ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი
ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

გოგა გორგოძე

პროცედურულად გენერირებადი სამყარო

ნაშრომი შესრულებულია ინფორმაციული ტექნოლოგიების
მაგისტრის აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: სრული პროფ. მანანა ხაჩიძე

თბილისი

2019

ანოტაცია

პროგრამული კონტენტის პროცედურულად გენერირება წარმოადგენს ერთ-ერთ ყველაზე მძლავრ მეთოდს თანამედროვე თამაშების შექმნისას, რომელიც ეხმარება თამაში დეველოპერს თამაშის სამყაროს სხვადა სხვა ელემენტების მოდელირებაში. ბოლო კვლევებმა აჩვენა პროცედურულად გენერირების ალგორითმების მოქნილობა ისეთი მიმართულებით როგორებიცაა, მანქანური სწავლება, მოდელების ტექსტურირება, გენეტიკური ალგორითმების იმპლემენტაცია და გამოყენება. 2D/3D განზომილებიანი თამაშების სპეციფიკიდან და დღევანდელი ტექნოლოგიების შეზღუდვებიდან გამომდინარე არსებობს უამრავი ღია პრობლემა. ამ ნაშრომის მიზანია აჩვენოს რამდენიმე მიდგომა, თუ როგორ შეიძლება შეიქმნას პროცედურულად გენერირებული სამყარო და შემდგომ მოხდეს მისი ელემენტების მომდევნო გამოყენება/მოდიფიცერება. პროცედურულად გენერირება მოხდება ერთ-ერთი თამაშის ძრავის Unity Engine გამოყენების მაგალითზე. შესაქმნელი სამყარო გალაქტიკა, რომელსაც აქვს უნიკალური ვარკვლავური სისტემები, უნიკალური პლანეტებით. ასევე განხილული იქნება უკვე არსებული თამაშების პროცედურულად გენერირების მეთოდები, როგორებიცაა ორ განზომილებიანი პლატფორმერის ლეველის გენერაცია გენეტიკური ალგორითმის გამოყენებით, და No Man Sky MMO-OpenWorld-RPG ჟანრის წარმომადგენლის პლანეტების ტექსტურების გენერირების ალგორითმი. ნაშრომის მიზანია ასევე აჩვენოს პროცედურულად გენერირების ეფექტურობა სისწრაფის კუთხით სამყაროს სიმულაციისას, რადგანაც სამყაროს ყოველი ელემენტის შექმნა მიმდინარეობს თამაშის დაწყებისთანავე პროცესორის მიერ, და ის არ საჭიროებს კომპიუტერის სხვა შედარებით ნელი რესურსების (მყარი მეხსიერება,ოპერატიული მეხსიერება...) გამოყენებას.

procedural world generation

Abstract

Procedural content generation (PCG) is one of the most advance and useful tool in modern gaming development, which helps game developer to create the elements of gaming world environment. Last researches shows the advantages of PCG algorithms in fields like machine learning , texture generation, genetic algorithm implementation etc… According to the 2D/3D games, specifications and nowadays technologies there are still many open problems waited their solutions. The aim of this thesis is to show the advantages of PCG using example of creating Procedural generation galaxy using unity engine as framework. Each time the galaxy has unique stars, and each of the star have their own unique star system full of planet. Also there would be shown some successful texture generation algorithm examples of already working Procedural generation games like No Man Sky MMO- Open world – RPG genre.

სარჩევი

შესავალი

თამაშები უკვე დიდი ხანია მხოლოდ დიდი კორპორაციების საქმიანობა აღარ არის. Valves Steam ის მსგავსი თამაშის online პლატფორმების გამოყენება უფრო პატარა და დამოუკიდებელ დეველოპერებს საშუალებას აძლევს უფრო მეტი შანსი იქონიონ მზარდი სათამაშო ბაზრის აღმოსაჩენად. მიუხედავად იმისა რომ შესაძლოა ამ თამაშებმა გარკვეული მასშტაბები მოიცვას და საინტერესოც ჩანდეს, მათ მაინც სჭირდებათ კონტენტის დიდი რაოდენობა, რათა მოთამაშეების ინტერესი და შეძენის მოტივაცია შენარჩუნდეს. ამ კონტენტის შექმნას შესაძლოა ბევრი დრო დასჭირდეს, განსაკუთრებით თუ თამაში მხოლოდ რამდენიმე ადამიანის მიერ ვითარდება/მუშავდება. იმის უზრუნველყოფა, რომ ეს კონტენტი ავტომატურად იქნება გენერირებული დეველოპერებს საშუალებას მისცემდა განვითარების სხვა ნაწილებზე მოეხდინათ ფოკუსირება, იმ დროს როდესაც კონტენტის მაღალ დონეს კვლავაც შეინარჩუნებდნენ.

სათამაშო ინდუსტრიის განვითარებასთან ერთად თამაშისთვის საჭირო კონტენტიც იზრდება. რაც უფრო მეტი კონტენტია საჭირო მოთამაშეთა დაინტერესების შესანარჩუნებლად, მით უფრო მეტი დიზაინერული სამუშაოს საჭიროებაა ამ მოთხოვნების შესასრულებლად. თამაშში კონტენტის შექმნა, მაგალითად მტრები, ერთეულების დონეები, დროს მოითხოვს და შესაძლებელია ძვირიც იყოს. ადამიანი როგორც წესი საკმაოდ ნელა მუშაობს კომპიუტერთან შედარებით. მსგავსი კონტენტი რომ ალგორითმულად გენერირებული იყოს, კომპანიები, რომლებიც თამაშებს ავითარებენ შეძლებდნენ უამრავი დროისა და ფულის დაზოგვას ამ ამოცანებისათვის. ეს საშუალებას მისცემდათ უფრო დიდი და კონტენტით მდიდარი თამაშების განვითარებისათვის, რომლებსაც მოთამაშეებზე პოზიტიური გავლენის მოხდენა შეეძლებოდათ. პროცედურულად გენერირება ასევე საშუალებას მისცემდა უფრო პატარა დეველოპერებს შედარებით დიდი თამაშები შექმნან, რაც მათ ხელს შეუწყობს თამაშის ბაზარზე ძლიერი პოზიციის დაკავებაში, მინიმუმ თამაშის კონტენტის მოცულობის გათვალისწინებით. ამ

ავტომატურ პროცესს პროცედურული კონტენტის წარმოება (PCG) ეწოდება. პროცედურული კონტენტის წარმოება (PCG) თანამედროვე თამაშებში არსებითი და სასურველი საკითხია და ათწლეულების განმავლობაში ფართოდ არის გამოყენებული. შესაბამისად პროცედურული კონტენტის წარმოების სხვადასხვა ობიექტების სამომავლო კვლევა აუცილებელია, რათა თამაშების განვითარებისთვის ახალი ტექნიკების მიწოდება უზრუნველყოს. აღსანიშნავია, რომ ეს უმნიშვნელოვანესი თემაა პატარა დამოუკიდებელი თამაშის განვითარების სტუდიებისთვის დაბალი ბიუჯეტის გამო, სადაც პროცედურული კონტენტის გამოვლენას შეუძლია ნაკლები ძალისხმევით მოახდინოს მეტი კონტენტის და ადამიანური რესურსების გენერირება.

ამასთანავე, თამაშის ინდუსტრიაში პროცედურული კონტენტის გამოვლენის მიმართ ინტერესი არ არის ახალი. პირველი პროცედურული თამაშები საკმაოდ ძველია ,უფრო მეტიც ისინი , კომპიუტერული თამაშებიც კი არ ყოფილან. ისინი ანალოგებს წარმოადგენდნენ და ადამიანი, მოთამაშე უნდა მიჰყოლოდა ინსტრუქციებს კონტენტის გენერირებისთვის. მათი განხილვა პროცედურული კონტენტის გამოვლენის კუთხით შესაძლებელია, როგორც პიონერების. თუმცა აქ ინტერესი უფრო მეტად ციფრულ თამაშებამდე დადის. ყველაზე ადრე ისინი დაინერგა 80 იან წლებში. მათ შორის ყველაზე კარგად ცნობილი წარმომადგენლები არიან Rogue და Elite. ისინი ორივე პროცედურული კონტენტის გამოვლენას იყენებს მოთამაშისთვის სათამაშო გარემოს შესაქმნელად.

უფრო ახალი მაგალითებია Darkspore, (მტრის გამოვლენა) ან Borderlands (იარაღის შექმნა). არსებობს რიგი სხვა მაგალითებიც მეტად რეალისტური გამოსახულებებიდან (SpeedTree) თამაშებამდე, რომლებიც მთლიანად გენერირებულ სამყაროებს იყენებენ. (Elite Dangerous, No Man's Sky). თამაშის ინდუსტრიის ინტერესის გარდა ეს აქტიური კვლევის სფეროცაა.

აქ მიზანი არა მხოლოდ სრულებით ახალი მეთოდების წარმოცენის პოვნაა, მაგრამ ასევე უკეთესი გზების ძიება ჩართული შემთხვევითი პროცესების გასაკონტროლებლად და შედეგის მისაღწევად, ერთის მხრივ უფრო სანდო და მეორეს მხრივ უფრო განსხვავებული.

