**АКАДЕМИЈА ТЕХНИЧКИХ СТРУКОВНИХ СТУДИЈА БЕОГРАД**

**ОДСЕК ПРИМЕЊЕНЕ ИНЖЕЊЕРСКЕ НАУКЕ**

****

**Душан Фајлер**

**Симулација групног кретања помоћу**

**OpenGL спецификације и C++ програмског језика**

**- ЗАВРШНИ РАД -**

**Пожаревац, oктобар 2022.**

**АКАДЕМИЈА ТЕХНИЧКИХ СТРУКОВНИХ СТУДИЈА БЕОГРАД**

**ОДСЕК ПРИМЕЊЕНЕ ИНЖЕЊЕРСКЕ НАУКЕ**

**Студијски програм:** Електротехника и рачунарство

**Модул:** Информатика и рачунарство

**Симулација групног кретања помоћу**

**OpenGL спецификације и C++ програмског језика**

**- ЗАВРШНИ РАД -**

**Студент: Ментор:**

Душан Фајлер 40197/2022 Алекса Срданов

**Пожаревац, октобар 2022.**

***библиографски подаци***

Научно поље Техничко-технолошке науке

Научна област Електротехничко и рачунарско инжењерство

Институција Академија техничких струковних студија Београд,Одсек Примењене инжењерске науке

Ментор рада Алекса Срданов

Број страница 25

Број слика 11

Број табела 0

Број прилога 0

Број литературних референци 8

Датум одбране \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оцена израде рада \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оцена одбране рада \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Коначна оцена \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Комисија за одбрану рада:

1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(председник комисије)

2. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ментор)

3. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(члан)

Садржај

[1. Увод 6](#_Toc117741377)

[2. Боиди 7](#_Toc117741378)

[3. *OpenGL* 8](#_Toc117741379)

[3.1. Екстензије 9](#_Toc117741380)

[3.2. Машина стања 9](#_Toc117741381)

[4. Библиотеке 10](#_Toc117741382)

[4.1. *Simple DirectMedia Layer (SDL)* 10](#_Toc117741383)

[4.2. *The* *OpenGL Extension Wrangler Library (GLEW)* 10](#_Toc117741384)

[4.3. *OpenGL Mathematics (GLM)* 10](#_Toc117741385)

[4.4. *PicoPNG* 10](#_Toc117741386)

[5. Поставка пројекта 11](#_Toc117741387)

[5.1. Развојно окружење 11](#_Toc117741388)

[5.2. Структура директоријума 11](#_Toc117741389)

[5.3. Библиотеке 11](#_Toc117741390)

[6. Објашњење кода 14](#_Toc117741391)

[6.1. *main.cpp* - Почетна тачка 14](#_Toc117741392)

[6.2. *MainGame* класа 15](#_Toc117741393)

[6.2.1. *MainGame.h* 15](#_Toc117741394)

[6.2.2. *MainGame.cpp* 17](#_Toc117741395)

[6.3. *BoidManager* класа 21](#_Toc117741396)

[6.3.1. *BoidManager.h* 21](#_Toc117741397)

[6.3.2. *BoidManager.cpp* 21](#_Toc117741398)

[6.4. *Boid* класа 24](#_Toc117741399)

[6.4.1. *Boid.h* 24](#_Toc117741400)

[6.4.2. *Boid.cpp* 25](#_Toc117741401)

[7. Zaključak 29](#_Toc117741402)

[8. Reference 30](#_Toc117741403)

# Увод

Тема овог рада је симулација групног кретања у природи. То може бити јато птица, крдо бизона, рој мушица итд.

Кретање јата птица једно је од ужитака природе. Јата и сродна синхронизована групна понашања као што су јата риба или крда копнених животиња су и лепа за посматрање и интригантна за размишљање. Јато  показује много контраста. Састоји се од дискретних птица, али свеукупно кретање изгледа течно. Једноставан је у концепту, али је визуелно сложен, изгледа насумично поређан, а ипак је синхронизован.

Ако претпоставимо да је јато једноставно резултат интеракције између понашања појединих птица, да бисмо симулирали јато, симулирамо понашање појединачне птице.

