VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Simulace davu

Crowd Simulation

2018 Adam Lasák

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

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 možnosti 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 srovnejte a vyhodnoťte.

Seznam doporučené odborné literatury:

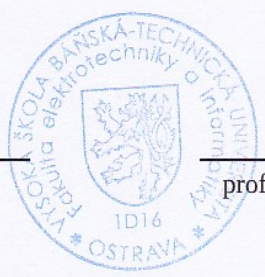
[1] Craig Reynolds: "Boids Background and Update",[online], http://www.red3d.com/cwr/boids/
[2] Addison-Wesley: "OpenGL Programming Guide",(Red book) [online].

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 Brandštetter, CSc.
děkan fakulty

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

V Ostravě 1. dubna 2018 .....

Tímto bych rád poděkoval svému vedoucímu, Ing. Martinu Němcovi, Ph.D., za poskytnuté odborné rady, za neocenitelné zkušenosti a za všechen 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. Toho se hojně využívá např. ve filmech, hrách či návrhu budov. Tato práce se zaměřuje na popis boidova algoritmu, který je dnes nejpoužívanější co se simulace davu týče.

Klíčová slova: dav, vizualizace, optimalizace, OpenGL, Boidův algoritmus, 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 the behavior, we can create real crowd form. This can be useful f.e. in movies, games or for building design. This thesis focuses on the description of boid's algorithm which is the most used crowd simulation principle today.

Key Words: crowd, visualization, optimization, OpenGL, Boid's algorithm, cohesion, separation, alignment, agent

Obsah

Seznam použitých zkratek a symbolů

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

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Simulační algoritmy se používají v širokém spektru odvětví od vědy, her, výpočetních úkonů až po kinematografii či stavbě budov [6]. Herní využití mívá velmi reálně implementována armáda [8], která tímto způsobem zaškoluje vojáky ve virtuálním simulačním boji jak v taktice tak způsobu nejefektivnějšího využití dostupných zbraní.

Pokud vememe 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.

Avšak po této základní konstrukční stránce, se musí také navrhnout optimální velikost budovy a kolik bude schopna pojmout lidí v jednom okamžiku. Kde bude vést úniková cesta v případě požáru a kolik času zabere davu, než se z budovy dostane ven. A zrovna co se bezpečnosti týče, mají simulační algoritmy nejširší využití. V jistém slova smyslu by se dalo řici, že byly vytvořeny primárně pro tento účel [1].

Při těchto simulacích se tak navrhují nejlepší varianty šířky chodeb, dostupnosti k únikovým východům, přístupu k požárnímu schodišti nebo vyladění 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 [2] pro tento druh simulace a realizaci výsledné aplikace.

2 Simulace

Pojem simulace lze shrnout do obecné věty: *"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 [5])

V našem případě se jedná o napodobení davu, kdy pomocí algoritmu *(popsaného v dalších kapitolách)* 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

Simulaci davu lze zrealizovat dvěmi metodami. Částicová simulace a simulace založená na umělé inteligenci. [12]

2.2 Částicová simulace

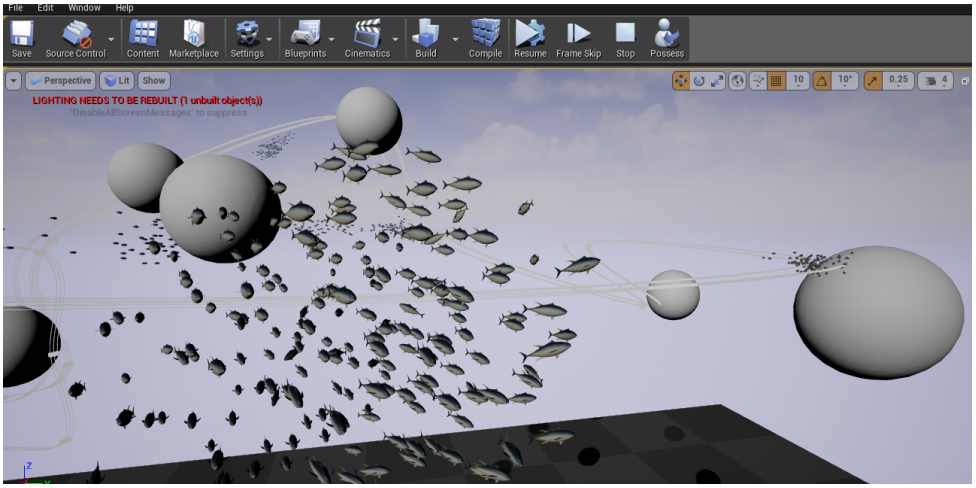
První zmíněná simulace je založena na principu přiřazení hmotného bodu každému prvku z množiny všech prvků které mají spolu iterovat[1]. Hlavní výhodou využití tohoto typu simulace je možnost použití velkého množství prvků z celkové množiny, neboť výpočty základních sil nejsou příliš náročné na výpočetní výkon. Příklady pro využití jsou typy založené na: magnetických silách, buněčném modelu a sociálních silách. [12]

