STM for Games

Code Listing

Author: Arkadiusz Bielecki
ID: C00139358
Date: 25.04.2014 Project Supervisor: Joseph Kehoe

Table of Contents

Data Structures

"STMForGames" solution – contains the main functionalities of the project, used to run various algorithms

"librstm" solution– contains all of the RSTM libraries, needed to run Software Transactional Memory

Classes Descriptions

ASearch.cpp - contains methods used to test A* search algorithm, various random objects throughout the game that need to be avoided - UNFINISHED

Flocking.cpp - contains methods used to test flocking algorithm - middle alien steers towards spaceship -> get average position of neighbour aliens (centre of mass) -> steer towards middle alien, and avoid collision

FPS.cpp - contains methods used to calculate frames per second

Game.cpp - contains all of the necessary code to run the game - initialize objects for according algorithms, update the window, run game loop, display health and frame rate

GameObjects.cpp - contains methods used for game objects - get position, draw, and update

GameObjectManager.cpp - contains methods used to get, remove and update game objects

Objects.cpp - contains methods used to load, draw and get position of the obstacles

Spaceship.cpp - contains methods used to get spaceship position and change its movement by pressing a keyboard key

STMforGames.cpp - defines the entry point for the console application

STMMain.cpp - contains methods used to interact with the user interface - like start algorithms, close the application

Swarming.cpp - contains methods used to test swarming algorithm - average position of all aliens (centre of mass) -> steer towards spaceship, avoid collision and steer in opposite direction if too close

Application Code

Header Files

ASearch.h

```
#pragma once
#include "stdafx.h"
#include "GameObjects.h"

class ASearch :
	public GameObjects
{
public:
	ASearch();
	~ASearch();

void Update(float elapsedTime);
void Draw(sf::RenderWindow& rw);

sf::Vector2f GetVelocity() const;
int GetAlienNo() const;

private:

	float speed;
	int alienNo;
	sf::Vector2f velocity;
};
```

Flocking.h

```
#pragma once
#include "stdafx.h"
#include "GameObjects.h"

class Flocking :
	public GameObjects
{
public:
	Flocking();
	~Flocking();

void Update(float elapsedTime);
```

```
void Draw(sf::RenderWindow& rw);

sf::Vector2f GetMVelocity() const;
sf::Vector2f GetCVelocity() const;
sf::Vector2f GetGVelocity() const;
sf::Vector2f Normalize(sf::Vector2f);
int GetAlienNo() const;

private:

        float maxSpeed;
        int alienNo;
        sf::Vector2f velocityMidd;
        sf::Vector2f velocityCurr;
        sf::Vector2f velocityGroup;
};
```

Game.h

```
#pragma once
#include "SFML/Window.hpp"
#include "SFML/Graphics.hpp"
#include "Spaceship.h"

#include "Flocking.h"
#include "Swarming.h"
#include "ASearch.h"

#include "Obstacle.h"

#include "GameObjectsManager.h"

class Game
{

public:
  static void Initialize(int type);
  static sf::RenderWindow& GetWindow();
  const static sf::Event& GetInput();
  const static GameObjectsManager& GetGameObjectsManager();
  const static int SCREEN_WIDTH = 800;
  const static int SCREEN_HEIGHT = 600;

private:
  static bool IsExiting();
  static void GameLoop();

  static Spaceship * player;

  static Flocking * alienFlocking;
  static Swarming * alienSwarming;
```

```
  static ASearch * alienASearch;

  enum GameState { Uninitialized, Playing, Exiting };

  static GameState _gameState;
  static sf::RenderWindow _mainWindow;
  static GameObjectsManager _gameObjectsManager;
};
```

GameObjects.h

```
#pragma once

class GameObjects
{
public:
        GameObjects();
        virtual ~GameObjects();

        virtual void Load(std::string filename);
        virtual void Draw(sf::RenderWindow & window);
        virtual void Update(float elapsedTime);

        virtual void SetPosition(float x, float y);

        static sf::Vector2f GetPosition2();

        virtual sf::Vector2f GetPosition() const;
        virtual float GetWidth() const;
        virtual float GetHeight() const;

        virtual sf::Rect<float> GetBoundingRect() const;
        virtual bool IsLoaded() const;

protected:
        sf::Sprite& GetSprite();
        //sf::Sprite  _sprite;

private:
        sf::Sprite  _sprite;
        sf::Texture _image;
        std::string _filename;
        bool _isLoaded;

};
```

GameObjectManager.h

```
#pragma once
#include "GameObjects.h"

class GameObjectsManager
{
```

```
public:
GameObjectsManager();
~GameObjectsManager();

void Add(int index, GameObjects* gameObject);
void Remove(int index);
int GetObjectsCount() const;
GameObjects* Get(int index) const;

void DrawAll(sf::RenderWindow& renderWindow);

void UpdateAll();

private:

std::map<int, GameObjects*> _gameObjects;

sf::Clock clock;

struct GameObjectsDeallocator
 {
   void operator()(const std::pair<int,GameObjects*> & p) const
   {
     delete p.second;
   }
 };
};
```

Obstacle.h

```
#pragma once
#include "stdafx.h"
#include "GameObjects.h"

class Obstacle :
        public GameObjects
{
public:
        Obstacle();
        ~Obstacle();

//void Update(float elapsedTime);
void Draw(sf::RenderWindow& rw);

sf::Vector2f GetPosition() const;
void Randomize();

private:

        sf::Vector2f position;
};
```

Spaceship.h

```
#pragma once
#include "stdafx.h"
#include "GameObjects.h"



class Spaceship :
	public GameObjects
{
public:
	Spaceship();
	~Spaceship();

int getHealth();
void reduceHealth(int reduceBy);

void Update(float elapsedTime);
void Draw(sf::RenderWindow& rw);

float GetLAcc() const;
float GetAngAcc() const;

private:

	float LAcc;
	float AngAcc;
	float MaxAngAcc;
	float MaxLAcc;
   float rotation;

	int health;
};
```