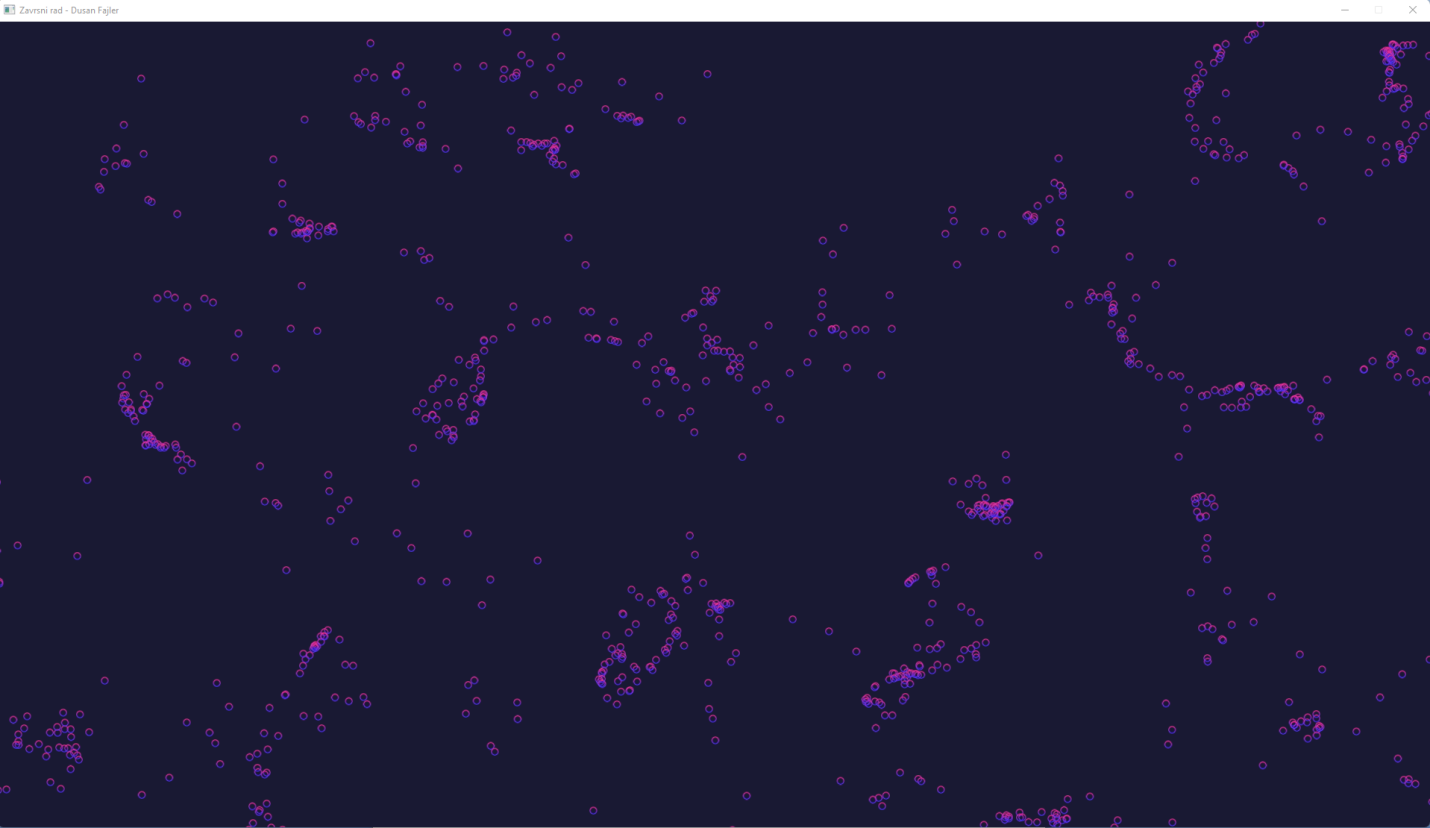
Ако овај симулирани модел птице има исправно понашање члана јата, све што би требало да буде потребно да би се креирало симулирано јато је да се креирају неке инстанце симулираног модела птице и да им се омогући интеракција.

Појединачну јединку у групи ћемо у овом раду називати „Боид“ (енг. *Boid*).

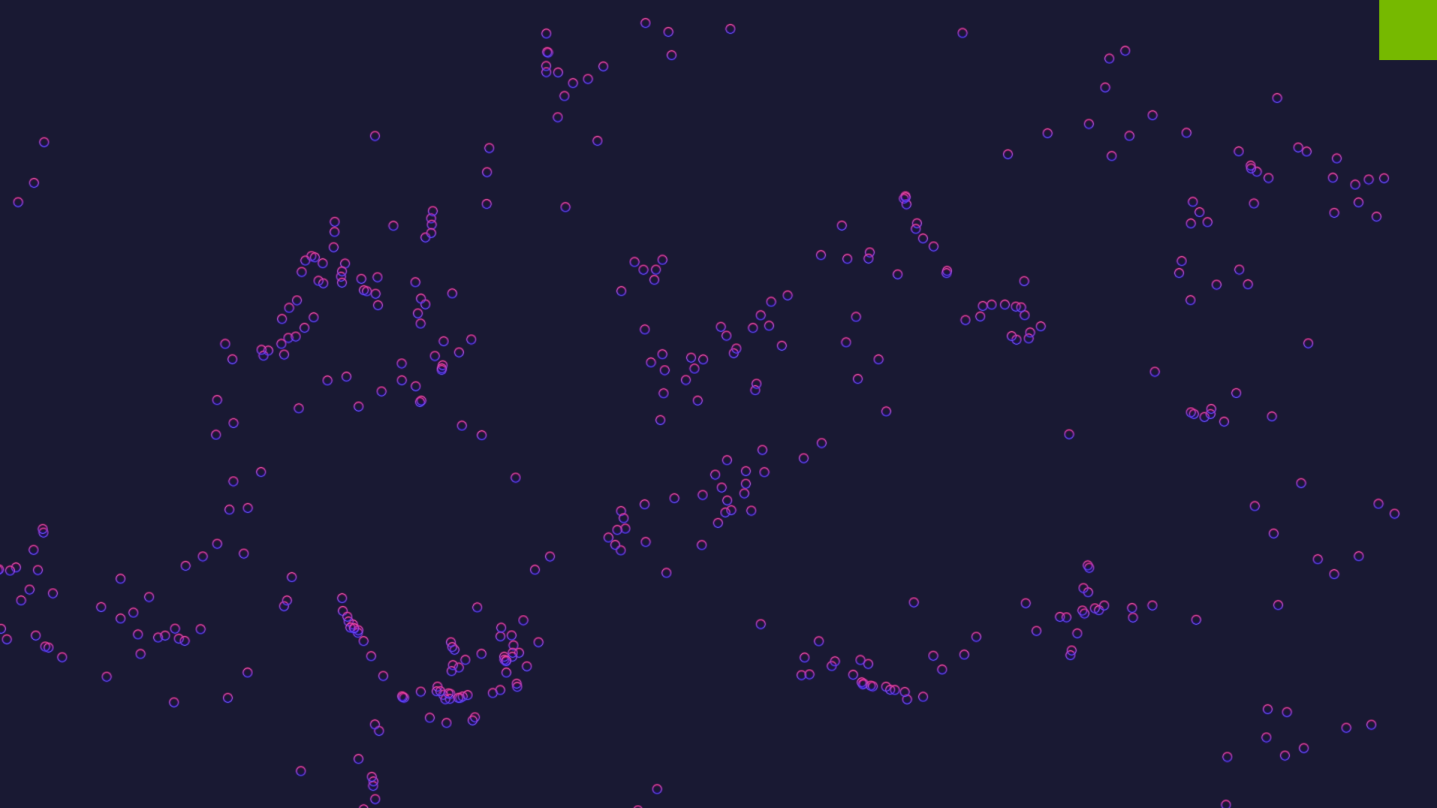
С обзиром да симулацију приказујемо без икаквих покретача игара или фрејмворка велики део кода је заправо поставка за приказивања симулације.

