Fakulteti i Shkencave të Natyrës

Dega: Informatikë

Pranoi: Dr. Olti Qirici

Punuan: 1. Anxhela Mehmeti

2. Amarildo Bitri

3. Ergi Kili

4. Kejsi Struga

5. Rigers Duka

Lënda: Animacion Kompjuterik

Tema: 3D Tirana Runner Game

Përmbajtje

Hyrje

Qellimi kryesor I ketij projekti ka qene studimi dhe krijimi I nje loje ne Unity 3D Game Engine. Nje tjeter qellim ishte dhe kuptimi dhe implemetimi I proceseve ne krijimin e nje loje. Loja eshte zhvilluar ne mjedis 3D, ajo konsiston ne gjenerimin e nje *Runner Game*, e cila lejon lojtarin kryesor te vrapoje ne vije te drejte, kundershtar dhe pengesa dalin gjate rruges se bashku me monedha. Lojtarit kryesor I duhet te shpetoje nga kundershtaret si dhe te kape monedhat ne menyre qe te mbledhe sa me shume pike. Loja eshte pershtatur per t'u luajtur dhe ne *Android Mobiles*.

Implementimi

Rregullat e lojes

1. Ne nivelin e pare duhet t'u behet balle pengesave dhe Derrick (karakteri kundershtar) I cili e ndjek lojtarin deri ne fund te lojes.

2. Loja quhet e fituar kur arrihet lojtarin arrin flamurin (objekti qe simbolizon fitoren)

3. Loja eshte e humbur nese nuk ka me energji ose nese lojtari kapet nga kundershtari

4. Gjate lojes shfaqen monedha te cilat shtojne numrin e modhave te kapura nga lojtari

5. Me kalimin e lojes sasia e energjise se lojtarit bie.

Modelimi i Karaktereve

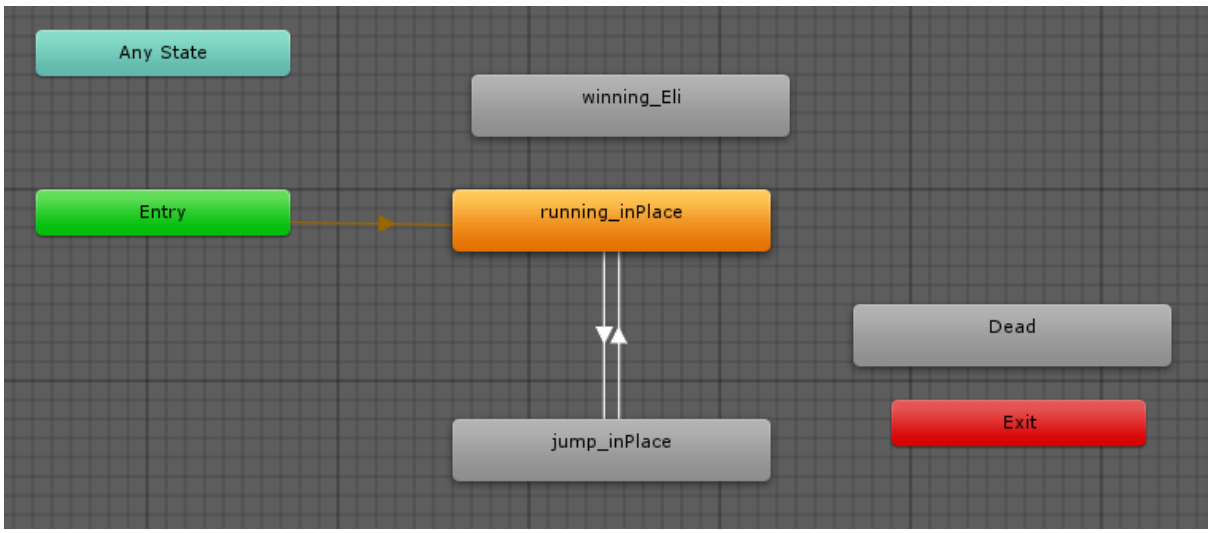
Karakteret kryesore te lojes jane renditur meposhte, ato jane marre si modele te gatshme, disa prej animacioneve te tyre jane modifikuar nga ne permes sistemit ***Mecanim*** I cili lejon qe te aplikohet ***Root Motion*** mbi karaktere.

