МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

**«Московский технический университет связи и информатики»**

Кафедра «Информатика»

Курсовая работа

**«Взлёт и посадка самолёта»**

по дисциплине

**Технологии программирования**

Выполнил: студент гр. БЭИ2202

Кулешов А. С.

Вариант 16.

Проверил: доц. Волков А.И.,

ст. преп. Загвоздкина А.В.

Москва, 2024 г.

**1 Задание**

Разработать приложение для моделирования взлёта и посадки самолёта в аэропорту. В этом приложении необходимо:

1. Создать класс, содержащий данные о некотором объекте (базовом) из заданной предметной области. Эти данные должны быть спрятаны от непосредственного доступа (инкапсулированы). Членами класса должны быть не менее 4-х свойств, 2-х методов, 1-го конструктора и 2-х событий. Дополнительно создать классы, описывающие вспомогательные объекты, необходимые для моделирования;
2. В главной программе создать экземпляры созданных классов и продемонстрировать возможности изменения свойств объектов, использования их методов и обработки реализованных событий (разработав соответствующие методы обработки событий, объявленных в классах). При реализации событий необходимо обратить особое внимание на организацию событийного взаимодействия объектов;
3. Создать еще один класс путем наследования имеющегося класса. В новом классе переопределить хотя бы один из методов базового класса;
4. В главной программе создать экземпляры базового и дочернего классов (а также вспомогательных классов) и продемонстрировать их возможности по изменению свойств, использованию методов и обработки событий. Организовать взаимодействие созданных объектов между собой или с другими вспомогательными объектами с использованием пользовательских событий;
5. Реализовать валидацию ввода, а также запретить взаимодействие с программой, приводящее к непредвиденному состоянию.

Список свойств на формах:

|  |  |
| --- | --- |
| **PlaneSimulator** | |
| **button1** | |
| Font | Microsoft Sans Serif; 14,25pt |
| Location | 872; 792 |
| Size | 112; 45 |
| Text | - |
| **button2** | |
| Font | Microsoft Sans Serif; 14,25pt |
| Location | 872; 891 |
| Size | 112; 45 |
| Text | + |

|  |  |
| --- | --- |
| **label1** | |
| Font | Microsoft Sans Serif; 14,25pt |
| Location | 914; 849 |
| Text | 3 |

|  |  |
| --- | --- |
| **Button3** | |
| Font | Microsoft Sans Serif; 14,25pt |
| Location | 992; 814 |
| Size | 225; 108 |
| Text | Запустить симуляцию |

**2 Логическая структура программы**

Поля:

|  |  |
| --- | --- |
| Тип поля | Название поля |

|  |  |
| --- | --- |
| **RotationalObject** | |
| int | size |
| String | ImagePath |
| Direction | CurrentDirection |

|  |  |
| --- | --- |
| **PlannedMovementObject** | |
| float | Speed |
| List<Point> | Movement\_path |
| int | progress |
| int | Mid\_point |

|  |  |
| --- | --- |
| **Plane** | |
| float | XVelocity |
| float | YVelocity |
| float | XAcceleration |
| float | YAcceleration |
| float | ReqXVelocity |
| float | TopXVelocity |
| float | TopYVelocity |
| String | ImagePath |
| String | state |
| bool | loaded |

|  |  |
| --- | --- |
| **Bus** | |
| int | ticks\_waiting |
| String | ImagePath |

|  |  |
| --- | --- |
| **CargoPlane** | |
| String | ImagePath |

|  |  |
| --- | --- |
| **CargoCar** | |
| String | ImagePath |

Методы:

|  |  |  |  |
| --- | --- | --- | --- |
| Возвращаемый тип | Название метода | Параметры метода | Описание метода |

|  |  |  |  |
| --- | --- | --- | --- |
| **RotationalObject** | | | |
| void | ChangeImage | String ImagePath, int sizeX, int sizeY | Изменяет отображаемое изображение на изображение из файла |
| void | InitInsect | Form^ World, Point location | Инициализирует объект на форме |
| void | Fly | Direction direction | Выполняет перемещение объекта по форме, поворачивая его в нужном направлении |

|  |  |  |  |
| --- | --- | --- | --- |
| **PlannedMovementObject** | | | |
| bool | isFinished |  | Выполняет проверку окончания запланированного маршрута |
| bool | isMidPoint |  | Выполняет проверку состояния маршрута на нахождении в середине пути |
| void | move |  | Перемещает объект далее по маршруту |

|  |  |  |  |
| --- | --- | --- | --- |
| **Plane** | | | |
| void | prepare\_to\_fly |  | Подготавливает самолёт к взлёту. Подгружает нужный маршрут, задаёт нужную скорость |
| void | prepare\_to\_land | List<Point> Movement\_path\_ | Подготавливает самолёт к посадке. Подгружает нужный маршрут, задаёт нужную скорость |
| void | prepare\_to\_wait | List<Point> Movement\_path\_ | Подготавливает самолёт к ожиданию загрузки. Самолёт начинает ждать (слушать) окончания загрузки машиной. |
| void | tick |  | Обновляет состояние самолёта |
| void | Fly\_VPP |  | Производит взлёт самолёта |
| void | land |  | Производит посадку самолёта |

|  |  |  |  |
| --- | --- | --- | --- |
| **Bus** | | | |
| void | tick |  | Обновляет состояние автобуса |

Конструкторы:

|  |  |
| --- | --- |
| Класс конструктора | Параметры конструктора |
| RotationalObject | Form^ World, Point location |
| PlanedMovementObject |  |
| Plane | Form^ World, Point location, int size\_ |
| Bus | Form^ World, Point location, int size\_ |
| CargoPlane | Form^ World, Point location, int size\_ |
| CargoCar | Form^ World, Point location, int size\_ |

События:

|  |  |  |
| --- | --- | --- |
| Тип обработчика события | Название события | Описание события |

|  |  |  |
| --- | --- | --- |
| **Plane** | | |
| FlewAwayHandler | FlewAway | Событие происходит при выходе самолёта с зоны прорисовки экрана. |
| LandedHandler | Landed | Событие происходит при посадке самолёта в соответствующий ему ангар. |
| WaitingForCargoHandler | WaitingForCargo | Событие происходит при попадании самолёта на зону загрузки. |