Међутим да бисмо мало смањили комплексност и обим кода, користићeмо следеће библиотеке које ће нам помоћи да брже стигнемо до исцртавања боида на екран:

* *Simple DirectMedia Layer (SDL)* – обезбеђује отварање прозора, детекцију уноса, мерење времена итд.
* *The OpenGL Extension Wrangler Library (GLEW)* – учитавање *OpenGL* екстензија
* *OpenGL Mathematics (GLM)* – операције над векторима
* *PicoPNG* – декодирање *.png* фајлова у бафер меморије



Сл. 1а. Симулација

****

Сл. 1б. Симулација

# 

# 2. Боиди

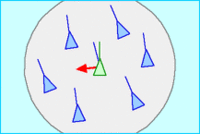
Као и код већине вештачких симулација живота, Боиди су пример појавног понашања, то јест, сложеност Боида произилази из интеракције појединачних агената (у овом случају боида) који се придржавају скупа једноставних правила. Правила која се примењују у најједноставнијем свету Боида су следећа:

* **Раздвајање**: усмерити се да би се избегла гужва локалних Боида из групе



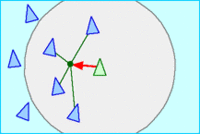
Сл. 2. Раздвајање

* **Поравнање**: усмерити се ка просечном курсу локалних Боида из групе



Сл. 3. Поравнање

* **Кохезија:** усмерити се да би се кретали ка просечној позицији (центру масе) локалних Боида из групе



Сл. 4. Кохезија

# 3. *OpenGL*

*OpenGL* се углавном сматра *API*-јем (*application programming interface*) који нам пружа велики скуп функција које можемо користити за манипулацију графиком и сликама. Међутим, *OpenGL* сам по себи није *API*, већ само спецификација, коју је развила и одржава Khronos Group.

*OpenGL* спецификација тачно специфицира шта би требало да буде резултат/излаз сваке функције и како треба да ради. Затим је на програмерима који имплементирају ову спецификацију да пронађу решење како би ова функција требало да функционише.

Људи који развијају *OpenGL* библиотеке су обично произвођачи графичких картица. Свака графичка картица коју купите подржава одређене верзије *OpenGL*-а које су развијене посебно за ту картицу (серију). Када се користи *Apple* систем, *OpenGL* библиотеку одржава сам *Apple*, а под *Linuxom* постоји комбинација верзија графичких добављача и адаптација ових библиотека од стране хобиста. Ово такође значи да кад год *OpenGL* показује чудно понашање које не би требало, то је највероватније грешка произвођача графичких картица (или ко год је развио/одржавао библиотеку).

У старим данима, коришћење *OpenGL*-а је значило развој у непосредном режиму (често се назива и цевовод фиксне функције (*fixed function pipeline*)) који је био лак за коришћење метода за цртање графике. Већина функционалности *OpenGL*-а била је скривена унутар библиотеке и програмери нису имали много контроле над начином на који *OpenGL* врши своје прорачуне. Програмери су на крају постали гладни за више флексибилности и временом су спецификације постале флексибилније.

Као резултат, програмери су добили већу контролу над својом графиком. Непосредни режим је заиста једноставан за коришћење и разумевање, али је такође изузетно неефикасан. Из тог разлога спецификација је почела да застарева и функционалност тренутног режима од верзије 3.2 надаље и почела је да мотивише програмере да развијају у *OpenGL*-у језгро-профил режим-у (*core profile*), који је део *OpenGL*-ове спецификације који је уклонио све старе застареле функционалности.

Када користимо *OpenGL*-ов основни профил, *OpenGL* нас приморава да користимо модерне праксе. Кад год покушамо да користимо неку од његових застарелих функција, он јавља грешку и зауставља цртање. Предност учења савременог приступа је што је веома флексибилан и ефикасан. Међутим, такође је теже научити. Непосредни режим је доста сажимао стварне операције које је *OpenGL* изводио и иако је било лако научити, било је тешко схватити како он заправо функционише. Савремени приступ захтева од програмера да заиста разуме *OpenGL* и графичко програмирање и иако је помало тешко, омогућава много већу флексибилност, већу ефикасност и што је најважније: много боље разумевање графичког програмирања.

## 3.1. Екстензије

Одлична карактеристика *OpenGL*-а је његова подршка екстензијама. Кад год графичка компанија смисли нову технику или нову велику оптимизацију за приказивање, то се често налази у екстензији имплементираној у драјверима. Ако хардвер на којем апликација ради подржава такво проширење, програмер може користити функционалност проширења за напреднију или ефикаснију графику. На овај начин, графички програмер и даље може да користи ове нове технике приказивања без потребе да чека да *OpenGL* укључи функционалност у своје будуће верзије, једноставно проверавајући да ли графичка картица подржава проширење. Често, када је проширење популарно или веома корисно, оно на крају постане део будућих верзија *OpenGL*-а.

## 3.2. Машина стања

*OpenGL* је сам по себи велика државна стања, тј. колекција варијабли које дефинишу како би *OpenGL* тренутно требало да ради. Стање *OpenGL*-а се обично назива *OpenGL* контекст. Када користимо *OpenGL*, често мењамо његово стање тако што постављамо неке опције, манипулишемо неким баферима и затим рендерујемо користећи тренутни контекст.

Кад год кажемо *OpenGL*-у да сада желимо да цртамо линије уместо троуглова, на пример, мењамо стање *OpenGL*-а променом неке променљиве контекста која поставља како би *OpenGL* требало да црта. Чим променимо контекст говорећи *OpenGL*-у да треба да црта линије, следеће команде за цртање ће сада цртати линије уместо троуглова.