Sistemi ***Animator*** eshte perdorur per te krijuar gjendje ne te cilat karakteret do te mund te ndodhen gjate lojes. Gjendjeve u vendoset *Motion* qe eshte pjese e modelit te karakterit. Duke qenese te dy kundershtaret (AI), kane te njejtin funksion atehere gjendjet e tyre jane te njejta por ndryshojne levizjet meqe jane karaktere te ndryshme.

Karakteret kryesore:

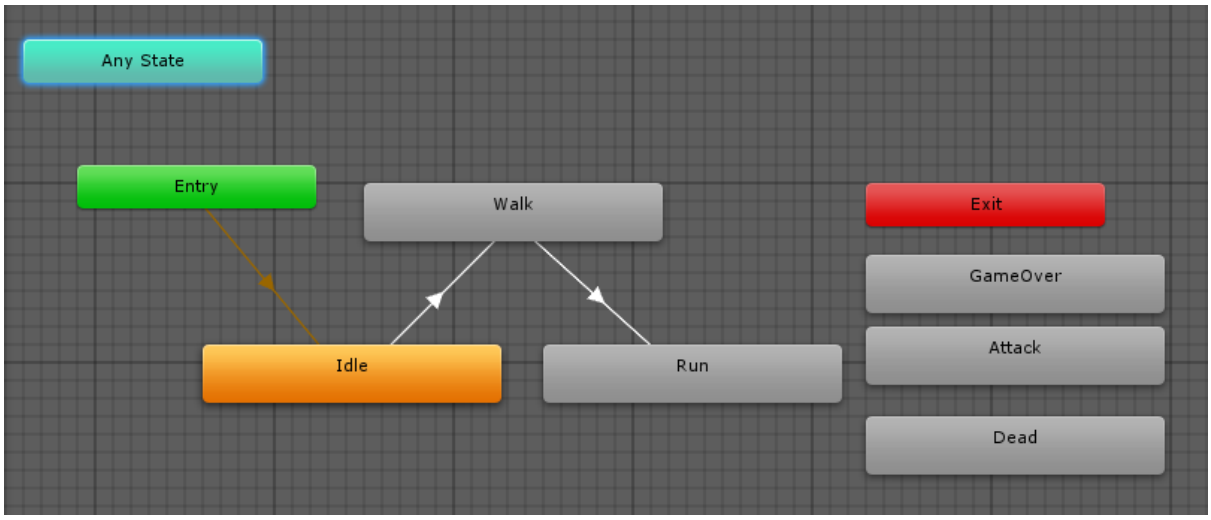
1. *Eli*

Eli I eshte aplikuar nje *Character Controller*, cka do te thote mund te detektoje pengesat (collisions/trupat e tjere qe kane colliders), edhe pa patur nje RigidBody te bashkangjitur. Gjithesesi qe kjo te ndodhe duhet qe levizjet e Eli te kryhen permes funksionit *Move(Vector3)* te CharacterController-it .



2. *Derick (Police) dhe Jill (Zombie)*

Jill dhe Derrick jane dy kundershtaret e Eli. Jill eshte prezente vetem ne nivelin e dyte, nderkohe qe Derrick eshte polici qe ndjek Eli ne cdo nivel. Transformimi I pozicionit te tyre kryhet permes translacionit, perpos levizjeve qe u aplikohen ne momentin kur ndjekin playerin ne path-in e percaktuar nga algoritmi A*.



Modelimi i Mjedisit

Per ndertimin e mjedisit, jane perdorur ndertesa nga Unity Asset Store. Ne nivelin e pare mjedisi perfaqeson nje qytet i perbere nga rruge, ndertesa, trotuare si dhe peme. GameObject-i perkates i mjedisit eshte ruajtur si nje prefab dhe gjenerohet dinamikisht ne Run Time, duke u bazuar ne pozicionin aktual te Player-it. Ne anen e majte dhe te djathte te rruges jane vendosur dhe dy kolona horizontale (borders) te cilat kane Box Collider dhe pengojne karakterin te dale jashte tyre.

