Számítógépes Grafika nagybeadandók

A beadandó feladattal a félév végén megajánlott jegy szerezhető. Aki elégtelentől különböző jegyet szerez a beadandójára, annak nem kötelező géptermi ZH-ra jönnie, a félévi jegye a megajánlott jegy lesz. **Beadandót csak az mutathat be, aki a félévközi számonkérések végén legalább 0 ponton áll** (tehát a két kisbeadandóra, a +/- okra és az órai munka során szerezhető pluszpontokra kapott összpontszáma legalább 0)**!**

Aki beadandót akar készíteni, annak EZEN a form-on kell ezt jeleznie, a választott feladatával együtt. Jelentkezni **2019. december 15. vasárnap 23:59**-ig lehet.

Az elkészült megoldások beadási határideje **2019. december 31. kedd 23:59**. A bemutatásra **2020. január 03. péntek 9:30 - 11:30, a géptermi UV idejében** kerül sor! (Előzetesen szükséges beadni!) Nem kell feljelentkezni a Neptunban vizsgaalkalomra a beadandót bemutatóknak.

A választott feladatot OpenGL-ben kell megoldani. A programban mindenki csak a saját gyakorlatvezetője által tanított API-kat és nyelveket használhatja. A feladatokra egyéni megoldásokat kell benyújtani, **csapatmunka nem engedélyezett**. Amennyiben valamilyen speciális modellre hivatkozik a feladat, akkor az helyettesíthető bármilyen, rendelkezésre állóval (Suzanne, űrhajó stb.).

Bemutatáskor csak kész, **működő**, a feladatot megoldó programot fogadunk el. A feladatokkal kapcsolatban a gyakorlatokon lehet további kérdések feltenni a gyakorlatvezetőknek ha valami nem egyértelmű. Értékelésről további részletek innen érhetőek el.

Általános követelmény, hogy program több objektumot töltsön be, és jelenítsen meg. Az objektumok legalább minimálisan animáljanak, legyenek textúrázva és bevilágítva. A színtérben **legalább két fényforrásnak** kell lennie, ezek típusa tetszőleges lehet (pontszerű, irány vagy akár spot fényforrás). A két fényforrásra legalább a **diffúz** és a **spekuláris** modellt meg kell valósítani! A nézetet lehessen egérrel vagy billentyűzettel tetszőlegesen változtatni (**forgatni** és **mozgatni**). Egyedüli kivétel, ha a feladat szövege kimondja, hogy ezek közül valamelyik nem kell.

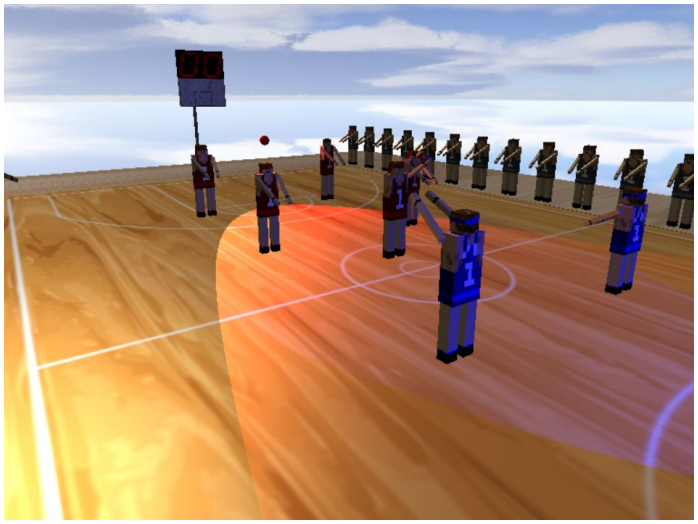
Pontozás:

0-44	45-49	50-69	70-84	85-100+
1	**2**	**3**	**4**	**5**

Zakkant kosarasok (130 pont)

Készíts egy téglalap alakú, 28x15 méretű **sík pályát**. A pálya két rövidebb oldalának közepétől mérve a pálya közepe felé 1.2 egységre helyezz le egy palánkot (téglalap/vékony téglatest), a pálya fölött 3 egység magasan. Erre merőlegesen illeszd a gyűrűt (tórusz) 3.05 magasságra a pályához képest (vízszintesen pedig a palánk közepén helyezkedjen el).

https://c2.staticflickr.com/8/7237/7177981837_f516db0cfa_b.jpg



A pályán helyezz el 5 db piros és 5 db kék játékost, 3 szürke mezes bírót (1-et a pályára, 2-t pedig a pálya szélén), illetve egy gömb alakú labdát (narancssárga minden fénymodellben).

A játékosokat ellipszoidok, téglatestek, hengerek segítségével állítsd össze, (lehetőleg) legyen: fejük, törzsük, karjuk, lábuk. A labda kezdetben valamelyik játékos kezében legyen. A játékosok legyenek változatos magasságúak és szélességűek, de ne teljesen szürreálisak (magasság: ~1.8-2.3, szélesség ~0.4-0.7).

Mindig legyen mindkét csapatból egy játékos 'aktív' állapotban, legyen billentyűk segítségével állítható a kijelölés. Az aktív játékos feje fölött jelenjen meg egy kék/piros gömb, így jelezve az elképesztő aktívságot.

Minden felhasznált modell legyen **textúrázva**, és hasson rá a megvilágítás, ami egy napszerű mozgást végző **irányfényforrás**. A terep tetszőlegesen bejárható legyen a **kamera** segítségével. **(5p, kötelező!)**

Az aktív játékosokat a WASD gombok (egyik) illetve a nyilak segítségével (másik) lehessen a pályán mozgatni, kivéve ha a kijelölt játékos a labdás játékos (Ő nem mozoghat.). A pályát ne tudják elhagyni a játékosok. **(5%)**

A játékosok karjai minden esetben előre nézzenek, legyenek egymással párhuzamosak és 3 különböző szögben állíthatóak: 60/90/120 fokot zárjanak be a törzsével (lefele tartott, előre tartott, felemelt állapot). (Mindig az aktív játékos karjainak állapotát lehessen gombnyomásra

változtatni.) A labdát birtokló játékos esetén a labda a karjával együtt mozogjon. **(5p)** **Ezt megvalósítani is kötelező!**

Helyezz el mindkét palánk tetejéről, a pálya közepe felé mutató spot fényforrást, a spot-ok színe legyen a csapat színének megfelelő. **(5p)**

A space (szóköz) billentyű lenyomására az éppen labdát birtokló játékos dobja a labdát a megfelelő gyűrűbe. A dobás egy ferde hajításként legyen szimulálva és a dobó játékos keze kerüljön a 'felemelt' pozícióba. Ezek után az ellenfél csapat egy véletlenszerű játékosa kezébe kerüljön a labda. (Ne legyen teljesen szürreális a labda pályája a dobás közben) **(10p)**

- Helyezz a palánkok fölé egy 2 digitális számjegyet megjelenítő eredményjelzőt. A palánk fölött az arra a kosárra dobott pontok szerepeljenek. **(5p)**
- A játékosok kosárszerzési esélye a gyűrűtől való távolságukkal négyzetesen csökkenjen. Amennyiben valaki dobást ront, úgy a labda jól látható módon a gyűrű mellé essen, majd a gyűrű vonalát elérve egy véletlenszerűen kiválasztott ellenfél játékos kezébe kerüljön. **(5p)**

Az ENTER billentyű nyomására a labdát birtokló játékos passzolja a labdát az aktív csapattársának. **(2p)**

- Egérrel ha a labdát birtokló csapat egy játékosára kattintasz akkor passzolja neki a labdát. Ezt több féle képpen is meg lehet oldani (A megvalósítások közt kizáró vagy művelet van érvényben):
    - A kattintást a pálya síkján vizsgáljuk, tehát nem a játékosokra lehet kattintani, hanem a pálya egy pontjára **(10p)**
    - A játékost közelítjük egy befoglaló testtel, pl AABB-vel vagy elipszoiddal és erre vizsgálunk metszést **(20p)**
    - Precíz ütközésvizsgálat (a modellek összes háromszögére) **(25p)**
    - A fentiekhez mintakódokért lásd: Haladó grafika
- A labda pályája függjön a passzoló játékos kezének állásától. 'Felemelt' esetben ívelés (ferde hajítás), 'előre tartott' esetben egyenes vonalú egyenletes mozgással, 'lefele tartott' esetben pedig pattintva kerüljön a másik kezébe a labda. **(5+5+5p)**
- Amennyiben a labda egy ellenfél játékost érint, akkor labdaszerzés következik be és a labda az érintett játékoshoz kerül. (Itt a játékosokat hengerrel/téglatesttel közelítsd, a labdát pedig egy gömbbel, tehát önmagával.) **(10p)**

A pálya köré helyezz szurkolókat (legalább 40 db ember figura, épp mint a játékosok) akik szintén változatos méretűek és a kezük mindig 60/90 fokot zár be a testükkel, illetve pontszerzés (kosár) esetén mindenki egy másodpercre felteszi a kezét (120° a törzsükkel bezárt szög). Minden játékos a pálya irányába nézzen. (Ez jelentheti a pálya középpontját is, illetve hogy simán a jó 'égtáj' felé) Mivel nagyon lelkes szurkolókról van szó így nem számít melyik csapat szerez pontot, minden esetben felteszik a kezüket. Hogy ne csak a levegőben álljanak a pálya köré még kb. 5 egységnyi (a pályától különböző színű) talajt helyezz le. **(8p)**

Ennek hatására az **FPS** a laboros gépeken még **100 fölött** maradjon. (hint: http://cg.elte.hu/~msc_cg/Gyak/RegiFelevek/2016/10/01_Instancing.zip) **(7p)**

További haladó feladatok (mintakódokért lásd: Haladó grafika):

- A pályát felhős ég, skybox vegye körül. **(8p)**
- Az emberek hagyjanak árnyékot a talajon, shadowmap segítségével. **(15p)**

3D Pipes (130 pont)

Motiváció

Remélhetőleg mindenki emlékszik a régebbi Windows operációs rendszerek legkedveltebb képernyőkímélőjére, a 3D Pipes-ra. Dinamikusan generálódó különböző színű és orientációjú 3D csöveket előállító rendszer, melyeknek egymásba tekeredésük és fonódásuk hosszas perceken keresztül teszik lehetővé a semmittevésünket a képernyő előtt.

A leírt (és motivációs videóban látható) rendszer egy OpenGL-es változatát kell elkészíteni ebben a Számítógépes Grafika nagybeadandóban!

Kötelezően elvégzendő feladatok (60 pont):

Kezdjük a csőgenerálást! (Összesen 20 pont)

Szükségünk van csőelemekre, egy egység átmérőjű gömbre és egy valamivel kisebb átmérőjű hengerre. A henger egyik végének lefedéséhez szükségünk van egy megfelelő méretű körlapra (másik oldalát nem szükséges lefedni ugyanis az mindig takarva lesz az épülő csővezeték által). **(5 pont)**

A csővezeték rendszer egy tetszőleges méretű (pl.: 64x64x64) kocka alakú térrészben generálódnak. Kiválasztunk egy véletlenszerű kezdeti (root) koordinátát, ahonnan kezdődni fog az új csővezeték ág generálása. A generálás során nyomon kell követnünk, hogy éppen melyik koordinátán tart a generálás, milyen csőelem van ott / következik és milyen irányba haladunk. A root helyén egy gömb csőelem jelenik

meg. A következő lépést mindig az határozza meg, hogy jelenleg milyen csőelemnél tart az adott csővezeték ág.