Swarming.h

```
#pragma once
#include "stdafx.h"
#include "GameObjects.h"

class Swarming :
	public GameObjects
{
public:
	Swarming();
	~Swarming();

void Update(float elapsedTime);
void Draw(sf::RenderWindow& rw);

sf::Vector2f GetVelocity() const;
sf::Vector2f Normalize(sf::Vector2f);
```

```
int GetAlienNo() const;

private:

	float maxSpeed;
	int alienNo;
	sf::Vector2f velocityV;
};
```

C++ Files

ASearch.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:		C00139358
 * Date: 27.04.2014
 *
 * ASearch.cpp - UNFINISHED
 * contains methods used to test A* search algorithm,
 * various random objects throughout the game that need to be avoided
 *
 */

// include statements
#include "stdafx.h"
#include "ASearch.h"
#include "Game.h"

ASearch::ASearch(): speed(80), alienNo(5)
{
	// load alien from image
	Load("img/alien.png");

	// throws error window if file loading fails
	assert(IsLoaded());
	//GetSprite().setOrigin(GetSprite().getLocalBounds().width /2, GetSprite().getLocalBounds().height / 2 - 50); //50
}

ASearch::~ASearch()
{
}

void ASearch::Draw(sf::RenderWindow & rw)
{
	GameObjects::Draw(rw);
}

sf::Vector2f ASearch::GetVelocity() const
{
	return velocity;
}
```

```
int ASearch::GetAlienNo() const
{
	return alienNo;
}

// update method runs in a loop and updates game objects
void ASearch::Update(float elapsedTime)
{
	// array of pointers to aliens
	ASearch** alien = new ASearch*[alienNo];

	// load spaceship
	Spaceship* spaceship = dynamic_cast<Spaceship*>(Game::GetGameObjectsManager().Get(0));

	// spaceship position vector
	sf::Vector2f spaceshipPos;

	// if spaceship is alive
	if(spaceship != NULL)
	{
		// get spaceship position
		spaceshipPos = spaceship->GetPosition();

		// get spaceship boundaries
		sf::Rect<float> spaceshipBounds = spaceship->GetBoundingRect();

		// check if spaceship intersects with other objects
		if(spaceshipBounds.intersects(GetBoundingRect()))
		{
			spaceship->reduceHealth(1);
		}

		// calculate alien speed
		float alienSpeed = speed*elapsedTime;

		// for loop goes through alien group and updates their position
		for (int index = 1; index < alienNo+1 ; index++)
		{
			// get game object
			alien[index] = dynamic_cast<ASearch*>(Game::GetGameObjectsManager().Get(index));

			// position vector
			sf::Vector2f alienGetPos = alien[index]->GetPosition();

			// calculate alien movement
			float dirx = (spaceshipPos.x - alienGetPos.x);
			float diry = (spaceshipPos.y - alienGetPos.y);
			float hyp = sqrt(dirx*dirx + diry*diry);
			dirx /= hyp;
			diry /= hyp;

