Inlämningsuppgift
Artificiell intelligens i dataspel
2014

Sebastian Zander
a12sebza@mail.his.se
870318

Institutionen för kommunikation och information
Högskolan i Skövde

1 Introduktion

Uppgiften för kursen *artificiell intelligens i dataspel* går ut på att först lösa fem valfria problem med fem olika tekniker:

1. Söktekniker
2. Evolutionära tekniker
3. Konnektivistiska tekniker
4. Skriptekniker
5. Tillståndstekniker

Sedan ska en större applikation (spel eller simulering) utvecklas med en större fördjupning inom en eller flera av teknikerna.

Kursen behandlar AI i dataspel. AI i dataspel utvecklas framförallt för att återspegla männskligt beteende, och många metoder (från andra dicipliner inom datorvetenskap än AI) används. Målet med AI i spel är att ge sken om intelligens och i många fall används speciella metoder för att uppnå det målet som ej är kopplat till traditionell AI. Exempelvis måste spel-AI ibland fuska, exempelvis genom att få mer resurser än en männsklig spelare, och ibland måste AI:n handikappas, genom att exempelvis tvinga AI:n missa skott i FPS-spel för att maskera det faktum att datorn annars skulle ha perfekt sikte. Datorns nackdelar jämfört med människor, avsaknad av kreativitet, samt dess fördelar, snabba uträkningar, måste alltså maskeras för spelaren för att ge sken av en att spelaren möter eller samarbetar med så männskliga agenter som möjligt.

AI i andra tillämpningar används istället för att praktiskt lösa problem som traditionellt en människa annars hade varit tvungen att lösa. Det finns ofta inget skäl att låtsas att AI:n faktiskt är en människa. Eftersom att AI i spel behöver agera på ett männskligt sätt döms ofta AI:n i spel utifrån det.

Exempel på dålig AI i spel är framförallt spel som bygger på att spelaren ska smyga eller diskret ta sig fram till ett mål (exempelvis Assasin's Creed). Det blir ganska tydligt att de andra karaktärerna i spelet inte är människor när de sekunden efter de upptäckt att deras kumpaner har dött av en knivhuggning fortsätter patrullera som om ingenting har hänt. Detta riskerar att göra spelaren uppmärksam på att den spelaren ett spel och på så sätt kan det sänka immersionen.

Exempel på bra AI i dataspel är AI:n i Dwarf Fortress som också är väldigt komplicerad. Världen i Dwarf Fortress är procedurellt genererad med hjälp av en mängd olika metoder i samverkan. När världen har genererats fram genereras även en historia fram, där karaktärer i historien även kan påverka landskapet eller andra karaktärer (exempelvis kan spelaren, i Adventure Mode, stöta på en karaktär som har specifika åsikter om en historisk karaktär). Väl i spelet reagerar karaktärerna i spelet väl och övertygande på händelser som beror på andra karaktärer eller spelaren. En fördel Dwarf Fortress har som andra rollspell inte har, exempelvis Skyrim, är en simplistisk grafik så utvecklaren behöver inte skapa animationer eller ljudspår för varje reaktion karaktärerna i spelet har.

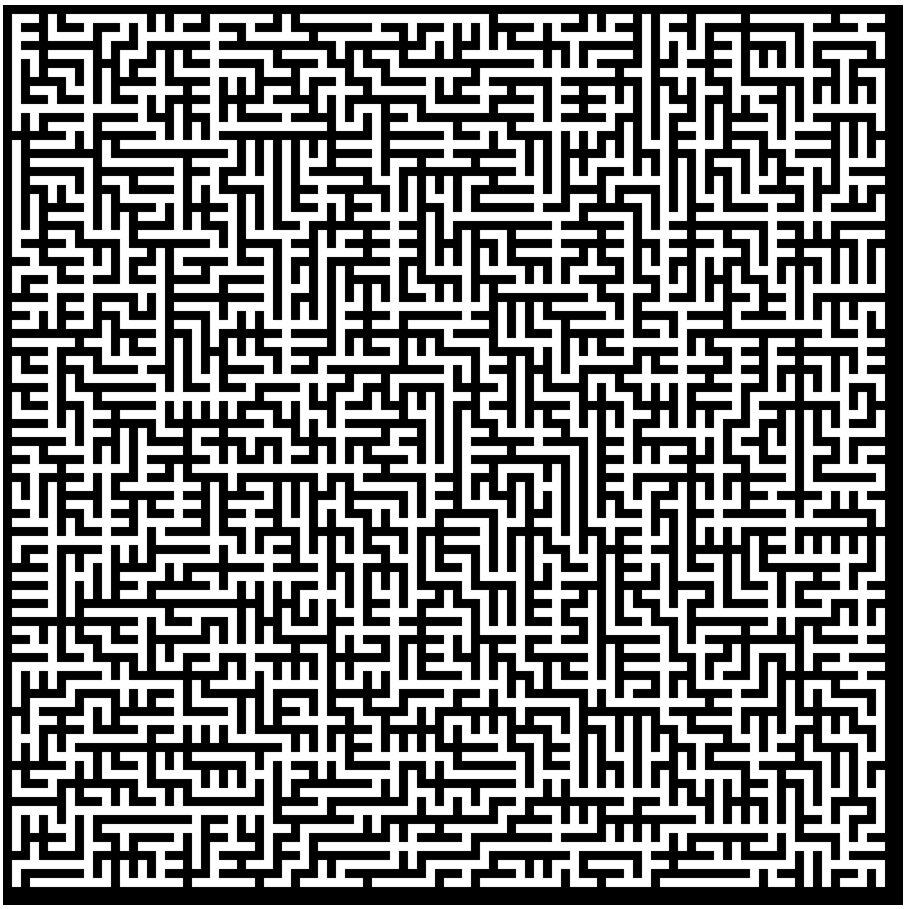
Rapporten kommer inledas med teknikerna, där varje teknik får ett eget delkapitel. Sedan kommer applikationen beskrivas. Därefter kommer en slutsats.

2 Tekniker

2.1 Söktekniker

2.1.1 Problem

Problemet är att givet en godtycklig, men lösbar, labyrint så ska en agent hitta en giltig väg mellan en startpunkt och en slutpunkt. Agenten får inte röra sig genom väggar i labyrinten. Som del av problemet ingår också att skapa labyrinten.



Figur 1. Ett exempel på en labyrint som skall lösas, där svart färg representerar väggar.

2.1.2 Design

En sökteknik används först för att skapa labyrinten (en slumpbaserad variant av Prims algoritm används), därefter används en annan sökteknik (A*) för att lösa labyrinten från en slumpad startpunkt till en slumpad slutpunkt. Med dessa algoritmer hittas garanterat en lösning sålänge som startpunkten och slutpunkten befinner sig inom labyrinten

Varianten av prims algoritm börjar med en grid av bara väggar. En cell väljs ut och läggs i en lista av celler. Från den listan väljs en slumpmässig cell ut: om cellen på motsatt sida är en vägg skapas en passage mellan dessa två celler och grannväggarna till dessa celler läggs in i listan av celler. Detta skapar ett spännande träd, vilket gör att alla celler går att nå från alla andra celler.

A* (a stjärna) är en utökning av Dijkstras Algoritm som får bättre tidsprestanda genom att använda sig av heuristik. Algoritmen letar efter den kortaste vägen mellan två noder genom bäst-först sökning. Heuristiken som används i den här lösningen är manhattan-avståndet mellan nuvarande nod och slutnoden. Algoritmen

är generell för vilken graf som helst. I den här lösningen består noderna av cellerna i labyrinten som inte är väggar och alla noder är kopplade med sina direkta grannar (ej diagonalt).

I algoritmen har varje nod (celler i labyrinten i den här lösningen) ett H-värde (heuristik-värdet), ett G-värde (konstnaden att ta sig till noden, initierat som oändligt) och ett F-värde (G + H). För att kunna ta utvägen behöver noder också peka på en föräldranod, den noden som är nåbar från noden. Det finns två listor av noder: den öppna listan och den stängda listan. Den öppna listan innehåller alla noder som ska beräknas och den stängda listan innehåller alla noder som har beräknats. Algoritmen startar med startnoden i den öppna listan (G-värdet för denna noden är 0, så F-värdet är detsamma som H-värdet).

I varje iteration av algoritmen tas den nod med lägst F-värde ut ur den öppna listan och behandlas. Om denna nod är slutnoden har en väg hittats, och vägen kan rekonstrueras genom att följa nodens förälder och förälderns förälder etc tills startnoden hittas. I annat fall lägst noden till den stängda listan och alla dess grannar (noderna direkt nåbara från den noden som behandlas) som inte redan finns i den stängda eller öppna listan behandlas.

Ett tillfälligt G-värde räknas ut för varje granne, som är G-värdet för den nuvarande noden plus avstånden mellan den noden och grannen. I den här lösningen är alla avstånd mellan grannar 1. Om detta värde är mindre än grannens G-värde (som alltså är oändligt från början) uppdateras den nodens G-värde (sätts till det tillfälliga G-värde) och F-värde (G + H) och dess föräldernod sätts till den nod som behandlades från början.

Iterationen börjar om så länge det finns noder i den öppna listan.

Till slut så kommer algoritmen alltså komma till slutnoden. Den nodens G-värde är antalet steg det tar att gå till den noden från startnoden. Vägen mellan startnoden och slutnoden går alltså också att tas ut med den här metoden genom att rekursivt följa noders föräldranod tillbaka startnoden.

2.1.3 Kod

2.1.3.1 Astar.h

```
#pragma once

#include "../Maze.h"
#include "../Coord.h"
#include "../Solver.h"

#include <queue>
#include <unordered_set>
#include <map>

class Astar: public Solver
{
public:
        Astar(Coord start, Coord end);

        /* returns true if there are more steps */
        bool Step(Maze& maze) override;
        void Reset() override;
        void Reset(Coord start, Coord end) override;

private:
        int Heuristic(Coord a, Coord b);

        int steps_;
        Coord start_, end_;
        Coord current_;
        Coord walker_;
```

```
        class PriorityQueueCompare
        {
        public:
                bool operator()(std::pair<int, Coord>& t1, std::pair<int, Coord>& t2)
                {
                        return t1.first > t2.first;
                }
        };

        std::unordered_set<Coord> closed_set_;
        std::unordered_set<Coord> open_set_;
        std::priority_queue<std::pair<int, Coord>, std::vector<std::pair<int, Coord>>, Priorit
yQueueCompare> open_queue_;
        std::unordered_map<Coord, int> g_;
        std::unordered_map<Coord, int> f_;
        std::unordered_map<Coord, Coord> parent_of_;
        std::vector<Coord> path_;

        Maze came_from_;
};
```

2.1.3.2 Maze.h

```
#pragma once

#include "Window.h"

class Maze
{
public:
        struct Tile
        {
                Tile()
                         :cost(-1)
                        , color(Color::black())
                {
                }

                Tile(int cost_, Color color_)
                        :cost(cost_)
                        ,color(color_)
                {
                }
                //Any negative cost means the tile is solid/closed/occupied etc..
                int cost;
                Color color;
        };
        Maze(int w, int h);
        Maze();
        Maze(const Maze& other);
        Maze& operator=(const Maze& other);

        void Render(Window& window, int x, int y, int w, int h);
        ~Maze();

        Tile& operator()(int x, int y);
        const Tile& operator()(int x, int y) const;

        const int Width() const { return w_; }
        const int Height() const { return h_; }

        bool IsBorderCoord(int x, int y) const;
```

```
private:
        Tile& Get(int x, int y);
        const Tile& Get(int x, int y) const;

        inline bool EnsureRange(int x, int y) const { return x >= 0 && x < w_ && y >= 0 && y <
 h_; };

        void InnerCopy(const Maze& other);

        std::vector<Tile> maze_;
        int w_, h_;
        SDL_Surface* surface_;
};
```

2.1.3.3 PrimGenerator.h

```
#pragma once

#include "Random.h"
#include "Maze.h"
#include <vector>
#include "Coord.h"

class PrimGenerator
{
public:
        PrimGenerator(const Coord& start);

        /* returns true if there are more steps */
        bool Step(Maze& maze);
        void Reset(Maze& maze, const Coord& start);

private:

        void TryAddWalls(const Coord& coord, const Maze& maze);
        void Clear(Maze& maze);

        std::vector<Coord> walls_;

        int steps_;
        Randomizer random_;

        Coord start_;
};
```

2.1.3.4 Solver.h

```
#pragma once

#include "Coord.h"
class Maze;


class Solver
{
public:
        virtual ~Solver() {}
        virtual bool Step(Maze& maze) = 0;
        virtual void Reset() = 0;
        virtual void Reset(Coord start, Coord end) = 0;
};
```

2.1.3.5 Astar.cpp

```
#include "Astar.h"

Astar::Astar(Coord start, Coord end)
        :start_(start)
        ,end_(end)
        ,current_(start)
        ,steps_(0)
{

}


bool Astar::Step(Maze& maze)
{
        if (steps_ == 0)
        {
                came_from_ = Maze(maze.Width(), maze.Height());
                maze(start_.x, start_.y).color = Color::green();
                maze(end_.x, end_.y).color = Color::red();

                Reset();
        }

        ++steps_;

        bool done = false;

        if (open_queue_.size() != 0)
        {
                current_ = open_queue_.top().second;

                if (current_ != end_)
                {
                        open_queue_.pop();
                        open_set_.erase(current_);
                        closed_set_.insert(current_);
                        Coord neighbours[] =
                        {
                                { current_.x -
 1, current_.y }, { current_.x + 1, current_.y },
                                { current_.x, current_.y -
 1 }, { current_.x, current_.y + 1 }
                        };

                        for (auto& n : neighbours)
                        {

        if (closed_set_.count(n) == 0 && !maze.IsBorderCoord(n.x, n.y) && maze(n.x, n.y).cost
>= 0)
                                {

        int tentative_g = g_[current_] + maze(current_.x, current_.y).cost;
                                        if (open_set_.count(n) == 0 || tentative_g < g_[n])
                                        {
                                                came_from_(n.x, n.y).cost = 1;
                                                if (n != end_)
                                                {

        maze(n.x, n.y).color = Color::blue();
                                                }
```

```
                                                if (open_set_.count(n) == 0)
                                                {

        parent_of_.insert(std::make_pair(n, current_));

        g_.insert(std::make_pair(n, tentative_g));

        f_.insert(std::make_pair(n, Heuristic(n, end_)));
                                                        open_set_.insert(n);

        open_queue_.push(std::make_pair(Heuristic(n, end_), n));
                                                }
                                                else
                                                {
                                                        parent_of_[n] = current_;
                                                        g_[n] = tentative_g;
                                                        f_[n] = Heuristic(n, end_);
                                                }
                                        }
                                }
                        }
                }
        }

        if (current_ == end_)
        {
                if (walker_ != start_)
                {
                        walker_ = parent_of_[walker_];
                        if (walker_ != end_ && walker_ != start_)
                        {
                                path_.push_back(walker_);

        maze(walker_.x, walker_.y).color = Color::make_from_bytes(127, 127, 255);
                        }
                }
                else if (!path_.empty())
                {
                        if (steps_ % 5 != 0)
                        {
                                return true;
                        }
                        Coord p = path_.back();
                        maze(p.x, p.y).color = Color::make_from_bytes(127, 127, 255);
                        path_.pop_back();
                        if (!path_.empty())
                        {
                                Coord p = path_.back();
                                maze(p.x, p.y).color = Color::make_from_bytes(0, 255, 255);
                        }
                }
                else
                {
                        return false;
                }
        }

        return !done;
}

void Astar::Reset(Coord start, Coord end)
```

```
{
        start_ = start;
        end_ = end;
        Reset();
}

void Astar::Reset()
{
        for (int i = 0; i < came_from_.Height(); i++)
        {
                for (int j = 0; j < came_from_.Width(); ++j)
                {
                        came_from_(j, i).cost = -1;
                }
        }
        current_ = start_;
        walker_ = end_;
        steps_ = 0;
        g_.clear();
        f_.clear();
        path_.clear();
        while (!open_queue_.empty())
        {
                open_queue_.pop();
        }
        closed_set_.clear();
        open_set_.clear();
        g_.insert(std::make_pair(start_, 0));
        f_.insert(std::make_pair(start_, Heuristic(start_, end_)));
        open_queue_.push(std::make_pair(Heuristic(start_, end_), start_));
        open_set_.insert(start_);
        parent_of_.clear();
}


int Astar::Heuristic(Coord a, Coord b)
{
        //Manhattan distance
        return std::abs((a.x - b.x)) + std::abs((a.y - b.y));
```

2.1.3.6 Maze.cpp

```
#include "Maze.h"
#include <algorithm>
#include <cassert>

Maze::Maze(int w, int h)
        :w_(w)
        ,h_(h)
        ,maze_(w * h)
        ,surface_(nullptr)
{
}

Maze::Maze()
        :w_(0)
        , h_(0)
        ,surface_(nullptr)
{}

Maze::~Maze()
```

```
{
        if (surface_ != nullptr)
        {
                SDL_FreeSurface(surface_);
        }
}

Maze::Maze(const Maze& other)
{
        InnerCopy(other);
}

Maze& Maze::operator=(const Maze& other)
{
        if (&other != this)
        {
                InnerCopy(other);
        }
        return *this;
}

void Maze::InnerCopy(const Maze& maze)
{
        w_ = maze.w_;
        h_ = maze.h_;;
        std::swap(maze_, std::vector<Tile>(h_ * w_));
        surface_ = nullptr;
}

void Maze::Render(Window& window, const int x, const int y, const int w, const int h)
{
        if (surface_ == nullptr)
        {
                surface_ = window.NewSurface(w_, h_);
        }

        int tile_w = w / w_;
        int tile_h = h / h_;

        for (int i = 0; i < h_; ++i)
        {
                for (int j = 0; j < w_; ++j)
                {
                        window.SetPixel(surface_, j, i, window.ToPixel(Get(j, i).color));
                }
        }
        window.RenderSurface(surface_, x, y, w, h);
}

Maze::Tile& Maze::operator()(int x, int y)
{
        return Get(x, y);
}

const Maze::Tile& Maze::operator()(int x, int y) const
{
        return Get(x, y);
}

Maze::Tile& Maze::Get(int x, int y)
{
```

```
        EnsureRange(x, y);
        return maze_[y * w_ + x];
}

const Maze::Tile& Maze::Get(int x, int y) const
{
        EnsureRange(x, y);;
        return maze_[y * w_ + x];
}


bool Maze::IsBorderCoord(const int x, const int y) const
{
        bool ret = (x <= 0 || y <= 0 || x >= w_- 1 || y >= h_ - 1);
        return ret;
}
```

2.1.3.7 PrimGenerator.cpp

```
#include "PrimGenerator.h"
#include <cassert>
#include <random>

PrimGenerator::PrimGenerator(const Coord& start)
        :steps_(0)
        ,start_(start.x + (1 - (start.x % 2)), start.x + (1 - (start.x % 2)))
{
        std::random_device rd;
        random_.Reseed(rd());
}

/* returns true if there are more steps */
bool PrimGenerator::Step(Maze& maze)
{
        if (steps_ == 0)
        {
                Clear(maze);
                TryAddWalls(start_, maze);
                maze(start_.x, start_.y).cost = 1;
                maze(start_.x, start_.y).color = Color::white();
        }

        if (walls_.size() == 0)
        {
                return false;
        }

        while (walls_.size() != 0)
        {
                int rand = random_.NextInt(0, walls_.size() - 1);
                Coord wall = walls_[rand];
                walls_[rand] = walls_.back();
                walls_.pop_back();

                auto horizontal = [this, &maze](const Coord& coord)
                {
                        return(maze(coord.x, coord.y - 1).cost == -
1 || maze(coord.x, coord.y + 1).cost == -1);
                };
                auto vertical = [this, &maze](const Coord& coord)
                {
```

```
                        return(maze(coord.x - 1, coord.y).cost == -
1 || maze(coord.x + 1, coord.y).cost == -1);
                };

                if (!vertical(wall) && !horizontal(wall))
                {
                        continue;
                }

                Coord node;
                bool vert = wall.x % 2 == 0;

                if (vert && vertical(wall))
                {
                        if (maze(wall.x - 1, wall.y).cost == -1)
                        {
                                node = Coord(wall.x - 1, wall.y);
                        }
                        else
                        {
                                node = Coord(wall.x + 1, wall.y);
                        }
                }
                else if (!vert && horizontal(wall))
                {
                        if (maze(wall.x, wall.y - 1).cost == -1)
                        {
                                node = Coord(wall.x, wall.y - 1);
                        }
                        else
                        {
                                node = Coord(wall.x, wall.y + 1);
                        }
                }
                else
                {
                        continue;
                }

                if (maze.IsBorderCoord(node.x, node.y))
                {
                        continue;
                }

                maze(node.x, node.y).cost = 1;
                maze(node.x, node.y).color = Color::white();
                maze(wall.x, wall.y).cost = 1;
                maze(wall.x, wall.y).color = Color::white();
                TryAddWalls(node, maze);
                break;
        }

        ++steps_;
        return walls_.size() != 0;
}

void PrimGenerator::Clear(Maze& maze)
{
        for (int i = 0; i < maze.Height(); ++i)
        {
                for (int j = 0; j < maze.Width(); ++j)
```

```
                {
                        auto& tile = maze(j, i);
                        tile.color = Color::black();
                        tile.cost = -1;
                }
        }
        walls_.clear();
}

void PrimGenerator::Reset(Maze& maze, const Coord& start)
{
        start_ = start;
        steps_ = 0;
}

void PrimGenerator::TryAddWalls(const Coord& coord, const Maze& maze)
{
        Coord coords[4] =
        {
                { coord.x - 1, coord.y }, { coord.x + 1, coord.y },
                { coord.x, coord.y - 1}, { coord.x, coord.y + 1 }
        };

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

        if (!maze.IsBorderCoord(coords[i].x, coords[i].y) && std::find(walls_.begin(), walls_.
end(), coords[i]) == walls_.end() && (maze(coords[i].x, coords[i].y).cost == -1))
                {
                        walls_.push_back(coords[i]);
                }
        }
}
```

