



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дипломна работа

по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

Тема: Разработка на игра с Rust и библиотеката Bevy

Дипломант:

Станислав Георгиев Иванов

Дипломен ръководител:

Николай Данаилов

СОФИЯ

2023



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 22.11.2022 г.
Дата на предаване: 22.02.2023 г.

Утвърждавам:
/проф. д-р инж. П. Якимов/

ЗАДАНИЕ за дипломна работа

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ
по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

на ученика Станислав Георгиев Иванов от 12 а клас

1. Тема: Разработка на игра с Rust и библиотеката Bevy
2. Изисквания:
 1. Система за движение на героя
 2. Показване на логически пъзел и възможност да бъде решен
 3. Възможност за преминаване на следващо ниво при успешно решаване на пъзела
3. Съдържание
 - 3.1 Теоретична част
 - 3.2 Практическа част
 - 3.3 Приложение

Дипломант:
/ Станислав Иванов /
Ръководител:
/ Николай Данаилов /

Директор:
/ доц. д-р инж. Ст. Стефанова /



Становище на дипломния ръководител

Дипломантът Станислав Иванов е покрил всички изисквания и умело е използвал похватите и добрите практики, свързани с ECS модела.

Предлагам за рецензент Александър Ангелов.

Дата: 21.02.2023 г.
гр. София

Подпись:
/ Николай Данаилов /

Увод

Видеоигрите са се превърнали в доста популярна форма на забавление, като милиони хора по света прекарват часове всяка седмица, потопени във виртуалните им светове. Въпреки че видеоигрите често биват наричани „безсмислени разсейвания“, има все повече изследвания, които предполагат, че те оказват значително влияние върху човешкото поведение. От подобряването на когнитивната функция до наಸърчаването на социализацията, видеоигрите могат да повлияят на живота ни по много начини.

През последните години дебатът около видеоигрите и тяхното въздействие върху човешкото поведение става все по-разгорещен. От една страна, някои твърдят, че видеоигрите са отговорни за редица отрицателни резултати, включително насилие, пристрастване и лошо академично представяне. От друга страна, много играчи и изследователи посочват положителните ефекти на видеоигрите, някои от които са подобрени умения за решаване на проблеми, повишена креативност и социализация.

Те неизменно влияят на формирането на личността, като най-голям ефект се наблюдава при играчите в юношеска възраст. Всяка игра може да има както положителни, така и отрицателни ефекти върху играта. Тяхното съотношение варира най-често с жанра ѝ.

Ако вземем за пример онлайн ролевите игри, те биха могли да подобрят комуникационните умения на играта, въображението му и уменията за решаване на проблеми. Но могат и да доведат до пристрастване, тъй като най-често в такива игри си създаваме играч, който ни наподобява, а в течението на играта,

характеристиките му се подобряват. По този начин ние създаваме перфектна версия на себе си във виртуалния свят и не искаме да прекарваме много време извън него.

Друг популяррен вид игри са стрелбите от първо лице. При тях играчите успяват да развият когнитивните си умения, вниманието и скоростта на реакциите си. Но за сметка на това, цялата агресия, която проявяват по време на игра, често се пренасят в истинския свят и стават част от техния живот.

Има и немалка група хора, неповлияни от изживяванията си с видеоигрите. Те не се поддават на промени, породени от виртуалните светове, като по този начин обрват тезата, че видеоигрите могат да променят човек завинаги, която е често използвана в негативен смисъл. [14]

Игрите с логически пъзели развиват мисленето, въображението, концентрацията, справянето с проблеми. Минус на повечето от тях е, че на играчите им става скучно преди да са завършили цялата игра. Решение на този проблем представлява някакъв вид история, която да поддържа любопитството на играта.

Целта на настоящата дипломна работа е да създаде игра, написана на езика Rust, използвайки библиотеката Bevy, която да спада в последната спомената категория, а именно тази на игрите с логически пъзели.

ПЪРВА ГЛАВА

Оглед на подобни проекти

1.1. Платформени игри

Платформените игри са такива, в които героят обикаля малък свят с неравен терен и пълен с платформи, които се намират на различна височина и понякога се местят, като трябва да се достигне някаква цел, избягвайки препятствията по пътя. Главните механики за изследването на картата са скачане и катерене. Допълнителни движения, които могат да се срещнат в такива игри са люлеене на лиани, отскачане от стени, реене или засилване в дадена посока във въздуха, изстрелване от оръдия и скачане върху трамплини за достигане на по-високи места. Гледната точка в двуизмерните платформени игри е странична - движението наляво и надясно за играта са напред и назад за героя в играта.

1.1.1. Hollow Knight

Hollow Knight е игра, разработена от Team Cherry. В нея играчите влизат в ролята на безименния насекомоподобен рицар. Играта не е разделена на нива, а представлява една огромна карта, разделена на сектори.

Започва с пристигането на рицаря в Dirlmouth - тихо градче, разположено над останките на някога процъфтяващото кралство Hallownest. Докато изследва руините му под земята, играчът може да разговаря с различни същества, които среща по пътя, за да научи повече за историята на кралството и как се е стигнало до сегашното му състояние.

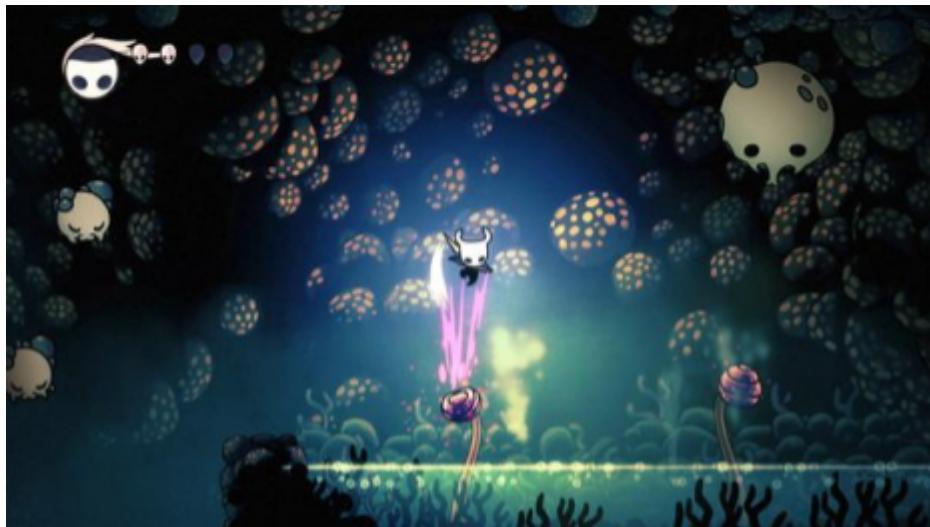
По време на играта, играчът трябва да се сражава с насекомоподобни опоненти, използвайки меча nail (в превод пирон), с помощта на който може да отскача от противниците, когато ги атакува отгоре. След като ги победи, те го награждават с Geo - валутата, която се използва за закупуване на допълнителни предмети и подобрения. С достигането на нов сектор, картата му не е известна, а трябва да се намери картографа Cornifer, от когото да се закупи копие от плана на сектора. Целта е да се изследва целият свят, като по този начин играчът да разкрие истината за Hallownest, забулена в мистерия с времето.



фиг. 1.1

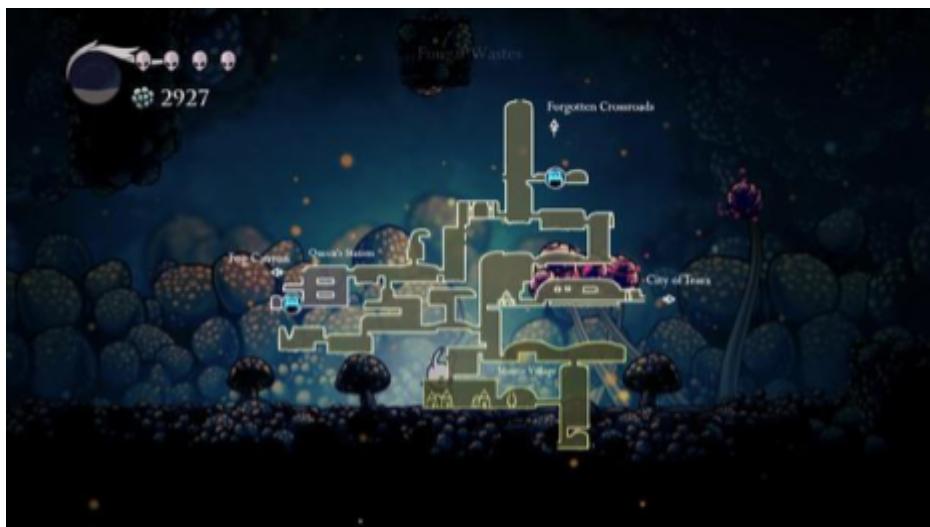


фиг. 1.2



фиг. 1.3

На фиг. 1.1 е показана една от атаките, които се отключват с течение на играта. На фиг. 1.2 може да се види стандартната атака, с която играчът започва, а на фиг. 1.3 как тя може да се използва надолу за отскок от конкретни обекти.



фиг. 1.4

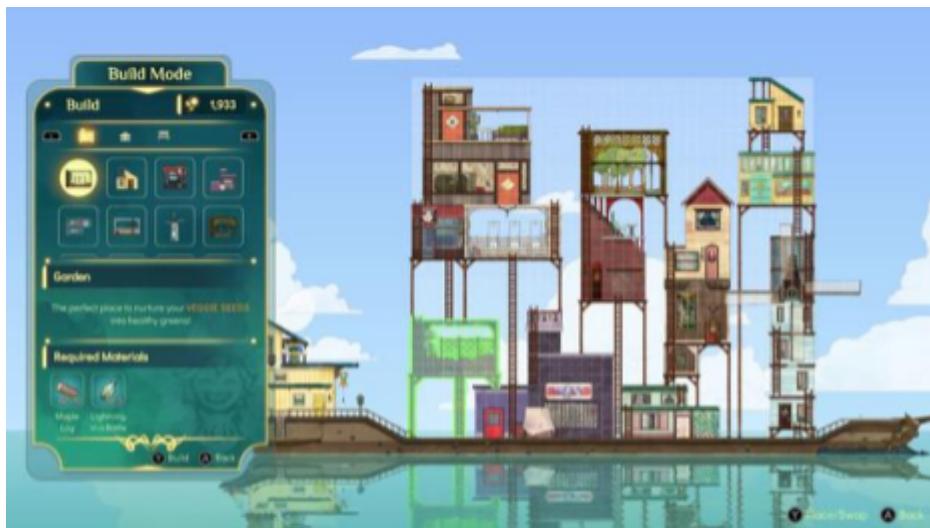
Фиг. 1.4 показва экрана, който се отваря, когато играчът достъпи картата. На картата са отбелязани пейките за почивка. Това са местата, на които играчът може да запази прогреса си до този момент. Последната използвана пейка е мястото, откъдето той продължава, ако бъде победен.

1.1.2. Spiritfarer

Spiritfarer е игра, разработена от Thunder Lotus Games, в която играчът влиза в ролята на Стела, която, заедно със своята котка Дафодил (в превод от английски - Нарцис), заема мястото на лодкаря Харон и трябва да се грижи за душите на хората преди да ги изпрати в задгробния живот.

Стела става капитан на ферибот, който използва, за да обикаля света и да събира души. Със стъпването си на него, те се превръщат в животни - всеки в своето духовно животно. Задължение на Стела е да се погрижи те да изпълнят последните си желания, като тя ги приютива на ферибота и се грижи за тях - храни ги и ги занимава, като за целта строи различни видове стаи на ферибота. Те ѝ плащат по един обол, който тя може да използва, за да подобрява движенията си - скок във въздуха или използване на шапката за реене във въздуха. С времето играчът трябва да събира ресурси и да разширява плавателния съд - строи стаи за пътниците си, а също и места, на които да отглежда различни видове продукти. Харон дава на Стела и котката ѝ Вечната светлина (Everlight), която могат да използват като редица предмети. Те използват този подарък, за да се справят с тежката задача, която им е поставена. Именно той задвижва ферибота, с който плават.

Когато са изпълнили последното си желание, Стела изпраща душите през Вечната врата (Everdoor), като в този момент те ѝ благодарят за всичко, което тя е направила за тях. През пътешествието си, Стела се среща няколко пъти с Хадес, който е във формата на бухал. Той ни разкрива информация за живота на Стела, като накрая тя самата преминава през Вечната врата.



фиг. 1.5



фиг. 1.6

На фиг. 1.5 е показано менюто, което показва цената за изграждане на нова конструкция или за надграждане вече съществуваща такава. Фиг. 1.6 представлява картата, от която се задава следващата дестинация на ферибота. От този екран може да се изследва морето за потънали кораби, близки острови или региони с медузи, с хващането на които се събира валутата Glim, която се използва за закупуването на семена и други предмети от различните продавачи.



фиг. 1.7

На фиг. 1.7 се вижда моментът, в който Стела се сбогува с Гуен (Gwen) - първата душа, която се качва на ферибота на Стела и домашния ѝ любимиц, след като тя ги завежда до него.