			// update alien position
			alien[index]->SetPosition(alienGetPos.x + (dirx*alienSpeed)/alienNo, alienGetPos.y +
(diry*alienSpeed)/alienNo);
		}
	}
}
```

Flocking.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:		C00139358
 * Date: 27.04.2014
 *
 * Flocking.cpp -
 * contains methods used to test flocking algorithm -
 * middle alien steers towards spaceship -> get average position of neighbour aliens (centre of mass) ->
 * steer towards middle alien, and avoid collision
 */

// include statements
#include "stdafx.h"
#include "Flocking.h"
#include "Game.h"
// openMP to deal with parallelization
#include <omp.h>
// define math pi
#define PI 3.14159265

Flocking::Flocking(): maxSpeed(70), alienNo(8)
{

	// load alien image
	Load("img/alien.png");

	//throws error window if file loading fails
	assert(IsLoaded());

	//GetSprite().setOrigin(GetSprite().getLocalBounds().width /2, GetSprite().getLocalBounds().height / 2 - 50); //50
}

Flocking::~Flocking()
{
}

void Flocking::Draw(sf::RenderWindow & rw)
{
	GameObjects::Draw(rw);
}

sf::Vector2f Flocking::GetMVelocity() const
{
	return velocityMidd;
}

sf::Vector2f Flocking::GetCVelocity() const
{
	return velocityCurr;
}

sf::Vector2f Flocking::GetGVelocity() const
```

```
{
	return velocityGroup;
}

// method to return a normalized vector
sf::Vector2f Flocking::Normalize(sf::Vector2f velocity)
{
	 float length = sqrt((velocity.x * velocity.x) + (velocity.y * velocity.y));
	 if (length != 0) {
		velocity.x /= length;
		velocity.y /= length;
	 }
	 return velocity;
}

// return the number of aliens
int Flocking::GetAlienNo() const
{
	return alienNo;
}

// update method runs in a loop and updates game objects
void Flocking::Update(float elapsedTime)
{
	// array of pointers to aliens
	Flocking** alien = new Flocking*[alienNo];

	// direction boolean - positive is forward, negative backward
	bool direction = true;

	// load spaceship
	Spaceship* spaceship = dynamic_cast<Spaceship*>(Game::GetGameObjectsManager().Get(0));

	// get spaceship position
	sf::Vector2f spaceshipPos;

	// if spaceship is alive
	if(spaceship != NULL) {

		// get spaceship position
		spaceshipPos = spaceship->GetPosition();

		// get spaceship boundaries
		sf::Rect<float> spaceshipBounds = spaceship->GetBoundingRect();

		// check if spaceship intersects with other objects
		if(spaceshipBounds.intersects(GetBoundingRect())) {
			spaceship->reduceHealth(1);
		}

		// get middle alien
		int middleAlienInt = alienNo/2 +1;

		// middle alien movement
		if (alienNo < 3){
			middleAlienInt = 1;
```

```
}

// get object (middle alien)
alien[middleAlienInt] = dynamic_cast<Flocking*>(Game::GetGameObjectsManager().Get(middleAlienInt));

// position vector
sf::Vector2f middleAlienPos = alien[middleAlienInt]->GetPosition();

// head towards spaceship
float dirx = (spaceshipPos.x - middleAlienPos.x);
float diry = (spaceshipPos.y - middleAlienPos.y);

// initial velocity
sf::Vector2f velocityMidd(dirx, diry);

// initial acceleration
sf::Vector2f accelerationV(0.0f, 0.0f);
sf::Vector2f separationV(0.0f, 0.0f);

int neighbourCount = 0;

// values for for loops
int index = 1;
int index2 = 1;

// openMP statement to deal with parallelization
#pragma omp parallel for schedule(dynamic, alienNo+1) ordered firstprivate(index2) lastprivate(index)

// alien group movement
for (index = 1 ; index < alienNo+1 ; index++) {

	// get current object (alien)
	alien[index] = dynamic_cast<Flocking*>(Game::GetGameObjectsManager().Get(index));

	// current alien position
	sf::Vector2f currentAlienPos = alien[index]->GetPosition();

	// head towards middle spaceship
	float dirx = (middleAlienPos.x - currentAlienPos.x);
	float diry = (middleAlienPos.y - currentAlienPos.y);

	// initial velocity
	sf::Vector2f velocityGroup(dirx, diry);//(dirx+100, diry+100);
	sf::Vector2f velocityDiff(dirx, diry);
	sf::Vector2f velocitySepp(0.0f, 0.0f);

	// alignment & cohesion - line up & steer towards average position of neibours
	for (index2 = 1 ; index2 < alienNo+1 ; index2++) {
		if (index != index2) {

		// get next object (alien)
		alien[index2] = dynamic_cast<Flocking*>(Game::GetGameObjectsManager().Get(index2));

		// next alien position
		sf::Vector2f nextAlienPos = alien[index2]->GetPosition();
```

```
		// Calculate the difference between the two objects.
		sf::Vector2f differenceV = currentAlienPos - nextAlienPos;
		float distance1 = sqrt(differenceV.x*differenceV.x + differenceV.y*differenceV.y);

		// check for nearby aliens and steer towards them
		float neighbourDist = 100;

		// initial velocity if alien has no neighbours
		if (neighbourCount == 0) velocityGroup = velocityCurr;

		// Check of the objects are closer that the collision distance.
		if (distance1 < neighbourDist) {

				neighbourCount++;

				// Calculate the difference between the two objects
				sf::Vector2f differenceV2 = middleAlienPos - currentAlienPos;

				// when a neighbor is found, the position of the neighbour is added to
the vector
				velocityGroup.x += (velocityCurr.x + differenceV2.x)*index*maxSpeed;
				velocityGroup.y += (velocityCurr.y + differenceV2.y)*index*maxSpeed;
				velocityGroup.x /= neighbourCount;
				velocityGroup.y /= neighbourCount;

				float separation = 40;

				// Check of the objects are closer that the collision distance
				if (distance1 < separation) {
					velocitySepp.x = currentAlienPos.x - nextAlienPos.x;
					velocitySepp.y = currentAlienPos.y - nextAlienPos.y;
					direction = false;
					}
			}
		}
	}

	// calculate velocity - if the vector is too big normalize it, if its too small maximize the speed
	velocityGroup += accelerationV*elapsedTime;
	if (((velocityGroup.x*velocityGroup.x)+(velocityGroup.y*velocityGroup.y))>maxSpeed*maxSpeed) {
		velocityGroup = Flocking::Normalize(velocityGroup);
		velocityGroup *= maxSpeed;
	}

	// calculate velocity - if the vector is too big normalize it, if its too small maximize the speed
	velocityDiff += accelerationV*elapsedTime;
	if (((velocityDiff.x*velocityDiff.x)+(velocityDiff.y*velocityDiff.y))>maxSpeed*maxSpeed) {
		velocityDiff = Flocking::Normalize(velocityDiff);
		velocityDiff *= maxSpeed;
	}

	// if current alien within collision distance - change its direction
	if(direction == false && index != middleAlienInt) {
		velocityGroup = velocitySepp;
		velocityGroup += accelerationV*elapsedTime;
```

```
                                        velocityGroup *= -1.0f;
                                        if
(((velocityGroup.x*velocityGroup.x)+(velocityGroup.y*velocityGroup.y))>maxSpeed*maxSpeed) {
                                        velocityGroup = Flocking::Normalize(velocityGroup);
                                        velocityGroup *= maxSpeed;
                                        }
                                        velocityGroup += velocityDiff;
                                        velocityGroup *= -1.0f;
                                }

                                // if middle alien within collision distance - change its direction
                                if(direction == false && index == middleAlienInt) {
                                        velocityMidd += accelerationV*elapsedTime;
                                        if (((velocityMidd.x*velocityMidd.x)+(velocityMidd.y*velocityMidd.y))>maxSpeed*maxSpeed)
{
                                        velocityMidd = Flocking::Normalize(velocityMidd);
                                        velocityMidd *= maxSpeed;
                                        }
                                }

                                else {
                                velocityMidd += accelerationV*elapsedTime;
                                if (((velocityMidd.x*velocityMidd.x)+(velocityMidd.y*velocityMidd.y))>maxSpeed*maxSpeed) {
                                        velocityMidd = Flocking::Normalize(velocityMidd);
                                        velocityMidd *= maxSpeed;
                                        }
                                }

                                // middle alien movement calculation
                                alien[middleAlienInt]->SetPosition(middleAlienPos.x + velocityMidd.x*elapsedTime/alienNo,
                                        middleAlienPos.y + velocityMidd.y*elapsedTime/alienNo);

                                // current alien movement calculation
                                if (index != middleAlienInt) {
                                alien[index]->SetPosition(currentAlienPos.x + (velocityGroup.x)*elapsedTime/alienNo
                                        + (velocityDiff.x)*elapsedTime/alienNo,
                                currentAlienPos.y + (velocityGroup.y)*elapsedTime/alienNo
                                         + (velocityDiff.y)*elapsedTime/alienNo);
                                }
                        }
                }
}
```

