

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Simulace davu

Crowd Simulation

2018

Adam Lasák

Zadání bakalářské práce

Student:

Adam Lasák

Studijní program:

B2647 Informační a komunikační technologie

Studijní obor:

2612R025 Informatika a výpočetní technika

Téma:

Simulace davu
Crowd Simulation

Jazyk vypracování:

čeština

Zásady pro vypracování:

V dnešní době stále přibývá důvodů, proč se zaobírat tématem počítačové simulace davu agentů a možnosti využití se stále rozšiřují. Dnes se již simulace davu využívá např. při testování návrhů veřejných budov nebo při vytváření animací davových scén ve filmu.

Cílem této práce je vytvořit grafickou testovací aplikaci, ve které bude možno simulovat, testovat a porovnávat chování davu s možností nastavení a úpravy zadání, včetně možnosti zasáhnout do chování agentů.

1. Nastudujte teorii zabývající se chováním davu, zaměřte se na algoritmy počítané v reálném čase (např. model Boids).
2. Popište základní algoritmy používané pro simulaci davu.
3. Vytvořte aplikaci simulující chování davu včetně vizualizace jednotlivých agentů.
4. Vytvořte praktické ukázky a jednotlivé testy srovnajte a vyhodnotěte.

Seznam doporučené odborné literatury:

- [1] Craig Reynolds: "Boids Background and Update", [online], <http://www.red3d.com/cwr/boids/>
[2] Addison-Wesley: "OpenGL Programming Guide", (Red book) [online].

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry



prof. Ing. Pavel Brandstetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 26. dubna 2018

.....

Tímto bych rád poděkoval svému vedoucímu, Ing. Martinu Němcovi, Ph.D., za poskytnuté odborné rady, za neocenitelné zkušenosti a za všechnen čas, jenž mi takto věnoval.

Abstrakt

Spousta věcí v přírodě je stejně působivých jako zvířata, která se mohou organizovat do větších a logicky orientovaných seskupení. Tím že dokážeme simulovat toto chování, můžeme vytvořit reálnou podobu davu. Reálné využití pak najdeme v konkrétních oborech lidské činnosti: návrhy veřejných budov, filmové projekty či programování PC her. Tato práce se zaměřuje na popis boidova algoritmu, který je dnes nejpoužívanější co se simulace davu týče.

Klíčová slova: Boidův algoritmus, dav, hejno, koheze, separace, zarovnání, agent

Abstract

Many things in nature are impressive like animals which can be organized into larger and logical oriented grouping. In that case when we can simulate this behavior than we can create real form of the crowd. Real use can be found in things like: creating public buildings, movie projects or developing games. This thesis focuses on the description of boid's algorithm which is the most used crowd simulation principle today.

Key Words: Boid's algorithm, crowd, flock, cohesion, separation, alignment, agent

Obsah

Seznam použitých zkratek a symbolů	2
Seznam obrázků	3
Seznam tabulek	4
Seznam výpisů zdrojového kódu	5
1 Úvod	6
2 Simulace	7
2.1 Simulace davu	7
2.2 Částicová simulace	7
2.3 Simulace na bázi AI (Artificial Intelligence)	7
2.4 Boidův algoritmus	8
2.5 Separace	9
2.6 Zarovnání	11
2.7 Koheze	13
2.8 Aplikování tří pravidel	14
3 Praktická část	16
3.1 Popis realizace demonstrační aplikace	16
3.2 Použité technologie	16
3.3 Problémy při implementaci	16
3.4 Steering behaviors	19
3.5 Třídní diagram	21
3.6 Výsledné aplikování pravidel	21
3.7 Konfigurační soubor	22
3.8 Spuštění scény	23
3.9 Únik z budovy	23
3.10 Testování úniku z budovy	28
3.11 Simulace hejna ryb	31
4 Zhodnocení	33
5 Závěr	36
Reference	37
Přílohy	38

Seznam použitých zkratek a symbolů

ACM	– Association for Computing Machinery - vědecky-vzdělávací instituce pro výpočetní technologie
AI	– Artificial Intelligence - umělá inteligence
FPS	– Frames per seconds - počet snímku za jednu sekundu
GLEW	– OpenGL Extension Wrangler Library - poskytuje realtime prostředky pro danou platformu
GLFW	– Graphics Library Framework - Multiplatformní technologie rozšiřující OpenGL o vytváření aplikačních oken
GLM	– OpenGL Mathematics - poskytuje širokou škálu matematických operací pro OpenGL
GPU	– Graphics processing unit – grafická karta, má svůj procesor i výpočetní paměť
IDE	– Integrated Development Environment - program usnadňující práci programátorem
Particle System	– Definuje shluky malých částic s podobným chováním a vlastnostmi
RAM	– Random Access Memory – operační paměť počítače
Realtime	– Vykreslování v reálném čase – snaha vykreslovat co nejrychleji
SIGGRAPH	– Special Interest Group on Computer GRAPHics and Interactive Techniques - výroční konference v počítačové grafice

Seznam obrázků

1	Ukázka simulace hejna ryb v Unreal Enginu 4 (Zdroj: [5])	8
2	Separace - vyhýbání se ostatním agentům (Zdroj: vlastní)	10
3	Zarovnání - určení směru agenta vůči průměrnému směru ostatních (Zdroj: vlastní)	12
4	Koheze - určení směru k průměrné lokaci okolních agentů (Zdroj: vlastní)	13
5	Graf závislosti paměťového růstu na čase	17
6	Diagram řídicích tříd	21
7	Ukázka nákresu budovy v textovém souboru	25
8	Ukázka vyrenderovaného půdorysu budovy	26
9	Problém uváznutí při špatném určení únikových bodů	27
10	Graf závislosti času úniku na počtu pater	29
11	Ukázka nákresu labyrintu	30
12	Ukázka hejna ryb	31
13	Vyrenderovaná dvoupatrová budova	34
14	Budova přepnuta do průhledného režimu se zviditelněním agentů a únikových bodů	34
15	Průhledný režim budovy z boční perspektivy	35
16	Vyrenderované hejno ryb s využitím se objektu	35

Seznam tabulek

1	Tabulka nárustu využití paměti RAM	17
2	Tabulka naměřených hodnot - schéma 1	28
3	Tabulka naměřených hodnot - schéma 2	29
4	Tabulka naměřených hodnot - labyrinth	30

Seznam výpisů zdrojového kódu

1	Pseudokód pro separaci (Zdroj: vlastní)	10
2	Pseudokód pro zarovnání (Zdroj: vlastní)	12
3	Pseudokód pro kohezi (Zdroj: vlastní)	14
4	Pseudokód pro aplikování třech pravidel (Zdroj: vlastní)	15
5	Ukázka eliminace nárustu paměti	17
6	Použití klasického cyklu	18
7	Použití optimalizovaného cyklu	18
8	Ukázka použití kopírovacího konstruktoru	18
9	Vypočtení seek vektoru	19
10	Nastavení výsledné pozice agenta	22
11	Ukázka komentáře v konfiguračním souboru	23
12	Princip metody getArriveVector(...)	26

1 Úvod

Simulační algoritmy se používají v širokém spektru odvětví od vědy, her, výpočetních úkonů až po kinematografii či stavbu budov [1]. Herní využití mívá velmi reálně implementována armáda [2], která tímto způsobem zaškoluje vojáky ve virtuálním simulačním boji jak v taktice, tak způsobu nejfektivnějšího využití dostupných zbraní.

Pokud vezmeme v potaz stavbu či projektování budov, nabízí se další subspektra druhů simulačních programů. Například projektant potřebuje nasimulovat jak velkou zátěž udrží hlavní nosníky, testování a simulování různých druhů materiálů či jak velké budou úniky tepla na základě pohybů osob v budově.

Avšak po této základní konstrukční stránce, se musí také navrhnout optimální velikost budovy a kolik bude schopna pojmit lidí v jednom okamžiku. Kde bude vést úniková cesta v případě požáru a kolik času zabere evakuovaným lidem (*davu*), než se z budovy dostane do bezpečí. A zrovna co se bezpečnosti týče, mají simulační algoritmy nejsírší využití. V jistém slova smyslu by se dalo říci, že byly vytvořeny primárně pro tento účel [3].

Při těchto simulacích se tak navrhují nejlepší varianty šířky chodeb, dostupnosti k únikovým východům, přístupu k požárnímu schodišti nebo vyládění ukazatelů směru úniku.

Pokročilejší algoritmy také reagují na různé překážky, jak horizontální tak vertikální. Dokáží simulovat dav, který jde z jednoho patra do druhého různými typy cest a střetává se tak s jinými davy. Cílem projektanta je pak vybrat vhodné únikové cesty z budovy a zpracovat nejoptimálnější únikový plán.

V této práci se seznámíme se základy simulování davu a hejna. Popíšeme nejpoužívanější algoritmus [4] pro tento druh simulace a realizaci výsledné aplikace.

2 Simulace

Pojem simulace lze vydefinovat: "*Počítačová simulace je napodobení skutečnosti pomocí numerického výpočtu, nezbytná součást modelování fyzikálních procesů. Dokáže předpovědět jak kvantitativní, tak kvalitativní výsledky pokusů při různých počátečních podmínkách. Umožňuje omezit výběr jevů, které celý pokus ovlivňují nejvíce a tím vysvětlit příčiny a podstatu procesů.*" (Citace [9])

V našem případě se jedná o napodobení davu, kdy pomocí algoritmu dokážeme předem modifikovat nadcházející krok a tím i změnit výsledné chování celého davu. Na základě vstupních dat nebo kombinací pravidel můžeme chování měnit a tím simulovat dav při různých situacích, jako je evakuace z budovy nebo běžné chození po prostorách nákupního střediska.