Ne nivelin e dyte mjedisi te jep pershtypjen e nje vendi malore ne Tirane, me shtepia te vogla njekateshe dhe me gjelberim. Kemi perdorur komponentin Terrain dhe nuk kemi gjenerim dinamik te mjedisit, si dhe Vendosja e karaktereve ne nivelin e dyte eshte e (pothuajse) e njejte si me ate ne nivelin e pare.

Mekanika

Kundershtaret

Algoritmi ***A**** gjen nje rruge mes dy pikave ne nje harte. Nderkohe qe shume algortime ekzistojne per gjetjen e nje rruge mes dy pikave, A* do te gjeje rrugen me te shkurter mes tyre. Akoma me tej, A* ruan dy lista, lista me nyjet e hapura dhe ajo me nyjet e mbyllura. Lista me nyjet e hapura konsiston ne nyje qe nuk jane eksploruar, nderkohe qe lista me nyjet e mbyllura konsiston ne te gjitha nyjet qe jane eksploruar. Nje nyje konsiderohet e eksploruar nqs algoritmi ka konsideruar te gjitha fqinjet e saj, pra ka llogaritur f e tyre (g+h), dhe I ka vendosur ato ne listen e nyjeve te hapura per eksplorim ne te ardhmen.

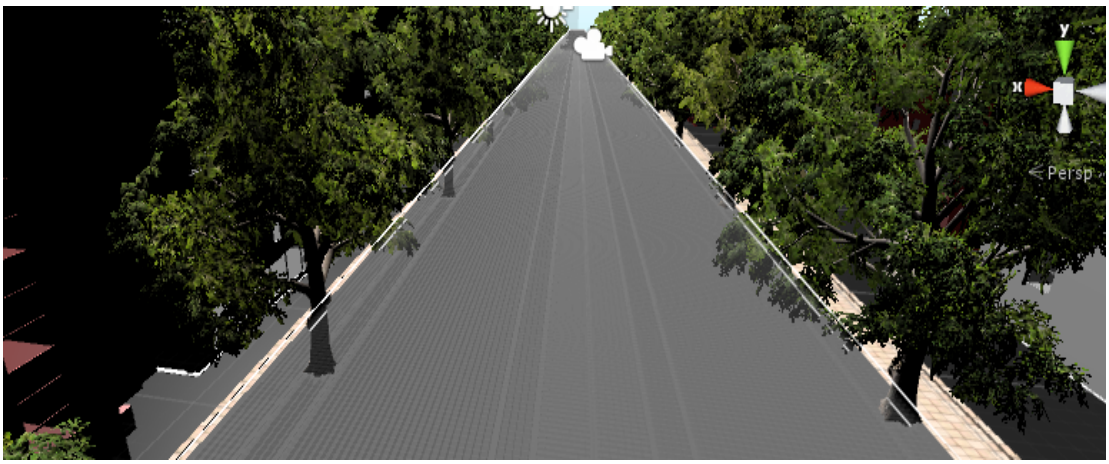
Ne formen e paster te tij A* mund te perdore shume memorie dhe do nje kohe shume te gjate per t'u ekzekutuar. Ekzistojne dy menyra se si mund te permiresojme kompleksitetin e llogaritjeve:

1. Te permiresohet heuristika
2. Ruajtja dhe kerkimi ne listen e hapur dhe ate te mbyllur te jete me eficente

Ne kemi zgjedhur opsionin e dyte, duke I ruajtur nyjet ne nje strukture Turre.

Elementet e A*:

1. **Harta** (ose grafi) eshte hapesira mbi te cilen algoritmi aplikohet per te gjetur nje rruge mes dy pozicioneve. Kjo jo domosdoshmerisht do te thote se harta duhet te jete nje harte gjeografike, me harte do nenkuptohet cdo hapesire 2D/3D e perbere nga nyje, keto nyje mund te jene te kalueshme ose te pakalueshme ne rastin kur ka pengesa dhe nuk mund te llogariten sit e vlefshme nga algoritmi.