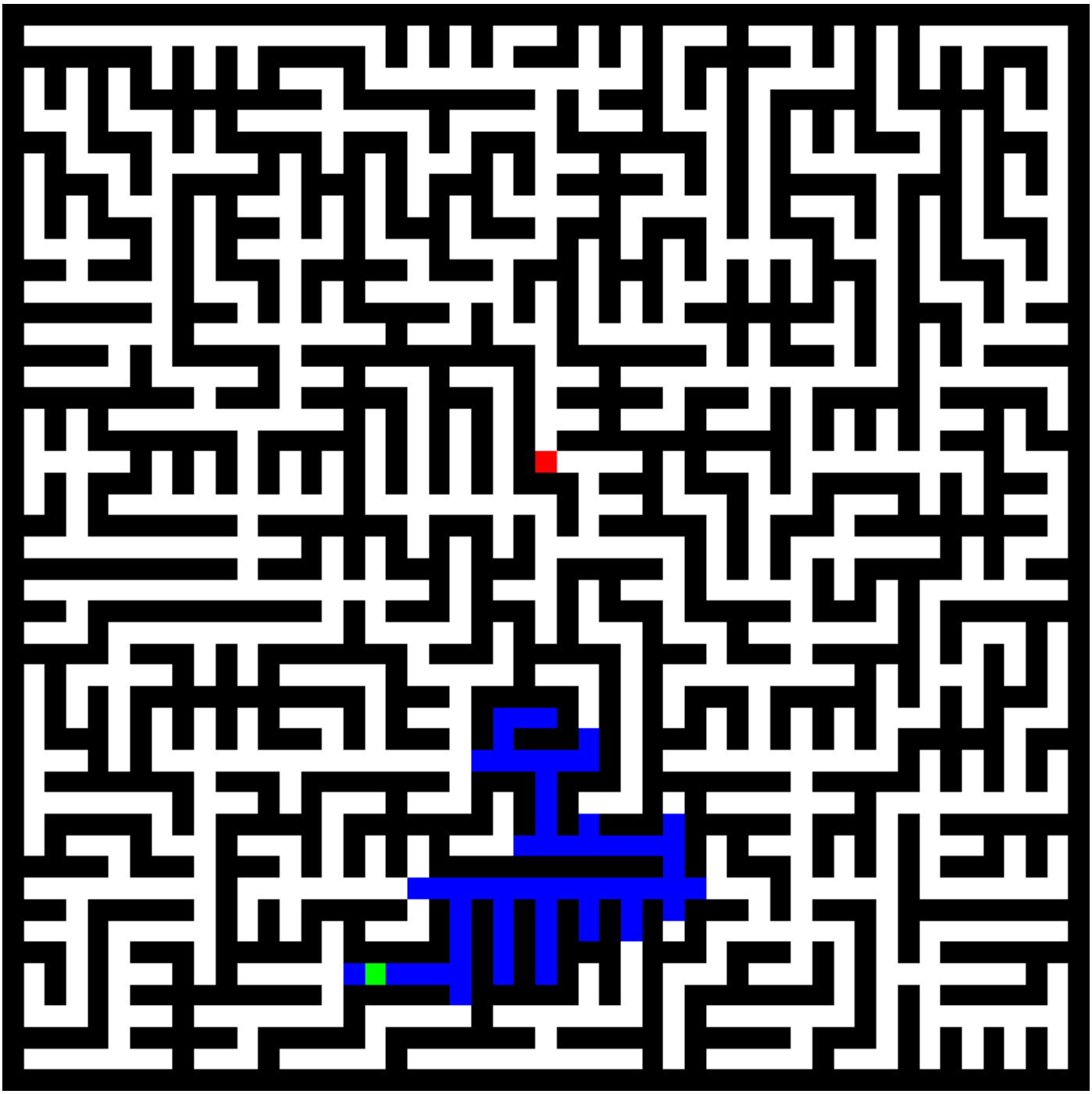
2.1.3.8 main.cpp

```
#include <SDL.h>
#include "Window.h"
#include "Timer.h"
#include "Maze.h"
#include "PrimGenerator.h"
#include "Coord.h"
#include "Astar/Astar.h"

int astar_main();

Coord RandomCoord(Randomizer& r, Maze& maze)
{
        Coord ret;
        ret = Coord(r.NextInt(1, maze.Width() - 3), r.NextInt(1, maze.Height() - 3));
        while (ret.x % 2 == 0 ||ret.y % 2 == 0)
                ret = Coord(r.NextInt(1, maze.Width() - 3), r.NextInt(1, maze.Height() - 3));
        return ret;
}

int main(int argc, char *argv[])
{
        return astar_main();
}

int astar_main()
{
```

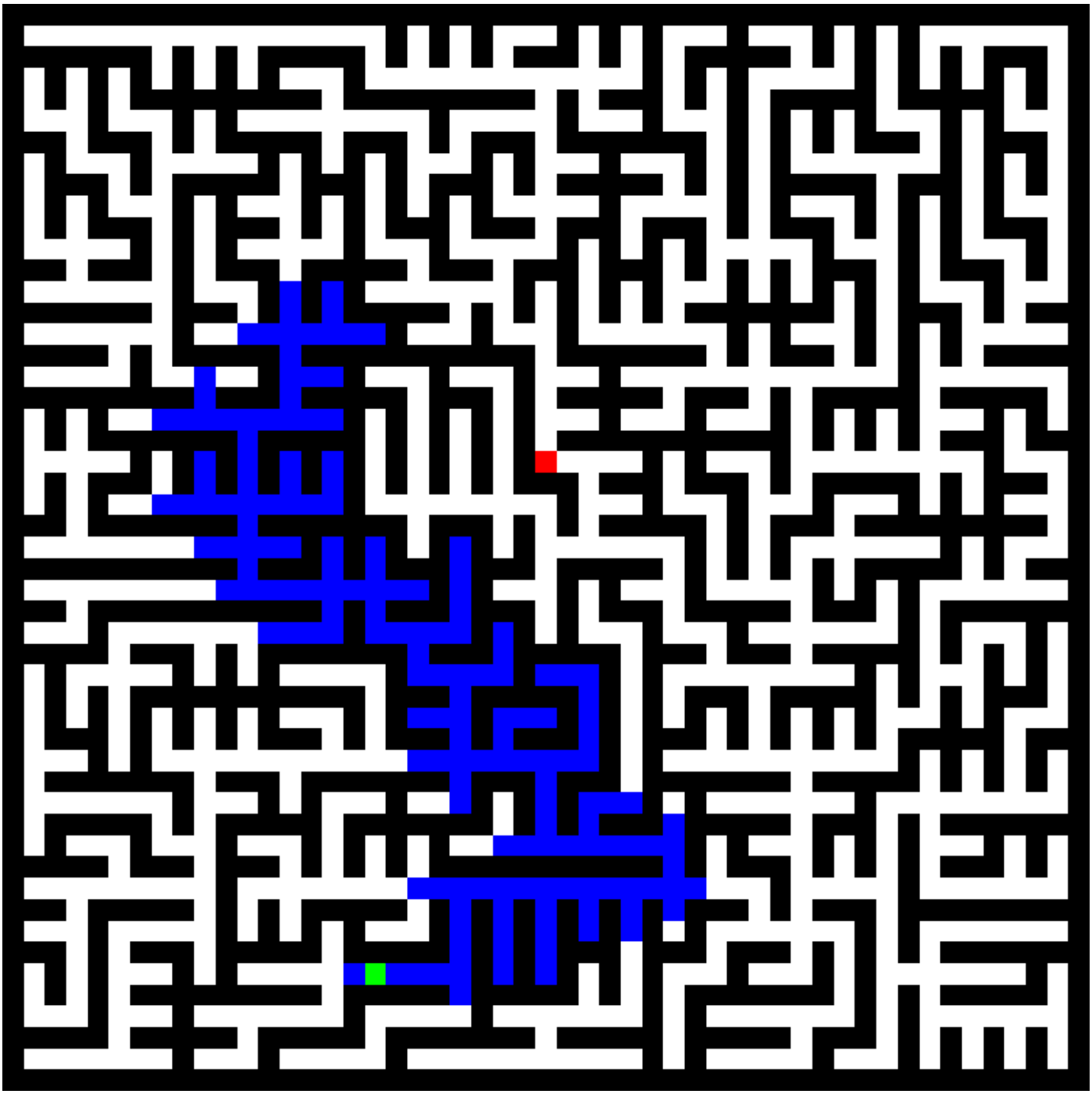
```
	Window win("Maze solver!", 800, 800);

	std::random_device rd;
	Randomizer rand(rd());

	Coord start, end;
	Maze maze(800, 800);
	PrimGenerator generator(RandomCoord(rand, maze));

	start = RandomCoord(rand, maze);
	end = RandomCoord(rand, maze);
	while (start == end)
		end = RandomCoord(rand, maze);

	Astar solver(start, end);

	long long step = 0;
	system("mkdir screens");
	while (win.Open())
	{
		win.PollEvents();

		if (!generator.Step(maze))
		{
			if (!solver.Step(maze))
			{
				maze = Maze(maze.Width() + 2, maze.Height() + 2);
				generator.Reset(maze, RandomCoord(rand, maze));

				start = RandomCoord(rand, maze);
				end = RandomCoord(rand, maze);
				while (end == start)
				{
					end = RandomCoord(rand, maze);
				}

			}
		}
		maze.Render(win, 0, 0, 800, 800);

		win.Display();

		step++;
		win.Clear();

	}
	return 0;
}
```

2.1.4 Körexempel

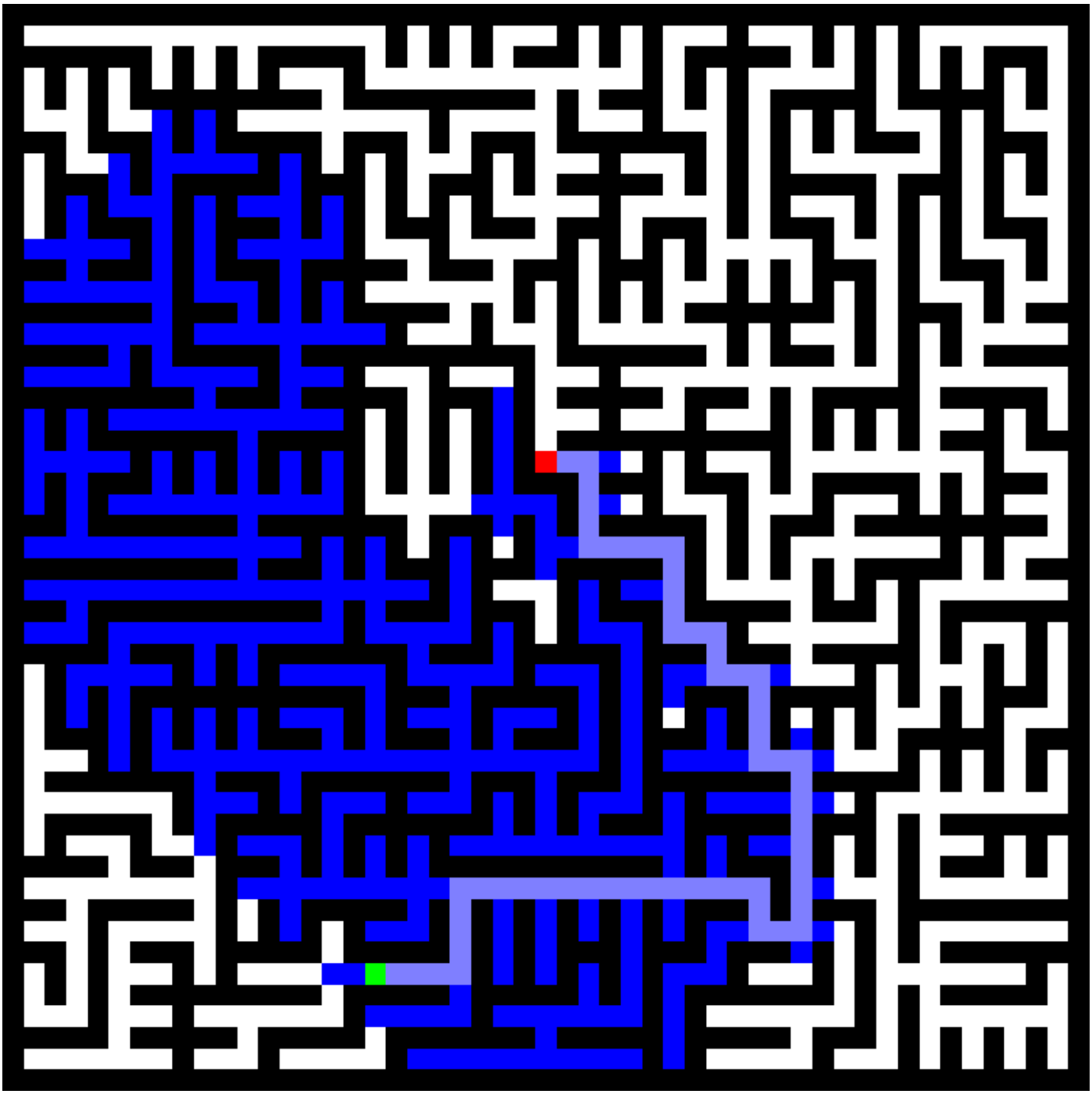
Figur 2. Labyrinten som ska lösas. Agenten ska hitta en väg från den gröna markören till den röda.



Figur 3. Blått representerar celler som har utforskats.



Figur 4. Här syns att agenten utforskar åt fel håll, den kommer att fortsätta göra det en längre tid



Figur 5. Agenten har hittat en väg. Som syns så har agenten sökt åt fel håll från början.

2.1.5 Analys

Lösningen är snabb och hittar rätt lösning på problemet. Ett problem med labyrinter är att de ofta kräver en lösning som tar stora omvägar, det är inte alls säkert att det kortaste manhattanavståndet ger en rätt bild av det hur många steg som krävs för att nå ett mål i labyrinten. Detta gör det svårare att hitta rätt väg i den här typen av miljö. Detta märks tydligt i körexemplet (se Figur 4) och blir extra tydligt för större labyrinter.

2.2 Evolutionära tekniker

2.2.1 Problem

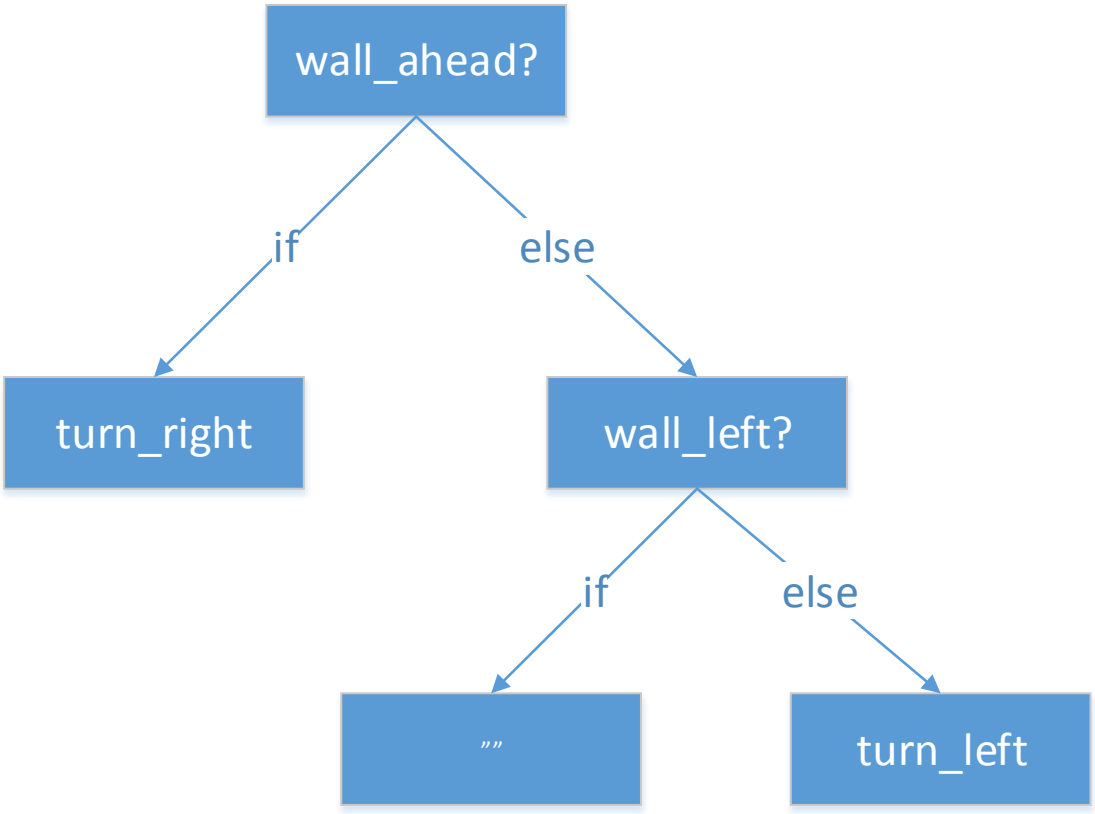
Problemet för är även här en labyrintlösning, med målet att se om labyrinten kan lösas på ett annat sätt än det tidigare exemplet (Söktekniker).

2.2.2 Design

Skapandet av labyrinten görs på samma sätt som förra problemet (Söktekniker), men labyrinten löses istället evolutionärt med genetisk programmering. Den evolutionära lösningen som tas fram är ett beteende som skapas av ett enkelt syntax-träd. Varje individ i populationen är en robot som exekverar en sådan kod-snutt, som alltså representeras av ett syntax-träd. Exekveringen görs inför varje steg och låter roboten ändra riktning beroende på vilka rutor runt den som är väggar, varpå roboten går ett steg framåt. Syntaxen består av följande:

1. 4 kommandon:
    a. ”turn_left”, vilket vrider roboten åt vänster.
    b. ”turn_right”, vilket vrider roboten åt höger.
    c. ””, vilket inte gör någonting
    d. ”if else”, vilket utför en av kommando-block utifrån om ett villkor är sant eller falskt (se nedan). Med hjälp av blanka kommandon kan koden uttrycka ett ”not”-villkor, genom en blank if, etc.
2. 3 villkor:
    a. ”wall_left”, sant om det finns en vägg direkt till vänster om roboten.
    b. ”wall_right”, sant om det finns en vägg direkt till höger om roboten.
    c. ”wall_forward”, sant om det finns en vägg direkt framför roboten.

Ett exempel på ett syntaxträd med ovan nämnda syntax skulle kunna se ut enligt nedan.



Figur 6. Exempel på ett syntaxträd för en robot.

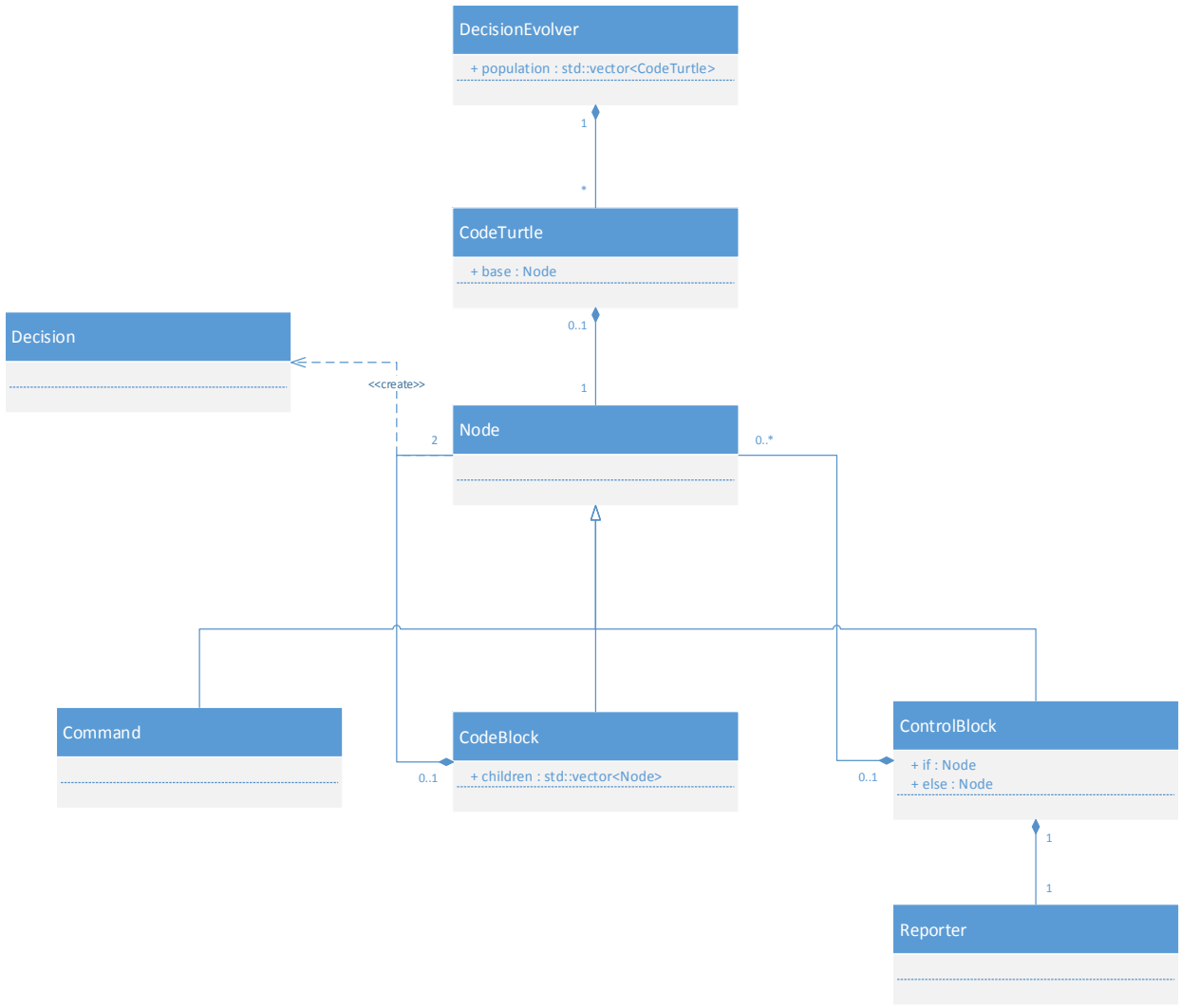
Ett sådant träd skulle översättas till kod som:

```
if (wall_ahead?)
    turn_right
else
    if (wall_left?)
    else
        turn_left
```

”if not” implementeras då som ett tom if-block, som i exemplet ovan som testar om det inte finns en vägg till vänster om roboten och i sådan fall vänder sig till vänster. Strategin som exemplet illustrerar är alltså att vända sig till höger om det finns en vägg framför roboten och i annat fall vända sig till vänster om det inte finns en vägg till vänster.

För att förhindra att en lösning tas fram som är specifik för bara en labyrint används samma population på flera labyrinter, och se om samma ”kod” kan användas för att lösa även andra labyrinter än den koden roboten har utvecklats för att lösa. Programmet startar därför med en liten labyrint. Varje gång en lösning tas fram skapas en ny labyrint som är lite större än den förra och samma population återanvänds.

En robot har ett begränsat antal steg på sig att lösa uppgiften, antalet steg beräknas från antalet totala rutor i labyrinten (båda väggar och gånger) dividerat på 2. Detta ger rätt så många steg, men även en mänskligt utvecklad kod kommer i värsta fall kräva att roboten går genom hela labyrinten för att hitta en lösning.

