Game Engine

Rendszerterv

Piller Imre

Tartalomjegyzék

1. Bevezetés

Az alábbi dokumentum a *Game Engine* részletes bemutatására szolgál.

Bizonyos helyen egy-egy konkrét játékra vonatkozó példák szerepelnek. Ezek mindössze azt szolgálják, hogy egyszerűbben meg lehessen érteni (és így oldani is) az adott problémát, és hogy bemutassa, hogy milyen lehetőségeket nyújt majd a játékmotor a vele elkészített játékokhoz.

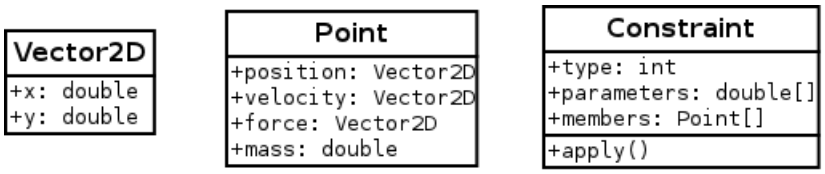
2. A játék fizikája

A játékmotor mindent a fizikára épít. A később bemutatásra kerülő animációk, és mesterséges intelligencia mind csak a kialakított dinamikus rendszerbe avatkozik be. Az irányítás is úgy lesz megoldva, hogy a játékos egy javaslatot tesz, hogy hogyan szeretné változtatni a karakterének a pozícióját, azt egy köztes réteg értelmezi, és kiszámítja, hogy vélhetően milyen beavatkozásokra van szükség, majd módosítja a karakter paramétereit úgy, hogy a fizikai szabályokat figyelembe véve az elvárt mozgás menjen végbe.

A megvalósítás egy részecskerendszerhez (*particle system*) hasonló lesz. Lesznek benne bizonyos elemek, amelyek előre meghatározott szabályok szerint tartják a kapcsolatot egymással. Ezek praktikusan úgy lettek megválasztva, hogy ne legyenek túl kicsik, mert az a számítások mennyiségét jelentősen megnöveli; de ne legyenek túl nagyok sem, mert akkor nehéz belőlük építkezni, több esetre ki kell térni. A következő részben ezeknek az alapelemeknek a bemutatására kerül sor.

2.1. Alapelemek

A játékmotor néhány egyszerű alapelemre építkezik. Az ezekhez tartozó osztályok az 1. ábrán láthatóak.



1. ábra. Az alapvető osztályok

A `Vector2D` osztály általánosan pontok és vektorok megadására is használható. Két mezője van, az `x` és `y`, segítségével a vektorműveletek egyszerűbben elvégezhetők.

A `Point` osztály a tömegpontokat reprezentálja. A `position` a pont helyét adja meg, a `velocity` a pont aktuális sebességét, a `force` a pontra ható erőket a `mass` pedig a tömegét. A mezőket nem tanácsos publikusként megadni általában, viszont ezek az osztályok struktúra jellegűek, inkább csak tárolják az adatot, különösebben ellenőrzésekre nincs szükség.

A pontok egymáshoz nem kapcsolódnak, csak bizonyos erőtér jellegű hatások változtathatják meg a sebességüket. A tömeg magától nem változik, speciális hatások megvalósításakor lehet majd szükség ennek az állítására.

A további alapelemek inkább csak megkötéseknek tekinthetők, amelyek adott számú tömegpontra vonatkoznak. Ezeket együtt használva lehet azután majd az objektumokat felépíteni.

A megkötéseket a `Constraint` osztály reprezentálja. Ebben szerepel egy `type` mező, ami a megkötés típusát azonosítja (*Constraint Type*). Ezeknek a lehetséges értékei a következőek lehetnek:

- `CT_FIXED`: Tetszőleges számú pontra vonatkozhat. Nincs paramétere. A pont rögzítve van, a rá ható erők nem érvényesülnek, nem változtatja meg a pozícióját.

- `CT_RIGID`: Két pont viszonylatában működik. Egy paramétere van, amely megadja, hogy milyen távolságra kell lennie egymástól a két pontnak.

- `CT_FRICTION`: Tetszőleges számú pontra vonatkozhat. Egy paramétere van, amely megadja hogy milyen mértékben csökkenjen a pont sebessége ahogyan halad a közegben. A sebességvektorral ellentétes irányú erőnek is tekinthető. A paraméter értéke nincs kikötés; ha a $[0, 1)$ intervallumban van, akkor a pont lassul, ha pedig nagyobb mint 1 akkor felgyorsítja a mozgását. A 0 érték hasonlóan működik mint ha a pontot rögzítenénk, viszont arra célszerű külön elnevezés, illetve várhatóan kevesebb számítással is jár, mint ez a fajta paraméteres megadás.

- `CT_SPRING_IN`: Két pontra vonatkozik. Egy rugót reprezentál. Első paraméterében a rugó hossza, másodikban pedig az erőssége adható meg. Az erők a két végpontból egymás felé mutatnak. Ha a két pont távolsága kisebb lenne mint a rugó hossza, akkor ez a megkötés nem érvényesül.

- `CT_SPRING_OUT`: Annyiban különbözik a `CT_SPRING_IN`-től, hogy itt az erők kifelé mutatnak, illetve akkor nem vált ki hatást, ha a két pont távolabb lenne egymástól mint a rugó hossza.
- `CT_SPRING`: Az előző kettő kombinációja, külön érték állítható be a ki- és befelé mutató erőknek.
- `CT_GRAVITY`: Tetszőleges számú pontra vonatkozhat. A gravitációs erő megvalósítására szolgál. Paraméterében a g gyorsulásvektor két értékét kell megadni.
- `CT_BLEND`: Három pontra vonatkozik. A három pont két egyenest határoz meg. A megkötés első paraméterezés az egyenesek által elvárt bezárt szöget adja meg, a második pedig az így kialakított rugó jellegű elem erősségét.