1.2. Игри с логически пъзели

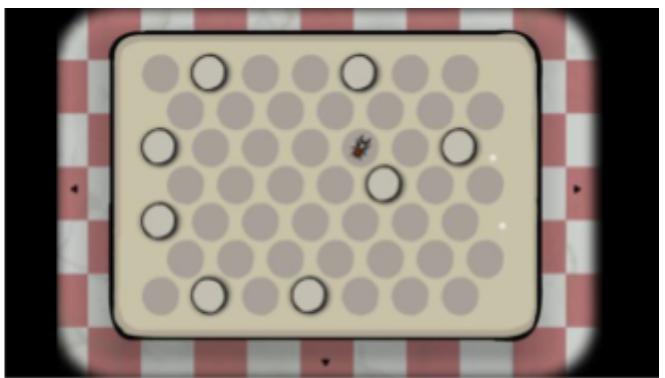
1.2.1. Rusty Lake

Започвайки с поредицата игри Cube Escape и продължавайки с поредицата Rusty Lake и единичните игри The White Door, Samsara Room и The Past Within, Rusty Lake е името не само на екипа, разработил тези игри, а и на вселената, която създават те.

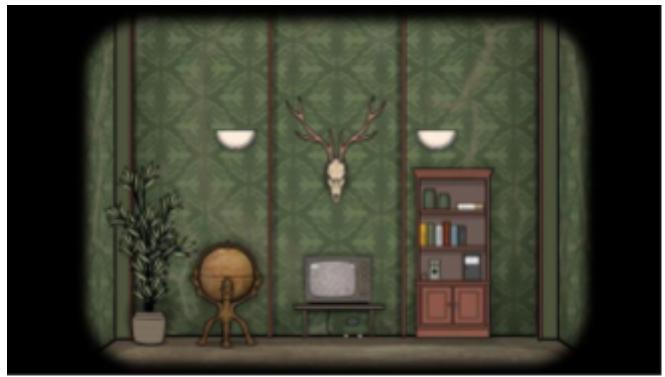
Всяка игра представлява кратка и сюрреалистична история, разказана чрез поредица от логически пъзели. Играчът трябва да събира предмети, които да използва за решаването им, което прави някои пъзели невъзможни за директно решаване - първо трябва да се съберат необходимите за отговора вещи или да се намерят необходимите подсказки. Приключенията, разказани от всяка игра, са свързани в една обща история, която продължава да се развива с всеки нов продукт на екипа.



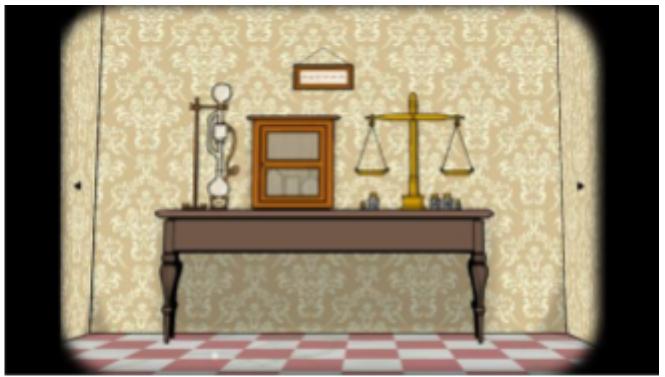
фиг. 1.8



фиг. 1.9



фиг. 1.10

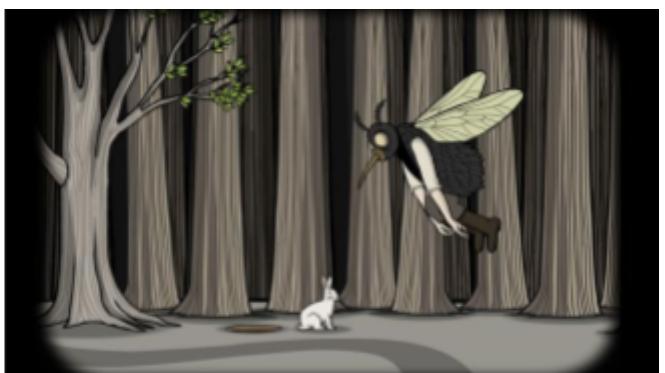


фиг. 1.11

На фиг. 1.8 и фиг. 1.9 са показани пъзели от различни игри от поредицата. На фиг. 1.10 и фиг. 1.11 са показани изгледите, когато не се решава пъзел, а се разхождаме из стаята. Типично за първата подпоредица, Cube Escape, е, че обхождането на картата представлява човек, застанал в средата на стая, който само се завърта към различните ѝ страни. От фиг. 1.12 и фиг. 1.13 можем да добием представа за специфичния стил на поредицата.



фиг. 1.12



фиг. 1.13

На фиг. 1.14 можем да видим как се случва оглеждането на някаква улика или подсказка.



фиг. 1.14

1.2.2. Adventure Escape



фиг. 1.15

Adventure Escape е поредица от мобилни игри, създадена от Haiku Games, в която най-често играчът проследява история, свързана с детектив Кейт, която разследва убийство. Тя е показана на фиг. 1.15, като фонът подсказва, че целта е да се достигне до извършителя на даденото престъпление.

Процесът се случва чрез обикаляне на местопрестълението за събиране на спомагателни предмети и улики и решаване на различни логически пъзели за достигане до повече информация относно извършителя.

След оглед на местопрестълението и достигане до извършителя, понякога той се опитва да избяга и трябва да го спрем, решавайки един последен пъзел за финал.



фиг. 1.16

На фиг. 1.16 е показано как можем да се разхождаме наоколо, събирайки предмети по пътя, които да използваме за по-късно. Те се пазят в панела от дясно. На фиг. 1.17 и фиг. 1.18 са показани, свалени от официалните страници на приложениета, снимки, съдържащи изображения на различни логически пъзели, които участват в различни игри.



фиг. 1.17



фиг. 1.18

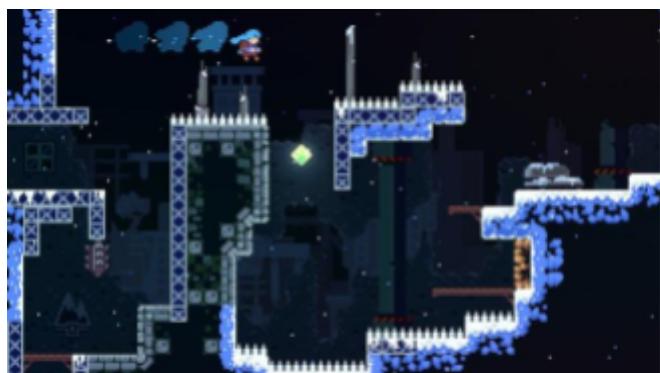
1.3. Платформени игри с пъзели

1.3.1. Celeste

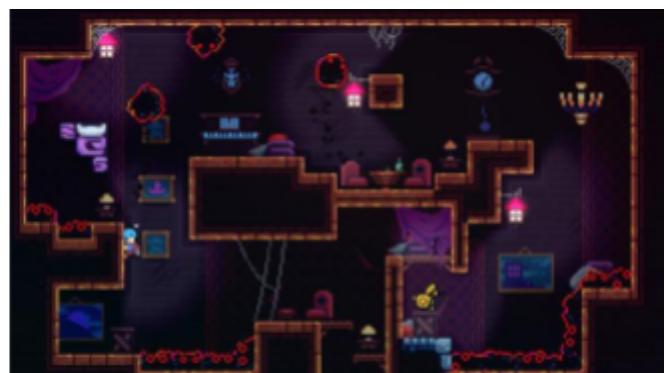
Celeste е двуизмерна платформена игра, разработена от Extremely OK Games, Ltd. и публикувана от Matt Makes Games Inc. Играчът трябва да контролира Маделин, за да минава през нивата, като по пътя събира различни предмети като ягоди и кристални сърца. Интересно за играта е, че всяко ниво се събира на екрана. По този начин няма скрита от играча информация за него.

Историята се развива в планината Celeste Mountain, която има различни сектори, включващи изоставен град и курорт. Играта започва в подножието на планината, където Маделин среща възрастна жена, която я предупреждава да се откаже от изкачването на планината. Маделин обаче не ѝ обръща внимание и тръгва към върха. По пътя си към върха, тя минава през различните сектори на планината, във всеки от които тя се сблъсква с ново предизвикателство, което да продължи историята.

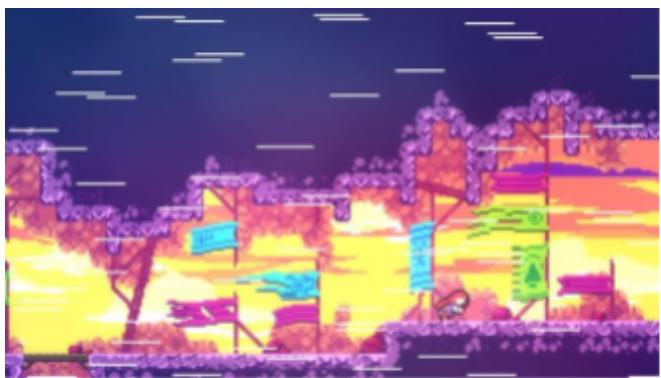
На фиг. 1.19 - фиг. 1.24 са показани някои нива, през които играчът трябва да премине. Всяко ниво има страна Б, която е по-трудна версия на нивото.



фиг. 1.19



фиг. 1.20



фиг. 1.21



фиг. 1.22



фиг. 1.23



фиг. 1.24

1.3.2. FEZ

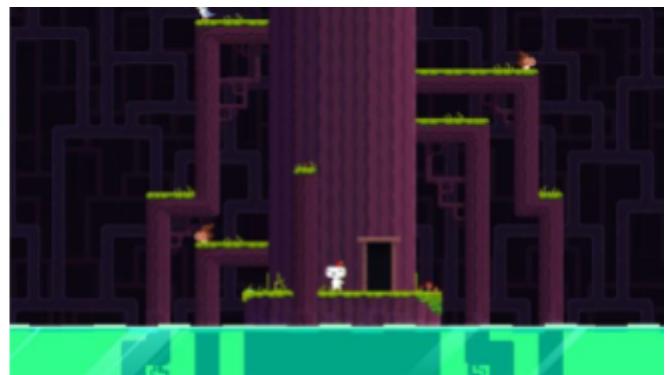
FEZ е двуизмерна платформена игра, развиваща се в триизмерен свят. Гомес води спокоен живот на двуизмерен самолет, докато не получава червен фес (тарбуш, шапка с пискюл), с което става свидетел на разкъсването на време-пространството, което му разкрива третото измерение.

Докато играчът движи Гомес наоколо, опитвайки се да стигне до края на нивото, той може да завърта двуизмерния свят, използвайки влиянието на третото измерение, като има общо четири гледни точки - от четирите страни на света. По този начин той свързва привидно несвързани платформи и намира начини да достигне до края на нивото. Тази механика е базова за играта.

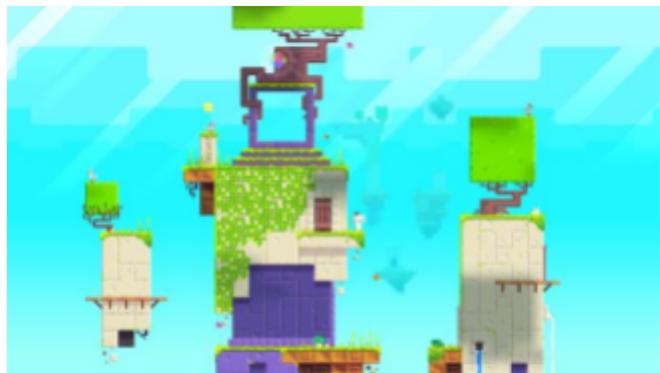
Целта на играта е да събира кубове и техни фрагменти в опит да възстанови равновесието във вселената. По време на играта могат да се намерят различни кодове и глифове, като играта оставя играчите без насоки за това дали те са малки пъзели, скрити в играта, или са просто фалшиви следи.



фиг. 1.25



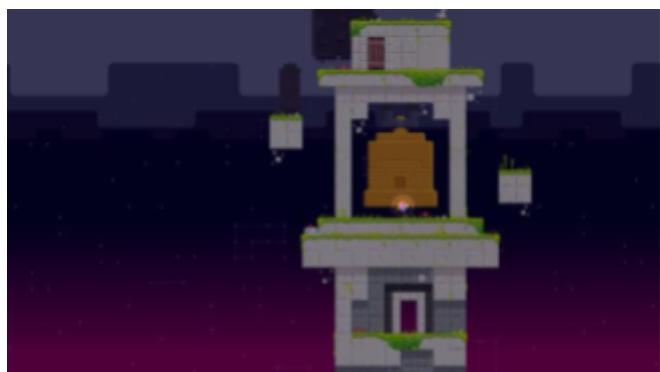
фиг. 1.26



фиг. 1.27



фиг. 1.28

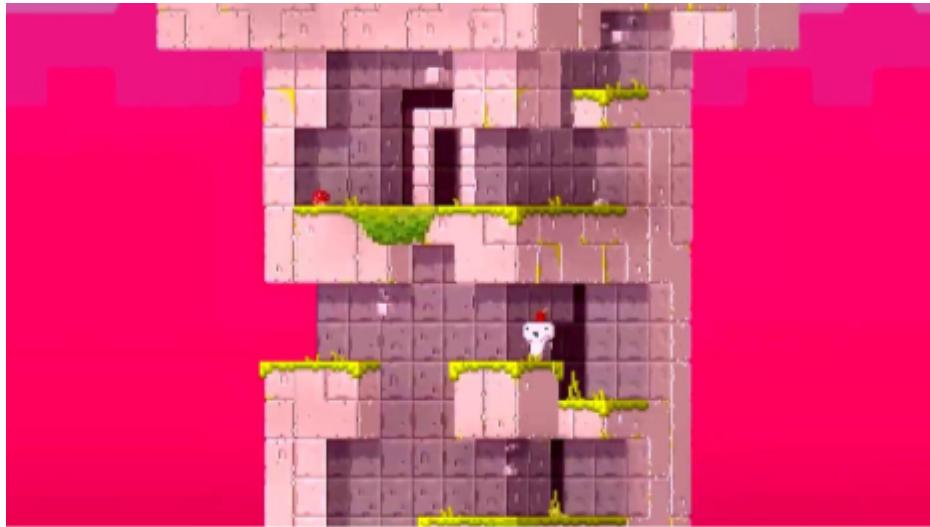


фиг. 1.29



фиг. 1.30

На фиг. 1.25 - фиг. 1.30 са изобразени различни нива, през които минава играчът по време на играта. На фиг. 1.31 е показан светът в момент, в който светът се завърта и гледната точка се измества от едната му страна към другата.



