира функцията, така че да му свърши работа. По подразбиране тя не прави нищо. При унищожение на обект, то той се маха от тези масиви, и след време бива изчищен от garbage collector-а на python. В действителност това са основните стъпки, които трябва да направи програмиста за да пише игри на нашата конзола. За други функционалности, програмиста може да разгледа библиотеката, която съдържа много функции за манипулация на спрайтове, игралното поле и сървъра/аудио класът. Ако някой иска да помогне с прогреса на софтуерната част на конзолата, то той може да се свърже с мен на

[momchil.angelov.1997@gmail.com](mailto:momchil.angelov.1997@gmail.com),

с цел по-бързо имплементиране на идеите, които му хрумнат.

## Пример за създаване на игра: Змия

Първа стъпка: Ние трябва да импортираме gameEngine модула в нашия игрови скрипт. Това се случва чрез следните 4 линии код:

```
import sys
import os
sys.path.insert(1, os.path.join(sys.path[0], '..'))
import gameEngine
```

Тъй като модула не се намира в текущата директория, то ние трябва да вкараме в os.path променливата бащината директория.

Втора стъпка: Трябва да дефинираме класът myPlayerHandler, който наследява класът all\_clients\_handler на gameEngine модула.

```
class myPlayersHandler(gameEngine.all_clients_handler):
    def __init__(self, names = ['Player1','Player2','Player3','Player4']):
        super().__init__(names)

    def call_function_with_player(self, data, player):
        global sprite
        if player == self.names[0]:
            if data == "0" and sprite.direction == 4:
                pass
            else:
                sprite.direction = 2
        if data == "90" and sprite.direction == 3:
            pass
        else:
            sprite.direction = 1
        if data == "180" and sprite.direction == 2:
            pass
        else:
            sprite.direction = 4
```

```
if data == "270" and sprite.direction == 1:
        pass
else:
        sprite.direction = 3
```

Този клас има за цел да ни предостави връзка между игровата конзола и входа, който подаваме. Логика на кода – names са имената, които се дават на всеки играч. Първия играч, който се върже към конзолата, взима първото име, втория – второто и т.н. Във функцията call\_function\_with\_player, ние дефинираме при какви получени данни, какво да се случи. В този случай, ако данните са получени от играч, различен от първия – то той се игнорира, защото змия е single-player игра. След това указваме на интерпретатора, че ще използваме глобалната променлива sprite, която е змията в нашия случай. След това, ние указваме какво да се случи при различни входни данни – при подаване на нагоре (90 градуса), змията се завърта нагоре (ако досега се е движила надолу, и ние кажем да отиде нагоре – то тогава тя ще се самоизяде. Затова слагаме условен оператор за да проверим дали е коректна подадената посока и моментната посока). Sprite.direction е посоката, в която змията се движи. Всичко това ще бъде разгледано след малко, засега само трябва да знаете, че в myPlayersHandler дефинираме интеракцията на потребителя с игровата конзола.

Трета стъпка: Трябва да дефинираме класът, който ще ни дава логиката при сблъсък на два обекта, какво да се случи спрямо типът им. Този клас е наследник на класът bumpable\_handler от модула gameEngine.

```

class myBumpableHandler(gameEngine.bumpable_handler):
    def __init__(self, pg):
        super().__init__(pg)

    def object_collision(self, object1, object2):
        if object1._type == "Snake" and object2._type == "Apple":
            object1.grow()
            object2.respawn(self.pg)
        if object1._type == "Apple" and object2._type == "Snake":
            object1.respawn(self.pg)
            object2.grow()

```

На този клас ние подаваме като аргумент на конструктора игралното поле pg.

Магията на този клас се случва чрез функцията object\_collision, която приема като аргументи двета обекта. За извикването на тази функция се грижи gameEngine модулът, но ние трябва да дефинираме какво се случва, на база типът на обектите. Ако обектите са например от 2 различни фракции, то те могат да си нанесът никакви щети. В нашия случай, ако двета обекта се сблъскат, то ние викаме метода grow() на обекта от тип „змия“, и метода respawn на обекта от тип „ябълка“. Покъсно ще разберем точно какво правят тези методи. Важното е да разберем какво се случва в този клас – интеракцията между обектите при тяхното сблъскване (bump).