2.1 Simulace davu

Existují v zásadě dvě možnosti, jak simulovat dav lidí. Částicová simulace a simulace založená na umělé inteligenci. [14]

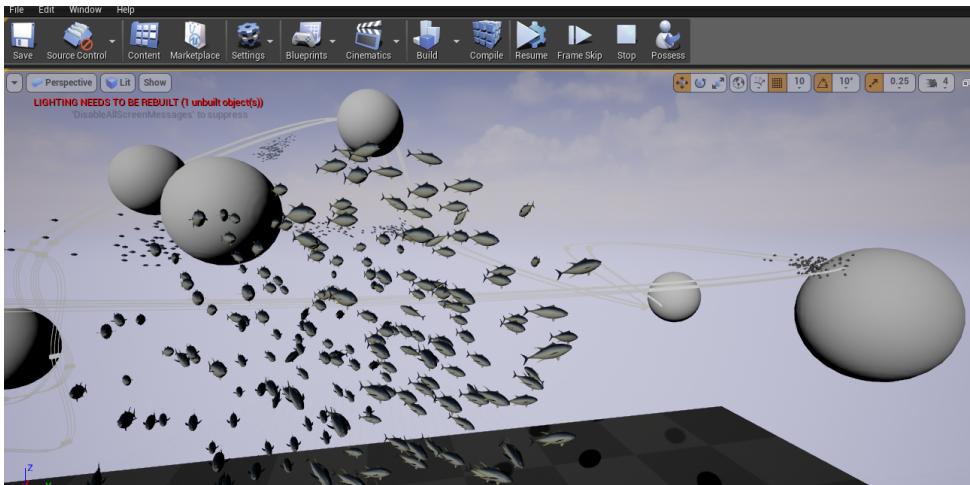
2.2 Částicová simulace

První zmíněná simulace je založena na principu přiřazení hmotného bodu každému chodci ve scéně, jehož pohyb je odviesen působením nejrůznějších sil. Hlavní výhoda spočívá v nasimulování velkého množství lidí z důvodu nenáročnosti výpočtu na výpočetní výkon, ale výsledná vizualizace není příliš reálná. Příklady pro využití jsou typy založené na: magnetických silách, buněčném modelu a sociálních silách. [14]

2.3 Simulace na bázi AI (Artificial Intelligence)

Druhý zmíněný typ je založen na bázi Agentů. Ti už nejsou reprezentováni jen obyčejnými silami (*i přesto že se jedná o diametrálně odlišný typ simulace davu, určité fyzikální vlastnosti agenti stále musí mít*) ale také přidanými vlastnostmi, kterými předchozí typ tolik nedisponoval. Mají hlavně přidané senzory, díky nimž dokáží vyhodnocovat danou situaci v reálném čase a následně se rozhodovat k dalšímu nejvhodnějšímu kroku. Pokud do scény vložíme dva a více agentů kteří budou na sebe reagovat, můžeme říci, že každý z nich má v jisté míře svůj mozek. At už jednodušší (*rozhodování dalšího kroku mezi zdmi*) nebo složitější (*např.: agent může v danou chvíli reagovat, zda-li má jít rychleji či pomaleji aby nezpůsobil kolizi s jiným agentem či agenty*).

Tento typ simulace hojně využívá herní průmysl, kdy v nějaké scéně - herní mapě, jsou nasazeny desítky agentů, kteří se ve své podstatě starají sami o sebe a přímo či nepřímo komunikují s uživatelem. Obecně však platí, že ve hrách je tato implementace mnohonásobně složitější než částicová simulace kvůli mnoha vlastnostem agentů. Konkrétními případy může být třeba hra Crysis (leden 2007 - stáda jelenů) a FarCry (2004 - hejna tropických papoušků [6]).



Obrázek 1: Ukázka simulace hejna ryb v Unreal Enginu 4 (Zdroj: [5])

Je třeba také zmínit kinematografický průmysl, který simuluje davu lidí. Tím pádem animátorům odpadne kus práce, kdy by museli každého agenta animovat zvlášť.

Jelikož byl v této práci využit tento typ simulace, tak v dalších kapitolách budeme každý bod ve scéně nazývat **“agentem”**.

2.4 Boidův algoritmus

Nejčastějším typem simulace založené na umělé inteligenci je Boidův algoritmus vytvořený v roce 1986 Craigem Reynoldem a oficiálně představený roku 1987 na konferenci ACM SIGGRAPH [12, 13].

Název Boidův algoritmus je odvozen od **“boidů”** (*zkratka z anglického bird-oid object*) což referuje na simlaci hejna ptáků. Jak již bylo zmíněno výše, simulace založená na umělé inteligenci obsahuje základní prvky částicové simulace, ale má také něco navíc. Výjma senzorů mají také přehled o celkové geometrii celé scény. Tzn. že každý agent (*boid*) má přehled o všech agentech v celé scéně. Pokud bychom tedy maximalizovali důležitá tři pravidla tohoto principu popsáné níže, znamená to, že každý iteruje s každým.

Mezi částicové prvky Boidova algoritmu lze použít například tření, zrychlení nebo okamžitou rychlost. Reálně lze tření ještě rozdělit na tření válcove, tření způsobené protivětrem nebo pokud bychom určili jako objekt auto, můžeme použít další fyzikální vlastnosti jako točivý moment případně brzdná dráha.

Existuje však celá řada vylepšení, která se dají s tímto algoritmem provést, například simulace chování dopravy. Ta funguje jako tzv. chování založeného na řízení (*Steering behaviors*) [15].

V této práci byla tato vylepšení použita, čímž lze chování davu více přiblížit realitě. Přidané fyzikální veličiny jsou již zmíněné: tření, zrychlení a okamžitá rychlosť.

Následuje popis Boidova algoritmu, který obsahuje tři základní vlastnosti:

1. Separace (*Separation*)
2. Zarovnání (*Alignment*)
3. Koheze (*Cohesion*)

2.5 Separace

Separace slouží k tomu, aby se dva a více agentů nepřiblížili příliš blízko sebe a tím vyvolali kolizi mezi sebou. Je to první a nejzákladnější vlastnost Boidova algoritmu bez které by nebylo možné vytvořit kompletní simulaci.

Oddělení od ostatních agentů funguje na principu základních operací s vektory (*2D nebo 3D*) a vytvoření pomyslného kruhu kolem každého agenta, který jej upozorní zda-li není příliš blízko jiného agenta.

Následující kroky provedou separaci agentů:

1. Odečtení vektoru agenta od vektoru aktuálně kontrolovaného agenta:

$$v_{sub} = v_{current} - v_{actual}$$

kde v_{sub} je výsledný vektor, $v_{current}$ je vektor agenta, který je právě v iteraci a v_{actual} je vektor aktuálně kontrolovaného agenta.

2. Normalizování výsledného vektoru z předchozího kroku:

$$v_{norm} = \sqrt{vx_{sub}^2 + vy_{sub}^2 + vz_{sub}^2}$$

kde v_{norm} je výsledný normalizovaný vektor v_{sub} .

3. Vydělení výsledku kroku č. 2 vzdáleností těchto dvou agentů. Výsledek se přičítá k výslednému vektoru v_{sum} . Matematický zápis pak vypadá následovně:

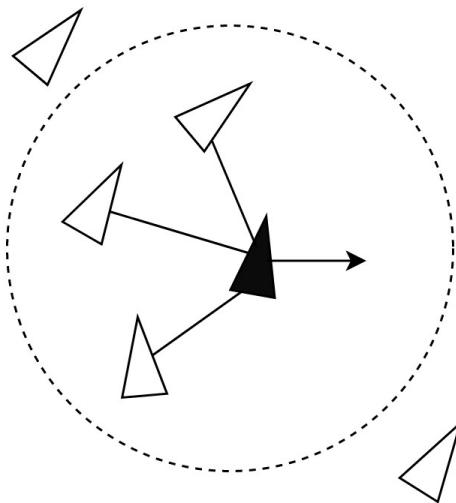
$$v_{distance} = \sqrt{d_x^2 + d_y^2 + d_z^2}$$

$$v_{divide} = \frac{v_{norm}}{v_{distance}}$$

$$v_{sum} = v_{sum} + v_{divide}$$

kde d_x , d_y a d_z jsou rozdíly daných souřadnic mezi agentem v iteraci a aktuálně kontrolovaným agentem, $v_{distance}$ je výsledný vektor vzdálenosti mezi těmito dvěma agenty a v_{divide} je výsledek vydělení vektorů.

4. Pokud je v okolí agenta více agentů se kterými koliduje, pak se na konci výsledný vektor vydělí počtem těchto agentů (viz obrázek č. 2).



Obrázek 2: Separace - vyhýbání se ostatním agentům (Zdroj: vlastní)

```

function Separation(boid paramBoid)
{
    Vector result
    int count

    for (b in allBoids) {
        if (b != paramBoid){
            distance = distanceBetween(b, paramBoid)
            if (distance < SEPARATION_DIAMETER){
                sub = subtractVectors(b, paramBoid) // b - paramBoid
                normalize(sub)
                divideScalar(sub, distance) // sub / distance
                result = result + sub
                count = count + 1
            }
        }
    }

    return divideScalar(result, count) // result / count
}

```

Výpis 1: Pseudokód pro separaci (Zdroj: vlastní)

2.6 Zarovnání

Dalším důležitým pravidlem pro fungování Boidova algoritmu je zarovnání. Spolu se separací bez koheze má dav již základní podobu chování.

Zarovnání má za úkol soudržnost. Všem agentům ve scéně poskytuje schopnost sladit se, jak je zobrazeno na obrázku č. 3.

Princip fungování je obdobný jako u separace, pokud se v blízkosti okruhu od daného agenta objeví jiný agent případně agenti, zprůměruje se jejich rychlosť a směr a na základě těchto údajů modifikujeme údaje konkrétního agenta a tím zajistíme soudržnost všech dohromady.

Pokud bychom chtěli aplikovat zarovnání a konkretizovat průběh, musíme se držet těchto kroků:

1. V cyklu, který prochází všechny agenty ve scéně se testuje vzdálenost konkrétního agenta od jiného, který je momentálně v dané iteraci cyklu pod určitým indexem. Pokud je tato vzdálenost menší než průměr kruhu, přičtou se souřadnice vektoru daného agenta k lokálnímu sumárnímu vektoru v_{sum} . Matematický zápis vypadá následovně:

$$v_{distance} = \sqrt{d_x^2 + d_y^2 + d_z^2}$$

$$v_{sum} = v_{sum} + v_{divide}$$

kde d_x , d_y a d_z jsou rozdíly daných souřadnic mezi agentem v iteraci a aktuálně kontrolovaným agentem, $v_{distance}$ je výsledný vektor vzdálenosti mezi těmito dvěma agenty.

2. Jestliže se podmínka nesplnila a tudíž sumární vektor v_{sum} je roven nule, pak se pravidlo zarovnání ukončuje. Pokud se alespoň jednou podmínka v kroku č. 1 provedla, tak následují kroky:

- sumární vektor v_{sum} se vydělí skalární hodnotou rovné počtu splněných podmínek v kroku č. 1:

$$v_{sum} = \frac{v_{sum}}{c}$$

kde c je počet splněných podmínek.

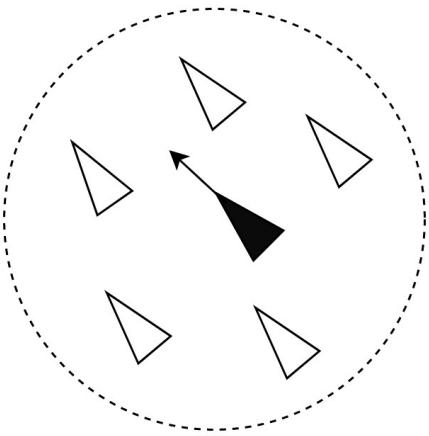
- normalizování sumárního vektoru v_{sum} :

$$v_{sum} = \sqrt{vx_{sum}^2 + vy_{sum}^2 + vz_{sum}^2}$$

- vypočtení výsledného vektoru dle kterého se upraví směr daného agenta vzorcem:

$$v_{res} = v_{sum} - v_{velocity}$$

kde v_{sum} je sumární vektor a $v_{velocity}$ je rychlosť aktuálneho kontrolovaného agenta.