A megkötésekhez mindig csak valós érték tartoznak, amit így a `parameters` tömbben típustól függetlenül egységesen lehet tárolni. A `members`-ben azoknak a pontoknak a listája van, amire a szabály vonatkozik. Ennek legalább egy eleműnek kell lennie (üres megkötéseknek nincs értelme).

A megkötések feladata, hogy a pontok értékeit módosítsák. A szabály alkalmazását az `apply` függvény meghívásával érhetjük el. Ez a típustól függően megváltoztathatja a pozíciót, sebességet illetve az erőt.

A játékban szereplő modellek kialakítása ezekre a szabályokra épül. Ha a hagyományos poligonhálót szeretnénk felépíteni, akkor az azt alkotó pontok közé `CT_RIGID` megkötéseket kell megadni.

Egy pont pozíciójának a meghatározása a megkötések használatával a következőképpen néz ki:

- Ki kell számítani a pontra ható erőket a megkötések alapján. Az előző eredmények ebben nem játszanak szerepet. Az erőket a megkötés alkalmazásakor a `Constraint` osztály feladata módosítani.
- Az eredő vektor és a keretidő alapján ki kell számítani a gyorsulást.

```
acceleration = force / mass
```

- A pont sebességét meg kell változtatni a gyorsulásvektora és a keretidő alapján.

```
velocity += acceleration * time
```

- A pontot el kell tolni a kapott sebességvektor és a keretidő alapján.

```
position += velocity * time
```

Mint látható a műveletek végrehatjási sorrendje is számít, ezért a megkötéseknél külön kell választani azokat az eseteket amelyek az erőket, sebességet illetve a pozíciót változtatják, és ebben a sorrendben kell őket végrehajtani. Az egyes típusok csoportosítva:

Módosítás	Típusok
erő	`CT_SPRING_IN`, `CT_SPRING_OUT`, `CT_SPRING`, `CT_GRAVITY`, `CT_BLEND`
sebesség	`CT_FRICTION`
pozíció	`CT_FIXED`, `CT_RIGID`

Amíg a megkötések dinamikus változtatására nincs szükség, elegendő az elemek megtervezésnél kijelölni a megfelelő sorrendet.

2.2. Ütközésvizsgálat

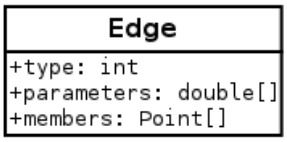
Az ütközések hatékony detektálása létfontosságú az ilyen jellegű játékmotorok esetében. Az eddigiekben arról volt szó, hogy a pontok közötti viszonyokat hogyan lehet megadni, majd az előírt szabályoknak megfelelően hogyan fog változni a helyzetük. Az objektumok határát is ki kell jelölni valahogy. Kézenfekvő megoldás, hogy itt is a kapott pontokat kössük össze.

Mivel minden alakzatot csak egyenesek határolnak, így az ütközések csak úgy fordulhatnak elő, hogy egy adott pont áthalad egy határoló szakaszon. Önmagában nem elég a pont helyzetét vizsgálni, mivel

vékony objektumoknál és nagyobb sebességeknél egyszerűen áthaladhatnának egymáson. Mindig a pont eredeti- és a kapott pozícióját összekötő szakaszt kell metszeni a felülettel.

Jelenleg még úgy tekinthetjük, hogy az objektumok nem lehetnek egymásban, minimális metszés az ütközés során elképzelhető, de teljes átfedés már nem. (Később ezen a rétegek koncepciója árnyalni fog.) Elvileg így elegendő csak az ütközést vizsgálni, mert abból kiderül, hogy melyik az objektum belső- és külső oldala. Az egyes tesztekből lesz látható, hogy ez a gyakorlatban elegendő-e, vagy szükséges felületi normálisokkal is dolgozni. A probléma akkor jelentkezhet, ha egy metszésvizsgálat hibázik valahol, és egy pont egy másik objektum belsejébe kerül, ugyanis ekkor onnan már nem valószínű, hogy ki tudna jönni, és így az objektumok összeragadhatnak.

Az élek megadása formailag és működés tekintetében is nagyon hasonlóak a két ponthoz tartozó megkötések megadásához. Egy él leírásához is egy külön osztály tartozik (`Edge`).



2. ábra. A határoló vonalak tárolására használt osztály.

Ennek a `type` mezője tartalmazza itt is a típus leírását (*Edge Type*), amelynek az értékei a következők lehetnek:

- `ET_RIGID`: Egyszerű határoló vonal. A vonalon áthaladó pontot a szakasz külső határvonalára helyezi.
- `ET_FRICTION`: Ugyanazt végrehajta mint a `ET_RIGID`, viszont ez még csökkenti is az ütköző pont sebességét az ütközés irányával ellentétesen. Az első paramétere azt határozza meg, hogy ez milyen mértékű legyen.
- `ET_STICKY`: Az ütköző pontnak egy, a felület felé mutató sebességvektort állít be, így azt csak egy nagyobb erőhatással lehet onnan eltávolítani, ezzel ragadós felületek hozhatóak létre. (A paraméterek módosításával csúszós felületek is kialakíthatóak így.)
- `ET_SWAMP`: Hagyja a pontot áthaladni a vonalon, majd az áthatolás mértékétől függően állítja be az erőhatásokat, hogy ezzel a pont lelassuljon.