Четвърта стъпка: дефиниране на спрайтовете. В тази стъпка ние ще дефинираме нашите спрайтове. Този случай, това са спрайт от тип Ябълка и спрайт от тип Змия. Първо ще започнем с дефиницията на класът Ябълка, защото е по-лесен.

```

class Apple(gameEngine.Sprite):
    def __init__(self, _type, matrix=[[5,2,3]]):
        super().__init__(type, matrix)
        self._type = _type

    def respawn(self, pg):
        self.putAtRandomPlace(pg)

```

От тук се вижда, че класът Ябълка наследява класът спрайт на gameEngine. В конструктора на този клас ние подаваме 2 аргумента – type е типът на спрайта, който използваме в **bumpable\_handler**. Matrix аргумента е масив от масиви с по 3 стойности, които са Y, X, и стойността на игралното поле в тази част. Използваме само 2 стойности, тъй като полето, върху което играем, ще е двуизмерно. Това по-късно ще покажем как го дефинираме. Методът respawn приема като аргумент игралното поле, и използва вградения метод putAtRandomPlace на класът Sprite, който слага обекта на произволно, празно място на полето.

Класът Змия е малко по-сложен, но пак е сравнително лесен.

```

class Snake(gameEngine.Sprite):
    def __init__(self, _type, direction, matrix):
        super().__init__(type, matrix)
        self.direction = direction
        self.last_block = [0,0,0]
        self._type = _type

    def grow(self):
        self.shape.insert(0, self.last_block)

```

```

def move(self, pg):
    pg.clearThis(self.shape)
    curr = self.shape[-1]
    direction = self.direction
    if direction == 2:
        _list = [curr[0], curr[1]+1, curr[2]]
    elif direction == 3:
        _list = [curr[0]+1, curr[1], curr[2]]
    elif direction == 4:
        _list = [curr[0], curr[1]-1, curr[2]]
    elif direction == 1:
        _list = [curr[0]-1, curr[1], curr[2]]

    if _list in self.shape:
        return 0

    self.last_block = self.shape[0]
    del self.shape[0]
    self.shape.append(_list)
    self.draw(pg)

```

Както се забелязва, конструкторът на класа Змия приема и аргумент direction. Този аргумент ни показва накъде ще се движки змията. Тъй като ние не я движим, само я насочваме, ние дефинираме и метод move(), който е задължителен за всички обекти, които ще се придвижват сами по игралното поле. Метода е лесен, взима най-предния елемент (главата) на змията, и спрямо посоката в която се движжи, прави нов елемент и го слага в масива, който е формата на змията. След това първия елемент на масива се изтрива (опашката). В самия обект се пази опашката на змията. Това е така, за да може при изяждане на ябълката, змията да може да регенерира своята опашка и да порастне (методът grow()). Чрез метода draw на класът Sprite, спрайтът се рисува на игралното поле.

Пета стъпка: в пета стъпка ние дефинираме gameServer обекта и myPlayersHandler обекта

Тази стъпка е лесна, защото ние вече сме дефинирали какво да се случва при получаване на някакви данни (Втора стъпка, myPlayerHandler). Затова ние правиме една инстанция на класът myPlayerHandler,

```
players_handler = myPlayersHandler(["Pesho","Dragan","Petko","Marin"])
```

и подаваме този обект като аргумент на gameServer., и го стартираме. Целта на gameServer е да подаде самите данни на myPlayerHandler.

```
game_server = gameEngine.gameServer(players_handler)  
game_server.start()
```

След стартирането на game\_server, той автоматично ще приема данни от контролерите, и ще ги обработва спрямо указанията в myPlayerHandler. Още малко

Шеста стъпка: инициализация на игралното поле.

```
pg = gameEngine.PlayGround2D()
```

Инициализацията на игралното поле е лесно – чрез записания по-горе запис ние дефинираме едно двумерно игрално поле, с големина 8x8, и стойност по подразбиране на полето 0. Ако са му зададени параметри, то първия ще е дължината по Y, втория – дължината по X, а третия – стойността по подразбиране на полето.