Obrázek 3: Zarovnání - určení směru agenta vůči průměrnému směru ostatních (Zdroj: vlastní)

```
function Alignment(boid paramBoid) {
    Vector sum
    int count

    for (b in allBoids) {
        if (b != paramBoid){
            distance = distanceBetween(b, paramBoid)
            if (distance < ALIGNMENT_DIAMETER){
                sum = sum + b.velocity
                count = count + 1
            }
        }
    }

    if (count > 0){
        divideScalar(sum, count) // sum / count
        normalize(sum)
        return subtractVectors(sum, velocity) // sum - velocity
    } else {
        return sum
    }
}
```

Výpis 2: Pseudokód pro zarovnání (Zdroj: vlastní)

2.7 Koheze

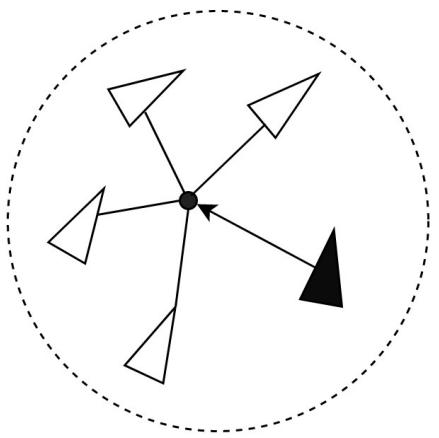
Při simulaci pomocí boidových třech pravidel je koheze¹ tím nejméně důležitým pravidlem. Pokud bychom aplikovali pouze dvě předchozí pravidla a sice separaci a zarovnání, pak by výsledek už vykazoval známky chování davu.

Funguje na principu spojení okolních agentů (*a následně vytvoření skupiny*), kteří zasahují do dalšího pomyslného kruhu. Aby toto pravidlo fungovalo správně a nekolidovalo s podmínkami separace pak musí platit $constCoh > constSep$, kde $constSep$ je maximální průměr kružnice ve které se při překročení tohoto průměru řeší podmínky separace a $constCoh$ je maximální průměr kružnice u koheze. Pokud by podmínka byla opačná, koheze se provede špatně nebo vůbec. Průměry kružnic koheze a zarovnání však na sebe vliv nemají.

Ve výsledné fázi se tedy musí určit průměrná lokace mezi všemi agenty kteří spadají do okruhu daného (*aktuálně kontrolovaného*) agenta. Ve chvíli kdy se tyto souřadnice zjistí, agent na ně začne směrovat.

Následují dvě základní operace které provedou změnu směru daného agenta k průměrné lokaci ostatních agentů, kteří jsou v jeho okruhu.

1. Hlavní cyklus který prochází všechny agenty ve scéně a kontroluje vzdálenosti ostatních agentů vůči konkrétnímu agentovi. Pokud libovolný agent spadá do okruhu nějakého konkrétního, opět se přičítá k sumárnímu vektoru v_{sum} avšak ne rychlost, ale lokace blízkého agenta.
2. Následně se provede tentýž krok jako u zarovnání, čili sumární vektor v_{sum} se vydělí skalárem počtu splněných podmínek v kroku č. 1



Obrázek 4: Koheze - určení směru k průměrné lokaci okolních agentů (Zdroj: vlastní)

¹Koheze je fyzikální síla držící pohromadě atomy či molekuly též látky či tělesa (zejména kapalného a pevného tělesa), pozn. upraveno [16]

Kód v následujícím pseudokódu je obdobný jako kód zarovnání vyjma řádku s normalizováním sumárního vektoru, avšak v implementační části je tento kód složitější v řádku vracení výsledné hodnoty, kdy se ještě řeší fyzikální vlastnosti.

```
function Cohesion(boid paramBoid)
{
    Vector sum
    int count

    for (b in allBoids) {
        if (b != paramBoid){
            distance = distanceBetween(b, paramBoid)
            if (distance < COHESION_DIAMETER){
                sum = sum + b.location
                count = count + 1
            }
        }
    }

    if (count > 0){
        divideScalar(sum, count) // sum / count
        return physicalProperty(subtractVectors(sum, velocity))
    } else {
        return sum
    }
}
```

Výpis 3: Pseudokód pro kohezi (Zdroj: vlastní)

2.8 Aplikování tří pravidel

Ke kompletní simulaci dojde ve chvíli spojením třech výše zmíněných pravidel. Technicky vzato se jedná o sečtení všech třech výsledných vektorů z každé ze tří metod a rychlosti daného agenta který je aktuálně v iteraci. Nejdříve se sčítají rychlosti a poté se z této sumy spočítá výsledná pozice (Výpis pseudokódu č. 4).

Avšak pravidel je možné použít více, jak je popsáno v publikaci od Craiga Reynolda, Steering Behaviors For Autonomous Characters [15]. Tudíž nemusí být nutně tři fixní pravidla, ale můžeme k sčítání výsledných vektorů jednotlivých pravidel připsat i pravidla jiná. Kombinacemi těchto pravidel lze dosáhnout různého výsledného chování agentů.

Následující pseudokód ukazuje, jak lze tato pravidla aplikovat.

```
function AllBoidsToNewPosition()
{
    Vector v1, v2, v3
    Boid b

    for (b in allBoids) {
        v1 = Separation(b)
        v2 = Alignment(b)
        v3 = Cohesion(b)
        // v4 = anotherRule(b)

        b.velocity = b.velocity + v1 + v2 + v3
        b.position = b.position + b.velocity
    }
}
```

Výpis 4: Pseudokód pro aplikování třech pravidel (Zdroj: vlastní)

Tento algoritmus se ještě aplikuje do tzv. Flocking algoritmu, který všechny agenty ve scéně vytváří a koordinuje jejich průběh s vizualizací.

3 Praktická část

Následující sekce se zabývá použitými technologiemi a celkové implementaci aplikace, v niž byla použita metoda simulace na bázi umělé inteligence (*Artificial Intelligence*) zmíněné v kapitole 2.3 a boidův algoritmus zmíněný v kapitole 2.4.

3.1 Popis realizace demonstrační aplikace

Cílem demonstrační aplikace je zrealizování a následné otestování Boidova algoritmu. Tento algoritmus lze aplikovat mnoha způsoby avšak byly vybrány dvě varianty:

- Simulace úniku davu lidí z budovy podle nastavených únikových cest.
- Simulace hejna ryb s vyhýbáním se objektu.

Vetšinu parametrů lze měnit/kombinovat v konfiguračním souboru 3.8 (*lze i kombinovat jednotlivá tři pravidla postupným zapínáním/vypínáním*) a měnit mapu včetně únikových bodů lze v souboru map.txt případně si vytvořit vlastní soubor (viz kapitola 3.8 - spuštění scény), která je použitelná pouze pro první uvedený typ.

Při vypnutých pravidlech kromě separace se dav bude chovat stylem „*každý si jede kam chce*“.

3.2 Použité technologie

Výsledná aplikace je naprogramována v jazyce C++ a jako hlavní nástroj bylo použito Visual Studio verze 2017 [21], protože je nejpoužívanějším IDE pro vývoj [20].

Jako grafická knihovna byla použita OpenGL [22] s rozšířením pro aplikační okna GLFW (*verze 3.2.1*) [23] a realtime mechanismy GLEW (*verze 2.1.0*) [24].

K načítání objektů posloužila knihovna Assimp [26] (*verze 3.2*), která je vhodná kvůli velké škále podporovaných grafických formátů.

Pro matematické operace byla použita knihovna GLM [25] (*verze 0.9.8.5*) hlavně pro počítání s maticemi.

Pro zobrazování parametrů a proměnných byla použita GUI AntTweakBar knihovna (*verze 1.16*) [29] kvůli snadné implementaci a intuitivnímu rozhraní.

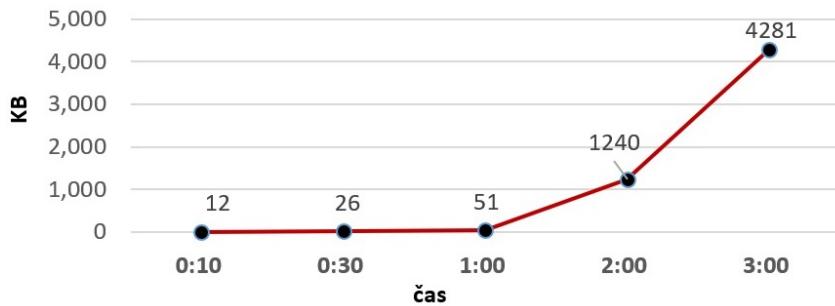
3.3 Problémy při implementaci

První problém nastal kvůli paměťovým nárokům. Důvod byl v obrovském nárustu paměti po zhruba dvou minutách běhu aplikace. Následující tabulka zobrazuje nárust v kB vůči času od posledního načtení objektu, tj. od počátku vizualizace scény.

Tabulka 1: Tabulka nárustu využití paměti RAM

10s	30s	1min	2min	5min
+12kB	+26kB	+51kB	+1240kB	+4281kB

Jak je patrné z hodnot, tak od druhé minuty nárust paměti stoupá takřka exponenciálně, jak můžeme vidět na obrázku č. 5. Důvod tak rychlého nárustu byl ten, že jak se postupně k sobě agenti přibližovali, začínalo se aplikovat více pravidel, které ještě neměly optimalizované operace s pamětíma. Tzn. nejdříve se začaly aplikovat pravidla s větším průměrem (*bud' koheze nebo zarovnání*) kružnice, ale když byli agenti blízko sebe, každému se vyhodnocovala všechna paměťově neoptimalizovaná pravidla. Po pěti minutách byl nárust tak obrovský, že hodnota FPS klesla na 10.



Obrázek 5: Graf závislosti paměťového růstu na čase

Pro veškeré výpočetní operace s vektory byla vytvořena třída *MyVector* z níž dědí třída *MyVector2D*, která je využita pro pohyb davu v budově. První zmíněná je určená pro hejno. Důvod využití vlastní třídy je přenositelnost a nezávislost algoritmu na platformě. Všechny důležité třídy využívají k vytvoření scény pouze nativní knihovny C++.