განვიხილოთ კონტენტის გენერირების რამდენიმე მეთოდი:

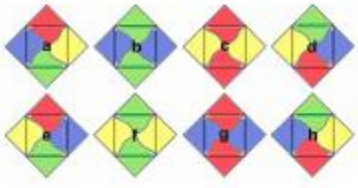
- წინასწარ გაკეთებული/დამზადებული ნაწილების კომბინაცია
- ძებნაზე დაფუძნებული მეთოდები
- ხმაურის წარმომქმნელები
- შეზღუდვებზე დაფუძნებული მეთოდები
- გრამატიკაზე დაფუძნებული მეთოდები

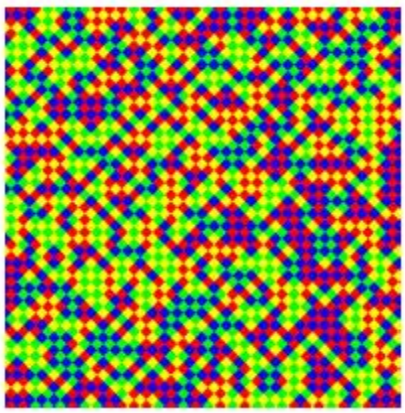
- **კონსტრუქციული მეთოდები**

პროცეურულად გენერირების ძირითადი ალგორითმები

წინასწარ გაკეთებული/დამზადებული ნაწილების კომბინაცია:

კონტენტის შექმნის ერთ-ერთი ყველაზე მარტივი და ფართოდ გავრცელებული მეთოდია წინასწარ დამზადებული ნაწილების გამოყენება და მათი კომბინირება ახალი გზით. ეს ტექნიკა გარკვეულ წესებს იყენებს იმის დასადგენად თუ რომელი ნაწილები მეორდება და სად არიან ისინი განთავსებული. ჰაო ვანგმა ვანგის ფილები გამოიგონა/ Wang tiles 1961 წელს, რომლებიც დღესდღეობით გამოიყენება, ბევრ სხვა რამესთან ერთად, დონეების წარმოქმნისთვის. მზგავსი ფილა, როგორც წესი წარმოადგენს კვადრატს, კუთხეებში ოთხი სხვადასხვა ფერით. მსგავსი ფილების ნაკრებია განთავსებული ნიმუშის ფორმირებისთვის, რაიმე სახის ბრუნვის, ან გამოსახულების გარეშე. მეზობელ ფილებს მსგავსი ფერი უნდა ჰქონდეთ.





მარცხენა სურათი აჩვენებს მსგავს რვა ფილას, რომლებიც გამოგონილ იქნა მ.ფ კოენის მიერ. მარჯვენა სურათი აჩვენებს ამ ფილებით შექმნილი ფილების მაგალითს.
მსგავს მეთოდი, რომელსაც დაკავებით რეგულირებადი გაფართოება ეწოდება, ბოლო ხანს წარმატებით იქნა გამოყენებული სუპერ მარიოს დონეების კიდევ ერთი საყოველთაო მეთოდის დონეების შესაქმნელად. კიდევ ერთი საერთო მეთოდი აქ რითმზე

დაფუძნებული მიდგომის გამოყენებაა. რითმზე დაფუძნებული მეთოდები ცდილობენ დონის წარმოქმნის ინტერპრეტიპება მოახდინონ მუსიკის ნიმუშის მსგავსად. ვინაიდან მუსიკა სხვადასხვა ნოტებისგან შედგება, დონე ასევე შეიცავს პატარა (ატომურ) ნაწილებს. ეს განსაკუთრებით გამოსადეგია 2D პლატფორმებისთვის, მას შემდეგ რაც თამაშებისთვის დონეები სიმღერის მსგავსად სწორხაზოვანია.

ძებნაზე დაფუძნებული მეთოდები

ძებნაზე დაფუძნებული ტექნიკები ცდილობენ ოპტიმიზაციის პრობლემიდან პპტიმალური გამოსავალი მოძებნონ. ეს ნიშნავს, რომ მსგავსი ტიპის ალგორითმები პპტიმალურ გამოსავალს საძიებო სივრცის მიღმა ეძებენ. საძიებო სივრცე მოიცავს ყველა შესაძლო გადაწყვეტას - კარგსაც და ცუდსაც. უფრო მეტიც, მორგების ფუნქციაა საჭირო გადაწყვეტის ხარისხის განსაზღვრისთვის. მსგავსი მეთოდების განხორციელება ხშირად საკმაო გამოწვევებს წარმოშობს. გონივრული მორგების ფუნქციის პოვნა შესაძლოა რთული იყოს. ამასთანავე, გადაწყვეტის შესაბამისი კოდირებაა საჭირო. მსგავსი წარმომადგენლობა შესაძლოა ისეთივე მარტივი იყოს როგორც ციფრების ვექტორი, ან დონის ნაწილების, ან ნიმუშების თანმიმდევრობამდე. საერთო ძებნის მეთოდია ევოლუციური ალგორითმი. მსგავსი მეთოდების დიდი ნაკლია მისი მწირი ეფექტურობა, განსაკუთრებით დიდი საძიებო სივრცეებისთვის.

მეტაევრისტიკულ მიდგომებს/ანალიზებს, როგორიცაა ევოლუციური ალგორითმი, ან გენეტიკური ალგორითმი, არ გააჩნიათ გარანტია, რომ ისინი გლობალუროპტიმალურ გამოსავალს მოძებნიან. უმეტეს შემთხვევებში ისინი მხოლოდ ადგილობრივ მაქსიმუმს იპოვიან. კვლავაც, უთვალავი წარმატებული მაგალითი არსებობს ძებნაზე დაფუძნებული მეთოდების შესაბამისობასა და დონეების წარმოქმნამდე. მაგალითად ხის სტრუქტურის გამოყენება. ხის საკვანძო წერტილი წარმოადგენს ნაწილობრივ დონეს (უმეტესწილად ოთახებს) და ნაწიბური წარმოადგენენ ოთახების მაკავშირებლებს. მსგავსი მეთოდი იქნა გამოყენებული სუპერ მარიოს თამაშისთვის. ისინი იყენებდნენ მიკრო-ნიმუშების თანმიმდევრობას, როგორც დონის კოდირებას.

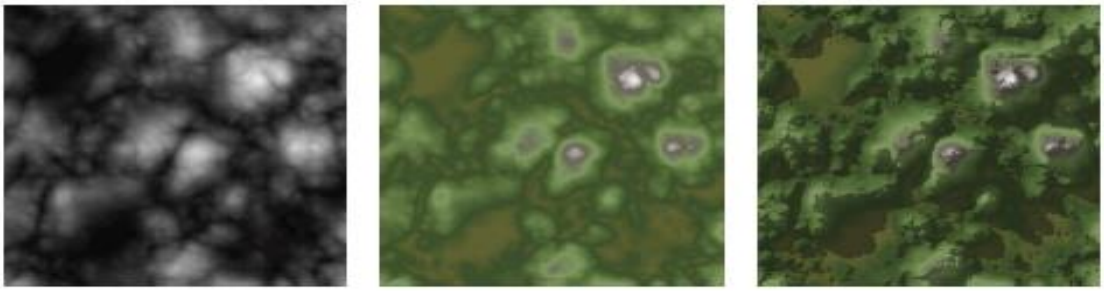
ხმაურის წარმომქმნელები მეთოდები

მარტივად რომ ითქვას ხმაური არის ციფრების შემთხვევითი ნაკრები და ხმაურის წარმომქმნელი არის ფუნქცია, რომელიც მრავალ შემთხვევით მონაცემთა ნაკრებს

წარმოქმნის.. ნეიტრალურ საგნებს თითქმის ყოველთვის ახასიათებთ რაიმე ტიპის ხმაური - ბუნდოვანი, ან არაგლუვი გამოსახულებები, ხმაურიანი ჟღერაობა და ა.შ ხმაური არის ის რაც ბუნებრივ ობიექტებს ასეთებად წარმოაჩენს. შესაბამისად, რაიმეს პროცედურულად წარმოქმნისთვის, რომელიც ნატურალურად გამოიყურება ხმაური საჭიროა. ხშირ შემთხვევაში, შემთხვევითი ციფრების უბრალოდ გამოყენება არ მიგვიყვანს სასურველ შედეგებამდე. მაგალითად დონის წარმომქმნელების შემთხვევაში, ობიექტთა შემთხვევით პოზიციებზე განთავსება არ წარმოადგენს იმას, რასაც დიზაინერი გააკეთებდა. დიზაინერმა უნდა დააჯგუფოს გარკვეული ობიექტები ერთად და სხვები რუკის გარშემო განათავსოს.