Седма стъпка: инициализация на спрайтовете и на bumpable handler.

```
snake = Snake("Snake", 3, snake_matrix)
apple = Apple("Apple")
```

```
bumpable_handler = myBumpableHandler(pg)
bumpable_handler.add(snake)
bumpable_handler.add(apple)
```

Единственото неразгледано до този момент в този код е метода add на класът gameEngine.Bumplable\_Handler. Този метод вкарва обекти в масива, който проверяваме дали 2 негови обекта са се бълснали.

Осма стъпка: инициализация на movable\_handler

```
movable_handler = gameEngine.movable_handler(pg, bumpable_handler)
movable_handler.add_object(snake)
movable_handler.run()
```

Класът movable\_handler има за цел коректно да придвижва всички обекти, които трябва сами да се движат. Аргументите на този клас са bumpable\_handler(за да може да проверява дали 2 обекта са се бълснали при всяко движение), игралното поле, и таймер, който ни казва през колко секунди се вика метода move на обектите вътре в handler-а. По подразбиране е 0.5 секунди. Вътре в самия клас, ние държим всички обекти, които трябва да се движат (те се добавят чрез метода add\_object). В нашия случай, единствения обект, който трябва да се движки е snake.

Девета стъпка: последни стъпки!

И това е края на „Как да си направим игра: Змия“.

Последните няколко реда код, които трябва да бъдат написани, са:

```
snake.draw(pg)  
apple.draw(pg)  
pg.draw()
```

*while 1:*

```
    pass
```

Чрез методите draw на класът Sprite, с аргумент игралното поле, ние казваме на игралното поле да сложи спрайтовете върху себе си. От тук, мога само да кажа: Happy Coding, and let the Force be with you! :)

# Заключение

Целта на тази дипломна работа беше да се постигне продукт, който да забавлява хората. Но целта беше също в това да разбера повече за езика за програмиране Python, както и работа в embedded среда. Други знания, които натрупах от тази дипломна, са работа с различните интернет протоколи и тяхната разлика, кой кога да ползвам, модулно програмиране, както и работа в екип. Всички те бяха включени в направата на тази дипломна работа.

За усъвършенстването на дипломната работа – тъй като това е голям проект, то за усъвършенстване може да се намерят по-ефективни начини за обработката на информацията, оправяне на някои все още ненамерени бъгове.

За обогатяване на дипломната работа – за обогатяване могат да се кажат някои неща. Примерно може да се добавят още функционалности на библиотеката за създаване на игри, като поддържане на някои по-разпространени видове хардуер(пиезоелементи, резистори за затопляне/охлажддане на контролера), обратна връзка към контролера, както и писане на драйвери за използване на възможно най-много различни екрани. Откъм сървърна страна, може да се поддържа комуникация между две различни конзоли през сървъра, което ще създаде нужда от пренаписване на сървъра.

При изпълнение на точките по-горе, ние ще можем да предоставим нашата игрова конзола на пазара. Това е начина за монетизация на този проект – чрез продаване на самата игра заедно с контролерите и визуализиращите устройства.

# Използвана литература

Основни източници на информация са stack\_overflow, stack\_exchange и документацията на python.