Příklad ošetření jedné části kódu:

```
this->tmpVector->set();
this->tmpVectorMem = this->tmpVector;
this->tmpVector = this->tmpVector->subTwoVector(location, Boidss->at(i))->
    location);
// ...operations with tmpVector
this->steer->addVector(this->tmpVector);
delete this->tmpVector;
this->tmpVector = this->tmpVectorMem;
```

Výpis 5: Ukázka eliminace nárustu paměti

Důležitý je třetí řádek, kdy se volá metoda *subTwoVector(...)*, která vrací výsledek odečtení dvou vektoru jako odkaz na nově vytvořenou paměť. Abychom zabránili nárustu paměti,

musíme tento výpočet uložit do dočasné paměti *tmpVector* se kterou provedeme patřičné operace a ve chvíli kdy jí už nebudeme potřebovat, tak ji smažeme.

Musíme však zachovat původní adresu *tmpVector*. Pro tento účel slouží proměnná *tmpVectorMem* která slouží jako prostředník pro uchování adresy. Pak tedy můžeme tuto původní adresu přiřadit zpět, v tomto případě poslední řádek kódu.

Pro větší úsporu paměti ve třídě *Boids*, jejíž instance je volána v každé smyčce, tak jednotlivá pravidla i metody pro mezivýpočty používali mezi sebou stejné adresy paměti.

Aby nedocházelo k neoprávněným přístupům a záměnou adres, tak byla využita pomocná proměnná do které se dočasně uložil výsledek pravidla a pak pomocí metody *set()* z třídy *MyVector* šlo bezpečně nastavil hodnoty vektoru bez jakéhokoliv úniku paměti. Tento způsob byl aplikován na všechna použitá pravidla, včetně vyhýbání se zdem.

Vůči 100 agentům ve scéně měla starší verze aplikace využití RAM 110 MB včetně použitých textur a objektů. Po těchto doposud zmíněných změnách bylo využití 91 MB.

Další vylepšení bylo lepší použití cyklů. Klasický zápis cyklu který byl použit před tímto typem optimalizace je následující:

```
for (int i = 0; i < this->models->size(); i++) {...}
```

Výpis 6: Použití klasického cyklu

Dle oficiálních referencí pro C++ [18] bylo zjištěno, že při každé iteraci cyklem se stále volá metoda *size()*, která tento výkon podstatně snižuje. Řešením je uložit tuto hodnotu do proměnné a tu pak následně porovnávat v podmínce. Dalším vylepšením je použití prefixové inkrementace proměnné *i*. Toto použití je mnohem účinnější než postfixová inkrementace z důvodu, že nepoužívá dočasné úložiště a vkládání potřebného kódu na pozadí [19]. Výsledný cyklus pak vypadá takto:

```
for (int i = 0, len = this->models->size(); i < len; ++i) {...}
```

Výpis 7: Použití optimalizovaného cyklu

Tyto změny důležitých cyklů v hlavní smyčce dokázaly přidat až 5 FPS.

Další problém nastal v načítání objektů. Původní verze nahrávání objektů do scény probíhala v obyčejné smyčce, takže stejné objekty (*v tomto případě objekty agentů*) se načítali vícekrát.

Řešení spočívalo v kopírovacím konstruktoru. Daný typ objektu byl načten pouze jednou a poté se v cyklu předávala adresa načteného objektu kopírovacímu konstruktoru, který tuto paměť nakopíroval pro svou instanci. Při načítání 100 agentů bylo celkově ušetřeno 40 sekund.

```
// loading object method in Flock class
Model *tmpBoidsModel = new Model(this->cfg->OBJ_BOID, this->cfg);
```

```

for (int i = 0; i < this->numberOfBoids; ++i) {
    if (tmpBoidsModel != NULL)
        this->boidsModel[i] = new Model(*tmpBoidsModel);
}
}

// copy constructor in Model class
Model(const Model& model)
:
    textures_loaded(model.textures_loaded),
    meshes(model.meshes),
    directory(model.directory),
    gammaCorrection(model.gammaCorrection),
    cfg(model.cfg)
{}

```

Výpis 8: Ukázka použití kopírovacího konstruktoru

3.4 Steering behaviors

Řízení chování [15] již zmíněné v teoretické části bylo použito pro reálnější podobu chování celého davu.

Jako první vlastnost byla použita metoda seek (*hledání*) primárně používané u Koheze při návratové hodnotě (viz kapitola 2.7 - kohezní síla), kde bylo upozorněno na složitější implementační část oproti zarovnání.

Hledání je určeno k tomu, aby agent směřoval k nějakému cíli, tak že upravuje aktuální rychlosť (*velocity*). V případě koheze je cíl průměrná lokace okolních agentů. Výsledek této operace je seek vektor.

Následující kód ukazuje jak se seek vektor vypočítá.

```

this->tmpVectorMem = desired->subTwoVector(v, this->location);
this->desired->set(this->tmpVectorMem->vec.x, this->tmpVectorMem->vec.y, this->
    tmpVectorMem->vec.z);
delete this->tmpVectorMem;

this->desired->normalize();
this->desired->mulScalar(maxSpeed);

this->seekResult->set();
this->tmpVectorMem = seekResult->subTwoVector(this->desired, this->velocity);

```

```
this->seekResult->set(this->tmpVectorMem->vec.x, this->tmpVectorMem->vec.y,  
    this->tmpVectorMem->vec.z);  
delete this->tmpVectorMem;  
  
seekResult->limit(maxForce);
```

Výpis 9: Vypočtení seek vektoru

Nejprve se spočte lokace na kterou má agent směřovat (*vektor desired*), a nastavíme maximální zrychlení aby se na místo dostavil co nejdříve.

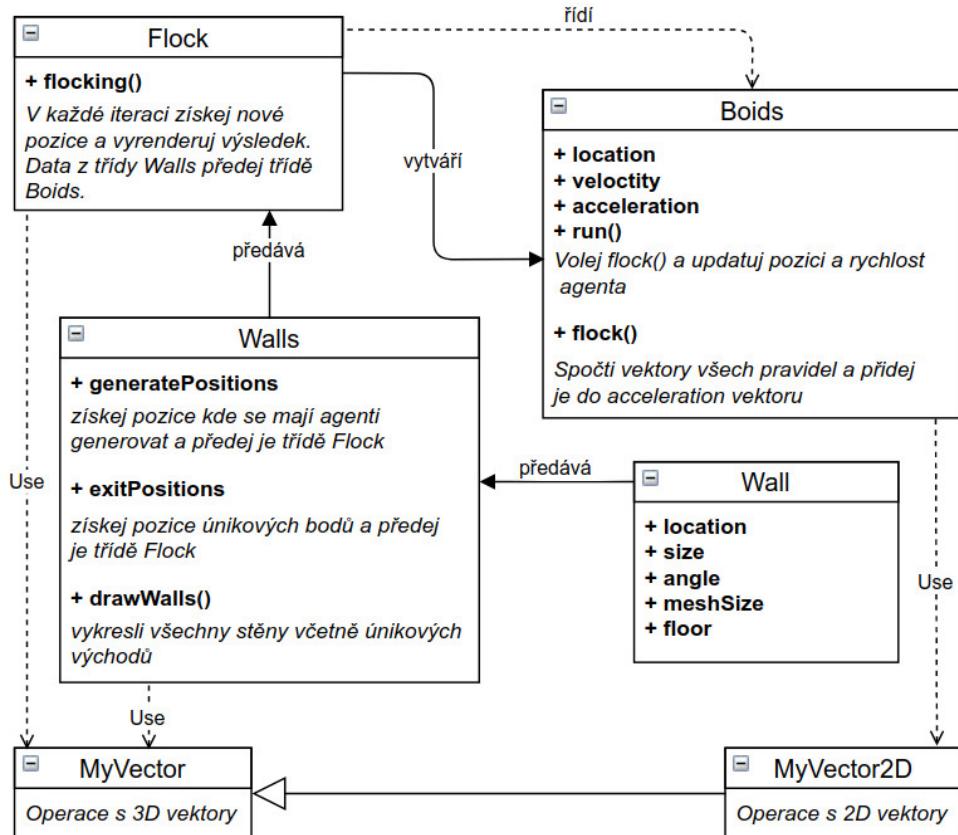
Následuje vypočtení samotného seek vektoru, který je roven rozdílu mezi požadovanou a aktuální rychlostí. V poslední řadě pak musíme vektor limitovat konstantou tření (*maxForce*).

Další vlastností která byla použita je následování bodů (*Path Following*). V této práci slouží jako následování kontrolních bodů v budově.

Po určení tohoto jednoho či více bodů na ně agent začně postupně směřovat. Při následování bodů se používá funkce pro vypočtení seek vektoru, který je poté aplikován jako další pravidlo, tzn. k výsledné rychlosti se přičte další vektor (viz kapitola 2.8 - aplikování tří pravidel).

3.5 Třídní diagram

Pro provoz simulace a chování davu slouží tři základní a tři pomocné třídy.



Obrázek 6: Diagram řídicích tříd

Hlavní třídou je třída **Flock**, která koordinuje komunikaci mezi třídou **Walls** a **Boids**. Zároveň se stará o veškeré vytváření agentů, kterým přiřažuje jednotlivé objekty ve scéně, jenž jsou vizualizovány uživateli. Také v každé iteraci volá simulační mechanismy pomocí metody **run()**.

Třída **Walls** vytváří zdi podle předaných dat z mapy a udržuje informace o generačních pozicích a pozicích únikových bodů. Díky tomu může třída **Flock** vygenerovat agenty do patřičného patra a pozice. Zároveň jsou tato data sdílena s třídou **Boids**, která může kontrolovat kolize se zdmi a počítat pozice únikových bodů na které má agent směřovat.

Jak již bylo zmíněno v kapitole 3.3, tak pro operace s vektory byla použita vlastní třída **MyVector** ze které dědí třída **MyVector2D** z důvodu přenositelnosti a nezávislosti algoritmu na platformě.

3.6 Výsledné aplikování pravidel

Všechna pravidla jsou aplikována jako součet výsledků jednotlivých pravidel (*viz kapitola 2.8 - aplikování tří pravidel*). V této práci k sečtení a vytvoření výsledného vektoru slouží metoda

`applyForce(...)`), která přidá do výsledného vektoru *acceleration* další vektor (*výsledek jednoho z pravidel*). Tento vektor je pak aplikován v metodě `update()`, která nastaví agentovi novou pozici a rychlost.

```
void Boids::update() {
    acceleration->mulScalar((float)(.4));
    velocity->addVector(acceleration);
    velocity->limit(maxSpeed);
    location->addVector(velocity);
    acceleration->mulScalar(0);
}
```

Výpis 10: Nastavení výsledné pozice agenta

Po vynásobení konstantou 0.4, která slouží pro nastavení citlivosti (*může a nemusí být použita, zde je nastavena po experimentech jako ideální*) následuje přidání, limitování vektoru a vypočtení výsledné pozice. Na konci musíme vektor *acceleration* vynulovat pro použití v další iteraci.