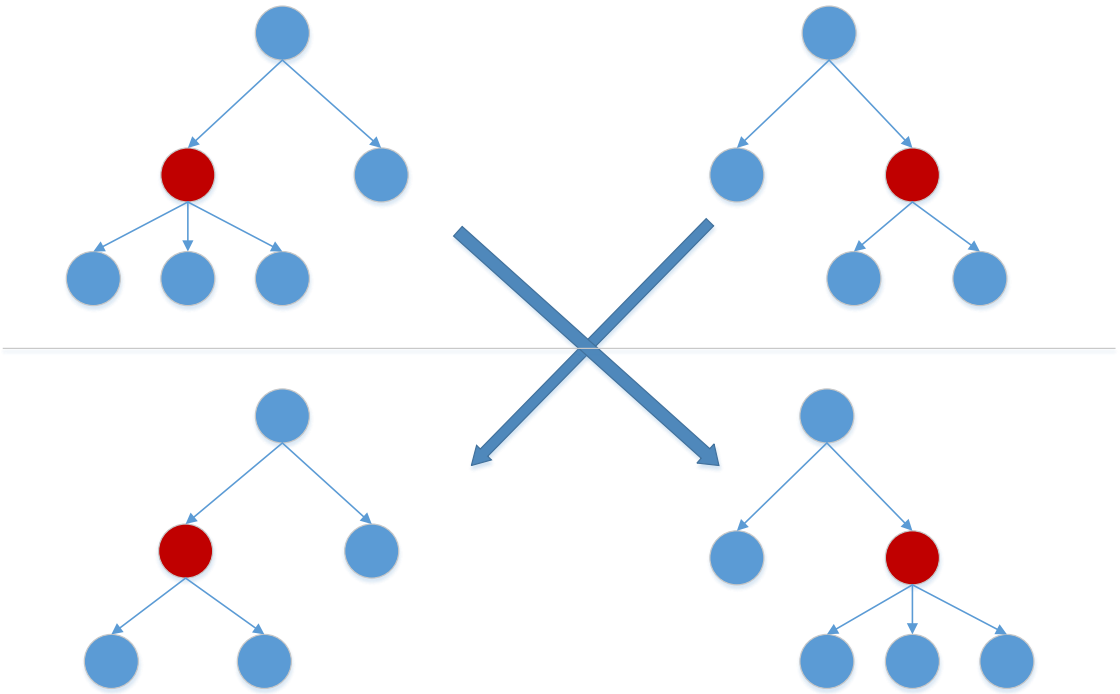


Figur 7. Klassdiagram över lösningen.

Figur 7 visar hur lösningen är designad med hjälp av ett klassdiagram. Det finns tre barnklasser till nodklassen, varav två har barnnoder. ControlBlock har en barnnod för if-blocket och en för else-blocket, medan Codeblock har ett godtyckligt antal. Rotnoden som finns i varje bot (CodeTurtle) är ingångspunkten för bottens ”kod”, vilken körs genom att kalla evaluate(). CodeBlock propagerar kallet till sina barn (i turordning). ControlBlock å andra sidan propagerar kallet till if-noden om reportern (vilket representerar kontrollvillkoret) evalueras till sant, i andra fall propageras kallet till else-noden. Command skapar rätt Decision beroende på vilken typ av kommando det är.

För att kunna mutera och överkorsa gener behövs ett sätt att välja en nod, vilket görs med funktionen select_node(). En nod väljs ut i stort sett slumpmässigt, men chansen är större att en nod högre upp i hierarkin väljs ut. Detta görs genom att både CodeBlock och ControlBlock har en viss chans att välja ett av barnen, eller låta ett av barnen välja en nod (genom att propagera kallet). Command väljer alltid ut sig själv.

För att överkorsa gener mellan två genom (bottar) väljs två noder ut i vardera genom. Dessa två noder byter plats, vilket visas i Figur 8.



Figur 8. Överkorsning

Mutation fungerar genom att mutera den valda genen, detta görs genom att kalla mutate() på noden. ControlBlock slumpar sitt vilkor och kallar mutate() på sina två barnnoder. Command slumpar sin typ. CodeBlock slumpar om ordningen på sina noder, och kan ta bort eller lägga till noder slumpmässigt (dock begränsat till ett visst antal noder som max). Djupet på trädet måste begränsas, dels för att programmet inte ska ta för lång tid, men också för att förhindra stack overflow.

Fitnessfunktionen är: antalet steg till målet från den position botten slutar plus antalet steg botten har gått. Med andra ord så kommer en bot som kommit fram till målet ha så stor fitness som det antal steg den tog, medan en bot som inte nått fram har en fitness motsvarande avstånd till mål plus max antal tillåtna steg. En högre fitness betyder alltså att botten är mindre ”bra”. Avståndet i fitnessfunktion är manhattanavstånd.

2.2.3 Kod

2.2.3.1 CodeBlock.h

```
#pragma once

#include "Decision.h"
#include <vector>
```

```
#include <memory>
#include "Node.h"

class Maze;
class Randomizer;

class CodeBlock: public Node
{
public:
        typedef std::shared_ptr<CodeBlock> ptr;
        static ptr make();

        CodeBlock();

        CodeBlock(const CodeBlock&) = delete;
        CodeBlock& operator=(const CodeBlock&) = delete;
        CodeBlock(CodeBlock&&) = delete;
        CodeBlock& operator=(CodeBlock&&) = delete;

        Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision) override
;
        Node::ptr clone() override;
        Node::ptr select_node(Randomizer& random) override;
        void mutate(Randomizer& random, int depth) override;
        void replace(Node::ptr& from, Node::ptr& to) override;
        void format(Formatter& format) override;
        int max_depth() override;

private:

        std::vector<Node::ptr> children_;
};
```

2.2.3.2 Command.h

```
#pragma once

#include "Decision.h"
#include <vector>
#include <memory>
#include "Node.h"

class Maze;
class Randomizer;

class Command : public Node
{
public:
        enum Type { BLANK, TURN_LEFT, TURN_RIGHT};
        Command(Type type);

        typedef std::shared_ptr<Command> ptr;
        static ptr make(Type type);

        Command(const Command&) = delete;
        Command& operator=(const Command&) = delete;
        Command(Command&&) = delete;
        Command& operator=(Command&&) = delete;

        Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision) override
;
        Node::ptr clone() override;
```

```
        Node::ptr select_node(Randomizer& random) override;
        void mutate(Randomizer& random, int depth) override;
        void replace(Node::ptr& from, Node::ptr& to) override;
        void format(Formatter& format) override;
        int max_depth() override;

private:

        Type type_;
};
```

2.2.3.3 ControlBlock.h

```
#pragma once

#include "Decision.h"
#include <vector>
#include <memory>
#include "Node.h"
#include "Reporter.h"

class Maze;
class Randomizer;

class ControlBlock : public Node
{
public:
        typedef std::shared_ptr<ControlBlock> ptr;
        static ptr make();

        ControlBlock();

        ControlBlock(const ControlBlock&) = delete;
        ControlBlock& operator=(const ControlBlock&) = delete;
        ControlBlock(ControlBlock&&) = delete;
        ControlBlock& operator=(ControlBlock&&) = delete;

        Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision) override
;
        Node::ptr clone() override;
        Node::ptr select_node(Randomizer& random) override;
        void mutate(Randomizer& random, int depth) override;
        void replace(Node::ptr& from, Node::ptr& to) override;
        void format(Formatter& format) override;
        int max_depth() override;

private:

        Node::ptr if_;
        Node::ptr else_;
        Reporter::ptr reporter_;
};
```

2.2.3.4 Decision.h

```
#pragma once
#include "../Coord.h"

struct Decision
{
public:
        enum Action { TURN_LEFT = -1, TURN_RIGHT = 1 };
```

```
        enum Direction { UP, RIGHT, DOWN, LEFT };

        Decision(Direction direction);
        Decision();

        Coord forward(Coord coord);
        Decision perform(Action action);

        Direction direction() const { return direction_; }
private:
        Direction direction_;
};
```

2.2.3.5 Formatter.h

```
#pragma once

#include <sstream>

class Formatter
{
public:
        Formatter();
        void append(std::string line);
        void push_indent();
        void pop_indent();

        std::string to_string();
        void clear();
private:
        std::stringstream ss_;
        int indent_;
};
```

2.2.3.6 Node.h

```
#pragma once

#include "../Coord.h"
#include "Decision.h"
#include <vector>
#include <memory>
#include "Formatter.h"

class Maze;
class Randomizer;

class Node : std::enable_shared_from_this<Node>
{
public:
        typedef std::shared_ptr<Node> ptr;
        //returns a random node (with no child nodes)
        static ptr make_random(Randomizer& random);

        Node(const Node&) = delete;
        Node& operator=(const Node&) = delete;
        Node(Node&&) = delete;
        Node& operator=(Node&&) = delete;

        virtual Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision)
= 0;
        virtual Node::ptr clone() = 0;
```

```
        virtual void mutate(Randomizer& random, int depth) = 0;
        virtual Node::ptr select_node(Randomizer& random) = 0;
        virtual void replace(Node::ptr& from, Node::ptr& to) = 0;
        virtual void format(Formatter& format) = 0;
        virtual int max_depth() = 0;

        Node::ptr get_this();
protected:
        Node() {};
};
```

2.2.3.7 Reporter.h

```
#pragma once

#include "../Coord.h"
#include <memory>
#include "Decision.h"
#include <string>

class Maze;
class Randomizer;

class Reporter
{
public:
        typedef std::unique_ptr<Reporter> ptr;
        static ptr make(Randomizer& random);

        virtual ~Reporter() {};
        virtual bool evaluate(Maze& maze, Coord current_coord, Decision decision) = 0;
        virtual ptr clone() = 0;
        virtual std::string to_string() const = 0;
protected:
        bool evaluate_helper(Maze& maze, Coord current_coord, Decision decision);
};

class WallAhead : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
        ptr clone() override;
};

class WallLeft : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
        ptr clone() override;
};

class WallRight : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
        ptr clone() override;
};

class TrueReporter : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
```

```
        ptr clone() override;
};
```

2.2.3.8 CodeTurtle.h

```
#pragma once

#include "Node.h"
#include "Decision.h"

class CodeTurtle
{
public:
        CodeTurtle();

        //returns amount of steps since latest start position reset
        int step(Maze& maze);
        void set_start_position(Coord coord);

        const Node::ptr& get_base() const;
        Node::ptr& get_base();

        Decision get_current_decision() const;
        Coord get_current_coord() const;

        void set_fitness(int fitness) { fitness_ = fitness; }
        int get_fitness() const { return fitness_; }
private:
        int steps_;
        int fitness_;
        Node::ptr base_;
        Decision current_decision_;
        Coord current_coord_;
};
```

2.2.3.9 DecisionEvolver.h

```
#pragma once

#include "CodeTurtle.h"
#include "../Random.h"
#include "../Coord.h"
#include "../Solver.h"
#include <utility>

class DecisionEvolver : public Solver
{
public:
        DecisionEvolver();
        DecisionEvolver(Coord start, Coord end, int max_turtle_steps, int population_size, dou
ble combination_probability, double mutation_probability);

        bool Step(Maze& maze) override;
        void Reset() override;
        void Reset(Coord start, Coord end) override;
        void DecisionEvolver::NewMaze(Coord start, Coord end, int max_steps);

private:
        void crossover(CodeTurtle&, CodeTurtle&);
        void mutate(CodeTurtle& n);

        int calculate_fitness(CodeTurtle& g, Maze& maze);
```

```
        void calculate_fitness(Maze& maze);
        void sort_by_fitness();

        void evolve(Maze& maze);

        int distance(Coord a, Coord b);

        Randomizer rand_;
        std::vector<CodeTurtle> population_;
        CodeTurtle& select();

        int population_size_;
        float combination_p_, mutation_p_;
        Coord start_, end_;
        int steps_, max_turtle_steps_;
};
```

2.2.3.10 CodeBlock.cpp

```
#include "CodeBlock.h"
#include "../Random.h"
#include <algorithm>

CodeBlock::ptr CodeBlock::make()
{
        return std::make_shared<CodeBlock>();
}

CodeBlock::CodeBlock()
{

}

Decision CodeBlock::evaluate(Maze& maze, Coord current_coord, Decision current_decision)
{
        for (Node::ptr n : children_)
        {
                current_decision = n->evaluate(maze, current_coord, current_decision);
        }
        return current_decision;
}

Node::ptr CodeBlock::select_node(Randomizer& random)
{
        //select a child at random

        int index = random.NextInt(0, children_.size());

        //Only select self if there are no children or once in every lenght of child.
        if (index == 0 || children_.size() == 0)
        {
                return get_this();
        }

        Node::ptr s = children_[index - 1];
        //select that child half the time, select a childs child half the time.
        if (random.NextInt(0, 2) == 0)
        {
                return s;
        }
        else
        {
```

```
                return s->select_node(random);
        }
}

Node::ptr CodeBlock::clone()
{
        CodeBlock::ptr copy = std::make_shared<CodeBlock>();
        for (Node::ptr n : children_)
        {
                copy->children_.push_back(std::move(n->clone()));
        }
        return std::move(copy);
}

void CodeBlock::mutate(Randomizer& random, int depth)
{
        int size = children_.size();
        if (size != 0)
        {
                int size_change = random.NextInt(-2 - (2 - children_.size()), 2 - (2 -
 children_.size()));

                auto rand = [this, &random, size](int i){return random.NextInt(0, i - 1); };

                //shuffle nodes
                try
                {
                        std::random_shuffle(children_.begin(), children_.end(), rand);
                }
                catch (...)
                {
                        throw;
                }

        int rem = std::min(static_cast<int>(children_.size()), std::max(0, rand(size) + size_c
hange));
                //erase a random amount of nodes
                children_.erase(children_.begin() + rem, children_.end());
                //mutate some of the remaining children
                for (auto& child : children_)
                {
                        if (random.NextInt(0, 1) == 0)
                        {
                                if (random.NextInt(0, 1) == 0)
                                {
                                        child = Node::make_random(random);
                                }

                                if (depth > 0)
                                {
                                        child->mutate(random, depth - 1);
                                }
                        }
                }
                //add a random amount of nodes, such that on average there are as many
                //nodes left as there were nodes to start with.
                for (int i = 0; i < rand(size) + size_change; ++i)
                {
                        Node::ptr c = Node::make_random(random);
                        if (depth > 0)
                        {
```

```
                                c->mutate(random, depth - 1);
                        }
                        children_.push_back(c);
                }

        }
        else
        {
                int size = random.NextInt(0, 2);
                for (int i = 0; i < size; ++i)
                {
                        //if no nodes, just add a random node.
                        Node::ptr c = Node::make_random(random);
                        if (depth > 0)
                        {
                                c->mutate(random, depth - 1);
                        }
                        children_.push_back(c);
                }
        }
}

void CodeBlock::replace(Node::ptr& from, Node::ptr& to)
{
        //can't replace self
        if (this != from.get())
        {
                for (Node::ptr& c : children_)
                {
                        if (c == from)
                        {
                                c = to;
                                break;
                        }
                        c->replace(from, to);
                }
        }
}

void CodeBlock::format(Formatter& format)
{
        for (Node::ptr& n : children_)
        {
                n->format(format);
        }
}

int CodeBlock::max_depth()
{
        int max = 0;
        for (Node::ptr& n : children_)
        {
                max = std::max(n->max_depth(), max);
        }
        return max + 1;
}
```

2.2.3.11 Command.cpp

```
#include "Command.h"
#include "../Random.h"
#include <algorithm>
```

```
Command::ptr Command::make(Command::Type type)
{
	return std::make_shared<Command>(type);
}


Command::Command(Command::Type type)
{
	type_ = type;
}

Decision Command::evaluate(Maze& maze, Coord current_coord, Decision current_decision)
{
	switch (type_)
	{
	case Command::BLANK:
		break;
	case Command::TURN_LEFT:
		current_decision.perform(Decision::TURN_LEFT);
		break;
	case Command::TURN_RIGHT:
		current_decision.perform(Decision::TURN_RIGHT);
		break;
	default:
		break;
	}
	return current_decision;
}

Node::ptr Command::select_node(Randomizer& random)
{
	return get_this();
}

Node::ptr Command::clone()
{
	return std::move(Command::make(type_));
}

void Command::mutate(Randomizer& random, int depth)
{
	int blank = random.NextInt(0, 1);
	type_ = static_cast<Command::Type>(random.NextInt(blank, 2));
}

void Command::replace(Node::ptr& from, Node::ptr& to)
{

}

void Command::format(Formatter& format)
{
	switch (type_)
	{
	case Command::BLANK:
		break;
	case Command::TURN_LEFT:
		format.append("turn_left");
		break;
	case Command::TURN_RIGHT:
```

```
                format.append("turn_right");
                break;
        default:
                break;
        }
}

int Command::max_depth()
{
        return 0;
}
```

2.2.3.12 ControlBlock.cpp

```
#include "ControlBlock.h"
#include "../Random.h"
#include <algorithm>
#include "Command.h"

ControlBlock::ptr ControlBlock::make()
{
        return std::make_shared<ControlBlock>();
}


ControlBlock::ControlBlock()
{
        reporter_ = std::make_unique<TrueReporter>();
        if_ = Command::make(Command::BLANK);
        else_ = Command::make(Command::BLANK);
}

Decision ControlBlock::evaluate(Maze& maze, Coord current_coord, Decision current_decision)
{
        if (reporter_->evaluate(maze, current_coord, current_decision))
        {
                return if_->evaluate(maze, current_coord, current_decision);
        }
        else
        {
                return else_->evaluate(maze, current_coord, current_decision);
        }

}

Node::ptr ControlBlock::select_node(Randomizer& random)
{
        int r = random.NextInt(0, 2);
        if (r == 0)
        {
                return get_this();
        }
        else if (r == 1)
        {
                return if_;
        }
        else
        {
                return else_;
        }
}
```

```
Node::ptr ControlBlock::clone()
{
	ControlBlock::ptr copy = std::make_shared<ControlBlock>();
	copy->reporter_ = reporter_->clone();
	copy->if_ = if_->clone();
	copy->else_ = else_->clone();
	return std::move(copy);
}

void ControlBlock::mutate(Randomizer& random, int depth)
{
	//half the time randomize the reporter
	if (random.NextInt(0, 1) == 0 || dynamic_cast<TrueReporter*>(reporter_.get()) != nullp
tr)
	{
		reporter_ = Reporter::make(random);
	}
	//half the time randomize the if
	if (random.NextInt(0, 1) == 0)
	{
		if (random.NextInt(0, 1) == 0)
		{
			if_ = Node::make_random(random);
		}
		if (depth > 0)
		{
			if_->mutate(random, depth - 1);
		}
	}
	//half the time randomize the else
	if (random.NextInt(0, 1) == 0)
	{
		//half the time randomize a type
		if (random.NextInt(0, 1) == 0)
		{
			else_ = Node::make_random(random);
		}
		if (depth > 0)
		{
			else_->mutate(random, depth);
		}
	}
}

void ControlBlock::replace(Node::ptr& from, Node::ptr& to)
{
	//can't replace self
	if (this != from.get())
	{
		if (if_ == from)
		{
			if_ = to;
		}
		else if (else_ == from)
		{
			else_ = to;
		}
		else
		{
			if_->replace(from, to);
			else_->replace(from, to);
```

```
                }

        }
}

void ControlBlock::format(Formatter& format)
{
        format.append("if (" + reporter_->to_string() + "?)");
        format.push_indent();
        if_->format(format);
        format.pop_indent();
        format.append("else");
        format.push_indent();
        else_->format(format);
        format.pop_indent();
}

int ControlBlock::max_depth()
{
        return std::max(else_->max_depth(), if_->max_depth()) + 1;
}
```

2.2.3.13 Decision.cpp

```
#include "Decision.h"

Decision::Decision(Decision::Direction direction)
{
        direction_ = direction;
}

Decision::Decision()
{
        direction_ = Decision::UP;
}

Coord Decision::forward(Coord coord)
{
        switch (direction_)
        {
        case Decision::UP:
                return Coord(coord.x, coord.y - 1);
                break;
        case Decision::RIGHT:
                return Coord(coord.x + 1, coord.y);
                break;
        case Decision::DOWN:
                return Coord(coord.x, coord.y + 1);
                break;
        case Decision::LEFT:
                return Coord(coord.x - 1, coord.y);
                break;
        default:
                return coord;
                break;
        }
}

Decision Decision::perform(Decision::Action action)
{
        int n = static_cast<int>(direction_)+static_cast<int>(action);
        if (n < 0)
```

```
        {
                n = 3;
        }
        direction_ = static_cast<Direction>(n % 4);
        return *this;
}
```

2.2.3.14 Formatter.cpp

```
#include "Formatter.h"
#include <exception>

Formatter::Formatter()
        :indent_(0)
{}

void Formatter::append(std::string line)
{
        for (int i = 0; i < indent_; ++i)
        {
                ss_ << "    ";
        }
        ss_ << line << "\n";
}

void Formatter::push_indent()
{
        indent_++;
}

void Formatter::pop_indent()
{
        if (indent_ == 0)
        {
                throw std::out_of_range("Too many pops");
        }
        indent_--;
}

std::string Formatter::to_string()
{
        return ss_.str();
}

void Formatter::clear()
{
        ss_.clear();
}
```

2.2.3.15 Node.cpp

```
#include "Node.h"
#include "../Random.h"
#include "CodeBlock.h"
#include "ControlBlock.h"
#include "Command.h"

Node::ptr Node::get_this()
{
        return shared_from_this();
}
```

```
Node::ptr Node::make_random(Randomizer& random)
{
	int r = random.NextInt(0, 5);
	switch (r)
	{
	case 0:
		return ControlBlock::make();
	case 1:
		return CodeBlock::make();
	case 2:
		return CodeBlock::make();
	case 3:
		return Command::make(Command::BLANK);
	case 4:
		return Command::make(Command::BLANK);
	case 5:
		return Command::make(Command::BLANK);
	default:

		//supress warning
		return nullptr;
	}
}
```

2.2.3.16 Reporter.cpp

```
#include "Reporter.h"
#include "../Random.h"
#include "../Maze.h"

Reporter::ptr Reporter::make(Randomizer& random)
{
	int r = random.NextInt(0, 2);
	switch (r)
	{
	case 0:
		return std::move(std::make_unique<WallAhead>());
	case 1:
		return std::move(std::make_unique<WallLeft>());
	case 2:
		return std::move(std::make_unique<WallRight>());
	default:
		return nullptr;
	}
}

Reporter::ptr WallAhead::clone()
{
	return std::move(std::make_unique<WallAhead>());
}

Reporter::ptr WallLeft::clone()
{
	return std::move(std::make_unique<WallLeft>());
}

Reporter::ptr WallRight::clone()
{
	return std::move(std::make_unique<WallRight>());
}

Reporter::ptr TrueReporter::clone()
```

```
{
        return std::move(std::make_unique<TrueReporter>());
}

bool Reporter::evaluate_helper(Maze& maze, Coord current_coord, Decision decision)
{
        auto dir = decision.direction();
        switch (dir)
        {
        case Decision::UP:
                return maze(current_coord.x, current_coord.y - 1).cost == -1;
        case Decision::RIGHT:
                return maze(current_coord.x + 1, current_coord.y).cost == -1;
        case Decision::DOWN:
                return maze(current_coord.x, current_coord.y + 1).cost == -1;
        case Decision::LEFT:
                return maze(current_coord.x - 1, current_coord.y).cost == -1;
        default:
                return false;
        }
}

bool WallAhead::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        return evaluate_helper(maze, current_coord, decision);
}

bool WallLeft::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        decision.perform(Decision::TURN_LEFT);
        return evaluate_helper(maze, current_coord, decision);
}

bool WallRight::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        decision.perform(Decision::TURN_RIGHT);
        return evaluate_helper(maze, current_coord, decision);
}

bool TrueReporter::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        return true;
}

std::string WallAhead::to_string() const
{
        return "wall_ahead";
}

std::string WallLeft::to_string() const
{
        return "wall_left";
}

std::string WallRight::to_string() const
{
        return "wall_right";
}

std::string TrueReporter::to_string() const
{
```

```
        return "true";
}
```