Gömb csőelem esetén: Generáláskor ellenőrizzük a környező irányokat (jobbra, balra, fel, le, előre, hátra, (+X, -X, +Y, -Y, +Z, -Z)), milyen irányban szabad a szomszédos mező. A szabad irányok közül kiválasztunk egyet, erre fog haladni a csővezeték építése. A megadott irányban generálunk egy megfelelő orientációjú henger csőelemet. Amennyiben nincs szabad irány, a generálás elakadt ezen ágon. Ekkor egy új véletlenszerű helyre kell a root-ot áthelyezni és onnan folytatni a generálást. **(5 pont)**

Henger csőelem esetén: Generáláskor ellenőrzi, hogy az adott haladási irányba (melyet még az előző gömb csőelem generált) tud-e haladni, szabad-e a következő mező. Ha szabad a haladási irány, akkor az iránynak megfelelő szomszédos mezőben megjelenik egy henger csőelem (az iránynak megfelelő orientációval). A henger körlappal fedett része a haladási irány felé néz (így nem lehet belelátni a hengerbe, mert a fedetlen része az előző csőelemmel érintkezik és takarva van). Ha nem szabad a haladási irány akkor egy gömb csőelem generálódik. Valamekkora eséllyel a csővezeték megáll az adott ponton és egy gömb elemet generál. Ekkor a következő lépésben egy másik irányba folytatja a generálást a gömbelemből. **(10 pont)**

Additív rajzolás! (Összesen 40 pont)

Hosszabb csővezetékek esetén teljesítményigényes lesz minden képkocka előállításakor a teljes csővezetéket újra megjeleníteni. Ezért egy optimálisabb additív renderelést fogunk alkalmazni, amikor a kamera (és a később implementált fényforrás) változatlan. Additív renderelés esetén mindig csak az újonnan generált csőelemeket kell hozzárajzolni az előző képkockához és ez rendkívül gyorsan fog futni. Ha a kamera (vagy fényforrás) változott, akkor hagyományos módon a teljes képet újra kell rajzolni. Ehhez egy újfajta OpenGL erőforrást kell felhasználnunk, a **FrameBufferObject**-et!

A FrameBufferObject (továbbiakban **FBO**) hasonló egy textúrához, melybe renderelhetünk! A kiadott rajzolási utasítások eredménye az FBO textúrába fog íródni (gyakorlatban a hagyományos renderelésnél is ez történik, mert a rendszer biztosít nekünk egy alapértelmezett FBO-t az ablakunkhoz). Létre fogunk hozni egy ablak szélességű-magasságú FBO-t, majd a Render függvényben a csővezeték kirajzolását erre az FBO-ra fogjuk elvégezni. Ez még nem fog megjelenni a képernyőn (mert egyelőre csak egy memóriában tárolt textúrában van). Ezután visszaállunk az alapértelmezett FBO-ra (így már közvetlenül az ablakra rajzolunk). Létrehozunk egy teljes ablakot lefedő síklapot, melyre textúraként ráfeszítjük az FBO-nk tartalmát, tehát az előbb kirajzolt csővezetéket. Ez után a kirajzolás után megjelenik az ablakban a csővezeték.

A honlapon található FBO projekt bemutatja az új erőforrás felhasználását:

A fejállományban deklarálunk pár FBO-hoz szükséges változót:

```
bool m_frameBufferCreated{ false };        //Korábban hoztunk már létre FBO-t?
GLuint m_depthBuffer;              //FBO Mélységbuffer textúra azonosító
GLuint m_colorBuffer;              //FBO Színbuffer textúra azonosító
GLuint m_frameBuffer;              //FBO azonosító
```

A CreateFrameBuffer metódussal a megfelelő felbontású FBO-t létrehozzuk. Ennek meghívására szükség van az Init függvényben (640, 480 paraméterekkel, mert ez az alapértelmezett ablakméret) és amikor átméretezzük az ablakot (tehát a Resize függvényben a _w és _h paraméterekkel).

A CreateFrameBuffer metódus először ellenőrzi, hogy az FBO-t létrehoztuk-e korábban (ez esetben először törölnie kell az előző erőforrásokat). Létrehoz egy FBO-t, majd létrehozza a szükséges színbuffer textúrát és mélységbuffer textúrát, melyekbe az FBO-ba renderelés eredménye lesz tárolva. Ezeket GL_COLOR_ATTACHMENT0 és GL_DEPTH_ATTACHMENT mellékletként csatolja az FBO-hoz. Végül a glCheckFramebufferStatus segítségével ellenőrzi, hogy a konfigurálás sikeres volt-e.

Miután az FBO objektumunkat helyesen inicializáltuk, megkezdhetjük a Render módosítását: Mikor az új FBO-ba szeretnénk rajzolni, ki kell adni a glBindFramebuffer(GL_FRAMEBUFFER, m_frameBuffer) utasítást (ezzel tudatjuk az OpenGL-t, hogy hová szeretnénk írni a rajzolási utasításaink eredményét). Ez után ugyanazokat a renderelési parancsokat alkalmazzuk, mint korábban. Miután mindent belerajzoltunk az FBO-ba, vissza kell állni az alapértelmezett FBO-ra glBindFramebuffer(GL_FRAMEBUFFER, 0) (ez az amely az ablakunkban megjelenik) és ide kell elvégezni a rajzolást.