A megkötések és az élek megvalósítása tulajdonképpen megegyezik. Ami a fő különbséget jelenti (és indokolja, hogy miért nem szabad mégsem teljesen egységesen kezelni), az az hogy míg a megkötéseknél mindig lehet tudni, hogy pontosan mely pontokra is vonatkozik a szabály, addig az ütközések kezelésénél ezt előre meg kell határozni.

Az ütközések vizsgálatához ismerni kell a pont ütközés előtti és utáni helyét is. Azért, hogy ne kelljen minden egyes tömegponthoz még egy helyvektort tárolni, ezért az ütközés vizsgálatát a pont mozgatásakor kell detektálni.

Alap esetben minden elmozdított pontot minden egyes éllel össze kellene hasonlítani. Ez hatalmas számításigényt jelentene, ezért a teret fel kell osztani kisebb részekre, ezeket hierarchiába kell rendezni, és így minimalizálni a vizsgálandó esetek számát.

Nem kifejezetten az ütközésvizsgálathoz tartozik, de a konstrukció lehetőséget ad arra, hogy a vizet (vagy hasonló viselkedésű játékbeli elemet) is modellezhetjük ilyen módon. Ehhez még a később részletezésre kerülő réteg koncepciót is meg kell majd vizsgálni.

3. Entitások

3.1. Definíció

A játékban jelenlévő, a játékmenethez szervesen kapcsolódó objektumokat egységekként kell kezelni. Ezeket a továbbiakban *entitás*oknak nevezzük. A megjelenítés és az irányítás miatt van szükség erre, a fizika szempontjából teljesen lényegtelen, hogy az adott pont és élhalmaz milyen részekből épül fel.

3.2. Megjelenítés

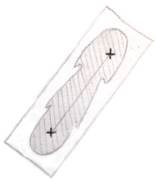
A megjelenítés abból áll, hogy a pontok helyzetéből a játékmotor kirajzolja magát az entitást. Az élek ebben nem játszanak szerepet, mivel azokat is a pontok határozzák meg.

A megjelenítés módjai

- Textúrázott háromszögek segítségével. Ez a legegyszerűbb eset, ennél elegendő az entitás három pontját kiválasztani, majd hozzá megadni a textúrát.
- Egy ponthoz megadható egyetlen sprite. Egyszerűen kivitelezhető vele a hagyományos, részecskerendszerek esetén (pl.: tűz, füst) megjelenítés.

3. ábra. Sprite egy pont megadásával.

- Két ponthoz lehet megadni egy téglalap alakú textúrarészt, amelynek így a nyújtása is változhat a pontok távolságától függően.



4. ábra. Két pontra feszített téglalap alakú textúra.

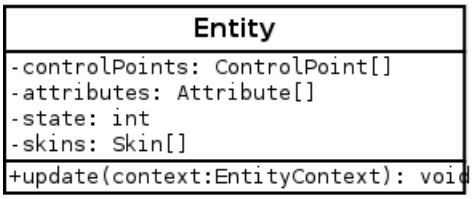
- Három ponthoz szintén megadható téglalap alakú textúra is. Ekkor mindkét irányba nyújtható a textúra.

5. ábra. Három pontra feszített téglalap alakú textúra.

Azoknál az entitásoknál, amelyeknek meg kell tudni fordulni (tipikusan ilyenek a karakterek) a pontok elhelyezkedésétől függően változtatni kell a megjelenített részeit. Ehhez is szabályokat kell megadni (például a pontok közötti távolságokra, vagy a bezárt szögekre).

Az entitásnak meg kell adni kontrollpontokat, amivel a pozíciója jól leírható. Ezeknek nem feltétlenül kell egybeesniük az entitást alkotó pontokkal, mivel bizonyos esetekben más szabályok vonatkoznak rájuk (például a forgatásnál a rögzített fizikai szabályokat figyelembe nem lehetne megfordulni.)

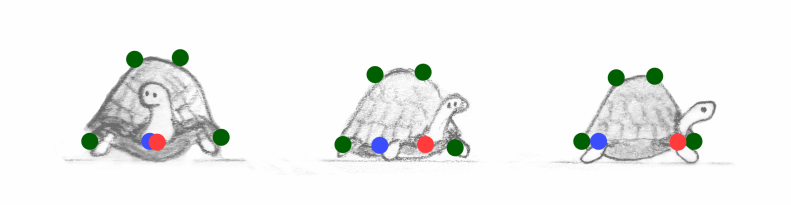
Egy entitás leírására az `Entity` osztály szolgál.



6. ábra. Az entitásokat reprezentáló osztály

Itt már szükséges, hogy a kontrollpontokhoz neveket rendeljünk, hogy meg tudjuk őket különböztetni. Ez nem igényli az eredeti `Point` osztály bővítését, mivel az entitások felépítésekor ismertek a kitüntetett pontok, így elég azokra csak hivatkozni az entitásból. Egy ilyen pont adatait `ControlPoint` osztályok tárolják majd. Ez annyiban bővebb mint a `Point`, hogy van egy `name` mezője is.

7. ábra. Egy egyszerű forgatható modell.



8. ábra. A modell pontjai (zöld) és két kontrollpont (piros és kék).

A játékmotor a Python nyelvet (vagy annak egy egyszerűsített változatát) fogja használni majd a viselkedés (értve ez alatt a megjelenítést és a mesterséges intelligenciát is) leírására. Legyen például egy `Turtle` nevű entitás és legyen két pont, amelynek a neve `head` illetve `tail`. Ezeket azután egy feltételrendszerbe illesztve kiválaszthatjuk, hogy éppen mit is kell megjelenítni, például:

```
distance = head.x - tail.x
if abs(distance) < 10:
    Turtle.skin = "front"
else if distance > 0:
    Turtle.skin = "right"
else
    Turtle.skin = "left"
```

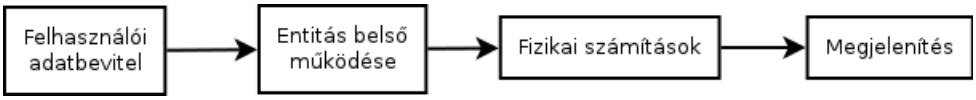
Ehhez az egyes *skin*-eket is definiálni kell. Ezeknél mindig szükséges a név megadása.