FPS.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:        C00139358
 * Date: 27.04.2014
 *
 * FPS.cpp -
 * contains methods used to calculate frames per second
 */

// include statements
#include "stdafx.h"
```

```
class FPS
{
public:
	// constructor with initialization
	FPS() : mFrame(0), mFps(0) {}

	// get the current FPS and return it
	const unsigned int getFPS() const { return mFps; }

private:
	unsigned int mFrame;
	unsigned int mFps;
	sf::Clock mClock;

public:
	// constantly updates the values
	void update()
	{
		if(mClock.getElapsedTime().asSeconds() >= 1.f)
		{
			mFps = mFrame;
			mFrame = 0;
			mClock.restart();
		}
		++mFrame;
	}
};
```

Game.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:      C00139358
 * Date: 27.04.2014
 *
 * Game.cpp -
 * contains all of the necessary code to run the game - initialize objects for according algorithms, update the window, run game loop,
 * display health and frame rate
 */

// include statements
#include "stdafx.h"
#include "Game.h"
#include "Spaceship.h"
#include "FPS.cpp"
#include <iostream>
#include <sstream>
// to get the vector class definition
#include <vector>
using std::vector;
using namespace std;



// objects from other classes
Spaceship * Game::player;
```

```
Flocking * Game::alienFlocking;
Swarming * Game::alienSwarming;
ASearch * Game::alienASearch;

// the number of obstacles
int obstNo = 15;

sf::Sprite background;
sf::Font font;
FPS fps;

// Converts an int into a string
static inline std::string int2Str(int x)
{
	std::stringstream type;
	type << x;
	return type.str();
}

// initializes the game
void Game::Initialize(int type)
{
	if(_gameState != Uninitialized)
		return;
	_mainWindow.setFramerateLimit(50);
	_mainWindow.create(sf::VideoMode(SCREEN_WIDTH, SCREEN_HEIGHT),"STM Game");

	// selected algorithm type
	if (type == 1) // flocking selected
	{
		Flocking * AlienObj = new Flocking();
		alienFlocking = AlienObj;

		// get the number of aliens
		int alienNo = alienFlocking->GetAlienNo();

		// array of pointers to aliens
		Flocking** alien = new Flocking*[alienNo];

		// initialize objects
		for (int index = 1; index < alienNo+1 ; index++)
		{
		alien[index] = new Flocking();

		// set the initial position
		alien[index]->SetPosition(50*index,(SCREEN_HEIGHT-600));
		_gameObjectsManager.Add(index, alien[index]);
		}
	}

	else if (type == 2) // swarming selected
	{
		Swarming * AlienObj = new Swarming();
		alienSwarming = AlienObj;

		// get the number of aliens
		int alienNo = alienSwarming->GetAlienNo();
```

```
		// array of pointers to aliens
		Swarming** alien = new Swarming*[alienNo];

		// initialize objects
		for (int index = 1; index < alienNo+1 ; index++)
		{
		alien[index] = new Swarming();

		// set the initial position
		alien[index]->SetPosition(50*index,(SCREEN_HEIGHT-600));
		_gameObjectsManager.Add(index, alien[index]);
		}
}

else if (type == 3) // A* search algorithm - not complete
{
		ASearch * AlienObj = new ASearch();
		alienASearch = AlienObj;

		int alienNo = alienASearch->GetAlienNo();

		// array of pointers to obstacles
		Obstacle** obstacle = new Obstacle*[obstNo];

		for (int index = alienNo+1 ; index < alienNo+obstNo ; index++)
		{
		obstacle[index] = new Obstacle();

		// random numbers for obstacles position
		int randX = rand() % (Game::SCREEN_WIDTH-200);
		int randY = rand() % (Game::SCREEN_HEIGHT-200);

		// add game obstacle
		obstacle[index]->SetPosition(randX+50, randY+50);
		_gameObjectsManager.Add(index, obstacle[index]);
		}

		//printf("AlienNo Game.cpp: %i /n", alienNo);

		// array of pointers to aliens
		ASearch** alien = new ASearch*[alienNo];

		for (int index = 1; index < alienNo+1 ; index++)
		{
		alien[index] = new ASearch();
		// initialize objects
		alien[index]->SetPosition(20+20*index,(SCREEN_HEIGHT-600));
		_gameObjectsManager.Add(index, alien[index]);
		}

}

// spaceship object and initial position
Spaceship *spaceship = new Spaceship();
spaceship->SetPosition((SCREEN_WIDTH/2),(SCREEN_HEIGHT-100));
```

```
	_gameObjectsManager.Add(0, spaceship);

	player = spaceship;

	// initialize stars in the background
	sf::Texture stars;

	if (!stars.loadFromFile("img/stars.png"))
	{
		// error...
		printf("File not found");
	}

	// set repeated background
	stars.setRepeated(true);
	background.setTexture(stars);
	background.setTextureRect(sf::IntRect(0,0,Game::SCREEN_WIDTH,Game::SCREEN_HEIGHT));

	// Load font from a file
	if (!font.loadFromFile("font/sansation.ttf"))
	{
		printf("Error loading font\n");
	}

	// start game loop
	_gameState = Game::Playing;

	while(!IsExiting())
	{
		GameLoop();
	}

	_mainWindow.close();
}


bool Game::IsExiting()
{
	if(_gameState == Game::Exiting)
		return true;
	else
		return false;
}

sf::RenderWindow& Game::GetWindow()
{
	return _mainWindow;
}

const sf::Event& Game::GetInput()
{
	sf::Event currentEvent;
	_mainWindow.pollEvent(currentEvent);
	return currentEvent;
}
```

```
const GameObjectsManager& Game::GetGameObjectsManager()
{
	return Game::_gameObjectsManager;
}

void Game::GameLoop()
{

	sf::Event currentEvent;
	_mainWindow.pollEvent(currentEvent);

	// health
	string healStr = int2Str(player->getHealth());

	// draw health on the screen
	sf::Text healthText;
	healthText.setFont(font);
	healthText.setCharacterSize(20);
	healthText.setStyle(sf::Text::Bold);
	healthText.setColor(sf::Color::White);
	healthText.setPosition(0,0);
	healthText.setString("Health: "  + healStr);

	// draw frames per second on the screen
	sf::Text fpsText;
	fpsText.setFont(font);
	fpsText.setCharacterSize(18);
	fpsText.setColor(sf::Color::White);
	fpsText.setPosition(Game::SCREEN_WIDTH-80,0);
	//int health = player->getHealth();
	fpsText.setString("FPS: ");
	std::ostringstream ss;

	// if player has no more health display a game over message and remove the spaceship
	if (player->getHealth() < 1)
		{
			healthText.setCharacterSize(80);
			healthText.setPosition(SCREEN_WIDTH/2-250, SCREEN_HEIGHT/2-50);
			healthText.setString("GAME OVER");
			_gameObjectsManager.Remove(0);
			_mainWindow.clear();
		}

	switch(_gameState)
	{
	case Game::Playing:
		{
			_mainWindow.clear(sf::Color(sf::Color(0,0,0)));
			_mainWindow.draw(background);
			_mainWindow.draw(healthText);

			// update frames per second
			fps.update();
			ss << fps.getFPS();
			fpsText.setString("FPS: " + ss.str());
			_mainWindow.draw(fpsText);

			_gameObjectsManager.UpdateAll();
```

```
                                _gameObjectsManager.DrawAll(_mainWindow);

                                // Finally, display rendered frame on screen
                                _mainWindow.display();

                                if(currentEvent.type == sf::Event::Closed) _gameState =
                                        Game::Exiting;

                                if(currentEvent.type == sf::Event::KeyPressed)
                                {
                                        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape)) _mainWindow.close();
                                }
                                break;
                        }
                }
}

Game::GameState Game::_gameState = Uninitialized;
sf::RenderWindow Game::_mainWindow;
GameObjectsManager Game::_gameObjectsManager;
```