**3 Схемы алгоритмов**

a

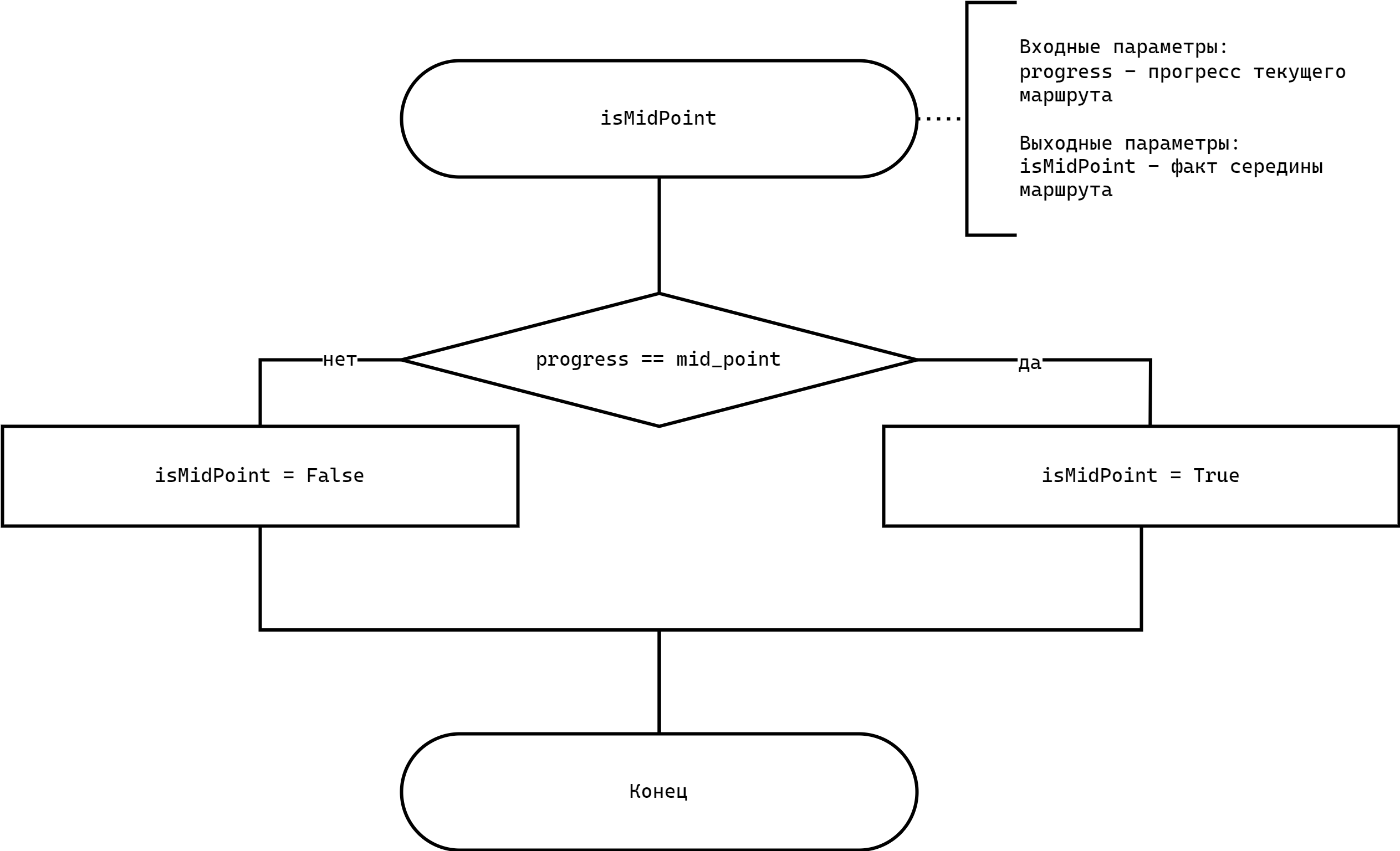


Рисунок 1 – Схема метода isMidPoint

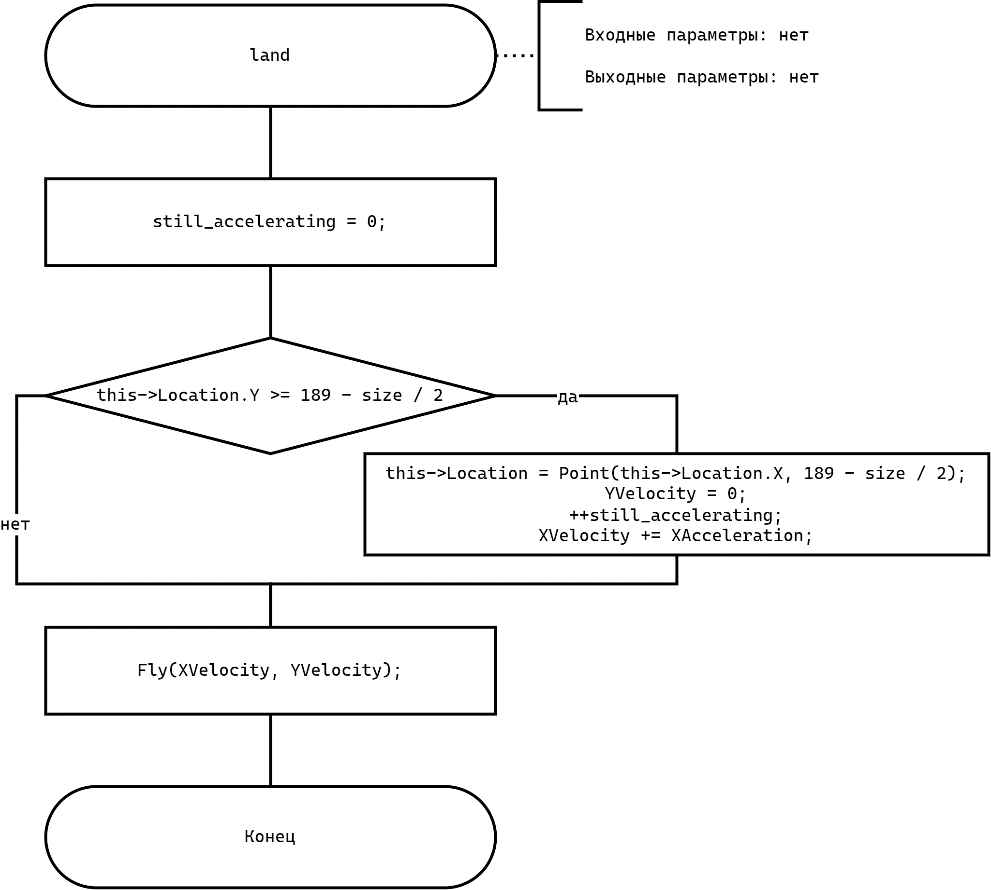


Рисунок 2 – Схема метода land

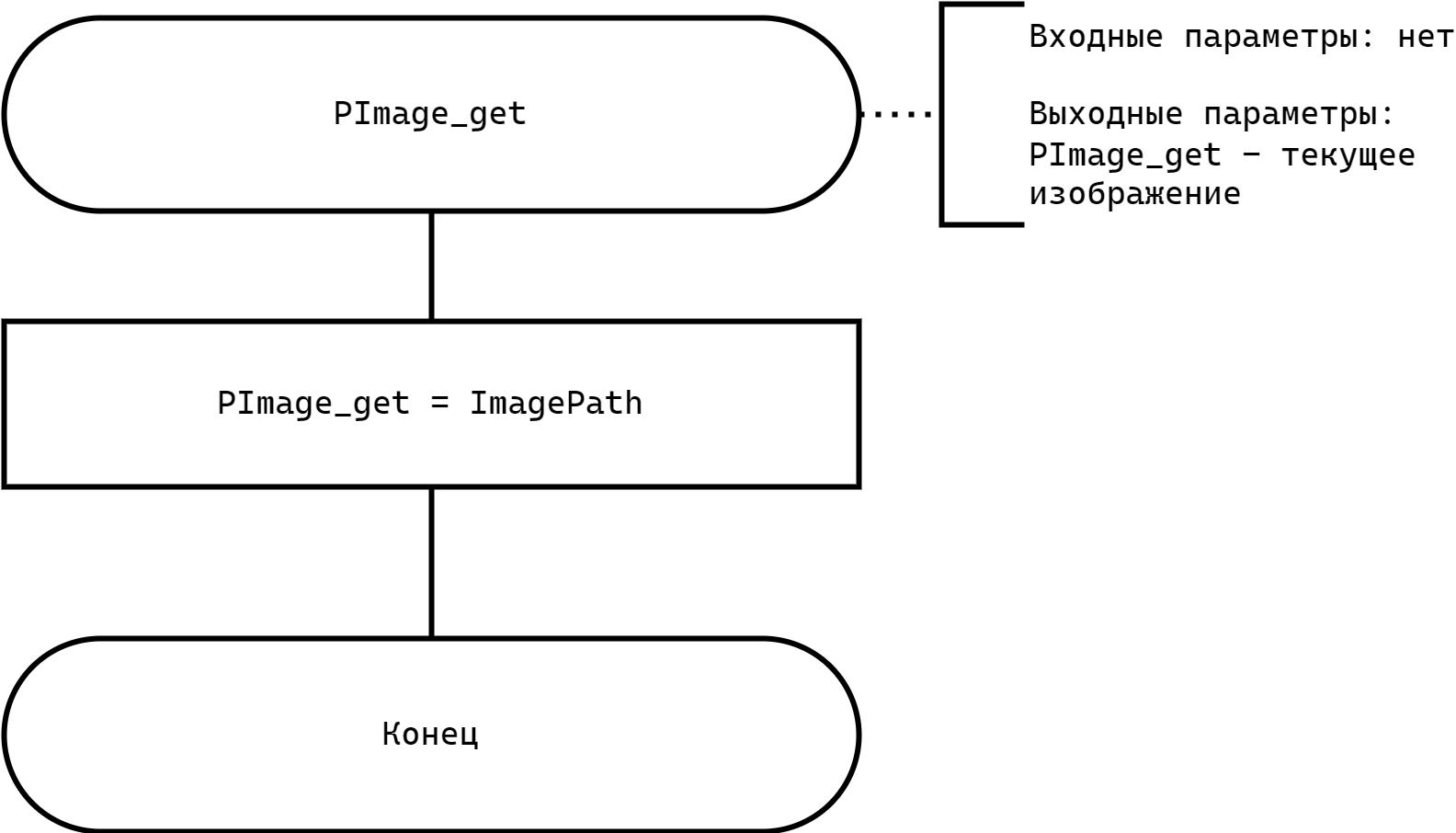


Рисунок 3 – Схема свойства PImage, get

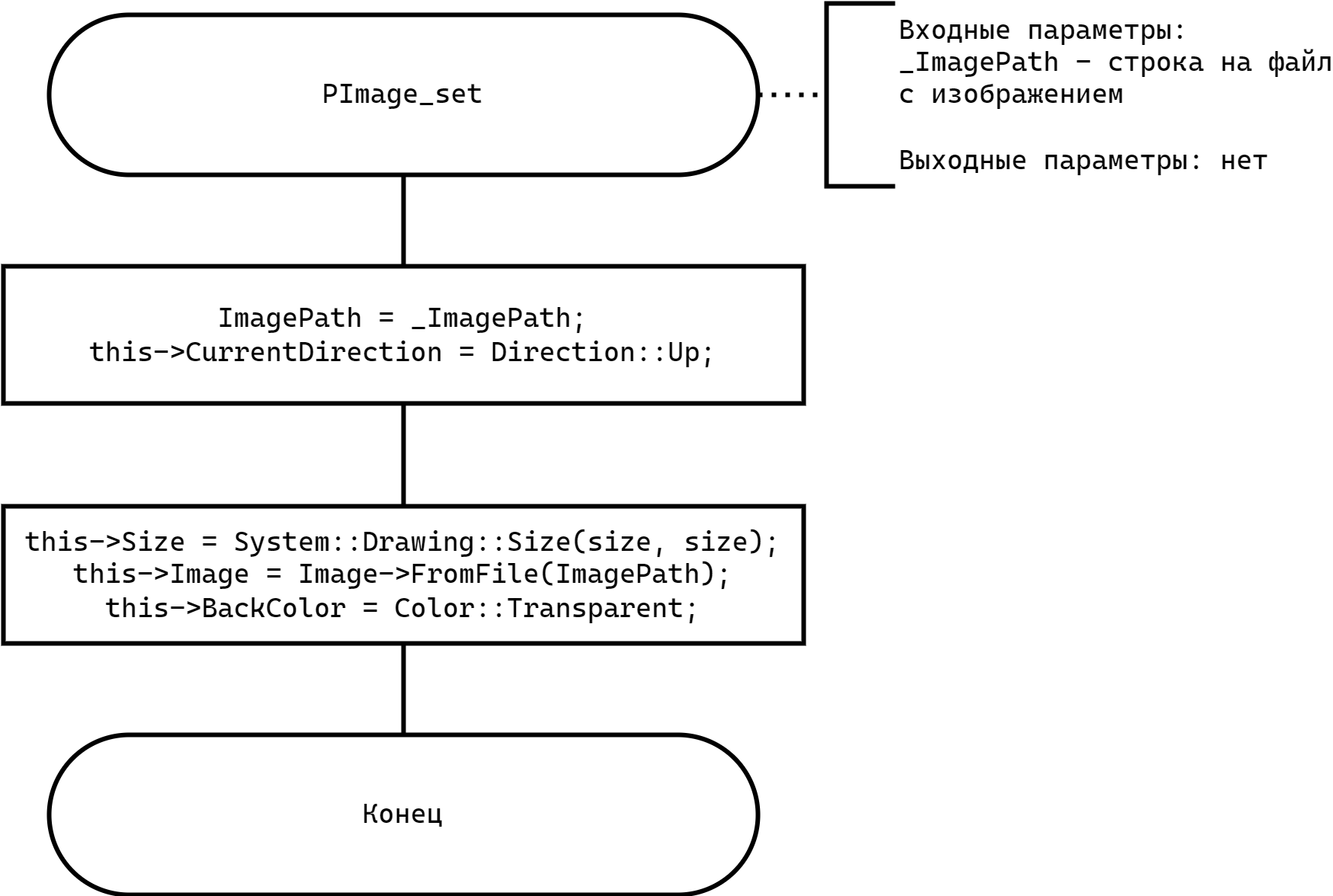


Рисунок 4 – Схема свойства PImage, set

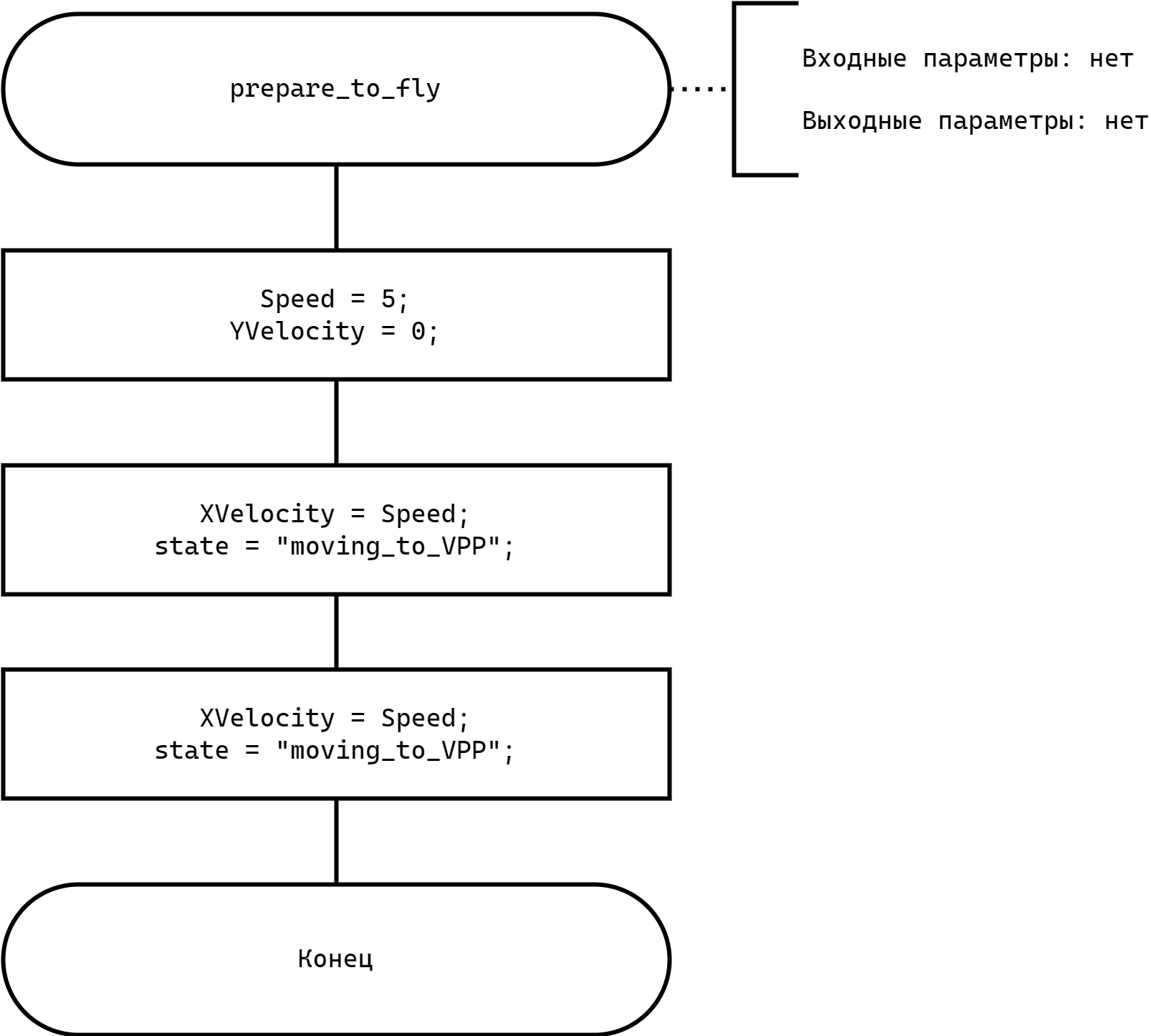


Рисунок 5 – Схема метода prepare\_to\_fly