3.3. Irányítás

A játékmotor az animációt a hagyományos értelemben nem támogatja. Nem kerülnek bele előre beállított pozíciók, vagy szögek, hanem csak a mozgássor véghezviteléhez szükséges leírások, amelyek alapján a fizikai számításokhoz szükséges jellemzők változni fognak. Ez is azt a célt szolgálja, hogy a játék kinézete minél egységesebb legyen, az előírt szabályok mindvégig érvényben maradnak, csak annak a keretei között lehet bármilyen változtatást véghezvinni a játékban.

Az entitások tulajdonképpen állapotgépek. Véges számú állapottal, illetve lehetséges állapotátmenettel rendelkeznek. A belső működéshez is programozni kell, viszont ezt már elég a játékmotor szkriptnyelvével megoldani.

A felhasználó a karakterét nem közvetlenül irányítja, csak tanácsokat ad arra vonatkozóan, hogy merre szeretne elmozdulni, és mit fog csinálni. Maga az elmozdulás több fázison keresztül megy végbe, míg eljut a megjelenítésig (9. ábra).



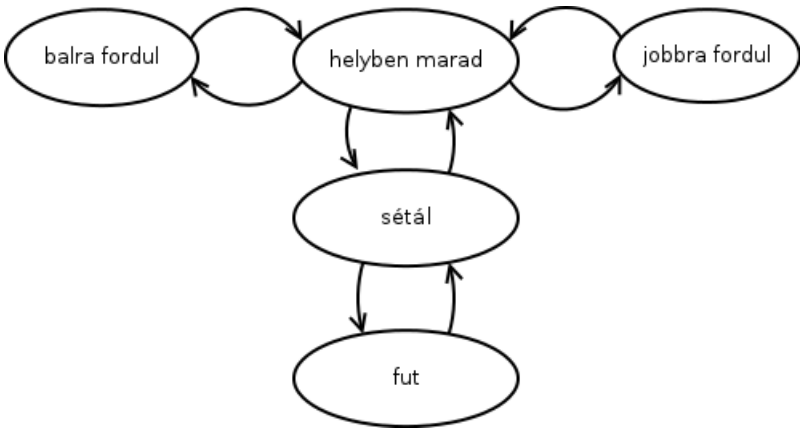
9. ábra. A karakterek irányításának folyamata.

A karakter irányítását a játékos szemszögéből egy későbbi fejezet tárgyalja.

A nem a játékos által irányított entitások mozgatásának két fő, egymásra épülő összetevője van:

- **absztrakt terv szint**: az entitás valamilyen tervet szeretne végrehajtani (pl.: támadni, védekezni, gyűjtögetni).
- **véghezviteli szint**: a konkrét, a terv végrehajtása érdekében tett műveletek (pl.: mászás, fordulás, ugrás).

Mindkettő állapotokkal jellemezhető. Például egy mozgássorozathoz tartozó állapotdiagram látható a 10. ábrán. A két szintnek kommunikálnia kell egymással, mivel a konkrét mozgás a tervből következik, illetve a terv kialakításához ismerni kell a környezetet.



10. ábra. A mozgásokhoz tartozó állapotok.

Az állapotoknak a nevét kell tárolni, hogy később a szkriptekből lehessen rájuk hivatkozni (`state` mező).

A végrehajtásnál a fizikai számításokba már nem lehet közvetlenül belenyúlni, csak a pontokra ható erőket lehet állítani. Amikor az entitás például egy adott irányba halad, akkor egy olyan vektort kell

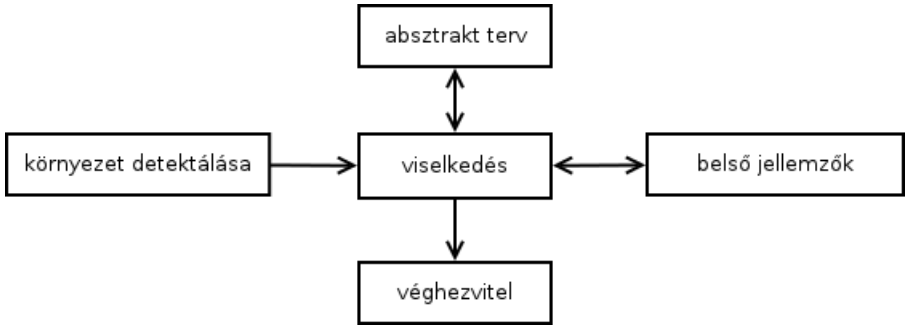
megadni, mint amivel húznánk a pontot. Ezt elég csak néhány kontrollpontra elvégezni, mivel a többi részét a fizikai rész már megoldja.

Egy entitáshoz tetszőleges számú belső jellemzőnek kell tartoznia. Ez az entitás osztályban közvetlenül nem oldható meg, mivel akkor minden egys elkészítendő entitástípushoz (vagyis a játékban szereplő konkrét modellekhez) származtatni kellene egy külön entitás alosztályt. E helyett egy külön `Attribute` osztály szolgál majd arra, hogy a *név - érték* párokat tárolja. Az érték itt lehet egy szám (egész vagy lebegőpontos is) vagy szöveg. Ezeket az `attributes` mező tárolja egy listában.