2.2.3.17 CodeTurtle.cpp

```
#include "CodeTurtle.h"
#include "Command.h"
#include "../Maze.h"

CodeTurtle::CodeTurtle()
{
        base_ = Command::make(Command::BLANK);
        steps_ = 0;
}

int CodeTurtle::step(Maze& maze)
{
        current_decision_ = base_->evaluate(maze, current_coord_, current_decision_);
        Coord next = current_decision_.forward(current_coord_);

        if (maze(next.x, next.y).cost != -1)
        {
                current_coord_ = next;
        }
        ++steps_;
        return steps_;
}

void CodeTurtle::set_start_position(Coord coord)
{
        current_decision_ = Decision();
        steps_ = 0;
        current_coord_ = coord;
}

const Node::ptr& CodeTurtle::get_base() const
{
        return base_;
}

Node::ptr& CodeTurtle::get_base()
{
        return base_;
}

Decision CodeTurtle::get_current_decision() const
{
        return current_decision_;
}

Coord CodeTurtle::get_current_coord() const
{
        return current_coord_;
}
```

2.2.3.18 DecisionEvolver.cpp

```
#include "DecisionEvolver.h"
#include "../Maze.h"

DecisionEvolver::DecisionEvolver()
        :population_size_(0)
```

```
        ,combination_p_(0.0f)
        ,mutation_p_(0.0f)
        ,steps_(0)
        ,max_turtle_steps_(0)
{
}

DecisionEvolver::DecisionEvolver(Coord start, Coord end, int max_turtle_steps, int population_
size, double combination_probability, double mutation_probability)
        :population_size_(population_size)
        , combination_p_(combination_probability)
        , mutation_p_(mutation_probability)
        , steps_(0)
        , start_(start)
        , end_(end)
        , max_turtle_steps_(max_turtle_steps)
{

}

bool DecisionEvolver::Step(Maze& maze)
{
        if (steps_ == 0)
        {
                for (int i = 0; i < population_size_; ++i)
                {
                        population_.push_back(CodeTurtle());

                        CodeTurtle& turtle = population_[i];
                        turtle.get_base() = Node::make_random(rand_);
                        turtle.get_base()->mutate(rand_, 5);
                }
        }

        /*Formatter format;

        format.append("Population");
        format.push_indent();

        for (int i = 0; i < population_.size(); ++i)
        {

        format.append(std::to_string(i) + "(" + std::to_string(population_[i].get_fitness()) +
 "):");
                format.push_indent();
                population_[i].get_base()->format(format);
                format.pop_indent();
        }
        format.pop_indent();

        std::cout << format.to_string() << std::endl;*/


        if (steps_ % 2 == 0)
        {
                evolve(maze);

                /*for (auto& bot : population_)
                {
                        std::cout << bot.get_fitness() << ", ";
                }
```

```
                std::cout << std::endl;*/
        }
        else
        {
                for (int i = 0; i < maze.Height(); ++i)
                {
                        for (int j = 0; j < maze.Width(); ++j)
                        {
                                if (maze(j, i).cost != -1)
                                {
                                        maze(j, i).color = Color::white();
                                }
                        }
                }

                for (auto& bot : population_)
                {
                        bot.set_start_position(start_);
                        while (bot.step(maze) < max_turtle_steps_)
                        {
                                if (bot.get_current_coord() == end_)
                                {
                                        Formatter format;

        format.append("Good bot! (" + std::to_string(bot.get_fitness()) + ")");
                                        format.push_indent();

                                        format.push_indent();
                                        bot.get_base()->format(format);
                                        format.pop_indent();

                                        format.pop_indent();
                                        std::cout << format.to_string() << std::endl;
                                        return false;
                                }
                        }

                        Coord c = bot.get_current_coord();
                        maze(c.x, c.y).color = Color::blue();
                }
                maze(start_.x, start_.y).color = Color::green();
                maze(end_.x, end_.y).color = Color::red();

        }
        steps_++;

        return true;
}

void DecisionEvolver::evolve(Maze& maze)
{
        calculate_fitness(maze);
        sort_by_fitness();

        std::vector<CodeTurtle> child_population;
        while (child_population.size() < population_size_)
        {
                Formatter format;

                CodeTurtle& c0 = select();
                CodeTurtle& c1 = select();
```

```
                if (rand_.NextBool(combination_p_))
                {
                        crossover(c0, c1);
                }
                if (rand_.NextBool(mutation_p_))
                {
                        mutate(c0);
                }
                if (rand_.NextBool(mutation_p_))
                {
                        mutate(c1);
                }

                child_population.push_back(c0);
                child_population.push_back(c1);
        }
        population_ = child_population;

        calculate_fitness(maze);
}

void DecisionEvolver::Reset()
{
        steps_ = 0;
        population_.clear();
}

void DecisionEvolver::NewMaze(Coord start, Coord end, int max_steps)
{
        start_ = start;
        end_ = end;
        max_turtle_steps_ = max_steps;
}

void DecisionEvolver::Reset(Coord start, Coord end)
{
        start_ = start;
        end_ = end;
        Reset();
}

CodeTurtle& DecisionEvolver::select()
{
        //tournament selection
        auto rind = [this]() { return rand_.NextInt(0, population_.size() - 1); };

        int best = rind();

        for (int i = 0; i < population_.size() - 1; ++i)
        {
                int curr = rind();
                if (population_[curr].get_fitness() < population_[best].get_fitness())
                {
                        best = curr;
                }
        }
        return population_[best];
}

void DecisionEvolver::sort_by_fitness()
```

```
{
        std::sort(population_.begin(), population_.end(), [&](const CodeTurtle& g0, const Code
Turtle& g1) {return g0.get_fitness() < g1.get_fitness(); });
}

int DecisionEvolver::calculate_fitness(CodeTurtle& g, Maze& maze)
{
        g.set_start_position(start_);
        for (int i = 0; i < max_turtle_steps_; ++i)
        {
                if (g.get_current_coord() == end_)
                {
                        return i;
                }
                g.step(maze);
        }
        while (g.step(maze) <= max_turtle_steps_)
                ;

        return max_turtle_steps_ + distance(g.get_current_coord(), end_);
}

void DecisionEvolver::calculate_fitness(Maze& maze)
{
        for (CodeTurtle& gene : population_)
        {
                int fitness = calculate_fitness(gene, maze);
                gene.set_fitness(fitness);
        }
}

void DecisionEvolver::crossover(CodeTurtle& t0, CodeTurtle& t1)
{
        Node::ptr b0 = t0.get_base();
        Node::ptr b1 = t1.get_base();
        Node::ptr s0 = b0->select_node(rand_);
        Node::ptr s1 = b1->select_node(rand_);
        if (b0 == s0)
        {
                b0 = b1->clone();
        }
        else
        {
                int max_d = b0->max_depth();
                int select_d = max_d - s1->max_depth();
                //Force the code to a certain depth
                if (max_d - select_d >= 5)
                {
                        Node::ptr n = Node::make_random(rand_);
                        n->mutate(rand_, std::max(0, std::min(2, 5 - select_d)));
                        b0->replace(s0, n);
                }
                else
                {
                        b0->replace(s0, s1->clone());
                }
        }
        t0.get_base() = b0;

        if (b1 == s1)
        {
```

```
                b1 = b0->clone();
        }
        else
        {
                int max_d = b1->max_depth();
                int select_d = max_d - s0->max_depth();
                //Force the code to a certain depth
                if (max_d - select_d > 5)
                {
                        Node::ptr n = Node::make_random(rand_);
                        n->mutate(rand_, std::max(0, std::min(2, 5 - select_d)));
                        b1->replace(s0, n);
                }
                else
                {
                        b1->replace(s1, s0->clone());
                }
        }
        t1.get_base() = b1;
}

void DecisionEvolver::mutate(CodeTurtle& t)
{
        Node::ptr s = t.get_base()->select_node(rand_);
        Node::ptr n = s->clone();

        int max_d = t.get_base()->max_depth();
        int select_d = max_d - s->max_depth();

        if (rand_.NextInt(0, 1) == 1 || max_d - select_d >= 5)
        {
                n = Node::make_random(rand_);
        }

        n->mutate(rand_, std::max(0, std::min(2, 5 - select_d)));
        if (s == t.get_base())
        {
                t.get_base() = n;
        }
        else
        {
                t.get_base()->replace(s, n);
        }
}

int DecisionEvolver::distance(Coord a, Coord b)
{
        return abs(a.x - b.x) + abs(a.y - b.y);
}
```

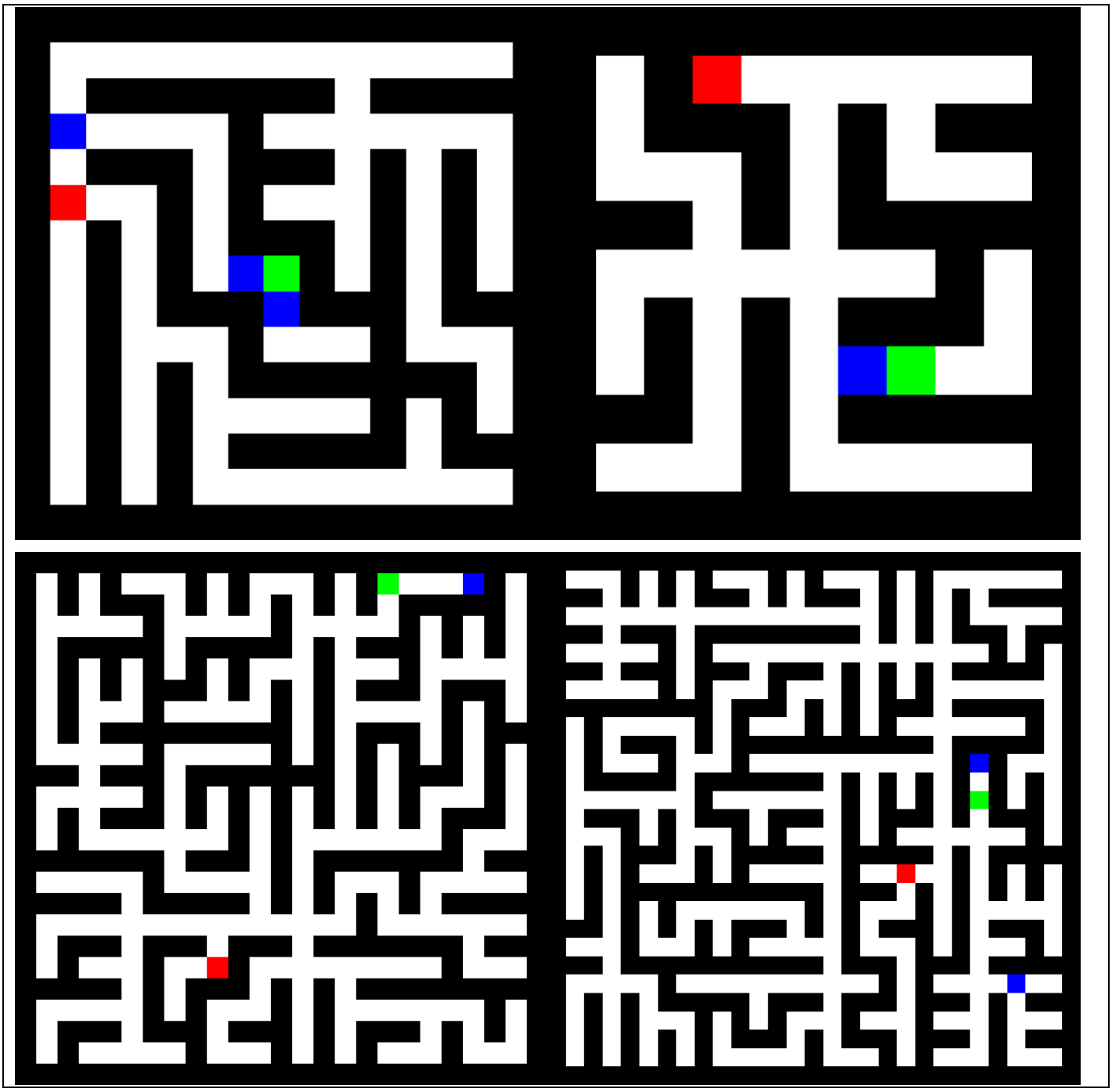
2.2.4 Körexempel

Körexemplet baseras på följande funktion:

```
int decision_based_main()
{
	Window win("Maze solver!", 800, 800);

	std::random_device rd;
	Randomizer rand(rd());

	Coord start, end;
	Maze maze(11, 11);
	PrimGenerator generator(RandomCoord(rand, maze));

	start = RandomCoord(rand, maze);
	end = RandomCoord(rand, maze);
	while (start == end)
		end = RandomCoord(rand, maze);

	DecisionEvolver solver(start, end, maze.Width() * maze.Height(), 100, 0.5f, 0.01f);
	while (win.Open())
	{
		win.PollEvents();
		while (generator.Step(maze))
			;

		if (!generator.Step(maze))
		{
			if (!solver.Step(maze))
			{
				std::cout << "SOLVED!" << std::endl;
				maze = Maze(maze.Width() + 2, maze.Height() + 2);
				generator.Reset(maze, RandomCoord(rand, maze));

				start = RandomCoord(rand, maze);
				end = RandomCoord(rand, maze);
				while (end == start)
				{
					end = RandomCoord(rand, maze);
				}
				solver.NewMaze(start, end, maze.Width() * maze.Height() / 2);
			}
		}

		maze.Render(win, 0, 0, 800, 800);

		win.Display();
		win.Clear();

	}
	return 0;
}
```

Varje gång en lösning hittas skapas labyrinten om med en storlek större (två tiles), men populationen förblir densamma. DecisionEvolver skriver ut koden för den ”vinnande” botten till konsolen varje gång. Utskriftterna och ett antal skärmdumpar demonstrerar körexemplet. Även om funktionen kan köras i oändligheten kommer det här körexemplet avbrytas efter bara ett antal hittade lösningar. En observation är att väldigt ofta löser samma bot flera labyrinter i rad.

2.2.4.1 Skärmdumpar



I bilderna ovan är väggar svarta och korrider vita. Målet är rött och startpunkten är grön. Det blå punkterna representerar en eller flera bottars slutposition.

2.2.4.2 Konsolutskrifter

```
Good bot! (34)
        if (wall_right?)
            if (wall_ahead?)
                turn_right
            else
        else
            if (wall_right?)
```

```
                if (wall_right?)
                    if (wall_right?)
                        if (wall_right?)
                            turn_left
                        else
                            turn_left
                    else
                        turn_right
                else
                    if (wall_right?)
                        if (wall_right?)
                            turn_left
                        else
                            turn_left
                    else
                        turn_right
            else
                turn_right

SOLVED!
Good bot! (128)
        if (wall_right?)
            turn_left
        else
            if (wall_right?)
                if (wall_right?)
                    if (wall_right?)
                        turn_left
                    else
                        turn_left
                else
                    turn_right
            else
                if (wall_right?)
                    if (wall_right?)
                        turn_left
                    else
                        turn_left
                else
                    turn_right

SOLVED!
Good bot! (102)
        if (wall_right?)
            if (wall_ahead?)
                turn_left
            else
        else
            turn_right

SOLVED!
Good bot! (114)
        if (wall_right?)
            if (wall_ahead?)
                turn_left
            else
        else
            if (wall_right?)
                if (wall_right?)
                    if (wall_left?)
                    else
```

```
                        turn_left
                else
                    if (wall_left?)
                        turn_right
                    else
            else
                if (wall_right?)
                    if (wall_right?)
                        if (wall_left?)
                        else
                            turn_left
                    else
                        if (wall_left?)
                            turn_right
                        else
                else
                    if (wall_right?)
                        if (wall_left?)
                        else
                            turn_left
                    else
                        if (wall_left?)
                            turn_right
                        else

SOLVED!
Good bot! (102)
        if (wall_right?)
            if (wall_ahead?)
                turn_left
            else
        else
            turn_right

SOLVED!
Good bot! (120)
        if (wall_right?)
            if (wall_right?)
                if (wall_right?)
                    if (wall_right?)
                        if (wall_left?)
                        else
                            turn_left
                    else
                        if (wall_left?)
                            turn_right
                        else
                else
                    if (wall_right?)
                        if (wall_right?)
                            if (wall_left?)
                            else
                                turn_left
                        else
                            if (wall_left?)
                                turn_right
                            else
                    else
                        if (wall_right?)
                            if (wall_left?)
                            else
```

```
                                turn_left
                        else
                            if (wall_left?)
                                turn_right
                            else
            else
                if (wall_right?)
                    if (wall_left?)
                    else
                        turn_left
                else
                    if (wall_left?)
                        turn_right
                    else
        else
            turn_right

SOLVED!
Good bot! (69)
        if (wall_right?)
            if (wall_right?)
                turn_left
            else
                if (wall_right?)
                else
        else
            if (wall_right?)
            else
                turn_right

SOLVED!
Good bot! (281)
        turn_left
        turn_right
        if (wall_left?)
        else
            turn_left
        turn_right
        if (wall_ahead?)
        else
        turn_left
        turn_right
        if (wall_left?)
        else
            turn_left
        turn_right
        if (wall_ahead?)
        else
        turn_left

SOLVED!
Good bot! (186)
        if (wall_right?)
        else
            turn_right
        turn_left
        turn_left
        turn_right
        turn_right
        if (wall_ahead?)
        else
```

```
        turn_left
        if (wall_right?)
        else
            turn_right

SOLVED!
Good bot! (40)
        if (wall_ahead?)
            turn_right
        else
            if (wall_ahead?)
                turn_left
            else
        turn_right
        turn_left
        if (wall_ahead?)
            turn_left
        else
        turn_right
        turn_left
        if (wall_ahead?)
            turn_right
        else
        turn_right
        if (wall_right?)
        else
            turn_right
        turn_right
        if (wall_right?)
        else
            turn_right
        turn_left
        if (wall_ahead?)
            turn_right
        else
            if (wall_ahead?)
                turn_left
            else
        turn_left
        turn_right

SOLVED!
```

2.2.5 Analys

Här gäller samma sak som i 2.1.5 vad gäller att lösa labyrinter i allmänhet. Vad gäller den här specifika lösningen var ett mål att se om AI:n kunde lyckas nå en elegant lösning på labyrinter i allmänhet, vilket är skälet att samma population används i många olika labyrinter; för att se hur specifik koden för labyrinten. Det gick att se att flera labyrinter kunde lösa med få, och ibland inga, populationer emellan, vilket tyder på att koden (för åtminstone små labyrinter) var tillräckligt ospecifik för labyrinten de utvecklades i.

Ett annat mål var att se om en det kunde utvecklas en algoritm för labyrintlösning som på något sätt liknar den klassiska metoden för labyrintlösning som går ut på att alltid hålla en hand mot samma vägg. Via en sådan metod går det alltid att nå målet, åtminstone för den typen av labyrint som skulle lösas här. Det är osäkert om en sådan algoritm tillkom, men det är intressant att försöka gå igenom koden som tas fram och jämföra med en sådan metod.

Jämför med A* är den här metoden både väldigt långsam och inte det minsta praktisk. Däremot kan den säkert göras bättre genom att ge robotarna fler villkor. Andra villkor skulle kunna vara att kontrollera antalet steg till närmaste vägg framåt exempelvis. Det är svårt att se vilka extra kommandon som skulle kunna göras tillgängliga. I vilket fall som helst borde det vara svårt för den att, åtminstone generellt, snabbare hitta en väg än A*.

2.3 Konnektionistiska tekniker

2.3.1 Problem

Problemet är att låta en AI lära sig att klassifiera indata utan övervakning. Indatan består av ett antal färger, som representeras i 3D, och AI:n ska representera dessa i ett 2D-fält som består av pixlar.

Figur 9. Färgerna som indatan består av.

2.3.2 Design

För att lösa problemet används en sk Self-Organizing Map (SOM), som är ett typ av artificiellt neuralt nätverk. En SOM används för att visualisera data med högt antal dimensioner i lägre dimensioner. Det som gör att SOM står ut är att det kan göra detta utan övervakning.

2D-fältets pixlar initieras först med slumpmässiga färger, dessa färger bilder nätverket och varje pixel är en nod i nätverket. En nod har en vikt som representeras som en vektor i tre dimensioner (röd, grön och blå) och en position. Träningen av nätverket pågår i iterationer. En iteration fungerar så att en färgerna för av indatan väljs ut, och den nod vars data är mest lik den (där avstånd är avståndet mellan färgernas vektor i 3D-rummet) väljs ut.

Den utvalda nodens grannskap betraktas också. Grannskapet är de noder som är närmast den utvalda nodens position. Vilka noder som ingår i den utvalda nodens grannskap minskar med antalet iterationer. Radien för grannskapet beräknas med:

$$\sigma_t = \sigma_0 e^{\frac{-t}{\lambda}}$$

Formel 1

Där λ är en tidskonstant som beror på hur många iterationer som ska utföras samt 2D-fältets storlek och t är den nuvarande iteration. σ_0 sätt till hälften av 2D-fältets storlek. Alla noder som faller innanför denna radie kommer alltså ingå i den utvalda nodens grannskap för den nuvarande iterationen t.

Vikten (W) för dessa noder, alltså deras färger, kommer anpassas till den valda indatas vektor (V). Hur mycket beror på inlärningshastigheten (L) som även den minskar med antalet iterationer. Vikten anpassas enligt följande:

$$W_{t+1} = W_t + \theta_t L_t (V_t - W_t)$$

Formel 2

Där θ_t defineras som:

$$\theta_t = e^{-\frac{dist^2}{2\sigma_t{}^2}}$$

Formel 3

och dist är avståndet mellan noden som ska anpassas och den utvalda noden.