фиг. 1.31

1.4. Игри, направени с Rust и Bevy

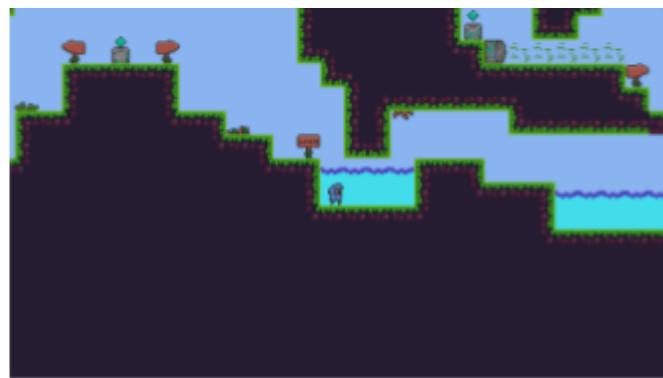
1.4.1. Elemental Sorcerer

Elemental Sorcerer е игра, чиято разработка е започнала в Bevy Jam 2 - състезание за игри, написани с Bevy, което продължило 10 дни и е било с тема „Combine“. Louis Tarvin продължил проекта и завършил малката игра, която представлява сега.

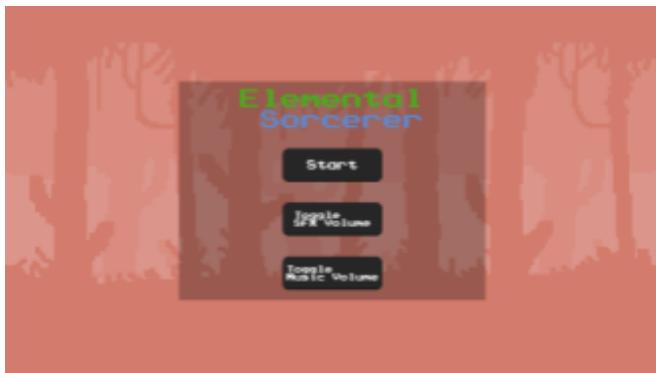
Целта на играча е да обиколи света, събирайки предметите и овладявайки елементите, като комбинира по едно от двете категории, за да се справи с различни препятствия. Например, комбинацията на наметалото с водния елемент позволява на играча да оцелява под вода, докато комбинацията му с въздушния елемент предпазва героя от избутване от въздушните течения. Ако комбинираме въздушния елемент с ботуши, получаваме правото на скок във въздуха.



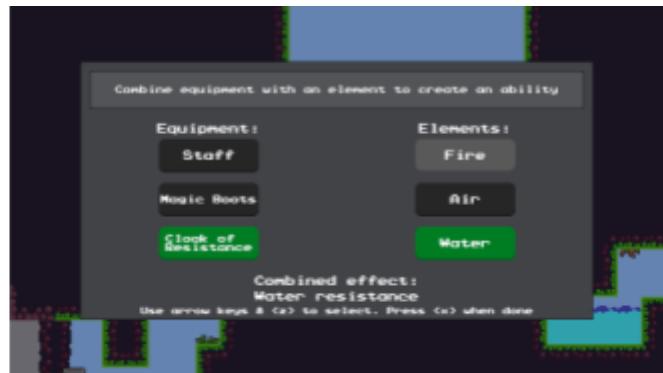
фиг. 1.32



фиг. 1.33



фиг. 1.34

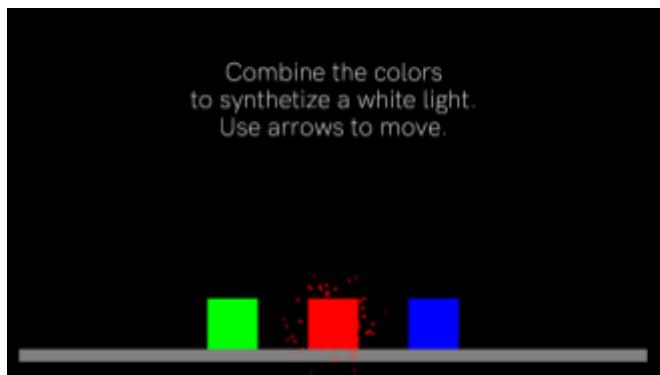


фиг. 1.35

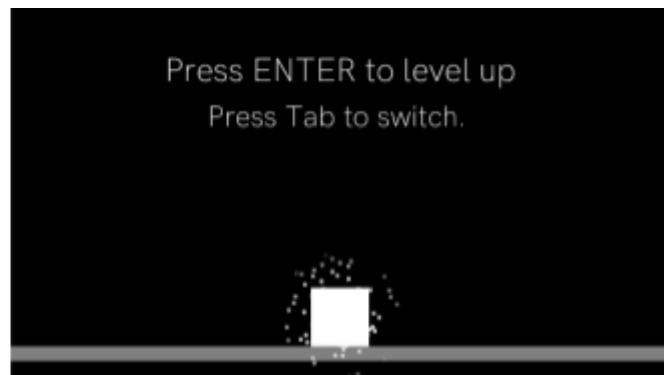
На фиг. 1.32 се вижда началната част от картата, заедно с логото на играта. На фиг. 1.33 е показано как играчът може да се движи под вода, като за тази цел се използва менюто, показано на фиг. 1.35. На фиг. 1.34 е показано началното меню, с което се започва играта.

1.4.2. Lux synthesē

Lux synthesē е друг проект, започнат на Bevy Jam 2, от Pascal Engélibert. Поради темата „Combine“, в играта играчът навигира кубове с различни цветове, които при контакт се обединяват и цветовете им се смесват. Те могат да минат през филтър, който премахва своя цвят от цвета на преминалия куб. Целта е да се достигне до куб с бял цвят. Когато това бъде постигнато, може да се премине на следващото ниво с бутона *Enter*.

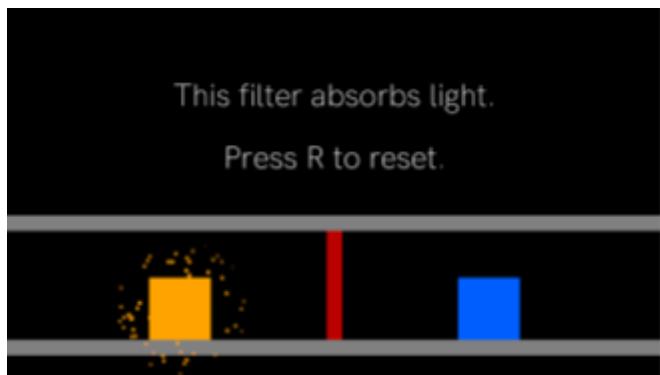


фиг. 1.36

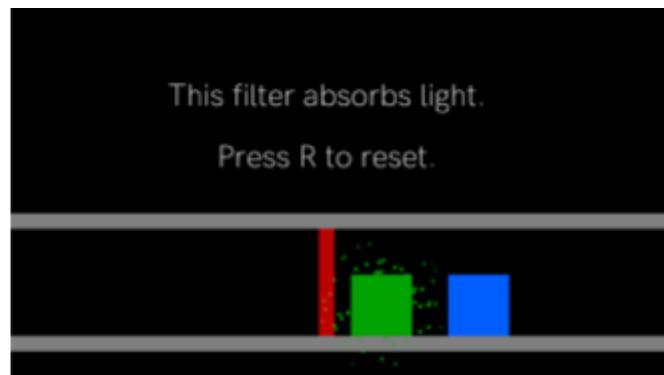


фиг. 1.37

На фиг. 1.36 е показано първото ниво от играта. Активният куб, който играчът мести със стрелките, има този ефект на излъчване на светлина. Активният куб се сменя с бутона *Tab*. Когато кубовете се комбинират така, че да се получи такъв с бял цвят, се появява текстът за преминаване на следващото ниво, както е показано на фиг. 1.37. На фиг. 1.38 е показано ниво, на което има филтър. Когато оранжевият куб премине през червения филтър, той става зелен. Този резултат може да се види на фиг. 1.39.



фиг. 1.38



фиг. 1.39

ВТОРА ГЛАВА

Изисквания към проекта

2.1. Основни функционални изисквания

Играта трябва да бъде разделена на отделни нива, които да се зареждат от LDtk файл, след което да бъдат визуализирани. Този файл съдържа информация както за визуалната репрезентация на нивото, така и за различните обекти, които трябва да бъдат заредени.

2.1.1. Система за движение на героя.

При отсъствие на сблъсък с обект, който да блокира пътя му, играчът трябва да има свободно движение наляво и надясно. За съществуващия в играта герой ще е достатъчно да има вертикална гравитация, която да го привлече към земята. При движение наляво и надясно трябва да има ограничение, което да не превиши определени граници.

Движението на героя трябва да изглежда по-реалистично. За да се постигне това, скоростта на движение, силата на скока и забавянията при промяна на действията трябва да бъдат добре избрани, защото те дават усещането за масата и подвижността на героя. За да ги анализираме, нека обърнем внимание на Маделин от играта Celeste. Тя е един от най-добрите примери за реалистични и удобни за играчите движения.

Тичането може да се раздели на три части - ускорение, максимална скорост и забавяне. Ускорението е времето, което е необходимо на героя, след натискането на бутон за движение, да стигне от състояние на покой до максималната си скорост. Забавянето е времето, което му отнема, за да се върне в покой,

след като бутона е бил отпуснат. Времето, което е необходимо на Маделин за ускорение или забавяне, е достатъчно дълго, че тя да се чувства като човек, но достатъчно кратко, че почти мигновено да направи това, което играчът очаква от нея. Максималната скорост трябва да съответства на образа на героя, за да не изглежда, че се движи прекалено бързо или бавно в сравнение с останалите обекти от света.

Скачането също се разделя на три фази - издигане, задържане и падане. Маделин има кратко издигане и падане, с което тя изглежда реалистично, без да се рее във въздуха. В комбинация с това, тя има достатъчно дълго задържане във въздуха, даващо на играча контрол върху точните движения на Маделин и това къде, как и кога ще отиде тя. Силата на скока също трябва да бъде съгласувано с образа на героя. В зависимост от нея, героят може да изглежда доста силен или лек, ако може да скочи на голяма височина, или обратното при сравнително малък скок. Височината, на която може да скочи Маделин, се равнява на три пъти нейната собствена. [12]

2.1.2. Показване на логически пъзел и възможност да бъде решен.

Когато играчът влезе в контакт с врата, която представлява изход от настоящото ниво, може да бъде натиснат бутона *Space*, за да взаимодейства с нея. Ако тя е затворена, на екрана се изобразява загадка, която трябва да бъде решена. В това състояние, играта позволява две действия - подаване на отговор и връщане към картата.

Отговарянето става посредством клавишите с цифри за въвеждането им и *Backspace* за изтриване на въведена цифра. Проверка за верен отговор се прави посредством клавишът *Enter*. Ако въведе правилен отговор, вратата се отваря, а героят бива върнат на картата с възможност да отиде и да отключи друга врата или да мине през току-що отворената такава.

Ако играчът не може да въведе отговора, той може да се върне на картата, използвайки бутона *Esc* (*Escape*), което му дава възможност да отиде до друга врата или да търси подсказки.

2.1.3. Възможност за преминаване на следващо ниво при успешно решаване на пъзела.

Когато играчът даде правилен отговор на загадка и съответната врата бъде отворена, той може отново да използва клавиша *Space*, за да мине през нея. Така той ще бъде отведен на следващото ниво, с което тя е свързана.

Вратите може да водят към вече посетено ниво. При зареждането на нивото, ако играчът е решил загадката на някоя врата, тя се визуализира като отворена и играчът може да мине директно през нея.

2.2. Използвани технологии

2.2.1. Rust

Rust е модерен език за програмиране, който е бърз и предоставя безопасен достъп до паметта. Всички видове шаблони (generics) се отразяват само върху времето за компилация, което запазва времето за изпълнение възможно най-кратко.

Rust предотвратява изтичането на памет (memory leaks) и използване и достъпване на вече несъществуващи референции, благодарение на своята система за притежание (Ownership system) и своя валидатор на заеманията (Borrow checker). Другите езици предоставят система, която освобождава вече неизползваната памет (Garbage collector), или оставят програмистите ръчно да се грижат за заделянето и освобождаването на паметта.

Моделът за притежание на Rust използва три правила за безопасно използване на паметта. Първото е, че всяка стойност има собственик. Второто - не може да съществува повече от един собственик на дадена информация в даден момент от времето. Последното правило гласи, че, когато собственикът стане недостъпен, информацията става невалидна и недостъпна.

Но понякога ни се налага да извършим операция върху променлива без да вземаме собствеността върху нея. В такъв случай се използват референции, които пазят адреса към информацията, която искаме да достъпим. Проверка за валидността на референциите се извършва от валидатора на заемания. Живот на променлива (lifetime) представлява периода, в който имаме достъп до нея. Това, което валидаторът прави, е да подсигури, че животът на променливата, която искаме да достъпим, е по-дълъг от този на референцията към нея. В противен случай, кодът няма да бъде компилиран.