GameObjects.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:      C00139358
 * Date: 27.04.2014
 *
 * GameObjects.cpp -
 * contains methods used for game objects - get position, draw, and update
 */

// include statements
#include "StdAfx.h"
#include "GameObjects.h"

GameObjects::GameObjects()
        : _isLoaded(false)
{
}

GameObjects::~GameObjects()
{
}

void GameObjects::Load(std::string filename)
{
        if(_image.loadFromFile(filename) == false)
        {
                _filename = "";
                _isLoaded = false;
        }
        else
        {
                _filename = filename;
                _sprite.setTexture(_image);
```

```
                _isLoaded = true;
        }
}

void GameObjects::Draw(sf::RenderWindow & renderWindow)
{
        if(_isLoaded)
        {
                renderWindow.draw(_sprite);
        }
}

void GameObjects::Update(float elapsedTime)
{
}

void GameObjects::SetPosition(float x, float y)
{
        if(_isLoaded)
        {
                _sprite.setPosition(x,y);
        }
}

sf::Vector2f GameObjects::GetPosition() const
{
 if(_isLoaded)
 {
  return _sprite.getPosition();
 }
 return sf::Vector2f();
}

float GameObjects::GetHeight() const
{
        return _sprite.getLocalBounds().height;
}

float GameObjects::GetWidth() const
{
        return _sprite.getLocalBounds().width;
}

sf::Rect<float> GameObjects::GetBoundingRect() const
{
        return _sprite.getGlobalBounds();
}

sf::Sprite& GameObjects::GetSprite()
{
 return _sprite;
}

bool GameObjects::IsLoaded() const
{
 return _isLoaded;
}
```

GameObjectManager.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:        C00139358
 * Date: 27.04.2014
 *
 * GameObjectManager.cpp -
 * contains methods used to get, remove and update game objects
 */

// include statements
#include "stdafx.h"
#include "GameObjectsManager.h"

GameObjectsManager::GameObjectsManager()
{
}

GameObjectsManager::~GameObjectsManager()
{
 std::for_each(_gameObjects.begin(),_gameObjects.end(),GameObjectsDeallocator());
}

void GameObjectsManager::Add(int index, GameObjects* gameObject)
{
 _gameObjects.insert(std::pair<int,GameObjects*>(index,gameObject));
}

void GameObjectsManager::Remove(int index)
{
 std::map<int, GameObjects*>::iterator results = _gameObjects.find(index);
 if(results != _gameObjects.end() )
 {
  delete results->second;
  _gameObjects.erase(results);
 }
}

GameObjects* GameObjectsManager::Get(int index) const
{
 std::map<int, GameObjects*>::const_iterator results = _gameObjects.find(index);
 if(results == _gameObjects.end() )
  return NULL;
  return results->second;
}

int GameObjectsManager::GetObjectsCount() const
{
 return _gameObjects.size();
}

void GameObjectsManager::DrawAll(sf::RenderWindow& renderWindow)
{
 std::map<int,GameObjects*>::const_iterator itr = _gameObjects.begin();
```

```
 while(itr != _gameObjects.end())
 {
  itr->second->Draw(renderWindow);
  itr++;
 }
}

void GameObjectsManager::UpdateAll()
{
	std::map<int,GameObjects*>::const_iterator itr = _gameObjects.begin();
	float timeDelta = clock.restart().asSeconds();

	while(itr != _gameObjects.end())
	{
		itr->second->Update(timeDelta);
		itr++;
	}

}
```

Obstacle.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:	C00139358
 * Date: 27.04.2014
 *
 * Objects.cpp -
 * contains methods used to load, draw and get position of the obstacles
 */

// include statements
#include "stdafx.h"
#include "Obstacle.h"
#include "Game.h"

Obstacle::Obstacle()
{
	// load obstacle
	Load("img/obstacle.png");