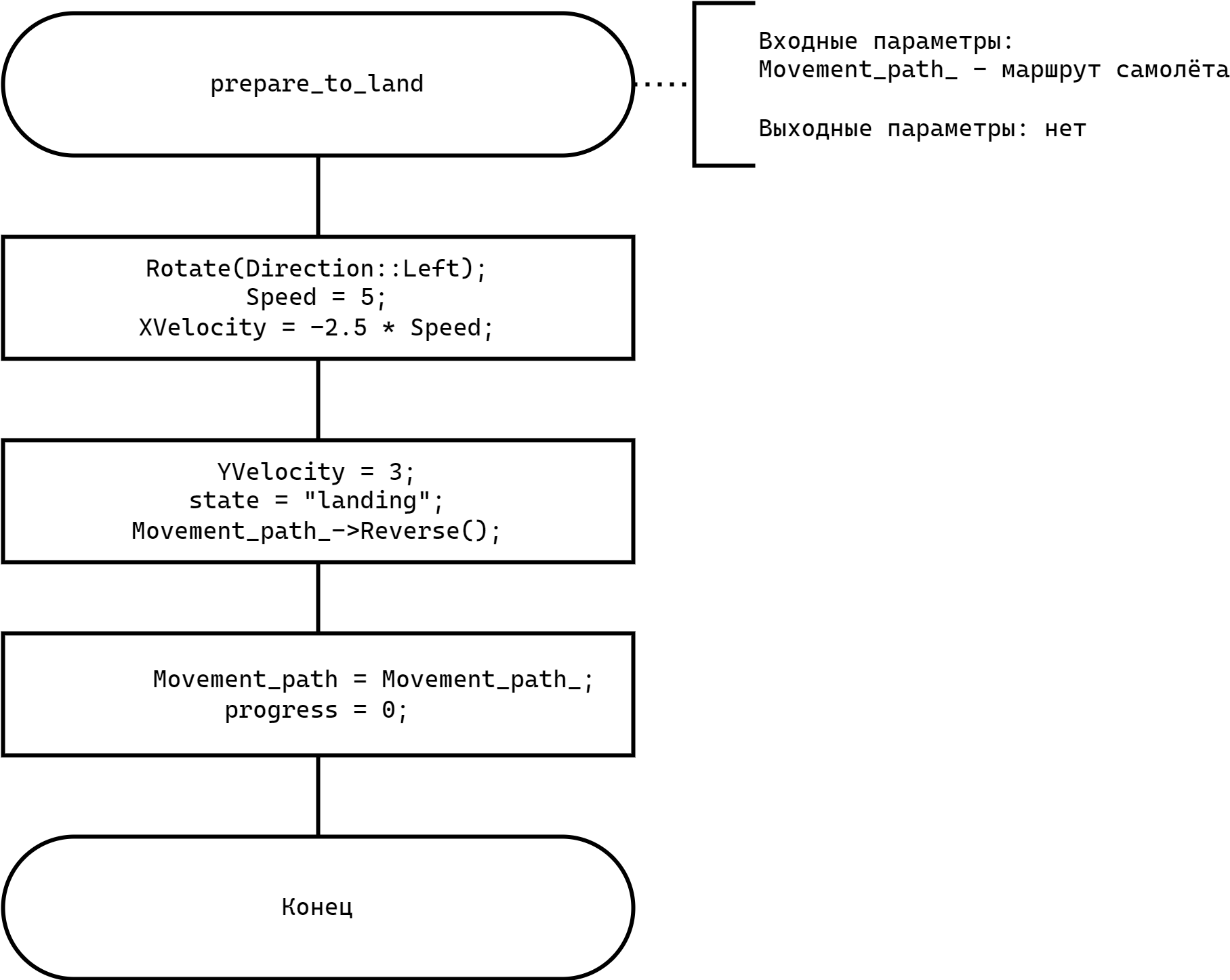


Рисунок 6 – Схема метода prepare\_to\_land

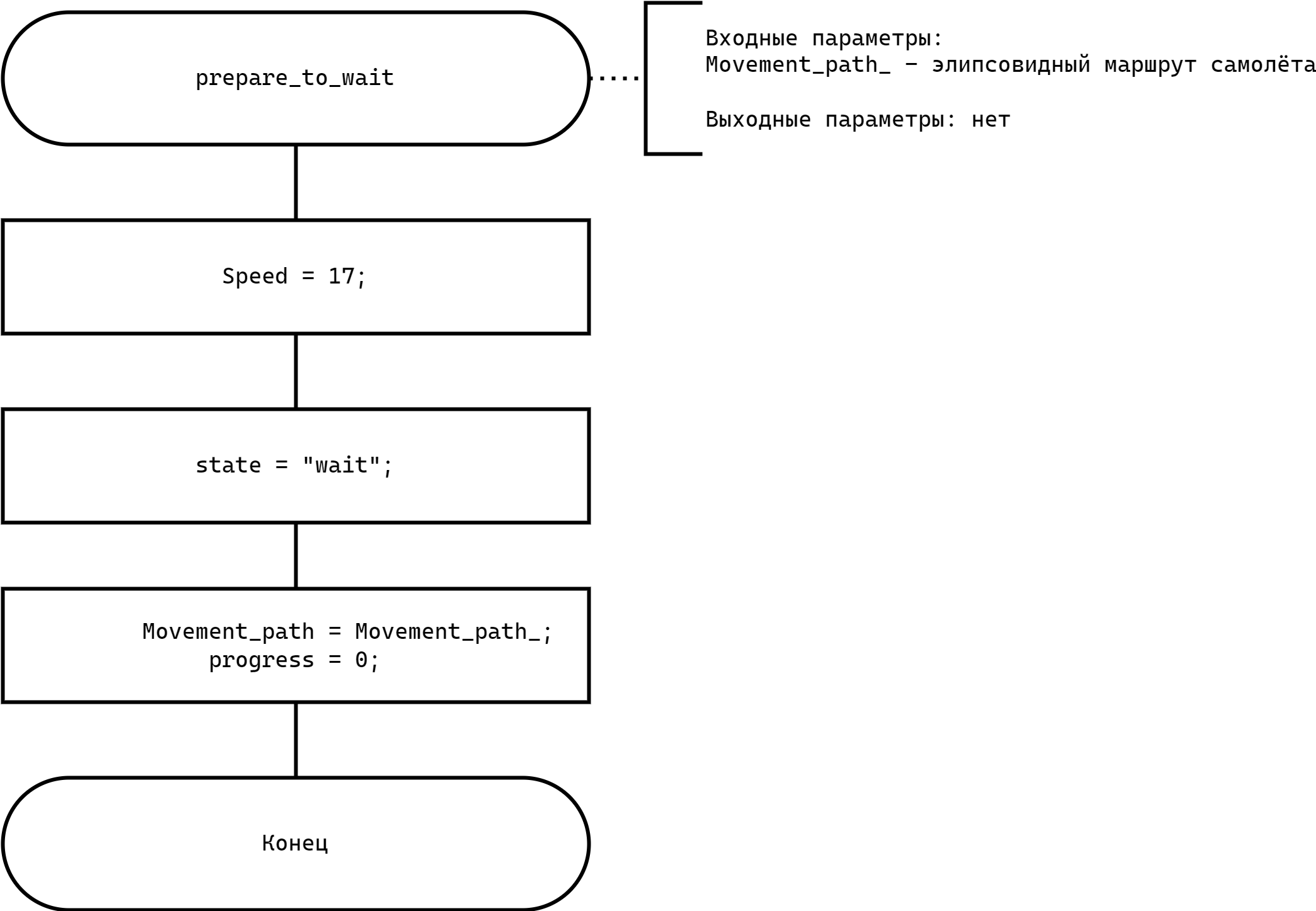


Рисунок 7 – Схема метода prepare\_to\_wait

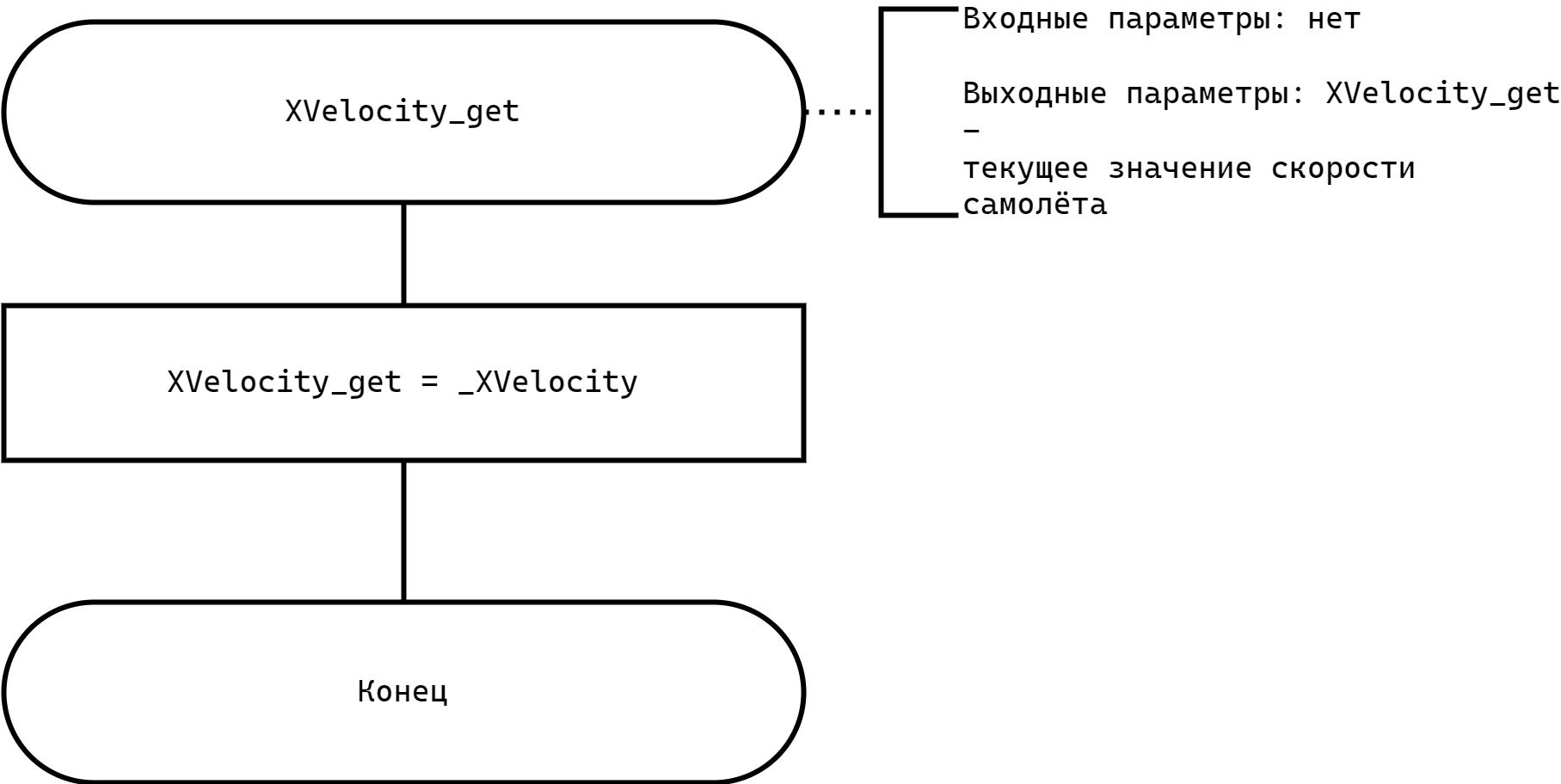


Рисунок 8 – Схема свойства XVelocity, get

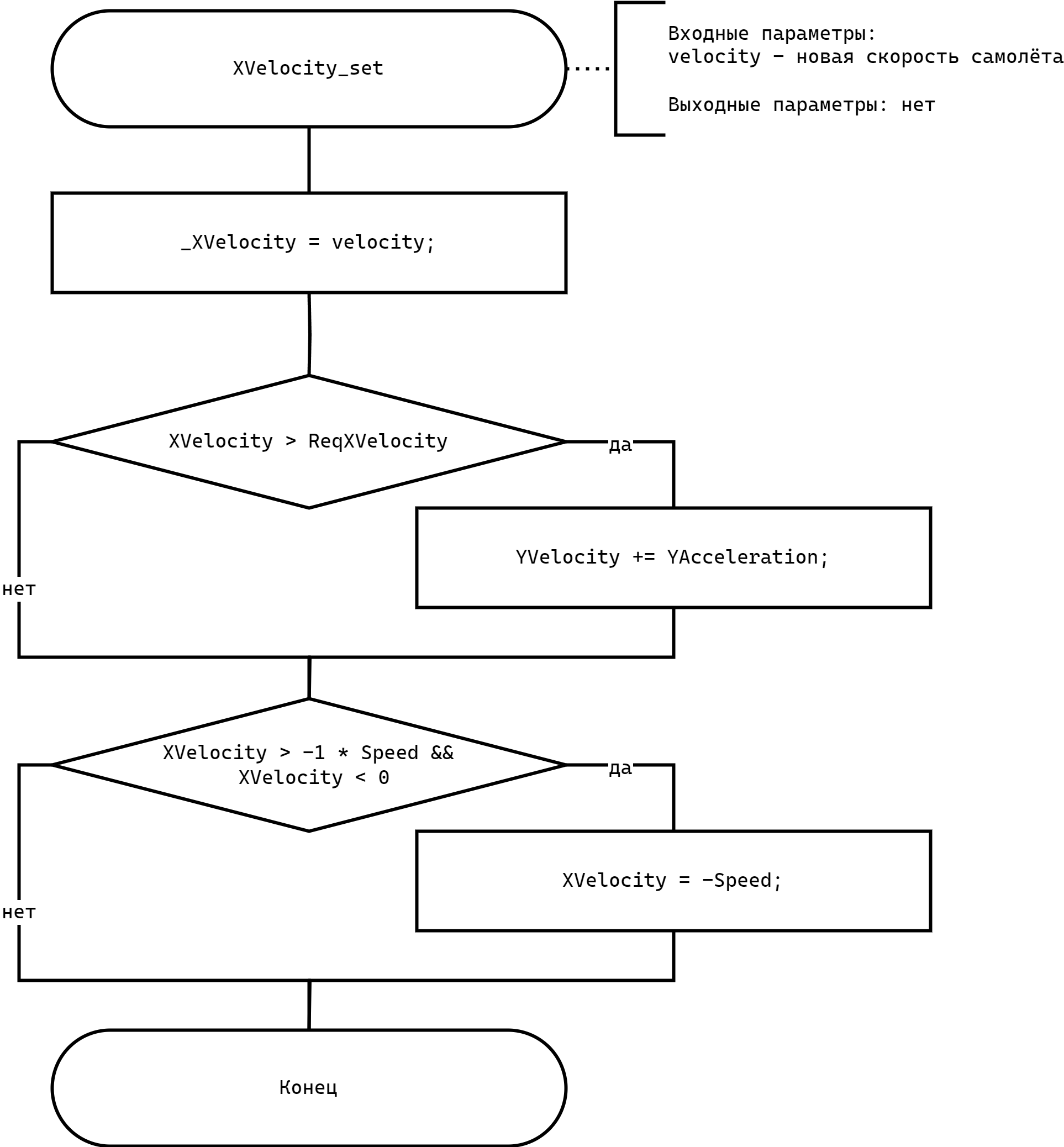


Рисунок 9 – Схема свойства XVelocity, set

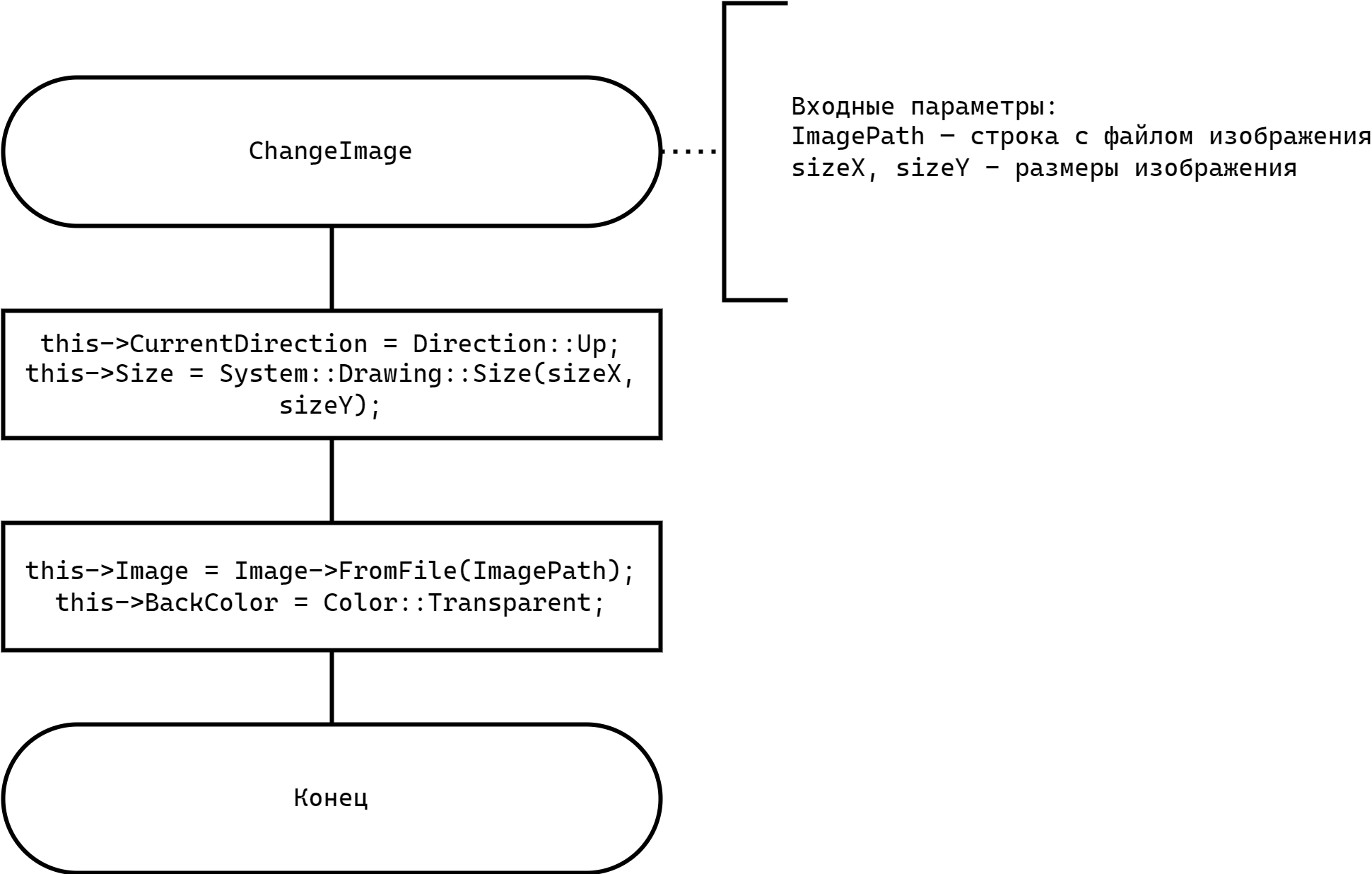


Рисунок 9 – Схема метода ChangeImage