Освен това, има правила относно съществуващите по едно и също време референции. Те трябва да бъдат неограничен брой референции за четене или само една за писане. Когато имаме някакви данни, ние не очакваме те да се променят неочеквано,

затова, докато имаме референция за четене, не бива да имаме такава за писане. От друга страна, четенето не пречи на друга част от кода да чете съдържанието на дадения адрес, което позволява няколко референции за четене по едно и също време.

В Rust идеята за липса на стойност (`null`) липсва, което намалява сложността на кода, заради липсата на множество проверки, които биха се случили по време на изпълнението на програмата. По този начин си подсигуряваме присъствието на обект, когато се опитваме да извършим дадена операция, и не губим време в проверки дали обектът съществува, защото това е подсигурено от компилатора. За подобно поведение в кода, когато не знаем дали даден ресурс съществува или не, стандартната библиотека предоставя изброяващия тип (`enum`) *Option*. Той съдържа два варианта - *Some*, когато данните съществуват, като те се подават като стойност на варианта, и *None*, когато не съществуват. *Option::None* е еквивалентът на *null*, *nil*, *None* в други езици за програмиране.

Rust има различен начин и за справяне с грешките. В езика съществуват два типа грешки - такива, с които можем да се справим, и такива, с които не можем. Тези, с които не можем да се справим, приключват изпълнението на програмата, а процесът се нарича паникьосване (използва се макроса *panic!*). Тези, с които можем да се справим, не се хвърлят директно в кода, защото по този начин не бихме могли да знаем кога и къде има възникнала грешка, т.е. това се разбира по време на изпълнението на програмата. Вместо това, стандартната библиотека предоставя изброяващия тип *Result*, който може да съдържа необходима

стойност (в *Ok* варианта), която да се върне при отсъствие на грешка, или възникналата при операция грешка (в *Err* варианта). По този начин още по време на компилация Rust може да провери типовете на променливите и дали обработваме случаите, при които възниква грешка.

Rust има стриктен компилатор - при наличие на дори една грешка, кодът не се компилира. Ако имаме разминаване в типовете на данните, не подсигурим обработване на грешка или има проблем с живота на някоя референция, кодът не се компилира. Това подсигурява, че не можем да получим изпълним файл, който има потенциални проблеми, които компилаторът е можел да предотврати - достъп до недостъпна или освободена памет, възникване на грешка, липса на обект.

Липсата на принципа за наследяване, който присъства в обектно ориентираното програмиране, налага редица добри практики при изграждането на игри. Вместо да се изгражда „X е Y“ връзка между обектите, каквато присъства при наследяването, обектите биват изградени от по-малки градивни единици, използва се т. нар. композиция. Например, вместо „играчът е живо човек“, в който играчът наследява цялата функционалност, която човекът има, се прави обект за играча, който има различни характеристики като „може да се движи“, „може да събира предмети“, „може да говори“. Това позволява много по-ефективно преизползване на код, понеже други обекти също биха могли да ги използват.

Езикът е доста нов и модерен, което позволява различни добри практики и принципи да бъдат част от самия му дизайн. Като вградени в езика инструменти, те го правят удобен за

разработването на различни видове програми. Те се компилират, а една от опциите за изходен код е WebAssembly (WASM), което означава, че на Rust могат да се пишат различни програми, които да се изпълняват в уеб браузърите.

Езикът е силно типизиран, което означава, че всяка променлива има тип, който не може да се променя, през цялото изпълнение на програмата. Rust има доста развит компилатор. Програмистът не е задължен да заявява типа на всяка променлива, защото компилаторът може да го изведе от контекста, в който се използва, и операциите, които се извършват с нея. Тази функционалност се нарича „type inference“. [13]

2.2.2. Bevy

Bevy е ято птици. Но също е и библиотека, предоставяща инструменти за разработване на игри. Вместо обичайното досега структуриране на кода така, че всеки обект да си има собствен метод за обновяване, който се вика на всеки кадър, Bevy използва структурата „Единица, компонент, система“ (ECS - Entity, component, system).

„Единица“ е всеки обект в играта - всяко блокче земя, всеки герой, всеки предмет. Представлява техния идентификационен номер и се записва като 16-битово число.

„Компонент“ е информацията, която се свързва с даден обект. Представлява Rust структура и се добавя към обект посредством идентификационният му номер.

„Система“ е последователност от операции, които се извършват всеки кадър. Като част от кода, това са Rust функции,

чиито параметри са обектите, които искаме да променим. Те се пускат на всеки кадър, но може да се конфигурира.

Този начин на структуриране на обектите предоставя различен начин на подреждането им в паметта, което води до по-бързото и по-ефективно изпълнение на програмата.

Този начин на структуриране на кода осигурява логическо взаимодействие между обекти да се намира на едно място (системите). Но тъй като различните системи могат да използват даден обект, промените върху обектите не се намират на едно и също място, което прави откриването и отстраняването на проблеми свързани с конкретен обект по-трудно.

Bevy е в доста ранен стадий от своето развитие и предоставеният API подлежи на промени. Също така липсват някои важни функционалности. По тези причини в хранилището си в GitHub, екипът, който го разработва, моли да не се създават големи и сериозни проекти. [1, 2, 3, 11, 15]

2.2.3. Bevy Rapier

Bevy Rapier е пакет, разработен за Bevy, с помощта на който можем да добавяме и симулираме физични величини и взаимодействия между обектите в играта. Можем да го използваме за добавяне на гравитация, засичане на сблъсъци между обектите или за ограничаване на движението на един обект спрямо друг.

Bevy Rapier е бесплатна библиотека с отворен код, базирана на Rapier и разработена за приложения, направени с Bevy. Стреми се да бъде бърз, стабилен и да работи на всички операционни системи, предоставяйки физичните изчисления в реално време. [7, 8, 9, 10]

2.2.4. Bevy ECS LDtk

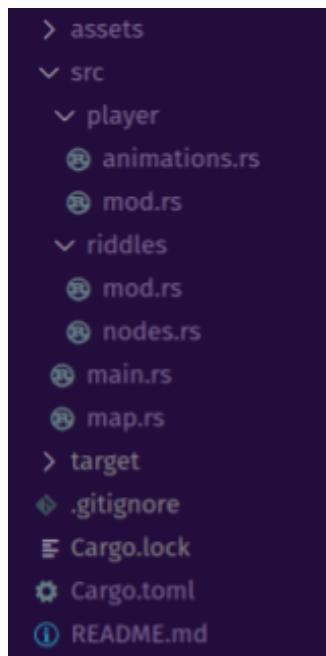
Bevy ECS LDtk е пакет, осъществяващ връзката между игра, написана с Bevy, и ldtk файл, получен, използвайки приложението Level Designer toolkit (LDtk). Това приложение позволява създаването на двуизмерни карти и добавянето на някакви обекти като земя или граница на ниво, като тази информация се записва във файл с разширение ldtk и формат JSON (JavaScript Object Notation). Форматът може да бъде променен през настройките на програмата.

Bevy ECS LDtk зарежда картата на дадено ниво визуално по подразбиране. За използване на обектите, описани в ldtk файла, на дадено ниво и обектите в него, трябва да дефинираме Rust структура, която да описва обекта, който искаме да заредим. Пакетът ни предоставя макроса *LdtkEntity*, което използваме като имплементираме предоставената от стандартната библиотека Rust характеристика (*trait*) *From* с него (*From<LdtkEntity>*). След това трябва да регистрираме обектите от картата като обекти от тази структура. Имплементираната от нас функция *from* действа като конструктор за тези обекти. [4, 5, 6]

ТРЕТА ГЛАВА

Постижения на проекта

3.1. Структура на проекта



фиг. 3.1

На фиг. 3.1 е показана структурата на проекта. Той се състои от директориите *assets*, *src*, и *target* и файловете *.gitignore*, *Cargo.lock*, *Cargo.toml* и *README.md*.

Директорията *assets* се засича автоматично от Bevy и в нея се намират всички допълнителни файлове, които са необходими на играта, за да функционира - всички изображения, аудио файлове, шрифтове, както и картата, която се зарежда чрез Bevy ECS LDtk.

Директорията *src* съдържа целият изходен код на проекта. Той е разделен на модули и подмодули, като съдържа и *main.rs* - входната точка на програмата.

Когато проектът бъде пуснат, системата за управление на проекти на Rust - *Cargo* - използва съдържанието на *Cargo.toml*

файла, в който са описани всички библиотеки, от които зависи проекта (*dependencies*), и генерира файла *Cargo.lock* с информация за използваните версии на тези пакети в текущата версия на изпълнимата програма. След това той взима файловете в *src* директорията и ги компилира, като резултатът се намира в директорията *target*, заедно с други генериирани файлове.

Останалите два файла от проекта - *.gitignore* и *README.md* - са свързани с хранилището на проекта - GitHub. Файлът *.gitignore* съдържа имената на всички файлове, които не трябва да бъдат качени в хранилището. Такива са всички автоматично генериирани генериирани файлове, тъй като те ще се появят при пускането на проекта. *README.md* съдържа кратко описание за съдържанието на хранилището, като GitHub автоматично го визуализира на уеб страницата на проекта.

3.2. Начало на програмата

```
fn main() {
    App::new() App
        .add_plugins(group: DefaultPlugins) &mut App
        .add_state(initial: GameState::LevelLoading) &mut App
        .add_plugin(RapierPhysicsPlugin::<NoUserData>::default()) &mut App
        .insert_resource(RapierConfiguration {
            gravity: Vec2::new(x: 0.0, y: -400.0),
            ..Default::default()
        }) &mut App
        .add_plugin(LdtkPlugin) &mut App
        .add_startup_system(setup_system) &mut App
        .add_plugin(PlayerPlugin) &mut App
        .add_plugin(MapPlugin) &mut App
        .add_plugin(RiddlesPlugin) &mut App
        .run();
}
```

фиг. 3.2: *src/main.rs*

Програмата започва своето изпълнение от функцията *main*, намираща се във файла *main.rs*. Както е показано на фиг. 3.2,

създаваме нова инстанция на структурата *App*, към която добавяме различни допълнения (Plugin) и системи. Допълненията са специално дефинирани структури, които имплементират характеристиката *Plugin*. По този начин те имат достъп до инстанцията на *App* структурата, която представлява нашата игра, и това ни позволява да разделим кода на модули. В случая добавяме допълненията предоставени от пакетите Bevy (*DefaultPlugins*), Bevy Rapier (*RapierPhysicsPlugin*), Bevy ECS LDtk (*LdtkPlugin*), както и допълненията, дефинирани в този проект - *PlayerPlugin*, *MapPlugin* и *RiddlesPlugin*.

Също така могат да се добавят и ресурси - инстанции на структури, които са достъпни в системите. Разликата им с компонентите е, че те не са свързани с конкретен обект, а са различни видове настройки, които влияят на целия свят. На фиг. 3.2 е показано как добавяме *RapierConfiguration* ресурса, като променяме гравитацията.

```
fn setup_system(mut commands: Commands) {
    commands.spawn_bundle(Camera2dBundle::default());
}
```

фиг. 3.3: src/main.rs

На фиг. 3.3 е показана и системата за подготовка на играта, която добавяме на фиг. 3.2. В нея взимаме достъп до света, използвайки *Commands*, и добавяме едва двуизмерна камера, като запазваме настройките по подразбиране.

3.3. Създаване на допълнение (Plugin)

```
pub struct MapPlugin;

impl Plugin for MapPlugin {
    fn build(&self, app: &mut App) {
        app.add_startup_system(map_setup_system) &mut App
            .insert_resource(CurrentLevel::from(STARTING_LEVEL)) &mut App
            .add_system_set(
                SystemSet::on_enter(GameState::LevelLoading).with_system(level_loading_system),
            ) &mut App
            .add_system_set(
                SystemSet::on_update(GameState::LevelLoading).with_system(level_loaded_system),
            ) &mut App
            .add_system_set(
                SystemSet::on_exit(GameState::LevelLoading).with_system(normalize_font_system),
            ) &mut App
            .add_system_set(SystemSet::on_exit(GameState::LevelLoading).with_system(center_map)) &mut App
            .register_ldtk_entity::<GroundTile>(entity_identifier: "Ground") &mut App
            .register_ldtk_entity::<GroundTile>(entity_identifier: "LevelBorder") &mut App
            .register_ldtk_entity::<Door>(entity_identifier: "Door") &mut App
            .register_ldtk_entity::<BoxTile>(entity_identifier: "Box") &mut App
            .register_ldtk_entity::<TextSignBundle>(entity_identifier: "TextSign");
    }
}
```

фиг. 3.4: src/map.rs

На фиг. 3.4 е показано как се дефинира допълнение. Първо създаваме структура, която правим публична с ключовата дума *pub*, за да можем да я използваме извън този файл. След това имплементираме характеристиката *Plugin*, която включва един метод - *build*. Той приема един параметър, представляващ приложението, към което ще добавяме системи и ресурси.

3.4. Състояния на играта

На фиг. 3.5 е показан изброяващият тип *GameState*. Той показва състоянията, в които може да се намира играта - Зареждане на ниво, изследване на картата или решавана не загадка.

```
#[derive(Clone, Debug, Eq, PartialEq, Hash)]
5 implementations
enum GameState {
    MapExploring,
    RiddleSolving,
    LevelLoading,
}
```

фиг. 3.5: src/main.rs

Той може да бъде използван като критерий за изпълнението на дадена система. Тя може да се изпълни когато се влезе или излезе от дадено състояние или докато играта се намира в дадено състояние. На фиг. 3.4 може да се види как се използва състоянието на играта като определящ фактор кога да се изпълнят различните системи. Например, системата, зареждаща нивото трябва да се изпълни веднъж, когато влезем в състояние на зареждане на ниво.