არსებობს გარკვეული რეგულარულობა, ნიმუში, რომელშიც არიან ობიექტები მოწყობილი. სწორედ ეს არის მიზეზი თუ რატომ განავითარა **კენ პერლინმა** პირველი ხმაურის წარმომქმნელი, სახელწოდებით პერლინი, 1983 წელს. ეს იყო პროცედურული წარმოებისთვის გამოყენებული ერთ-ერთი პირველი ალგორითმები. ხმაური წარმოიშობა წარმომქმნელი სივრცული გისოსის მიერ/ ბადის (ფსევდო) შემთხვევითი ღირებულებების მიერ, რომლებიც შემდეგ ინტერპოლირდებიან/ჯდებიან ამ სივრცულ ფისოსებს შორის სივრცის შესავსებად. მოგვიანებით კენ პერლინმა განავითარა Simplex ის ხმაური, რათა რიგი ნაკლოვანებები აღმოაფხვრა. Simplex ხმაურს უფრო მცირე გამოანგარიშების კომპლექსურობა გააჩნია და უფრო მნიშვნელოვანია, რომ ის არ განიცდის შესამჩნევი სირთულეების მიმართულების არტეფაქტებს. დეტალები შეიძლება მოიძებნოს მასში. Simplex ხმაურის ოდნავ განსხვავებული ვერსიაა OpenSimplex ხმაური. OpenSimplex იყენებს უფრო მცირე ცვლილებებს რათა თავიდან აიცილოს პატენტთან დაკავშირებული საკითხები. პატენტი უმეტესწილად მაღალი წარმოების ხმაურის მწარმოებლის მოზაიკის/tesselation ფუნქციის შესახებ არის. შესაბამისად 3D OpenSimplex იყენებს ოთხწახნაგოვან-რვაწახნაგოვან ფორებს ნაცვლად ტეტრაგონალური არასოლისებური ფორებისა Simplex ის ხმაურის 3D ვერსიიდან.

თავდაპირველად განვითარდა რა სტრუქტურებისთვის, ხმაურის წარმომქმნელები ახლა უკვე გაცილებით მეტი კონტენტისთვის გამოიყენება. მაგალითად სტრუქტურა სესაძლოა იყოს ინტერპრეტირებული, როგორც მაღალი, ლანდშაფტის შესაქმნელად, მსგავსად იმისა, რომელიც გრაფიკზეა ნაჩვენები. სხვადასხვა ხმაურის სიხშირეების გამოყენება ასევე შესაძლებელია თამაშის ობიექტების, ან სხვადასხვა სათამაშო არეალების განსაზღვრისას. ცნობილი თამაში Minecraft მსგავს ტექნიკას იყენებს **biomes** ის განსაზღვრისა და მათ შორის შესაბამისი ტრანზიციის უზრუნველსაყოფად.



შეზღუდვებზე დაფუძნებული მეთოდები

შეზღუდვებზე დამყარებული მეთოდები ცდილობენ გადაწყვეტის სასურველი თვისებების განსაზღვრას, ნაცვლად ნაბიჯების თანმიმდევრობის არწერისა, რათა გამოსადეგ/ხელსაყრელ გადაწყვეტილებას მიაღწიონ. ზოგიერთი ალგორითმი შესაძლოა ძლიერ და მსუბუქ შეზღუდვებს შორის იყოს გამიჯნული. მძიმე/მკაცრი შეზრუდვები უნდა შესრულდეს, მაშინ როდესაც მსუბუქი შეზღუდვები არჩევითია. ამისი უპირატესობა ის არის, რომ მხოლოდ მოსალოდნელი გადაწყვეტები წარმოიქმნება და მორგების ფუნქციას აღარ საჭიროებს. გარდა ამისა, საძიებო სივრცე გაცილებით მცირეა იმ მეთოდებთან შედარებით, რომლებიც შესაძლოა ასევე წარმოქმნიდნენ არასასურველ გადაწყვეტებს. ნაკლს წარმოადგენს ის, რომ შეზღუდვების განსაზღვრა საჭიროებს თამაშის მექანიკის საკმარის ცოდნას. ზოგადი შეზღუდვების მოძებნა, რომლებიც შესაბამისია ბევრი განსხვავებული თამაშისთვის და შეუძლია სათამაშო დონეები აწარმოოს მარტივი ამოცანა არ არის. პასუხზე დამყარებული პროგრამირება (ASP) ფართოდ გამოყენებული მეთოდია ახალი კონტენტის წარმოებისთვის. ის იყენებს დეკლარაციული ლოგიკის ენას, რომელსაც AnsProlog ეწოდება. AnsProlog ის გამოყენება შესაძლებელია იმის აღსაწერად თუ მოცემული პრობლემის გადაწყვეტა როგორ შეიძლება გამოიყურებოდეს მარტივი ლოგიკის წესებისა და განცხადებების დახმარებით. უკვე იქნა ნაჩვენები რომ ეს მეთოდი შეესაბამება გენერირების დონეებს, მაშინაც კი თუ ნაწარმოები 3D სტრუქტურებია შესაძლებელი.

გრამატიკაზე დაფუძნებული მეთოდები

თავდაპირველად დაინერგა ბუნებრივი ენის აღსაწერად. დღეისათვის გრამატიკაზე დაფუძნებული მეთოდები შესაბამისია ყველა ტიპის პრობლემისთვის კომპიუტერულ

მეცნიერებებში. ანბანი და წესების ნაკრებია საჭირო გრამატიკის განსასაზღვრად. წესი განსაზღვრავს ანბანის რომელი ასო უნდა ჩანაცვლდეს ერთი, ან მეტი სიმბოლოთი. წესები გამოდმებით გამოიყენება, ვიდრე დროის შეწყვეტის მდგომარეობა არ მიიღწევა, ანუ ვიდრე სიმბოლოების ჩანაცვლება აღარ იქნება შესაძლებელი. მსგავსი სისტემის მუშაობის შედეგის მარტივი მაგალითი შესაძლებელია დანახულ იქნეს დაბლა მოცემულ გრაფაზე. განსხვავებები კეთდება დეტერმინირებულ და არა დეტერმინირებულ გრამატიკებს შორის. პირველ შემთხვევაში წესები ყოველთვის ცალსახაა/ნათელია. თუმცა არადეტერმინირებულ გრამატიკას შესაძლოა მრავალი წესი ქონდეს ერთიდაიმავე სიმბოლოს თანმიმდევრობისთვის. მსგავსი მეთოდებით კონტენტის წარმოქმნა ძალიან ეფექტურია და შესაბამისად რეალურ დროში გამოყენებადიც. თუმცა, კონკრეტული ნაკლია ის რომ გრამატიკის კონფიგურაცია უნდა მოხდეს თითეული თამაშისთვის. ზოგადი წესები არ არსებობს სხვადასხვა თამაშის მოცვისთვის. შესაბამისად, მსგავსი ტექნიკის გამოყენება GGP სთვის შეზღუდულია.

კონსტრუქციული მეთოდები

კონსტრუქტიული მეთოდები ნაბიჯ-ნაბიჯ წარმოქმნიან კონტენტს. ერთი წარმოქმნილი შედეგი მხოლოდ ერთი გადაწყვეტაა/გამოსავალია. შესაბამისად, არ არსებობს ხელახლა შეფასებისა და წარმოქმნილი კონტენტის გაუმჯობესების განმეორებითი პროცესი. კონსტრუქციულ ალგორითმებს შეუძლიათ ორ კატეგორიას შორის განლაგდნენ: სივრცის დანაწევრება და ფიჭური ავტომატი./space partitioning and cellular automata. სივრცის დანაყოფების ალგორითმები იყოფა 2D ან 3D სათამაშო სივრცედ, მცირე დანაწევრებულ ქვეჯგუფებად. (უჯრედები). ხანდახან ქვედანაყოფები განმეორებით ვრცელდება. შესაბამისად თითეულ უჯრედს კიდევ უფრო პატარა უჯრედებად დაყოფა შეუძლია. ეს აისახება უჯრედების იერარქიაში – ხე. Cellular automata/ფიჭური ავტომატის საბაზისო კონცეფცია 1940 იან წლებში უკვე აღმოჩენილი იყო, მაგრამ მისი რეალური შემოტანა თამაშში კონცეპციის "Game of Life" ის მეშვეობით მოხდა. ის ჯონ ჰორტონ კონვეის მიერ 1970 წელს იქნა განვითარებული. ფიჭური ავტომატიზაცია არის თვით