2.3.3 Kod

2.3.3.1 Controller.h

```
#pragma once

#include "Window.h"
#include <vector>
```

```
#include "SOM.h"
#include "Coord.h"
#include "Random.h"


class Controller
{
public:
        Controller(int map_width, int map_height, const std::vector<Color>& training_set_);

        void train();
        bool is_done();
        void render(Window& window);
private:
        void create_data_set(const std::vector<Color>& training_set_);

        /* first is the color, second is a pointer to the closest node */
        std::vector<Color> training_set_;
        SOM::node_t* find_closest_match(Color& color);

        SOM som_;
        int w_, h_;
        double map_radius_;
        int iteration_count_, max_iterations_;
        double base_learning_rate_;

        Randomizer rand_;
};
```

2.3.3.2 Node.h

```
#pragma once

#include <array>
#include <unordered_set>

template <class position_t, class weight_t, size_t weights_dimensionality_>
class Node
{
public:
        typedef std::unordered_set<Node<position_t, weight_t, weights_dimensionality_>*> set_t
;


        Node(){}

        template <class InputIterator>
        Node(InputIterator iter_begin, InputIterator iter_end);

        inline const int dimensionality() const { return weights_dimensionality_; }
        inline weight_t& operator[](size_t index) { return data_[index]; };
        inline const weight_t& operator[](size_t index) const { return data_[index]; };

        inline position_t& x() { return position_[0]; }
        inline const position_t x() const { return position_[0]; }

        inline position_t& y() { return position_[1]; }
        inline const position_t y() const { return position_[1]; }

private:
        std::array<weight_t, weights_dimensionality_> data_;
```

```
        std::array<position_t, 2U> position_;
};


template <class position_t, class weight_t, size_t weights_dimensionality_>
template <class InputIterator>
Node<position_t, weight_t, weights_dimensionality_>::Node(InputIterator iter_begin, InputItera
tor iter_end)
{
        set_t::iterator it = neighbours_.begin();
        for (size_t i = 0; i < weights_dimensionality_; ++i)
        {
                vals_[i] = *iter_begin;
                iter_begin++;
        }
}
```

2.3.3.3 SOM.h

```
#pragma once

#include "Node.h"
#include <vector>
#include "Color.h"

class SOM
{
public:
        typedef Node<double, double, 3U> node_t;
        SOM(size_t w, size_t h);

        inline node_t operator()(int x, int y) { ensure_range(x, y); nodes_[y * static_cast<in
t>(w_) + x]; }
        inline size_t width() const { return w_; }
        inline size_t height() const{ return h_; }

        inline std::vector<node_t>::iterator begin(){ return nodes_.begin(); }
        inline std::vector<node_t>::iterator end(){ return nodes_.end(); }
        const inline std::vector<node_t>::const_iterator begin() const { return nodes_.begin()
; }
        const inline std::vector<node_t>::const_iterator end() const { return nodes_.end(); }
        const inline std::vector<node_t>::const_iterator cbegin() const { return nodes_.cbegin
(); }
        const inline std::vector<node_t>::const_iterator cend() const { return nodes_.cend();
}

        double euclidean_weight_distance(const node_t& n0, const node_t& n1) const;
        double euclidean_position_distance(const node_t& n0, const node_t& n1) const;
        double euclidean_color_distance(const node_t& n, const Color& c) const;
        Color to_color(const node_t& n) const;
private:
        inline bool ensure_range(int x, int y) const { return x >= 0 && x < w_ && y >= 0 && y
< h_; };

        size_t w_, h_;
        std::vector<node_t> nodes_;
};
```

2.3.3.4 Controller.cpp

```
#include "Controller.h"
#include "Random.h"
```

```
Controller::Controller(int map_width, int map_height, const std::vector<Color>& training_set)
        :w_(map_width)
        ,h_(map_height)
        ,som_(map_width, map_height)
        ,map_radius_(std::max(map_width, map_height) / 2)
        ,iteration_count_(0)
        ,base_learning_rate_(0.1f)
        ,max_iterations_(1000)
{
        create_data_set(training_set);
}

void Controller::train()
{
        if (is_done())
        {
                return;
        }

        ++iteration_count_;

        double time_constant = static_cast<double>(max_iterations_) / std::log(map_radius_);
        double neighbour_hood_radius = map_radius_ * exp(-
static_cast<double>(iteration_count_ / time_constant));
        double learning_rate = base_learning_rate_ * exp(-
static_cast<double>(iteration_count_/ max_iterations_));

        auto training_p= training_set_[rand_.NextInt(training_set_.size() - 1)];
        auto closest = find_closest_match(training_p);

        std::vector<SOM::node_t*> to_remove;
        for (auto& n : som_)
        {
                if (som_.euclidean_position_distance(n, *closest) < neighbour_hood_radius)
                {
                        Color& c = training_p;

        std::array<double, 3> v{ { c.get_red(), c.get_green(), c.get_blue() } };

                        double distance = som_.euclidean_position_distance(n, *closest);
                        double theta = std::exp(-(distance / neighbour_hood_radius));
                        for (int i = 0; i < 3; ++i)
                        {
                                n[i] = n[i] + theta * learning_rate * (v[i] - n[i]);
                        }
                }
        }

        Color& c = training_p;
        std::array<double, 3> v{ { c.get_red(), c.get_green(), c.get_blue() } };
        SOM::node_t& node = *closest;

        double distance = som_.euclidean_position_distance(node, *closest);
        double theta = std::exp(-(distance / neighbour_hood_radius));
        for (int i = 0; i < 3; ++i)
        {
                node[i] = node[i] + theta * learning_rate * (v[i] - node[i]);
        }

}
```

```
SOM::node_t* Controller::find_closest_match(Color& color)
{
	SOM::node_t* closest = nullptr;
	for (auto& n : som_)
	{
		for (auto& p : training_set_)
		{
			if (closest == nullptr)
			{
				closest = &n;
			}

	else if (som_.euclidean_color_distance(n, color) < som_.euclidean_color_distance(*clos
est, color))
			{
				closest = &n;
			}
		}
	}
	return closest;
}

bool Controller::is_done()
{
	return (iteration_count_ > max_iterations_);
}

void Controller::render(Window& window)
{
	double n_w = static_cast<double>(window.Width()) / static_cast<double>(w_) ;
	double n_h = static_cast<double>(window.Heigth()) / static_cast<double>(w_);

	for (auto& n : som_)
	{
		auto color = Color::make_from_floats(n[0], n[1], n[2]).clamp();
		window.RenderRectangle(n.x() * n_w, n.y() * n_h, n_w, n_h, color);
	}
}

void Controller::create_data_set(const std::vector<Color>& training_set)
{
	double x = 0.f;
	double y = 0.f;
	for (Color c : training_set)
	{
		training_set_.emplace_back(c);
	}

	for (auto& n : som_)
	{
		n.x() = x;
		n.y() = y;

		for (int i = 0; i < n.dimensionality(); ++i)
		{
			n[i] = rand_.NextDouble(0.f, 1.f);
		}

		x += 1.f;
		if (x >= static_cast<double>(w_))
```

```
                {
                        x = 0.f;
                        y += 1.f;
                }
        }
}
```

2.3.3.5 SOM.cpp

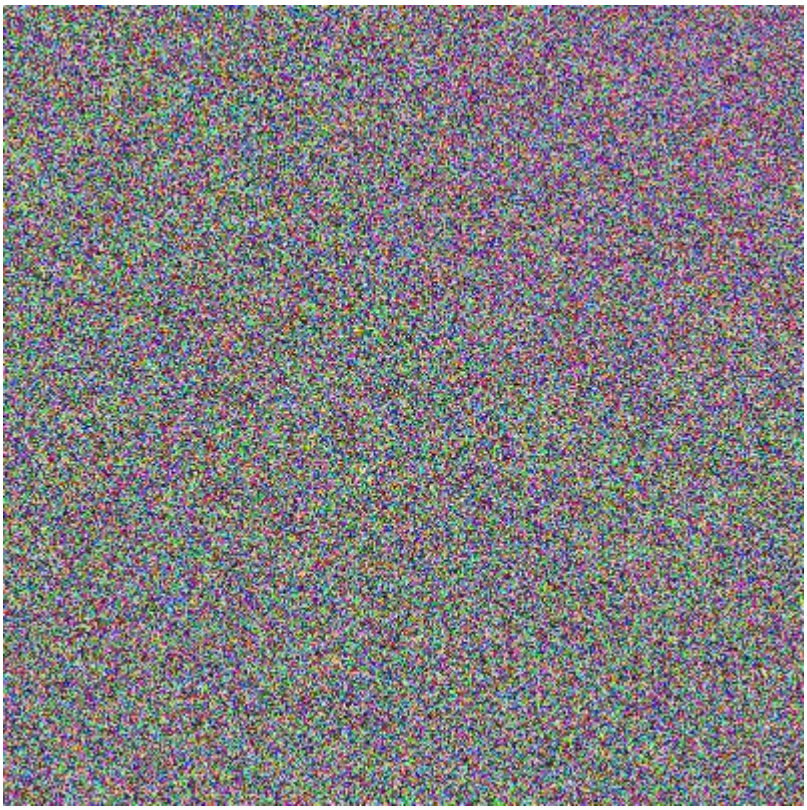
```
#include "SOM.h"

typedef Node<int, double, 3U> node_t;

SOM::SOM(size_t w, size_t h)
        :w_(w)
        ,h_(h)
        ,nodes_(w *h)
{

}

double SOM::euclidean_weight_distance(const node_t& n0, const node_t& n1) const
{
        double sum = 0.f;
        for (int i = 0; i < n0.dimensionality(); ++i)
        {
                sum += std::pow((n0[i] - n1[i]), 2.0f);
        }
        return std::sqrt(sum);
}

double SOM::euclidean_position_distance(const node_t& n0, const node_t& n1) const
{
        double sum = std::pow(n0.x() - n1.x(), 2.0f) + std::pow(n0.y() - n1.y(), 2.0f);
        return std::sqrt(sum);
}

double SOM::euclidean_color_distance(const node_t& n, const Color& c) const
{
        double sum = std::pow(n[0] - c.get_red(), 2.0f);
        sum += std::pow(n[1] - c.get_green(), 2.0f);
        sum += std::pow(n[2] - c.get_blue(), 2.0f);
        return std::sqrt(sum);
}

Color SOM::to_color(const node_t& n) const
{
        return Color::make_from_floats(n[0], n[1], n[2]);
}
```

2.3.3.6 main.cpp

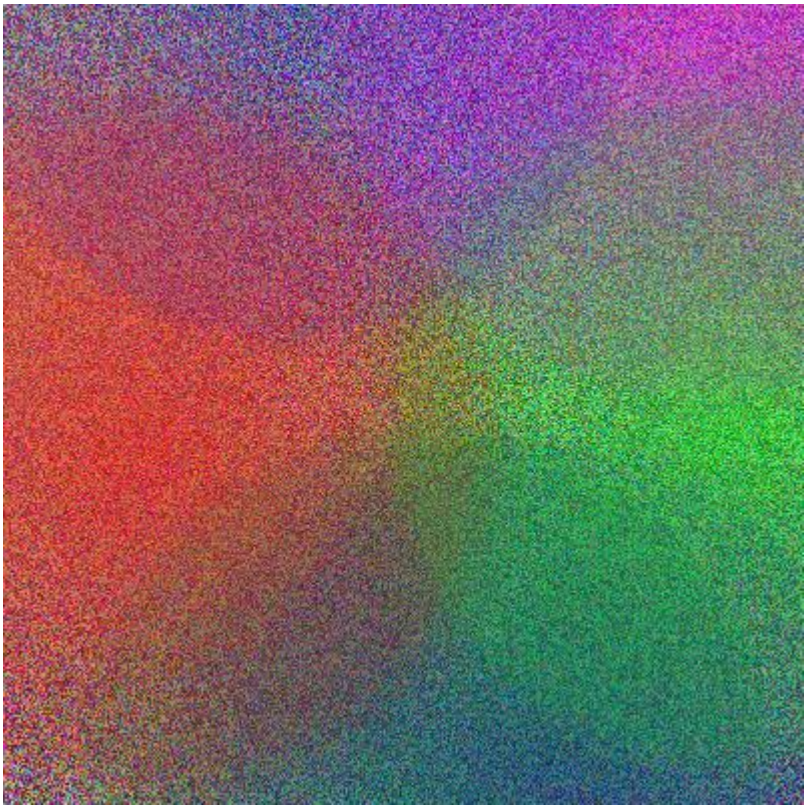
```
#include <SDL.h>
#include "Window.h"
#include "Controller.h"

int main(int argc, char *argv[])
{
        //create a data set
        Color red, green, blue, yellow, orange, purple, dark_green, dark_blue;

        red.set_red(1);
```

```
        red.set_green(0);
        red.set_blue(0);

        green.set_red(0);
        green.set_green(1);
        green.set_blue(0);

        dark_green.set_red(0);
        dark_green.set_green(0.5);
        dark_green.set_blue(0.25);

        blue.set_red(0);
        blue.set_green(0);
        blue.set_blue(1);

        dark_blue.set_red(0);
        dark_blue.set_green(0);
        dark_blue.set_blue(0.5);

        yellow.set_red(1);
        yellow.set_green(1);
        yellow.set_blue(0.2);

        orange.set_red(1);
        orange.set_green(0.4);
        orange.set_blue(0.25);

        purple.set_red(1);
        purple.set_green(0);
        purple.set_blue(1);

        Window win("Self Organizing Map", 400, 400);
        Controller c(40, 40, { red, green, blue, yellow, orange, purple, dark_green, dark_blue
 });

        while (win.Open())
        {
                win.PollEvents();
                c.train();
                c.render(win);
                win.Display();
                win.Clear();
        }
        return 0;
}
```

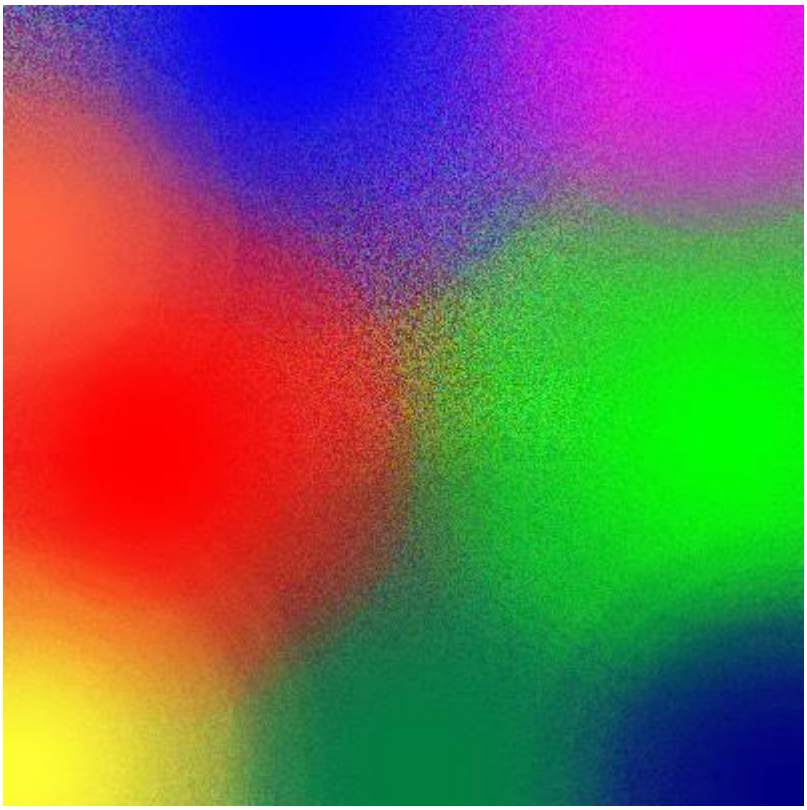
2.3.4 Körexempel



Figur 10. Startfältet.



Figur 11. Efter 50 iterationer börjar färgerna grupperas.



Figur 12. Efter 500 iterationer syns det hur nätvärker har valt att gruppera färgerna.

2.3.5 Analys

En kan tänka sig flera olika typer av vikter som kan lösas med liknande design. Exempelvis hade en möjligthet varit att låta vikterna representera en hel bild (vilket såklart hade medfört hög dimensionalitet). På detta sätta hade det gått att gruppera bilder som är lika varandra på motsvarande sätt som problemexemplet som tagits upp här. Det finns säkert många fler exempel på vad en Self-Organising Map hade kunnat tränas att kategorisera eller representera.

2.4 Skripttekniker

2.4.1 Problem

I ett begränsat spelfält, låt robotar spela ett enkelt tafatt-spel. Reglerna är att två robotar som rör vid varandra skadar varandra, men robotarna kan ha olika max-hälsa. Olika robotar ska kunna ha olika beteenden och det ska gå enkelt att lägga till och ändra beteenden utan att kompilera om programmet.

2.4.2 Design

För att låta robotornas beteende ändras utan att kompilera programmet används skriptning, specifikt används lua som skriptspråk. För att göra det enkelt att använda lua används luaccpinterface, ett open-source projekt (https://github.com/davidsiaw/luacppinterface). Luaccpinterface gör det enkelt att kalla funktioner i lua-miljön och vice versa genom att exempelvis sköta castingen av data typer. För att göra det ännu enklare wrappas detta runt en klass LuaHelper.

Ett gränssnitt som kan användas från lua görs tillgängligt:

- void run_script(std::string file);

Tar emot en fil med lua-script och exekverar den i lua-miljön.

- LuaTable register_robot(std::string name, LuaFunction<void(LuaTable)> update_function);

Registrerar en ny robot, och kopplar ett namn och en uppdatering-funktion till den. Värd-språket skapar en tabell i lua-miljön med variabler som kan sättas och läsas från lua och i sin tur läsas av värd-språket för att rita ut roboten och ta bort den om den får slut på hälsa. Efter att roboten är registerar kan färgen och maxhälsan sättas. Efter det kommer uppdateringsfunktionen kallas för varje frame från värdspråket och tabellen som representerar robotens värden (en analog till this-objektet) skickas som parameter till lua, där beteendet kan styras.

- void print(std::string str) const

Skriver ut en sträng till konsolfönstret för debug-syfte.

- int manhattan_distance(LuaTable robot0, LuaTable robot1) const

Räknar ut manhattan-avståndet mellan två robotar och returnerar det.

- int random(int min, int max)

Returnerar ett slumpvärde mellan min och max, inklusivt.

För att robotar ska komma åt andra robotars värde läggs de i en array i lua-miljön (”robots”), storleken på den arrayen läggs också in i lua-miljön (”robot_count”). Storleken på banan, höjd och vidd, läggs in i en tabell i lua-miljön (”level.width”, ”level.height”).

För att initiera lua-skripten kallas en filen ”robots.lua”, som kan innehålla godtycklig kod.

2.4.3 Kod

2.4.3.1 LuaHelper.h

```
#pragma once

#include <luacppinterface.h>
#include <string>
#include "Robot.h"
#include <map>
#include <vector>
#include <random>
```

```
class LuaHelper
{
public:
        LuaHelper(int level_w, int level_h);
        void init_standard_functions();
        void init_standard_values();

        void run_script(std::string file);

        int get_robot_count() const;
        Robot get_robot_at(int index) const;
        Robot get_robot(const std::string& name) const;


        template <typename func_t, typename... Ts>
        auto call(std::string func_name, Ts... ts) -
> typename LuaGenericFunction<typename func_t>::rettype
        {

        auto lua_func = lua_.GetGlobalEnvironment().Get<LuaFunction<func_t>>(func_name);
                auto result = lua_func.Invoke(ts ...);
                return result;
        }

        Lua get_lua() const { return lua_; }

        void update_robots();

private:
        LuaTable get_table_at(int index) const;
        LuaTable get_table(const std::string& name) const;
        LuaTable register_robot(std::string name, LuaFunction<void(LuaTable)> update_function)
;
        int manhattan_distance(LuaTable robot0, LuaTable robot1) const;
        int random(int min, int max);
        void print(std::string) const;

        Lua lua_;
        std::vector<std::pair<LuaTable, LuaFunction<void(LuaTable)>>> robots_;
        int level_w_, level_h_;

        std::mt19937 engine_;
};
```

2.4.3.2 Robot.h

```
#pragma once

struct Robot
{
        int x, y, health;
        float r, g, b;
        std::string name;
};
```

2.4.3.3 LuaHelper.cpp

```
#include "LuaHelper.h"
#include <fstream>
#include <iostream>
#include <algorithm>
```

```cpp
LuaHelper::LuaHelper(int level_w, int level_h)
	:level_w_(level_w)
	,level_h_(level_h)
{
	init_standard_functions();
	init_standard_values();
	std::random_device random;
	engine_ = std::mt19937(random());
}

void LuaHelper::run_script(std::string file)
{
	std::ifstream t(file);
	std::string what = lua_.RunScript({ std::istreambuf_iterator<char>(t), std::istreambuf_iterator<char>() });
	std::cout << "Loading: " << file << std::endl;
	std::cout << what << std::endl;
	std::cout << std::endl;
}

LuaTable LuaHelper::register_robot(std::string name, LuaFunction<void(LuaTable)> update_function)
{
	LuaTable table = lua_.CreateTable();
	table.Set("name", name);
	table.Set("x", random(0, level_w_ - 1));
	table.Set("y", random(0, level_h_ - 1));
	table.Set("r", 0.0f);
	table.Set("g", 0.0f);
	table.Set("b", 0.0f);
	table.Set("health", 100);

	robots_.push_back(std::make_pair(table, update_function));
	lua_.GetGlobalEnvironment().Set("robot_count", robots_.size());
	auto robots = lua_.GetGlobalEnvironment().Get<LuaTable>("robots");
	robots.Set(robots_.size() - 1, table);
	return table;
}

void LuaHelper::init_standard_functions()
{
	lua_.GetGlobalEnvironment().Set("run_script", lua_.CreateFunction<void(std::string)>([this](std::string file){ run_script(file); }));
	lua_.GetGlobalEnvironment().Set("register_robot", lua_.CreateFunction<LuaTable(std::string, LuaFunction<void(LuaTable)>)>([this](std::string file, LuaFunction<void(LuaTable)> update_func){ return register_robot(file, update_func); }));
	lua_.GetGlobalEnvironment().Set("manhattan_distance", lua_.CreateFunction<int(LuaTable, LuaTable)>([this](LuaTable r0, LuaTable r1){ return manhattan_distance(r0, r1); }));
	lua_.GetGlobalEnvironment().Set("random", lua_.CreateFunction<int(int, int)>([this](int min, int max){ return random(min, max); }));
	lua_.GetGlobalEnvironment().Set("print", lua_.CreateFunction<void(std::string)>([this](std::string str){ print(std::move(str)); }));
}

void LuaHelper::print(std::string str) const
{
	std::cout << std::move(str) << std::endl;
}

void LuaHelper::init_standard_values()
{
```

```
        lua_.GetGlobalEnvironment().Set("robot_count", 0);
        lua_.GetGlobalEnvironment().Set("robots", lua_.CreateTable());

        LuaTable map = lua_.CreateTable();
        lua_.GetGlobalEnvironment().Set("level", map);
        map.Set("width", level_w_);
        map.Set("height", level_w_);

}

void LuaHelper::update_robots()
{
        for (auto p : robots_)
        {
                p.second.Invoke(p.first);
        }

        for (int i = 0; i < robots_.size(); ++i)
        {
                for (int j = 0; j < robots_.size(); ++j)
                {
                        if (j != i)
                        {
                                int x0, y0, x1, y1;
                                x0 = robots_[i].first.Get<int>("x");
                                y0 = robots_[i].first.Get<int>("y");
                                x1 = robots_[j].first.Get<int>("x");
                                y1 = robots_[j].first.Get<int>("y");
                                if (x0 == x1 && y0 == y1)
                                {

        robots_[i].first.Set("health", robots_[i].first.Get<int>("health") - 10);

        robots_[j].first.Set("health", robots_[j].first.Get<int>("health") - 10);
                                }
                        }
                }
        }

        auto robots = lua_.GetGlobalEnvironment().Get<LuaTable>("robots");

        robots_.erase(std::remove_if(std::begin(robots_), std::end(robots_), [](const std::pai
r < LuaTable, LuaFunction<void(LuaTable)>>& p) { return p.first.Get<int>("health") <= 0; }), s
td::end(robots_));

        for (int i = 0; i < robots_.size(); ++i)
        {
                robots.Set(i, robots_[i].first);
        }
}

int LuaHelper::get_robot_count() const
{
        return robots_.size();
}

Robot LuaHelper::get_robot_at(int index) const
{
        Robot ret;
        LuaTable table = robots_[index].first;
        ret.x = table.Get<int>("x");
```

```
        ret.y = table.Get<int>("y");
        ret.r = table.Get<float>("r");
        ret.g = table.Get<float>("g");
        ret.b = table.Get<float>("b");
        ret.health = table.Get<int>("health");
        ret.name = table.Get<std::string>("name");
        return ret;
}
        //robots.Set(name, table);

LuaTable LuaHelper::get_table_at(int index) const
{
        return robots_[index].first;
}

Robot LuaHelper::get_robot(const std::string& name) const
{
        for (int i = 0; i < robots_.size(); ++i)
        {
                if (robots_[i].first.Get<std::string>("name") == name)
                {
                        return get_robot_at(i);
                }
        }
        throw std::out_of_range("No such robot");
}

LuaTable LuaHelper::get_table(const std::string& name) const
{
        for (int i = 0; i < robots_.size(); ++i)
        {
                if (robots_[i].first.Get<std::string>("name") == name)
                {
                        return robots_[i].first;
                }
        }
        throw std::out_of_range("No such robot");
}

int LuaHelper::manhattan_distance(LuaTable robot0, LuaTable robot1) const
{
        int x0, y0, x1, y1;
        x0 = robot0.Get<int>("x");
        y0 = robot0.Get<int>("y");
        x1 = robot1.Get<int>("x");
        y1 = robot1.Get<int>("y");
        int ret = std::abs(x0 - x1) + std::abs(y0 - y1);
        return ret;
}

int LuaHelper::random(int min, int max)
{
        std::uniform_int_distribution<int> dist(min, max);
        return dist(engine_);
}
```