3.5. Зареждане на картата

```
fn map_setup_system(mut commands: Commands, asset_server: Res<AssetServer>) {
    commands.spawn_bundle(LdtkWorldBundle {
        ldtk_handle: asset_server.load(path: "map.ldtk"),
        ..default()
    });
}
```

фиг. 3.6: src/map.rs

На фиг. 3.6 е показана системата, която зарежда картата в играта. За да заредим картата, трябва да използваме структурата *LdtkWorldBundle*, като за *ldtk_handle* подаваме резултата от метода *load* на ресурса *AssetServer*. Той взима пътя към файла, в който се намират нивата и връща заредената в програмата негова версия. Тъй като по подразбиране *AssetServer* търси файловете в директорията *assets*, ние подаваме само „map.ldtk“.

3.6. Зареждане на ниво

```
fn level_loading_system(
    current_level: Res<CurrentLevel>,
    mut level_set_info: Query<&mut LevelSet>,
) {
    let mut level_set: Mut<LevelSet> = level_set_info.single_mut();
    level_set.iids.clear();
    level_set.iids.insert(current_level.clone());
}

fn center_map(
    levels: Res<Assets<LdtkLevel>>,
    mut map_info: Query<(&Handle<LdtkLevel>, &mut Transform)>,
) {
    let (handle: &Handle<LdtkLevel>, mut map: Mut<Transform>) = map_info.single_mut();
    let level: &LdtkLevel = levels.get(handle).unwrap();

    map.translation.x = -level.level.px_wid as f32 / 2.0;
    map.translation.y = -level.level.px_hei as f32 / 2.0;
}
```

фиг. 3.7: src/map.rs

На фиг. 3.7 са показани системите за зареждане на ниво и за центриране на картата на екрана. Системата за зареждане на ниво приема два параметъра - ресурса *CurrentLevel*, който ние сме дефинирали, и всички заредени в момента нива. *Query* е инструментът, който използваме, когато искаме да вземем обекти на база на техните компоненти. То връща всички намерени обекти, отговарящи на зададените критерии. В случая го използваме, за да вземем множеството на заредените нива. Тъй като е гарантирано то да съществува, ние използваме метода *single_mut*, който ще накара програмата да се паникьоса, ако *Query* не съдържа точно един резултат. След това изчистваме множеството и зареждаме нивото, което се пази в *CurrentLevel*. На фиг. 3.8 е показано мястото, на което дефинираме този тип, като той е просто типът *String*, но с друго име. Също така е показана и дефиницията на константата *STARTING_LEVEL*, която използваме на фиг. 3.4 в комбинация със

системата *level_loading_system*, за да заредим първото ниво от играта.

```
const STARTING_LEVEL: &str = "6c6ef290-5110-11ed-90f2-ab2793fe3460";
const SMALL_TILE_SIZE: f32 = 16.0;
const LARGE_TILE_SIZE: f32 = 64.0;

pub type CurrentLevel = String;
```

фиг. 3.8: src/map.rs

На фиг. 3.7 също е показана и системата, която поставя зареденото ниво в центъра на екрана. Тази стъпка е необходима, тъй като различните нива могат да имат различни размери.

3.7. Зареждане на обект от картата

```
#[derive(Default, Bundle, LdtkEntity)]
3 implementations
struct LevelBorder {
    #[from_entity_instance]
    #[bundle]
    collider_bundle: ColliderBundle,
}
```

фиг. 3.9: src/map.rs

Когато искаме да заредим обект от картата, ние трябва да дефинираме две структури. За обяснение на процеса ще използваме границата на ниво, но процесът е аналогичен за останалите обекти. Първата структура, показана на фиг. 3.9, е обектът, който използваме при регистрацията на обекта, както е показано на фиг. 3.4 с метода *register_ldtk_entity*. В тази структура можем да добавяме и полета, които не са свързани с информация от ldtk файла.

Втората структура има за цел да вземе информацията от картата и да я превърне в използваеми компоненти. На фиг. 3.10

можем да видим структурата *ColliderBundle*, за която имплементираме характеристиката *From<LdtkEntity>*, която се използва за създаването на обект от тип *ColliderBundle* с подаване на *LdtkEntity* обект.

```
#[derive(Default, Bundle)]
3 implementations
struct ColliderBundle {
    collider: Collider,
    rigid_body: RigidBody,
}

impl From<EntityInstance> for ColliderBundle {
    fn from(entity_instance: EntityInstance) -> Self {
        match entity_instance.identifier.as_ref() {
            "Ground" / "LevelBorder" => Self {
                collider: Collider::cuboid(hx: SMALL_TILE_SIZE / 2.0, hy: SMALL_TILE_SIZE / 2.0),
                rigid_body: RigidBody::Fixed,
            },
            "Door" / "Box" => Self {
                collider: Collider::cuboid(hx: LARGE_TILE_SIZE / 2.0, hy: LARGE_TILE_SIZE / 2.0),
                rigid_body: RigidBody::Fixed,
            },
            _ => Self::default(),
        }
    }
}
```

фиг. 3.10: src/map.rs

В случая, функцията *build* първо проверява какъв обект се опитваме да създадем, защото *ColliderBundle* се използва и в други структури в проекта. В зависимост от типа, задаваме *Collider* и *RigidBody* полетата, като за първото използваме дефинираните константи, показани на фиг. 3.8. Те са необходими, понеже зареждането на обектите задава компонента *Transform*. Компонентът отговаря за местоположението по двете оси, завъртането и мащабирането на обекта.

3.8. Движение на героя

На фиг. 3.11 е показана системата, отговорна за движението на героя. Тя приема 4 параметъра - информация за бутоните на

клавиатурата, контекста на библиотеката за физика, *Query*, с което да вземем обекта на играта, и *Query*, което да вземе всички обекти, при които можем да скачаме, когато сме върху тях.

```
fn player_movement_system(
    keyboard_input: Res<Input<KeyCode>>,
    rapier_context: Res<RapierContext>,
    mut player_info: Query<Entity, &mut Velocity, &mut TextureAtlasSprite>, With<Player>,
    tile_info: Query<Entity, With<Ground>>,
) {
    let (player: Entity, mut velocity: Mut<Velocity>, mut sprite: Mut<TextureAtlasSprite>) = player_info.single_mut();
    let up: bool = keyboard_input.any_pressed(inputs: [KeyCode::Up, KeyCode::W]);
    let left: bool = keyboard_input.any_pressed(inputs: [KeyCode::Left, KeyCode::A]);
    let right: bool = keyboard_input.any_pressed(inputs: [KeyCode::Right, KeyCode::D]);

    velocity.linvel.x = if left {
        sprite.flip_x = true;
        -RUN_POWER
    } else if right {
        sprite.flip_x = false;
        RUN_POWER
    } else {
        0.0
    };

    if up {
        for tile: Entity in tile_info.iter() {
            if let Some(contact_pair: ContactPairView) = rapier_context.contact_pair(collider1: player, collider2: tile) {
                for manifold: ContactManifoldView in contact_pair.manifolds() {
                    let first_entity: Entity = manifold.ContactManifoldView
                        .rigid_body1() Option<Entity>
                        .expect(msg: "An entity is expected when collision is detected!");
                    if (first_entity == player && manifold.normal().y == -1.0)
                        // manifold.normal().y == 1.0
                    {
                        velocity.linvel.y = JUMP_POWER;
                        return;
                    }
                }
            }
        }
    }
}
```

фиг. 3.11: src/player/mod.rs

Първото нещо, което системата прави е да извлече обекта на играта в променлива и да провери дали бутоните за движение са натиснати. Стрелките за движение наляво и надясно, заедно с бутоните *A* и *D* се проверяват за хоризонтално движение, а стрелката за нагоре и клавишът *W* показват дали трябва да се извърши скок.

Първо се грижим за хоризонталното движение на героя. Ако трябва да се движим наляво, обръщаме героя така, че да гледа наляво, и слагаме скоростта му да бъде в същата посока и със сила

равна на константата за движение *RUN_POWER*, чиято дефиниция е показана на фиг. 3.12.

```
const PLAYER_WIDTH: f32 = 60.0;
const PLAYER_HEIGHT: f32 = 110.0;
const JUMP_POWER: f32 = 250.0;
const RUN_POWER: f32 = 100.0;
```

фиг. 3.12: src/player/mod.rs

Следва проверка за скок. Ако човекът се опитва да скочи, проверяваме дали героят се намира върху обект, от който има право на скок. За тази цел минаваме през всички обекти, които ни позволяват това действие, и, използвайки контекста на Bevy Rapier, проверяваме дали има сблъсък между обекта на героя и някой от тези обекти така. Сблъсъкът трябва да е между долната страна на героя и горната на другия обект, в противен случай героят не се намира върху обекта и това не определя дали има право на скок.

Ако всички условия са изпълнени - играчът се намира върху обект, от който може да скочи, и се опитва да скочи чрез натискане на един от съответните бутони - правим скоростта на играла нагоре равна на константата *JUMP_POWER*, чиято дефиниция също е показана на фиг. 3.12.

3.9. Анимиране на героя

За анимацията на героя има структура, съдържаща индексите на изображенията в различните състояния, като анимирането представлява една система, която да сменя изображението на героя в текущото действие (тичане, скачане, падане), и други системи, които да сменят текущото действие, когато се изпълнят дадени изисквания.

```

fn animate_player_system(
    time: Res<Time>,
    mut animation_info: Query<(&mut TextureAtlasSprite, &mut AnimationInfo), With<Player>>,
) {
    let (mut sprite: Mut<TextureAtlasSprite>, mut animation_info: Mut<AnimationInfo>) = animation_info.single_mut();
    if animation_info.timer.tick(time.delta()).just_finished() {
        animation_info.index = (animation_info.index + 1) % animation_info.current_animation.len();
        sprite.index = animation_info.current_animation[animation_info.index];
    }
}

```

фиг. 3.13: src/player/animations.rs

На фиг. 3.12 е показана системата, която анимира героя. Тя се нуждае от глобалното време на света, показано чрез ресурса *Time*, а също и от компонента за анимация на героя, за взимането на който се използва *Query*. Всяка анимация си има таймер за това кога да се смени кадърът ѝ. За тази цел се прави проверка дали таймерът е изтекъл. Ако е, кадърът се сменя, иначе си остава същият. На фиг. 3.14 е показана структурата, пазеща информацията за анимациите на героя и изброяващият тип за видовете анимации.

```

#[derive(Default, Eq, PartialEq, Hash)]

pub enum AnimationType {
    #[default]
    IDLE,
    RUN,
    JUMP,
    FALL,
}

You, 2 weeks ago | 1 author (You)
#[derive(Default, Component)]
3 implementations
pub struct AnimationInfo {
    animations: HashMap<AnimationType, Vec<u32>>,
    current_animation_type: AnimationType,
    current_animation: Vec<u32>,
    index: u32,
    timer: Timer,
}

```

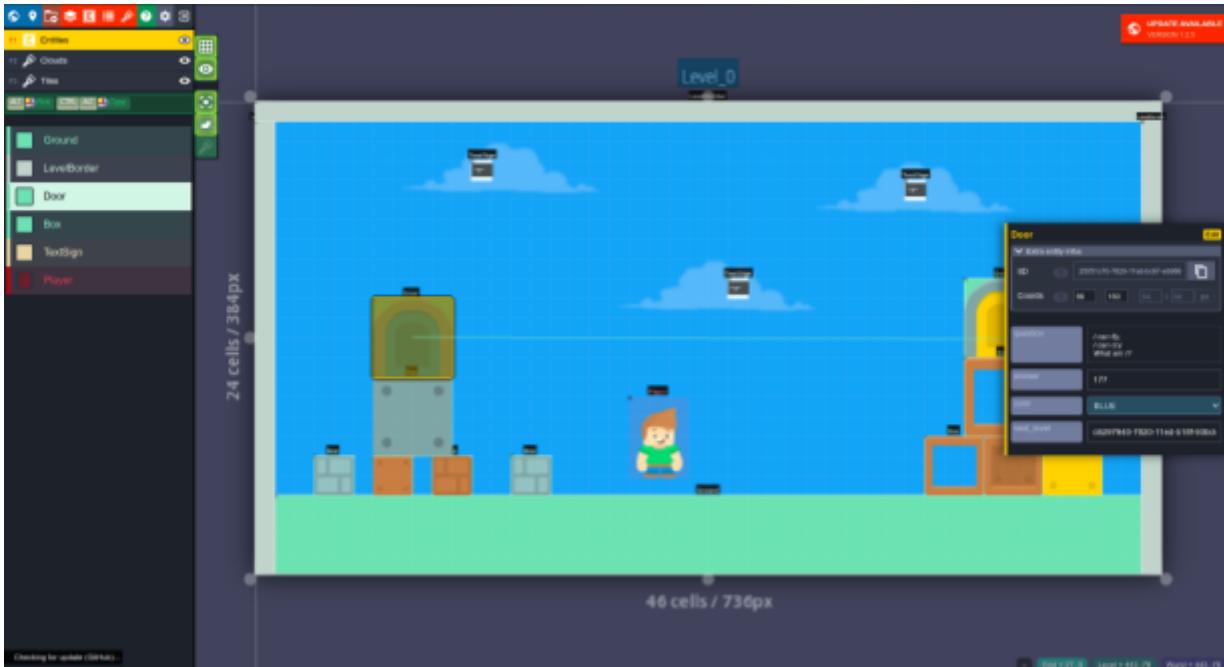
фиг. 3.14: src/player/animations.rs

На фиг. 3.15 е показана системата за преминаване в скачаща анимация. Останалите системи за смяна на анимацията са аналогични. Първо проверяваме дали текущата анимация на героя е такава, че да може да се премине в съответната анимация. В случая се опитваме да преминем в скок. Това може да стане от всяка друга анимация, затова проверяваме дали сме в скок в момента. Ако сме прекратяваме системата, защото, ако вече скачаме, няма нужда да проверяваме дали можем да преминем в скачаща анимация. Ако продължим изпълнението на системата, проверяваме дали героят докосва друг обект отдолу и дали се движи нагоре. Ако не докосва нищо и се движи нагоре, то тогава можем да сменим настоящата анимация. Това се случва с метода `set_animation`, който е дефиниран на създадената от нас структура `AnimationInfo`, която пази информацията за анимациите. Ако поне едно от условията не е изпълнено, анимацията си остава същата, като просто прекратяваме изпълнението на функцията.

```
fn jump_animation_trigger_system(
    mut events: EventReader<CollisionEvent>,
    mut animation_info: Query<(&mut AnimationInfo, &Velocity), With<Player>>,
) {
    let (mut animation_info: Mut<AnimationInfo>, velocity: &Velocity) = animation_info.single_mut();
    if animation_info.current_animation_type == AnimationType::JUMP {
        return;
    }
    for event: &CollisionEvent in events.iter() {
        if let CollisionEvent::Stopped(_, _, flag: &CollisionEventFlags) = event {
            if !flag.is_empty() {
                continue;
            }
            if velocity.linvel.y < 0.0 {
                continue;
            }
            animation_info.set_animation(AnimationType::JUMP);
            return;
        }
    }
}
```