3.4. Környezet detektálása

A nem játékos karaktereknek kell tudni tájékozódniuk a játék környezetében. Mivel a fizikai szabályok minden esetben érvényesek, ezért ha ez nem így lenne az sem jelentene különösebb fennakadást.

A környezet detektálását csak a látómezők vizsgálatával praktikusan nem lehet megoldani, mivel nagyon nagy lenne a számításigénye (ütközésvizsgálat, képfeldolgozás), ezért egy egyszerűbb modell szükséges. Az irányításnál bemutatott szinteket ez a detektálás, és a belső működések alapján hozott döntések kötik össze (a 11. ábra).



11. ábra. Az entitás viselkedése.

Az entitásoknak egymás detektálásához elegendő a kontrollpontjaikat vizsgálni.

A játékmotor az objektumok láthatóságát úgy oldja majd meg, hogy egy speciális `ET_PARAVAN` értékkel kijelöli azokat a részeket, amelyek takarják egymást. Ennek a száma lehet jóval kevesebb, mint a más típussal megadott határoló vonalaké.

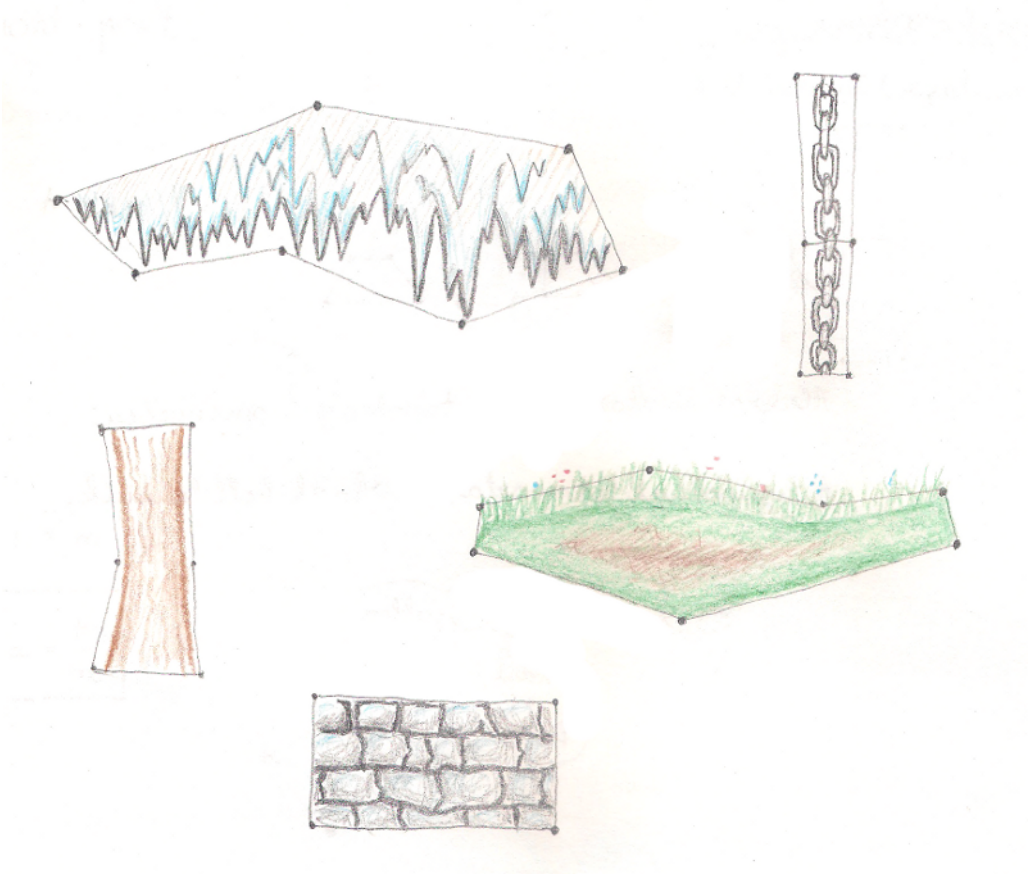
A magasabb szintű detektálás (pl.: ellenőrzési pontok, ellenfelek) során az entitásnak elég annyit tudnia, hogy melyik pont milyen irányban, milyen távolságra van tőle. Ezeket az adatokat a `World` objektum szolgáltatja az entitás `update` metódusán keresztül egy `EntityContext` objektumként.

4. Anyagok, felszínek

A játékmotor olyan játékokhoz készül, amelyekben a játékos nagy területeket szeretne majd bejárni. A karakterek, és a térkép egyes részeit külön-külön is meg lehetne rajzolni, viszont az rengeteg munkát venne igénybe, és akkor még olyan probléma is előfordulhat, hogy a térkép nem elég egységes. Egyszerű másolás is elképzelhető, aminél meg pont az lenne a gond, hogy túlságosan monoton képet kapunk. A megoldást itt a felszínek generálása jelenti. Létre kell hozni egy olyan modellt, amely

- elég általános ahhoz, hogy leírja a játékban előforduló összes, nem külön rajzként beillesztett képelemet,
- vannak benne véletlen tényezők, amivel így nagy terület is kitölthető látványos ismétlések nélkül,
- a különböző felülettípusokat át lehet vinni egymásba (interpolálhatóak),
- kicsi a számításigénye.

Az alábbi ábra bemutat néhány esetet, hogy hol is érdemes ezeket használni.



12. ábra. A fő határoló vonalak.