ორგანიზებული სტრუქტურა, რომელიც უჯრედების რეგულარული ქსელისგან შედგება. თითეულ უჯრედს რიგი მდგომარეობები აქვს, მაგალითად უმარტივეს შემთხვევაში ჩართვა და გამორთვა და მათი მეზობელი უჯრედების მინიშნება/მითითება. მათ ასევე აქვთ რაღაცა საწყისი მდგომარეობა. (t = 0). საბოლოოდ, არსებობს წესების ნაკრები, (ხშირ შემთხვევაში მათემატიკური ფუნქცია) რომელიც განსაზღვრავს თითეული უჯრედის შემდგომ მდგომარეობას.(t +1) ფუნქციის განმეორებითი გამოყენება აისახება სხვადასხვა ნიმუშებში. შედეგად მიღებული ნიმუში დიდწილად იქნება დამოკიდებული საწყის მდგომარეობაზე, წესებსა და იმაზე თუ რამდენი განმეორება განხორციელდა. ამის, როგორც დონის მწარმოებლის რეალურად გამოყენებისას ჩართვა/გამორთვის მდგომარეობები მაგალითად ინტერპრეტირებულია, როგორც კედლების/თავისუფალი გასვლა/პასაჟი. ჯონსონმა და სხვებმა ავტომატი/automation განსაზღვრეს, როგორც მღვიმის წარმოქმნა, როგორც სტრუქტურები, მხოლოდ სამი მდგომარეობით (სართული, კლდე, კედელი) და ორი მარტივი წესით. სხვა პუბლიკაციებიც მსგავს წესებს იყენებდნენ ხვრელების/საკნების, ან ლანდშაფტების შესაქმნელად. ფიჭური ავტომატის (cellular automata) გამოყენებას რიგი უპირატესობები აქვს. ერთი, განხორციელება უმეტესწილად მარტივია და შედეგად ალგორითმები საკმაოდ ეფექტურია. მას ასევე ის უპირატესობაც აქვს რომ უსასრულობის დონეები შესაძლებელია წარმოიქმნას და მისი ეფექტურობის გამო, მას ასევე შეუძლია დროულად განხორციელდეს, თამაშისას. უდიდეს ნაკლს წარმოადგენს ამ ალგორითმის პირდაპირი კონტროლის სიმცირე. მაგალითად, ან ძალიან რთულია, ან შეუძლებელია იმის უზრუნველყოფა, რომ წარმოქმნილი ხვრელები მოთამაშის მიერ მიღწევადი იქნებიან. მარტივი ხვრელები/საკნები შესაძლოა დანარჩენი დონიდან ჩამოიჭრას. დამატებითი მეთოდია საწირო იმის უზრუნველსაყოფად, რომ მსგავსი შემთხვევები აღმოიფხვრება. როგორც წინა განყოფილებებში იქნა წარმოდგენილი, ბევრი სხვადასხვა ტექნიკაა ხელმისაწვდომი კონტენტის წარმოებისთვის. ნაჩვენები ტექნიკების უმეტესობა მიესადაგება დონის დიზაინების შექმნას, მაგრამ ზოგადად არც ერთი მათგანი არ არის რეალურად უკეთესი სხვაზე. ყოველ მათგანს სხვადასხვა დადებითი და უარყოფითი მხარეები გააჩნია. ჯ. ტოგელიუსი და სხვები სასურველ თვისებებს აღწერენ, როგორც PCG სისტემას, რომელსაც უნდა ჰქონდეს: სიჩქარე, სანდოობა, კონტროლირებადობა, გამოხატულება და განსხვავებულობა, კრეატიულობა და დამაჯერებლობა. ეს თვისებები შესაძლებელია გამოყენებულ იქნას წარმოდგენილი მიდგომების კატეგორიზაციისთვის, მაგრამ ზოგიერთი მათგანი სუბიექტურია და შესაბამისად კლასიფიცირება რთულია. თუმცა, ადვილი მისახვედრია, რომ ყველა

სასურველი თვისების შესრულება თითქმის შეუძლებელია არსებული მდგომარეობით. არსებული ალგორითმები მხოლოდ გარკვეულ დონეზე აარქივებენ ზოგიერთ თვისებას.ამისი გაუმჯობესება შესაძლებელია მრავალი მეთოდის კომბინაციის შედეგად ახალი მიდგომის ჩამოყალიბებით. თითქმის ყველა თვისება, შესაძლებელია სიჩქარის გარდა, შესაძლებელია მსგავსი ჰიბრიდული მეთოდებით გაძლიერდეს. თუმცა ჰიბრიდული მეთოდებითაც კარგი ბალანსის მოძებნა უნდა მოხერხდეს ამ ხარისხის მახასიათებლებს შორის.უფრო მეტიც ჯ. ტოგელიუსმა დეტალური სისტემატიკის მოწოდება უზრუნველყო პროცედურული კონტენტის მწარმოებლებისთვის. ეს გამოსადეგია კონტენტის წარმოების კატეგორიზაციის, ან აღწერისთვის. გამოყენებული მთავარი მიდგომა აქ უმეტესწილად იგივეა. მთლიანი სისტემის მხოლოდ მცირე კომპონენტი შეიცვლება.შესაბამისად, ამ სისტემატიკის მიხედვით კატეგორიზაცია თითქმის მსგავსი იქნება ორივე ვარიანტისთვის. სხვა ავტორთა უმეტესობა, რომლებიც ახალ მწარმოებლებს წარმოადგენენ, როგორც წესი თავიანთ მიდგომას შესაბამის ლიტერატურაში არსებულ მეთოდებს ადარებენ. სამწუხაროდ, გამოყენებული შედარების მეთოდების უმეტესობა მხოლოდ ერთი/ერთხელ გამოყენების შემთხვევისთვის არის სპეციფიური.

თამაში როგორც პროგრამული უზრუნველყოფა

ჩვენ განვიხილეთ პროცედურულად გენერირებადი სამყაროების რამდენიმე ძირიტადი ალგორითმი ახლა კი , სანამ პროცედურული გენერირების კონკრეტულ ალგორითმზე გადავიდოდით, განვიხილოთ თამაშის შექმნის პროგრამული თამაშების მაგალითებზე. აქედან გამომდინარე, ჩამოვაყალიბოთ თამაში როგორც პროგრამულული უზრუნველყოფა და განვიხილით მისი სპეციფიკაციები. მოდით განვიხილოთ თამაშის შექმნის გზა დასაწყისიდან, მის სრულ დამთავრებამდე. პირველ რიგში, უნდა გავითვალისწინოთ , რომ თამაში არის ინფორმაციული სამყაროს ნაწილი, ანუ ის წარმოადგენს პროგრამულ უზრუნველყოფას. აქედან გამომდინარე, ის შეიცავს პროგრამული უზრუნველყოფის შექმნის თითქმის ყველა ეტაპს, თუმცა ამასთანავე აქვს გარკვეული , თავისებური სპეციფიკაც.

ყველაფერი იწყება “GameDesignDocument”-ის შემუშავებიდან. ეს დოკუმენტი წარმოადგენს თამაშის სიტყვიერ აღწერას, თუ რა ელემენტებს უნდა შეიცავდეს თამაში, რას წარმოედგენს თამაშის პროცესი, რამდენი მოთამაშე ყავს, როგორი უნდა იყოს უშუალოდ თამაშის პროცესი და ა.შ. აღსანიშნავია ,რომ ეს დოკუმენტი შეიძლება შეიცვალოს თამაშის შექმნის პროცესში, შეიძება შეიცვალოს ზოგი ელემენტები, ზოგზე სრულად უარი ითქვეს, თუმცა ძირითადი ჩონჩხი, ბაზური სტრუქტურა უცვლელი რჩება. დოკუმენტის ჩამოყალიბების შემდგომ, ირჩევა ტექნოლოგიები, რომლებიც ყველაზე მოსახერხებელი და გამოყენებადი იქნება კონკრეტული თამაშისათვის.

ტექნოლოგიებში იგულისხმება ძრავა, GameEngine- ინსტრუმენტების არეალი, რომლის გამოყენებითაც უნდა აღიწეროს და ხორცი შეესხას თამაშს , როგორც ვიზუალურად , ასევე ფუნქიონალურადაც. დღეისათვის, ძალიან ბევრი ასეთი ძრავა არსებობს, ყველაზე პოპულარულები არიან : Uneal, Unity. ძრავის არჩევის ეტაპი ძალიან მნიშნველვანია, რადგანაც ის განსაზღვრავს თითქმის მთელ შემდომ შექნის პროცესს. ყოველ ძრავას აქვს გარკვეული შეზღუდვები,ჩარჩოები , რომლის ფარგლებ გარეთაც ვერც ერთი პროგრამისტი ვერ გავა. მაგალითისთვის განხილოთ Unity-ს ფრეიმვორკი.მას აქვს C# და javascript-ის მხარდაჭერა, რაც თავის მხვრივ გულისხმობს, რომ ის თავსებადია ობიექტზე ორიენტირებული პრინციპებთან და ამავდროულად მას აქვს ფუნქციონალური პროგრამირების დიდი პოტენციალი.ძრავის შიდა ბიბლიოთეკები შესაძლებლობას იძლევა დამზადდეს როგორც ორ განზომილებიანი ასევე სამ განზომილებიანი თამაში, აქვს CANVAS ობიექტების ცნება, რაც გაცილებით აადვილებს ორ განზომილებიან ობიექტებთან მუშაობას და ასევე ფართოდ გამოიყენება არამხოლოდ თამაშის, არამედ აპლიკაციების დასაწერადაც. ახალი პროექტის შექმნისას გენერირდება არჩევის მიხედვით 3D/2D სამყარო, რომლის გამოსახულების რენდერის ზომები შემოფარგლულუა ცენტრიდან float ტიპის მაქსიმალური მაჩვენებლით 9223372036854775807. ეს შეზღუდვა განპირობებულია იმით, რომ ძრავას არა აქვს double ტიპის მხარდაჭერა სამყაროსთან მუშაობის დროს. თუმცა არავინ გიზღუდავთ გამოიყენოთ ის კლასის აღწერისას ან სხვა საჭირო მათემატიკური ფუქნციის წერისას. ამ შემთხვევაში მოხდება მისი კონვერტირება ძრავის ფრეიმვორკის მხარდაჭეილ ტიპზე. ყოველი ობიექტი ფრეიმვორკის ვირტუალურ სამყაროში აქვს GameObject კლასის ობიექტი თვისებათ, რომელიც გამოიყენება ამ სამყაროში ინტერაქციისათვის.მას აქვს Transform თვისება ორ განზომილებიანი ან სამ განზომილებიანი სამყაროს მიხედვით. ეს თვისება წარმოადგენს გაჩუმებით პარამეტრს, რომელიც აქვს ყველა ობიექტს

განურჩევლად. ამასთანავე იქნება ეს ობიექტი კამერა, კანვასის ობიექტი თუ სხვა პირდაპირ დამოკიდებულია პროგრამისტზე. Transform და სხვა კლასის თვისებებს ასევე კომპონენტებს უწოდებენ. ძრავის მოქნილობა შესაძლებლობას იძლევა სხვადასხვა გვარი დაშენებების და პლაგინების მიბმას სხვადასხვა გვარი შეზღუდვების თავიდან ასაცილებლად. მაგალითად ზემოთ ხსენებული float ტიპის შეზღუდვა შეგვიძლია თუ გამოვიყენებთ სამყაროს ცენტრის გადაადგილებას სამყაროში არსებული ყველა ობიექტის პოზიციის ცვლილებით იმ ერთი ობიექტის მიმართ რომელის პოზიცია გაცილებით აღემატება float ტიპის მაჩვენებელს. ეს მიდგომა ცნობილია როგორც პირველად წერტილის დასმა (Origin Point).