фиг. 3.15: src/player/animations.rs

3.10. Зареждане на вратите с гатанки



фиг. 3.16

В програмата Level Designer toolkit, показана на фиг. 3.16, можем да видим как на обектите от тип Door трябва да се зададат въпрос и отговор. Тази информация се зарежда в кода през *build* функцията от характеристиката *From<LdkEntity>*, имплементирана за конкретната структура, чрез която зареждаме вратите с гатанки.

```
impl From<EntityInstance> for RiddleInfo {
    fn from(entity_instance: EntityInstance) -> Self {
        let fields = HashMap::from_iter(entity_instance.field_instances.iter().map(|field| {
            (
                field.identifier.clone(),
                match field.value.clone() {
                    FieldValue::String(Some(value)) => value,
                    _ => "".to_string(),
                },
            )
        }));
        Self {
            question: fields
                .get("question")
                .expect("A question is required for a riddle!")
                .clone(),
            answer: fields
                .get("answer")
                .expect("An answer is required for a riddle!")
                .clone(),
            next_level: fields
                .get("next_level")
                .expect("A next level is required for a riddle!")
                .clone(),
            ..Default::default()
        }
    }
}
```

фиг. 3.17: src/riddles/mod.rs

На фиг. 3.17 е представен кодът, който зарежда тази информация. Всички налични променливи, подадени от файла, се зареждат в *HashMap*, като след това просто търсим необходимите ни такива по ключ, който е тяхното име - в случая „question“ и „answer“.

3.11. Създаване на екраните с въпроси

На фиг. 3.18 е показано съдържанието на *src/riddles/nodes.rs*. За всеки елемент от экрана с гатанка има функция, което да го създаде. По този начин те са като конфигурация за самия елемент.

Един экран с гатанка се състои от *NodeBundle*, който е визуалния тип елемент (UI node) и който да държи останалите такива елементи. Текстът на гатанката е от тип *TextBundle*, както и трите позиции за отговор, които обаче се намират в друг *NodeBundle*, за да бъдат позиционирани на един ред и да имат разстояние помежду си.

Визуалният тип елементи имат допълнителен компонент *Style*, който описва как те да изглеждат на экрана. Те определят цвета на фона, цвета на текста, разстоянието помежду им, размера и позицията им и други параметри, отговарящи за репрезентацията им на экрана.

```

pub fn root_node() -> NodeBundle {
    NodeBundle {
        style: Style {
            display: Display::None,
            flex_direction: FlexDirection::ColumnReverse,
            justify_content: JustifyContent::SpaceAround,
            align_items: AlignItems::Center,
            size: Size::new(width: Val::Percent(100.0), height: Val::Percent(100.0)),
            ..Default::default()
        },
        color: UIColor::from(Color::rgb(r: 0.5, g: 0.5, b: 0.85)),
        visibility: Visibility { is_visible: false },
        ..Default::default()
    }
}

pub fn question_text(asset_server: &Res<AssetServer>, question: &String) -> TextBundle {
    TextBundle {
        text: Text::from_section(
            value: question,
            style: TextStyle {
                font: asset_server.load(path: "fonts/MontserratAlternates-MediumItalic.ttf"),
                font_size: 60.0,
                color: Color::WHITE,
            },
        ) Text
        .with_alignment(TextAlignment::CENTER),
        ..Default::default()
    }
}

pub fn answer_container() -> NodeBundle {
    NodeBundle {
        color: UIColor::from(Color::NONE),
        style: Style {
            justify_content: JustifyContent::SpaceAround,
            min_size: Size::new(width: Val::Percent(30.0), height: Val::Auto),
            ..Default::default()
        },
        ..Default::default()
    }
}

pub fn answer_position(asset_server: &Res<AssetServer>, color: Color) -> TextBundle {
    TextBundle {
        text: Text::from_section(
            value: "_".to_string(),
            style: TextStyle {
                font: asset_server.load(path: "fonts/MontserratAlternates-MediumItalic.ttf"),
                font_size: 60.0,
                color,
            },
        ),
        ..Default::default()
    }
}

```

Фиг. 3.18: src/riddles/nodes.rs

3.12. Взаимодействие с еcranите с въпроси

Когато играчът се опита да отвори врата, еcranът с гатанка се отваря. В този момент той може да се опита да отговори. Това се случва чрез системата *answering_riddle_system*. На фиг. 3.19 е показан кодът на системата.

```
fn answering_riddle_system(
    mut input: EventReader<ReceivedCharacter>,
    mut container_info: Query<(&mut AnswerContainer, &ComputedVisibility)>,
    mut answer_nodes: Query<(&mut Text, &ComputedVisibility, &Answer)>,
) {
    for character: &ReceivedCharacter in input.iter() {
        if !('0'..='9').contains(item: &character.char) {
            continue;
        }
        let (mut container: Mut<AnswerContainer>, _) = container_info.Query<(&mut AnswerContainer, ...)>
            .iter_mut() QueryIter<(&mut AnswerContainer, ...), ...>
            .find(/(_, visibility: &&ComputedVisibility) / visibility.is_visible()) Option<(Mut<AnswerContainer>, ...)>
            .expect(msg: "A visible container is expected while this system is running!");
        let (mut answer: Mut<Text>, _, _) = answer_nodes.Query<(&mut Text, &ComputedVisibility, ...)>
            .iter_mut() QueryIter<(&mut Text, &ComputedVisibility, ...), ...>
            .find(/(_, visibility: &&ComputedVisibility, answer: &&Answer) / visibility.is_visible()
                && answer.position == container.index) Option<(Mut<Text>, &ComputedVisibility, ...)>
            .expect(msg: "The container is expected to have answer positions and the container's index is always valid!");
        answer.sections[0].value = character.char.to_string();
        container.index = (container.index + 1) % container.answer_length;
    }
}
```

фиг. 3.19: src/riddles/mod.rs

Тя приема 3 параметър - събития, генериирани при натискането на бутон, *Query* за елементите, които държат трите позиции за отговор, и самите позиции за отговор. Системата проверява всички генериирани от клавиатурата събития, като продължава да обработва само тези, които са генериирани от цифрови клавиши. При наличие на подобно събитие, то се слага на настоящата позиция от отговора и индексът за това, коя е тя, се премества с едно надясно, като се връща в началото при достигане на края.

По подобен начин работи и системата за изтриване на въведен отговор, но тя работи само при натискането на бутона *Backspace*, като връща текущата позиция наляво преди да извърши изтриването. По този начин изтрива последно въведената цифра.

Тази система е показана на фиг. 3.20. Тя не връща индекса за текущата позиция в края на отговора при достигане на неговото началото.

```
fn delete_digit_system(
    keyboard_input: Res<Input<KeyCode>>,
    mut container_info: Query<(&mut AnswerContainer, &ComputedVisibility)>,
    mut answer_nodes: Query<(&mut Text, &ComputedVisibility, &Answer)>,
) {
    if !keyboard_input.just_pressed(KeyCode::Back) {
        return;
    }
    let (mut container: Mut<AnswerContainer>, _) = container_info.query_mut();
    .iter_mut() QueryIter<(&mut AnswerContainer, ...), ->
    .find(|(_, visibility: &ComputedVisibility)| visibility.is_visible()) Option<(Mut<AnswerContainer>, ...)>
    .expect(msg: "A visible container is expected while this system is running!");
    if container.index == 0 {
        container.index = container.answer_length;
    }
    container.index -= 1;
    let (mut answer: Mut<Text>, _, _) = answer_nodes.query_mut();
    .iter_mut() QueryIter<(&mut Text, &ComputedVisibility, ...), ->
    .find(|(_, visibility: &ComputedVisibility, answer: &Answer)| visibility.is_visible()
        && answer.position == container.index) Option<(Mut<Text>, &ComputedVisibility, ...)>
    .expect(msg: "The container is expected to have answer positions and the container's index is always valid!");
    answer.sections[0].value = "_".to_string();
}
```

фиг. 3.20: src/riddles/mod.rs

Ако играчът не знае отговора на загадката, той може да затвори екрана със загадката, което да върне играта в режим на изследване на картата. Това се случва чрез натискането на бутона *Esc* (*Escape*), и за процеса се грижи системата *close_riddle_system*.

```
fn close_riddle_system(
    keyboard_input: Res<Input<KeyCode>>,
    mut state: ResMut<State<GameState>>,
    mut doors: Query<&mut RiddleInfo>,
    mut riddle_nodes: Query<(&mut Style, &mut Visibility), With<RiddleNode>>,
) {
    if keyboard_input.just_pressed(KeyCode::Escape) {
        let mut door: Mut<RiddleInfo> = doors.query_mut();
        .iter_mut() QueryIter<&mut RiddleInfo, ...>
        .find(|door: &Mut<RiddleInfo>| door.active) Option<Mut<RiddleInfo>>
        .expect(msg: "Exactly one active door is expected while this system is running!");
        door.active = false;
        let (mut riddle_style: Mut<Style>, mut riddle_visibility: Mut<Visibility>) = riddle_nodes.query_mut();
        .iter_mut() QueryIter<(&mut Style, &mut ...), ->
        .find(|(_, visibility: &Mut<Visibility>)| visibility.is_visible()) Option<(Mut<Style>, Mut<Visibility>)>
        .expect(msg: "Exactly one visible riddle node is expected while this system is running!");
        riddle_style.display = Display::None;
        riddle_visibility.is_visible = false;
        state.set(state::MapExploring).unwrap();
    }
}
```

фиг. 3.21: src/riddles/mod.rs

За да провери дали е въвел правилния отговор, играчът може да използва бутона *Enter*. Това ще задейства системата *correct_guess_system*, която прави проверката. Тя е показана на фиг. 3.22. При успешен опит, еcranът се затваря, играчът е върнат на картата, но вратата, която досега се е опитвал да отвори, е вече отворена.

```
fn correct_answer_system(
    mut commands: Commands,
    keyboard_input: Res<Input<KeyCode>>,
    mut answered_riddles: ResMut<AnsweredRiddles>,
    mut state: ResMut<State<GameState>>,
    mut doors: Query<(&mut RiddleInfo, &mut TextureAtlasSprite)>,
    answer_nodes: Queryy<(&Text, &ComputedVisibility, &Answer)>,
) {
    if keyboard_input.any_just_pressed(inputs: [KeyCode::Return, KeyCode::NumpadEnter]) {
        let mut answer_nodes: Vec<(usize, String)> = Vec::from_iter(
            answer_nodes.Query(&Text, &ComputedVisibility, ..)
                .iter() QueryIter<(&Text, &ComputedVisibility, ..), ..>
                .filter(|(_, visibility)| &&ComputedVisibility == visibility.is_visible()) impl Iterator<Item = (&Text, ..)>
                .map(|(text: &Text, .., pos: &Answer)| (pos.position, text.sections[0].value.clone())),
        );
        answer_nodes.sort_by_key(|(pos, _size, ..)| *pos);
        let answer: String = answer_nodes_vec<(usize, String)>
            .into_iter() IntoIter<(usize, String)>
            .map(|(_, value: String)| value) impl Iterator<Item = String>
            .collect();
        let (mut door: Mut<RiddleInfo>, mut sprite: Mut<TextureAtlasSprite>) = doors_query(&mut RiddleInfo, &mut ..)
            .iter_mut() QueryIter<(&mut RiddleInfo, ..), ..>
            .find(|(door: &Mut<RiddleInfo>, ..)| door.active) Option<Mut<RiddleInfo>, ..>
            .expect(msg: "Only one door should be active while answering a riddle!");
        if answer != door.answer {
            return;
        }
        answered_riddles.insert(door.question.clone());
        commandsCommands
            .entity(
                door.riddle Option<Entity>
                    .expect(msg: "The riddle entity is supposed to be set by the init_riddles_system!"),
            ) EntityCommands
            .despawn_recursive();
        sprite index = 75;
        door.active = false;
        state.set(state: GameState::MapExploring).unwrap();
    }
}
```

фиг. 3.22: src/riddles/mod.rs

Опитът за преминаване през вратата ще смени стойността на ресурса *CurrentLevel*, който е създаден на фиг. 3.4 във функцията *main*, и *level_loading_system* ще зареди следващото ниво, което е новата стойност.