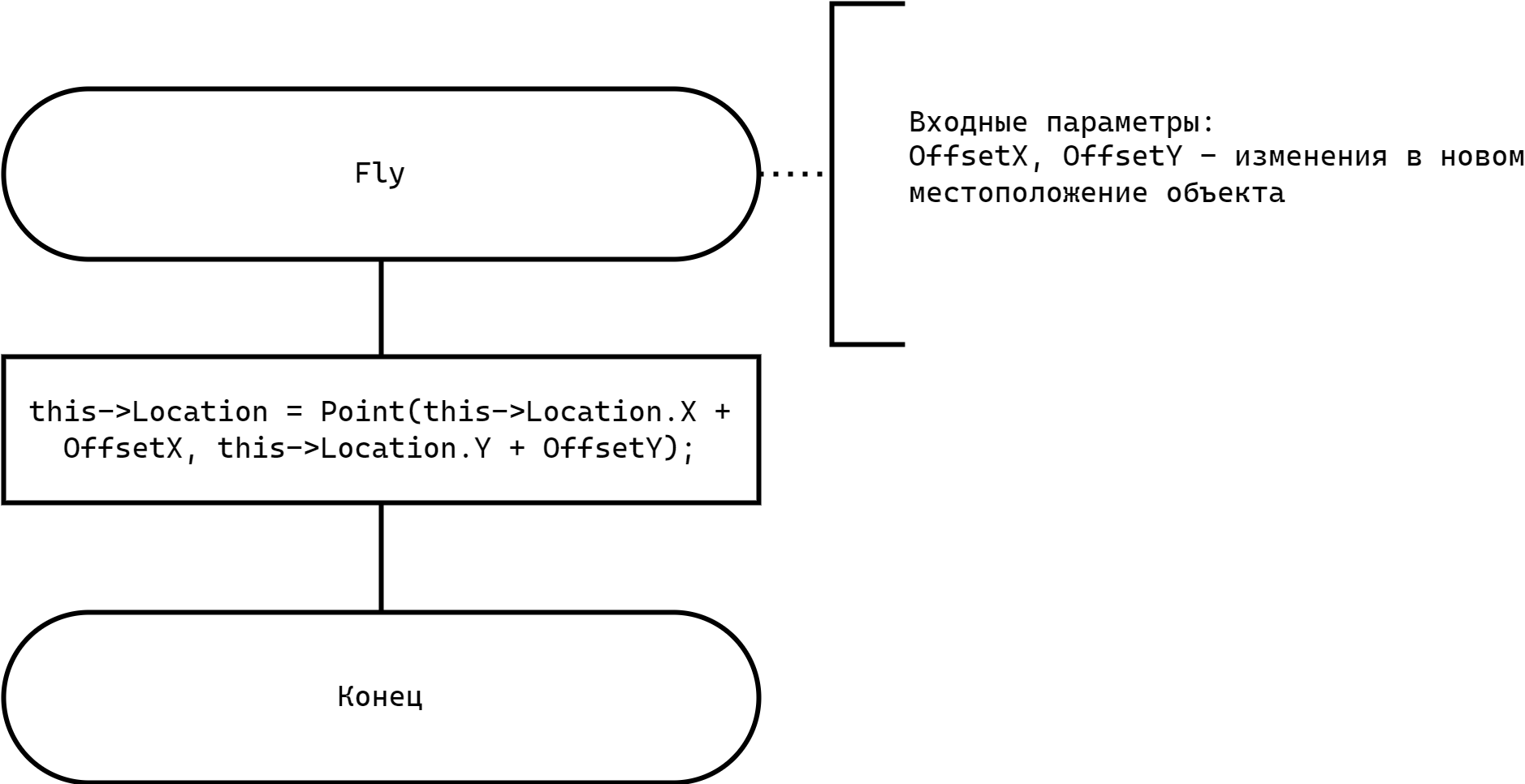


Рисунок 10 – Схема метода Fly

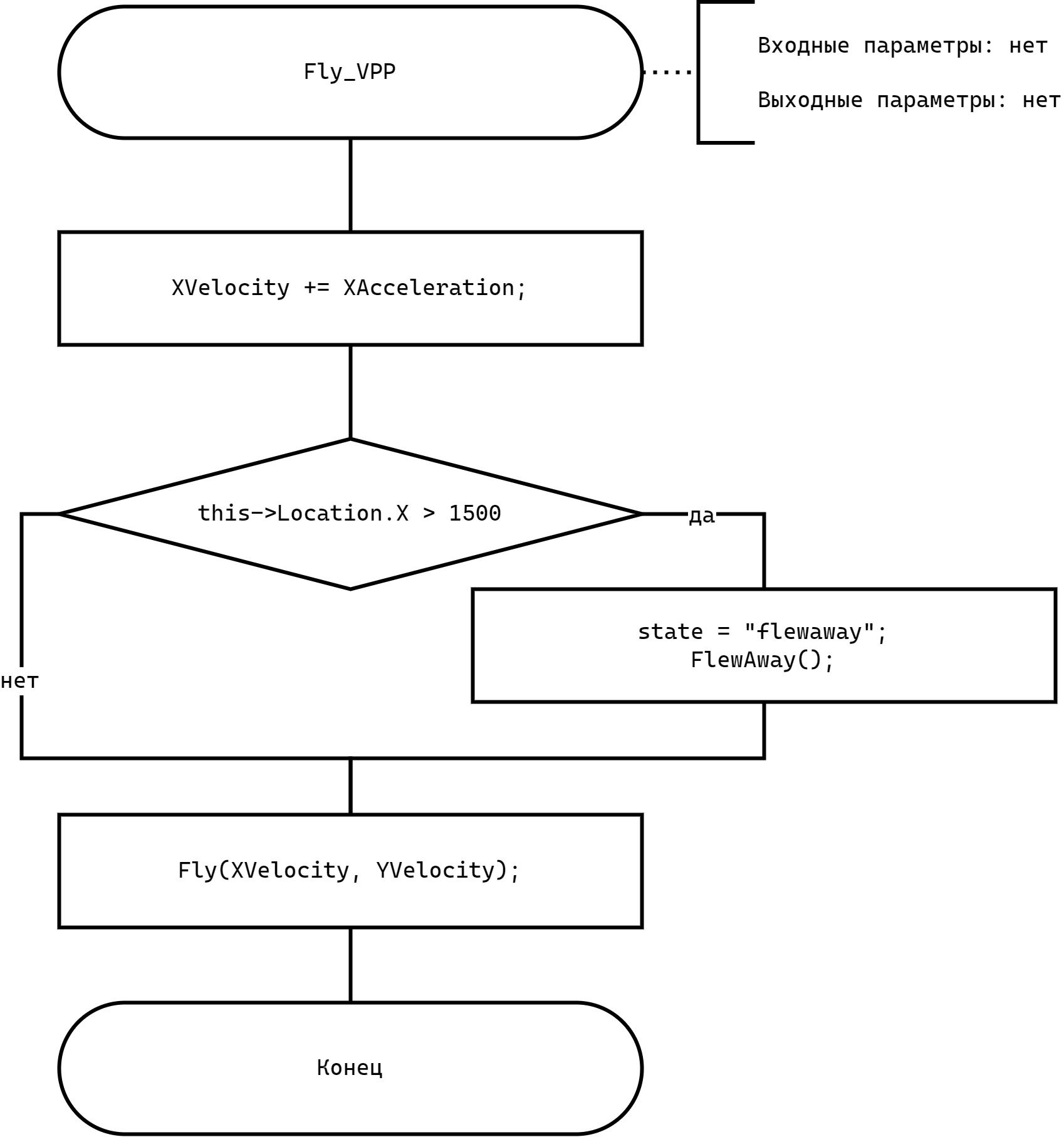


Рисунок 11 – Схема метода Fly\_VPP

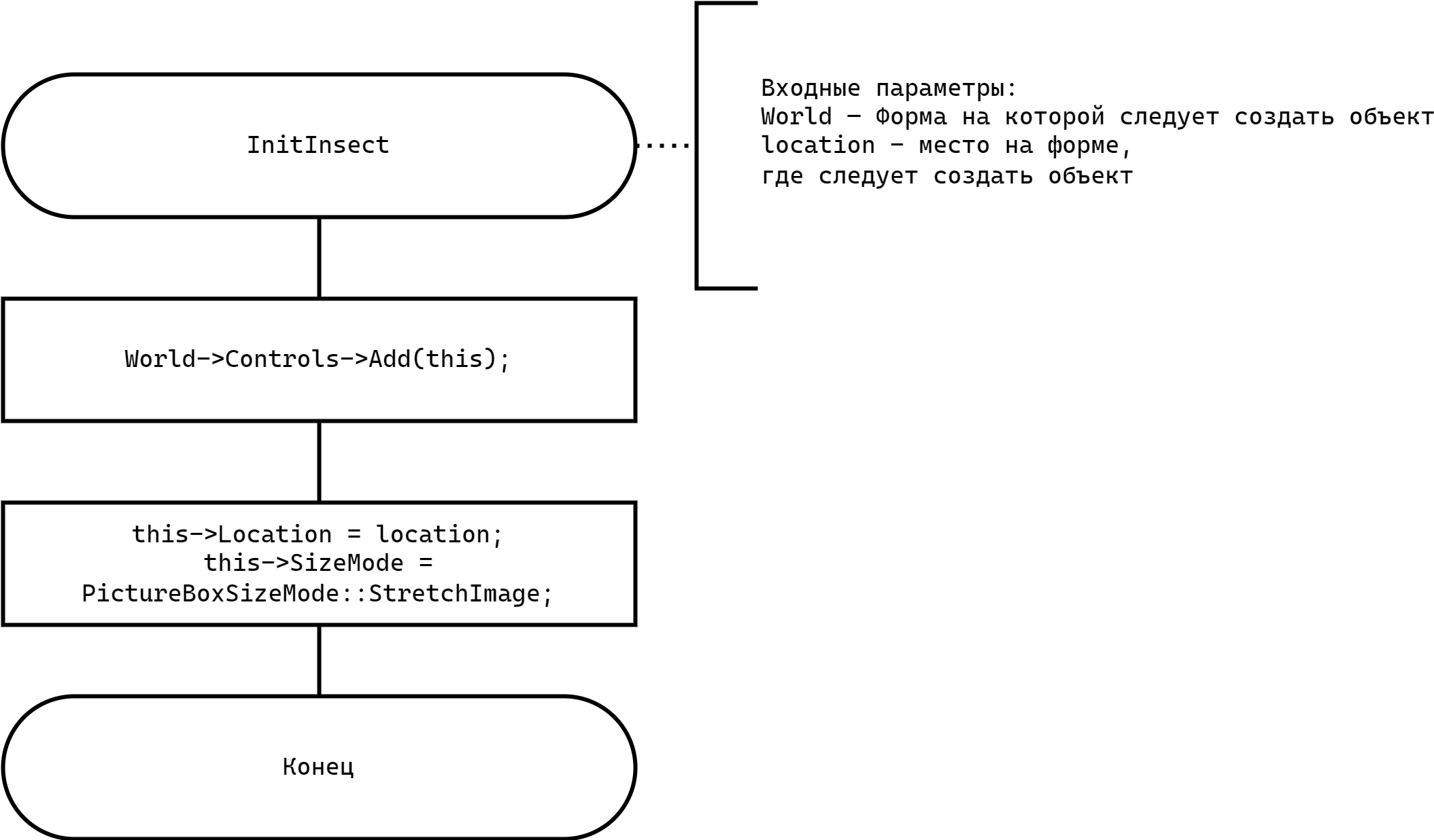


Рисунок 12 – Схема метода InitInsect

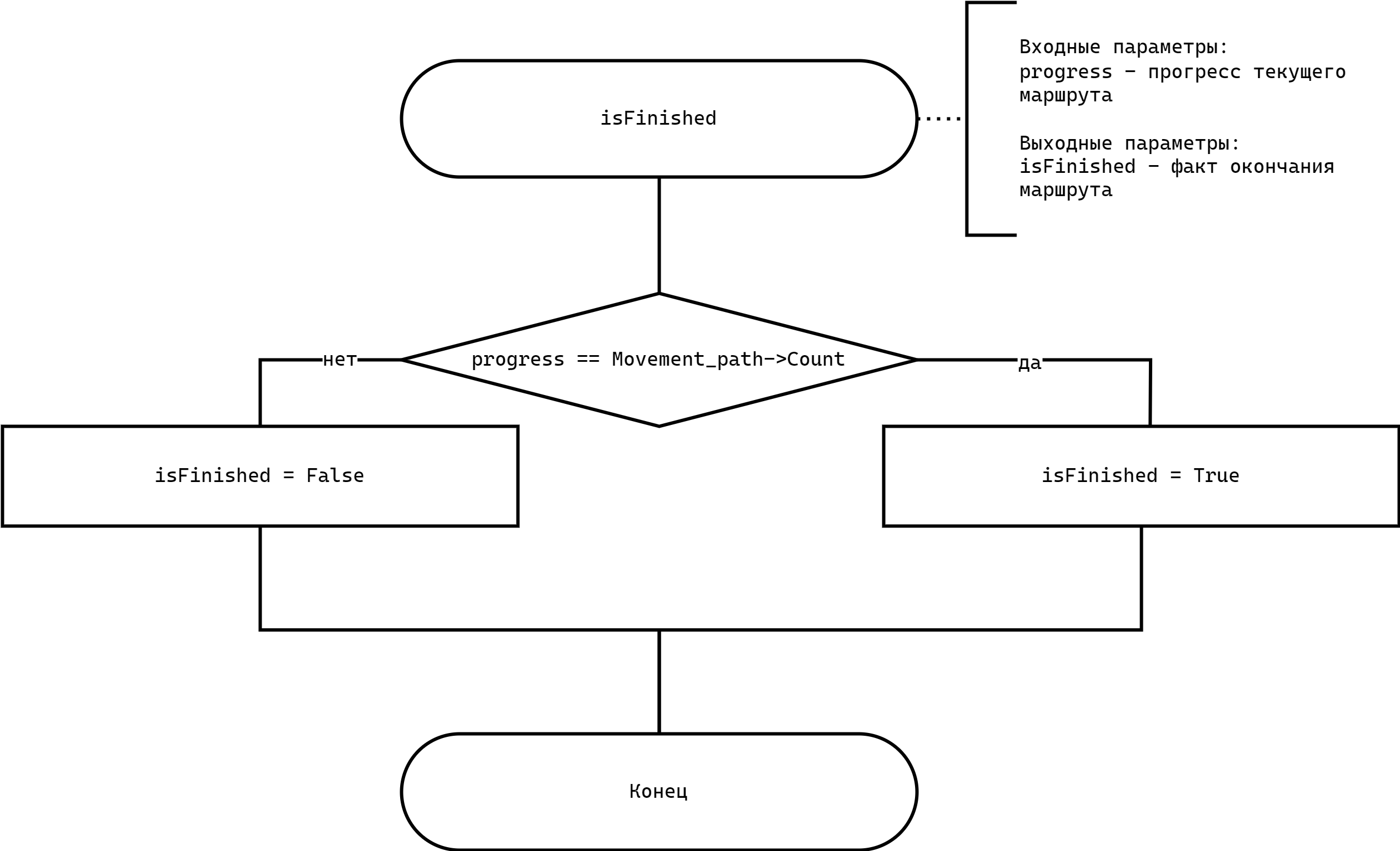


Рисунок 13 – Схема метода isFinished

**4 Текст программы**

Далее представлен программный код файла **AllClasses.h**

#pragma once

using namespace System;

using namespace System::Windows::Forms;

using namespace System::Collections::Generic;

using namespace System::Drawing;

namespace Creatures {

public ref class RotationalObject : PictureBox {

public: int size = 70;

protected: static String^ ImagePath = "test.bmp";

public: enum class Direction

{

None,

Up,

Right,

Down,

Left

};

property String^ PImage {

String^ get() {

return ImagePath;