3.7 Konfigurační soubor

Konfigurační soubor je rozdělen do čtyř sekcí. První sekce se zaobírá nastavením základních parametrů jako pozice kamery, šířka či výška okna až po citlivost myši a nastavení scény (*2D nebo 3D*).

Druhá sekce je zaměřena na únik z budovy, kdy se nastavují důležité parametry jako FLOOR_TIME_DURATION, který zajišťuje čas, po který daný agent sestupuje z patra, nebo PATH_TO_FIND_RADIUS zajišťující velikost kruhu kolem únikového bodu. Pokud tedy bude kruh větší, agent začne dříve směrovat k druhému bodu.

Syntaxe souboru je následující:

JMENO_PROMENNE = HODNOTA

Dále následuje celkové chování agentů jako rychlost, tření, počet agentů a počet objektů kterým se budou agenti vyhýbat. Toto nastavení má vliv na celkovou scéně nejvíce.

Třetí sekce nastavuje rychlosť agentů, jejich velikost, velikost generačního pole ve kterém se agenti generují a hlavně důležitá pravidla jako velikost kruhu ve kterém se nachází agent se kterým bude iterovat či nastavení konstanty BOID_DESIRED_SEPARATION, která indikuje minimální vzdálenost agentů, kteří se přiblíží k sobě.

Čtvrtá sekce je zaměřena k načítání objektů do scény. Cesta, která se udává k objektu je relativní a tudíž objekt musí být umístěn v adresáři *Models*, který se nachází na stejném adresářovém úrovni jako výsledný spustitelný soubor *Bachelor.exe*

Komentáře jsou řešeny jako ve výpisu níže:

```
#####
# Section 2.
# Building escape
#####
```

Výpis 11: Ukázka komentáře v konfiguračním souboru

K označení řádku s komentářem indikujeme pomocí křížku, který aplikace ignoruje.

3.8 Spuštění scény

Aplikaci lze spustit tak, že spustitelný soubor *Boid.exe* přijímá jako první parametr jméno konfiguračního souboru a jako druhý parametr jméno textového souboru, ve kterém je uložené schéma budovy.

Pokud není uveden žádný z parametrů, aplikace načte konfiguraci ze souboru *config.cfg* a schéma budovy ze souboru *map.txt*.

Pro dva typy scén (*únik z budovy, simulace hejna ryb*) jsou připraveny dva konfigurační soubory: **configEscape.cfg** a **configFish.cfg**, které může uživatel modifikovat.

K ukázce různých scén slouží soubory s příponou **.bat*, které spouštějí aplikaci s předem definovanými parametry. K výslednému programu jsou připraveny scény:

- **boid_escape_variant_1.bat** - varianta úniku z vícepatrové budovy, využívá schéma ze souboru *map_variant_escape_1.txt*
- **boid_escape_variant_2.bat** - varianta úniku z vícepatrové budovy, využívá schéma ze souboru *map_variant_escape_2.txt*
- **boid_labyrinth_variant_1.bat** - varianta úniku z labyrintu, využívá schéma ze souboru *map_variant_labyrinth_1.txt*
- **boid_labyrinth_variant_2.bat** - varianta úniku z labyrintu, využívá schéma ze souboru *map_variant_labyrinth_2.txt*
- **boid_fishes_variant_1.bat** - varianta hejna ryb s kolizním objektem kvádru (*cube*)
- **boid_fishes_variant_2.bat** - varianta hejna ryb s kolizním objektem sféry (*sphere*)

Informace, které byly zaznamenány pro úniky z bodov různých typů, jsou zmíněné v kapitole 3.10 - testování úniku z budovy.

3.9 Únik z budovy

První část využití Boidova algoritmu je simulace úniku lidí z budovy podle nastavených únikových cest. Funkčně je tato část určena k tomu, aby po skončení simulace mohl uživatel zjistit, kolik

času by zabralo evakuovat lidi z předem definovaného plánu budovy. V případě časově náročné evakuace může uživatel upravit plán budovy a únikové cesty tak, aby budova splňovala dané normy.

Plán budovy můžeme upravovat v souboru *map.txt*, případně využití vlastního schématu (viz kapitola 3.8 - spuštění scény) včetně definování dalších pater a únikových cest. Jelikož se jedná o textový soubor, tak každý znak reprezentuje jinou funkci. Popis funkčních znaků můžeme vidět níže:

- **G** generační místo pro dav, který lze použít vícekrát v každém patře či místnostech
- - reprezentuje vertikální stěnu
- | reprezentuje horizontální stěnu
- + reprezentuje spojovací stěnu jako kombinace horizontální a vertikální stěny
- * blok stěny jednoho bodu, proto ji lze použít jako reprezentaci sloupu v patře
- = reprezentuje začátek schématu nového patra
- / ukončovací znak schématu patra
- **A-Z, a-z** únikové body podle který se dav řídí (*mimo G jako generační bod a F jako ukončovací bod zřetězeného řízení davu, viz níže*)
- **1-9, F** zřetězené řízení davu použité jako únik z labyrintu pomocí 10 únikových bodů

Pro únik v jedné části budovy jsou požity dva znaky (*např.: velké písmeno A a malé a*). Nejdříve dav směruje k menšímu znaku a poté k většímu, který indikuje finální bod. V každém patře může být těchto únikových bodů libovolný počet, kdy pokud v jednom patře končíme např.: znakem D, pak se v dalším patře začíná od znaku E. Jenotlivý agent určuje ke kterému bodu jít pomocí vzdálenosti od něj (*tj. k nejbližšímu*).

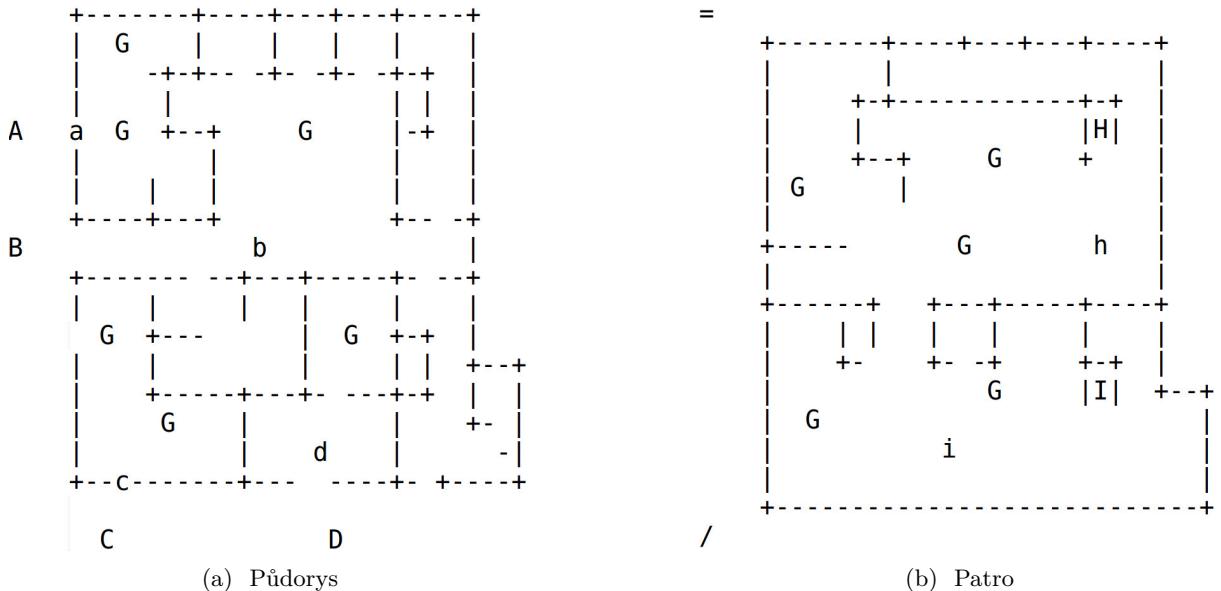
Druhým typem úniku je únik z labyrintu, tedy složitějších prostor budovy kdy je potřeba více než dva únikové body. Dav začne směrovat k nejbližšímu nejmenšímu číslu a pokud tohoto čísla dosáhne, tak poté začne směrovat k dalšímu o 1 větší dokud neskončí u písmene F. Tento způsob lze aplikovat pouze u vykreslení jednoho patra.

Při spuštění aplikace, která vykreslí definované schéma lze scénu ovládat pomocí následujících kláves:

- **1-9** klávesou lze přepínat mezi pohledy jednotlivých pater, přičemž se neaktuální patro přepne do režimu průhlednosti
- **E** přepnutí celé budovy do řežimu průhlednosti se zvýrazněním davů

- **Q** opětovné vykreslení celé budovy bez režimu průhlednosti pater
 - **F** přepnutí do režimu celé obrazovky, opětovným stiskem lze aplikaci přepnout zpět do režimu okna
 - **Mezerník** spuštění simulace, opětovným stiskem lze simulaci ukončit

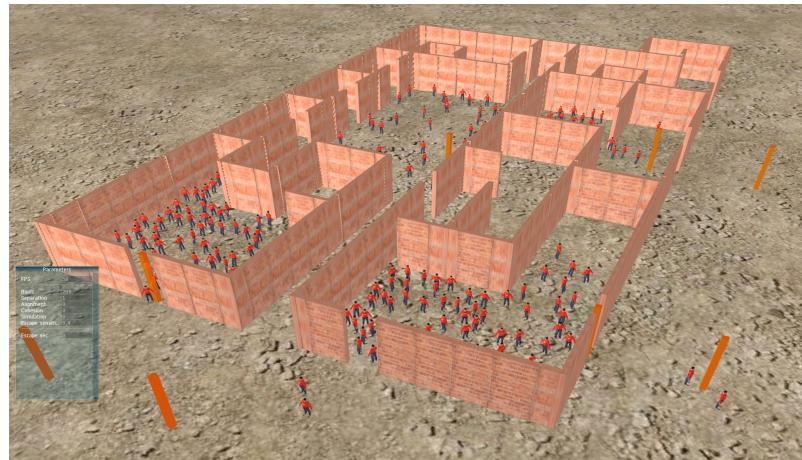
Výslednou scénu lze přepnout do fullscreen režimu pomocí klávesy F případně zpět.



Obrázek 7: Ukázka nákresu budovy v textovém souboru

Při zapnutí simulace jsou agenti v půdorysu okamžitě vedení směrem ven mimo budovu, avšak agenti v dalších patrech musí využít jiné evakuační cesty aby se dostali do spodního patra odkud můžou vyjít ven². Proto pokud se agent v patře dostane k finálnímu bodu úniku (*velké písmeno*), začne "cestovat" do přízemí budovy. Sestup však zabere nějaký čas, jednak kvůli zatížení únikových cest tak samotného sestupu. Souvisí s tím také to, že agentovi v posledním patře bude sestup trvat déle než agentovi v prvním patře. Hodnotu míry opuštění budovy (*konkrétněji: evakuační výtah je rychlejší než evakuační schodiště*) lze měnit v konfiguračním souboru pomocí konstanty FLOOR_TIME_DURATION.