ყოველი ძრავა მუშაობს სცენის პრინციპის გამოყენებით, სხვა სიტყვებით რომ ვთქვათ, არსებობს ვირტუალური სივრცე, რომელიც ივსება თამაშისათვის საჭირო ობიექტებით და რადგანაც ძრავა ფაქტიურად არის პროგრამული გარემო, ყოველი ობიექტი შესაძლოა განვილიხოთ როგორც გარკვეული კლასის ობიექტი. აქედან გამომდინარე, ძალიან მნიშნელოვანია, რომ შერჩეული ძრავა აკმაყოფილებდეს თამაშის მოთხოვნებს. ამასთანავე არსებობს სრულად მეორე შემთხვევა, შეიძლება შეარჩიოთ ძრავა, რომელიც გაცილებით მეტ შესაძლებლობას იძლევა, ვიდრემ, თამაშს ეს რეალურად ჭირდება. შესაბამისად ოპტიმიზაციის ფარგლებში მნიშვნელოვანია დავიჭიროთ ეგრედწოდებული , ოქროს შუალედი.

პროცედურული გენერაცია სამყაროების გენერაციისთვის გვაძლევს ოპტიმიზაციის შესაძლებლობას ძალიან დაბალ აპარატურულ დონეზე, რადგანაც ის ძალიან მოქნილია და დამოკიდებულია ალგორითებზე, რომელზეც არის ის დაფუძნებული, თუმცა ამაზე ცოტა მოგვიანებით.

ზოგადად პროცედურული გენერაციის გამოყენების ორი მიდგომა არსებობს : ნაწილობრივი ან სრული. ნაწილობრივი გენერაციის პირობებში გენერაციის ალგორითმები გამოიყენება სამყაროში არსებული ობიექტების მხოლოდ გარკვეული თვისებების გენერაციისთვის (ტიპი,ფერი,სკალა, კლასის თვისებები…), ხოლო სრული გენერაციისას ყველა ობიექტი რომელიც არსებობს ძრავის სცენაში გენერირდება ნულიდან და ხდება მათი პარამეტრიზაცია. ცხადია , რომ პროცედურულად გენერირების ტექნიკა და გამოყენების არეალი, როგორც ყველა ტექნოლოგიური კონცეპტი არის შეზღუდული დღევანდელი აპარატურული სიმძლავრით და შესაძლებლობებით, თუმცა თამაშების ინდუსტრიაში არსებობს მაგალითები სადაც პროცედურულად გენერირების

მეთოდი ბრწყინვალედ არის გამოყენებული, მაგ: კოსმოსის გენერაციისათვის(No Man Sky), გრაფული ტექნოლოგიის ვიზუალიზაციისათვის(Apparance).
იქედან გამომდინარე, რომ მთლიანი კონტენტის გენერირება ხდება პროგრამის უშუალოდ გაშვებბისას(runtime), შესაძლებელი გახდა საჭირო ფიზიკური მეხსიერების მკვეთრად შემცირება. ამასთანავე, ძრავის სცენაში ობიექტების პარალელურად ჩატვირთვის საჭიროებაც მოხსნილია, რადგანაც ეს ობიექტების გენერირდება პროგრამის გაშვებუსთანავე. მიუხედავად იმისა, რომ ყველაფერი პროცედურულად გენერირდება, სამყაროს ყოველი ობიექტს ენიჭება სერიული ნომერი, იდენტიფიკატორი, რაც ,თავის მხვრივ, კასტომიზაციის, თვისებების შეცვლის საშუალებას იძლევა. სამყაროს გენერაცია მიმდინარეეობს რეალურ დროში, აქედან გამომდინარე, პროგრამისტს არ ჭირდება თავიდან კომპილირება კოდის, და მისი მოდიფიკაცია შესაძლებელია სამყაროს შექმნის შემდეგაც. მაშასადამე, პროცედურულად გენერირების მოდელი ობიექტების მასშტაბის, თვისებების და სცენების კომპლექსურობის დონის რეგულირების საშუალებას იძლევა. აქედან გამომდინარე, დეველოპერს შეუძლია მეტი დრო დაუთმოს კრეატიულ პროცესს, რადგანაც ყოველი მძიმე სამუშაო, რომელიც გულისხმობს სცენის შექმნას, შესრულდება მისი უშუალოდ ჩარევის გარეშე.

გალაქტიკის პროცედურული გენერაციის ალგორითმი

გალაქტიკის შექმნისთვის ჩვენ გამოვიყენებთ რამდენიმე გენერაციის ალგორითმის ერთობილობას , ყველა მათგანი დამოკიდებულია ფესვის რიცვზე რომელსაც ჩვენ ვირჩევთ ეს შეიძლება იყოს ნებიმიერი ნატურალური რიცვი

- ვარსკვლავებს შორის დისტანციის გენერირება
- გალაქტიკის ფორმის გენერაცია
- ვარსკვლავების პარამეტრების გენერირება
- პლანეტების პარამეტრების(პოზიცია, რესურსები...), და ვიზუალური ტექსტურების გენერირება.

ვარსკვლავებს შორის დისტანციის გენერირება

დისტანციის გენერირების შედეგი იქნება 3 განზომილებიანი ვექტორების ლისტი. ალგორითმი იყენებს შემდეგ ძირითად ბაზისურ ფორმულებს:

F1=Sqrt(-2.0*Log(S)/S) სადაც S= u*u+v*v ,ხოლო u=v=2.0*Random.Seed-1.0

f=(-F1,0);

q=(0;F1);

Xcoord=randomRadius*sin(f1)*sin(q)

Ycoord= randomRadius*cos(f1);

Zcoord=randomRadius*sin(f)*cos(q);

შედეგად კი გვექნება ვექტორების (Xcoord, Ycoord, Zcoord) ერთობლიობა (დეტალური კოდი იხილეთ დანართი 1 ში)

გალაქტიკის ფორმის გენერაცია

ფორმის განსაზღვის შედეგი არის ვარსკვავების გროვების პოზიციები(3 განზომილებიანი ვექტორები) ალგორითმი იყენებს შემდეგ ძირითად ბაზისურ ფორმულებს:

თითოეული ვარსკვავისთვის გამოითვლება გროვის რიცხვი

q=i/sum ,სადაც I არის ვარკსვკლავის ადგილი, ხოლო sum გალაქტიკაში ვარკსვლავების რაოდენობა.

გროვის პოზიცია განისაზღვრება O=(q+Ranom.Seed)*Pi*2 , შემდეგ გამოითვლება გაბნევის მაჩვენებელი x,y ზე

X=Pow3(RandomeRange(0,0.9999)-Pow3(RandomeRange(0,0.999999))*Galaxy.Seed,

სადაც Pow3- აქვს სტაბილიზატორის ფუქნცია.

Pows3(X){

V= x-(0.5)* (0.5)* (0.5) *4+0.5f

Result= V/ან სისტემური მაქსიმალური მნიშვნელობა

}

საბოლოო კოორდინატებს კი აქვთ შემდეგი სახე

Xcoord=cos(X)*q/2+0.5+X;

Ycoord= X;

Zcoord= sin(X)*q/2+0.5+X;

შედეგად კი გვექნება ვექტორების (Xcoord, Ycoord, Zcoord) ერთობლიობა (დეტალური კოდი იხილეთ დანართი 1 ში)

ვარსკვლავების პარამეტრების გენერირება

ვარსკვლავს აქვს შემდეგი პარამეტრები:

უნიკალურრი იდენტიფიკატორი, ტიპი, კლასი, კოდური სახელი, გამოსაჩენი სახელი, ფერი, ტემპერატურა,მასა, რადიუსი, პოზიცის x,y,z კოორდინატები, პლანეტების რაოდენობა და გალაქტიკის უნიკალური ფესვი.