A minta projektben ehhez egy újabb shader-t használtunk fel (postprocess.vert, postprocess.frag). Ezt kell használatba venni és a saját FBO-nk által előállított textúrát átadni. A síklap melyre ráfeszítjük a textúrát a vertex shader-ben van definiálva így csak egy 4 csúcspontos rajzolási utasítást kell kiadnunk.

```
m_programPostprocess.Use();
m_programPostprocess.SetTexture("frameTex", 0, m_colorBuffer);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

És készen is vagyunk! Saját FBO segítségével rajzoltunk az ablakra. **(25 pont)**

Ez így még mindig ugyanolyan lassú, mint FBO előtt. Készítsük el az additív renderelést az alábbiak szerint:

Ha az előző képkockához képest a kamera (vagy későbbi fényforrás) elmozdult/módosult:

A teljes színtér, teljes csővezeték újra renderelésére van szükség. Először az FBO-ra renderelünk, ahol először letöröljük a jelenlegi tartalmat glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT),

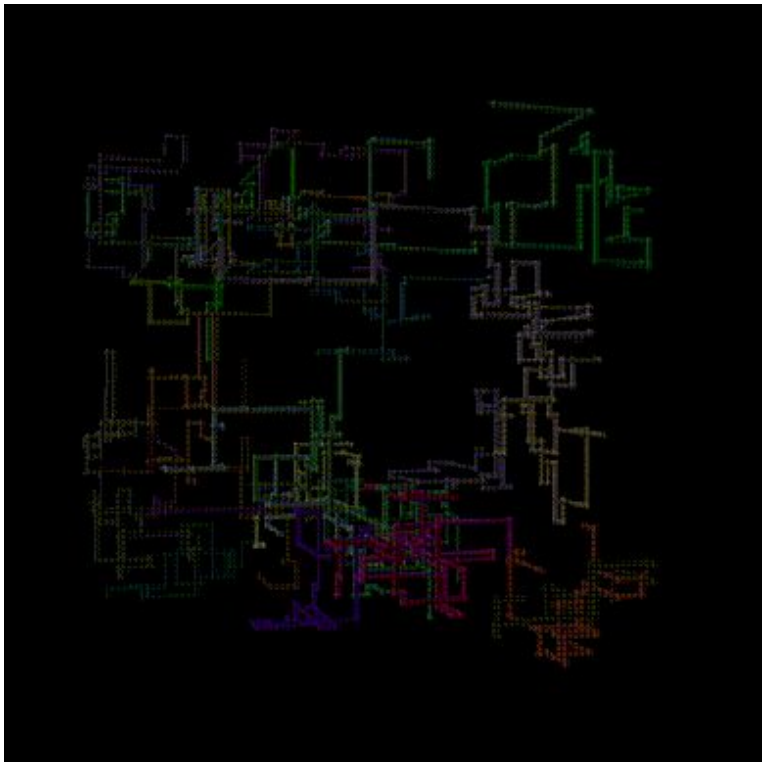
majd a teljes csővezetékrendszert újra rendereljük. Ezután visszaállunk az alapértelmezett FBO-ra, letöröljük tartalmát és a postprocess shaderrel újra rendereljük a képet. **(5 pont)**

Ha az előző képkockához képest a kamera (és későbbi fényforrás) mozdulatlan/ugyanolyan:

Csak az előző képkocka kirajzolása után megjelent új csőelemeket kell kirajzolni. Először az FBO-ra renderelünk, viszont nem töröljük le az előző képkockát (mert additíven rajzoljuk hozzá). Kirajzoljuk csak az új csőelemeket. Ezután visszaállunk az alapértelmezett FBO-ra, letöröljük tartalmát és a postprocess shaderrel újra rendereljük a képet. **(10 pont)**

A fenti feladatok elvégzése kötelező, amennyiben nem kerültek megvalósításra a beadandó nem értékelhető!

Opcionális feladatok (70 pont):



Kanyarodó csőelem is megjelenhet a csővezetékben! A kanyarodó csőelem egy negyed tórusz melynek egyik vége le van zárva egy körlappal (hasonlóan mint a henger esetében). **(3 pont)** Kanyarodó csőelem akkor keletkezhet, mikor egy henger csőelem szabadon haladhatna, mégis megáll és egy másik irányba folytatja útját. Ekkor véletlenszerűen egy gömb csőelem generálódik (melyből bármerre folytathatja az útját), vagy egy véletlenszerű orientációjú kanyarodó csőelem (ekkor csak a kanyarodás irányába folytathatja az útját. Kanyarodó csőelem generálása előtt ellenőrizzük, hogy az adott irányba tényleg haladhat-e a csővezeték! **(7 pont)**

A csővezeték rendszer generálása egyszerre több ágon zajlik. A generálás kezdetekor nem egy hanem N (pl. 5) kezdeti (root) csőelemet helyezünk el és mindegyik a fentebb leírt módon generálja a saját csővezeték ágát. Ügyeljünk rá, hogy a külön generálódó ágak se ütközzenek össze, kerüljék el egymást ütközés előtt. Ha az adott ág beszorul egy zsákutcába akkor fentebb leírt módon új helyről folytatja a generálást. **(5 pont)**

Minden csővezeték ágnak más színe van. Az új ág kezdetekor egy random színt választunk. A csővezeték ág minden eleme ezzel a színnel jelenik meg. Zsákutcába jutáskor új színt generálunk az új ágnak. **(5 pont)**

A csővezeték rendszert egy irány fényforrás világítja meg. A geometriákhoz generáljunk helyes normálvektorokat és programozzuk le az árnyalási algoritmust. **(5 pont)** A fény iránya futás közben módosítható. Módosítás esetén a teljes csővezeték újra rajzolódik, egyébként továbbra is additív rendereléssel jelenítjük meg az új elemeket. **(5 pont)**

A csővezeték generálása UI segítségével befolyásolható:

- „Reset" Gombra kattintva a teljes generált csővezetékrendszer törlődik és egy új generálásába kezdünk. **(2 pont)**
    - A csővezetékrendszert befoglaló doboz mérete állítható. Az újonnan beállított méret az előbbi „Reset" gombra kattintva lép érvénybe. **(5 pont)**
- Állítható a generáló ágak száma. Ha egy jelenleginél nagyobb számot állítunk be új ágakon is elkezdődik a generálás, ha kisebbet akkor egyes szálak „elakadnak" és abba hagyják a generálást. **(5 pont)**
- A generálás sebessége állítható. **(1 pont)**
- A fény iránya UI segítségével állítható. **(1 pont)**
- A generált ágak egy listában jelennek meg (pl. azonosítókkal: branch 1, branch 2, branch 3, stb⋯). **(2 pont)**
    - A listában kijelölt ág fehér színnel, árnyalás nélkül jelenik meg (ez teljes újra renderelést igényel). Másik ág kijelölésekor visszaváltozik az eredeti (generált) színére. **(10 pont)**
    - A kijelölt ág színe módosítható. A korábban legenerált elemek és az újonnan létrehozottak színe is módosul. Ez teljes újra renderelést igényel. **(5 pont)**
    - A kijelölt ág generálása szüneteltethető. Ekkor az adott ág nem generál újabb elemeket míg újra el nem indítjuk a generálást. **(2 pont)**
    - A kijelölt ág megállítható. Ekkor az ág „elakad" és új helyről kezdi egy új ág generálását. (**2 pont)**
    - A kijelölt ág törölhető. Ekkor törlődik az összes csővezeték elem mely az adott ághoz tartozott. Az újonnan generálódó csővezeték elemek elfoglalhatják a törölt elemek pozícióit. Ez teljes újra renderelést igényel. Ha egy jelenleg generálódó ágat törlünk akkor az adott ág „elakad", törlődik, majd új helyről kezdi a generálást. **(5 pont)**

Minecraft (247 pont)

Rövid feladatleírás

A feladatban a Minecraft egy egyszerűsített változatát kell létrehozni. A játéktér 1*1*1 m-es kocka alakú blokkokból épül fel. A blokkok a világ koordinátatengelyeivel párhuzamos szabályos kockarácson helyezkednek el. A különböző típusú blokkok ebben a feladatban mindig kitöltik a teljes kockát (nem feladat pl. ládák és egyéb, speciális modellel rendelkező blokkok leprogramozása). A textúrákat textúraatlaszból vesszük. A játék fontos része, hogy a környezet procedurálisan generált, a feladatban ennek is megoldandó egy egyszerűsített változata. A generált világ ezután szabadon módosítható: a blokkok törhetőek, lerakhatók új blokkok.

Kötelezően megoldandó rész (62 pont)

Megjelenítés (20 pont)

A gyors megjelenítés érdekében nem egyesével rajzoljuk ki a kockákat. Egyszerre sok blokkot rajzolunk, és csak azokat a blokk-oldalakat rajzoljuk ki, amik megjelenhetnek, azaz levegővel érintkeznek az adott oldalon. Ezeket rakjuk a VBO-ba.
Mivel módosításkor újra kell számolni a kirajzolandó oldalakat, érdemes a blokkokat nagyobb egységekbe fogni, és minden egységhez egy VBO-t rendelni. Ezek lesznek a chunk-ok. A Minecraft-ban egy chunk mérete 16*256*16 blokk, az órákon is megszokott módon az y-tengely a függőleges tengely. (A chunk-méretet érdemes megtartani, ekkora mérettel még jól lehet valós időben számolni a változásokat.) Az alapfeladatban jelenjen meg 10*10 chunk. (A kirajzolás további optimalizációjáról szól az első választható felada.)

Kamera (2 pont)

Az OGLBase-es kamerát alakítsuk át úgy, hogy ne kelljen nyomva tartani a bal egérgombot a kamera forgatásához. Hogy a kurzor ne tudja elhagyni az ablakot, használjuk az SDL erre kitalált függvényét: *SDL_SetRelativeMouseMode*, ezzel eltüntethetjük a kurzort. Ezt lehessen ki-be kapcsolni (pl. valamelyik billentyű megnyomásával), hogy a Gui-t is lehessen használni egérrel (ilyenkor a kamera ne mozogjon).

Textúrák (20 pont)


A textúrázást egy textúraatlasszal kell megoldani (link, vagy használható más, akár saját textúra is). A Minecraftban ma már nem ez van, de itt ez a feladat. Az atlasz 16*16 db azonos méretű résztextúrából áll. A blokkokhoz szerepel az összes szükséges textúra, így csak ezt az egy fájlt kell betölteni, és a megfelelő részéről olvasni. A textúrát *GL_NEAREST* nagyító szűréssel mintavételezzük (így kapjuk meg a Minecraftra jellemző pixeles textúrákat). Ha a textúra mérete kettőhatvány, akkor a MipMap is jól fog működni (különben összemosódnának az egymás mellett lévő textúrák a kisebb szinteken), ez most teljesül, így használható kicsinyítő szűrésre, pl. *GL_NEAREST_MIPMAP_LINEAR* beállítással. Legyen legalább 5 különböző textúrájú blokk, és ezek között legyen olyan, aminek nem minden oldalán azonos a textúra (pl. fűblokk: az alja föld textúra (1. sor 3.), az oldalai föld + fű a tetején (1. sor 4.) és a teteje fű (1. sor 1.).

Tájgenerálás (20 pont)

Egyszerű procedurális világgenerálás megvalósítása. Szükséges valamilyen kétdimenziós magasságtérkép. Ez vehető tetszőleges véletlen algoritmusból: pl. Perlin-zaj vagy Diamond-square algoritmus, de más is megfelel. A világ egy adott (x,z) koordinátájú pontján a magasságtérkép határozza meg, hogy a generálódó legmagasabb blokk milyen magasan (y) legyen. A különböző magasságokban legyenek különböző blokkok, pl. alul kő, feljebb föld és a legtetején egy fűblokk. (A váltások legyenek a maximális magassághoz relatívan meghatározva.)

Az eddig felsorolt feladatok kötelezőek, ezek nélkül nem értékelhető a beadandó!

Választható feladatok (165 pont)

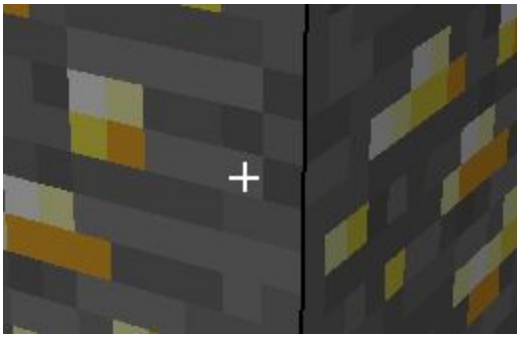
Rajzolás optimalizálása (20 pont)

Spóroljunk a video-memóriával! A VAO-ba próbáljunk meg minél kevesebb adatot felvenni, és ezt minél kompaktabb módon átadni. A kirajzolási pozíció vertex-attribútum egész típus is lehet, hiszen a blokkok csak egész koordinátákon lehetnek. Ugyanígy a textúrapozíció is kiszámolható a shaderben egy textúraindexből. Ezt kihasználva kisebb helyen is átadhatók a rajzoláshoz szükséges információk. Ha 4*32 biten sikerül eltárolni akkor **10 pont** (pl. egyetlen ivec4/uvec4

attribútum), ha 32 biten sikerül eltárolni, akkor **20 pont** (pl. egyetlen int/uint attribútum). Egész típusú attribútumokhoz segítség a feladat végén.

Megjelenítés (22 pont)

Célkereszt (7 pont)

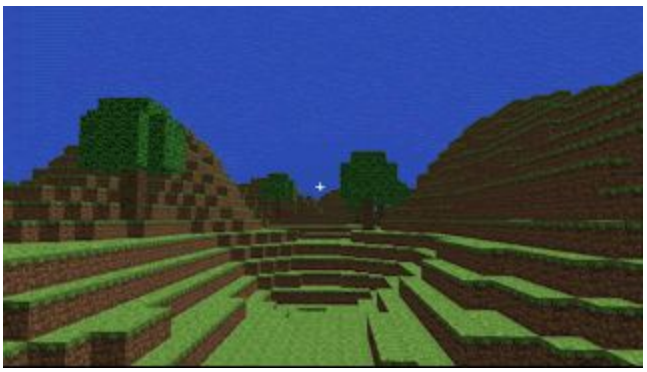


A képernyő közepén legyen egy célkereszt. A skyboxnál is használt trükkel a közeli vágósíkra hozható a geometria, így mindig látszani fog. Megvalósítás tetszőleges. Néhány példa/ötlet: két egyszerű vonal-primitív; egy pont-primitív nagy pontmérettel és a kereszten kívüli részek procedurálisan eldobva (discard); egy pont-primitív nagy pontmérettel, textúrával.

Block highlight (5 pont)

A nézett blokk legyen kiemelve egy “keret”-tel. Ehhez természetesen szükség van a blokkkiütéshez is használt sugárkövetésre (lásd egy kicsit később).

Skybox (10 pont)

 A háttérben legyen procedurális skybox: vízszintes alatt sötétkék, vízszintes fölött világoskék, a kettő között egy rövid színátmenet (**2 pont**). Hold és nap: a skyboxhoz hasonlóan rajzolva, de rendes texúraként betöltve – a skyboxnál látott trükkel a távoli vágósíkon megjelenítve. A hold és a nap egymással mindig pontosan átellenben keringjenek a világ körül (mint a Minecraftban), az x- vagy a z-tengely körül (**8 pont**).

Interakciók (48 pont)

Blokktörés (20 pont)

Lehessen a blokkokat kiütni a bal egérgombbal. Ehhez a kamerából kell indítani egy sugarat, és meghatározni, hogy melyik az első blokk, amit eltalál. Ezt a blokkot kell törölni, azaz levegővel helyettesíteni. A sugárkövetéshez segítség a feladat végén található.

Blokk lehelyezése (10 pont)

Blokk kiütése [bal egérgomb] mellett lehessen új blokkokat lehelyezni az éppen nézett blokk mellé (a nézett lapra) a jobb egérgomb megnyomásával.

Pick block (3 pont)

A középső egérgomb (görgő) megnyomására a lehelyezendő blokk típusa legyen olyan, mint a nézett blokk (**1 pont**). A görgővel (is) lehessen a támogatott blokkok között lépkedni (**2 pont**).

ImGUI (15 pont)

A gui-n jelenjen meg, hogy milyen típusú blokk van éppen kiválasztva lehelyezésre (**1 pont**). A lehelyezendő blokktípus legyen kiválasztható, pl. lenyíló lista (**4 pont**). A nap/hold keringési ideje legyen változtatható (**2 pont**). Lehessen betölteni új textúraatlaszt képfájlból (**8 pont**).

Mentés és betöltés (15 pont)

Lehessen fájlba menteni a világban lévő blokkokat, valamint az így elmentett fájlokat megnyitni a programmal, és betölteni az elmentett világot.

Dinamikus chunk-betöltés/generálás (20 pont)

Ahogy a kamera mozog a messze lévő chunkok ne rajzolódjanak és ne legyenek memóriában sem, valamint még nem látott helyre érve generálódjon “végtelen módon” a táj (**10 pont**). Perzisztencia: ha olyan területre érünk, ami már volt betöltve, az ne generálódjon újra, azaz pl. ha volt rajta változtatás (blokk lehelyezés, blokktörés), akkor azok jelenjenek meg. (**10 pont**).

Játék, “fizika” (25 pont)

Legyen egy karakterünk, amivel mozoghatunk, mint egy fps-játékban (~Minecraft creative mode repülés nélkül), azaz legyen ütközésvizsgálat, ne tudjunk átmenni a blokkokon. Hasson a karakterre gravitáció is. A játékos hitbox-a egy egyszerű téglatest: 1.8 blokk magas, 0.6 blokk széles és hosszú, oldalai mindig a blokkokkal párhuzamosak, ezzel végezzük az ütközésvizsgálatot. A játékos gyalogsebessége 4.317 blokk/másodperc, futási sebessége 5.612 blokk/másodperc. Ugrás: kb. 1.25 blokk magasra tud a karakter ugrani, tehát egy blokkra fel tud ugrani, de kettőre már nem. Lehessen váltani e között és a szabad repülés között (~Minecraft spectator mode, az alapfeladatban leírtak szerint).

Víz és láva (15 pont)

Legyen víz- és láva-blokk is a játékban. Legyenek animálva (a mellékelt textúraatlasz jobb alsó sarkában van 5-5 frame) (**3 pont**). Legyen a víz megfelelően átlátszó, azaz látsszanak alatta/mögötte a szilárd blokkok. Legalább arra az esetre működjön helyesen, ha a generált világban egy adott szint alatt minden levegőblokk le van cserélve vízre generáláskor. A részben átlátszó textúrákat mindenképpen utolsóként kell kirajzolni, ezért ezt le kell választani a többi blokk rajzolásától – de a shader attól lehet azonos (**12 pont**).

Segítség az implementációhoz

Egész típusú attribútumok

Az egész típusú attribútumokhoz *glVertexAttribPointer* helyett a *glVertexAttribIPointer* függvényt kell használni. Ehhez az OGLBase-ben néhány dolgot meg kell változtatni (az OGLBase előtti projektekben elég az előbb említett függvényt használni). Az első, amit ki kell egészíteni, az a *VertexArryObject.h*-ban az *AttributeData::Apply* függvény, ez legyen a következő:

```
void Apply()
{
	glEnableVertexAttribArray(index);
	if (type == GL_INT || type == GL_UNSIGNED_INT) {
		glVertexAttribIPointer(index, size, type, stride, ptr);
	}
	else {
		glVertexAttribPointer(index, size, type, normalized, stride, ptr);
	}
}
```

Emellett a *GLconversions.hpp*-ben a 70. sor körül egészítsük ki a definíciókat a következőkkel:

```
template <> constexpr GLenum NativeTypeToOpenGLType<int>() { return GL_INT; }
template <> constexpr GLenum NativeTypeToOpenGLType<unsigned int>() { return GL_UNSIGNED_INT; }
```

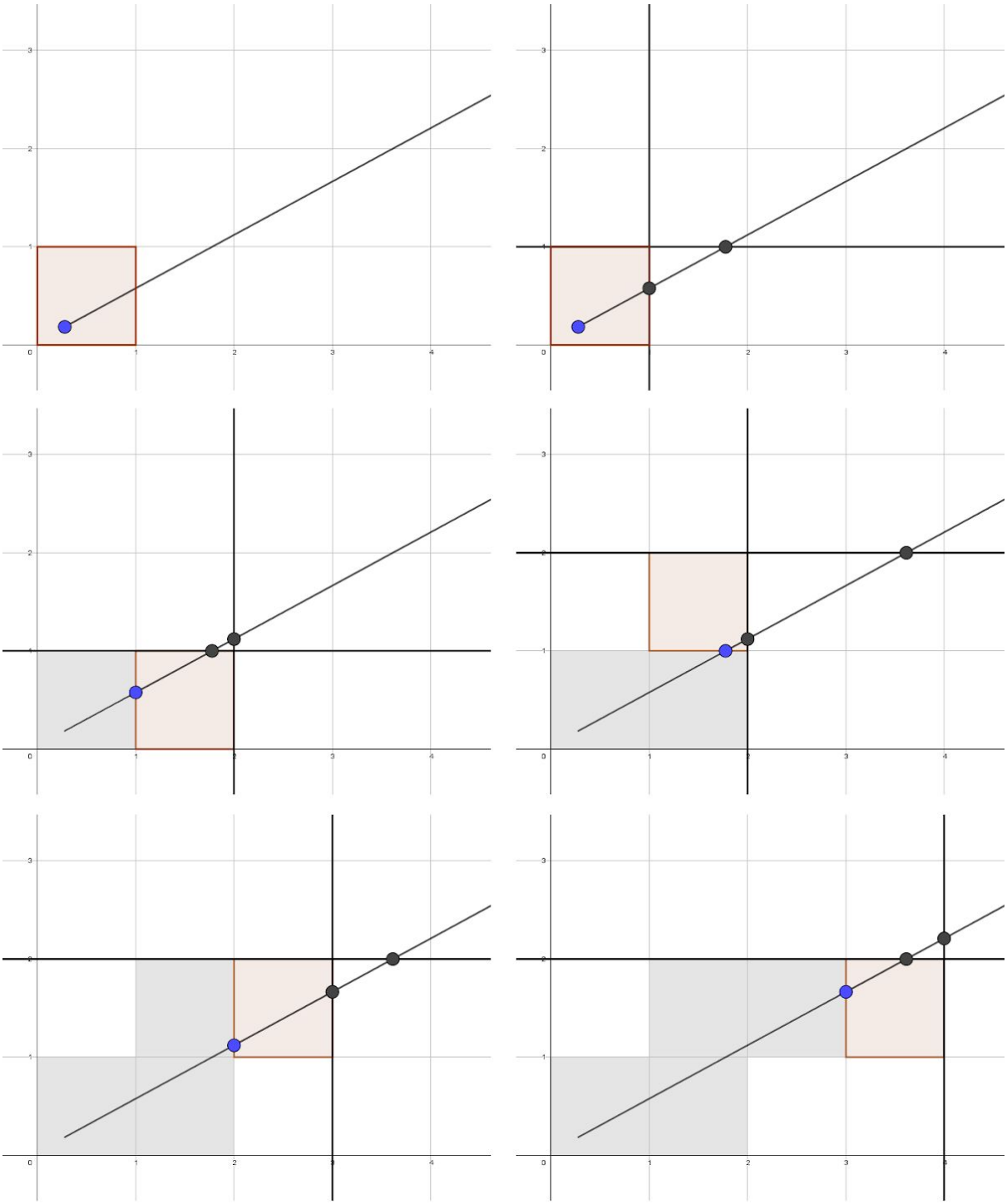
Végül ugyanebben a fájlban a 80. sor körül illesszük be ezeket:

```
COMPONENT_COUNT_SPECIALIZATION(glm::ivec2, 2, 1)
COMPONENT_COUNT_SPECIALIZATION(glm::ivec3, 3, 1)
COMPONENT_COUNT_SPECIALIZATION(glm::ivec4, 4, 1)
```

(Az első kettő együtt lehetővé teszi, hogy a *CreateAttribute*-ot használjuk *int* és *unsigned int* paraméterekkel. A harmadik pedig a *ProgramObject::SetUniform*-ot teszi használhatóvá a *glm::ivec* típusokkal és belőlük képzett tömbökkel, vektorokkal.)

Sugárkövetés kockarácson

Egy speciális sugárkövetést kell implementálni, ami a kockákból álló térben keresi az első nem-levegő blokkot, amit metsz a kamerából induló sugár. (Érdemes lehet egy távolsági korlátot is beépíteni “reach”.) Segítség az implementációhoz: a sugáron való haladáskor a különböző oldallapokkal való metszéspontokon kell végigmenni. Ezek gyorsan számolhatók, hiszen az oldallapok párhuzamosak a tengelysíkokkal. Elég tehát mindig a három különböző síkkal megtalálni a metszéspontot, és kiválasztani a legközelebbit, így biztosan nem hagyunk ki egy olyan blokkot sem, amin átmegy a sugár. A képeken lekövethető az algoritmus működése 2D-ben. 3D-ben annyi a változás, hogy a 2 egyenes helyett 3 síkkal kell metszeni.



Kék pont: aktuális hely, barna négyzet: aktuális blokk. A kivastagított, tengelyekkel párhuzamos egyenesek a haladási iránynak megfelelően, az aktuális blokk éleinek meghosszabbításai. A fekete pontok a két egyenes és a sugár metszéspontjai. A közelebbi metszéspont lesz a következő lépés (és ez határozza meg azt is, hogy melyik a következő blokk).

FPS játék (170 pont)

Bevezetés: ebben a feladatban egy kezdetleges FPS játékot valósítunk meg, ezzel bemutatva, hogy milyen technikákon alapulnak akár a mai FPS-ek is.
Először is szükségünk lesz a kamera megfelelő beállítására, hogy ténylegesen egy FP nézetes programot kapjunk, illetve egy fegyver meshének betöltésére és elhelyezésére. Ezt követően egy tetszőleges pálya elkészítésével folytatjuk. Amint ez megvan, értelmet is nyer az ütközésvizsgálat, azaz a Collision detection, melynek köszönhetően a karakterünk már nem tud átsétálni objektumokon. A következő feladat az ellenségek létrehozása, és nekik valamiféle alap logika megvalósítása. Következőként a lövés és a találatok regisztrálását valósítjuk meg. A megfelelő hangulat eléréséhez a fényekkel is tudunk játszani egy kicsit, végül pedig az ellenségek megsemmisítéséhez és a lövések leadásához vezetünk be hangokat.

Megjegyzés: feltűnhet, hogy a feladat során sok alakzatot kell majd számon tartanunk és használnunk, amelyek hasonló tulajdonságokkal rendelkeznek. Éppen ezért érdemes már a feladat legelején megtervezni egy osztályhierarchiát, amely nagyban hozzájárul az ehhez hasonló bonyolultabb projektek kezelhetőségéhez. Példa: minden objektum rendelkezik egy unique ID-val, az objektumok között AABB collision-detectiont kell végezni.

A feladat

----- Kötelező rész ------
(71 pont)

Alap nézet létrehozása (10 pont):
A legelső dolgunk a játékos karakter elkészítése. Mivel kívülről nem látjuk, ezért nem lesz szükség karakter meshre, viszont egy fegyver meshét be kell tölteni, és úgy elhellyezni, hogy mindig a nézet jobb alsó sarkában legyen. **(5 pont).**
Továbbá a játékosunk fizikájára is szükségünk lesz, azaz mindegy, hogy vertikálisan merre nézünk, a karakter vertikális pozíciója ne változzon (azaz maradjunk a földön, ne tudjunk repülni). **(5 pont)**

Pálya létrehozása: (5 pont)
Ha van játékosunk, akkor készítsünk is egy pályát. A pálya legalább 3 teremből álljon, mindegyik tartalmazzon belül is falakat. Az oldala legyen zárt, ne tudjuk elhagyni. A falak valamekkora pozitív vastagsággal rendelkezzenek, hogy a későbbiekben a collision detectiont implementálni tudjuk. **(5 pont)**
A pálya elkészíthető pl beégetett geometriákkal vagy esetleg fájlból beolvasva.

Ütközésvizsgálat: (20 pont)
A collision detectiont a karakterünk, és az ellenséges karakterek, és a pálya falai között kell majd végezni. (A 2D-s változathoz segítség ezen a linken található, nekünk a 3D-s

megvalósítására lesz szükségünk). A célunk az, hogy ne tudjunk olyan helyre eljutni, ahol egy másik objektummal ütköznénk.
Fontos: ezt mindenképpen építsük be az osztályhierarchiánkba, ha szép és egyszerű megoldást szeretnénk! Ehhez minden objektumhoz(falak, ellenségek, saját karakter) tartozzon egy AABB (egy doboz, amely befoglalja a karakterünket), és minden frameben le kell ellenőrizni, mielőtt egy objektum pozícióját változtatjuk, hogy az új pozíción ütközik-e az ő AABB-je bármelyik másik objektum AABB-jével. Ha igen, ne tudjon oda mozogni, egyébként igen. **(20 pont)**

Ellenségek: (3 pont)

Az ellenségeinket mesh-ből töltsük be. Lehetőleg olyan mesht válasszunk, amellyel nem néz ki nagyon rosszul az animáció hiánya. **(3 pont)**

Lövés: (25 pont)

A következő feladat a fegyverünk használhatóvá tétele, azaz a lövés lehetőségének megteremtése. A lövedékek sebessége végtelen nagy. A lövés pillanatában a lövedékünk eltalálja a célkeresztnél (a képernyő közepe) lévő objektumot.
Ennek a megvalósításához egy Frame buffert érdemes használni (példa projekt itt elérhető). Ez egy extra buffer, amibe a végső kép kirenderelése előtt egyéb információkat tárolhatunk el.

A mintakód részletesebb tárgyalása:

A FrameBufferObject (továbbiakban **FBO**) hasonló egy textúrához, melybe renderelhetünk! A kiadott rajzolási utasítások eredménye az FBO textúrába fog íródni (gyakorlatban a hagyományos renderelésnél is ez történik, mert a rendszer biztosít nekünk egy alapértelmezett FBO-t az ablakunkhoz). Létre fogunk hozni egy ablak szélességű-magasságú FBO-t, majd a Render függvényben a csővezeték kirajzolását erre az FBO-ra fogjuk elvégezni. Ez még nem fog megjelenni a képernyőn (mert egyelőre csak egy memóriában tárolt textúrában van). Ezután visszaállunk az alapértelmezett FBO-ra (így már közvetlenül az ablakra rajzolunk). Végül létrehozunk egy teljes ablakot lefedő síklapot, melyre textúraként ráfeszítjük az FBO-nk tartalmát.

A fejállományban deklarálunk pár FBO-hoz szükséges változót:

```
bool m_frameBufferCreated{ false };      //Korábban hoztunk már létre FBO-t?
GLuint m_depthBuffer;                    //FBO Mélységbuffer textúra azonosító
GLuint m_colorBuffer;                    //FBO Színbuffer textúra azonosító
GLuint m_frameBuffer;                    //FBO azonosító
```

A CreateFrameBuffer metódussal a megfelelő felbontású FBO-t létrehozzuk. Ennek meghívására szükség van az Init függvényben (640, 480 paraméterekkel, mert ez az alapértelmezett ablakméret) és amikor átméretezzük az ablakot (tehát a Resize függvényben a _w és _h paraméterekkel).

A CreateFrameBuffer metódus először ellenőrzi, hogy az FBO-t létrehoztuk-e korábban (ez esetben először törölnie kell az előző erőforrásokat). Létrehoz egy FBO-t, majd létrehozza a szükséges színbuffer textúrát és mélységbuffer textúrát, melyekbe az FBO-ba renderelés eredménye lesz tárolva. Ezeket GL_COLOR_ATTACHMENT0 és GL_DEPTH_ATTACHMENT

mellékletként csatolja az FBO-hoz. Végül a glCheckFramebufferStatus segítségével ellenőrzi, hogy a konfigurálás sikeres volt-e.

Miután az FBO objektumunkat helyesen inicializáltuk, megkezdhetjük a Render módosítását: Mikor az új FBO-ba szeretnénk rajzolni, ki kell adni a glBindFramebuffer(GL_FRAMEBUFFER, m_frameBuffer) utasítást (ezzel tudatjuk az OpenGL-t, hogy hová szeretnénk írni a rajzolási utasításaink eredményét). Ez után ugyanazokat a renderelési parancsokat alkalmazzuk, mint korábban. Miután mindent belerajzoltunk az FBO-ba, vissza kell állni az alapértelmezett FBO-ra glBindFramebuffer(GL_FRAMEBUFFER, 0) (ez az amely az ablakunkban megjelenik) és ide kell elvégezni a rajzolást.

A minta projektben ehhez egy újabb shader-t használtunk fel (postprocess.vert, postprocess.frag). Ezt kell használatba venni és a saját FBO-nk által előállított textúrát átadni. A síklap melyre ráfeszítjük a textúrát a vertex shader-ben van definiálva így csak egy 4 csúcspontos rajzolási utasítást kell kiadnunk.

```
m_programPostprocess.Use();
m_programPostprocess.SetTexture("frameTex", 0, m_colorBuffer);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

A megvalósításhoz segítség:

A lövés logikája az lesz, hogy minden világban lévő objektum rendelkezik egy egyedi azonosítóval (osztályhierarhia). Minden lövés esetén egy FBO-ba renderelünk először, de nem színeket, hanem csak az azonosítójukat írják be az objektumok a bufferbe. Tehát a Framebufferünk colorbufferje elég, ha inteket tartalmaz, így ennek a megadása így nézne ki:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_R32UI, width, height, 0, GL_RED_INTEGER,
GL_UNSIGNED_INT, nullptr);
```

Az inputkezelésnél ugyanúgy tudunk rajzoló parancsokat kiadni, mint a render függvényben, így a bal egérgomb lenyomása esetén el tudjuk végezni a rajzolást, ami feltölti az FBO-nkat az objektumaink indexével.
A használt shaderekben annyi lesz a dolgunk, hogy átadjuk az MVP mátrixot, és ezzel eltoljuk a vertexeinket, ez lesz a gl_Position értéke (mint minden szokásos renderelésnél). A fragment shaderben outputja pedig nem egy vec4 lesz, hanem csak egy int, aminek az értéke az éppen rajzolt objektum id-je lesz.
Ezután a buffer tartalmazni fogja, hogy egy-egy pixelen melyik azonosítójú objektum volt a legközelebb hozzánk, így már csak ki kell olvasnunk CPU oldalon a középső pixelben lévő azonosítót, őt találtuk el. Ehhez a glReadPixels függvényt érdemes használni, a framebuffer bind-olása után:

```
glBindFramebuffer(GL_FRAMEBUFFER, m_frameBuffer);
glReadPixels(x, y, 1, 1, GL_RED_INTEGER, GL_UNSIGNED_INT, (void*)m_data);
```

Ahol x, y, a középső pixel koordinátái, m_data pedig egy int * típusú változó, aminek például a konstruktorban érdemes helyet foglalni.

Ezzel az m_data változónk egy olyan intre fog mutatni, aminek értéke megegyezik az eltalált objektum ID-jével. **(25 pont)**

Logika: (8 pont)

Az ellenségeknek legyen életük. Amikor eltalálunk egy ellenséget, életet veszít. **(3 pont)**
Az ellenségek megsemmisülnek, amennyiben 0-ra csökken az életerejük. **(2 pont)**
Az ellenségek néhány meghatározott pontban véletlenszerű időpillanatokban éledjenek, de a játék kezdete óta eltelt idővel arányosan egyre többen/gyakrabban jelenjenek meg. **(3 pont)**

----- Opcionális rész ------
(99 pont)

Fények: (7 pont)

A világban az ambiens fény nagyon gyenge legyen. A fegyverünket egy elemlámpát szimuláló spotlight-tal lássuk el, amely mindig megvilágítjuk azt a területet, ahova éppen nézünk. **(7 pont)**

Sebződés: (7 pont)

A mi karakterünk is rendelkezzen életerővel. Amennyiben egy bizonyos távolságon belül kerülünk egy ellenséghez, veszítsünk valamennyi életet. (Ez ellenségenként pl. csak 1 másodpercenként egyszer történhet meg) **(5 pont)**

UI: (10 pont)

Célkereszt, az életerőnk és az eddig megsemmisített ellenségek számának folyamatos kijelzése. A megjelenítésük módja tetszőleges. (Nem feltétlenül szövegként kell ezt a megjelenítést elvégezni.) **(10 pont)**

Okosabb ellenségek: (7 pont)

Rendeljünk egy alap logikát is az ellenségekhez: amint egyenes vonalban látható a karakterünk az ő pozíciójukból, kezdjenek el mozogni felénk. **(7 pont)**

Játék vége: (5 pont)

A játék véget ér, amennyiben a mi életerőnk éri el a 0-t. Ekkor előről tudjuk kezdeni. **(5 pont)**

Fegyver viselkedése: (3 pont)

A lövéseink között el kell telnie valamennyi időnek, és ne tudjunk egyfolytában lövöldözni, ha rákönyökölünk a bal egérgombra. **(3 pont)**

Hang: (10 pont)

A programunkba vezessük be a hangokat. Ehhez az SDL audio modulját használjuk!
A lövéseknek és az ellenségek halálának/megjelenésének legyen hangja. (Mintakód elérhető itt) **(10 pont)**

Új fegyverek: (5 pont)

A pályán helyezzünk el néhány helyen más fegyvereket. Ha ezeken átsétálunk, akkor cserélődjön le a nálunk lévő fegyver meshe és a tulajdonságai(rate of fire (lövések között eltelt idő), sebzés). **(5 pont)**

Újratöltés: (5 pont)

Ne legyen végtelen lőszer a tárban, x lövés után újra kell tölteni, az R billentyűvel. Az újratöltés ideje alatt természetesen tudunk mozogni, de nem tudunk lőni. Ha nincs lőszer a tárban, akkor a lövés hatására is induljon el az újratöltés. Játszunk le hangot is az újratöltéshez! **(5 pont)**

Gyógyulás: (5 pont)

Amikor megölünk egy ellenséget, valamilyen eséllyel dobjon egy élettöltő objektumot (rendeljünk hozzá valamilyen mesht/primitívet). Amikor átsétálunk rajta, töltődjön az életerőnk. **(5 pont)**

Más fegyvertípusok: (10 pont)

Vezessünk be egy másfajta lövési módot, amikor a lövedékünk sebessége nem végtelen. Lövéskor készítsünk egy új objektumot (pl gömb), amely a nézeti irányunkba repül valamilyen sebességgel. Amikor collision detection segítségével észrevesszük, hogy ellenséggel ütközik, akkor sebződjön az ellenség. Ha fallal ütközik, akkor tűnjön el. **(10 pont)**

Lövésnyomok: (10/12 pont)

Amikor a lövedékünk eltalál egy falat, akkor a falon jelenjen meg egy becsapódási nyom. Ehhez például kirajzolhatunk egy kört valamilyen textúrával. **(5 pont)**

Az idő múlásával fokozatosan menjen át a becsapódási textúra a fal
eredeti textúrájába **(3 pont)**

VAGY

Az idő múlásával egyre átlátszóbb legyen az új objektum. **(5 pont)**

Amikor már teljes mértékben a fal textúra van a körön, vagy az teljesen átlátszó, semmisüljön meg a kör objektum. **(2 pont)**

Objektumok ID-jének FBO-ba renderelése minden frameben: (10 pont)

Ne csak bal egérgomb lenyomására nézzük meg, hogy melyik pixelen milyen ID-jű objektum van hozzánk legközelebb, hanem minden frameben. Ehhez azt csináljuk, hogy a kirajzolást ne a default bufferbe végezzük, hanem egy framebufferbe, de úgy, hogy a színeknél az alfa csatornába az ID-ket írjuk (itt egy okos int - float és float - int konverziót kell megírnunk a textúrába íráshoz és az onnan való olvasáshoz). **(10 pont)**

Célzott objektum kiemelése: (5 pont)

Az előbbi feladatot felhasználva megcsinálhatjuk azt, hogy mindig kiolvassuk az előző frameben a középpontban lévő objektum azonosítóját, és egy picit máshogy rajzoljuk ki, mint egyébként tennénk. (Fehérebb, pirosabb, jobban hat rá a diffúz fény, akármi más...) **(5 pont)**

3D Fraktál (150 pont)

Rövid feladatleírás

A feladat valamilyen fraktál sphere-tracing eljárás segítségével való megjelenítése. A megjelenítés raycast technika alkalmazásával történik, egy speciális implicitreprezentáción, a távolságfüggvényeken. Ehhez minden pixelre ki kell számolni egy sugár paramétereit. Ezen sugár és felület metszetét a sphere-tracing algoritmussal kapjuk meg. A felületi normálist numerikusan számíthatjuk ki, melynek segítségével már a felület könnyedén árnyalható.

Alapfeladat (70p)

Először legyen az implicitfüggvényünk az $f(\mathbf{p}) = \|\mathbf{p}\|_2 - 1$ ($\mathbf{p} \in \mathbb{R}^3$) képlettel definiált. Ezzen egységgömböt a raymarch ill. sphere tracing algoritmussal jelenítsük megy egy az (4,0,0) pontban elhelyezett (0,0,0)-ba néző kamerával (itt még nem kell perspektív kamera legyen, lehet egyszerű vetítés az YZ síkra) úgy, hogy a sphere tracing során kapott mélységérték ötödött állítjuk mindhárom színcsatornába.

Data: $\boldsymbol{p}, \boldsymbol{v} \in \mathbb{R}^3$ sugár
Result: $t \in \mathbb{R}^+$ sugáron megtett távolság
$t := 0; i := 0;$
$ft := f(\boldsymbol{p} + t\boldsymbol{v});$
while $ft < threshold$ **and** $i < max\_iterations$ **do**
  $ft := f(\boldsymbol{p} + t\boldsymbol{v});$
  $t := t + ft; \quad i := i + 1;$
end

Algoritmus 1: Sphere-trace sugárkövetési eljárás. Hart [1994]

A kamera **eye**, **at** és **up** vektorát uniform változóban adjuk át, és számoljunk ki egy kamera koordináta rendszert, valamint a pozícióhoz tartozó sugarat. A látószög és vetítési sík távolsága lehet konstans. A kamerát lehehessen WASD billenytűkkel és az egérrel mozgatni, forgatni. Ablak átméretezésnél ne torzuljanak a benne látott objektumok, és alap esetben is arányosak legyenek.

$$\boldsymbol{w} := \frac{\boldsymbol{eye} - \boldsymbol{at}}{\|\boldsymbol{eye} - \boldsymbol{at}\|_2}$$

$$\boldsymbol{u} := \frac{\boldsymbol{w} \times \boldsymbol{up}}{\|\boldsymbol{w} \times \boldsymbol{up}\|_2}$$

$$\boldsymbol{v} := \boldsymbol{w} \times \boldsymbol{u}$$

$$\alpha := \frac{width \cdot \tan(fov)}{\sqrt{width^2 + height^2}}$$

$$\beta := \frac{height \cdot \tan(fov)}{\sqrt{width^2 + height^2}}$$

$$\boldsymbol{s}(t) = \boldsymbol{eye} + t\frac{x\alpha\boldsymbol{u} + y\beta\boldsymbol{v} - \boldsymbol{w}}{\|x\alpha\boldsymbol{u} + y\beta\boldsymbol{v} - \boldsymbol{w}\|_2}$$

Ha ez megvan, akkor a még mindig mélységtérképpel színezett színtér bejárható lesz.

Normálisokat a derivált normalizálásával kaphatunk könnyen:

$$\mathbf{norm}(\boldsymbol{p}) = \frac{f'(\boldsymbol{p})}{\|f'(\boldsymbol{p})\|_2}$$

A derivált numrikus számítására ajánlott a szimmetrikus differenciaképletet használni (pl. ε = 0.001):

$$f'(x,y,z) = \frac{1}{2\epsilon} \cdot \begin{bmatrix} f(x+\epsilon,y,z) - f(x-\epsilon,y,z) \\ f(x,y+\epsilon,z) - f(x,y-\epsilon,z) \\ f(x,y,z+\epsilon) - f(x,y,z-\epsilon) \end{bmatrix} + \mathcal{O}\left(\epsilon^2\right)$$

Aminek segítségével implementáljuk a Phong árnyalást egy pontfényforrsára.

Ezek után cseréljük le az *f*-et valamilyen érdekes felületére, például fraktáléra. Fraktál távolságfüggvényének előállítása nehéz feladat, ezért itt az **interneten talált kódot elfogadunk, amennyiben a forrás és az azon végzett módosítások megtalálhatóak a kódban** (komment formájában). Javasolt források:

- https://www.shadertoy.com
- http://www.iquilezles.org/www/index.htm

Fraktál előállítása nem kötelező, csak ajánlott. Amennyiben nem fraktált jelenítünk meg, készítsünk egy ötletes CSG színteret, azaz primitívek uniója, metszete, különbségeként előállított alakzatot. A primitívek között legyen sík, gömb, téglatest, tórusz, kúp. Ezekhez innen lehet lesni:

- http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Az alapfeladat összesen **70 pont**. (Kevés pont ugyan levonható, de ennek lényegében meg kell lennie a jegyszerzéshez.)

Választható részfeladatok (80p):

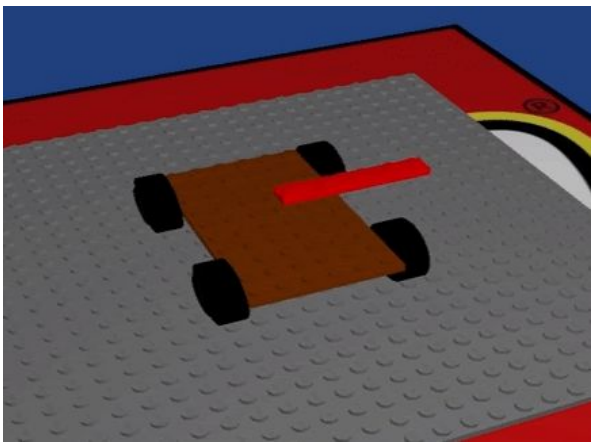
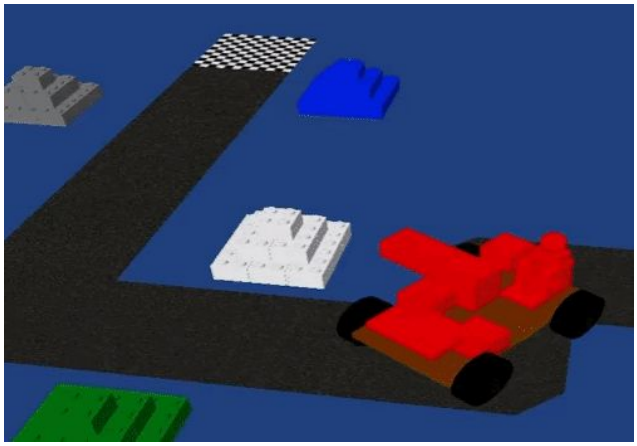