	//throws error window if file loading fails
	assert(IsLoaded());
}

Obstacle::~Obstacle()
{
}

void Obstacle::Draw(sf::RenderWindow & rw)
{
	GameObjects::Draw(rw);
}

sf::Vector2f Obstacle::GetPosition() const
```

```
{
	return position;
}
```

Spaceship.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:		C00139358
 * Date: 27.04.2014
 *
 * Spaceship.cpp -
 * contains methods used to get spaceship position and change its movement by pressing a keyboard key
 */

// include statements
#include "stdafx.h"
#include "Spaceship.h"
#include "Game.h"
#define _USE_MATH_DEFINES
#include <math.h>

Spaceship::Spaceship(): MaxAngAcc(60), MaxLAcc(150), LAcc(0), AngAcc(0), health(500), rotation(0)
{
	// load spaceship image
	Load("img/spaceship.png");

	//throws error window if file loading fails
	assert(IsLoaded());

	GetSprite().setOrigin(GetSprite().getLocalBounds().width /2, GetSprite().getLocalBounds().height / 2 - 50);
}

Spaceship::~Spaceship()
{
}

void Spaceship::Draw(sf::RenderWindow & rw)
{
	GameObjects::Draw(rw);
}

float Spaceship::GetLAcc() const
{
	return LAcc;
}

float Spaceship::GetAngAcc() const
{
	return AngAcc;
}

// get spaceship health
int Spaceship::getHealth()
{
	return health;
}
```

```
// reduce spaceship health
void Spaceship::reduceHealth(int reduceBy)
{
	health -= reduceBy;
	//return health;
}

void Spaceship::Update(float elapsedTime)
{

	// get spaceship position
	sf::Vector2f position = GetSprite().getPosition();

	// // get spaceship orientation

	float orientation = GetSprite().getRotation(); // angle

	// keyboard listeners - change value accordingly to the keyboard press
	if(sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
	{
		LAcc+=15;
	}

	if(sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
	{
		LAcc-=15;
	}

	if(sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
	{
		AngAcc-=10;
	}
	if(sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
	{
		AngAcc+=10;
	}

	rotation = AngAcc *elapsedTime;
	float speed = LAcc *elapsedTime;

	// convert oriantation to radians
	float orientInRad = (M_PI/ 180) * (orientation -90);

	float movementX = cos(orientInRad)*speed;
	float movementY = sin(orientInRad)*speed;

	// get spaceship boundaries
	float localbx = GetSprite().getLocalBounds().width/8;
	float localby = GetSprite().getLocalBounds().height/8;

	// adjust speed and direction according to the spaceship position and game window boundaries
	if(position.x < localbx)
	{
		LAcc = 0;
		movementX = 1.5;
```

```
	}

	if(position.x > (Game::SCREEN_WIDTH - localbx))
	{
		LAcc = 0;
		movementX = -1.5;
	}

	if(position.y < localby)
	{
		LAcc = 0;
		movementY = 1.5;
	}

	if(position.y > (Game::SCREEN_HEIGHT - localby))
	{
		LAcc = 0;
		movementY = -1.5;
	}

	// move the spaceship
	GetSprite().move(movementX, movementY);

	// rotate the spaceship
	GetSprite().rotate(rotation);

	// adjust the acceleration according to the maximum values
	if(LAcc > 0)
		LAcc--;

	if(LAcc < 0)
		LAcc++;

	if(LAcc > MaxLAcc)
		LAcc = MaxLAcc;

	if(LAcc < -100)
		LAcc = -100;



	if(AngAcc > 0)
		AngAcc--;

	if(AngAcc < 0)
		AngAcc++;

	if(AngAcc > MaxAngAcc)
		AngAcc = MaxAngAcc;

	if(AngAcc < -50)
		AngAcc = -50;
}
```

STMforGames.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:		C00139358
 * Date: 27.04.2014
 *
 * STMforGames.cpp -
 * defines the entry point for the console application
 */


#include "stdafx.h"

int _tmainMenu(int argc, _TCHAR* argv[])
{
	return 0;
}
```

STMMain.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:		C00139358
 * Date: 27.04.2014
 *
 * STMMain.cpp -
 * contains methods used to interact with the user interface - like start algorithms, close the application
 */


// include statements
#include "stdafx.h"
#include <SFML/Graphics.hpp>
#include "Game.h"
#include <iostream>
#include <atomic>

// OpemMP to deal with parallelization
#include <omp.h>

// RSTM library
#include "stm\rstm.hpp"
#include "stm\api\rstm_api.hpp"

std::atomic<bool> ready (false);
std::atomic_flag winner = ATOMIC_FLAG_INIT;

void insert(int);

int main()
{
	insert(5);

	// render main app window
   sf::RenderWindow window(sf::VideoMode(1024, 600), "STM for Games");
   sf::Texture mainMenu;
	if (!mainMenu.loadFromFile("img/mainMenu.png"))
```

```
		{
			// error...
			printf("File not found");
		}
	sf::Sprite sprite;
	sprite.setTexture(mainMenu);

	// MainMenu open
while (window.isOpen())
{
		// event listener
		sf::Event event;
		window.draw(sprite);
		window.display();

		while (window.pollEvent(event))
		{
  window.display();

		// mouse click listener
		if(sf::Mouse::isButtonPressed(sf::Mouse::Left))
		{
			// transform the mouse position from window coordinates to world coordinates
			sf::Vector2f mouse = window.mapPixelToCoords(sf::Mouse::getPosition(window));
			float mouseY = mouse.y;

			if (mouseY > 300)
			{
				window.close();
			}

			if (mouseY < 300)
			{
				while (window.isOpen()) // STMMenu window
				{
				sf::Event event;
				while (window.pollEvent(event))
				{
				// transform the mouse position from window coordinates to world coordinates
				sf::Vector2f mouse = window.mapPixelToCoords(sf::Mouse::getPosition(window));
				float mouseY = mouse.y;

				sf::Texture STMMainMenu;
				if (!STMMainMenu.loadFromFile("img/STMMainMenu.png"))
					{
					// error...
						printf("File not found");
					}
				sf::Sprite sprite;
				sprite.setTexture(STMMainMenu);
				window.clear();
				window.draw(sprite);
				window.display();

				// mouse click listener
				if(sf::Mouse::isButtonPressed(sf::Mouse::Left))
				{
					if (mouseY > 5 && mouseY < 150)
```

```
                                        {
                                                printf("FLOCKING");
                                                window.close();
                                                //Game::Initialize(1);
                                                Game::Initialize(1);


                                        }
                                        else if (mouseY > 150 && mouseY < 300)
                                        {
                                                printf("SWARMING");
                                                window.close();
                                                Game::Initialize(2);
                                        }
                                        else if (mouseY > 300 && mouseY < 450)
                                        {
                                                printf("A*");
                                                window.close();
                                                Game::Initialize(3);
                                        }
                                        else if (mouseY > 450)
                                        {
                                                window.close();
                                        }
                                }
                                if (event.type == sf::Event::Closed)
                                window.close();
                                }
                                }
                        }
                        printf("%f", mouseY);
                }
                if (event.type == sf::Event::Closed)
                 window.close();
                }

    }
    return 0;
}

// used to implement stm algorithm - not finished

void insert(int val) {
        /*int max = 10;*/
        //stm::
        rstm::tx_alloc;
        stm::tx_alloc;
        //rstm::tx_alloc;
        ATOMIC_INT_LOCK_FREE;
        {
        val = 4;
        } ATOMIC_INT_LOCK_FREE;

    //BEGIN_TRANSACTION {
    //   /* while (val > 1) {
    //       val--;
    //  }*/
    //} END_TRANSACTION;
}
```