²Agenty v patře lze také směrovat rovnou ven mimo budovu, tj. forma venkovního evakuačního schodiště. Dole v přízemí mimo budovu však musí být definován další evakuační bod (pod "schodištěm"), jinak začnou agenti směrovat k nejbližšímu bodu, který by jinak byl v budově.



Obrázek 8: Ukázka vyrenderovaného půdorysu budovy

Pro následování bodů byla použita metoda `arriveTo(...)` která využívá metodu `seek(...)` (*viz kapitola 3.4 - steering behaviors*). Abychom správně určili, který agent má kam směřovat, musíme metodě

`arriveTo(...)` předat správný vektor, který získáme metodou `getArriveVector(...)`. Ta vrátí výsledný vektor který je předán `seek()` metodě. Výsledek je pak nový vektor, který se aplikuje jako nové pravidlo (*viz kapitola 2.8 - aplikování tří pravidel*) a tím zajistíme že agent bude směřovat na nejbližší únikový bod a následně na finální bod.

```
// getting minIndex from exitPoints...
if ((walls->exitPositions.at(minIndex)->vec.y / walls->floorDiferencial) ==
    this->floor) {

    if (location->distance(walls->exitPositions.at(minIndex)) < cfg->
        PATH_TO_FIND_RADIUS && !this->incrementedOnce) {
        ++minIndex;
        this->incrementedOnce = true;
    }
    // code...
    return walls->exitPositions.at(minIndex);
}
```

Výpis 12: Princip metody `getArriveVector(...)`

V první části kódu, zde nezmíněné kvůli délce, je získání *minIndex* hodnoty ve které je uložen index z kontejneru *map*, odkud začínají souřadnice únikových bodů ve stejném patře jako je aktuálně kontrolovaný agent. Pokud tedy v půdorysu bude 8 únikových bodů, jak je zobrazeno na obrázku č. 7, tj. 4 finální únikové cesty, pak *minIndex* v následujícím patře bude 9. Je třeba upozornit, že hodnota *minIndex* vždy odkazuje na index menšího písmena.

Následuje podmínka jenž kontroluje, zda-li se agent nachází ve stejém patře jako únikový bod (*souřadnice Y únikového bodu vydělená výškou jednoho patra - konstanta třídy Walls*) a následná kontrola, zda se agent dostal do blízkosti okruhu únikového bodu. Pokud ano, může agent pokračovat k finálnímu bodu tak, že hodnotu *minIndex* zvýšíme o 1.

Mimo další operace které se nachází mezi návratovou hodnotou a předchozími podmínkami, je nakonec vrácen vektor s indexem *minIndex*.

Útěk ze složitějších prostor pomocí čísel (*útěk z labyrintu*) funguje na stejné bázi s tím rozdílem, že se hodnota inkrementuje do té doby dokud agent nenarazí na písmeno F, čili koncový bod.

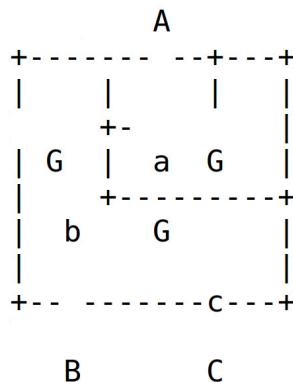
3.9.1 Problémy při úniku z budovy

Agenti se můžou občas dostat do problémů zejména díky špatnému návrhu budovy, nebo také kvůli nedokonalosti Boidova algoritmu.

Například pokud při úniku z budovy zapneme všechna tři pravidla (*separace, zarovnání, koheze*), může nastat efekt procházení zdí. Tento problém vzniká kvůli nastaveným konstantám citlivosti u každého pravidla. Tyto konstanty byly nastaveny po řadě experimentů jako nevhodnější a jsou neměnné (*kromě konstanty citlivosti úniku ESCAPE_SENSITIVITY, kterou lze měnit v konfiguračním souboru*). Pokud je konstanta příliš nízká, dané pravidlo by zaniklo mezi ostatními. Pokud příliš vysoká, tak by dané pravidlo překrylo ostatní. U konstanty citlivosti úniku je však občas potřebné přenastavení. Částečné řešení spočívá jen v použití separace. Dav se tedy chová jako dav skutečných lidí (*každý si jede kam chce*).

Další problém může nastat při generování agentů, kdy agenti vyberou jako cestu k nejkratšímu bodu tu variantu, že se bod nachází za zdí či v jiné místnosti (viz obrázek č. 9). S tím souvisí možnost zaseknutí o zed, kdy má agent před sebou překážku za kterou je cílový bod a např.: dva bloky od něj je východ. Má však nastavenou cestu přímo na bod (*přes zed*).

S generováním souvisí ještě jedna věc, a sice při generování většího množství agentů na malém prostoru se může některý z nich „zabořit“ do zdi a už se z ní nemůže dostat.



Obrázek 9: Problém uváznutí při špatném určení únikových bodů

Dále můžou nastat problémy při úniku ze složitějších prostor (*labyrinth*), kde uživatel nastaví číslo úniku u hrany stěny o kterou se může agent zaseknout. V případě většího davu (*okolo 300 lidí*) a úzkých prostor můžou být ti, kteří splnili podmínku a dostali se k bodu, “vytlačení” ostatními přes zed a tím se dostat do situace že následující bod se nachází za zdí.

V neposlední řadě s výše zmíněným problém může nastat situace, že je agent úspěšně “vytlačen”, ale vrací se zpět na předchozí bod, který kvůli předešlé vysoké koncentraci agentů němel označen jako splněný. Částečné řešení spočívá v nastavení konstanty velikosti kruhu kolem únikového bodu pomocí konstanty PATH_TO_FIND_RADIUS.

3.10 Testování úniku z budovy

Testování proběhlo na PC s parametry: CPU Intel i5-2500 @ 3.3GHz, grafická karta Radeon R9 280X Dual-X, RAM 12GB. Hodnoty se můžou na jiných PC lišit.

Všechny části testů byly provedeny na schématech budov s parcelami první budovy 30x17 bloků a druhé budovy 30x21, kdy jeden blok odpovídá jednomu znaku v textovém souboru, ve kterém je definováno schéma budovy, a na 1x1 blok se vejde průměrně 5 agentů bez větších kolizí.

Tabulka zobrazuje jaké byly hodnoty FPS při daném počtu agentů, počtu patr a složitosti budovy (*počet zdí*). Hodnoty FPS jsou odečteny před a během simulace. Poslední hodnotou je čas opuštění budovy.

První část testů proběhla nad schématem budovy, které je dostupné ve výsledném adresáři v souboru s názvem *map_variant_escape_1.txt*.

Testy proběhly při nastavených hodnotách:

- počtu agentů 200
- rychlostí agentů 3
- maximální hodnotou tření 0.1
- míra opuštění jednoho patra na 8 sekund

Tabulka 2: Tabulka naměřených hodnot - schéma 1

Pater	Únikových bodů	Počet zdí	FPS před	FPS během	Čas [s]
0	8	193	56	55	61.6
1	10	343	57	55	121.7
2	14	513	52	45	153.1
3	18	683	46	38	162.4
4	22	853	41	39	173.3
5	24	1023	40	36	246.5

Výsledné hodnoty jsou závislé zejména na dvou konstantách a sice ESCAPE_SENSITIVITY, která udává přímočarost následování dalšího bodu a PATH_TO_FIND_RADIUS, která indikuje velikost radiusu do kterého pokud se agent přiblíží, přesměruje se na další bod.

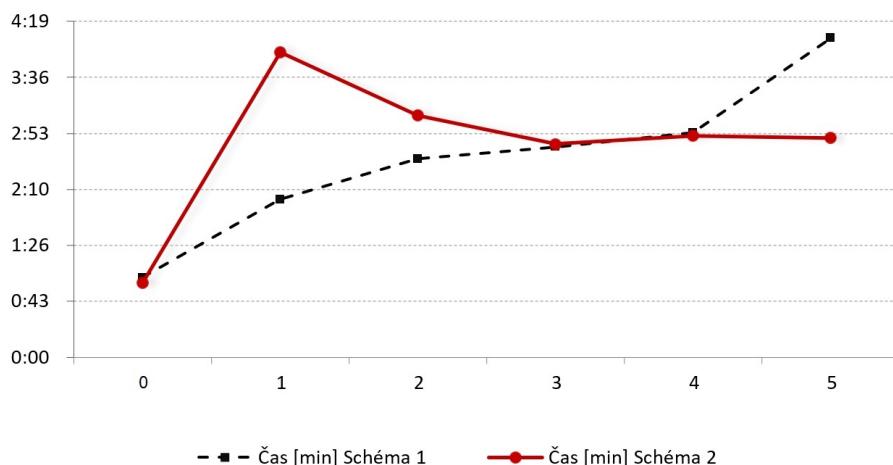
Další testy proběhly nad schématem uloženého v souboru *map_variant_escape_2.txt*.

Tabulka 3: Tabulka naměřených hodnot - schéma 2

Pater	Únikových bodů	Počet zdí	FPS před	FPS během	Čas [s]
0	8	172	102	99	57.5
1	10	362	86	84	235.2
2	16	547	57	57	186.6
3	22	732	46	38	164.5
4	28	917	49	49	170.9
5	34	1102	44	43	169.1

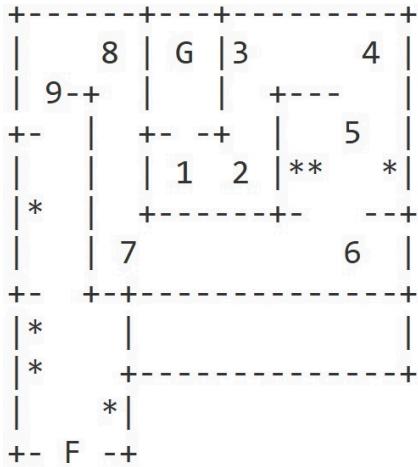
Kompletní plány budov lze nalézt v souborech: *map_completed_variant_1.txt* a *map_completed_variant_1.txt*.

Jak je patrné z grafu na obrázku č. 10, tak v prvním patře druhého schématu byl únik z budovy časově náročnější z důvodu použití jednoho únikového bodu. Dále však bylo toto schéma, co se času úniku týče lepší, neboť schéma obsahuje jednodušší prostory budovy a lepší návrhy únikových bodů.