}

void set(String^ \_ImagePath) {

ImagePath = \_ImagePath;

this->CurrentDirection = Direction::Up;

this->Size = System::Drawing::Size(size, size);

this->Image = Image->FromFile(ImagePath);

this->BackColor = Color::Transparent;

}

}

private: Direction CurrentDirection = Direction::Up;

//private: static String^ ImagePath = "\\..\\pictures\\Insect.png"; // still sucks

public: RotationalObject::RotationalObject() {

}

public: void ChangeImage(String^ ImagePath, int sizeX, int sizeY) {

this->CurrentDirection = Direction::Up;

this->Size = System::Drawing::Size(sizeX, sizeY);

this->Image = Image->FromFile(ImagePath);

this->BackColor = Color::Transparent;

}

public: void ChangeImage(String^ ImagePath, int size) {

ChangeImage(ImagePath, size, size);

}

public: RotationalObject::RotationalObject(Form^ World, Point location) {

this->Size = System::Drawing::Size(size, size);

this->Image = Image->FromFile(ImagePath);

this->BackColor = Color::Transparent;

InitInsect(World, location);

}

protected: void InitInsect(Form^ World, Point location) {

World->Controls->Add(this);

this->Location = location;

this->SizeMode = PictureBoxSizeMode::StretchImage;

}

public: void Fly(int OffsetX, int OffsetY) {

this->Location = Point(this->Location.X + OffsetX, this->Location.Y + OffsetY);

}

public : void Rotate(Direction direction) {

int WhereToRotate = Convert::ToInt32(direction) - Convert::ToInt32(CurrentDirection);

if (WhereToRotate < 0) {

WhereToRotate += 4;

}

auto img = this->Image;

img->RotateFlip(RotateFlipType(WhereToRotate));

this->Image = img;

this->CurrentDirection = direction;

}

Direction get\_direction() {

return CurrentDirection;

}

};

public ref class PlanedMovementObject : RotationalObject{

public:

float Speed = 5;

List<Point>^ Movement\_path;

int progress = 0;

int mid\_point = 3;

bool isFinished() {

return progress == Movement\_path->Count;

}

bool isMidPoint() {

return progress == mid\_point;

}

void move() {

Point Center = this->Location;

Center.Offset(Point(size / 2, size / 2));

if (isFinished()) return;

if (Center == Movement\_path[progress]) { progress++; return; }

Point delta = Point(0, 0);

delta.Offset(Center);

delta.X \*= -1;

delta.Y \*= -1;

delta.Offset(Movement\_path[progress]);

if (delta.X > Speed) { delta.X = Speed; Rotate(Direction::Right); }

if (delta.X < -Speed) { delta.X = -Speed; Rotate(Direction::Left); }

if (delta.Y > Speed) { delta.Y = Speed; Rotate(Direction::Down); }

if (delta.Y < -Speed) { delta.Y = -Speed; Rotate(Direction::Up); }

Fly(delta.X, delta.Y);

}

};

public delegate void FlewAwayHandler();

public delegate void LandedHandler();

public delegate void WaitingForCargoHandler();

public ref class Plane : PlanedMovementObject {

protected:

float \_XVelocity = 0;

float YVelocity = 0;

const float XAcceleration = 0.1;

const float YAcceleration = -0.1;

const float ReqXVelocity = 7;

const float TopXVelocity = 15;

const float TopYVelocity = 5;

property float XVelocity {

float get() {

return \_XVelocity;

}

void set(float velocity) {

\_XVelocity = velocity;

if (XVelocity > ReqXVelocity) {

YVelocity += YAcceleration;

}

if (XVelocity > -1 \* Speed && XVelocity < 0) {

XVelocity = -Speed;

}

}

}

protected:

static String^ ImagePath = "plane.png";

public:

List<Point>^ LandingMovement\_path;

event FlewAwayHandler^ FlewAway;

event LandedHandler^ Landed;

event WaitingForCargoHandler^ WaitingForCargo;

String^ state = "Idle3";

bool loaded = true;

bool airport\_is\_busy = true;

bool isCargo = false;

Plane::Plane() {

}

Plane::Plane(Form^ World, Point location, int size\_) {

state = "Idle3";

size = size\_;

ChangeImage(ImagePath, size);

location.Offset(Point(-size / 2, -size / 2));

InitInsect(World, location);

}

void prepare\_to\_fly() {

Speed = 5;

YVelocity = 0;

XVelocity = Speed;

state = "moving\_to\_VPP";

Movement\_path->Reverse();

progress = 0;

}

void prepare\_to\_land(List<Point>^ Movement\_path\_) {

Rotate(Direction::Left);

Speed = 5;

XVelocity = -2.5 \* Speed;

YVelocity = 3;

state = "landing";

Movement\_path\_->Reverse();

Movement\_path = Movement\_path\_;

progress = 0;

}

void prepare\_to\_wait(List<Point>^ Movement\_path\_) {

Speed = 17;

state = "wait";

Movement\_path = Movement\_path\_;

progress = 0;

}

public: enum class Callback

{

Default,

Bus

};

int idler = 0;

Callback tick() {

if (state == "flewaway") { Console::WriteLine("H1"); return Callback::Default; }

if (state == "Idle") { Console::WriteLine("Idle"); return Callback::Default; }

if (state == "wait" && isFinished()) { if (airport\_is\_busy) { progress = 0; } else { prepare\_to\_land(LandingMovement\_path); } Console::WriteLine("wait"); return Callback::Default; }

if (state == "wait") { move(); Console::WriteLine("wait"); return Callback::Default; }

if (state == "landing") { land(); Console::WriteLine("H2"); return Callback::Default; }

if (state == "moving\_to\_hung" && isFinished()) { state = "Idle"; Console::WriteLine("H3"); Landed(); return Callback::Default; }

if (state == "moving\_to\_hung" && (!isMidPoint() || !loaded)) { move(); Console::WriteLine("H4"); return Callback::Default; }

if (state == "moving\_to\_hung") { Console::WriteLine("cargo"); WaitingForCargo(); return Callback::Bus; }

if ((state == "moving\_to\_VPP" || state == "flying") && isFinished()) { state = "flying"; Fly\_VPP(); Rotate(Direction::Right); Console::WriteLine("H5"); return Callback::Default; };

if (state == "moving\_to\_VPP" && (!isMidPoint() || loaded)) { move(); Console::WriteLine("H6"); return Callback::Default; }

if (state == "moving\_to\_VPP") { Console::WriteLine("cargo"); WaitingForCargo(); return Callback::Bus; }

}

void Fly\_VPP() {

XVelocity += XAcceleration;

if (this->Location.X > 1500) {

state = "flewaway";

FlewAway();

}

Fly(XVelocity, YVelocity);

}

void land() {

int still\_accelerating = 0;

if (this->Location.Y >= 189 - size / 2) {

this->Location = Point(this->Location.X, 189 - size / 2);

YVelocity = 0;

++still\_accelerating;

XVelocity += XAcceleration;

}

if (XVelocity == -Speed) {

++still\_accelerating;

}

if (still\_accelerating == 2) {

state = "moving\_to\_hung";

}

Fly(XVelocity, YVelocity);

}

};

public ref class Bus : PlanedMovementObject {

static String^ ImagePath = "bus2.png";

public:

Bus::Bus() {}

Bus::Bus(Form^ World, Point location, int size\_) {

size = size\_;

ChangeImage(ImagePath, size, size \* 2);

location.Offset(Point(-size / 2, -size / 2));

InitInsect(World, location);

}

enum class Callback

{

Default,

FinishedLoading,

Delivered

};

int ticks\_waiting = 0;

Callback tick() {

move();

if (isFinished()) ticks\_waiting++;

if (ticks\_waiting == 100) {

progress = 0;

Movement\_path->Reverse();

ticks\_waiting++;

}

if (ticks\_waiting >= 100) {

if (isFinished()) { progress = 0; ticks\_waiting = 0; Movement\_path->Reverse(); Console::WriteLine("delivered"); return Callback::Delivered; }

return Callback::FinishedLoading;

}

return Callback::Default;

}

};

public ref class CargoCar : Bus {

static String^ ImagePath = "cargocar.png";

public:

CargoCar::CargoCar(Form^ World, Point location, int size\_) {

size = size\_;

ChangeImage(ImagePath, size, size \* 2);

location.Offset(Point(-size / 2, -size / 2));

InitInsect(World, location);

}

Bus::Callback tick() {

Bus::Callback ret = Bus::tick();

if (get\_direction() == Direction::Right || get\_direction() == Direction::Left) ChangeImage(ImagePath, size \* 2, size);

else ChangeImage(ImagePath, size, size \* 2);

return ret;

}

};

public ref class CargoPlane : Plane {

protected:

static String^ ImagePath = "cargoplane.png";

public:

CargoPlane::CargoPlane(Form^ World, Point location, int size\_) {

state = "Idle3";

size = size\_; isCargo = true;

ChangeImage(ImagePath, size);

location.Offset(Point(-size / 2, -size / 2));

InitInsect(World, location);

}

};

}

Далее представлен программный код файла **PlaneSimulator.h**

public ref class PlaneSimulator : public System::Windows::Forms::Form

{  
 int\* clock\_intervals = new int[6];

int clock\_carr = 3;

Plane^ p;

List<Plane^> planes;

int current\_plane = -1;

int waiting\_plane = -1;

List<Point>^ Movement\_path0 = gcnew List<Point>;

List<Point>^ Movement\_path1 = gcnew List<Point>;

List<Point>^ Movement\_path2 = gcnew List<Point>;

List<Point>^ Movement\_path3 = gcnew List<Point>;

List<Point>^ Movement\_path\_bus = gcnew List<Point>;

List<Point>^ Movement\_path\_cargo\_car = gcnew List<Point>;

List<Point>^ Elipse = gcnew List<Point>;

Random^ rnd = gcnew Random();

Bus^ Buss;

CargoCar^ CargoCarr;

Point in\_sky;

private: System::Void PlaneSimulator\_Load(System::Object^ sender, System::EventArgs^ e) {

clock\_intervals[0] = 5;

clock\_intervals[1] = 10;

clock\_intervals[2] = 20;

clock\_intervals[3] = 50;

clock\_intervals[4] = 100;

clock\_intervals[5] = 200;

in\_sky = left\_VPP->Location;

in\_sky.X += 1300;

in\_sky.Y -= 150;

const int points\_discretion = 60;

const int radius = 500;

for (int i = 0; i <= points\_discretion; ++i) {

Elipse->Add(Point(in\_sky.X - radius + radius \* Math::Cos(i \* 2 \* Math::PI / points\_discretion), in\_sky.Y + radius / 8 \* Math::Sin(i \* 2 \* Math::PI / points\_discretion)));

}

Buss = gcnew Bus(this, bus\_stop->Location, 30);

Buss->BringToFront();

CargoCarr = gcnew CargoCar(this, cargo\_stop->Location, 35);

CargoCarr->BringToFront();

Movement\_path\_bus->Add(bus\_stop->Location);

Movement\_path\_bus->Add(bus\_plane\_stop->Location);

Buss->Movement\_path = Movement\_path\_bus;

Movement\_path\_cargo\_car->Add(cargo\_stop->Location);

Movement\_path\_cargo\_car->Add(cargo\_cross->Location);

Movement\_path\_cargo\_car->Add(bus\_plane\_stop->Location);

CargoCarr->Movement\_path = Movement\_path\_cargo\_car;

Movement\_path0->Add(hung\_0->Location);

Movement\_path0->Add(hung\_cross1->Location);