4. Библиотеке

## **4.1. *Simple DirectMedia Layer (SDL)***

Да бисмо уопште могли да цртамо графику помоћу OpenGL-а, потребан нам је прозор у коме ћемо да цртамо. Отварање прозора је специфична операција за сваки оперативни систем и *OpenGL* намерно покушава да се апстрактује од оперативних система. То значи да морамо да креирамо прозор, дефинишемо контекст и сами управљамо корисничким уносом.

SDL је вишеплатформска развојна библиотека дизајнирана да обезбеди приступ ниског нивоа аудио, тастатури, мишу, џојстику и графичком хардверу преко *OpenGL/Direct3D/Metal/Vulkan* спецификација. Користе га софтвери за репродукцију видео записа, емулатори, видео игре итд.

SDL званично подржава *Windows, MacOS, Linux, iOS* и *Android*. Подршка за друге платформе се може наћи у изворном коду.

4.2. *The* *OpenGL Extension Wrangler Library (GLEW)*

Пошто је *OpenGL* заправо само стандард/спецификација, на произвођачу драјвера је да имплементира спецификацију за драјвер који одређена графичка картица подржава. Пошто постоји много различитих верзија *OpenGL* драјвера, локација већине његових функција није позната у време компајлирања и треба да буде упитана током извршавања. Тада је задатак програмера да пронађе локацију функција које су му потребне и сачува их у показивачима функција за каснију употребу. Преузимање тих локација је специфично за сваки оперативни систем и јако је тежак и напоран посао. Ту „ускаче“ *GLEW*.

*GLEW* jе вишеплатформска библиотека за учитавање проширења *C/C++* отвореног кода. *GLEW* обезбеђује ефикасне механизме извођења за одређивање које *OpenGL* екстензије су подржане на циљној платформи. *OpenGL Core* и функционалност проширења су изложени у једној датотеци заглавља.

4.3. *OpenGL Mathematics (GLM)*

*GLM* је библиотека математике за C++ и садржана је само од заглавља (.*hpp*), а заснована је на спецификацијама *OpenGL Shading Language* - a (*GLSL*) и обезбеђује класе и функције за математичке операције: матричне трансформације, кватернионе, паковање података, случајне бројеве, шум, итд...

## 4.4. *PicoPNG*

*PicoPNG* је мало заглавље за декодирање и енкодирање .*png* фајлова. Користићемо га за декодирање текстуре којом ћемо да представимо боиде.

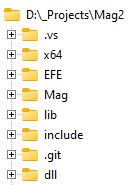
5. Поставка пројекта

## 5.1. Развојно окружење

Као развојно окружење ћемо користити *Visual Studio Community 2022*. То је софтверски програм који служи за писање и уређивање кода. Његов кориснички интерфејс се користи за развој софтвера за уређивање, отклањање грешака и прављење кода. *Visual Studio* укључује уређивач кода који подржава *IntelliSense* (компоненту за довршавање кода) као и рефакторисање кода.

## 5.2. Структура директоријума

Кеш и привремене информације о тренутном пројекту



Извршни директоријум

Наш фрејмворк

Наш главни пројекат

Статичке библиотеке

“Header” фајлови библиотека

Информације о *git* репозиторијуму

Сл. 5. Структура

директоријума

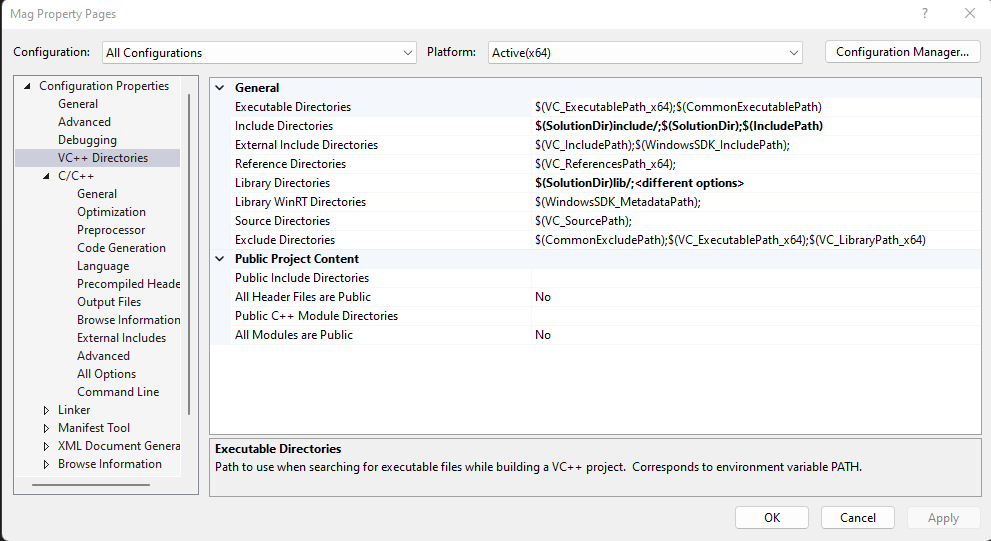
Динамичке библиотеке

## 5.3. Увоз библиотека

Да бисмо приступили библиотекама, морамо да прво да их подесимо за наш пројекат. С обзром да користимо Visual Studio, користићемо његов систем за подешавање библиотека.