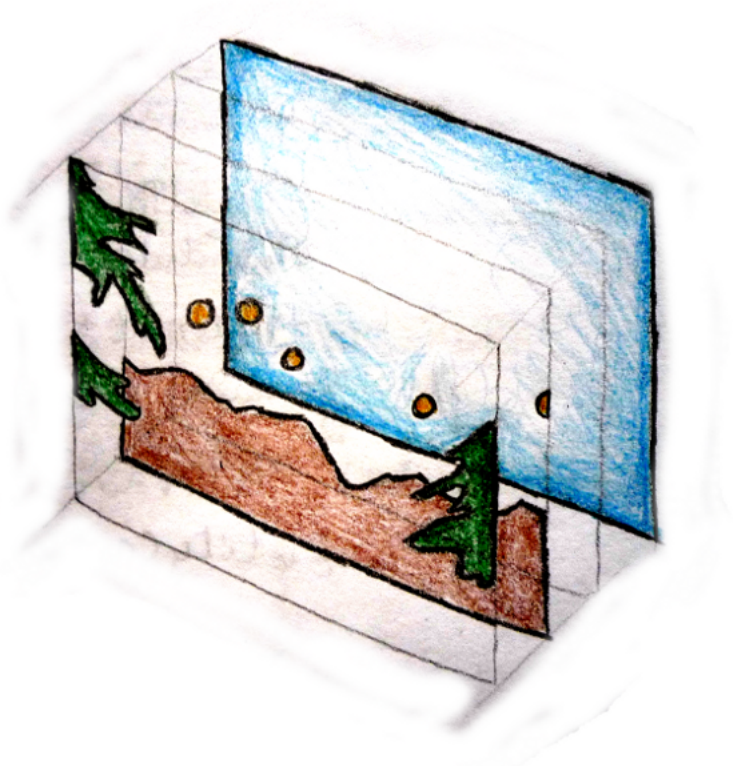
Az anyagokat mindig egy poligonon belül lehet megadni. Meg kell különböztetni azokat az eseteket, amikor a generált textúra teljesen kitölti a sokszöget, illetve azt amikor csak az egyes széleit érinti, vagy csak a közepén van. Ennek a beállításához elég egy jelzőbittel beállítani, hogy az hozzátartozik-e az anyaghoz.

A felszínek procedurális generálása az entitásoknál bemutatott szkripteket használó megoldással képzelhető el. Ezeknek is kell hogy legyen típusa, illetve attribútumai. Belső állapottal ezek már nem rendelkeznek. A generálást a térkép felépítésekor kell elvégezni, illetve nagyobb térképeknél folyamatosan, ahogy a karakter halad.

5. Rétegek kezelése

A játékmotor a következő rétegeket fogja kezelni (13. ábra):

- *Background*: Leghátul helyezkedik el. Csak illusztrációként szolgál, a játék eseményeire nincs befolyással.
- *Interaction layer*: Olyan objektumok helyezkednek el benne, amelyekkel az entitások interakcióba tudnak lépni. Ilyenek például a kötelek, csapdák, felszedhető dolgok. Ezek direkten nem befolyásolják a mozgást, az entitás külön jelezheti, hogy ezeket szeretné elérni, vagy csak egyszerűen áthalad rajtuk.
- *Collision layer*: Ebben helyezkedik el az összes lényeges szereplő. A rétegben lévő entitások ütköznek egymással, nem lehetnek egymás alatt, mivel csak egy rétegről van szó.
- *Foreground*: A háttérhez hasonlóan ennek is csak illusztratív szerepe van. Ez a réteg az összes többi felé kerül.



13. ábra. Az egymáson lévő rétegek.

A rétegek megvalósítására egy nagyon kényelmes megoldást biztosít az OpenGL. Igaz, hogy a játék alapvetően síkban játszódik, viszont van lehetőség a Z-buffer használatára is, és így elég csak 4 különböző mélységet megválasztani, és a kirajzolás sorrendje tetszőleges lehet. Ez főleg a két középső réteg miatt érdekes, mivel ott a tömegpontok lehetnek közösek.

A rétegek kezeléséhez az entitásokat, illetve az azokat alkotó pontokat és megkötéseket rétegekbe kell szervezni. A `Layer` objektumnak az lesz a feladata, hogy ezek adatait összefogja, a vizsgálatokat elvégezze illetve megjelenítse a végeredményt.

6. A karakter kezelése

A *karakter* kifejezés ebben a részben kifejezetten arra az entitásra vonatkozik, amelyet a játékos irányít.

6.1. Kamerakezelés

A kamera kezelésének az alábbi módjai kerülnek majd bele a játékmotorba:

- *static*: A kamera fixen marad, mindig csak egy adott rész látszódik a térképből.
- *auto*: A kamera helye és nagyítása a térképnek megfelelően külön be van állítva, az egyes kameraállások között interpolációval megy át a karakter pozíciójának megfelelően. A különbség a statikussal szemben, hogy itt a váltás folytonosan történik, míg ott rögtön a másik pozícióba kerül a kamera.
- *centralized*: A karakter mindig középen marad, csak a térkép mozog. A gyakori ugrások miatt ezt lehet csak vízszintes irányban alkalmazni. Ennek egy alesete, amikor a pozíciót a képtartomány két szélétől adott távolságra állítjuk be, és csak szükség esetén görgetünk. Ez általában jobb, mivel kevésbé terheli a szemet, mint ha az egész kép folyamatosan mozogna.
- *axial*: Az egérkurzor és a karakter pozícióját összekötő szakasz közepe mindig a képernyő közepén van. Megadható úgy is, hogy az egérkurzorral csak a nézeti irányt adjuk meg, majd a karaktertől való távolsággal jelezzük a szükséges nagyítást.

6.2. Az irányítás módja

Az irányításhoz megvalósításához a játékmotor a billentyűzet és az egérbemeneteket fogja támogatni. Ezek jól kezelhetők az SDL-el, és az `InputManager` osztály absztrakciós réteget biztosít a játékmotor számára.