- Árnyéksugarak számítása: a metszéspontból indítsunk el egy sugárkövetést a fényforrás(ok) irányába amennyiben a 〈 n, toLight 〉 > 0, azaz a fényforrás nem a felület mögött van. Amennyiben nem történt metszés számoljuk ki a fényforrás intenzitáskontribúcióját. Hogy biztosan ne történjen önmetszés a sugár kezdőpontját toljuk el a normális irányában egy kicsit. **20p**
- Ambiens eltakarás számolása. Az ambiens fénykomponenst kell beszorozni az eltakarás mértékével. Az alábbi számítás paraméterei az *N*: közelítés iterációinak a száma, *δ*: lépték, *k*: effekt erőssége (pl. *k*=0.5). **10p**

$$A_{\mathbf{p}} = 1 - k \cdot \sum_{k=1}^{N} \frac{1}{2^k} \cdot \big(k\delta - f(\mathbf{p} + k\delta \cdot \mathbf{n})\big)$$

- Fraktál vagy színtérelemek ötletes animálása, fényforrások animációja. **5p**
- Blinn-Phong árnyalás implementálása **5p**
- Skybox vegye körül a színteret. **5p**
    - Az ambiens színt a normális irányba mutató égszín határozza meg. Ehhez külön, egy nagyon elmosott képet töltsünk be. **5p**
    - A felületről számoljunk visszapattanó sugarat, és ezt színezzük a skybox segítségével. **5p**
- Implementáljunk a tükrözött sugarat. **5p**
- Implementáljunk fénytört sugarat. Itt figyeljünk arra, hogy az új sugarat a felület túloldalán indítsuk el (toljuk el a **-n** irányba). **10p**
- Ha mindkét előzőt implementáljuk **+10p**. Megj.: Shaderkódban nem lehet rekurziót írni.

Összesen **150 pont** szerezhető erre a feladatra.

Lego racers (145 pont)

Figyelem! A dőltbetűs, aláhúzott feladatrészek kötelezőek, anélkül nem értékelhető a beadandó!!

Készíts lapos hengerekből és téglatestekből különböző vastagságú és nagyságú építőelemeket, mint a Lego-ban az alábbiak figyelembe vételével: Legyen vékony és vastag elemtípus, 3 vékony elem egymáson = 1 vastag. Lehessen választani 1x1, 2x1, 3x1, 4x1, 6x1, 8x1, 2x2, 2x3, 2x4, 2x6 és 2x8 (széles x hosszú) elemek közül. Az építőkockák megválasztható színűek legyenek. (Minimum: piros, sárga, zöld, kék, fehér, fekete.) Feltétlenül a programban hozd ezeket létre! (Tipp: használj olyan elemtípust, ami transzformációk, uniform változók és saját kirajzolófüggvény segítségével tesz egy elemet egyedivé. Így az adott típusból vektort/listát/tömböt létrehozva könnyen lehet autót építeni később.) **(15p)**
Legyenek különleges elemek is, mint kerék és kormány. Készíts egy versenyzőt is - ezeknél a sematikus ábrázolás elég! **(10p)**

Lehessen ezekből tetszőleges kocsit építeni:
Legyen mindig két pár kerék a talajon (talajt lásd később), páronként 2x4-es vékony elemmel összekötve, 6 egység távolságban egymás előtt. Ez lesz a kocsi kiindulási alapja. **(5p)**
Egyesével tudjunk hozzá alapelemeket építeni. Az elemek között tudjunk választani, elemenként egyedi színekkel (betűk segítségével lehessen változtatni a színt és méretet). A billentyűk lenyomására a kijelölt szabad elem változzon meg! **(10p)**
Az elemeket tudjuk mozgatni (számbillentyűk segítségével). Egy billentyű lenyomására 1 illesztésnyit (4,6: balra és jobbra; 2,8: hátra és előre), vagy egy vékony elem magasságányit (3,9: le és fel) mozduljon. Az 5-re forduljon el 90 fokkal. Ha megnyomjuk a 0-t, akkor az elem legyen rögzített, azaz tekintsük beépítettnek. Ekkor új beépíthető szabad elem jelenjen meg. **(15p)**
Legyen a kocsiban kormány és sofőr! Figyelj arra, hogy a kormány mindig a sofőr előtt legyen. (Egységben helyezzük őket a kocsiba.) Ugyanúgy mi határozhassuk meg a helyüket, mint a többi elemnél. **(5p)**

A kocsi kész, ha entert nyomunk, és minden kötelező elem van rajta. (Csak a beépített elemek tartozzanak a kocsihoz!) **(5p)**

Készíts hozzá pályát!
Egy nagy síkra textúrával rajzolj aszfaltcsíkot! (Paintben vagy hasonlóan igényes rajzolóban is elkészíthető). Legyen rajta startvonal, célvonal és közöttük egyenes és legalább három, köríven kanyarodó szakasz is. Amikor van egy kész kocsink, a felfele nyíllal induljunk el a pályán, a felfelé/lefelé nyilak kontrollálják az állandó haladási sebességet. A kocsi kövesse az azfaltcsík pályáját! **(10p)**
A pályán mi dönthessük el billentyűparanccsal, hogy nappal legyen (sárgás fények, irányfényforrás), vagy éjszaka (kékes fények egy világoskék pontfényforrással a pálya felett). **(5p)**
A kamera kövesse a kocsit, mindig mögötte és fölötte, a kocsi eleje felé nézve. **(5p)**
Legyenek a pályán az elemekből épített tereptárgyak is dekorációként. **(5p)**
A célba érve a kocsi álljon meg, piros fény villogjon, és a kamera tegyen egy kört a kocsi körül azt nézve. Ezután dönthessünk, hogy újra versenyezni, vagy építeni kezdünk. **(10p)**

Plusz pontért:

Figyelj arra, hogy az építésnél az elemek rögzítéskor mindig érintkezzenek egy korábbi elemmel, de ne legyenek egymásban. Figyelj arra is, hogy ne építs a föld alá. **(+10p)**
Ha az 1-et nyomjuk meg, akkor az addigi elemek tetejére illeszti az adott helyen az új elemet. **(+5p)**
Tedd szerkeszthetővé a pályát is! **(+5p)**
Legyen a sík lego-bütykös. **(+5p)**
A kocsi elején legyen egy reflektor, azaz egy spotlight, ami mindig a kocsi elé világít. **(+10p)**
Legyen lehetőség egy-két előre gyártott kocsival versenyezni a pályán, amik konstans sebességgel haladnak végig rajta. Jelölje szürke szín őket. **(+10p)**

Összesen: **145p**

Metaballs (160 pont)

Feladat röviden

Valósíts meg egy valósidejű metaball tracer programot OpenGL-ben, amely egy tetszőleges, futásidőben is dinamikusan módosítható színteret jelenít meg! A teljes képernyőt elfoglaló téglalapot jelenítsünk meg, és a sugárkövető programot a fragment shaderben írjuk meg! Általános követelményeknek nem kell teljesülniük!

A feladat során implicit felületet jelenítünk meg 3D ben, ami olyan mint egy szintvonal a (2D-s) térképen (egyenlő magasságban lévő pontok). Hogy melyik szintvonalat rajzoljunk ki azt a *tr* thresholddal fogjuk tudni állítani (lásd később). Implicit felületekből metalabdákat fogunk megjeleníteni, ezek hasonlóak a vízcseppekhez, ezért egy részecske-rendszer szimulációja megjeleníthető folyadékként ennek segítségével.

Metalabdák és meta-felületek

Implicit függvényként adjuk meg őket, tehát az egész színteret le tudjuk írni egy folytonos függvénnyel: legyen $tr \in \mathbb{R}$, $F : \mathbb{R}^3 \rightarrow \mathbb{R}$, ahol

$$F(x,y,z) - tr = 0$$

egyenlet x,y,z megoldása a színtér felületének pontja. Ahol $F(x,y,z) > tr$ az van “kívül”, és ahol $F(x,y,z) < tr$ az van “belül” az alakzatban. Például az $x^2+y^2+z^2-1=0$ a gömb implicit egyenlete, részletekért lásd előadás diáit.

Az implicit függvény és raymarching technika lehetőséget ad alakzatok uniója, metszete, “összege” stb. alakzatok konstruálására. Ez a beadandó ezt fogja demonstrálni.

Legyen $f_{c,r} : \mathbb{R}^3 \rightarrow \mathbb{R}$ függvény a metalabda implicit függvénye, ahol $c \in \mathbb{R}^3$ a középpont, $r \in \mathbb{R}$ a sugár. Definiáljuk a következőképp: (lehet sokféleképp) ($p \in \mathbb{R}^3$ pont)

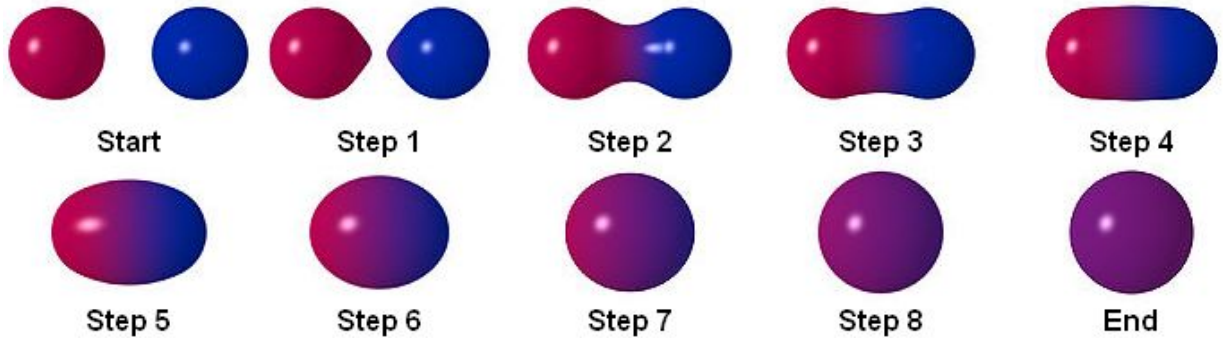
$f_{c,r}(p) := 1$,ha $d < 0$

$f_{c,r}(p) := 2 \cdot d^3 - 3 \cdot d^2 + 1$,ha $d \in [0,1]$

$f_{c,r}(p) := 0$,ha $d > 1$

ahol $d = \|c-p\|_2 / r$ normalizált távolság, valamint *tr* ∈ (0,1) pedig egy "globális" küszöbszám.

Ekkor pl az $F(x,y,z)=f_{(1,0,0),1}(x,y,z)+f_{(-1,0,0),1}(x,y,z)=tr$ implicit egyenlet két metalabdát írnak le (1,0,0) középponttal és 1 sugárral, valamint (-1,0,0) középponttal és 1 sugárral. Ha fixáljuk a *tr* =0.5-re, akkor két labda fog egymásba érni. Ha egymáshoz közelítünk kettő ilyen labát (középpontjaiktól), akkor (kép a wikiről):



Színtér leírása

A labdákat uniform tömbben adjuk át, amiben vec4-ek vannak. Az első három koordináta a metalabda középpontjának pozíciója, az utolsó a sugarát jelentse (milyen távolságra "hat"). Adjunk át 4-5 metalabdát, legalább az egyik labda pozíciója, és legalább az egyik labda sugara legyen animálva (időfüggő). Írjuk meg a fenti függvényeket a fragment shaderben. Legyen egy F függvény mely egy **vec3**-hoz egy **float**-ot rendel, és a fent látott módon leírja a színteret (uniformokat felhasználja). (**15%**)

Raymarch

Kezdetben készítsünk úgynevezett mélységtérképet: a távolság függvényében színezzük szürke árnyalatosra a képet. Mivel kamera még nincs, induljunk ki pl. az y=4 magasságban egy négyzetből és pásztázzuk lefelé.

A pásztázás, azaz raymarch algoritmusa úgy működik, hogy egy p_0+t*v_0 egyenletű sugárnál a *t*=0 ból kiindulva, egy kicsi értékkel *t*-t növelgetjük, és a kapott térbeli, sugáron lévő pontot behelyettesítjük az F függvénybe. Amint F értéke előjelet vált, a t és az előző t közt volt egy metszéspont a felülettel. (Bolzano tétel)

A fenti algoritmus segítségével készítsünk el egy mélységtérképet, azaz mindhárom színcsatorna legyen arányos a metszéspont $p_0 + t_0 \cdot v_0$ kamerától való távolságával, azaz t_0-al. (egy maximális távolság előtt álljunk meg). (Raymarch **20p**)