Obrázek 10: Graf závislosti času úniku na počtu pater

Další části testování proběhly nad složitějšími prostory budovy o parcele 23x12. Jak již bylo zmíněno v kapitole 3.9 - únik z budovy, tak únik z labyrintu (*složitějších prostor*) funguje na principu umisťování čísel do mapy, kdy agenti směřují od nejmenší po největší číslo a následně k finálnímu bodu F.



Obrázek 11: Ukázka nákresu labyrintu

Testování času úniku proběhlo na schématu, které můžeme vidět na obrázku č. 11, při nastavené rychlosti na 1.5 a tření na 0.1. Srovnání výsledků s jinými patry však nelze zrealizovat, protože u této metody lze použít pouze jedno patro.

Tabulka 4: Tabulka naměřených hodnot - labyrinth

Počet agentů	FPS před	FPS během	Čas [s]
20	280	279	46.4
30	275	265	51.8
40	267	261	63.2
50	260	245	74.6

U této metody je také potřeba opomenout nastavení konstanty PATH_TO_FIND_RADIUS. Pokud je radius větší, agent splní podmínu dosažení bodu když bude dál od něj. Toto nastavení je vhodné zejména pro větší prostory budov z důvodu eliminace přímočarého pohybu k únikovým bodům.

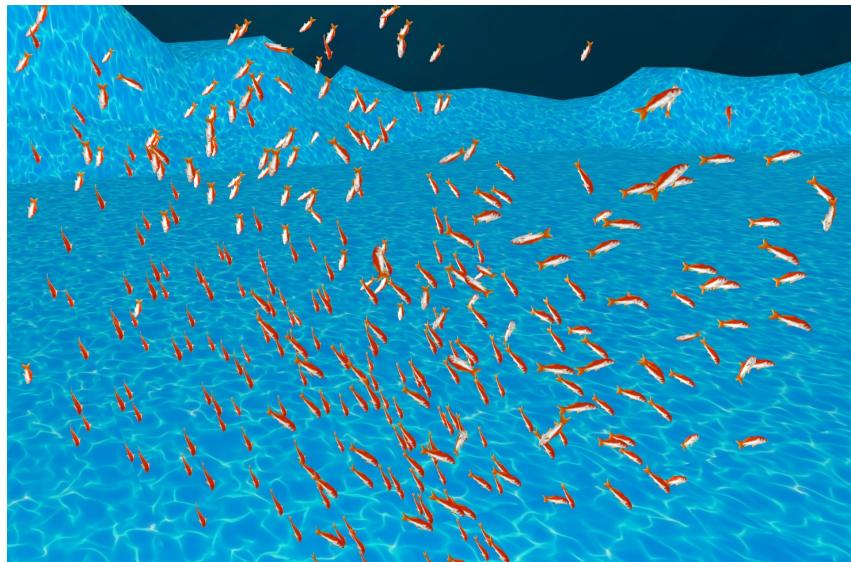
Ve výsledném adresáři jsou připraveny soubory *boid_labyrinth_variant_1.bat* a *boid_labyrinth_variant_2.bat*, kdy první zmíněný spouští scénu s počtem agentů 20 a druhý s počtem 50. Při spuštění druhé varianty lze spatřit efekt zaboření do zdi, případně vytlačení agenta ostatními, díky velkému počtu agentů na malém prostoru, zmíněné v kapitole 3.9.1 - problémy při úniku z budovy.

3.11 Simulace hejna ryb

Druhý typ využití Boidova algoritmu je simulace hejna. V tomto případě je použita simulace ryb, ale při změně načítaného objektu v konfiguračním souboru můžeme vytvořit simulaci hejna čehokoliv. Pro změnu scény je třeba přepsat několik parametrů, zejména pak SCENE_TYPE na 3D. U výsledné aplikace je připraven obsah souboru *configFish.cfg* pro simulaci ryb. Pro výslednou scénu byly připraveny jednotlivé varianty (viz kapitola 3.8 - spuštění scény)

Algoritmus funguje stejně jako při úniku z budovy, výjma využití třídy *MyVector2D*, která je nahrazena výpočty ve 3D třídou předka *MyVector*.

V konfiguračním souboru můžeme nastavit počet objektů, kterým se agenti budou vyhýbat. V případě, že se agent dostane do blízkosti objektu, pak vyhýbání funguje obdobně jako separace, akorát normalizování vektoru a vydělení vektoru počtem okolních agentů je nahrazeno vynásobením skalárem. Můžeme také nastavit útvar, ve kterém se budou agenti pohybovat. Na výběr je sféra (*sphere*), nebo kvádr (*cube*) nastavitelné pomocí konstanty BOID_SPACE_TYPE.



Obrázek 12: Ukázka hejna ryb

U této simulace je průměrná hodnota FPS 65 s počtem agentů 400 při parametrech PC zmíněných v kapitole 3.10 - testování úniku z budovy.

3.11.1 Problémy při simulaci hejna

U této simulace nastávají problémy "cukání" pokud agent začne vyhodnocovat, že se přiblížil k okraji mapy. Důvod tohoto efektu je aplikování kolize s ostatními agenty a zároveň kolize s okrajem mapy. Tento problém byl zmírněn při nastavení různých citlivostí jednotlivých pravidel.

Pro menší množství agentů (např.: 20 agentů na větší ploše), nebyl tento efekt tak viditelný, jako při vizualizaci několika set agentů.

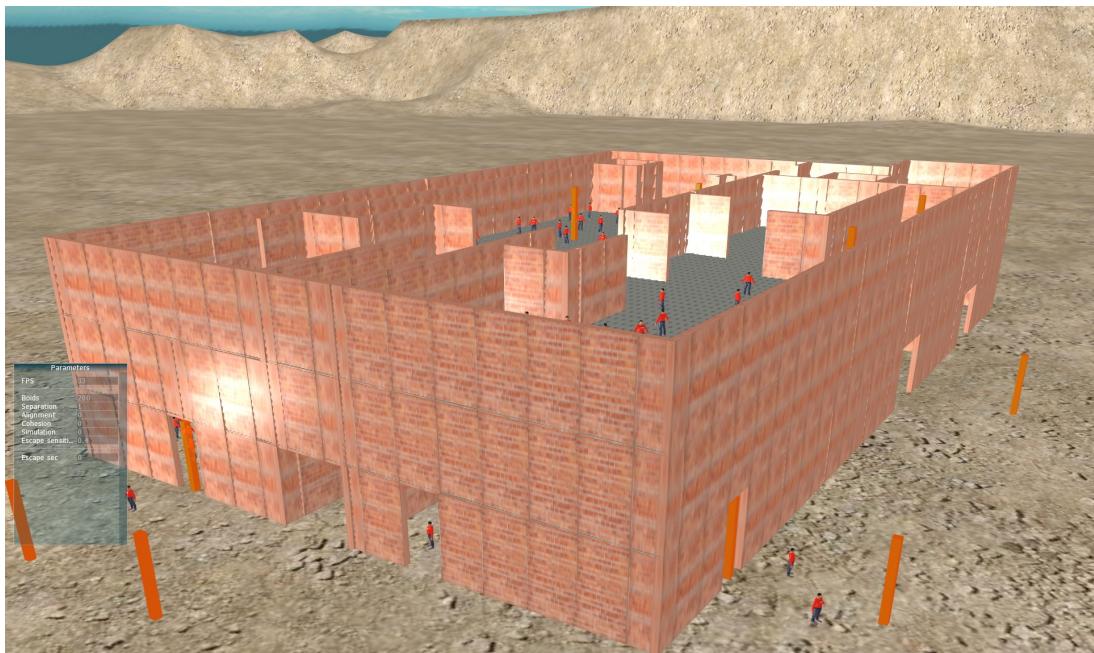
Nějvětší problémy nastávaly při nastavování konstant jako např: konstanty pro jednotlivá pravidla, kdy při špatném nastavení pohybu agentů chování neodpovídalo realitě.

4 Zhodnocení

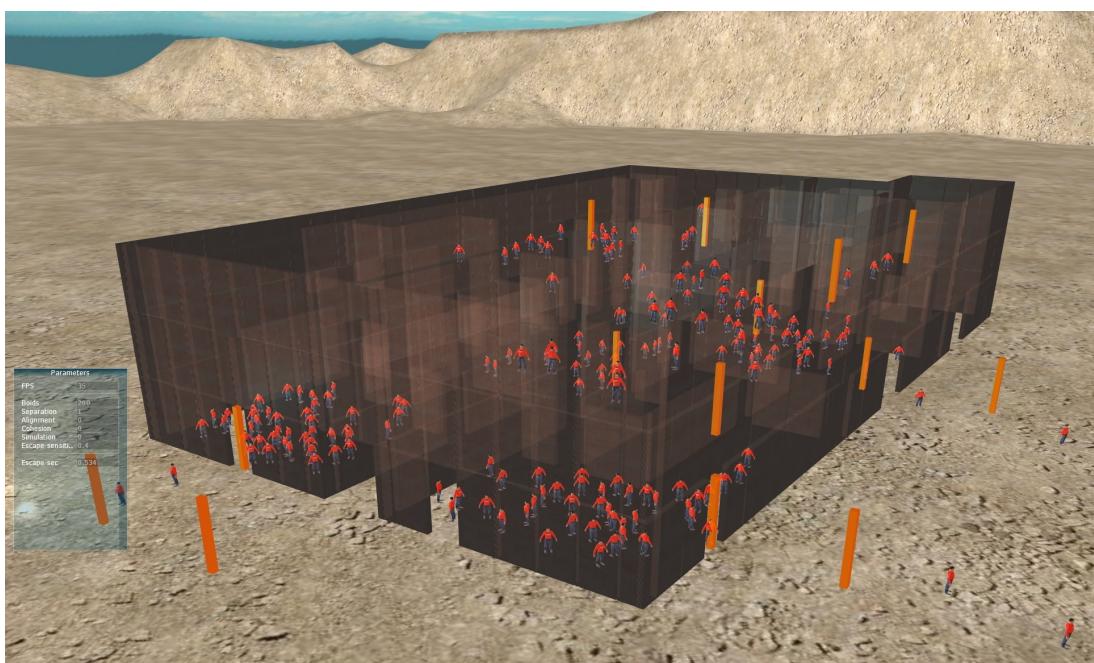
I když byly použity vylepšené techniky Boidova algoritmu (seek, path following) a vyladěná všechna tři pravidla (separace, zarovnání, koheze), tak stále tento algoritmus není dokonalý (kapitoly 3.9.1 - problémy při úniku z budovy, 3.11.1 - problémy při simulaci hejna). Jako hlavní problém byl vyhodnocen stav, kdy se v dané scéně musí nastavit důležité konstaty (viz níže) citlivosti tak, aby pohyb agentů byl reálný.