ЧЕТВЪРТА ГЛАВА

Ръководство на потребителя

4.1. Инсталация

4.1.1. Необходими налични технологии

За да бъде инсталиран проектът, са необходими две технологии - Git и Rust. Всички останали необходими технологии са описани в *Cargo.toml* файла на проекта и се инсталират автоматично при първото пускане на проекта.

Възможно е проектът да бъде инсталиран без помощта на Git, но това има своите недостатъци. Те, както и самият процес, са обяснени в точка 4.1.5. от този документ.

4.1.2. Инсталиране на Git

Независимо от операционната система, може да бъде последван следният линк за инсталиране на Git - <https://git-scm.com/downloads>. Там трябва да се избере операционната система, за която да се свали инсталатор. След изпълнението му, Git трябва да бъде наличен в терминала.

При инсталация за Linux операционна система, може да се използва система за управление на пакети, каквато идва с всяка дистрибуция, за по-бърза инсталация на Git.

4.1.3. Инсталиране на Rust

Препоръчителният начин за инсталиране на компилатор и други инструменти за езика е да се изпълни следната команда в терминала:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Други начини за инсталација на компилатора и допълнителните инструменти инструменти може да се намерят на следния линк:

<https://forge.rust-lang.org/infra/other-installation-methods.html>

4.1.4. Инсталација на проекта (чрез Git)

Има две опции за инсталација на проекта през Git:

- чрез HTTPS линк:

git clone https://github.com/Stanislav04/Puzzle-up

- използвайки SSH ключ:

git clone git@github.com:Stanislav04/Puzzle-up

Независимо от команда, резултатът е нова директория, представляваща клонирания проект. Може да се влезе в нея със следната команда:

cd Puzzle-up

Проектът е готов за пускане, като стъпките са описани в 4.2. от този документ.

4.1.5. Инсталација на проекта (без използването на Git)

GitHub предоставя още една опция за сваляне на проекта. Можете да посетите следния линк и да натиснете бутона Code:

<https://github.com/Stanislav04/Puzzle-up>

След това да изберете опцията ZIP. Това ще свали архив, който трябва да разархивираме, за да получите локално копие на проекта. Този подход обаче има голям недостатък. Този начин на инсталација разделя локалното копие, което се сваля, от проекта в хранилището GitHub. Това означава, че при промяна на проекта,

трябва да изпълните всички стъпки, описани в тази точка, наново. Вместо това, пре инсталация чрез Git, можете просто да изпълните следната команда за актуализация на локалното копие на проекта:

```
git pull
```

4.2. Изпълнение на проекта

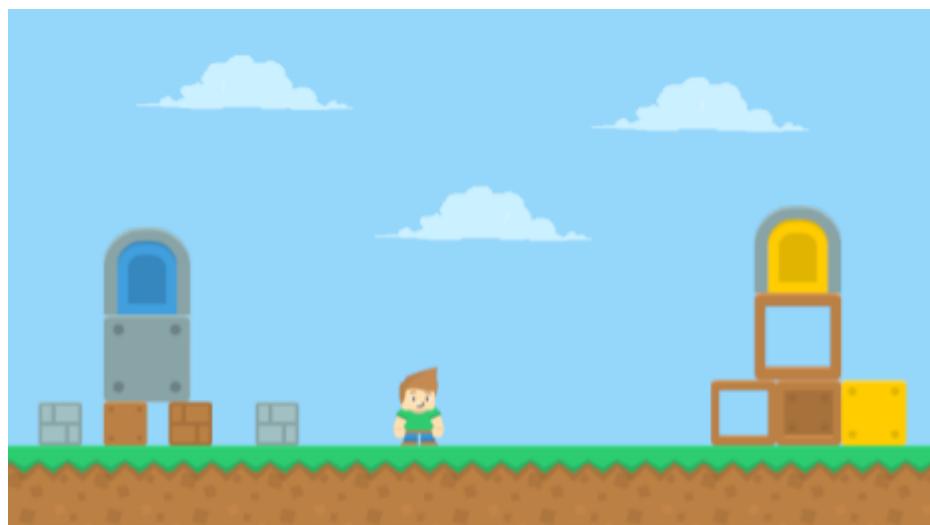
Когато свалите проекта и влезете в директорията му, трябва да изпълните следната програма, за да изпълните проекта:

```
cargo run
```

Първото изпълнение на тази команда след инсталацията на проекта ще свали всички необходими за неговото изпълнение пакети, описани в *Cargo.toml* файла на проекта.

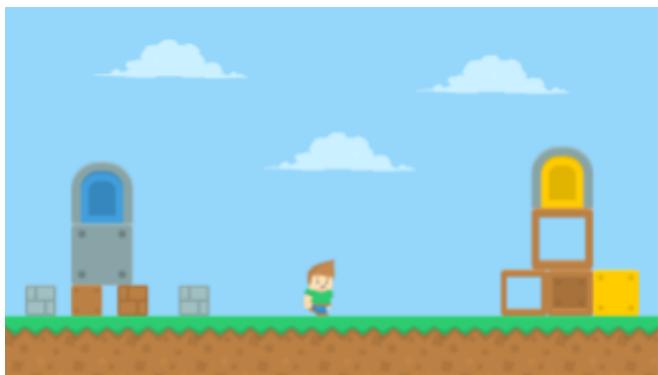
4.3. По време на изпълнението на програмата

След стартирането на програмата, играта ще се зареди и ще се появи показаното на фиг. 4.1. Това е началното състояние на играта и представлява първото ѝ ниво.

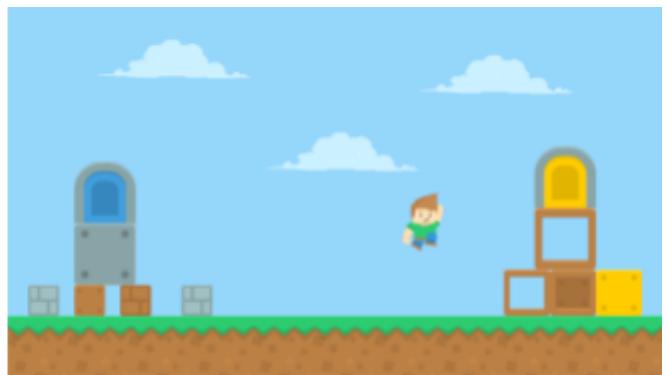


фиг. 4.1

4.3.1. Движение на героя

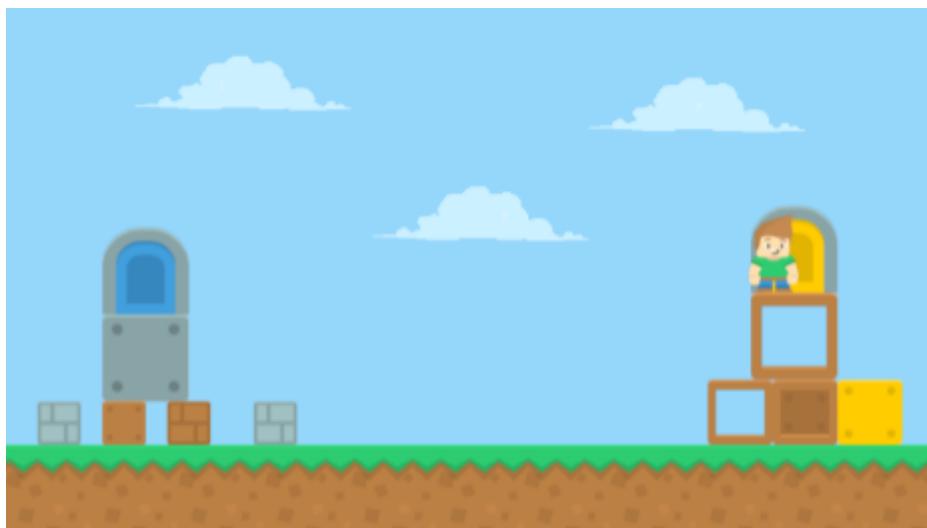


фиг. 3.2



фиг. 3.3

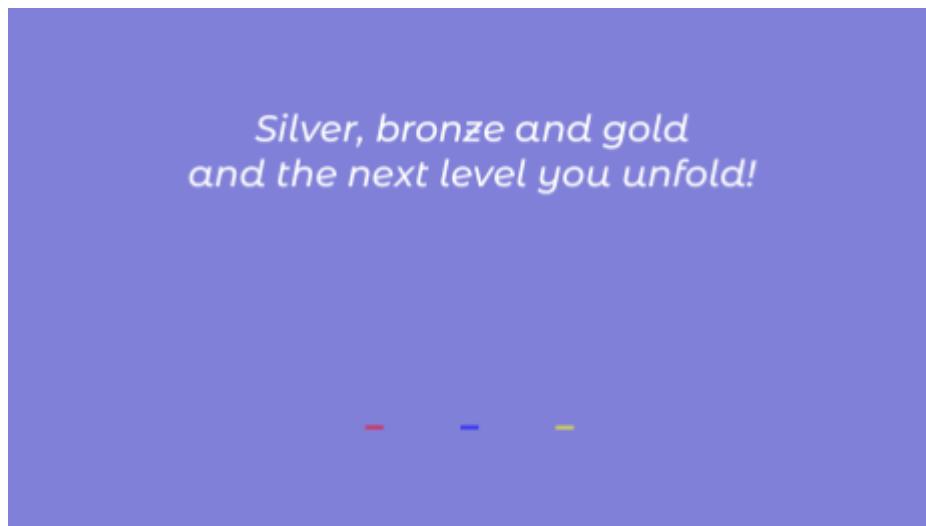
След това можем да отидем до жълтата врата. Когато тръгнем да се движим надясно, анимацията на героя се сменя, както е показано на фиг. 3.2. Също така се налага и да скочим няколко пъти, за да стигнем до върха на тази купчина от кутии. Когато се опитаме да скочим, героят променя анимацията си така, че да изглежда по същия начин като на фиг. 3.3. Когато героят ни е пред вратата като на фиг. 3.4, можем да натиснем *Space*, за да визуализираме загадката, съответстваща на вратата.



фиг. 4.4

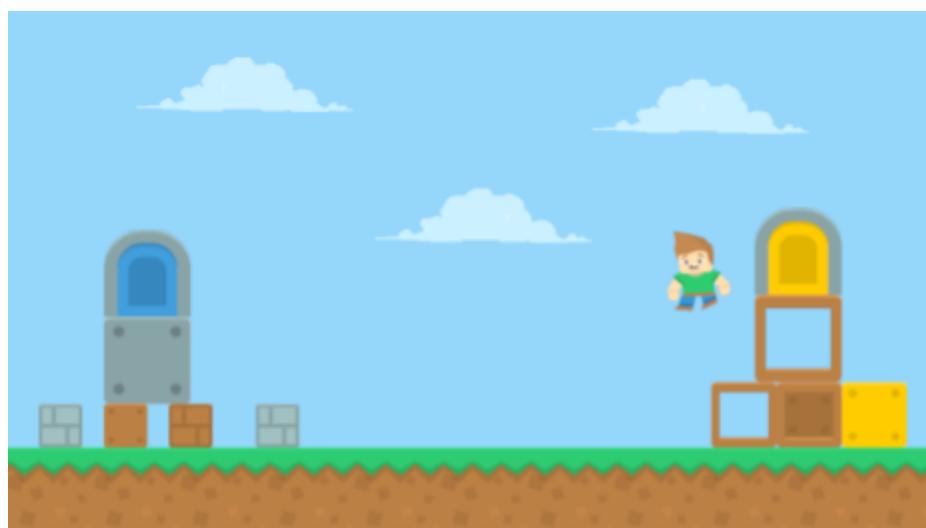
4.3.2. Отваряне и затваряне на еcran със загадка

На фиг. 4.5 е показан еcranът, който се появява при изпълнението на това действие. В горната половина на екрана се намира текстът на загадката, а в долната - отговорът, който даваме. Тъй като все още не сме отговорили нищо, трите позиции са „_“.



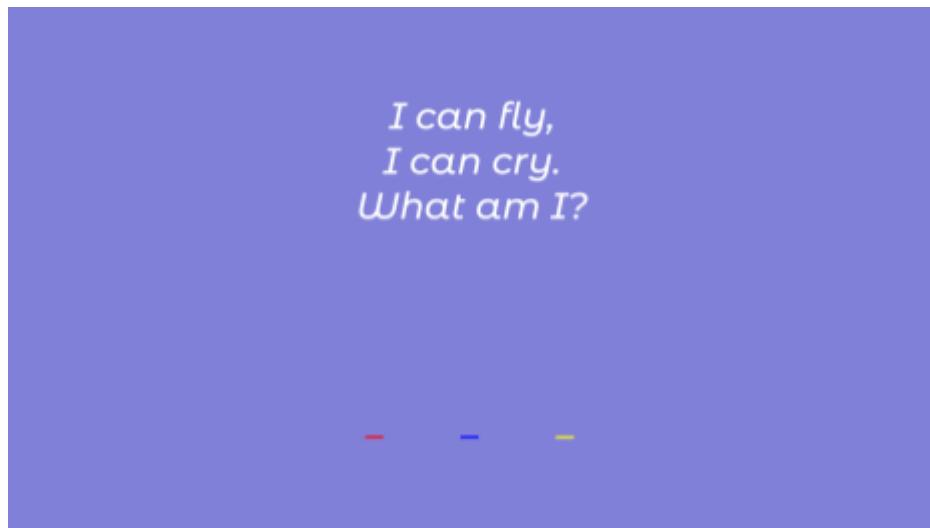
фиг. 4.5

Ако не можем да решим загадката, можем да натиснем бутона *Esc (Escape)*, за да се върнем на картата, и да отидем до другата врата. Връщайки се на картата и тръгвайки надолу от купчината с кутии, анимацията на героя се променя за действието падане. Тя е показана на фиг. 4.6.