Ekzistojne menyra te ndryshme per ndertimin e nje harte ne loje, ne kemi zgjedhur ta perfaqsojme ate permes nje sistemi ne forme Gride, ne menyre te tille qe ajo perbehet nga nyje ku mund te kalohet dhe nyje ne te cilat nuk mund te kalohet. Si rrjedhoje rruga me e shkurter do gjendet mes dy nyje te caktuara (ne rastin tone ***Player-i*** dhe **Kundershtari**).
Krijohet nje 3D Plan ne Unity dhe nje, ketij plani I shoqerojme strukturen e te dhenave ***Node***, pervec faktit qe nje nyje mund te jete e lejueshme per t'u kaluar apo jo, struktura pemban dhe nje tjeter element qe eshte pozicioni I nyjes ne skene, meqe loja jone eshte 3D ky element do te jete nje **Vector3.**
Struktura e dyte e te dhenave eshte **Grid-a** do te kete madhesi (do perfaqesohet nga nje **Vector2**=> *boshti Y ne 2D eshte boshti I Z ne 3D*), rrezen e nyjes (duke qenese cdo nyje do jete pjese e grides), si dhe nje matrice me nyje. Grida eshte e vendosur ne qender te hapesires (0,0,0). Duke patur madhesine e grides mund te llogarisim se sa nyje mund te vendosim brenda saj. Per te kuptuar nese nje nyje eshte e kalueshme apo jo duhet te bredhim ne te gjitha nyjet e grides dhe te marrim pozicionin e nyjeve.

Per te detektuar nese nje nyje eshte e kalueshme perdorim funksionin ***Physics.CheckSphere(Vector3 worldPoint, float nodeRadius),*** e cila do ten a ktheje true ne rast se ka colliders qe bejne overlap me sferen e percaktuar nga pozicioni dhe rrezja ne hapesiren e botes.

2. **Nyjet** jane struktura te dhenash qe perfaqesojne pozicionet ne harte. Gjithesesi harta do perfaqesohet nga nje structure me vete te dhenash. Nyjet ruajne informacionet me te rendesishme per harten qe lidhen drejt per drejt me progresin e *pathfinding*.
Nese nyja eshte e pakalueshme atehere I vendosim nje Layer ne menyre qe ta detektojme kur te bejme kontrrolin ne momentin qe krijohet grida.

   Nyjet do te kene dhe vlerat e **g** dhe **h** si dhe vleren e funksionit **f** qe perftohet nga keto te dyja.

3. **Distanca (heuristika)** perdoret per te percaktuar se sa e pershtatshme eshte nyja per t'u eksploruar.

- ***g*** eshte kostoja per te arritur ne nyjen koherente nga nyja e fillimit.
- ***h*** eshte kostoja e estimuar per te arritur nga nyja koherente ne nyjen qellim (ne rastin tone playeri)

- ***f*** eshte shuma e g dhe h dhe perfaqeson alternativen me te mire per te ndjekur, sa me e vogel vlera e f aq me I mire pathi qe eshte formuar.

4. Gjetja e rruges me te shkurter

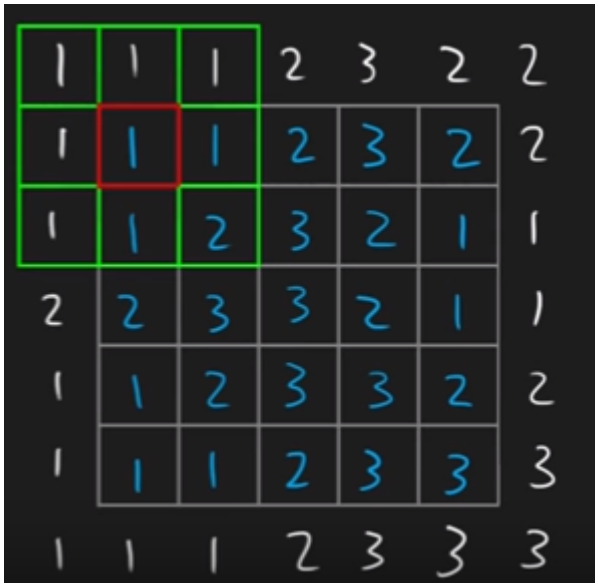
Fillimisht do te na duhet te formojme nje bashkesi me nyjet e hapura (aty ku kerkojme per nyjet me koston minimale te funksionit ***f***) dhe nje me nyjet e mbyllura. Per bashkesine e nyjeve te hapura kemi perdorur Turren (*Heap*) si struktura e te dhenave ku do te ruhen keto nyje, per bashkesine e nyjeve te mbyllura perdoret thjesht nje *HashSet*