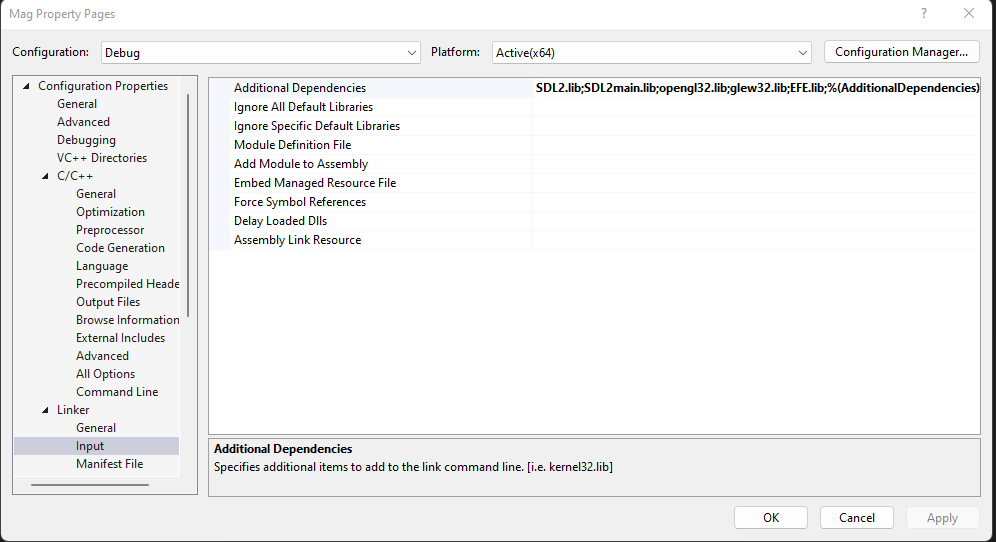
## 

Сл. 6. Локација подешавања пројекта

**** Сл. 7. Увезивање библиотека у пројекат

“Header” fajlovi

Статичке библиотеке



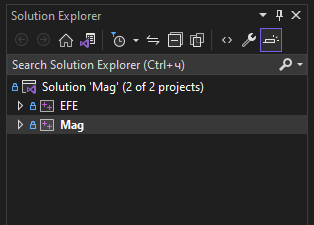
Сл. 8. Навођење статичких библиотека за линкер



Сл. 9. Провера исправности увезених библиотека

Укључивањем библиотека у код и компајлирањем, уверавамо се да смо добро увезали библиотеке

# 6. Објашњење кода



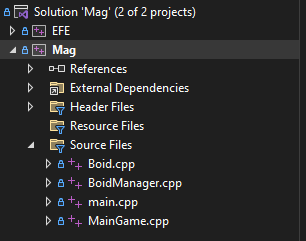
Сл. 10. Пројекти у солуцији *Visual Studio-a*

Да не бисмо мешали логику самих Боида и исцртавања на екран, примања улаза из миша, тастатуре итд., померање камере, учитавање текстура итд. Направићемо наш фрејмворк који ћемо касније, у главном програму само да користимо и тиме ћемо да имамо доста јаснији код и олакшаћемо рад. Такође, можемо користи исти фрејмворк за друге пројекте.

* *EFE* – Наш фрејмворк који је статичка библиотека коју ћемо користити у *Mag* пројекту
* *Mag* – Извршни програм у коме ћемо писати логику за Боиде

## 6.1. *main.cpp* - Почетна тачка

Почетна тачка сваког *C++* програма је фајл *main.cpp*. C обзиром да нам је *Mag* пројекат који је заправо главна апликација, ту ћемо и да га ставимо. Ту су и остали фајлови који чине апликацију о којимa ћемо да причамо мало касније.



Сл. 11. Локација *main.cpp* фајла

#include <iostream>

#include "MainGame.h"

int main(int argc, char\*\* argv)

{

MainGame mainGame;

mainGame.run();

return 0;

}

У *main.cpp* у main() функцији се само направи инстанца класе *MainGame* и покрене се “*run()*” функција из те инстанце

## 6.2. *MainGame* класа

### 6.2.1. *MainGame.h*

На почетку *MainGame.h*  дефинишемо следећи код:

#pragma once

#include <SDL/SDL.h>

#include <GL/glew.h>

#include <EFE/EFE.h>

#include <EFE/Sprite.h>

#include <EFE/GLSLProgram.h>

#include <EFE/GLTexture.h>

#include <EFE/Window.h>

#include <EFE/Camera2D.h>

#include <EFE/SpriteBatch.h>

#include <EFE/InputManager.h>

#include <EFE/Timing.h>

#include <vector>

#include "Boid.h"

#include "BoidManager.h"

enum class GameState { PLAY, EXIT };

У овом делу, увозимо сва потребна заглавља које ћемо да користимо. Приметити да постоји доста увоза из путање *EFE/*. То су заглавља из нашег фрејмворка о којима ћемо причати касније.

Такође дефинишемо енумерацију стања игре. Докле год је *PLAY* стање активно, нормално извршавамо програм. Када се деси промена стања на *EXIT* програм завршава са радом.

class MainGame

{

public:

MainGame();

~MainGame();

// Polazna tačka programa

void run();

private:

// Inicijalizacija potrebnih sistema

void initSystems();

// Inicijalizacija *OpenGL* šejder programa

void initShaders();

// Glavna petlja za simulaciju

void gameLoop();

// Primanje inputa

void processInput();

// iscrtavanje simulacije

void drawGame();

GameState \_gameState;

BoidManager \_boidManager;

efe::Window \_window;

efe::Camera2D \_camera;

efe::GLSLProgram \_colorProgram;

efe::InputManager \_inputManager;

efe::SpriteBatch \_spriteBatch;

efe::FpsLimiter \_fpsLimiter;

// Širina i visina prozora

int \_screenWidth;

int \_screenHeight;

// Maksimalni broj frejmova u sekundi

float \_maxFPS;

// Trenutni broj frejmova u sekundi

float \_fps;

};

### 6.2.2. *MainGame.cpp*

#include "MainGame.h"

#include <iostream>

#include <string>

#include <EFE/errors.h>

#include <EFE/ResourceManager.h>

// Konstruktor: Inicijalizacija privatnih varijabli koristeći inicijaliznu listu

MainGame::MainGame() :

\_screenWidth(1920),

\_screenHeight(1080),

\_gameState(GameState::PLAY),

\_maxFPS(75.f),

\_fps(0.0f)

{

\_camera.init(\_screenWidth, \_screenHeight);

}

// U dekonstruktoru trenutno ne radimo ništa

MainGame::~MainGame()

{

}

// Polazna tačka programa

void MainGame::run()

{

// Inicijalizacija potrebnih sistema

initSystems();

// Pokretanje glavne petlje za simulaciju

gameLoop();