Swarming.cpp

```
/*
 * Author Arkadiusz Bielecki
 * ID:		C00139358
 * Date: 27.04.2014
 *
 * Swarming.cpp -
 * contains methods used to test swarming algorithm -
 * average position of all aliens (centre of mass) -> steer towards spaceship,
 * avoid collision and steer in opposite direction if too close
 */

// include statements
#include "stdafx.h"
#include "Swarming.h"
#include "Game.h"
// openMP to deal with parallelization
#include <omp.h>
//#include <atomic>
#include "stm\rstm.hpp"
#include "stm\api\rstm_api.hpp"

// define math pi
#define PI 3.14159265

Swarming::Swarming(): maxSpeed(70), alienNo(15)
{
	// load alien image
	Load("img/alien.png");

	//throws error window if file loading fails
	assert(IsLoaded());
	//GetSprite().setOrigin(GetSprite().getLocalBounds().width /2, GetSprite().getLocalBounds().height / 2 - 50); //50
}

Swarming::~Swarming()
{
}

void Swarming::Draw(sf::RenderWindow & rw)
{
	GameObjects::Draw(rw);
}

sf::Vector2f Swarming::GetVelocity() const
{
	return velocityV;
}

// method to return a normalized vector
sf::Vector2f Swarming::Normalize(sf::Vector2f velocity)
{
	 float length = sqrt((velocity.x * velocity.x) + (velocity.y * velocity.y));
	 if (length != 0) {
		velocity.x /= length;
		velocity.y /= length;
	 }
```

```
		return velocity;
}

// return the number of aliens
int Swarming::GetAlienNo() const
{
		return alienNo;
}

//void insert(int val) {
//		BEGIN_TRANSACTION;
//		const node* previous = head->open_RO();
//		// points to sentinel node
//		const node* current = previous->next->open_RO();
//		// points to first real node
//		while (current != NULL) {
//		if (current->val >= val) break;
//		previous = current;
//		current = current->next->open_RO();
//		}
//		if (!current || current->val > val) {
//		node* n = new node(val, current->shared());
//		// uses Object<T>::operator new
//		previous->open_RW()->next = new Shared<node>(n);
//		}
//		END_TRANSACTION;
//}

// update method runs in a loop and updates game objects
void Swarming::Update(float elapsedTime)
{
		// array of pointers to aliens
		Swarming** alien = new Swarming*[alienNo];

		// direction boolean - positive is forward, negative backward
		bool direction = true;

		// load spaceship
		Spaceship* spaceship = dynamic_cast<Spaceship*>(Game::GetGameObjectsManager().Get(0));

		// get spaceship position
		sf::Vector2f spaceshipPos;

		// if spaceship is alive
		if(spaceship != NULL) {
			// get spaceship position
			spaceshipPos = spaceship->GetPosition();

			// get spaceship boundaries
			sf::Rect<float> spaceshipBounds = spaceship->GetBoundingRect();

			// check if spaceship intersects with other objects
			if(spaceshipBounds.intersects(GetBoundingRect())) { //(GetPosition().x + moveByX + (GetSprite().GetSize().x
/2),GetPosition().y + (GetSprite().GetSize().y /2) + moveByY))
				spaceship->reduceHealth(1);
			}
```

```
// initial velocities
sf::Vector2f velocityGroup(0.0f, 0.0f);
sf::Vector2f velocityAvg(0.0f, 0.0f);
sf::Vector2f centerOfMassV(0.0f, 0.0f);
sf::Vector2f centerOfMassPos(0.0f, 0.0f);
sf::Vector2f velocitySepp(0.0f, 0.0f);
sf::Vector2f velocityRandom(0.0f, 0.0f);
sf::Vector2f velocityToShip(0.0f, 0.0f);

// initial acceleration
sf::Vector2f accelerationV(0.0f, 0.0f);

// stm code
/*stm::init;
rstm::thr_init;

class ìnteger : public stm::Object  {
        int currentAlienPosX;
        };
BEGIN_TRANSACTION
        currentAlienPosX++;
END_TRANSACTION*/

// openMP statement to deal with parallelization
#pragma omp parallel for schedule(dynamic, alienNo+1)

// alien group movement - first part get average velocity
for (int index = 1; index < alienNo+1 ; index++) {

        // get current object (alien)
        alien[index] = dynamic_cast<Swarming*>(Game::GetGameObjectsManager().Get(index));

        // get current alien position
        sf::Vector2f currentAlienPos = alien[index]->GetPosition();

        // add every alien to the average position of the group
        velocityGroup += currentAlienPos;
}

// calculate average velocity
velocityAvg.x = velocityGroup.x/alienNo;
velocityAvg.y = velocityGroup.y/alienNo;

// calculate average center of mass position
centerOfMassPos.x = velocityGroup.x/alienNo;
centerOfMassPos.y = velocityGroup.y/alienNo;

// values for for loops
int index = 1;
int index2 = 1;

// openMP statement to deal with parallelization
#pragma omp parallel for schedule(dynamic, alienNo+1) ordered firstprivate(index2) lastprivate(index)

// alien group movement - second part move towards average velocity
for (int index = 1; index < alienNo+1 ; index++) {
```

```
                        // get current object (alien)
                        alien[index] = dynamic_cast<Swarming*>(Game::GetGameObjectsManager().Get(index));

                        // current alien position
                        sf::Vector2f currentAlienPos = alien[index]->GetPosition();

                        // head towards average velocity
                        float dirx = (velocityAvg.x - currentAlienPos.x);
                        float diry = (velocityAvg.y - currentAlienPos.y);

                        // set directions
                        sf::Vector2f centerOfMassV(dirx, diry);