Fillimisht ne OpenSet do te vendoset startPos, do te kerkojme ne openSet derisa most e kete me nyje qe nuk jane konsideruar.

Per secilin nga nyjet fqinj te nje nyje (te cilat I marrim permes nje funksionit *Neighbours(),* funksioni I cili merr si parameter nje nyje dhe kthen nje liste me gjithe nyjet fqinje*)*
Me pas ne loop kerkojme nese fqinji eshte I kalueshem dhe nuk ndodhet ne listen me nyjet e mbyllura. Nese rruga(funksioni ***f***) per tek ky fqinj eshte me I vogel se sa per tek nyjet e tjera (dhe fqinji nuk eshte ne listen e hapur) atehere e vendosim ate ne listen e nyjeve te hapura. (ketu perdorim funksionin *GetDistance(n1, n2))*

Peshat e zbutura (kernels, convultion matrix, mask)

Secila nga nyjet do jete nje mesatare e nyjeve qe e rrethojne ate, kernel-I eshte berthama e grides qe do formohet rreth seciles nyje, (3x3 ne menyre qe te kete nje nyje te mesit, ne figure tregohen nyjet me heuristiken e tyre, secila nga vlera e nyjeve do te jete nje mesatare e nyjeve qe e rrethojne ate).
Mbledhim te gjitha velrat Brenda grides se vogel (= 10 dhe e pjestojme me numrin e katroreve Brenda grides se vogel =9, kete rezultat e ruajme ne nje grid te re), keto veprime I kryejme per secilen nga nyjet e grides, dhe marrim terrenin me peshat rezultat. Arsyeja se pse aplikojme nje algoritem te tille eshte qe kundershtari te mos ece fiks ne nyjet qe ka percaktuar algoritmi, po ecja te jete me natyrale.