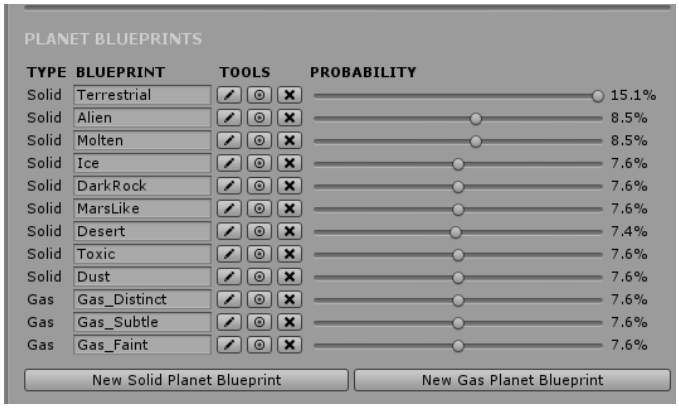
ვარკვლავების გენერირებისას შეუძლებელია უგულბელვყოთ შემდეგი მინიმალური კონსტანტები:

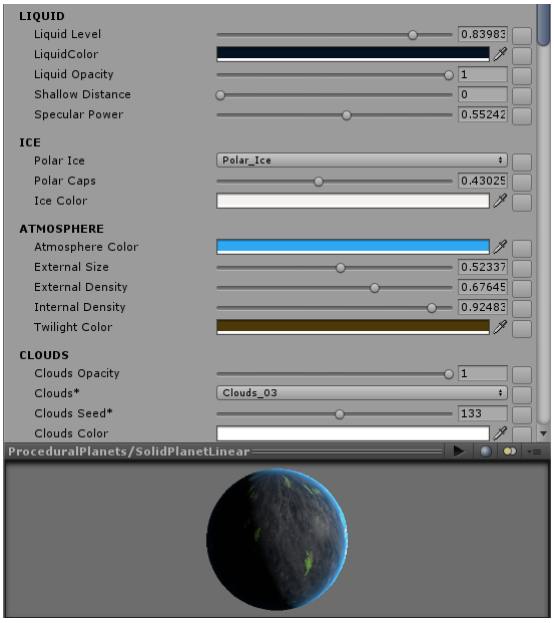
- გრავიტაციული კონსტანტა `6.673e-11.`
- დედამიწის მასა `5.972e+24.`
- მზის მასა `1.989e+30.`
- მზის რადიუსი `695.7e+6.`
- წამი წელიწადში `31536000.`
- პლანეტების მაქსიმალური რაოდენობა 16
- მზის კაშკაშის კოეფიციენტი`3.828e+26.`
- ცელსიუსის/კელვინი `-273.`
- სინათლის წელიწადი/პარსეკი `3.26.`
- სინათლის სიჩქარე `299792458.`

გენერირების ალგორითმის პრაქტიკული რეალიზაცია იხილეთ დანართი1 ში

პლანეტების პარამეტრების და ვიზუალური ტექსტურების გენერირება.

თუ ვარკსვლავის შემთხვევში ის ვიზუალურად ის წარმოადგენს თამაშის გარემოში უბრალო ნათებას, არაფიზიკურ ობიექტს, პლანეტის შემთხვევაში ყველაფერი უფრო რთულად არის რადგანაც მისი ვიზუალური გგამოსახულება შედგება ანიმირებული ტექსტურების ერთობლიობასთან. ამავდროულად არ უნდა დაგვავიწყდეს, რომ გვავს 3 ტიპის პლანეტა გაზის პლანეტა სადაც მხოლოდ გაზის ელემენტები არსებობს, მყარი/სოლიდ ტიპის პლანეტა , სადაც მხოლოდ მყარი ელემენტები არსებობს, და სელენას ტიპის პლანეტა სადაც გვხვდება როგორც გაზის ასევე მყარი ელემენტები. პლანეტების ტექსტურირების გენერატორი იყენებს დუპლიკატების(წინასწარ შესარჩევი მატერიალების) და შემთხვევითი ალბათობის მაჩვენებლების მიხედვით გვაძლევს ტექსტურების ნაკრებს რომელიც გადაეკრობა სფეროს. სინამდვილეში თითოეული პლანეტა წარმოადგენს 3 განზომილებიან სფეროს, რომელსაც გადაკრული აქვს შესაბამისი ტექსტურათა ნაკრები. ასევე მნიშვნელოვანია გენერირების მოქნილობაც. გენერირების ამ ასპექტში გამოიყენება ნახაზების, ე.წ ბლუპრინტების ცნება. ბლუ პრინტი- არის წინასწარ შექმნილი პლანეტა, რომლის ტექსტურები შესაძლოა გამოიყენებოდეს სხვა პლანეტის შესაქმნელად. ყველა ალგორითმი ინტეგრირებული არის Unity ძრავაში და ჩაშენებული OnGui ფრეიმვორკის გამოყენებით თითოეული მოძრავი პარამეტრი გამოტანილია Unity-ს გრაფიკულ ინტერფეისში.





ასევე შესაძლებელია ახალი ბლუპრინტების დამატებაც. (ლანეტების ტექსტურების გენერირების პროგრამული კოდი დეტალურად იხილეთ დანართი 1-ში)

დასკვნა

ამ ნაშრომში წარმოდგენილ იქნა ახალი პროცედურული დონის გენერატორი/მწარმოებელი ვარკსვლავების და პლანეტების გენერაციის მაგალითზე. მას შეუძლია აწარმოოს დასაშვები და საკმაოდ გამომწვევი დონეები თვითმართვადი თამაშიდან. პრგორამული ჩარევა დეველოპერის მხრიდან ჩარევა არ არის საჭირო.
ამ ურთიერთქმედების კომუნიკაციის არხის მეშვეობით ყველაფრის გაკეთება შესაძლებელია ავტომატურად.მიუხედავად ამისა, ადამიანს შეუძლია ჩაერიოს დიზაინის კუთხით და უკეთესი შედეგების მისაღებად მოახდინოს მწარმოებელის კონტროლი. შემოთავაზებული მიდგომა იყენებს PCG ჩარჩოს თამაშის აღწერის გასაანალიზებლად. თამაშის აღწერა შეიცავს მნიშვნელოვან ინფორმაციას, რომელიც შემდეგში გამოიყენება დონის არაპირდაპირი წარმომადგენლობის ასაგებად. ეს წარმომადგენლობა სათამაშო დონე არ არის. ეს არის აბსტრაქტული აღწერილობა იმაზე თუ როგორ შეიძლება გამოიყურებოდეს შესაძლო დონე.

განვიხილეთ ტექსტურების გენერირება პლანეტებზე ,სადაც კოდირებულია ინფორმაცია, როგორიცაა ზომა, რომელი და რამდენი ელფერი შეიძლება გააჩნდეს დონეს და როგორ არიან ისინი გადანაწილებული. გადანაწილება არწერს დონის სტრუქტურას, ან დიზაინს. ის პასუხობს კითხვაზე, რომელიც ყოველი თამაშის დიზაინერს ექნებოდა: როგორ შეიძლება გამოვიყენოთ მატერიალები,როგორც ელფერების სახეობები? განლაგებულია ისინი სპეციფიური ფორმით ლაბირინთის, ან საჭადრაკო დაფის ფორმით? ეს განლაგება/შეთანხმება მიღწეულია ახალი ტექნიკის გამოყენებით, რომელსაც მსგავსების მატრიცა ეწოდება/Likeliness-Matrix. სხვადასხვა სახის მსგავსების მატრიცები გამოიყენება თითეული ელფერის განთავსებისას. ამას დამატებული, ევოლუციური ალგორითმი გამოიყენება წარმოებული დონეების გასაუმჯობესებლად. ის ბევრ შესაძლო გადაწყვეტას შორის ეძებს და ცდილობს რაიმე გზით შეცვალოს ისინი უკეთესი დონეების შექმნის მიზნით. მორგების ფუნქცია თითოეულ გადაწყვეტას ადარებს და სიმულაციაზეა დაფუძნებული. ის სხვადასხვა აგენტებს იყენებს თითოეული გადაწყვეტის კანდიდატთან სათამაშოდ. შედეგი განსაზღვრავს დონის ხარისხს.

ამ სამაგისტრო ნაშრომის მთავარ იდეას წარმოადგენდა PCG ის, როგორც თამაშის მექანიკის შესაძლებლობების აღმოჩენა, ვიდეო თამაშებზე მათი შესაბამისობამისობის კუთხით აქცენტით. მისი მთავარი მიზანი იყო შესაძლებლობათა დემონსტრაციის უზრუნველყოფა, ისევე როგორც PCG ის თამაშის მექანიკის განვითარების ყოველი ასპექტის უზრუნველყოფა, საერთო უფასო მოხმარების სათამაშო მოწყობილობებისთვის/ძრავებისთვის. საერთო ჯამში ის უნდა წარმოადგენდეს ლაკონურ მიმოხილვას და PCG ს თამაშის მექანიკის განვითარების საწყისს, პროტოტიპის განვითარების მაგალითის სამუშაო პროცესებს. კერძოდ, ნაშრომი ასრულებს ყველა მოთხოვნილ პირობას, რომელიც PCG სთვის უნიკალური თამაშის იდეების სიაშია, თამაშის მოწყობილობის პროტოტიპის სიღრმისეულ განვითარებამდე. ცვალებადი იარაღების; მექანიკის განვითარება აჩვენებს PCG ის, როგორც თამაშის მექანიკის პოტენციალს სამომავლო მუშაობისთვის და დაინტერესებულ დეველოპერებს ეხმარებათ სწრაფად აითვისონ თითეული საჭირო ცოდნა, რათა ახალი თამაშების განვითარება დაიწყონ PCG თამაშის მექანიკებით. ეს უჩვენებს სივრცეს გაუმჯობესებისთვის და წარმოადგენს არსებითი განვითარების განხილვებს PCG თამაშის მექანიკი განხორციელებისთვის, რათა მოულოდნელი შეცდომები არ იქნას დაშვებული. შეჯამებისთვის, ეს ნაშრომი ასახავს და წარმოადგენს საერთო სახელმძღვანელოს PCG თამაშის მექანიკის თამაშში შემოტანისთვის.