2.4.3.4 main.cpp

```
#include <lua.hpp>
#include <luacppinterface.h>
#include <string>
#include <fstream>
```

```
#include <iostream>
#include "LuaHelper.h"
#include "Window.h"
#include "Timer.h"

void render_robots(Window& window, const LuaHelper& helper)
{
        for (int i = 0; i < helper.get_robot_count(); ++i)
        {
                Robot robot = helper.get_robot_at(i);

        window.RenderRectangle(robot.x * 8, robot.y * 8, 8, 8, Color::make_from_floats(robot.r
, robot.g, robot.b));
        }
}

int main(int argc, char **argv)
{
        LuaHelper helper(800 / 8, 800 / 8);
        Window window("robot wars", 800, 800);
        helper.run_script("robots.lua");

        Timer timer;
        timer.Start();

        while (window.Open())
        {
                window.PollEvents();
                if (timer.ElapsedMilliseconds() > 1000 / 10)
                {
                        timer.Start();
                        helper.update_robots();
                }
                render_robots(window, helper);
                window.Display();
                window.Clear();
        }

        return 0;
}
```

2.4.4 Körexempel

För körexemplet används följande filer:

```
robot_names = {}
robot_names[0] = "red_robot.lua"
robot_names[1] = "blue_robot.lua"
robot_names[2] = "green_robot.lua"

for i = 1, 100 do
        run_script(robot_names[random(0, 2)])
end
```

Figure 1. robots.lua

```
-- move randomly

new_robot = register_robot("blue_robot"..robot_count,
        function(this)
```

```
                this.x = this.x + random(-1, 1);
                this.y = this.y + random(-1, 1);

                if this.x < 0 then this.x = 0 end
                if this.y < 0 then this.y = 0 end
                if this.x >= level.width then this.x = level.width - 1 end
                if this.y >= level.height then this.y = level.height - 1 end
        end
);

new_robot.b = 1
new_robot.health = 50
```

Figure 2. blue_bot.lua

```
-- Always chase the closest robot

new_robot = register_robot("green_robot"..robot_count,
        function(this)
                robot = nil
                closest_robot = nil
                for i = 1, robot_count do
                        robot = robots[i - 1]
                        if robot.name ~= this.name then
                                if closest_robot == nil then
                                        closest_robot = robot

        elseif manhattan_distance(this, closest_robot) > manhattan_distance(this, robot) then
                                        closest_robot = robot
                                end
                        end
                end

                if closest_robot ~= nil then
                        d_x = closest_robot.x - this.x
                        d_y = closest_robot.y - this.y

                        if d_x < 0 then
                                this.x = this.x - 1
                        end
                        if d_x > 0 then
                                this.x = this.x + 1
                        end
                        if d_y < 0 then
                                this.y = this.y - 1
                        end
                        if d_y > 0 then
                                this.y = this.y + 1
                        end
                        if d_y == 0 and d_x == 0 then
                                this.x = this.x + random(-1, 1)
                                this.y = this.y + random(-1, 1)
                        end
                end

                if this.x < 0 then this.x = 0 end
                if this.y < 0 then this.y = 0 end
                if this.x >= level.width then this.x = level.width - 1 end
                if this.y >= level.height then this.y = level.height - 1 end
        end
);
```

```
new_robot.g = 1
new_robot.health = 200
```

Figure 3. green_bot.lua

```
-- run away from all other robots

new_robot = register_robot("red_robot"..robot_count,
        function(this)
                d_x = 0
                d_y = 0
                for i = 1, robot_count do
                        robot = robots[i - 1]

        if robot.name ~= this.name and manhattan_distance(this, robot) < 20 then
                                d_x = d_x + this.x - robot.x
                                d_y = d_y + this.y - robot.y
                        end
                end

                if d_x < 0 then
                        this.x = this.x - 1
                end
                if d_x > 0 then
                        this.x = this.x + 1
                end
                if d_y < 0 then
                        this.y = this.y - 1
                end
                if d_y > 0 then
                        this.y = this.y + 1
                end

                if this.x < 0 then this.x = 0 end
                if this.y < 0 then this.y = 0 end
                if this.x >= level.width then this.x = level.width - 1 end
                if this.y >= level.height then this.y = level.height - 1 end
        end
);

new_robot.r = 1
new_robot.health = 10
```

Figure 4. red_robot.lua

De olika robarna fungerar alltså följande:

- Den röda roboten försöker undvika alla andra robotar.
- Den blå roboten rör sig helt slumpmässigt.
- Den gröna roboten försöker alltid jaga den närmaste roboten.

Deras maxhälsa är helt godtycklig.

Figure 5 och Figure 6 visar skärmdumpar från ett körexempel med dessa skript.

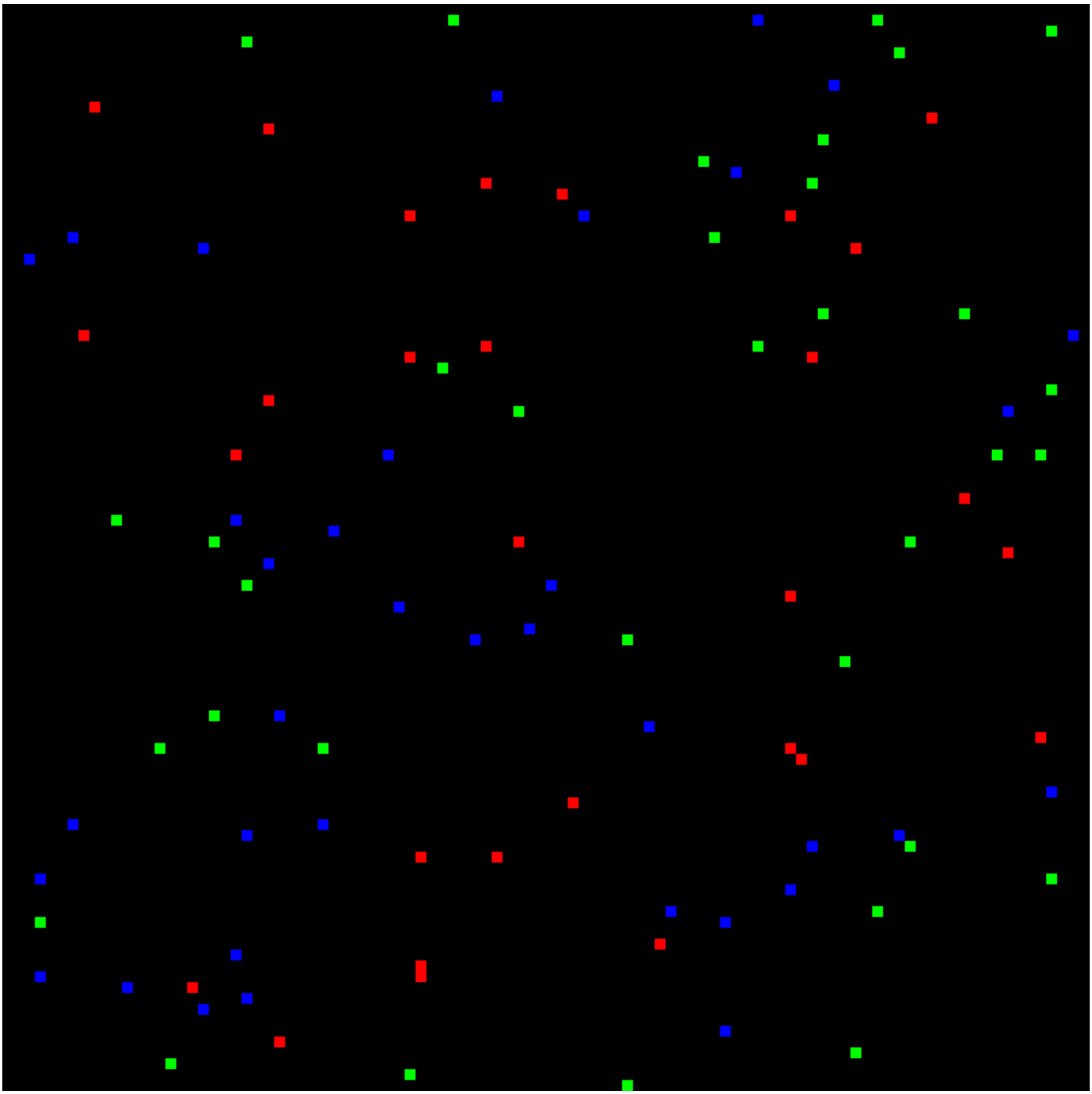


Figure 5. En skärmdump från första framen av ett körexempel.

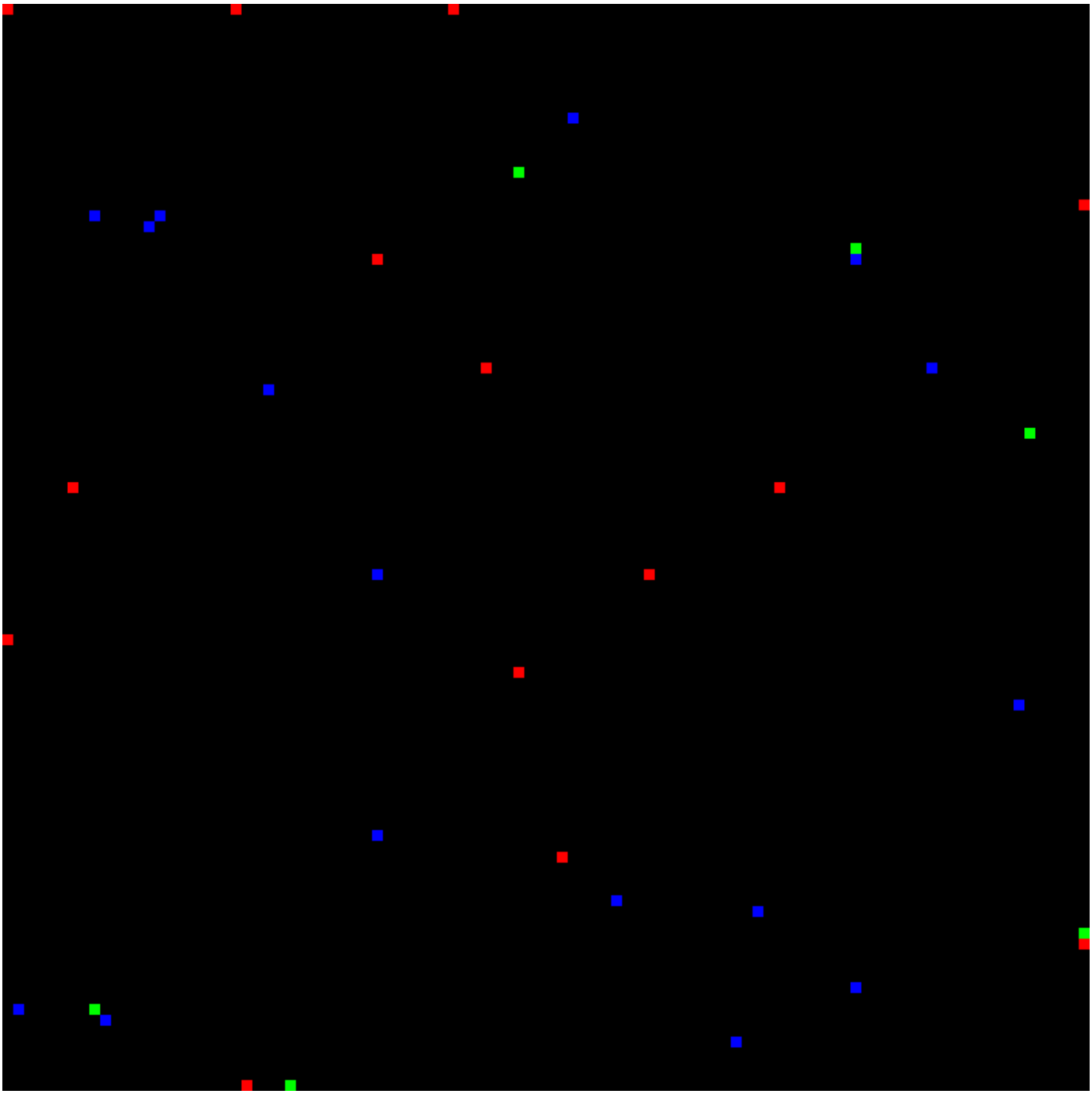


Figure 6. Samma körexempel, 1000 frames senare.

2.4.5 Analys

Programmet låter programmeraren testa många olika typer av robotar utan att kompilera om programmet, vilket gör det enkelt att iterera fram olika beteended utan att vara beroende av vare sig kompileringstider eller ens källkoden. En observation är också att det finns väldigt lite källkod eftersom att mycket att koden finns i skript. Den enda värdkoden som behövs är kopplingen mellan värdspråk och skriptspråk samt utritning.

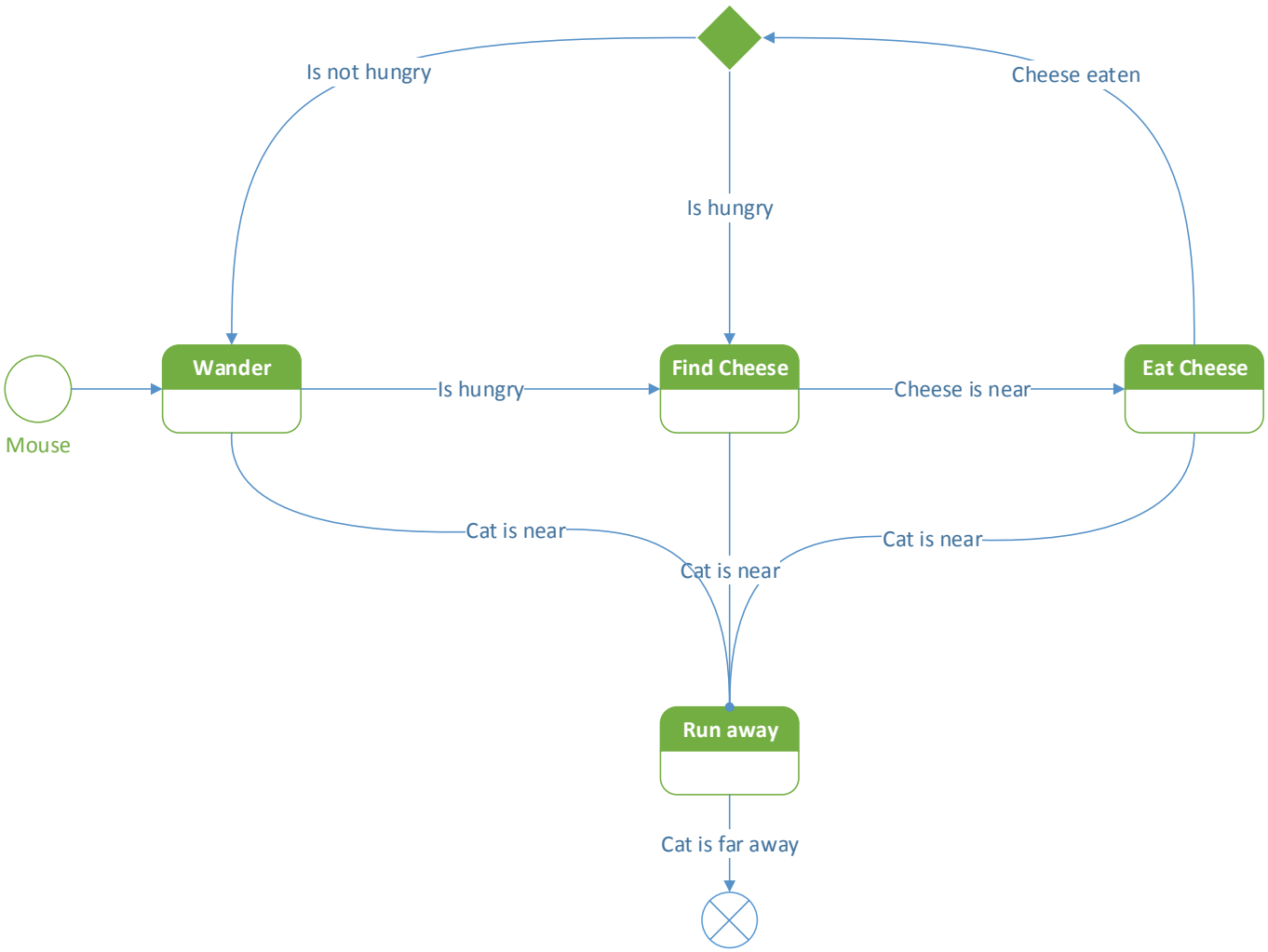
2.5 Tillståndstekniker

2.5.1 Problem

Problemet som ska lösas är att en mus i ett begränsat spelfält dels ska leta efter och äta ostar (som är statiska object i spelfältet) om den är hungrig men också undvika katter, som går runt i spelfältet slumpmässigt.

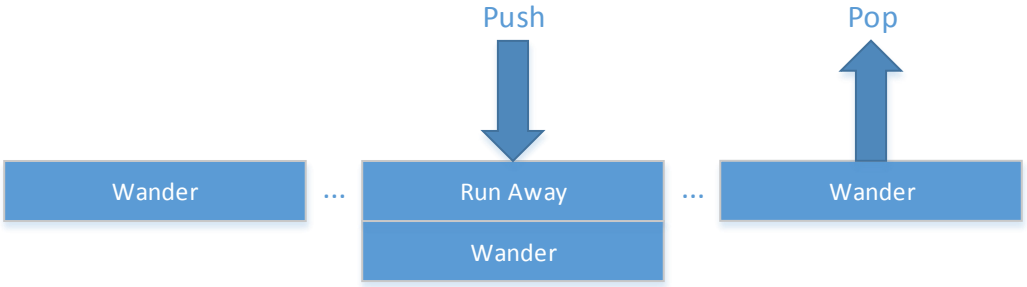
2.5.2 Design

För att lösa problemet används en stackbaserad tillståndsmaskin. Tillståndsmaskinen för att musen kan befinna sig i olika tillstånd baserat på vad som händer på spelfältet, men eftersom att musen ska kunna sättas i flyende tillstånd oavsett vilket annat tillstånd den befinner sig i och samtidigt kunna gå tillbaka till samma tillstånd igen behöver tillståndsmaskinen vara stackbaserad. Figuren nedan illustrerar denna tillståndsmaskin.



Figur 13. Diagram för musens tillståndståndsmaskin.

Tillstånden är en funktionspekare. Dessa funktioner kan manipulera stacken genom att antingen poppa ett state, eller pusha ett nytt state. För att utföra en tillståndsövergång kan funktionen poppa först och sen pusha ett state. För att, enligt figuren ovan, gå till ”Run Away”-tillståndet pushar det nuvarande tillståndet bara denna funktionen på stacken. När då katten är tillräckligt långt bort igen poppar bara detta tillståndet sig själv. Endast det översta tillståndet i stacken kallas och kallas då vid uppdatering. Funktionen kan utöver att manipulera stacken även manipulera musens position på kartan och kontrollera andra objekts position på kartan.



Figur 14. Exempel på hur stacken används för att musen ska fly. Det översta tillståndet är den funktionen som kallas.

För att visualisera händelserna i spelfältet representeras möss som blåa, katter som röda och ostar som gula.