1. <https://www.arduboy.com/> - сайтът на игровата конзола Arduboy
2. <https://github.com/Arduboy/Arduboy> – кодът на библиотеката на Arduboy
3. <http://logicalzero.com/gamby/> - сайтът на игровата конзола GAMBY
4. <https://github.com/logicalzero/gamby> – кодът на библиотеката на GAMBY
5. <http://www.ladyada.net/make/fuzebox/> - сайтът на игровата конзола Fuzebox
6. <https://github.com/Uzebox/uzebox> – кодът на библиотеката на Fuzebox
7. <https://nootropicdesign.com/hackvision/> - сайтът на игровата конзола hackvision
8. <https://nootropicdesign.com/hackvision/games.html> – кодът на някои от игрите за hackvision
9. <https://www.techwillsaveus.com/shop/diy-gamer-kit/> - сайтът на игровия комплект DIY gamer-kit
10. <https://github.com/techwillsaveus/Gamer> – кодът на примерните игри на конзолата DIY gamer-kit
11. <http://gamebuino.com/> - сайтът на игровата конзола Gamebuino
12. <https://github.com/Rodot/Gamebuino> – кодът на игровата библиотека на Gamebuino
13. <http://apcmag.com/arduino-project-7-build-a-retro-gamebox.htm/> - сайтът на retro gamebox
14. <https://en.wikipedia.org/wiki/Bluetooth> – статия в Уикипедия за технологията Bluetooth

15. <http://www.differencebetween.net/technology/internet/difference-between-tcp-and-http/> - статия за разликата между протоколите TCP и HTTP
16. [https://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol) – статия в Уикипедия за TCP протокола
17. [https://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) – статия в Уикипедия за HTTP протокола
18. <https://www.elprocus.com/types-of-wireless-communication-applications/> - статия за различните видове безжични комуникации в днешно време
19. <http://www.gamedev.net/topic/628013-what-are-the-basics-to-making-a-game-engine> – форум за разработка на игри
20. <http://docs.unity3d.com/Manual/AnimationOverview.html> – Unity game engine, подраздел свързан с анимации
21.  
[https://en.wikipedia.org/wiki/Artificial\\_intelligence\\_\(video\\_games\)](https://en.wikipedia.org/wiki/Artificial_intelligence_(video_games)) – изкуствен интелект във видео игрите
22. [https://en.wikipedia.org/wiki/Artificial\\_intelligence](https://en.wikipedia.org/wiki/Artificial_intelligence) – изкуствен интелект в реалния свят
23. <https://en.wikipedia.org/wiki/Pathfinding> – описание на алгоритъм на Дейкстра, описание на A\* алгоритъм
24. <http://www.jenovachen.com/flowingames/designfig.htm> – описание на game flow system, съвети за по-добър gameplay на игрите
25. <http://gamedev.stackexchange.com/questions/33453/how-smartly-implement-scripting-in-game> – начин за имплементация на скриптове в game engine
26. <https://books.google.bg/books?>

[`id=0fPRBQAAQBAJ&pg=PA45&lpg=PA45&dq=object+authority+policy+games&sou`](#)  
[`rce=bl&ots=5TkWJWQiKV&sig=4ydWrAF\_TRHQnmGYdHpZMqhmv2A&hl=en&sa=X&ved=`](#)  
[`0ahUKEwiz45zps43LAhWsJZoKHacbCBCQ6AEIGjAA#v=onepage&q=object`](#)

[`%20authority%20policy%20games&f=false`](#) – книга за архитектурата на един game engine

27. - <https://docs.python.org/3/> - документация на езика python версия 3

28. - <http://stackoverflow.com/> - сайт, където можем да намерим решение на повечето от проблемите си

## Съдържание

Член-променливи.....	47
controller_input.....	47
Методи.....	47
Член-променливи.....	48
server.....	48
Методи.....	48
Член-променливи.....	49
3.4. Описание на ROM-овете.....	50
3.5. Описание на сървъра.....	52
4. Четвърта глава. Ръководство на потребителя.....	55
4.1. Стартiranе на игровата конзола.....	55
USB to TTL.....	55
Чрез Network cable и рутер.....	63
4.2. Стартiranе на игралните контроли.....	65
4.3. Наръчник на програмиста.....	66
Workflow.....	66
Пример за създаване на игра: Змия.....	69
Първа стъпка.....	69
Втора стъпка.....	69
Трета стъпка.....	70
Четвърта стъпка.....	71
Пета стъпка.....	74
Пета стъпка.....	74
Шеста стъпка.....	74
Седма стъпка.....	75
Осма стъпка.....	75
Девета стъпка.....	75
Заключение.....	77
Използвана литература.....	78