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 vlastostni agenti stále musí mít)* ale také přidanými vlastnostmi, kterými předchozí typ tolik nedisponoval. Mají hlavně přidané senzory, díky nímž dokáží vyhodnocovat danou situaci v reálném čase a následně se rozhodovat k dalšímu nejvýhodnějšímu 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. Ať už jednodušší *(rozhodování dalšího kroku mezi zdmi)* či složitějšího *(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

[1] Iterace je v programování proces při kterém se soubor pokynů opakuje přesně daným počtem opakování, nebo dokud je splněna podmínka.[9]



Obrázek 1: Ukázka simulace hejna ryb v Unreal Enginu 4
(Zdroj: `http://blog.csdn.net/nosix/article/details/52859160`)

než částicová simulace kvůli mnoha vlastnostem agentů. Konkrétními případy může být třeba hra Crysis *(leden 2007 - stáda jelenů)* a FarCry *(2004 - hejna tropických papoušků)*.

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

Jelikož jsem využil tento typ simulace, tak v dalších kapitolách budeme každý bod ve scéně nazývat agentem.

2.4 Craig Reynoldův algoritmus

Nejčastějším typem tohoto typu simulace je Craig Reynoldův algoritmus umělé inteligence vytvořený v roce 1986 a oficiálně představený roku 1987 na konferenci ACM SIGGRAPH [10, 11].

Nejčastěji se však setkáme s názvem Boidův algoritmus, který je odvozen od boidů *(boids)* čili alternativnímu názvu agentů. 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 popsané níže, znamená to, že každý iteruje [9] s každým.

Mezi částicové prvky Reynoldova algoritmu lze použit 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)* [13].

V této práci jsem toto vylepšení použil a tím přiblížil chování davu více realitě. Přidané fyzikální veličiny jsou již zmíněné: tření, zrychlení a okamžitá rychlost.

Nyní se zaměříme na popis Reynoldova algoritmu, který obsahuje tři základní vlastnosti:

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

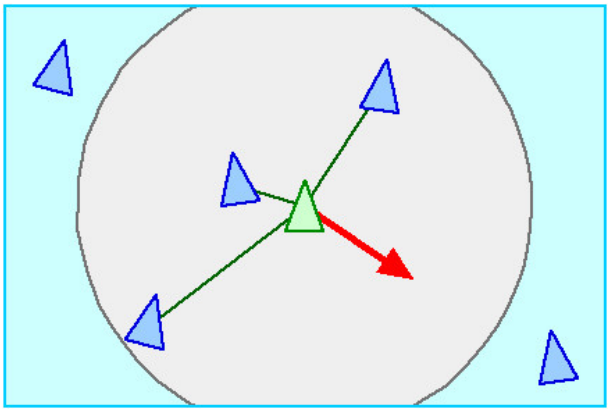
2.5 Separace

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 Reynoldova 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.

Následuje sled základních operací které provedou separaci od ostatních agentů.

1. Odečtení dvou vektorů podle vzorce $vSub = v1 - v2$, kde $v1$ je vektor aktuálního agenta a $v2$ je vektor agenta ke kterému jsme se přiblížili.
2. Normalizování vektoru podle vzorce $vNorm = norm(vSub)$, kde $vSub$ je výsledek z předchozího kroku.
3. Vydělení výsledku kroku č. 2 podle vzorce $vRes = vNorm/d$, kde d je vzdálenost těchto agentů mezi sebou a $vRes$ je celkový výsledek těchto základních operací, které se následně využívají v dalších krocích *(ty budou popsány v implementaci výsledné aplikace)*.



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

[2]Koheze je fyzikální síla držící pohromadě atomy či molekuly téže látky či tělesa (zejména kapalného a pevného tělesa), pozn. upraveno [14]

Následuje ukázka jak vypadá pseudokód pro separaci.

```
PROCEDURE Separation(boid bJ)

Vector result;

FOR EACH BOID b
   IF b != bJ THEN
      IF |b.position - bJ.position| < 100 THEN
         sub = b.position - bJ.position
         Normalize(sub)
         result = sum / |b.position - bJ.position|
      END IF
   END IF
END

RETURN result

END PROCEDURE
```

Výpis 1: Pseudokód pro separaci

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.