2.5.3 Kod

2.5.3.1 Cat.h

```
#pragma once

#include "Rect.h"
#include "StateMachine.h"
#include "World.h"
#include <random>
#include "Wanderer.h"

class Cat
{
public:
        Cat(World& world, float x, float y);
private:
        void update(float dt);

        void wander(World&);

        StateMachine state_machine_;

        Rect rect_;
        float delta_time_;
        float hunger_meter_;

        Wanderer wanderer_;

        std::mt19937 random_engine_;

        int id_;
};
```

2.5.3.2 Mouse.h

```
#pragma once

#include "Rect.h"
#include "StateMachine.h"
#include "World.h"
#include <random>
#include "Wanderer.h"
#include "Timer.h"

class Mouse
{
public:
```

```
        Mouse(World& world, float x, float y);
private:
        void update(float dt);

        void wander(World&);
        void find_cheese(World&);
        void eat_cheese(World&, int cheese_id, int start_time);
        void run_away(World&, std::function<const Rect&()> rect);
        void detect_cat(World&);

        StateMachine state_machine_;

        Rect rect_;
        float delta_time_;
        float hunger_meter_;

        Wanderer wanderer_;

        std::mt19937 random_engine_;
        Timer timer_;

        int id_;
};
```

2.5.3.3 StateMachine.h

```
#pragma once

#include <vector>
#include <functional>

class StateMachine
{
public:
        void pop_state();
        void push_state(std::function<void ()>);
        void execute_current();
private:
        std::vector<std::function<void()>> state_stack_;
};
```

2.5.3.4 Wanderer.h

```
#pragma once

#include <utility>
#include "Rect.h"
#include "World.h"
#include <random>

class Wanderer
{
public:
        Wanderer(World& world);
        void wander(Rect& rect, float dt, float speed);
        void reset();
        void force_target(const Rect& rect, float x, float y);
        void force_reverse_target(const Rect& rect, float x, float y);
private:
        float target_x_, target_y_;
        float start_x_, start_y_;
        float wander_len_;
```

```
        float target_wander_len_;

        bool has_target_;
        std::function<std::pair<float, float>()> random_coord_;

        std::mt19937 random_;
        std::uniform_real_distribution<float> dist_x_;
        std::uniform_real_distribution<float> dist_y_;
};
```

2.5.3.5 World.h

```
#pragma once

#include <functional>
#include <vector>
#include <map>

#include "Coord.h"
#include "Window.h"
#include "Rect.h"

class World
{
public:
        World(float w, float h);
        int add_object(std::string tag, std::function<void(float)> update, std::function<const
 Rect&()> rect);
        void update(float dt);
        void remove_object(int id);
        void render(Window& window);

        std::vector<std::pair<int, std::function<const Rect&()>>> find_with_tag(const std::str
ing& tag) const;

        float width() const
        {
                return w_;
        }

        float height() const
        {
                return h_;
        }
private:
        struct Object
        {
                std::string tag;
                std::function<void(float)> update;
                std::function<const Rect&()> rect;
        };

        std::map<int, Object> objects_;

        std::vector<int> to_remove_;
        int next_id_;

        void cull_dead();
        float w_, h_;
};
```

2.5.3.6 Cat.cpp

```
#include "Cat.h"


Cat::Cat(World& world, float x, float y)
        :rect_(x, y, 10.f, Color::red())
        , wanderer_(world)
{
        std::random_device rd;
        random_engine_.seed(rd());
        hunger_meter_ = 10;
        id_ = world.add_object("cat", [this](float dt){ update(dt); }, [this]() -
> const Rect&{ return rect_; });
        state_machine_.push_state
        (
                [this, &world](){ wander(world);
        });
}

void Cat::update(float dt)
{
        delta_time_ = dt;
        state_machine_.execute_current();
}

void Cat::wander(World& world)
{
        std::uniform_real_distribution<float> dist(-1.f, 1.f);
        float x = dist(random_engine_);
        float y = dist(random_engine_);
        wanderer_.wander(rect_, delta_time_, 50.f);
}
```

2.5.3.7 main.cpp

```
#include "Window.h"
#include "Timer.h"
#include "Mouse.h"
#include "World.h"
#include "Cat.h"

int main(int argc, char **argv)
{
        std::vector<Cat> cats;
        //can't move cats around in memory, due to world essentially saving their pointers
        //given the lambdas.
        cats.reserve(5);

        std::random_device rd;
        std::mt19937 random_engine(rd());
        std::uniform_real_distribution<float> dist(0, 800.f);

        World world(800, 800);
        Mouse mouse(world, 50.f, 50.f);

        for (int i = 0; i < 5; ++i)
        {
                cats.emplace_back(world, dist(random_engine), dist(random_engine));
        }

        for (int i = 0; i < 5; ++i)
```

```
        {
                float x = dist(random_engine);
                float y = dist(random_engine);
                Rect cheese(x, y, 10.f, Color::make_from_floats(1.f, 1.f, 0.f));
                world.add_object("cheese", [](float dt){}, [cheese]() {return cheese; });
        }

        Window window("mouse and cat and cheese", 800, 800);

        Timer timer;
        timer.Start();
        int last_update = timer.ElapsedMilliseconds();

        while (window.Open())
        {
                window.PollEvents();

                if (timer.ElapsedMilliseconds() != last_update)
                {
                        long elapsed = timer.ElapsedMilliseconds();
                        float delta = (elapsed - last_update) / 1000.f;
                        last_update = elapsed;
                        world.update(delta);

                }
                world.render(window);

                window.Display();
                window.Clear();
        }

        return 0;
}
```

2.5.3.8 Mouse.cpp

```
#include "Mouse.h"


Mouse::Mouse(World& world, float x, float y)
        :rect_(x, y, 10.f, Color::blue())
        ,wanderer_(world)
{
        timer_.Start();
        std::random_device rd;
        random_engine_.seed(rd());
        hunger_meter_ = 10;
        id_ = world.add_object("mouse", [this](float dt){ update(dt); }, [this]() -
> const Rect& {return rect_; });
        state_machine_.push_state([this, &world](){ wander(world); });
}

void Mouse::update(float dt)
{
        delta_time_ = dt;
        hunger_meter_ -= dt;
;
        state_machine_.execute_current();
}

void Mouse::wander(World& world)
{
```

```
        if (hunger_meter_ < 5)
        {
                state_machine_.pop_state();
                state_machine_.push_state([this, &world](){ find_cheese(world); });
                //don't waste an update.
                find_cheese(world);
        }
        else
        {
                std::uniform_real_distribution<float> dist(-1.f, 1.f);
                float x = dist(random_engine_);
                float y = dist(random_engine_);
                wanderer_.wander(rect_, delta_time_, 50.f);
        }

        detect_cat(world);

}

void Mouse::find_cheese(World& world)
{
        auto cheeses = world.find_with_tag("cheese");
        if (cheeses.size() != 0)
        {
                auto begin = cheeses.begin();
                auto end = cheeses.end();

                auto current = (begin++);
                for (; begin != end; ++begin)
                {
                        if (current->second().distance_to(rect_) >
                                begin->second().distance_to(rect_))
                        {
                                current = begin;
                        }
                }
                auto closest_cheese = current;
                wanderer_.force_target(rect_, closest_cheese-
>second().center_x(), closest_cheese->second().center_y());
                wanderer_.wander(rect_, delta_time_, 50.f);
                float distance = rect_.distance_to(closest_cheese->second());
                if (rect_.circular_collision(closest_cheese->second()))
                {
                        int elapsed = timer_.ElapsedMilliseconds();
                        int id = closest_cheese->first;
                        state_machine_.pop_state();

        state_machine_.push_state([this, &world, id, elapsed](){ eat_cheese(world, id, elapsed
); });
                        //don't waste an update.
                        eat_cheese(world, id, elapsed);
                }
        }

        detect_cat(world);
}

void Mouse::eat_cheese(World& world, int cheese_id, int start_time)
{

        if (timer_.ElapsedMilliseconds() - start_time > 1000)
```

```
        {
                world.remove_object(cheese_id);
                hunger_meter_ += 5;
                if (hunger_meter_ < 5)
                {
                        state_machine_.pop_state();
                        state_machine_.push_state([this, &world](){ find_cheese(world); });
                        //don't waste an update.
                        find_cheese(world);
                }
                else
                {
                        state_machine_.pop_state();
                        state_machine_.push_state([this, &world](){ wander(world); });
                        //don't waste an update.
                        wander(world);
                }
        }

        detect_cat(world);
}

void Mouse::run_away(World& world, std::function<const Rect&()> rect_func)
{
        const Rect& rect = rect_func();
        if (rect.distance_to(rect_) < 400.f)
        {
                wanderer_.force_reverse_target(rect_, rect.center_x(), rect.center_y());
                wanderer_.wander(rect_, delta_time_, 100.f);
        }
        else
        {
                state_machine_.pop_state();
        }
}

void Mouse::detect_cat(World& world)
{
        auto cats = world.find_with_tag("cat");
        if (cats.size() != 0)
        {
                for (auto& cat : cats)
                {
                        if (cat.second().distance_to(rect_) < 200.f)
                        {
                                auto func = cat.second;

        state_machine_.push_state([this, &world, func](){ run_away(world, func); });
                                break;
                        }
                }
        }
}
```

2.5.3.9 StateMachine.cpp

```
#include "StateMachine.h"

void StateMachine::pop_state()
{
        state_stack_.pop_back();
}
```

```
void StateMachine::push_state(std::function<void()> state)
{
	state_stack_.push_back(state);
}

void StateMachine::execute_current()
{
	state_stack_.back()();
}
```

2.5.3.10 Wanderer.cpp

```
#include "Wanderer.h"
#include <random>

Wanderer::Wanderer(World& world)
	: has_target_(false)
	, dist_x_(0.f, world.width())
	, dist_y_(0.f, world.height())
{
	std::random_device rd;
	random_.seed(rd());
	random_coord_ = [this]{ return std::make_pair(dist_x_(random_), dist_y_(random_)); };
}

void Wanderer::reset()
{
	has_target_ = false;
}

void Wanderer::force_target(const Rect& rect, float x, float y)
{
	has_target_ = true;

	target_x_ = x;
	target_y_ = y;
	start_x_ = rect.center_x();
	start_y_ = rect.center_y();
	x = target_x_ - start_x_;
	y = target_y_ - start_y_;
	float len = std::sqrtf((x * x) + (y * y));
	target_wander_len_ = len;
	wander_len_ = 0.f;
}

void Wanderer::force_reverse_target(const Rect& rect, float x, float y)
{
	has_target_ = true;

	start_x_ = rect.center_x();
	start_y_ = rect.center_y();

	target_x_ = start_x_ - x;
	target_y_ = start_y_ - y;

	x = target_x_ - start_x_;
	y = target_y_ - start_y_;
	float len = std::sqrtf((x * x) + (y * y));
	target_wander_len_ = len;
	wander_len_ = 0.f;
}
```

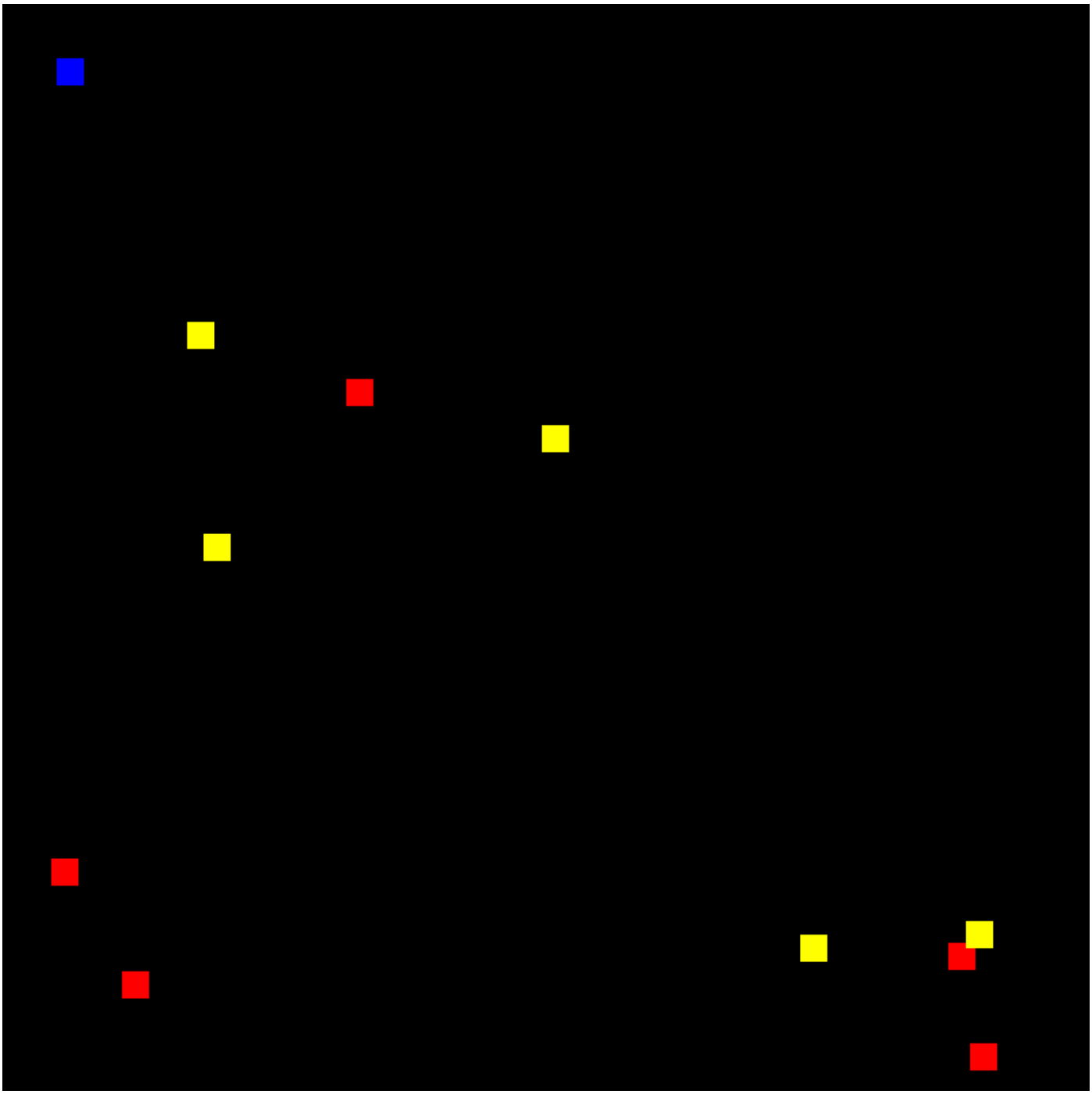
```
void Wanderer::wander(Rect& rect, float dt, float speed)
{
	if (!has_target_)
	{
		auto target = random_coord_();
		force_target(rect, target.first, target.second);
	}
	float x = target_x_ - rect.center_x();
	float y = target_y_ - rect.center_y();

	float len = std::sqrtf((x * x) + (y * y));
	x = (x / len) * dt * speed;
	y = (y / len) * dt * speed;

	rect.move(x, y);
	len = std::sqrtf((x * x) + (y * y));
	wander_len_ += len;

	if (wander_len_ > target_wander_len_)
	{
		has_target_ = false;
	}
}
```

2.5.3.11 World.cpp

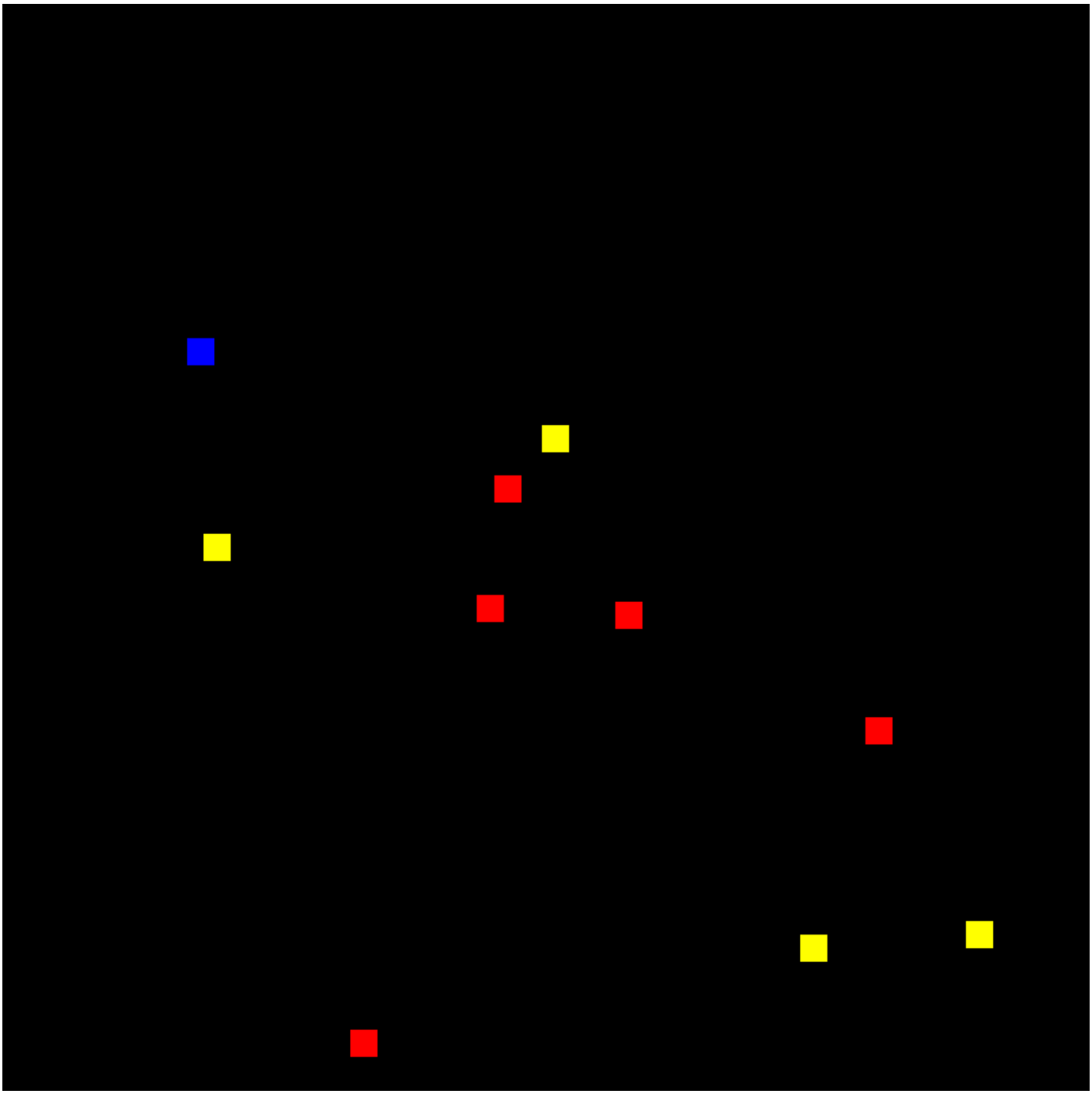
```
#include "World.h"

World::World(float w, float h)
	:w_(w), h_(h) {}

int World::add_object(std::string tag, std::function<void(float)> update, std::function<const
Rect&()> rect)
{
	objects_.emplace(next_id_, Object{ tag, update, rect });
	return next_id_++;
}

void World::update(float dt)
{
	cull_dead();
	for (auto& obj : objects_)
	{
		obj.second.update(dt);
	}
}

std::vector<std::pair<int, std::function<const Rect&()>>> World::find_with_tag(const std::stri
ng& tag) const
{
	std::vector<std::pair<int, std::function<const Rect&()>>> ret;
	for (auto& obj : objects_)
	{
		if (obj.second.tag == tag)
		{
			ret.emplace_back(obj.first, obj.second.rect);
		}
	}
	return std::move(ret);
}
```

```
void World::remove_object(int id)
{
        to_remove_.push_back(id);
}

void World::cull_dead()
{
        for (int i : to_remove_)
        {
                objects_.erase(i);
        }
        to_remove_.clear();
}

void World::render(Window& window)
{
        for (auto& obj : objects_)
        {
                const Rect& r = obj.second.rect();
                window.RenderRectangle(r.center_x() - r.extends(), r.center_y() -
 r.extends(), r.extends() * 2, r.extends() * 2, r.color());
        }
}
```

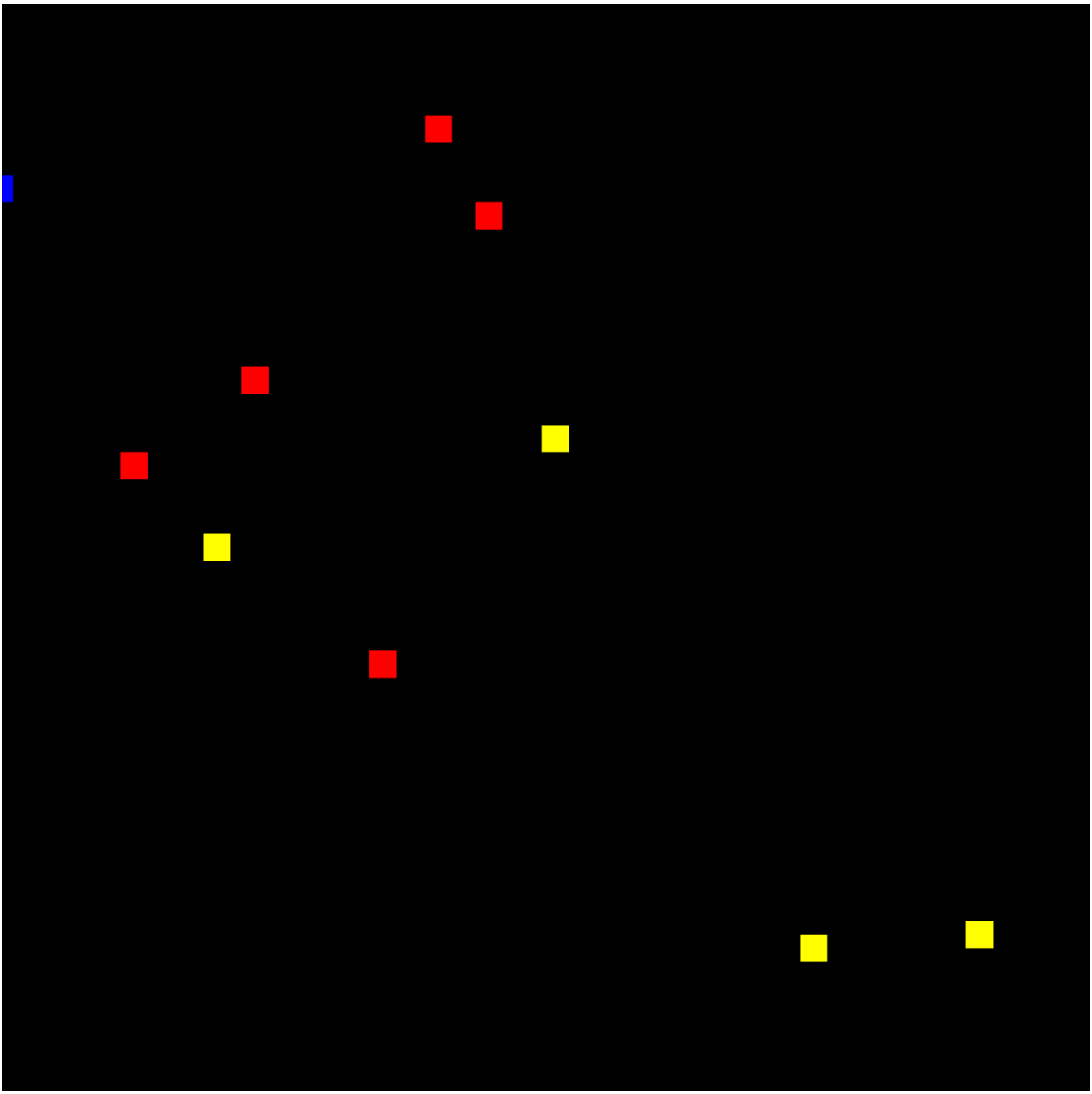
2.5.4 Körexempel



Figur 15. Startfältet, med objekt på godtyckliga positioner i fältet.



Figur 16. Här lyckas musen äta upp en ost.



Figur 17. Här är musen nästan utjagad ur spelfältet.