фиг. 4.6

Отивайки при вратата, отново натискаме бутона *Space* и виждаме загадката, съответстваща на тази врата. Фиг. 4.7 я показва.



фиг. 4.7

4.3.3. Търсене на отговор

Прочитайки гатанката, за много кратко време откриваме отговора ѝ, а именно „облак“. Връщаме се отново на картата чрез клавиша *Esc*, и с по-близък оглед на облаците по картата можем да видим представеното на фиг. 4.8 - фиг. 4.10.



фиг. 4.8



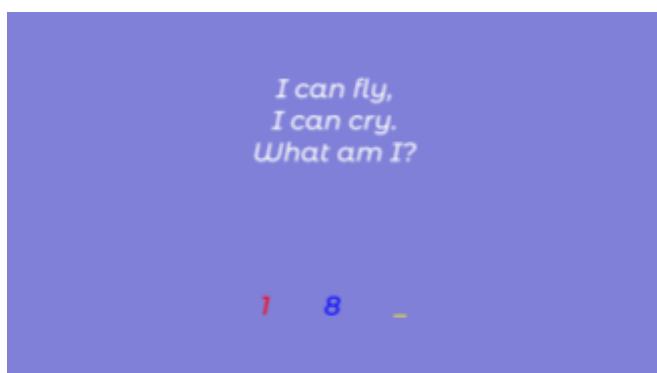
фиг. 4.9



фиг. 4.10

4.3.4. Отговаряне на загадка

Намирайки отговора в трите облака на картата, ние се връщаме на екрана с гатанката, за да го въведем. Започваме, натискайки „1“, но след това вместо „7“, натискаме „8“. Намираме се в състоянието, показано на фиг. 3.11. За щастие, можем да натиснем клавиша *Backspace*, което ще изтрие въведената цифра „8“ и ще ни върне в състоянието, показано на фиг. 3.12.

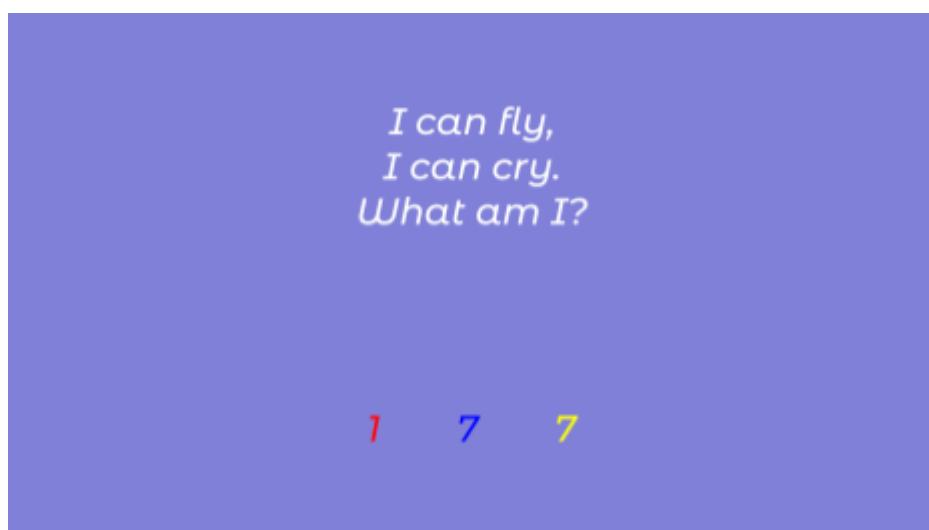


фиг. 3.11



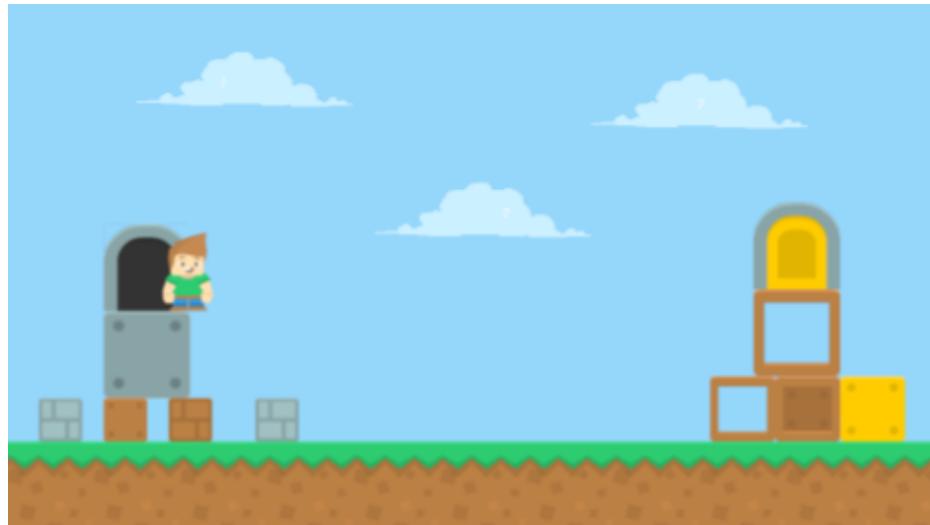
фиг. 3.12

След като вече имаме само правилни цифри на екрана, довършваме отговора си, като натискаме бутона „7“ два поредни пъти. В този момент еcranът изглежда като фиг. 3.13.



фиг. 4.13

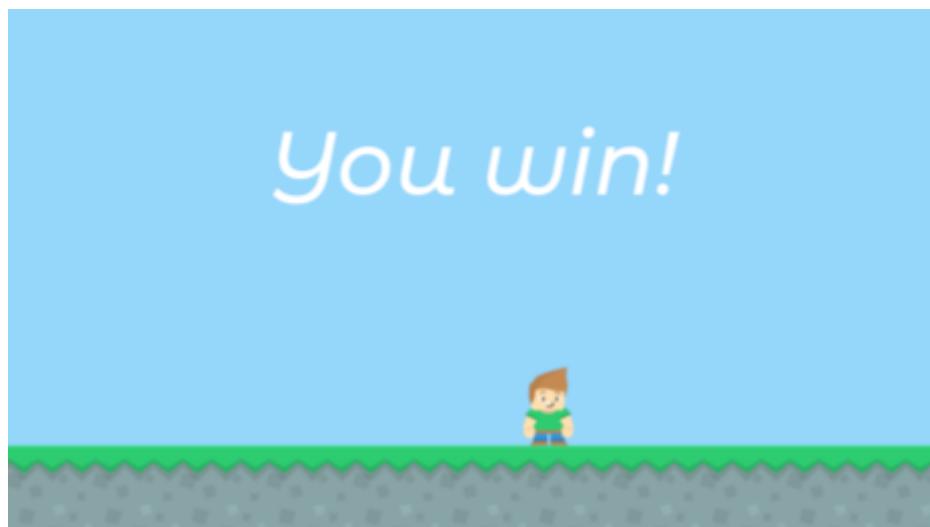
За да проверим дали се прави, натискаме клавиша *Enter*, което ни затваря екрана със загадката и ни връща на картата. Верният отговор ни отваря вратата, което е показано на фиг. 3.14.



фиг. 4.14

4.3.5. Преминаване на следващото ниво

Когато сме отворили врата и героят е в контакт с нея, както е на фиг. 4.14, можем отново да използваме бутона *Space*, за да преминем на следващото ниво. В момента на създаване на книжното тяло, играта продължава до достигане на финалното ниво, показано на фиг. 3.15.



фиг. 4.15

ЗАКЛЮЧЕНИЕ

Настоящата дипломна работа реализира платформена игра с елемент на логически пъзели. Всяко ниво има няколко изхода, най-често два, като за да се мине през него, трябва да бъде решена гатанка. Отговорът ѝ насочва игрacha към предмет, който да намери на картата за да достигне до трицифрен код, който да въведе, за да отвори вратата към следващото ниво.

Бъдещото развитие на проекта включва добавяне на повече видове текстови обекти, зареждани от картата, което ще позволи добавянето на история към играта. Когато играчът се намира на определено място, може да се появява текст, който е част от историята, която той може да използва, за да избере правилният изход на даденото ниво. Също така над вратите може да бъде написано накъде водят, като по този начин те съответстват на сюжета. Могат да бъдат добавени и предмети, които да се събират от игрacha. Някои загадки могат да се показват само ако той е съbral конкретни предмети, които трябва да е срецнал по пътя си до този момент от историята. Проектът може да бъде подобрен и по редица други начини, един от които е подобряване на цветовете на екраните със загадки.

Използвана литература

- [1] Bevy - crate on crates.io, <https://crates.io/crates/bevy>
- [2] Bevy - project on GitHub, <https://github.com/bevyengine/bevy>
- [3] Bevy - docs on docs.rs, <https://docs.rs/bevy/latest/bevy/>
- [4] Bevy ECS LDtk - crate on crates.io,
https://crates.io/crates/bevy_ecs_ldtk
- [5] Bevy ECS LDtk - project on GitHub,
https://docs.rs/bevy_ecs_ldtk/latest/bevy_ecs_ldtk/
- [6] Bevy ECS LDtk - docs on docs.rs,
https://docs.rs/bevy_ecs_ldtk/latest/bevy_ecs_ldtk/
- [7] Bevy Rapier - crate on crates.io,
https://crates.io/crates/bevy_rapier2d
- [8] Bevy Rapier - project on GitHub,
https://github.com/dimforge/bevy_rapier
- [9] Bevy Rapier - docs on docs.rs,
https://docs.rs/bevy_rapier2d/latest/bevy_rapier2d/
- [10] Bevy Rapier,
https://rapier.rs/docs/user_guides/bevy_plugin/getting_started_bevy/
- [11] Bevy, <https://bevyengine.org/>
- [12] Game Maker's Toolkit, Why Does Celeste Feel So Good To Play,
<https://www.youtube.com/watch?v=yorTG9at90g>
- [13] Rust, <https://www.rust-lang.org>
- [14] The Impact of Video Games on the Players Behaviors: A Survey,
<https://www.sciencedirect.com/science/article/pii/S1877050919305393>
- [15] Unofficial Bevy Cheat Book, <https://bevy-cheatbook.github.io/>

Съдържание

| | |
|---|-----------|
| Увод | 4 |
| ПЪРВА ГЛАВА | 6 |
| 1.1. Платформени игри | 6 |
| 1.1.1. Hollow Knight | 6 |
| 1.1.2. Spiritfarer | 9 |
| 1.2. Игри с логически пъзели | 11 |
| 1.2.1. Rusty Lake | 11 |
| 1.2.2. Adventure Escape | 13 |
| 1.3. Платформени игри с пъзели | 15 |
| 1.3.1. Celeste | 15 |
| 1.3.2. FEZ | 16 |
| 1.4. Игри, направени с Rust и Bevy | 18 |
| 1.4.1. Elemental Sorcerer | 18 |
| 1.4.2. Lux synthesē | 19 |
| ВТОРА ГЛАВА | 21 |
| 2.1. Основни функционални изисквания | 21 |
| 2.1.1. Система за движение на героя. | 21 |
| 2.1.2. Показване на логически пъзел и възможност да бъде решен. | 22 |
| 2.1.3. Възможност за преминаване на следващо ниво при успешно решаване на пъзела. | 23 |
| 2.2. Използвани технологии | 23 |
| 2.2.1. Rust | 23 |
| 2.2.2. Bevy | 27 |
| 2.2.3. Bevy Rapier | 28 |
| 2.2.4. Bevy ECS LDtk | 29 |
| ТРЕТА ГЛАВА | 30 |
| 3.1. Структура на проекта | 30 |
| 3.2. Начало на програмата | 31 |
| 3.3. Създаване на допълнение (Plugin) | 33 |
| 3.4. Състояния на играта | 33 |
| 3.5. Зареждане на картата | 34 |

| | |
|---|-----------|
| 3.6. Зареждане на ниво | 35 |
| 3.7. Зареждане на обект от картата | 36 |
| 3.8. Движение на героя | 37 |
| 3.9. Анимиране на героя | 39 |
| 3.10. Зареждане на вратите с гатанки | 42 |
| 3.11. Създаване на еcranите с въпроси | 43 |
| 3.12. Взаимодействие с еcranите с въпроси | 45 |
| ЧЕТВЪРТА ГЛАВА | 48 |
| 4.1. Инсталация | 48 |
| 4.1.1. Необходими налични технологии | 48 |
| 4.1.2. Инсталиране на Git | 48 |
| 4.1.3. Инсталиране на Rust | 48 |
| 4.1.4. Инсталиране на проекта (чрез Git) | 49 |
| 4.1.5. Инсталиране на проекта (без използването на Git) | 49 |
| 4.2. Изпълнение на проекта | 50 |
| 4.3. По време на изпълнението на програмата | 50 |
| 4.3.1. Движение на героя | 51 |
| 4.3.2. Отваряне и затваряне на еcran със загадка | 52 |
| 4.3.3. Търсене на отговор | 53 |
| 4.3.4. Отговаряне на загадка | 54 |
| 4.3.5. Преминаване на следващото ниво | 55 |
| ЗАКЛЮЧЕНИЕ | 56 |
| Използвана литература | 57 |
| Съдържание | 58 |