                        dirx = (spaceshipPos.x - centerOfMassPos.x);
                        diry = (spaceshipPos.y - centerOfMassPos.y);
                        sf::Vector2f velocityDiff(dirx, diry);
                        sf::Vector2f velocityToShip(dirx, diry);

                        // for loop to get every neighbour
                        for (int index2 = 1; index2 < alienNo+1 ; index2++) {
                                if (index != index2) {

                                // get current object (alien)
                                alien[index2] =
dynamic_cast<Swarming*>(Game::GetGameObjectsManager().Get(index2));

                                // current alien position
                                sf::Vector2f nextAlienPos = alien[index2]->GetPosition();

                                // Calculate the difference between the two objects.
                                sf::Vector2f differenceV = currentAlienPos - nextAlienPos;
                                float distance1 = sqrt(differenceV.x*differenceV.x + differenceV.y*differenceV.y);

                                // game objects separation
                                float separation = 50;

                                // random y velocity for each alien in a swarm
                                if (distance1 < 300) {
                                        velocityRandom.y = spaceshipPos.y - (rand() % 1000);
                                        }

                                // Check of the objects are closer that the collision distance
                                if (distance1 < separation) {
                                        velocitySepp.x = currentAlienPos.x - nextAlienPos.x;
                                        velocitySepp.y = currentAlienPos.y - nextAlienPos.y;
                                        direction = false;
                                        }
                                }
                        }

                        // calculate velocity - if the vector is too big normalize it, if its too small maximize the speed
                        velocityToShip += accelerationV*elapsedTime;
                        if (((velocityToShip.x*velocityToShip.x)+(velocityToShip.y*velocityToShip.y))>maxSpeed*maxSpeed)
{
                                velocityToShip = Swarming::Normalize(velocityToShip);
                                velocityToShip *= maxSpeed;
```

```
                }

                // calculate velocity - if the vector is too big normalize it, if its too small maximize the speed
                velocitySepp += accelerationV*elapsedTime;
                if (((velocitySepp.x*velocitySepp.x)+(velocitySepp.y*velocitySepp.y))>maxSpeed*maxSpeed) {
                        velocitySepp = Swarming::Normalize(velocitySepp);
                        velocitySepp *= maxSpeed;
                }

                // calculate velocity - if the vector is too big normalize it, if its too small maximize the speed
                velocityRandom += velocityRandom*elapsedTime;
                if
(((velocityRandom.x*velocityRandom.x)+(velocityRandom.y*velocityRandom.y))>maxSpeed*maxSpeed) {
                        velocityRandom = Swarming::Normalize(velocityRandom);
                        velocityRandom *= maxSpeed;
                }

                // calculate velocity - if the vector is too big normalize it, if its too small maximize the speed
                velocityDiff += accelerationV*elapsedTime;
                if (((velocityDiff.x*velocityDiff.x)+(velocityDiff.y*velocityDiff.y))>maxSpeed*maxSpeed) {
                        velocityDiff = Swarming::Normalize(velocityDiff);
                        velocityDiff *= maxSpeed;
                }

                // if current alien within collision distance - change its direction
                if(direction == false) {
                        centerOfMassV = velocitySepp;
                        centerOfMassV += accelerationV*elapsedTime;
                        centerOfMassV *= -1.0f;
                        if
(((centerOfMassV.x*centerOfMassV.x)+(centerOfMassV.y*centerOfMassV.y))>maxSpeed*maxSpeed) {
                        centerOfMassV = Swarming::Normalize(centerOfMassV);
                        centerOfMassV *= maxSpeed;
                        }
                        centerOfMassV += velocitySepp;
                        centerOfMassV *= -1.0f;
                }

                else {
                centerOfMassV += accelerationV*elapsedTime;
                if
(((centerOfMassV.x*centerOfMassV.x)+(centerOfMassV.y*centerOfMassV.y))>maxSpeed*maxSpeed) {
                        centerOfMassV = Swarming::Normalize(centerOfMassV);
                        centerOfMassV *= maxSpeed;
                        }
                }

                // current alien movement calculation
                alien[index]->SetPosition(currentAlienPos.x + (centerOfMassV.x)*elapsedTime/alienNo
                        + (velocityToShip.x)*elapsedTime/alienNo,
                currentAlienPos.y + (centerOfMassV.y)*elapsedTime/alienNo
                        + (velocityToShip.y)*elapsedTime/alienNo
                        + (velocityRandom.y)*elapsedTime/alienNo);
        }
    }
}
```