}

// Inicijalizacija *OpenGL* šejder programa

void MainGame::initSystems()

{

// Kreiranje prozora, sa imenom, širinom i visinom i

// dodatnim opcijama za tip prozora

\_window.create("Zavrsni rad - Dusan Fajler", \_screenWidth, \_screenHeight, 0);

// Inicijalizacija *OpenGL* šejder programa

initShaders();

// Inicijalizacija \_spriteBatch instance koju ćemo da koristimo za grupno iscrtavanje boida

\_spriteBatch.init();

// Inicijalizacija \_fpsLimiter instance koju koristimo za limitiranje koliko ćemo

// frejmova u sekundi obrađivati

\_fpsLimiter.init(\_maxFPS);

// Postavljanje granica u kojima će se stvoriti boidi

\_boidManager.updateBounds(\_screenWidth, \_screenHeight);

// Stvaranje boida

\_boidManager.spawnBoids(300, glm::vec2(0.f, 0.f));

}

// Inicijalizacija *OpenGL* šejder programa

void MainGame::initShaders()

{

// Kompajliranje *OpenGL* šejdera iz putanje

\_colorProgram.compileShaders("shaders/colorShading.vert", "shaders/colorShading.frag");

// Dodavanje atributa \_colorProgram instanci koju ćemo da koristimo za

// iscrtavanje boida

\_colorProgram.addAttribute("vertexPosition");

\_colorProgram.addAttribute("vertexColor");

\_colorProgram.addAttribute("vertexUV");

// Linkovanje šejdera

\_colorProgram.linkShaders();

}

void MainGame::gameLoop()

{

// Dok stanje igre nije EXIT, obrađivati igru

while (\_gameState != GameState::EXIT)

{

// Signaliziranje \_fpsLimiter-u da je frejm počeo

\_fpsLimiter.beginFrame();

// Provera da li je dobijen neki ulaz iz miša ili tastature

processInput();

// Računanje simulacije za sledeći frejm

\_boidManager.updateBoids();

// Ažuriranje kamere

\_camera.update();

// iscrtavanje simulacije

drawGame();

// Signaliziranje \_fpsLimiter-u da se frejm završio

\_fps = \_fpsLimiter.end();

}

}

// Provera da li je dobijen neki ulaz iz miša ili tastature

void MainGame::processInput()

{

// SDL struct događaja koji ćemo da napunimo uskoro

SDL\_Event evnt;

// Setovanje brzine kretanja kamere i brzine skaliranja

const float CAMERA\_SPEED = 10.f;

const float SCALE\_SPEED = 0.02f;

// Punjenje evnt varijable i prolazak kroz petlju za sve inpute

while (SDL\_PollEvent(&evnt))

{

switch (evnt.type)

{

// Ako je kliknut X u gornjem desnom uglu prozora menjamo

// stanje igre u EXIT i u sledećem frejmu izlazimo iz igre

case SDL\_QUIT:

\_gameState = GameState::EXIT;

break;

// Signaliziranje našem input manageru da je stisnuto dugme

// i prosleđivanje koje je dugme stisnuto

case SDL\_KEYDOWN:

\_inputManager.pressKey(evnt.key.keysym.sym);

break;

// Signaliziranje našem input manageru da je pušteno dugme

// i prosleđivanje koje je dugme pušteno

case SDL\_KEYUP:

\_inputManager.releaseKey(evnt.key.keysym.sym);

break;

// Signaliziranje našem input manageru da je stisnut klik

// i prosleđivanje koji je klik stisnut

case SDL\_MOUSEBUTTONDOWN:

\_inputManager.pressKey(evnt.button.button);

break;

// Signaliziranje našem input manageru da je pušten klik

// i prosleđivanje koji je klik pušten

case SDL\_MOUSEBUTTONUP:

\_inputManager.releaseKey(evnt.button.button);

break;

// Signaliziranje našem input manageru da je miš

// pomeren na neko novo mesto i prosleđivanje koorinata

case SDL\_MOUSEMOTION:

\_inputManager.setMouseCoords(evnt.motion.x, evnt.motion.y);

break;

}

}

// Pomeranje kamere Gore, Dole, Levo, Desno,

// ako su stisnuti WSAD dugmici, respektivno

if (\_inputManager.isKeyPressed(SDLK\_w))

{

\_camera.setPosition(\_camera.getPosition() + glm::vec2(0.f, CAMERA\_SPEED));

}

if (\_inputManager.isKeyPressed(SDLK\_s))

{

\_camera.setPosition(\_camera.getPosition() + glm::vec2(0.f, -CAMERA\_SPEED));

}

if (\_inputManager.isKeyPressed(SDLK\_d))

{

\_camera.setPosition(\_camera.getPosition() + glm::vec2(CAMERA\_SPEED, 0.f));

}

if (\_inputManager.isKeyPressed(SDLK\_a))

{

\_camera.setPosition(\_camera.getPosition() + glm::vec2(-CAMERA\_SPEED, 0.f));

}

// Zumiranje kamere ako je stisnuto Q ili E dugme

if (\_inputManager.isKeyPressed(SDLK\_q))

{

\_camera.setScale(\_camera.getScale() + -SCALE\_SPEED);

}

if (\_inputManager.isKeyPressed(SDLK\_e))

{

\_camera.setScale(\_camera.getScale() + SCALE\_SPEED);

}

// Stvaranje boida ako je stisnut levi klik

if (\_inputManager.isKeyPressed(SDL\_BUTTON\_LEFT))

{

// Uzimamo koorinate miša iz našeg input managera

glm::vec2 mouseCoords = \_inputManager.getMouseCoords();

// Konvertujemo koordinate miša na prozoru u koordinate u svetu

mouseCoords = \_camera.convertScreenToWorld(mouseCoords);

// Stvaramo boide

\_boidManager.spawnBoids(1, mouseCoords);

}

}

void MainGame::drawGame()

{

// Setovanje dubine koju će *OpenGL* da očisti

glClearDepth(1.0);