გამოყენებული ლიტერატურა

- Jonathan Ackerman. Procedural content generation: Creating a universe. URL: http://blog.rabidgremlin.com/2015/01/14/procedural-content-generationcreating-a-universe/ (visited on 06/20/2016).
- Ernest Adams. Fundamentals of game design. Pearson Education, 2014.
- H.A. Diaz-Furlong and A.L. Solis-Gonzalez Cosio. “An approach to level design using procedural content generation and difficulty curves”. In: Computational Intelligence in Games (CIG), 2013 IEEE Conference on. 2013.
- Jordan Fisher. How to Make Insane, Procedural Platformer Levels. 2012. URL: http://www.gamasutra.com/view/feature/170049/how_to_make_insane_ procedural_.php

- Bay12Games.DwarfFortress.2006.URL:http://www.bay12games.com/dwarves/ (visited on 05/30/2016). [12] Hello Games. About No Man's Sky. 2016. URL: http://www.no-mans-sky.com/ about/
- Marcin Gollent. "Landscape creation and rendering in REDengine 3". In: Game Developers Conference (GDC), San Fransisco. 2014.

დანართი

ვარსკვლავებს შორის დისტანციის გენერირება მოცემული ფესვით

```
double u, v, S;
            double randomRadius;
            float f, q;
            List<Vector3> result = new List<Vector3>();

            for (int i = 0; i < starNum; i++)
            {
                do
                {
                    do
                    {
                        u = 2.0 * Random.value - 1.0;
                        v = 2.0 * Random.value - 1.0;
                        S = u * u + v * v;
                    }
                    while (S >= 1.0);

                    double fac = System.Math.Sqrt(-2.0 * System.Math.Log(S) / S);
                    randomRadius = u * fac;
```

```
                }
                while (Mathf.Abs((float)randomRadius) > gNormSize);
                randomRadius *= gSize;



                f = Random.Range(minF, maxF) * (Mathf.PI / 180f); //Method1
                q = Random.Range(0, maxQ) * (Mathf.PI / 180f);

                var xcoord = (float)randomRadius * Mathf.Sin(f) * Mathf.Sin(q);
                var ycoord = (float)randomRadius * Mathf.Cos(f);
                var zcoord = (float)randomRadius * Mathf.Sin(f) * Mathf.Cos(q);
                Vector3 pos = new Vector3(xcoord, ycoord, zcoord);
                result.Add(pos);
            }

                return result;
```

გალაქტიკის ფორმის გენერაცია მოცემული ფესვით

```
        public Vector3[] GenerateArm(int numOfStars, float rotation, float spin,
double armSpread, double starsAtCenterRatio, float gSize)
        {
            //System.Random rdn = new System.Random();
            Vector3[] result = new Vector3[numOfStars];

            for (int i = 0; i < numOfStars; i++)
            {
                double part = (double)i / (double)numOfStars;
                part = System.Math.Pow(part, starsAtCenterRatio);

                float distanceFromCenter = (float)part;
                double position = (part * spin + rotation) * Mathf.PI * 2;

                double xFluctuation = (Pow3Constrained(Random.Range(0, 0.99999f)) -
Pow3Constrained(Random.Range(0, 0.99999f))) * armSpread;
                double yFluctuation = (Pow3Constrained(Random.Range(0, 0.99999f)) -
Pow3Constrained(Random.Range(0, 0.99999f))) * armSpread;

                float resultX = (float)System.Math.Cos(position) * distanceFromCenter
/ 2 + 0.5f + (float)xFluctuation;
                float resultZ = (float)System.Math.Sin(position) * distanceFromCenter
/ 2 + 0.5f + (float)yFluctuation;

                float resultY = Random.Range(-gSize / 40f, gSize / 40f);

                result[i] = new Vector3(resultX * gSize - gSize / 2f, resultY, resultZ
* gSize - gSize / 2f);
            }

                return result;
```

ვარსკვლავების პარამეტრების გენერირება

```
for (int i = 0; i < starPositions.Count; i++)
            {

                Vector3 pos = starPositions[i];
```

```
                //temp
                var randomRadius = 0;
                var f = 0;
                var q = 0;


                int typeID = GenerateStarType();
                var starMainType = GameStorageController.Instance.StarTypes.First(x =>
x.ID == typeID);

                var color = starMainType.Color;
                var starCls = starMainType.StarClass;
                float MassSolar = (float)Random.Range(starMainType.SolMassMin,
starMainType.SolMassMax);
                float RadiusSolar = (float)Random.Range(starMainType.SolRadiusMin,
starMainType.SolRadiusMax);
                float starTemperature = (float)Random.Range(starMainType.TempMin,
starMainType.TempMax);

                int sts;
                do
                {
                    sts = Random.Range(1, int.MaxValue);
                }
                while (stars.ContainsKey(sts));
                string stn = "GSC_" + sts.ToString();
                int planetnumber = Random.Range(0,
GlobalConstants.MaxDefaultPlanetNumber);
                Star star = new Star(sts, typeID, starCls, stn, color,
starTemperature, MassSolar,
                    RadiusSolar, f, q, randomRadius, pos.x, pos.y, pos.z,
planetnumber, galaxySeed);
                stars.Add(star.StarSeed, star);
                   }
```