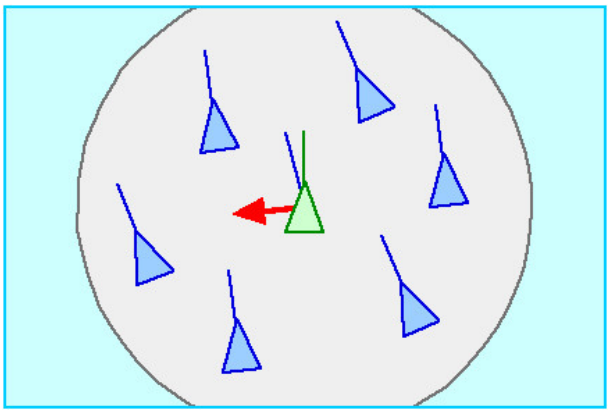
Princip fungování je obdobný jako u separace, pokud se v blízkosti okruhu od daného agenta objeví jiný agent případně agenti, zprůměruje se jejich rychlost a směr a na základě těchto údajů modifikujeme údaje konkrétního agenta a tím zajístíme soudržnost všech dohromady.

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 *sum*.

2. Jestliže se podmínka nesplnila a tudíž sumární vektor *sum* je roven nule pak se pravidlo zarovnání ukončuje. Pokud se alespoň jednou provedla podmínka v kroku č. 1 tak následuje sled následujících událostí.

- sumární vektor *sum* se vydělí skalární[3] hodnotou rovné počtu splněných podmínek v kroku č. 1
- normalizování sumárního vektoru *sum*
- vypočtení výsledného vektoru dle kterého se upraví směr daného agenta vzorcem $vRes = sum - vel$, kde *sum* je sumární vektor a *vel* je aktuální rychlost agenta.



Obrázek 3: Zarovnání - určení směru agenta vůči průměrnému směru ostatních (Zdroj: [2])

Následuje ukázka pseudokódu jak by mohla vypadat procedura Alignment.

```
PROCEDURE Alignment(boid bJ)

   Vector sum
   Integer count

   FOR EACH BOID bJ
      IF b != bJ THEN
         sum = sum + b.velocity
         count = count + 1
      END IF
   END

   IF count > 0 THEN
      sum = sum / count
      Normalize(sum)
      RETURN (sum - velocity)
   END ELSE
      RETURN sum
```

[3]Skalár je veličina, která je určená pouze svojí velikostí, tj. její hodnota je popsána jediným číslem (skalární veličiny ve fyzice jsou např. hmotnost, objem, teplota). (Citace [15])

```
    END
END PROCEDURE
```

Výpis 2: Pseudokód pro zarovnání

2.7 Koheze

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

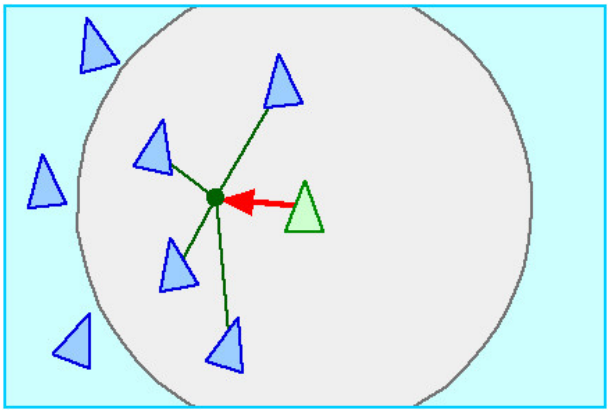
Funguje na principu spojení okolních agentů *(a následně vytvoření skupiny)*, kteří zasahují do dalšího pomyslného kruhu zarovnání. Aby fungovalo toto pravidlo a nekolidovalo s podmínkami separace musí platit $constCoh > constSep$, kde $constSep$ je maximální průměr kružnice ve které se při překročení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

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 směřovat.

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 *sum* avšak ne rychlost, ale lokace blízkého agenta.