// Čišćenje bafera boje i dubine

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// Setovanje koji će *OpenGL* program da se koristi

\_colorProgram.use();

// Setovanje aktivne teksture na prvu teksturu

glActiveTexture(GL\_TEXTURE0);

// Setovanje lokacije uniform varijable teksture

GLint textureLocation = \_colorProgram.getUniformLocation("mySampler");

glUniform1i(textureLocation, 0);

// Setovanje lokacije uniform varijable lokacije i skale kamere

GLint pLocation = \_colorProgram.getUniformLocation("P");

glm::mat4 cameraMatrix = \_camera.getCameraMatrix();

glUniformMatrix4fv(pLocation, 1, GL\_FALSE, &(cameraMatrix[0][0]));

// Pripremanje \_spriteBatch instance za crtanje

\_spriteBatch.begin();

// Crtanje boida pomoću \_spriteBatch instance

\_boidManager.drawBoids(\_spriteBatch);

// Završetak \_spriteBatch instance

\_spriteBatch.end();

// Konačno iscrtavanje na ekran

\_spriteBatch.renderBatch();

// Oslobađanje prve teksture

glBindTexture(GL\_TEXTURE\_2D, 0);

// Završetak *OpenGL* programa

\_colorProgram.unUse();

// Zamenjivanje trenutnog bafera sa prethodnim - Prikaz na ekran

\_window.swapBuffer();

}

### 6.3. *BoidManager* класа

*BoidManager* класа је одговорна за стварање боида и држања референце на њих, као и неке помоћне функције које ће сами боиди користити.

### 6.3.1. *BoidManager.h*

#pragma once

#include <EFE/SpriteBatch.h>

#include "Boid.h"

#include <vector>

class BoidManager

{

public:

BoidManager();

~BoidManager();

// Postavljanje granica u kojima će se stvoriti boidi

void updateBounds(const int x, const int y);

// Stvaranje boida

void spawnBoids(int num, glm::vec2 pos);

// Crtanje boida

void drawBoids(efe::SpriteBatch &spriteBatch);

// Ažuriranje boida

void updateBoids();

// Dobijanje svih boida stvorenih ovom instancom

std::vector<Boid> getAllBoids() const { return \_boids; }

// Dobijanje svih boida blizu datog boida

std::vector<Boid\*> getNearbyBoids(const Boid &boid, float range);

private:

// Vektor boida

std::vector<Boid> \_boids;

// Granice stvaranja

int \_boundX;

int \_boundY;

};

### 6.3.2. *BoidManager.cpp*

#include "BoidManager.h"

#include <random>

#include <ctime>

#include <glm/glm.hpp>

#include <iostream>

BoidManager::BoidManager() :

\_boundX(0.0f),

\_boundY(0.0f)

{

}

BoidManager::~BoidManager()

{

}

// Ažuriranje granica stvaranja

void BoidManager::updateBounds(const int x, const int y)

{

srand(time(NULL));

\_boundX = x;

\_boundY = y;

}

// Stvaranje boida

void BoidManager::spawnBoids(int num, glm::vec2 pos)

{

// Keširanje maximuma random floata

static const float randMax = static\_cast<float>(RAND\_MAX);

// Keširanje maximuma random floata

static const float halfRandMax = randMax / 2;

// Postavljanje random smera

glm::vec2 vel;

vel.x = (rand() - halfRandMax) / randMax;

vel.y = (rand() - halfRandMax) / randMax;

vel = glm::normalize(vel);

// Ako je početna pozicija nula

if (pos.x < 0.01f && pos.y < 0.01f && pos.x > -0.01f && pos.y > -0.01f)

{

for (int i = 0; i < num; i++)

{

// Postavljanje random pozicije u granicama

glm::vec2 randPos;

randPos.x = (rand() - halfRandMax) / randMax \* \_boundX;

randPos.y = (rand() - halfRandMax) / randMax \* \_boundY;

// Stvaranje boida sa parametrima u konstruktoru

\_boids.emplace\_back(randPos, vel, 1.5f, \*this, \_boids.size());

}

}

// Ako je data tačna pozicija

else

{

for (int i = 0; i < num; i++)

{

// Stvaranje boida sa parametrima u konstruktoru

\_boids.emplace\_back(pos, vel, 1.5f, \*this, \_boids.size());

}

}

}

// Crtanje svih boida

void BoidManager::drawBoids(efe::SpriteBatch& spriteBatch)

{

for (int i = 0; i < \_boids.size(); i++)

{

\_boids[i].draw(spriteBatch);

}

}

// Ažuriranje boida

void BoidManager::updateBoids()

{

for (int i = 0; i < \_boids.size(); i++)

{

\_boids[i].update();

}

}

// Dobijanje svih boida blizu datog boida

std::vector<Boid\*> BoidManager::getNearbyBoids(const Boid& boid, float range)

{

std::vector<Boid\*> boids;

// setovanje pozicije trenutnog boida jer se ne menja

const glm::vec2 APos = boid.getPos();

for (int i = 0; i < \_boids.size(); i++)

{

// Ako je distance trenutnog boida manja od raspona,

// dodamo boid u vektor boida

if ((glm::distance(APos, \_boids[i].getPos()) <= range) &&

boid.getId() != \_boids[i].getId())

{

boids.push\_back(&\_boids[i]);

}

}

return boids;

}

## 6.4. *Boid* класа

У овој класи дефинишемо функционалност боида

### 6.4.1. *Boid.h*

#pragma once

#include <glm/glm.hpp>

#include <EFE/SpriteBatch.h>

class BoidManager;

class Boid

{

public:

// Konstruktor sa pozicijom, smerom, brzinom,

// referencom na Boid Manager instancu i id boida

Boid(glm::vec2 pos, glm::vec2 dir, float speed, BoidManager &boidManager, int id);

~Boid();

// Crtanje boida

void draw(efe::SpriteBatch& spriteBatch);

// Ažuriranje boida

void update();