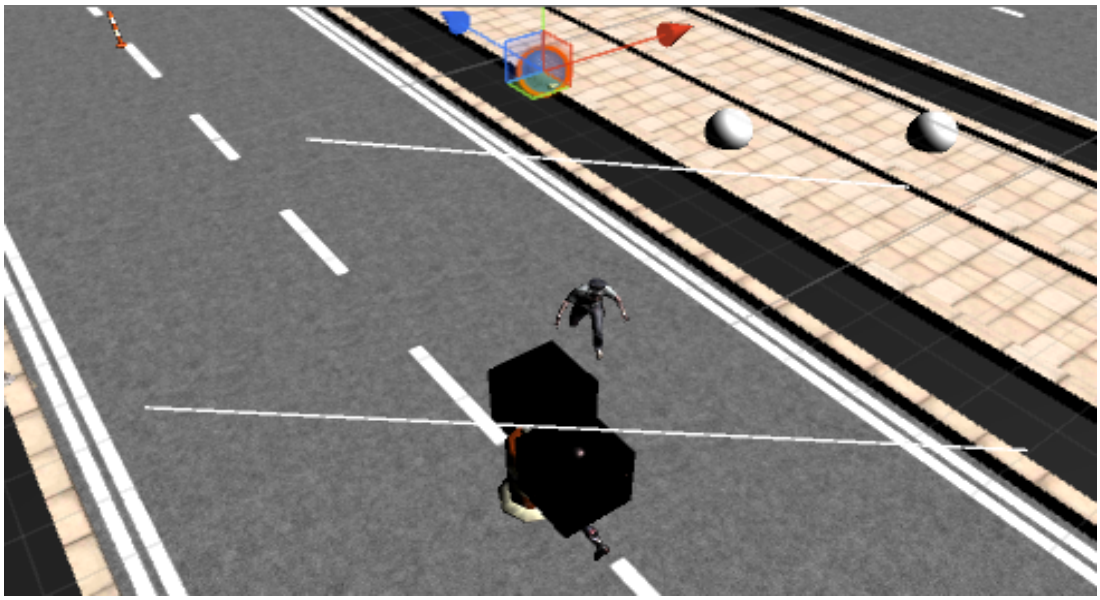


Zbutja e Rruges (rregullimi I shpejtesise qe do marre kundershtari kur ben kthesa dhe ndjek pikat e percaktuara nga algoritmi)

- Krijimi i nje vije kur na jepet nje pike ne ate vije

- Krijimi I nje vije pingul me vijen e krijuar ne vijen ku do ece kundershtari. Ne kete menyre Path-i qe do te krijohet mes pozicionit fillestar (kundershtari) dhe pozicionit qellim (player-i) do te kete nje Vector3[] me pikat qe do te ndiqen, nje Vector3 per kundershtarin dhe nje shpejtesi per kthesat.

- Meqe Playeri eshte ne levizje gjate gjithe kohes atehere na duhet te bejme update path-in ne interval te caktuara kohore.Ne heren e pare qe funksioni do te ekzekutohet nuk do te jete nevoja per te kontrrolluar nese path-I eshte update-uar, prandaj ne frame-in e pare nuk kryhet kontrrolli, nderkohe qe ne frame-et e tjera ky kontrroll kryhet cdo .3s ***(***pritjet gjate frameve kryhen permes Corutina-ve te cilat kane aftesine te pezullojne ekzekutimin e nje funksioni dhe ta vazhdojne ate ne frame-t e mevonshme ***(yield return new WaitForSeconds(.3f))).***
  Nderkohe qe po ndiqet path-i ne duam qe kundershtari ta kete drejtimin e shikimit nga lojtari kryesore, per kete arsye kemi perdorur funksionin Quaternion.LookRotation() I cili merr si parameter drejtimin nga duam te shikojme gjeneron nje quaternion, kete drejtim me pas, ja vendosim parametrit *rotation* te *transform*-it te kundershtarit *(Quaternion.Lerp(transform.rotation, targetRotation,Time.deltaTime * turnSpeed)*



Pamje nga loja na shfaqen nyjet qe perfaqesojne kundershtarin dhe player-in (Gizmos Enabled)

Patrolling

Ndjekja nga kundershtari per tek player-I fillon vetem nese distance mes player-it dhe kundeshtarit eshte nen nje rreze te caktuar, si rrjedhoje, nese kundershtari nuk e ka detektuar player-in ai do te jete duke ecur neper 4 pika ne mjedis.

Ne momentin qe kundershtarit I duhet te ndryshoja piken e vendodhjes (prat e shokje drejte waypoint-it tjeter) atehere, transformojme orientimin e kundershtarit permes nje interpolimi sferik, pra vektoret I trajtojme si drejtime *(jo si pika ne hapesire sic ndodh me interpolimin linear,* ***lerp****)*.

Fillimisht llogarisim pozicionin relative si diferenca e pozicioneve mes pikes dhe kundershtarit. Percaktojme drejtimin e shikimit te kundeshtarit duke perdorur *Quaternion.LookRotation*,

Drejtimi I vektorit rezultant nga interpolimi sferik interpolohet permes kendit dhe gjatesia e tij interpolohet mes gjatesive te vektoreve te kundershtarit dhe waypointit te rradhes, shpejtesia me te cilen ndodh rrotullimi vendoset paraprakisht nga ne si dhe e shumezojme me Time.deltaTime *(Time.deltaTime eshte koha ne sekonda qe ka kaluar qe nga update-I I fundit I frame-it, e perdorim pe rte levizur objektet me nje shpejtesi konstante, duke qenese koha qe zgjat nje frame nuk eshte gjithnje fikse, nese nuk do ta perdornim atehere shpejtesia do ishte e ndryshme ne makina te ndryshme).*

E njejta logjike ndiqet dhe ne rastin kur kundershtari detekton player-in , ai duhet te dale nga gjendja patrolling dhe te ndjeke path-in e percaktuar nga algoritmi A*. Pra e shkepusim kundershtarin nga gjendja *Patroll* dhe e drejtojme orientimin e tij per nga player-i.