2. Následně se provede tentýž krok jako u zarovnání, čili sumární vektor *sum* se vydělí skalárem počtu splněných podmínek v kroku č. 1



Obrázek 4: Koheze - určení směru k průměrné lokaci okolních agentů (Zdroj: [2])

Následuje ukázka pseudokódu jak by vypadala procedura Cohesion. Kód je v tomto pseudokódu obdobný jako kód zarovnání výjma řádku s normalizováním sumárního vektoru, avšak v implementační části je tento kód složitější v řádku vracení výsledné hodnoty, kdy se ještě řeší fyzikální vlastnosti.

```
PROCEDURE Cohesion(boid bJ)

   Vector sum
   Integer count

   FOR EACH BOID bJ
      IF b != bJ THEN
         sum = sum + b.location
      END IF
   END

   IF count > 0 THEN
      sum = sum / count
      RETURN (sum - velocity)
   END ELSE
      RETURN sum
   END

END PROCEDURE
```

Výpis 3: Pseudokód pro kohezi

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 procedur 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 [13]. 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.

```
PROCEDURE AllBoidsToNewPosition()

   Vector v1, v2, v3
   Boid b

   FOR EACH BOID b
      v1 = Separation(b)
      v2 = Alignment(b)
      v3 = Cohesion(b)

      b.velocity = b.velocity + v1 + v2 + v3
      b.position = b.position + b.velocity
   END

END PROCEDURE
```

Výpis 4: Pseudokód pro aplikování třech pravidel

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

3 Implementace

3.1 Popis realizace demonstrační aplikace

Cílem demostrační aplikace je zrealizování a zvizualizování Craig Reynoldova algoritmu do využitelné podoby. Tento algoritmus lze aplikovat mnoha způsoby avšak rozhodl jsem se pro dvě spodní varianty:

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

Vetšinu parametrů lze měnit/kombinovat v config.cfg souboru *(lze i kombinovat jednotlivá tři pravidla postupným zapínáním/vypínáním)* a měnit mapu včetně únikových bodů lze v souboru map.txt, která je použitelná pouze pro první uvedený typ.

Při vypnutých pravidlech kromě separace se dav bude chovat stylem *"každý si jde kam chce"*.

3.2 Použité technologie

Výsledná aplikace je naprogramována v jazyce C++ a jako hlavní nástroj jsem použil Visual Studio verze 2017 [20], protože je nejpoužívanějším IDE[4] pro vývoj [19].

Jako grafickou knihovnu jsem použil OpenGL [21] s rozšířením pro aplikační okna GLFW *(verze 3.2.1)* [22] a realtime mechanismy GLEW *(verze 2.1.0)* [23].

K načítání objektů jsem použil knihovnu Assimp [25] *(verze 3.2)*, která je vhodná kvůli velké škále podporovaných grafických formátů.

Pro matematické operace jsem použil knihovnu GLM [24] *(verze 0.9.8.5)* hlavně pro počítání s maticemi.

Pro zobrazování parametrů a proměnných byla použita GUI AntTweakBar knihovna *(verze 1.16)* [28] kvůli snadnému použití a intuitivnímu rozhraní.

3.3 Optimalizace

První optimalizace proběhla kvůli paměťovým nárokům. Problém 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

[4]Integrated Development Environment - program, který nabízí velkou škálu nástrojů usnadňující programátoskou práci. [16])

Jak je patrné z hodnot, tak od druhé minuty nárust paměti stoupá takřka exponenciálně. Důvod tak rychlého nárustu byl ten, že jak se postupně k sobě agenti přibližovali, začínalo se aplikovat více pravidel, které ještě neměli optimalizované operace s pamětmi. Po pěti minutách byl nárust tak obrovský že hodnota FPS klesla na 10.

Začal jsem tedy hledat slabá místa a nakonec se mi veškeré operace s pamětma podařilo dostat pod kontrolu a eliminovat tak veškerý nárust využití paměti.

Příklad ošetření jedné z mnoha částí kódu můžeme vidět zde.

```
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* který slouží jako prostředník pro uchování adresy. Pak tedy můžeme tuto původní adresu přiřadit zpět, v tomto případě poslední řádek kódu.

Jelikož jsem se snažil maximálně šetřit využití paměťi ve třídě *Boids*, jejíž instance je volána v každé smyčce tak jednotlivá pravidla i metody pro mezivýpočty používali mezi sebou stejné adresy pamětí.

Aby nedocházelo k neoprávněným přístupům a záměnou adres, využil jsem pomocnou proměnnou do které jsem dočasně uložil výsledek pravidla a pak pomocí metody *set* bezpečně nastavil hodnoty vektoru bez jakéhokoliv úniku paměti. Tento způsob jsem aplikoval na všechna použitá pravidla, včetně vyhýbání se zdem.

Vůči 100 agentům ve scéně měla aplikace využití 110 MB včetně použitých textur a objektů, po těchto doposud zmíněných optimalizací je využití 91 MB.

Další vylepšení bylo lepší použití cyklů. Klasický zápis cyklu který jsem před tímto typem optimalizace používal je následujicí:

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

Výpis 6: Použití klasického cyklu

Po pročtení oficiálních referencí pro C++ [17] jsem zjistil že při každé iteraci cyklem se stále volá metoda *size()*, která tento výkon podstatně snižuje. Řešením je uložit tuto hodnotu do proměnné. Dalším vylepšením je použití pre-inkrementace proměnné *i*. Toto použití je mnohem účinnější než post-inkrementace z důvodu, že nepoužívá dočasné úložiště a vkládání potřebného kódu na pozadí [18]. Výsledný cyklus pak vypadá takto:

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

Výpis 7: Použití optimalizovaného cyklu

Tyto zdánlivě bezvýznamné změny v důležitých cyklech dokázali přidat až 5 FPS.

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

Toto jsem vyřešil pomocí kopírovacího konstruktoru. Daný typ objektu jsem načetl pouze jednou a pak v cyklu předával adresu načteného objektu dalšímu konstruktoru, který tuto paměť nakopíroval pro svou instanci. Tato část optimalizace ušetřila při načítání až 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 [13] již zmíněné v teoretické části jsem použil pro reálnější podobu chování celého davu.

Jako první vlastnost jsem použil hledání *(seek)* primárně používané u Koheze při návratové hodnotě (viz kapitola 2.7), kde jsem upozornil na složitější implementační část oproti zarovnání.

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

Následující kód ukazuje jak se seek vektor vypočte.