// Dobijanje pozicije

glm::vec2 getPos() const { return \_position; }

// Dobijanje brzine

glm::vec2 getVel() const { return \_vel; }

// Dobijanje id-a

int getId() const { return \_id; }

private:

// Kretanje boida

void move();

// Računanje brzine

void calculateVelocity();

// Izbegavanje drugih boida

void avoidOtherBoids();

// Ograničavanje brzine

void limitSpeed();

// Letenje ka centru mase boida (kohezija)

void flyTowardsCenter();

// korigovati brzinu prema drugim boidima

void matchVelocity();

// Pointer na BoidManager instancu koja je stvorila ovaj boid

BoidManager\* \_boidManager;

// maksimalna brzina

float \_speed;

// trenutna brzina (i smer)

glm::vec2 \_vel;

// trenutna poticija

glm::vec2 \_position;

int \_id;

// Vektor pokazivača na trenutne obližnje boide

std::vector<Boid\*> \_nearbyBoids;

};

### 6.4.2. *Boid.cpp*

#include "Boid.h"

#include <EFE/ResourceManager.h>

#include <iostream>

#include "BoidManager.h"

#include <cmath>

Boid::Boid(glm::vec2 pos, glm::vec2 dir, float speed, BoidManager& boidManager, int id) :

\_speed(0.f),

\_vel(0.f),

\_position(0.f),

\_id(-1)

{

\_speed = speed;

\_vel = dir;

\_position = pos;

\_boidManager = &boidManager;

\_id = id;

}

Boid::~Boid()

{

}

// Crtanje boida

void Boid::draw(efe::SpriteBatch& spriteBatch)

{

// Setovanje UV koordinata

glm::vec4 uv(0, 0, 1, 1);

// Setovanje pozicije i skale

glm::vec4 posAndSize = glm::vec4(\_position.x, \_position.y, 15.f, 15.f);

// Setovanje teksture

static efe::GLTexture texture = efe::ResourceManager::getTexture("textures/boid.png");

// Setovanje boje. Bela boja znači da se uzima prava boja teksture

efe::Color color;

color.r = 255;

color.b = 255;

color.g = 255;

color.a = 255;

// Crtanje

spriteBatch.draw(posAndSize, uv, texture.id, color, 0);

}

// Ažuriranje boida

void Boid::update()

{

// Dobijanje svih boida blizu ovog boida

\_nearbyBoids = \_boidManager->getNearbyBoids(\*this, 400.f);

// Računanje brzine

calculateVelocity();

// Kretanje

move();

}

// Kretanje

void Boid::move()

{

\_position += \_vel \* \_speed;

}

// Računanje brzine

void Boid::calculateVelocity()

{

// Izbegavanje obližnjih boida - Razdvajanje

avoidOtherBoids();

// Letenje ka centru mase obližnjih boida - Kohezija

flyTowardsCenter();

// Korigovati brzinu prema obližnjim boidima - Poravnanje

matchVelocity();

// Ograničavanje brzine

limitSpeed();

}

// Izbegavanje obližnjih boida - Razdvajanje

void Boid::avoidOtherBoids()

{

// Koliko će se snažno odvajati

const static float avoidAmount = 0.1f;

glm::vec2 dt = {0.f, 0.f};

// Zbir razlike brzina ovog boida i bliznjih

for (int i = 0; i < \_nearbyBoids.size(); i++)

{

dt += \_vel - \_nearbyBoids[i]->getVel();

}

// Rezultat se dodaje u velocity

\_vel += dt \* avoidAmount;

}

// Ograničavanje brzine

void Boid::limitSpeed()

{

const float speed = sqrt(\_vel.x \* \_vel.x + \_vel.y \* \_vel.y);

if (speed > \_speed)

{

\_vel.x = (\_vel.x / speed) \* \_speed;

\_vel.y = (\_vel.y / speed) \* \_speed;

}

}

// Letenje ka centru mase obližnjih boida - Kohezija

void Boid::flyTowardsCenter()

{

if (\_nearbyBoids.size() == 0) return;

const static float centerAmount = 0.003f;

glm::vec2 center = { 0.f, 0.f };

// Zbir svih pozicija

for (int i = 0; i < \_nearbyBoids.size(); i++)

{

center += \_nearbyBoids[i]->getPos();

}

// srednja vrednost svih pozicija

center /= (float)\_nearbyBoids.size();

\_vel += (center - \_position) \* centerAmount;

}

// Korigovati brzinu prema obližnjim boidima - Poravnanje

void Boid::matchVelocity()

{

if (\_nearbyBoids.size() == 0) return;

const static float amount = 0.06f;

glm::vec2 avgVel = { 0.f, 0.f };

// Zbir svih brzina

for (int i = 0; i < \_nearbyBoids.size(); i++)

{

avgVel += \_nearbyBoids[i]->getVel();

}

// Srednja brzina

avgVel /= \_nearbyBoids.size();

\_vel += (avgVel - \_vel) \* amount;

}

# 7. Zaključak

Модел се заснива на симулацији понашања сваке птице независно. Самостално, птице покушавају да се држе заједно и да избегну сударе једна са другом и са другим објектима у свом окружењу. Симулирана јата изграђена на основу овог модела одговарају интуитивном појму посматрача о томе шта чини „кретање налик јату“. Међутим, тешко је објективно измерити колико су ове симулације валидне.

# 8. Reference

1. MakingGamesWithBen <https://www.youtube.com/playlist?list=PLSPw4ASQYyymu3PfG9gxywSPghnSMiOAW>
2. Ben Eater, Boids - [https://eater.net/boids](https://eater.net/boidshttps:/en.wikipedia.org/wiki/Boids)
3. Boids - <https://en.wikipedia.org/wiki/Boids>
4. Boids - <http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/>
5. OpenGL - [https://learn*OpenGL*.com/](https://learnopengl.com/)
6. GLM - <https://github.com/g-truc/glm>
7. SDL - <https://wiki.libsdl.org/>
8. SDL - <https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer>
9. PicoPNG - <https://lodev.org/lodepng/>