Az irányítás az entitások és a szabályrendszer kialakítása miatt nem lehet teljesen automatikus. A játékos a mozgások irányát adja meg, a karakternek korrigálni kell még az apróbb akadályok miatt keletkező hibákat. (Például egy rögösebb útvonalon nem szabad a játékost arra kényszeríteni, hogy folyamatosan ugráljon, viszont azt sem szabad engedni, hogy csak úgy átrohanjon rajta. Ilyen esetben a karakter megbotlik, és hosszabb időbe telik átjutni az akadályokon.)

A karakter irányításánál az alábbi műveleteket lehet használni:

- mozgás balra/jobbra
- ugrás
- nézeti irány állítása (ha szükséges)
- interakció (pl.: tárgyak felvétele, kapcsolók, ajtók)
- egyéb speciális műveletek (pl.: támadás, védekezés)

Az ezek használatos szükséges eseményeket már a játékmotor konfigurációs állományaiban kell majd definiálni.

7. A játékmotor függvénykönyvtára

Ez a szakasz az előző részekben bemutatott elemekhez elkészült függvénykönyvtárat taglalja. A játék fizikájának megvalósításához, a térkép generálásához illetve az entitások viselkedéséhez is rengeteg számításra van szükség.

7.1. Matematikai és fizikai függvények

A fizikai rész az `engine/physic` jegyzékbe került.

7.2. Generátorok

A felületeket és az entitások egy részét is generálni kell. Ehhez hatékony véletlenszám-generátorok kellenek.

7.3. Mesterséges intelligencia

A viselkedés megtervezéséhez szabályrendszereket kell definiálni. A játék hitelesebbé tehető, ha e mellett még más módszereket (pl.: Fuzzy halmazok, mesterséges neurális hálózatok) is alkalmazunk.

8. A játékmotor felépítése

8.0.1. A megjelenítés

A játékmotor a megjelenítéshez *OpenGL* függvénykönyvtárat használja, az ablak inicializálását, és más platformfüggő feladatokat *SDL*-el végzi. Ez a következőképpen néz ki:

```
SDL_Init(SDL_INIT_VIDEO);
const SDL_VideoInfo* info = SDL_GetVideoInfo();
int vidFlags = SDL_OPENGL | SDL_GL_DOUBLEBUFFER | SDL_RESIZABLE;

if (info->hw_available) {
    vidFlags |= SDL_HWSURFACE;
}
else {
    vidFlags |= SDL_SWSURFACE;
}

if (isFullScreen) {
    vidFlags |= SDL_FULLSCREEN;
}

int bpp = info -> vfmt -> BitsPerPixel;
SDL_SetVideoMode(windowWidth, windowHeight, bpp, vidFlags);

// Set OpenGL
glViewport(0, 0, windowWidth, windowHeight);

// Disable Depth test
glDisable( GL_DEPTH_TEST );

// Set transparency
glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);

// Orthogonal view
glOrtho(0, windowWidth, windowHeight, 0, -1, 1);
```

Ebben már a kameratranszformációi is szerepel, mivel a játékmotor mindig két dimenziót használ, így elég beállítani azt az inicializáláskor.

8.1. Erőforrások betöltése

Itt az *erőforrás* minden olyan adatot jelöl, amely szükséges ahhoz, hogy a játékmotor meg tudja jeleníteni a játék virtuális világát. Mivel a játékmotor offline játékok készítését teszi majd lehetővé, ezért nem kell a hálózati erőforrásokat számításba venni, minden adat fájlok formájában áll rendelkezésre.

8.1.1. Konfigurációs fájlok

A konfigurációs fájlok (a fejlesztési fázisban mindenképpen) szöveges fájlok lesznek. Ebben olyan alapvető információk szerepelnek majd, mint a főbb elérési utak, paraméterek az indításhoz, verziószám, esetlegesen hibakereséshez beállítások. A játékok készítése során ezekben kell mindent beállítani, hogy a játékmotor be tudja majd tölteni az indítóképernyőt, a menüket, majd magát a játékot.

8.1.2. Entitások adatai

Minden entitástípus egy külön jegyzékben kap helyet. Később tömörítés használatával ezek külön fájlokba is kerülhetnek.

A pontok és az élek szerkezete elég egyszerű, ezért ezeket érdemes bináris fájlokban tárolni majd. A kezdeti változatokban a tesztelést elősegítendő ezek egy, az XML-hez hasonló, de annál könnyebben szerkeszthető, tömörebb és gyorsabban feldolgozható fájlformátumban szerepelnek.

A viselkedés leírását egyszerű szöveges szkriptfájlok tárolják majd.

8.1.3. Felületek adatai

Szintén szöveges formában lesznek tárolva. Ezek mérete várhatóan így sem lesz túlságosan nagy, mivel ezek csak az általánosan használható felület-generátorok paramétereit tartalmazzák.

A textúrákat tároló fájlok formátuma PNG vagy TGA lesz. A PNG elterjedtebb, viszont a betöltés lassabb lehet. A játékmotor tervezésénél szempont, hogy minél több összetevőt procedurálisan állítson elő (így kisebb méretet és nagyobb változatosságot elérve), ezért a PNG képek használata nem biztos, hogy jelentősen lassítja majd a működést.

8.1.4. Hanghatások, aláfestő zenék

A hanghatásokat és az aláfestő zenét érdemes külön kezelni. A hanghatásokhoz inkább az egyszerűbb WAV formátumokat fogja majd a rendszer használni, a hosszabb zenékhez pedig OGG vagy MP3 fájlokat.