Pengesat

Ne secilin nga nivelet kemi nga 2 objekte (ose me shume)te cilat do te behen *Instatiated* ne menyre dinamike.

1. *Pozicioni i pengesave eshte I ndervarur nga pozicionet e:*

- Player-it
- Bordurave (objekte ne terren qe nuk lejojne Player-in te leviz me shume se c'duhet ne drejtimin e bushtit te X-eve)

Fillimisht pozicioni (vlera e X-it) I pengesave vendoset si nje vlere e rastesishme ne nje interval qe percaktohet nga pozicioni I Player-it, nese Player-I eshte shume afer me bordurat ne nuk duam qe pengesat te dalin jashte bordurave, pasi ajo eshte nje zone e palejuar per Player-in, dhe si rrjedhoje ndryshojme dhe vlerat e rastesishme ne varesi te bordurave.

Nderkohe qe dimensioni Y eshte I fiksuar, z eshte I parametrizueshem dhe si rrjedhoje ne vendosem qe ne nivelin e dyte keto vlera te ishin me te vogla (pra pengesat te gjeneroheshin me afer Player-it).

2. *Koha kur do te behen spawn pengesat:*

- ***spawnWait*** merr vlera te rastesishme mes nje maximum dhe nje minimumi, keto dy te fundit jane vlerat qe do vendosin kohen qe do pritet per gjenerimin e pengeses se ardhshme, dhe keto jane te parametrizueshme, si rrjedhoje I vendosim vlera me te vogla ne nivelin e dyte (pengesat te gjenerohen me shpesh).
- ***startWait*** percakton se kur do te filloje gjenerimi I pengesave (meqe nuk duam qe sa te filloje loja te gjenerohen dhe pengesat). Pra ne metoden Start() do te vendosim *StartCoroutine(waitSpwaner())*
- Vete ne coroutine do themi se duam qe kjo te filloje ekzekutimin pas ***startWait*** sekondash *yield return new WaitForSeconds(startWait)*
- Ne rast se loja perfundon pengesat nuk gjenerohen me te pengesa
- Ne rast se Player-I pengohet ne nje pengese I ulet energjia me shpejt

- Ne rast se Player-I pengohet ne nje pengese nuk gjenerohen me te tjera

Sistemi i pikeve

Sistemi I grimcave

Nje tjeter funksionalitet I perdorur ne sistemin e pikeve eshte dhe *Particle System.* Fillimisht ndertuam nje *gameObject,* dhe ketij I bashkangjitem nje *PartcileSystem.* Momenti kur player-I kap nje coin, detektohet nga funksioni *OnTriggerEnter()* (meqe coin-it I kemi bashkangjitur nje *Sphere Collider* me *Trigger* te aktivizuar). Brenda ketij funksioni do te behet Instatiate *gameObject* I mesiperme, ne pozicionin e coin-it.

Disa nga atributet me kryesore qe I kemi vendosur jane

1. Forma: Hemisphere me nje rreze .22

2. Mesh Render-I nga I cili grimcat do gjenerohen si dhe materiali I grimcave

3. Bursts: numri I grimcave qe do te emetohen eshte 20 dhe koha 0.0 meqe duam ta gjenerojme vetem njehere

Referenca

[1] AI Game Programming Wisdom, Steve Rabin

[2] Computer Animation Algorithms & Techniques, Rick Parent

[3] https://unity3d.com/community

[4] https://forum.unity3d.com/categories/graphics.75

[5] http://www.gamasutra.com

[6] http://dota2.gamepedia.com