პლანეტების პარამეტრების და ვიზუალური ტექსტურების გენერირება:

```
Random.InitState(StarSeed);
        var tmpPlanets = new List<Planet>();
        planets = new List<Planet>();
        for (int i = 0; i < PlanetNumber; i++)
        {
           var planet = new Planet();
           planet.ParentSeed = StarSeed;
           planet.Seed = Random.Range(1, int.MaxValue); //rdn.Range(1, int.MaxValue);
           var tRnd = Random.Range(0, 3);  //rdn.Range(0, 3);
           planet.PlanetType = ((PlanetTypes)tRnd).ToString();
           switch (tRnd)
           {
              case 0:
                 planet.Radius = Random.Range(2000, 14000); //rdn.Range(2000, 14000);
                 planet.AtmosphericPresure = Random.Range(0.1f, 10f);
```

```
                planet.LocalBiosphereAgresivness = planet.CalculateBio(tRnd,
Random.Range(0, 1001));
                break;
            case 1:
                planet.Radius = Random.Range(12000, 120000);
                planet.AtmosphericPresure = Random.Range(100f, 10000f);
                planet.LocalBiosphereAgresivness = planet.CalculateBio(tRnd,
Random.Range(0, 1001));
                break;
            case 2:
                planet.Radius = Random.Range(200, 3000);
                planet.AtmosphericPresure = 0;
                planet.LocalBiosphereAgresivness = planet.CalculateBio(tRnd,
Random.Range(0, 1001));
                break;
        }

        planet.ExploreDifficulty = planet.CalculateExploreDifficulty(tRnd,
Random.Range(10f, 25f), planet.Radius, planet.LocalBiosphereAgresivness);

        double ElemSum = 0;
        double ASum = 0;
        var eCount = GameStorageController.Instance.ElementsList.Where(x => x.Rarity
== 1 || x.Rarity == 2).Count() / 2;
        //planet.CompositionCore = new List<ElementComposition>();
        //planet.CompositionAtmosphere = new List<ElementComposition>();

        List<ElementComposition> compositionCore = new
List<ElementComposition>();
        List<ElementComposition> compositionAtmosphere = new
List<ElementComposition>();



        var atmoCount = GameStorageController.Instance.AtmoElements.Count / 2;
        if (planet.PlanetType == "Solid")
        {
            foreach (var elem in GameStorageController.Instance.AtmoElements)
            {
                var aRnd = Random.Range(0, 2);
                var eRnd = Random.Range(0, 100);
                var checkpoint = Random.Range(0, 4);
                if (aRnd != 0 && checkpoint != 0)
                {
```

```
                    var quantity = aRnd * eRnd;
                    if (Random.Range(0, atmoCount) == 0)
                    {
                        quantity *= 10;
                    }

                    compositionAtmosphere.Add(new ElementComposition { Element =
elem, Rarity = quantity });
                    //planet.CompositionCore.AddExt(new ElementComposition { element =
elem, Rarity = quantity }, planet.ParentSeed, planet.Seed);
                    ASum += quantity;
                }
            }
            foreach (var elem in compositionAtmosphere)
            {
                elem.Rarity = elem.Rarity / ASum;
            }
        }

        if (planet.PlanetType == "Solid" || planet.PlanetType == "Selena")
        {
            foreach (var elem in GameStorageController.Instance.CoreElements)
            {
                if (elem.Rarity == 1 || elem.Rarity == 2)
                {
                    var pRnd = Random.Range(1, 4);
                    var eRnd = Random.Range(50000, 150000);
                    var quantity = pRnd * eRnd;
                    var checkpoint = Random.Range(0, 10);
                    if (checkpoint == 0)
                    {
                        if (Random.Range(0, eCount) == 0)
                        {
                            quantity *= 100;
                        }
                        compositionCore.Add(new ElementComposition { Element = elem,
Rarity = quantity });
                        //planet.CompositionCore.AddExt(new ElementComposition { element
= elem, Rarity = quantity }, planet.ParentSeed, planet.Seed);

                        ElemSum += quantity;
                    }
                }
                else if (elem.Rarity == 3)
```

```
                {
                    var pRnd = Random.Range(1, 4);
                    var eRnd = Random.Range(1000, 5000);
                    var quantity = pRnd * eRnd;
                    var checkpoint = Random.Range(0, 20);
                    if (checkpoint == 0)
                    {
                        if (Random.Range(0, 5000) > 4900)
                        {
                            quantity *= 100;
                        }
                        compositionCore.Add(new ElementComposition { Element = elem,
Rarity = quantity });
                        //planet.CompositionCore.AddExt(new ElementComposition { element
= elem, Rarity = quantity }, planet.ParentSeed, planet.Seed);
                        ElemSum += quantity;
                    }
                }
                else
                {
                    var pRnd = Random.Range(1, 3);
                    var eRnd = Random.Range(10, 500);
                    var quantity = pRnd * eRnd;
                    var checkpoint = Random.Range(0, 300);
                    if (checkpoint == 0)
                    {
                        if (Random.Range(0, 10000) > 9900)
                        {
                            quantity *= 100;
                        }
                        compositionCore.Add(new ElementComposition { Element = elem,
Rarity = quantity });
                        //planet.CompositionCore.AddExt(new ElementComposition { element
= elem, Rarity = quantity }, planet.ParentSeed, planet.Seed);
                        ElemSum += quantity;
                    }
                }
            }

            //es shesacvlelia. prosto rom ar gvqondes uelemento planetebi.
            if (compositionCore.Count == 0)
            {
                var defElem = GameStorageController.Instance.CoreElements.Where(x =>
x.ID == 26).FirstOrDefault();
```

```
                compositionCore.Add(new ElementComposition { Element = defElem,
Rarity = 1 });
                ElemSum = 1;
            }

            planet.Mass = 0;
            planet.Volume = planet.CalculateVolume();
            foreach (var elem in compositionCore)
            {
                elem.Rarity = elem.Rarity / ElemSum;
                planet.Mass += (float)elem.Rarity * elem.Element.Density * planet.Volume;
            }
        }
        else
        {
            foreach (var elem in GameStorageController.Instance.GasElements)
            {
                var pRnd = Random.Range(0, 2);
                var eRnd = Random.Range(1000, 5000);
                var quantity = pRnd * eRnd;
                var checkpoint = Random.Range(0, 5);
                if (quantity != 0 && checkpoint != 0)
                {
                    compositionCore.Add(new ElementComposition { Element = elem,
Rarity = quantity });
                    //planet.CompositionCore.AddExt(new ElementComposition { element =
elem, Rarity = quantity }, planet.ParentSeed, planet.Seed);
                    ElemSum += quantity;
                }
            }
            planet.Mass = 0;
            planet.Volume = planet.CalculateVolume();
            foreach (var elem in compositionCore)
            {
                elem.Rarity = elem.Rarity / ElemSum;
                planet.Mass += (float)elem.Rarity * elem.Element.Density * planet.Volume *
50;
            }


        }


        planet.SemiMajorAxis = GeneratePlanetSemimajorAxis(tmpPlanets);
```

```
            planet.SurfaceRadiation = Random.Range(0.1f, 10f);
            planet.Albedo = Random.Range(0.1f, 0.75f);

            planet.DetectionDifficulty = ((planet.SemiMajorAxis * GlobalConstants.AU /
1000f) / (planet.Radius * planet.Albedo)) / 10000f;

            var tmp1 = (StarLumin * GlobalConstants.SunLuminosity * (1f - planet.Albedo)) /
4;
            var tmp2 = 4f * GlobalConstants.PI * 0.96f * GlobalConstants.StefanBoltzmann *
Mathf.Pow(
               planet.SemiMajorAxis * GlobalConstants.AU, 2);

            var vTmp = Random.Range(0, 2);
            if (vTmp == 0 || planet.PlanetType == "Gas")
            {
               planet.VolcanicActivity = 0;
            }
            else
            {
               planet.VolcanicActivity = Random.Range(1, 11); //es dasanastroikebelia rom 1
ufro xshiri ikos, vidre 10
            }

            if (planet.PlanetType == "Selena")
            {
               planet.SurfaceAverageTemp = Mathf.Sqrt(Mathf.Sqrt(tmp1 / tmp2)) +
GlobalConstants.CelsiusinCalvin + planet.VolcanicActivity *
GlobalConstants.VolcanicTemperatureEffect;
            }
            else
            {
               planet.SurfaceAverageTemp = Mathf.Sqrt(Mathf.Sqrt(tmp1 / tmp2)) +
GlobalConstants.CelsiusinCalvin +
                  planet.VolcanicActivity * GlobalConstants.VolcanicTemperatureEffect +
GlobalConstants.GreenHouseEffect;
            }


            planet.DayLengthInStandardDays = Random.Range(0.1f, 11f);
            planet.YearLengthInStandardYear =
planet.CalculateYearLength(planet.SemiMajorAxis, StarMass);
            planet.IsInHabZone = planet.HabZone(InnerHabitableZoneInAU(),
OuterHabitableZoneInAU());
```

```
                planet.IsPopulated = false;
                planet.PopulationQuantity = 0;
                planet.PrefabName = "";
                planet.ExploreStage = DetectionStages.notDetected;

                planet.CompositionCore = compositionCore;
                planet.CompositionAtmosphere = compositionAtmosphere;

                tmpPlanets.Add(planet);
            }

            int plName = 1;

            planets = tmpPlanets.OrderBy(x => x.SemiMajorAxis).ToList();
            foreach (var item in planets)
            {
                item.PlanetName = StarName + "_P " + plName.ToString();
                item.PlanetDisplayName = item.PlanetName;
                plName++;
                //Debug.Log(item.PlanetName + ",E: " + item.ExploreDifficulty + ",D: " +
        item.DetectionDifficulty);
            }
```

პლანეტების ტექსტურების გენერირების ალგორითმი:

```
  public static Planet CreatePlanet(Vector3 _position, int _planetSeed = -1, string
_blueprintName = "", string _jsonString = null)
        {
            if ((int) DebugLevel > 0) Debug.Log("PlanetManager.cs:CreatePlanet(" +
_position + "," + _planetSeed + ", " + _blueprintName + "," + _jsonString + ")");

            // ბლუპრინტების ლისტის რეაქტივაცია
            Instance.RefreshBlueprintDictionary();

            // თუ არ არის ფესვი მითითებული გამოიყენება შემთხვევითი რიცხვი
            if (_planetSeed < 0) _planetSeed = Random.Range(0, int.MaxValue -
1000000);

            // შემთხვევითი რიცხვის აღება
            float _r = Random.Range(0.0f, 1.0f);

            // Select a blueprint
            float _previousValue = 0f;
            Object _previousKey = null;
            BlueprintPlanet _newPlanetBlueprint = null;
            foreach (KeyValuePair<BlueprintPlanet, float> _k in
Instance._planetBlueprintDictionary)
            {
                // ბლუ პრინტის შემთხვევით ამოღება ალბათობების პროცენტული მაჩვენებლების
მიხედვით
                if (_blueprintName == "")
                {
                    // ბლუპრინტების მოძენბა და შემთხვევითი რიცხვის შედარება
```

```
                    if (_r > _previousValue && _r < _k.Value)
                        _newPlanetBlueprint = _k.Key;
                    _previousValue = _k.Value;
                    _previousKey = _k.Key;
                }
                else
                {
                    // ბლუ პრინტის მინიჭება
                    if (_blueprintName == _k.Key.name)
                        _newPlanetBlueprint = _k.Key;
                }
            }


            // ახალი პლანეტის ობიექტის შექმნა
            GameObject _planetGameObject = new GameObject();
            _planetGameObject.name = "New Procedural Planet";
            _planetGameObject.transform.position = _position;

            // ბლუ პრინტის მოძებნა და პლანეტის კლასის განსაზღვრა
            System.Type _planetClass = System.Type.GetType("ProceduralPlanets." +
_newPlanetBlueprint.GetType().Name.Replace("Blueprint",""));


                // ბლუ პრინტის მინიჭება

_p.SetPlanetBlueprint(Instance.GetPlanetBlueprintIndex(_newPlanetBlueprint), false,
false);


                return null;
            }

            // გენერირებული პლანეტის დაბრუნება
            return _planetGameObject.GetComponent<Planet>();
        }
```