Největší problémy nastavování konstant citlivostí, konkrétněji pěti hlavních (citlivosti jednotlivých tří pravidel, vyhýbání se okraje kolizních zdí a celková citlivost v metodě `update()`, viz kapitola 3.6 - výsledné aplikování pravidel), nastávaly u scény se simulací hejna ryb, kdy při špatném nastavení konstant byl pohyb přímočarý, tzn: změna směrového vektoru byla provedena okamžitě a neplynule.

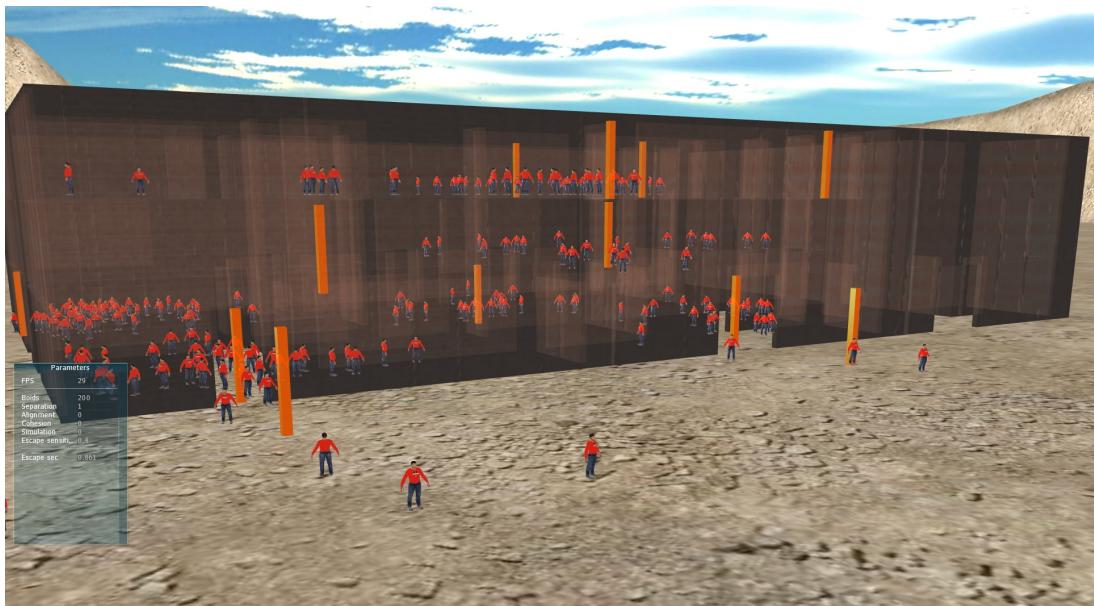
I přes tyto nedostatky lze Boidův algoritmus efektivně aplikovat jako simulaci jakéhokoliv hejna (v této práci hejna ryb viditelné na obrázku č. 16), což je také nejvyužitelnější způsob.



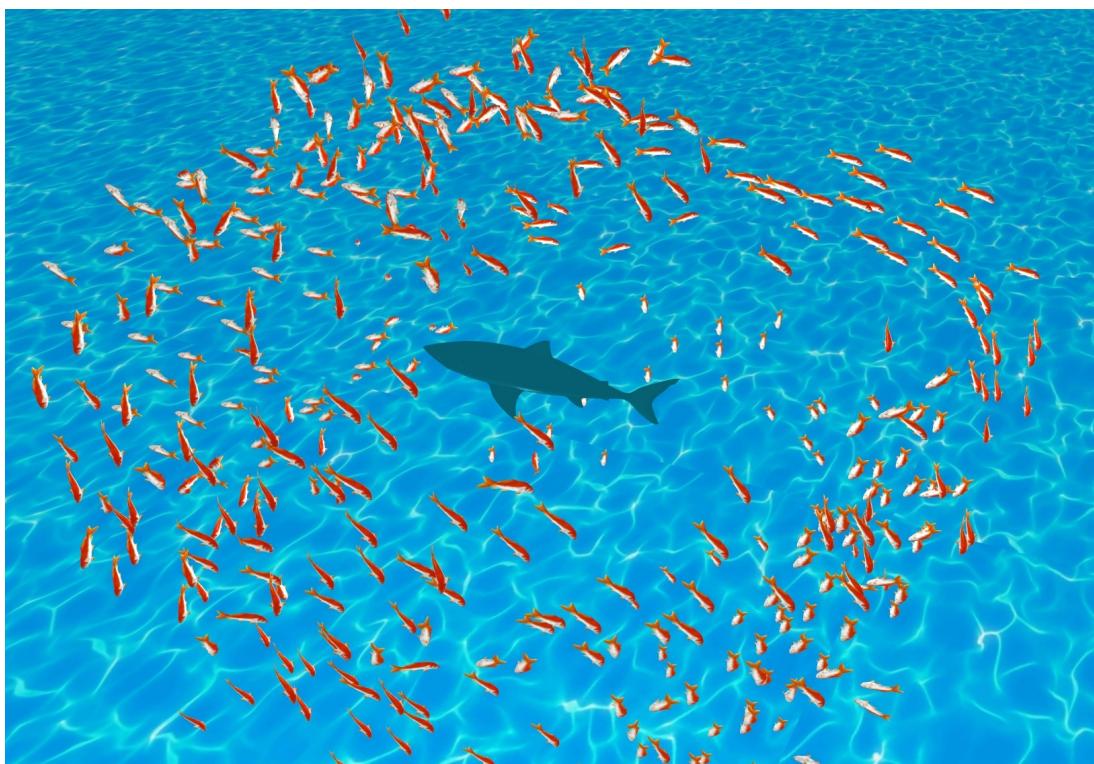
Obrázek 13: Vyrenderovaná dvoupatrová budova



Obrázek 14: Budova přepnuta do průhledného režimu se zviditelněním agentů a únikových bodů



Obrázek 15: Průhledný režim budovy z boční perspektivy



Obrázek 16: Vyrenderované hejno ryb s vyhnutím se objektu

5 Závěr

Výsledkem této práce je aplikace, která má za úkol zrealizovat simulační algoritmus davu. Teoreticky byly popsány dvě varianty simulace (částicová simulace a simulace na bázi AI - umělé inteligence), z níž pro realizaci byla vybrána simulace na bázi AI a pro implementaci byl vybrán Boidův algoritmus. V první možnosti využití v případě evakuace budovy se Boidův algoritmus a jeho vylepšení neosvědčil na plno, avšak pro simulaci hejna byl vyhodnocen jako plně dostačující a využitelný v mnoha odvětví.

Aplikace nabízí dvě možnosti využití. První, jak již bylo zmíněno, je simulace úniku lidí z budovy a druhá je využití obecného Boidova algoritmu jakožto simulace hejna (v této práci hejno ryb). U první varianty je možnost nadefinovat si vlastní schéma budovy včetně únikových východů a u druhé je možnost nadefinovat počet objektů, jimiž se bude hejno vyhýbat a typ kolizního objektu, ve kterém se hejno pohybuje, a sice sféra (sphere) nebo krychle (cube). Uživatel si tak může vyzkoušet na obou příkladech výhody a nevýhody Boidova algoritmu a také vliv jednotlivých parametrů na výslednou scénu.

Reference

- [1] Oasys, *Crowd Simulation*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <http://www.oasys-software.com/products/engineering/massmotion.html>
- [2] FAAC Military, , [online]. 2018 [cit. 2018-2-1]
Dostupné z: <https://www.faac.com/military/>
- [3] Journal of the royal society interface, *Crowd behaviour during high-stress evacuations in an immersive virtual environment*, [online]. 2016 [cit. 2018-2-1]
Dostupné z: <http://rsif.royalsocietypublishing.org/content/13/122/20160414>
- [4] Boids, *Background and Update by Craig Reynolds*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <https://www.red3d.com/cwr/boids/>
- [5] Unreal Engine 4 simulace ryb, [online]. 2016 [cit. 2018-2-1]
Dostupné z: <http://blog.csdn.net/nosix/article/details/52859160>
- [6] Crytek, [online]. 2018 [cit. 2018-4-19]
Dostupné z: <http://www.crytek.com/>
- [7] The nature of code, *Chapter 6. Autonomous Agents*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <http://natureofcode.com/book/chapter-6-autonomous-agents/>
- [8] Boids Pseudocode, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <http://www.kfish.org/boids/pseudocode.html>
- [9] Simulace, *Glosář Aldebaran*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <http://www.aldebaran.cz/glossary/print.php?id=1299>
- [10] Earthlight, *Spacewalk*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <http://www.earthlightvr.com/spacewalk/>
- [11] Techopedia, *Iteration*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <https://www.techopedia.com/definition/3821/iteration>
- [12] ACM, *Association for Computing Machinery*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <https://www.acm.org/about-acm/what-is-acm>
- [13] SIGGRAPH, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <https://www.siggraph.org/about/what-is-acm-siggraph>
- [14] Simulace davu, Mgr. Jan Stria,
Matematicko-fyzikální fakulta (MFF), [online]. 2018 [cit. 2018-2-1]
Dostupné z: <https://is.cuni.cz/webapps/zzp/detail/60388/>

- [15] Steering Behaviors For Autonomous Characters
Craig W. Reynolds, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <http://www.red3d.com/cwr/steer/gdc99/>
- [16] Kohezní síla, *Glosář Aldebaran*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: <http://www.aldebaran.cz/glossary/print.php?id=1894>
- [17] Techopedia, *IDE*, [online]. 2018 [cit. 2018-2-4]
Dostupné z:
<https://www.techopedia.com/definition/26860/integrated-development-environment-ide>
- [18] CppReference.com, , [online]. 2018 [cit. 2018-2-4]
Dostupné z: <http://en.cppreference.com/w/>
- [19] CppReference.com, *Increment/decrement operators*, [online]. 2018 [cit. 2018-2-4]
Dostupné z: http://en.cppreference.com/w/cpp/language/operator_incdec
- [20] Top IDE Index, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <https://pyp1.github.io/IDE.html>
- [21] Visual Studio IDE, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <https://www.visualstudio.com>
- [22] OpenGL, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <https://www.opengl.org/>
- [23] GLFW, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <http://www.glfw.org/>
- [24] GLEW, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <http://glew.sourceforge.net/>
- [25] GLM, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <https://glm.g-truc.net/0.9.8/index.html>
- [26] Assimp, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <http://assimp.sourceforge.net/>
- [27] Stb image, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <https://github.com/nothings/stb>
- [28] OpenCV, [online]. 2018 [cit. 2018-2-4]
Dostupné z: <https://opencv.org/>
- [29] AntTweakBar, [online]. 2018 [cit. 2018-2-13]
Dostupné z: <http://anttweakbar.sourceforge.net/doc/>

Přílohy

Součástí přiloženého CD je:

- Elektronická verze této práce ve formátu PDF.
- Zdrojové kódy demonstrační aplikace.
- Spustitelný soubor demonstrační aplikace.