Movement\_path1->Add(hung\_1->Location);

Movement\_path1->Add(hung\_cross1->Location);

Movement\_path2->Add(hung\_2->Location);

Movement\_path2->Add(hung\_cross2->Location);

Movement\_path3->Add(hung\_3->Location);

Movement\_path3->Add(hung\_cross2->Location);

Movement\_path0->Add(human\_cross->Location);

Movement\_path0->Add(left\_down->Location);

Movement\_path0->Add(left\_VPP->Location);

Movement\_path1->Add(human\_cross->Location);

Movement\_path1->Add(left\_down->Location);

Movement\_path1->Add(left\_VPP->Location);

Movement\_path2->Add(human\_cross->Location);

Movement\_path2->Add(left\_down->Location);

Movement\_path2->Add(left\_VPP->Location);

Movement\_path3->Add(human\_cross->Location);

Movement\_path3->Add(left\_down->Location);

Movement\_path3->Add(left\_VPP->Location);

planes.Add(nullptr);

planes.Add(nullptr);

planes.Add(nullptr);

planes.Add(nullptr);

NextPlane();

}

List<Point>^ Movement\_path;

List<Point>^ generate\_path(int index) {

if (index == 0) return Movement\_path = Movement\_path0;

if (index == 1) return Movement\_path = Movement\_path1;

if (index == 2) return Movement\_path = Movement\_path2;

if (index == 3) return Movement\_path = Movement\_path3;

}

void LandPlane() {

Console::WriteLine("LandPlane");

current\_plane = -1;

for (int i = 0; i < 4; ++i) {

if (planes[i] == nullptr) {

current\_plane = i;

planes[i] = get\_plane();

break;

}

}

planes[current\_plane]->Rotate(RotationalObject::Direction::Left);

planes[current\_plane]->BringToFront();

planes[current\_plane]->prepare\_to\_land(generate\_path(current\_plane));

//planes[current\_plane]->Landed += gcnew LandedHandler(this, &PlaneSimulator::NextPlane);

}

void FlyPlane() {

Console::WriteLine("FlyPlane");

current\_plane = -1;

int there\_is\_plane = -1;

for (int i = 0; i < 4; ++i) {

if (planes[i] == nullptr) continue;

there\_is\_plane = i;

if (rnd->Next() < 0.3) current\_plane = i;

}

if (current\_plane == -1) current\_plane = there\_is\_plane;

Console::WriteLine(current\_plane);

planes[current\_plane]->prepare\_to\_fly();

Console::WriteLine(planes[current\_plane]->state);

//planes[current\_plane]->FlewAway += gcnew FlewAwayHandler(this, &PlaneSimulator::NextPlane);

}

void NextPlane() {

Thread::Sleep(100);

Console::WriteLine("NextPlane");

clear\_missing\_planes();

int real\_count = 0;

for (int i = 0; i < 4; ++i) {

if (planes[i] != nullptr) ++real\_count;

}

if (waiting\_plane != -1) {

current\_plane = waiting\_plane;

waiting\_plane = -1;

planes[current\_plane]->airport\_is\_busy = false;

return;

}

if (real\_count == 0) { LandPlane(); return; }

if (real\_count == 4) { FlyPlane(); return; }

if (rnd->NextDouble() < 0.5) {

LandPlane();

}

else {

FlyPlane();

}

}

void bus\_tick() {

Console::WriteLine("123");

if (planes[current\_plane]->isCargo) {

Bus::Callback clbb = CargoCarr->tick();

if (clbb == Bus::Callback::Delivered) planes[current\_plane]->loaded ^= 1;

}

else {

Bus::Callback clbb = Buss->tick();

if (clbb == Bus::Callback::Delivered) planes[current\_plane]->loaded ^= 1;

}

}

Plane^ get\_plane() {

Plane^ ret;

if (rnd->NextDouble() < 0.5) ret = gcnew Plane(this, in\_sky, 85);

else ret = gcnew CargoPlane(this, in\_sky, 100);

ret->Landed += gcnew LandedHandler(this, &PlaneSimulator::NextPlane);

ret->FlewAway += gcnew FlewAwayHandler(this, &PlaneSimulator::NextPlane);

ret->WaitingForCargo += gcnew WaitingForCargoHandler(this, &PlaneSimulator::bus\_tick);

return ret;

}

void add\_to\_wait() {

if (waiting\_plane != -1) return;

int there\_is\_plane = -1;

for (int i = 0; i < 4; ++i) {

if (planes[i] != nullptr) continue;

there\_is\_plane = i;

if (rnd->Next() < 0.3) waiting\_plane = i;

}

if (waiting\_plane == -1) waiting\_plane = there\_is\_plane;

if (waiting\_plane == -1) return;

planes[waiting\_plane] = get\_plane();

planes[waiting\_plane]->Rotate(RotationalObject::Direction::Left);

planes[waiting\_plane]->BringToFront();

planes[waiting\_plane]->prepare\_to\_wait(Elipse);

planes[waiting\_plane]->LandingMovement\_path = generate\_path(waiting\_plane);

//planes[waiting\_plane]->Landed += gcnew LandedHandler(this, &PlaneSimulator::NextPlane);

}

void clear\_missing\_planes() {

for (int i = 0; i < 4; ++i) {

if (planes[i] == nullptr) continue;

if (planes[i]->state == "flewaway") {

planes[i] = nullptr;

}

}

}

void main\_tick() {

Plane::Callback clb = planes[current\_plane]->tick();

if (clb == Plane::Callback::Default) return;

/\*if (clb == Plane::Callback::Bus) {

if (planes[current\_plane]->isCargo) {

Bus::Callback clbb = CargoCarr->tick();

if (clbb == Bus::Callback::Delivered) planes[current\_plane]->loaded ^= 1;

}

else {

Bus::Callback clbb = Buss->tick();

if (clbb == Bus::Callback::Delivered) planes[current\_plane]->loaded ^= 1;

}

}

\*/

}

private: System::Void timer1\_Tick(System::Object^ sender, System::EventArgs^ e) {

main\_tick();

if (rnd->NextDouble() < 1.0 / 1000.0) add\_to\_wait();

if (waiting\_plane != -1) planes[waiting\_plane]->tick();

}

private: System::Void button1\_Click(System::Object^ sender, System::EventArgs^ e) {

timer1->Enabled = !timer1->Enabled;

if (!timer1->Enabled) button1->Text = "Запустить симуляцию";

else button1->Text = "Остановить симуляцию";

}

private: System::Void PlaneSimulator\_MouseClick(System::Object^ sender, System::Windows::Forms::MouseEventArgs^ e) {

add\_to\_wait();

}

private: System::Void button8\_Click(System::Object^ sender, System::EventArgs^ e) {

if (++clock\_carr > 5) clock\_carr = 5;

timer1->Interval = clock\_intervals[clock\_carr];

label1->Text = ( 6 - clock\_carr).ToString();

}

private: System::Void button9\_Click(System::Object^ sender, System::EventArgs^ e) {

if (--clock\_carr < 0) clock\_carr = 0;

timer1->Interval = clock\_intervals[clock\_carr];

label1->Text = ( 6 - clock\_carr).ToString();

}

**5 Результаты тестирования программы**

При запуске программы мы попадаем на титульную форму, где указано авторство работы, название работы, имя учебного заведения и прочие данные. На титульной форме есть кнопка выхода и кнопка для перехода к симуляции. Перейдём к симуляции.

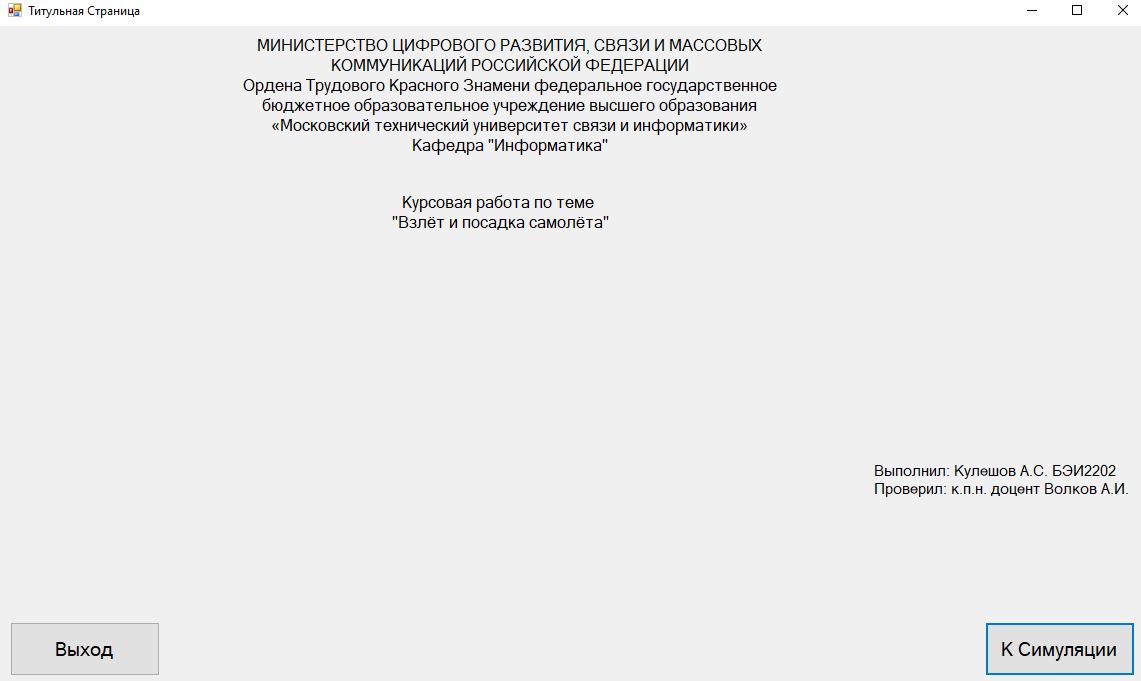


Рисунок 14 – Титульная форма

На форме с симуляцией пользователь может заметить разметку взлётных полос аэропорта, ангар, а также полосы для транспорта. Симуляция происходит автономна, без участия пользователя. Однако, он может ускорить, замедлить, а также приостановить симуляцию, нажимая соответствующие кнопки