```
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. Naposledy pak musíme vektor limitovat konstantou tření *(maxForce)*.

Další vlastností kterou jsem použil je následování bodů *(Path Following)*. V mé práci slouží jako kontrolní body 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).

3.5 Únik z budovy

4 Závěr

Reference

[1] Journal of the royal society interface, *Crowd behaviour during high-stress evacuations in an immersive virtual environment*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://rsif.royalsocietypublishing.org/content/13/122/20160414`

[2] Boids, *Background and Update by Craig Reynolds*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `https://www.red3d.com/cwr/boids/`

[3] The nature of code, *Chapter 6. Autonomous Agents*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://natureofcode.com/book/chapter-6-autonomous-agents/`

[4] Boids Pseudocode, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://www.kfish.org/boids/pseudocode.html`

[5] Simulace, *Glosář Aldebaran*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://www.aldebaran.cz/glossary/print.php?id=1299`

[6] Oasys, *Crowd Simulation*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://www.oasys-software.com/products/engineering/massmotion.html`

[7] Earthlight, *Spacewalk*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://www.earthlightvr.com/spacewalk/`

[8] FAAC Military, , [online]. 2018 [cit. 2018-2-1]
Dostupné z: `https://www.faac.com/military/`

[9] Techopedia, *Iteration*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `https://www.techopedia.com/definition/3821/iteration`

[10] ACM, *Association for Computing Machinery*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `https://www.acm.org/about-acm/what-is-acm`

[11] SIGGRAPH, , [online]. 2018 [cit. 2018-2-1]
Dostupné z: `https://www.siggraph.org/about/what-is-acm-siggraph`

[12] 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/`

[13] Steering Behaviors For Autonomous Characters
Craig W. Reynolds, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://www.red3d.com/cwr/steer/gdc99/`

[14] Kohezní síla, *Glosář Aldebaran*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `http://www.aldebaran.cz/glossary/print.php?id=1894`

[15] Katedra Mechaniky, *Skalár*, [online]. 2018 [cit. 2018-2-1]
Dostupné z: `https://www.kme.zcu.cz/kmet/bio/matskalvekt.php`

[16] Techopedia, *IDE*, [online]. 2018 [cit. 2018-2-4]
Dostupné z:
`https://www.techopedia.com/definition/26860/integrated-development-environment-ide`

[17] CppReference.com, , [online]. 2018 [cit. 2018-2-4]
Dostupné z: `http://en.cppreference.com/w/`

[18] CppReference.com, *Increment/decrement operators*, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `http://en.cppreference.com/w/cpp/language/operator_incdec`

[19] Top IDE Index, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `https://pypl.github.io/IDE.html`

[20] Visual Studio IDE, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `https://www.visualstudio.com`

[21] OpenGL, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `https://www.opengl.org/`

[22] GLFW, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `http://www.glfw.org/`

[23] GLEW, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `http://glew.sourceforge.net//`

[24] GLM, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `https://glm.g-truc.net/0.9.8/index.html`

[25] Assimp, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `http://assimp.sourceforge.net/`

[26] Stb image, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `https://github.com/nothings/stb`

[27] OpenCV, [online]. 2018 [cit. 2018-2-4]
Dostupné z: `https://opencv.org/`

[28] AntTweakBar, [online]. 2018 [cit. 2018-2-13]
Dostupné z: http://anttweakbar.sourceforge.net/doc/