Viszont ez közel nem ad megfelelően pontos eredményt, ezért logaritmikus kereséssel pontosítsuk az eredményt! (**10p**) A mélységtérképünk így sokkal simább kell legyen.

Kamera

A kamera **eye**, **at** és **up** vektorát uniform változóban adjuk át, és számoljunk ki egy kamera koordináta rendszert, valamint a pozícióhoz tartozó sugarat. (lásd előadás diái) (**10p**) A látószög és vetítési sík távolsága lehet konstans. A kamerát lehehessen WASD billenytűkkel és az egérrel mozgatni, forgatni. Ablak átméretezésnél ne torzuljon a színtér (**10p**)

Normálisok számítása

A normálisokat kétféleképp számolhatjuk, elég az egyiket implementálni. Tipp: a normális vektorokat jelenítsük meg a fragment színeként, hogy ellenőrizzük a módszert!

Kiszámolhatjuk a $f_{c,r}(x,y,z)$ deriváltját az *x,y,z* koordináták szerint, a kapott gradiensvektort pedig normalizálva kapjuk a felületi normálist. Ha a hallgató ezt az utat választja, mutassa meg a parciális deriváltak számolását! (**10p**)

A másik megoldás a centrális diferenciával közelítés. Legyen $\varepsilon > 0$ egy kicsi szám, ekkor a gradiens közelíthető az

$$[F(x+\varepsilon,y,z)-F(x-\varepsilon,y,z) \quad F(x,y+\varepsilon,z)-F(x,y-\varepsilon,z) \quad F(x,y,z+\varepsilon)-F(x,y,z-\varepsilon)]^T /2$$

vektorral. Normalizálásával normális irányú vektorhoz jutunk.(**10p**)

Fények számítása

Helyezzünk két-három fényforrást a színtérbe, számoljunk normálvektort és valósítsuk meg a Blinn-Phong árnyalást minden fényforrásra, és összegezzük az eredményt. (**5p**)

Számítsunk árnyéksugarakat! (**10p**);

Eddig összesen 90% szerezhető.

Egyéb feladatok:

A további feladatokra egymástól függetlenül összesen 70% szerezhető. A korábbi részfeladatoknak nem kell maradéktalanul meglenniük ezen pontok megszerzéséhez!

- Fényforrások erőssége csillapodjon a távolság függvényében (lásd előadáson a négyzetes csillapítás), fénynek legyen spekuláris, diffúz színe, adjunk át több fényforrást uniform tömbben. (**5p**)
- Készítsünk súlytalansági folyadékszimulációt! Egy láthatatan dobozon belül pattogjanak a labdák random kezdősebességgel és iránnyal. Ha két labda középpontjának távolsága kisebb mint $tr\cdot(r_1+r_2)$, akkor változtassák irányukat egymás a másik labda irányába, és mikor távolságuk már elég kicsi, akkor válljanak egy labdává! A megfelelő méretű sugarat késérletezzük ki! (**15p**)
- Találjunk ki más függvényeket is a $6\cdot d^6 - 15\cdot d^5 + 10\cdot d^3$ helyett, találjunk legalább négy ilyet és implementáljuk (azaz bemutatáskor írjuk át ott a kódot). (**5p**)
- Optimalizált kódra és egyszerűsítésekre szerezhető (**10p**)

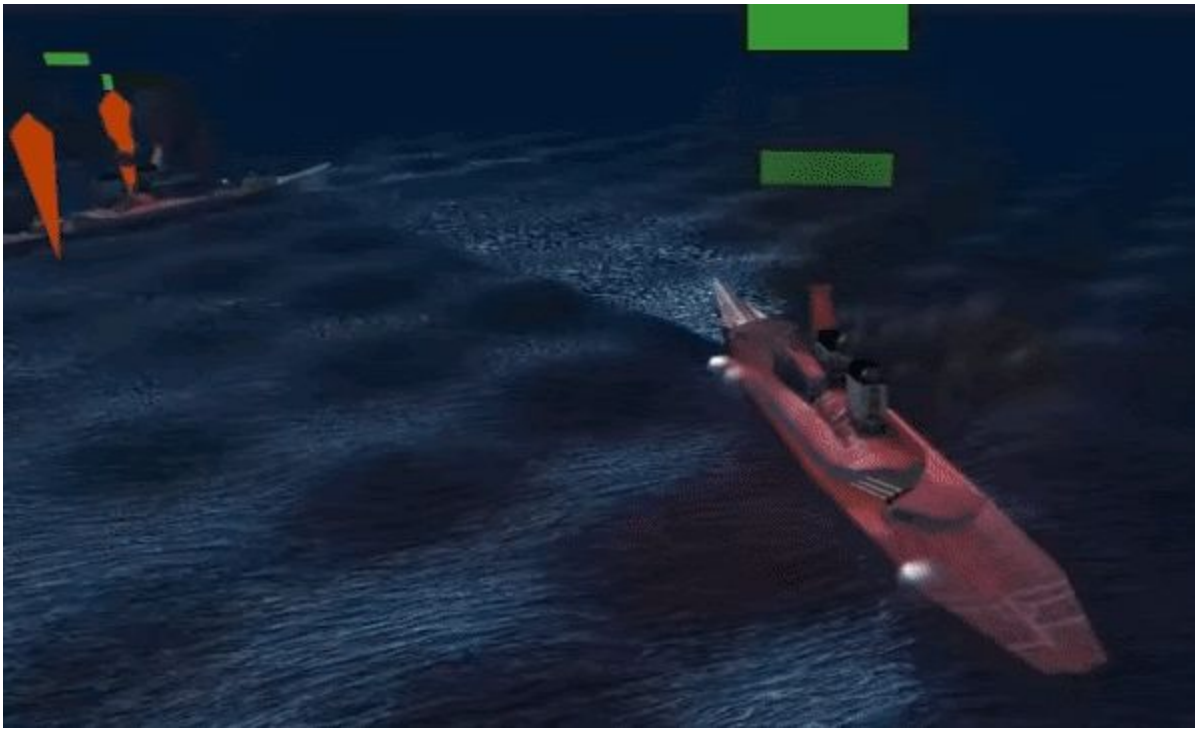
Metalabda álltalánosítása

A továbbiakhoz álltalánosítsuk a metalabdát. Legyen $A \subseteq \mathbb{R}^3$ egy kompakt halmaz, és legyen $p \in \mathbb{R}^3$ egy térbeli pont. Legyen $d \in \mathbb{R}$ olyan, hogy $d \cdot r$ a halmaz és a pont távolsága, ahol a $r \in \mathbb{R}$ egy az A halmazhoz adott “sugár”. Így a fent látott függvény ezzel a d távolsággal adja a meta-halmazt. Ha $A=\{c\}$ egyetlen pontból áll, kapjuk a meta-labdát. Világos, hogy egy konkért halmaz esetén a feladatunk a halmaz legközelebbi pontjának meghatározása, mert ha ez megvan, akkor ezt r-el elosztva a normalizált d-t kapjuk.

- Adjunk két ponttal definiált meta-szakaszt a színtérhez! (Kapszula lesz belőle.) (**10p**) (Építsünk stickman-t)
- Adjunk egy középponttal és egy sugárral, valamint egy normálissal definiált meta-kört a színtérhez! (Tórusz (szerű) lesz). Segítség: határozzuk meg a legközelebbi pontot. Vetítsük p-t a kör síkjára, majd majd határozzuk meg a kör középpontjából a vetületi pont irányában lévő, sugár távolságra lévő pontot. (**15p**)
- Adjunk meg egy ponthalmazt A nak, ekkor ezek nem “interferálnak” egymással! (**10p**)

Összesen **160%** szerezhető. Kérdéseitekkel forduljatok Bálint Csaba gyakorlatvezetőhöz. E-mail: csabix.balint@gmail.com.

World of Warships (130 pont)



A feladat egy csatahajókkal játszható játék megalkotása.

A vízfelület legyen egy kellően nagy méretű négyzet az XZ síkon, ami lassan hullámzik, különböző frekvenciájú szinuszoidok szerint X és Z irányban (tipp: a hullámzás könnyen megvalósítható a vertex shaderben!) **(5p)** Bump mappinggal vagy normal mappinggal együtt lássuk el víztextúrával! (A szükséges elméletről ITT találsz diákat) **(15p)**

A pályát vegye körbe egy hegyvonulat; a hegyet alkotó vertexek X és Z pozíciója legyen fix, de az Y magasságuk legyen véletlen generált. Vigyázzunk, hogy a véletlen generálás során ne kapjunk absztrakt/lehetetlenül kinéző hegyeket! Azaz a hegyvonulatot alkotó vertexek közül az egymáshoz közel esők Y magasságértéke legyen hasonló! **(5p)** A hegy színe az egyes vertexek/fregmensek esetén Y érték szerint változzon: azok a vertexek vagy fregmensek, akik vízszinten vannak, legyenek zöldek, és minél magasabban helyezkedik el egy vertex vagy fregmens, annál fehérebb! **(5p)**

A pálya két átellenes sarkán legyen egy-egy világítótorony (henger, tetején kúppal), mindkettőnek a csúcsában legyen egy-egy pontszerű fényforrás. Mindkét fényforrásnak legyen diffúz és spekuláris fénykomponense (ezek legyenek különbözőek!), illetve legyen ambiens fényünk is. **(10p)**

Modellezzünk csatahajókat! A csatahajók legyenek textúrázottak; a modellezésnél nem kell túlzásba esni, de ránézésre fel lehessen ismerni, hogy csatahajóról van szó! **(5%)** Minden

csatahajónak legyen lövegtornya (sokszög alapú hasábból és hengerből), a lövegtornyok külön tudnak forogni: egyrészt körbe tudjuk forgatni 360 fokkal, másrészt fel-le tudjuk állítani bizonyos dőlésszögben (vízszinteshez képest lefele 20 fokkal, felfele 60 fokkal). Több hajót is helyezzünk el a színtéren (legalább ötöt), ezek közül egyet mi fogunk irányítani, a többit a "mesterséges intelligencia". **(5p)**

A hajók haladni tudnak a vízen. A hajók az alábbi mozgára képesek: mozgás előre, mozgás hátra, fordulás, valamint a lövegtorony kezelése. A mozgás minden esetben legyen sima, azaz a fordulás is legyen animált (ne "ugorjon" a hajó). A hajók közül egyet mi irányíthatunk, a többi esetén az irányításba nincs beleszólásunk. Minden hajó követi a víz hullámzását! (Sose merül el, de nem is lebeg a víz felett.) **(10p)** A hajók gyorsulva illetve lassulva haladjanak (egy bizonyos maximális sebességig), ha fordulni kezdenek, veszítenek a sebességükből. **(5p)**

A hajók lövegtornyával lehessen lőni. Ekkor egy szürke henger (a lövedék) hagyja el a lövegtornyot az adott irányba. A lövedékre hat a gravitáció, azaz a lövedék egy parabola-pályát ír le. **(5p)** Ha a lövedék eltalált egy hajót, akkor a hajó megsérült: minden hajó 10 pontnyi "életerővel" rendelkezik, a találat 2-5 pontot vesz ebből le (véletlen generált érték). Ha a lövedék nem találta el hajót, akkor eltűnik ha a magassága a vízszint alá esik. Hegybe történő becsapódást nem kell vizsgálni. **(5p)** Minden hajó fölött legyen egy "életerőt" jelző csík (téglalap), ez mindig úgy álljon, hogy a kamera irányába néz (azaz bárhogy forognak a hajók és bárhol van a kamera, mindig teljes szélességben látjuk a csíkot, de a csík követi a hajó haladását) **(10p)** Ha egy hajó megsemmisül, akkor animálva elsüllyed: orral lefelé fordul, majd elindul a mélybe, és eltűnik ha már nem látjuk. **(5p)**

A kameránk végig követi a hajónkat (felette lebeg), de szabadon tudjuk körbeforgatni. Szóköz billentyű hatására a kamera "elengedi" a hajónkat, ekkor a pályát szabadon bejárhatjuk, de a szóköz újbóli megnyomására visszaugrunk a hajónkhoz. **(5p)**

Az ellenséges hajók maguktól mozogjanak és támadjanak. A mozgásuk legyen véletlen, de csak 5 mp-enként változzon (például elfordul valamekkora szöggel, majd halad 5 mp-ig, majd megint elfordul, megint halad, és így tovább). **(5p)** Nem úsznak ki a pálya széléről! **(5p)** Ha közel vannak hozzánk, akkor felénk fordítják a lövegtornyukat, és tüzelnek, ezt megpróbálják a lehető legpontosabban véghez vinni: ehhez számítsuk ki a lövegtorony állását! A víz hullámzását ne vegyük figyelembe, így persze időnként mellélőnek. **(15p)** Az ellenséges hajók csak bizonyos időközönként tüzelnek újra. (Ha így is gyorsan kilőnének minket, állítsuk nyugodtan magasabbra a mi hajónk kezdeti életerejét.)

A mi hajónk orrán legyen egy reflektor (spot lámpa), ami mindig előrefele áll. A reflektornak legyen diffúz komponense. Ügyeljünk rá, hogy a reflektor nem csak egy pontszerű fényforrás, hanem csak adott szögtartományban világít, egy bizonyos irányba! Ezen felül a spot lámpánk fényerőssége a távolság függvényében fokozatosan csökkenjen. **(10p)**