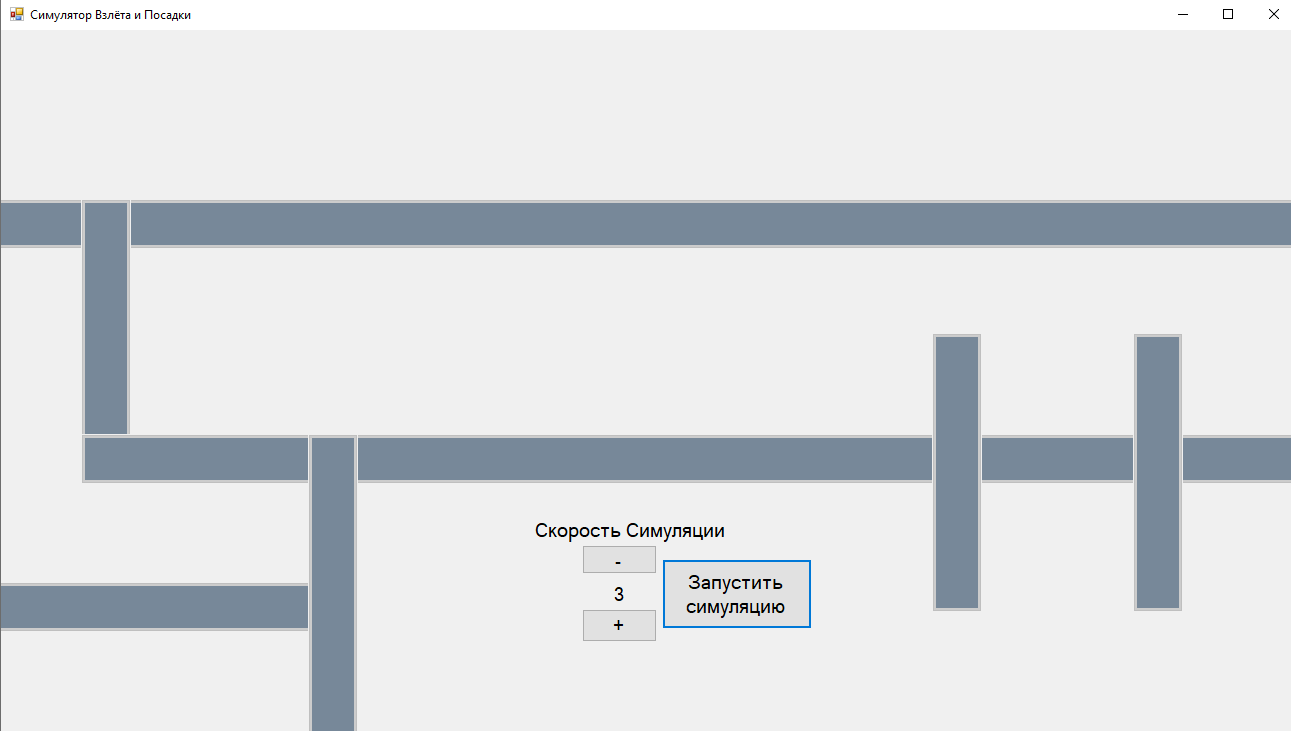
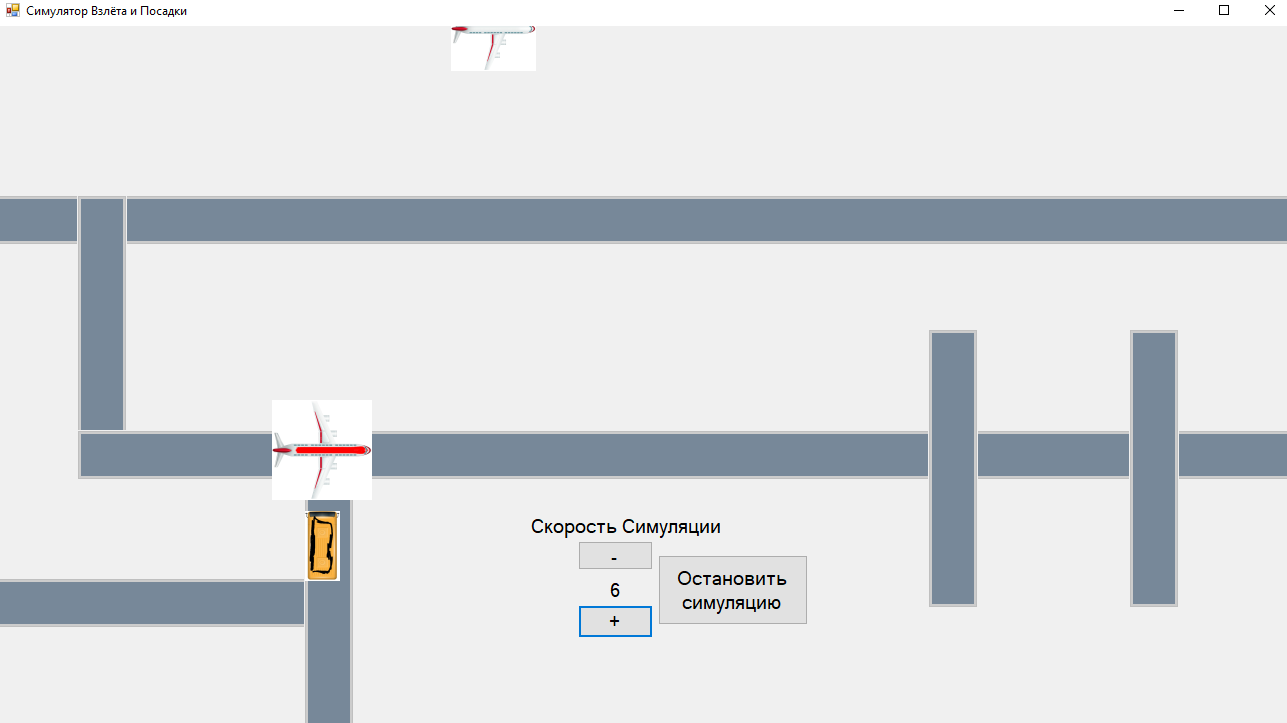


Рисунок 15 – Форма симуляции

После нажатия на кнопку симуляции на кнопку «Запустить симуляцию» аэропорт начнёт свою работу: самолёты будут производить посадку и вылет, такси будут разгружать пассажирские самолёты, а грузовые машины – грузовые самолёты.

 Рисунок 16 – Симуляция активна

Появление самолётов происходит произвольно, поэтому в случае если аэропорт уже обрабатывает какой-либо самолёт, то если некому другому придётся совершить посадку, то ему придётся ждать своей очереди на посадку.

Даже если уже какому-либо самолёту необходимо взлететь, то самолёту, ожидающему в воздухе отдаётся приоритет, поэтому ожидающий самолёт произведёт посадку, а только потом первый взлетит

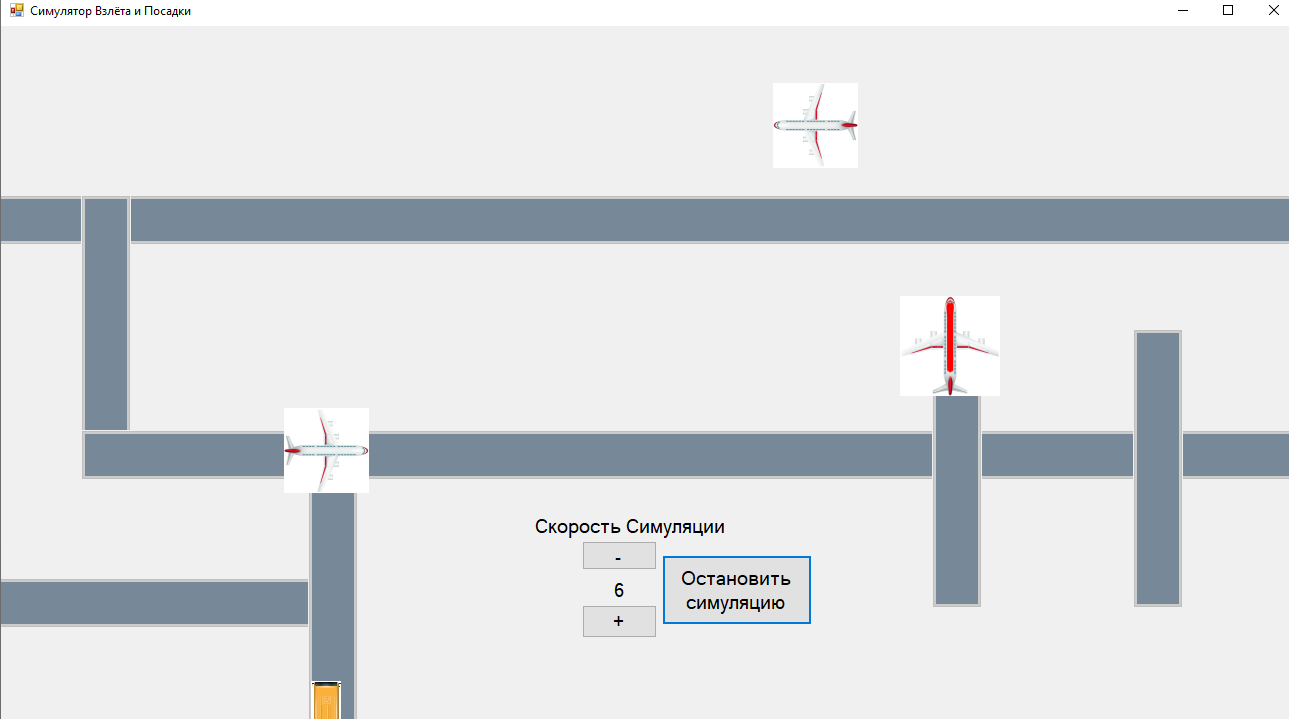


Рисунок 17 – Ожидающий самолёт в воздухе

Аэропорт обслуживает два вида самолёта: грузовой и пассажирский. Грузовой немного больше пассажирского, а также окрашен в красный, в отличии от пассажирского, белого и меньшего по размерам. Пассажирские самолёты загружаются и разгружаются автобусами, которые подъезжают по вертикальной дороге, в то время как грузовые – грузовыми машинами, которые используют горизонтальную.

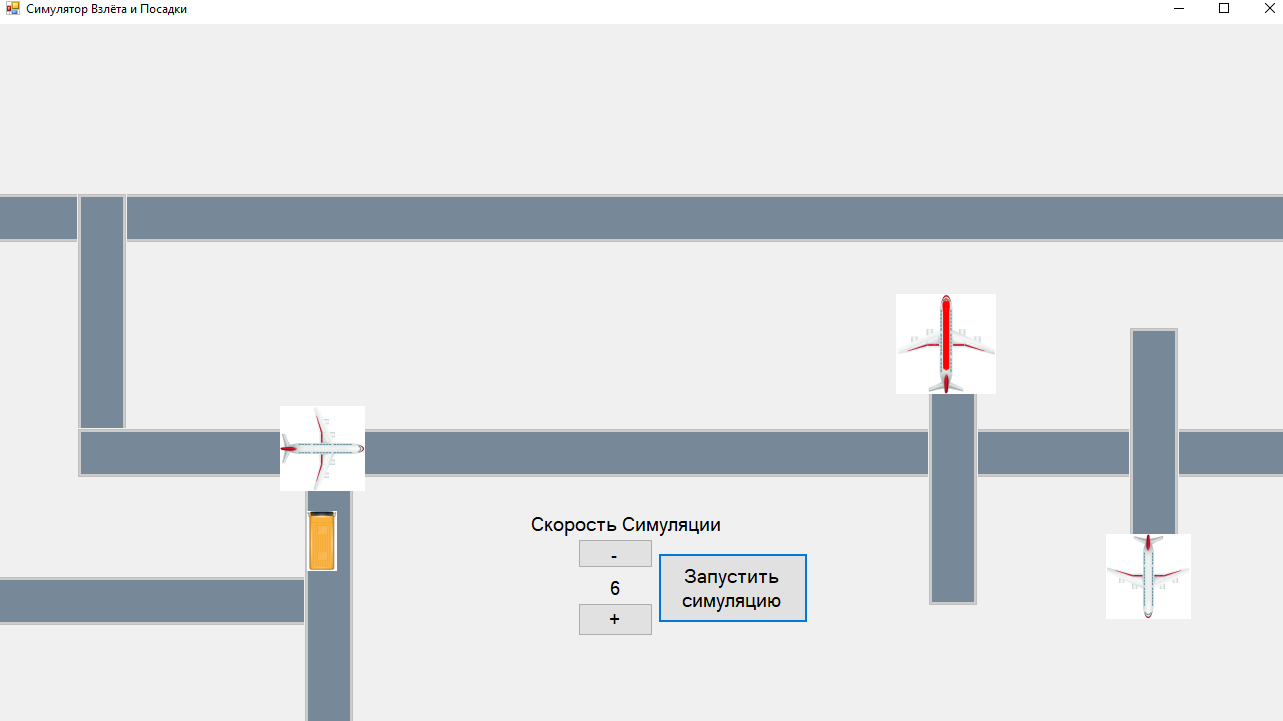


Рисунок 18 – Разгрузка пассажирского самолёта автобусом

Таким образом состояния каждого самолёта в симуляции можно выделить в короткий список. Состояния, в котором постепенно переходят от первого к последнему:

1. Ожидание в воздухе (опционально)
2. Посадка
3. Разгрузка
4. Ожидание в ангаре до следующего вылета
5. Выезд из ангара
6. Погрузка
7. Вылет
8. Конец. Самолёт покинул аэропорт