8.2. A játékmotor működésének folyamata

A fájlokat a `Loader` osztály fogja beolvasni. Mivel az ezekben lévő adatok inkább csak útmutatások a virtuális világ előállítására vonatkozóan, ezért az ebből származtatott `Factory` objektum lesz az, amely ténylegesen, a `World` objektum számára előkészít minden szükséges dolgot. A betöltési sorrend a következő kell, hogy legyen:

- konfigurációs fájlok betöltése
- a pontok beolvasása és beillesztése
- entitások létrehozása (esetlegesen új pontok beiktatásával)
- textúrák beolvasása
- felületek generálása
- szkriptek feldolgozása
- virtuális világ inicializálása

Az utolsó fázisban a `World` objektumnak le kell ellenőríznie, hogy minden beolvasás rendben ment-e, minden megfelelően lett-e kigenerálva és hogy nem maradt-e bármilyen, a működést befolyásoló tényező. Ez után már a `mainLoop` következik, és ezzel elindul maga a játék.

A játékmotor működése ezek után a következő lépésekből áll:

- Ellenőrzi a felhasználói bemeneteket.
- Végigmegy az entitások listáján, majd frissíti azokat a keretidőnek megfelelően.
- A fizikai motort használva végrehajta a keretidőnek megfelelő változtatásokat.
- Végigmegy az entitásokon és kirajzolja azokat.
- Ellátja a kirenderelt képet a megfelelő képhatásokkal.
- Megjeleníti a felhasználói felületet.

9. Játékok fejlesztéséhez biztosított eszközök

9.1. Keretrendszer

A keretrendszer arra szolgál, hogy a játékmotor bizonyos részeit interaktív módon kezelni, tesztelni tudjuk már a fejlesztés korai fázisaiban is. Két fő része van:

- a játékmotor objektumai, melyek ki lehetnek bővítve plusz paraméterekkel és metódusokkal, hogy segítsék tesztelést,
- egy parancsértelmező, amelyel a játékmotor által létrehozott világba be tudunk avatkozni. Erre fognak majd épülni a későbbi grafikus felületek.

A játékmotor aktuális részét, illetve a parancsértelmezőt érdemes külön szálban kezelni. Ez úgy valósul meg, hogy a program először inicializálja a játékmotort egy külön szálban, majd elindítja a parancsértelmezőt.

A kommunikáció megvalósításához egy külön üzenetkezelő (`MessageHandler`) objektumra van szükség. Ez a program objektumai között egy rugalmasabb kommunikációs csatornát biztosít, és szabályozottabb, mint ha azt egyszerűen csak globális változókkal oldanánk meg.

A `MessageHandler` egy statikus objektum, ezt az összes objektum látja.

Az üzenetek küldéséhez és fogadásához az objektumoknak előbb fel kell iratkozniuk. Ezt a

```
MessageHandler.register("objectName", receive_function);
```

A feliratkozó objektum neve itt `objectName`, és a kapott üzeneteket a `receive_function` függvénnyel fogja majd kezelni. Ennek a függvénynek csak egy `void*` típusú argumentuma van, ebben kell lennie annak az üzenetnek, amit az objektum kap. A formátumra vonatkozóan nincs megkötés, az objektumnak saját magának kell ellenőrízni azt.

Az üzenetek küldése például a következő képpen nézhet ki:

```
MessageHandler.send("objectName", message);
```

Az első paraméter értelem szerűen a címzett, a `message` pedig az az üzenet, amit továbbítani kell. A `MessageHandler`-nek tulajdonképpen csak az a feladata, hogy a nevek listájából visszakeresse a megfelelő függvényt, majd meghívja azt az üzenetként kapott paraméterrel.

Egy névhez több függvény is tartozhat, illetve több névhez is tartozhat egyazon függvény. A küldésnél hiba akkor keletkezhet, ha a név nem szerepel a listában.

A listáról le is lehet iratkozni az `unregister` függvénnyel. Ennek többféle alakja is van

```
MessageHandler.unregister("objectName");
```

Ez az összes `objectName` nevű bejegyzést törli. A másik lehetőség a

```
MessageHandler.unregister("objectName", receive_function);
```

alak, amely csak az adott név-függvény párt távolítja el. Ha egy üzenet kezelő függvényt szeretnénk eltávolítani, akkor a

```
MessageHandler.unregister(receive_function);
```

megadás is lehetséges.

9.2. SandBox

A tömegpontok, és az alapelemek jellemzőinek beállítása nem egyszerű feladat. Ezeket érdeme kipróbálgatni, és megnézni, hogy melyik a legjobb paraméterezés az adott helyzetben.

9.3. Material Editor

Az anyagok adott paraméterek melletti generálását nézhetjük meg vele, és adott névvel együtt elmenthetjük azokat.

9.4. Entity Editor

Az entitások összeállítása több lépcsőben történik:

- Meg kell adni a tömegpontokat, illetve az azok viszonyait leíró szabályokat.
- Meg kell határozni az entitás körvonalát, és hogy az milyen jellemzőkkel rendelkezik.
- A tömegpontok különböző helyzetéhez meg kell adni az entitás megjelenítési módját.
- Be kell állítani az alapvető mozgáskoordinációt, mozgástípusokat.
- Létre kell hozni az állapotokat, és ezzel meghatározni az entitás viselkedését.

9.5. Map Editor

Ezzel a már elkészült entitásokból lehet összeállítani a játék teljes térképét, anélkül, hogy az entitások belső működésével komolyabban foglalkozni kellene (az az Entity Editor feladata).

9.6. GUI Editor

Ahhoz, hogy a játéknak valóban program kinézete is legyen, meg kell tervezni a menüket is. Ennek a szerkesztőnek az esetében is inkább csak a már elkészült komponensek összerendezéséről van szó, nem kell, hogy külön bele legyen integrálva egy képszerkesztő program, mivel azok már egyébként is elérhetőek.