2.5.5 Analys

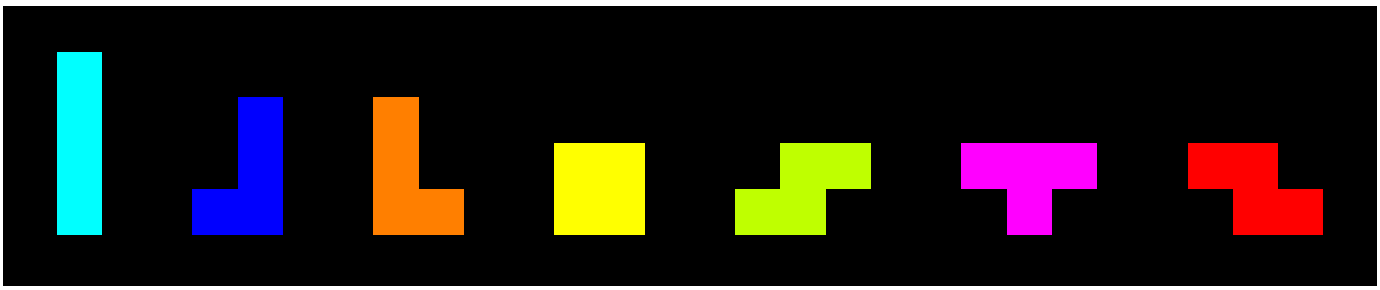
Tillståndsmaskinen gör det väldigt enkelt att implementera detta beteende, men framförallt gör det det enkelt att bygga ut beteendet. Exempelvis skulle det vara enkelt att lägga till tillstånd som låter musen sova eller interagera med andra möss. Framförallt att lösningen är stackbaserad, vilket gör det möjligt att gå tillbaka till ett tidigare tillstånd oavsett vilket det tillståndet var, gör det möjligt att uttöka beteendet.

3 Applikation

3.1 Problem

I den här rapporten är problemet för applikationen att skapa en AI som kan spela tetris.

I klassiskt testris kontrollerar spelare en fallande spelpjäs, en s.k. tetromino, av olika former. Det finns 7 olika former vilka är namngedda efter hur de ser ut:



Figur 18. Från vänster till höger: I, J, L, O, S, T och Z.

När pjäser landar fastnar dem på banan och en ny pjäs faller. Vilket pjäs som faller är slumpmässigt, men spelare kan alltid se vilket pjäs som kommer falla efter den pjäs spelaren för tillfället kontrollerar. Banan består av celler, 10 på bredden och 20 på höjden. När pjäsen träffar marken låses pjäsen in i spelplanen och nästa pjäs kan även träffa den. Målet är att låsa pjäser för att fylla rader i spelpjäsen. När spelare har fyllt en rad rensas den raden och alla rader ovanför faller ner en rad. Spelare kan som max rensa 4 rader per fallande pjäs, detta eftersom att den längsta pjäsen (I) är 4 rader lång. Spelaren får mer poäng för att rensa fler rader åt gången. Var tionde rad ökas nivån på spelet och pjäser faller snabbare. Nivån påverkar också hur många poäng spelaren får för radrensning:

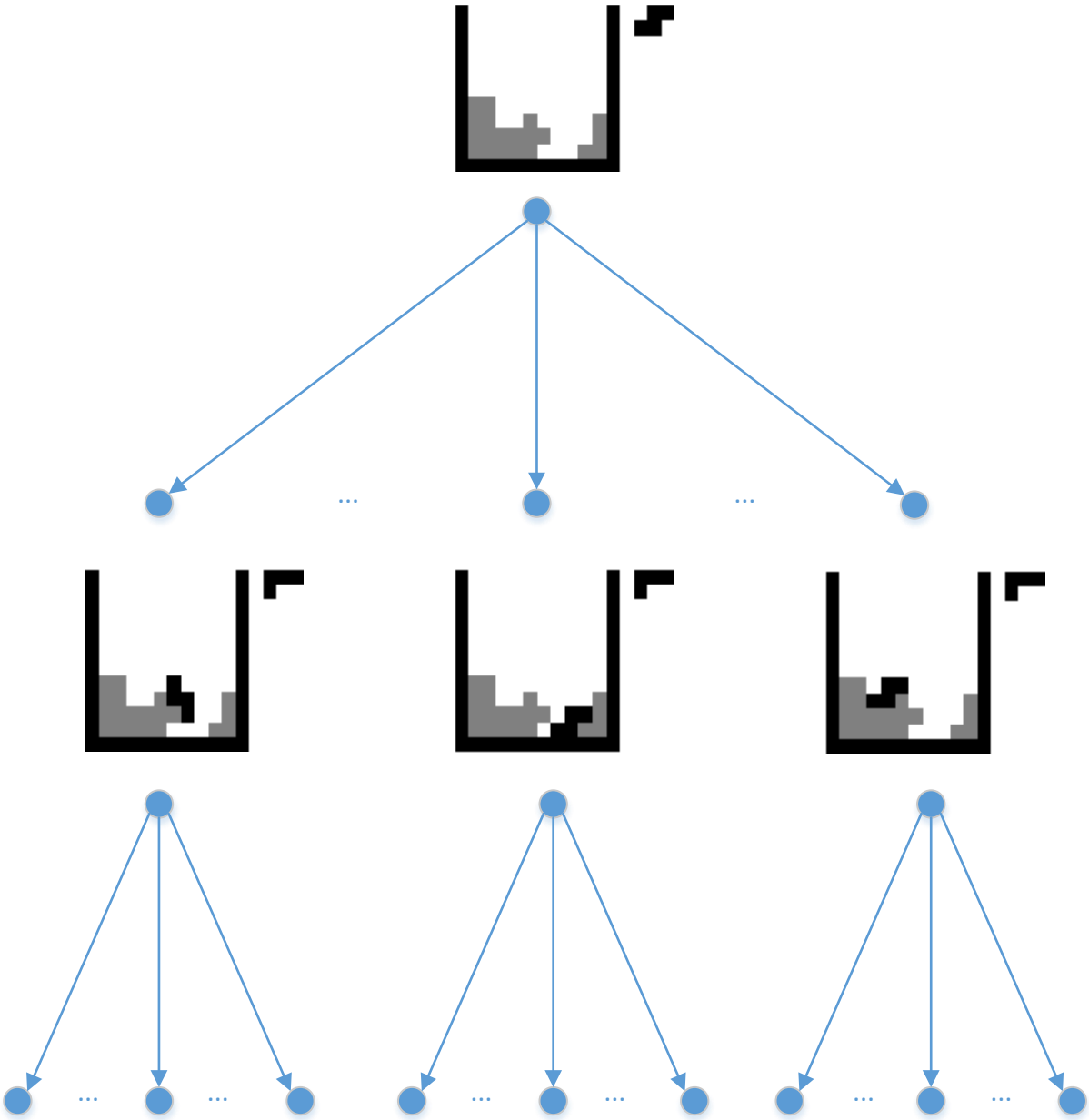
Nivå	1 rad	2 rader	3 rader	4 rader
n	40 * (n + 1)	100 * (n + 1)	300 * (n + 1)	1200 * (n + 1)

3.2 Design

Designen är två-fald. Dels ska tetris-spelet byggas, och dels ska en AI som kan lösa den tetris-implementationen byggas.

För att AI:n ska kunna spela spelet kan den ta reda på vilken pjäs som faller för tillfället, samt se framtida pjäser (det finns ingen gräns på hur många pjäser som går att få tag på annat än för minnet). AI:n kan styra pjäsen genom att skicka in händelser till tetris-implementationen.

Varje gång en ny tetromino faller söker AI:n igenom alla möjliga tillstånd som orskar ett lås (att tetrominon ”fastnar”) på tetris-planen. Den söker därefter igenom alla möjliga tillstånd för nästa tetromino, och bygger på så sätt upp ett träd av möjliga tillstånd som är n djupt, där n är hur många tetrominon som spelet förhandsvisar plus den tetromino som faller på spelplanen. I teorin kan n vara hur djupt som helst, men är i den här lösningen begränsad till den som fallar, och en som förhandsvisas.



Från detta har AI:n ett antal möjliga tillstånd. Av dessa tillstånd väljer AI:n ut en som ska användas, vilket beror på ett antal evalueringsfunktioner som på något sätt analyserar resultatet av detta tillståndet på spelplanen. Resultatet av dessa funktioner representeras i ett realtal, och alla resultat adderas. Det tillstånd som ger lägst värde är det som väljs ut av AI:n. AI:n kan sedan visualisera detta resultat, genom att flytta tetiminon till rätt position.

Figur 19. Exempel på ett träd av tillstånd från ett startfält.

Varje evalueringsfunktion har en vikt associerad till sig. Dessa vikter kan tas bland annat skrivas för hand eller så kan de tas fram med hjälp av en evolutionär teknik. I den här lösningen tas de fram med hjälp av en evolutionär teknik. Den populationen sparas också till en fil, så att programmet alltid kan fortsätta den evolutionära processesen, vilket gör det enklare att köra processen under en längre tid och på så sätt – förhoppningsvis – få bättre resultat. Detta betyder också att AI:n aldrig blir klar, den blir bara bättre och bättre –även detta en förhoppning. Evalueringsfunktionera analyserar spelfälet så som det ser ut i det aktuella tillståndet efter de två pjäserna har låsts (den nuvarande pjäsen och den pjäsen som förhandsvisas). Funktionerna är följande:

- Totala antalet rensade rader

Summan av antalet rensade rader för de båda pjäserna.

- Totala låshöjden

  Låshöjden är hur långt från marken pjäsen låstes. Den totala låshöjden är summan av de båda pjäsernas låshöjd.

- Totala antalet väggceller

  En väggcell är en tom cell vars vänstra och högra cell inte är tomma, eller är väggceller.

- Totala antalet kolonhåll

  Antalet tomma celler direkt under en solid cell.

- Totala antalet kolonövergångar

  En kolonövergång är en tom cell bredvid en solid cell i samma kolon.

- Totala antalet radövergångar

  En radövergång är en tom cell bredvid en solid cell i samma rad.

- Stackhöjd

  Antalet rader som innehåller minst ett solit block.

Generna i den evolutionära teknik som används är helt enkelt vikterna. Mutation av genomet görs genom att slumpmässigt variera vikterna, detta görs genom att låta den nya vikten få ett slumpat värde i ett interval run det gamla värdet (detta för att ge en gräns för hur stora eller små värdena kan bli, så att värdena inte kan bli godtyckligt höga/låga). Vid överkorsning skyfflas dessa vikter mellan två olika individer. En AI kan skapas med dessa vikter och spela ett spel. När varje individ i populationen har spelat ett spel används dess poäng som fitnessfunktion och en ny population skapas från den gamla.

3.3 Kod

3.3.1 Board.h

```
#pragma once

#include "Window.h"
#include <array>
#include "Piece.h"
#include <random>
#include <functional>
#include "PlayField.h"
#include <deque>

#define BOARD_HEIGHT 20
#define BOARD_WIDTH 10

class Board
{
public:
        enum Action { Rotate, Left, Right };

        Board(int x, int y, int tile_size);
        void render(Window& window);

        bool perform_action(Action action);

        int tick();
```

```
        bool test_collision(const Piece& piece) const;

        const Piece& get_current_piece() const;
        const Piece& get_next_piece(int index);
        int get_piece_count() const;

        PlayField create_play_field() const;

        int get_width() const { return BOARD_WIDTH; }
        int get_height() const { return BOARD_HEIGHT; }
private:
        struct  Tile
        {
                Tile()
                        :color(Color::black()), occupied(false)
                {}

                Tile(Color color_, bool occupied_)
                        :color(color_), occupied(occupied_)
                {}
                Color color;
                bool occupied;
        };

        void render_board(Window& window);
        void render_tiles();
        void render_live_piece();

        void set_color(Color color, int x, int y);
        void clear_colors();
        bool imprint_live_piece();
        int clear_rows();

        Piece random_piece();
        int next_random(int min, int max);

        int x_, y_, tile_size_;
        std::array<Color, BOARD_HEIGHT * BOARD_WIDTH> colors_;
        std::array<Tile, BOARD_HEIGHT * BOARD_WIDTH> tiles_;

        std::mt19937 random_engine_;
        std::vector<std::function<Piece(int, int, int)>> piece_makers;

        Piece current_piece_;
        std::deque<Piece> next_piece_queue_;
        int piece_count_;
};
```

3.3.2 EvaluationFunctions.h

```
#pragma once

#include "PlayField.h"

template<int ID>
struct EvaluationFunction
{

};
```

```
template<>
struct EvaluationFunction<0>
{
	/*
	Total Lines Cleared
	*/
	double operator()(const PlayField& from, const PlayField& to, const std::vector<Piece>
& locked_pieces)
	{
		int lines_cleared = from.get_cleared_rows() - to.get_cleared_rows();
		if (lines_cleared == 0)
		{
			return 0.f;
		}
		else if (lines_cleared == 1)
		{
			return 0.1f;
		}
		else if (lines_cleared == 2)
		{
			return 0.3f;
		}
		else if (lines_cleared == 3)
		{
			return 0.6f;
		}
		else if (lines_cleared >= 4)
		{
			return 1.0f;
		}
		return static_cast<double>(from.get_cleared_rows() - to.get_cleared_rows());
	};
};

template<>
struct EvaluationFunction<1>
{
	/*
	Total Lock Height
	*/
	double operator()(const PlayField& from, const PlayField& to, const std::vector<Piece>
& locked_pieces)
	{
		int lock_height = 0;
		for (auto& p : locked_pieces)
		{
			lock_height = get_lock_height(p, to.get_height());
		}
		return lock_height;
	};

	int get_lock_height(const Piece& piece, int height)
	{
		auto tiles = piece.get_tiles();
		int lowest_y = piece.get_y() - 2;
		for (int x = 0; x < PIECE_SIZE; ++x)
		{
			for (int y = 0; y < PIECE_SIZE; ++y)
			{
				bool occ = tiles[y * PIECE_SIZE + x];
				if (occ)
```

```
                                {
                                        lowest_y = (piece.get_y() - 2) + y;
                                }
                        }
                }
                return (height - lowest_y) + 1;
        }
};


template<>
struct EvaluationFunction<2>
{
        /*
                Total Well Cells
        */
        double operator()(const PlayField& from, const PlayField& to, const std::vector<Piece>
& locked_pieces)
        {
                int wall_cells = 0;
                for (int x = 0; x < to.get_width(); ++x)
                {
                        int first_top = -1;
                        for (int y = 0; y < to.get_height(); ++y)
                        {
                                if (to.get(x, y))
                                {
                                        first_top = y - 1;
                                        break;
                                }
                        }
                        if (first_top != -1 && first_top != to.get_height() - 1)
                        {
                                if (x == 0 || x == to.get_width() - 1)
                                {
                                        wall_cells++;
                                }
                                else if (!to.get(x -
 1, first_top) && !to.get(x + 1, first_top))
                                {
                                        wall_cells++;
                                }
                        }
                }
                return static_cast<double>(wall_cells);
        };
};


template<>
struct EvaluationFunction<3>
{
        /*
        Total Column Holes
        */
        double operator()(const PlayField& from, const PlayField& to, const std::vector<Piece>
& locked_pieces)
        {
                int holes = 0;
                for (int x = 0; x < to.get_width(); ++x)
                {
```

```
                        for (int y = 1; y < to.get_height() - 1; ++y)
                        {
                                if (!to.get(x, y) && to.get(x, y - 1))
                                {
                                        holes++;
                                }
                        }

                }
                return static_cast<double>(holes);
        };
};


template<>
struct EvaluationFunction<4>
{
        /*
        Total Column Transitions
        */
        double operator()(const PlayField& from, const PlayField& to, const std::vector<Piece>
& locked_pieces)
        {
                int transitions = 0;
                for (int x = 0; x < to.get_width(); ++x)
                {
                        for (int y = 1; y < to.get_height(); ++y)
                        {
                                if (!to.get(x, y - 1) && to.get(x, y))
                                {
                                        transitions++;
                                }
                        }

                }
                return static_cast<double>(transitions);
        };

};

template<>
struct EvaluationFunction<5>
{
        /*
        Total Row Transitions
        */
        double operator()(const PlayField& from, const PlayField& to, const std::vector<Piece>
& locked_pieces)
        {
                int transitions = 0;
                for (int y = 1; y < to.get_height() - 1; ++y)
                {
                        int this_row = 0;
                        bool empty = true;
                        for (int x = 0; x < to.get_width(); ++x)
                        {
                                if (to.get(x, y))
                                {
                                        if (!to.get(x - 1, y))
                                        {
                                                transitions++;
```

```
                                        }
                                        empty = false;
                                        break;
                                }
                        }
                        if (!empty)
                        {
                                transitions += this_row;
                        }

                }
                return static_cast<double>(transitions);
        };
};

template<>
struct EvaluationFunction<6>
{
        /*
        Pile Height
        */
        double operator()(const PlayField& from, const PlayField& to, const std::vector<Piece>
& locked_pieces)
        {
                int pile_height = 0;
                for (int y = 1; y < to.get_height() - 1; ++y)
                {
                        for (int x = 0; x < to.get_width(); ++x)
                        {
                                if (to.get(x, y))
                                {
                                        pile_height++;
                                        break;
                                }
                        }
                }
                return static_cast<double>(pile_height);
        };
};
```

3.3.3 Piece.h

```
#pragma once

#include <array>
#include "Color.h"
#include <bitset>

#define PIECE_SIZE 5

class Piece
{
public:
        Piece() {};

        static Piece make_O(int x, int y, int rotation);
        static Piece make_I(int x, int y, int rotation);
        static Piece make_S(int x, int y, int rotation);
        static Piece make_Z(int x, int y, int rotation);
        static Piece make_L(int x, int y, int rotation);
```

```
        static Piece make_J(int x, int y, int rotation);
        static Piece make_T(int x, int y, int rotation);

        void rotate_left();
        void rotate_right();
        void move(int dx, int dy);

        std::array<int, PIECE_SIZE * PIECE_SIZE> get_tiles() const;
        Color get_color() const;
        int get_x() const;
        int get_y() const;
        int get_rotation() const;
        int get_max_rotations() const;

        void set(int x, int y, int rotation);
private:
        static void setup_O();
        static void setup_I();
        static void setup_S();
        static void setup_Z();
        static void setup_L();
        static void setup_J();
        static void setup_T();

        Piece(int rotation, int x, int y, int type);

        struct Settings
        {
                Settings()
                        :setup(false)
                {}


        Settings(std::array<int, PIECE_SIZE * PIECE_SIZE> tiles_, Color color_, int max_rotati
ons_, bool sz_exception_, bool reverse_rotate_)
                        : color(color_)
                        , max_rotations(max_rotations_)
                        , sz_exception(sz_exception_)
                        , reverse_rotate(reverse_rotate_)
                        , setup(true)
                {
                        for (int i = 0; i < PIECE_SIZE * PIECE_SIZE; ++i)
                        {
                                tiles[i] = tiles_[i] == 1;
                        }
                }
                //These can easily be feather weighted.
                std::bitset<PIECE_SIZE * PIECE_SIZE> tiles;
                Color color;
                bool sz_exception;
                bool reverse_rotate;
                int max_rotations;
                bool setup;
        };

        static std::array<Settings, 7U> settings_;

        int rotation_, x_, y_, type_;

};
```

3.3.4 PlayField.h

```
#pragma once

/*	A simplistic, less memory-intense version of Board.
	For use with the Solver
*/

#include "Piece.h"
#include <vector>

class PlayField
{
public:
	PlayField(int w, int h);
	void set(int x, int y, bool occupied);
	bool get(int x, int y) const;

	bool test_collision(const Piece& piece) const;

	/* This returns true if the piece was imprintable.
	 * If this returns false, but test_collision returns true
	 * this means the player has lost the game.
	 */
	bool imprint(const Piece& piece);

	int get_width() const { return w_; }
	int get_height() const { return h_; }
	int get_cleared_rows() const { return cleared_rows_; };

private:
	int index_of(int x, int y) const;
	/* Returns number of rows cleared. */
	int clear_rows();

	std::vector<bool> tiles_;
	int cleared_rows_;
	int w_, h_;
};
```

3.3.5 Solver.h

```
#pragma once

#include "Board.h"
#include <deque>
#include "MultiArray.h"
#include "StateQueue.h"

class Solver
{
public:
	typedef std::function<double(const PlayField& from, const PlayField& to, const std::ve
ctor<Piece>& piece_queue)> evaluation_function;

	Solver(std::vector<std::pair<evaluation_function, double>> evaluations);
	Solver();
	void update(Board& board);
```

```
        bool is_done() const;
        double get_score() const;
private:
        typedef std::deque<std::deque<Board::Action>> Recording;
        typedef MultiArray<State::ptr, 4> StateArray;
        State::ptr search(StateArray& states, PlayField& original_play_field, PlayField& play_
field, int depth, const std::vector<Piece>& piece_queue,  std::vector<Piece>& locked_pieces);
        void start_search(Board& board);
        void play_recorded_actions(Board& board);
        StateArray build_states(const PlayField& play_field, const std::vector<Piece>& piece_q
ueue) const;
        Recording make_recording(State::ptr prev_state, State::ptr start) const;
        void handle_tick(int tick);

        double evaluate_play_field(const PlayField& from, const PlayField& to, const std::vect
or<Piece>& piece_queue) const;

        std::vector<std::pair<evaluation_function, double>> evaluations_;

        /* return true if the state is valid, regardless if the state to the queue was added o
r not */
        bool add_state_to_queue(StateArray& states, StateQueue& queue, State::ptr prev_state,
const PlayField& board, const Piece& piece, int depth);

        Recording action_recording_;
        int current_piece_count_;
        int level_;
        int lines_cleared_this_level_;

        double score_;
        bool running_;

};
```

3.3.6 SolverEvolver.h

```
#pragma once

#include "Random.h"
#include "Solver.h"
#include "Board.h"
#include <utility>

class SolverEvolver
{
public:
        SolverEvolver(int population_size, double combination_probability, double mutation_pro
bability);

        void update(Window& window);
private:
        typedef std::array<double, 7U> Genome;
        struct Individual
        {
                Genome genome;
                std::shared_ptr<Solver> solver;
                std::shared_ptr<Board> board;
        };

        void initate_population();
```

```
        void crossover(Genome&, Genome&);
        void mutate(Genome& n);
        void evolve();

        void sort_by_fitness();

        template<int i>
        std::pair<Solver::evaluation_function, double> get_evaluation(const Genome& g)
        {
                return std::make_pair(EvaluationFunction<i>(), g[i]);
        }

        Randomizer rand_;
        std::vector<Individual> population_;
        Individual& select();
        Genome random_genome();
        double random_double(double middle);

        int population_size_;
        float combination_p_, mutation_p_;
        int steps_;
};
```

3.3.7 State

```
#pragma once

/*
 *      http://meatfighter.com/nintendotetrisai/#The_Algorithm
 */

#include <utility>
#include <memory>
#include "Board.h"

struct State
{
        typedef std::shared_ptr<State> ptr;
        typedef std::weak_ptr<State> weak_ptr;

        std::deque<std::deque<Board::Action>> recording_;

        Piece piece;
        bool visited;
        weak_ptr predecessor;
        weak_ptr next;
};
```

3.3.8 StateQueue.h

```
#pragma once

#include "State.h"
class StateQueue
{
public:
        void enqueue(State::ptr state)
        {
                if (head == nullptr)
                {
```

```
                        head = state;
                        tail = state;
                }
                else
                {
                        tail->next = state;
                        tail = state;
                }
                state->next.lock() = nullptr;
        }

        State::ptr dequeue()
        {
                auto state = head;
                if (head != nullptr)
                {
                        if (head == tail)
                        {
                                head = nullptr;
                                tail = nullptr;
                        }
                        else
                        {
                                head = head->next.lock();
                        }
                }
                return state;
        }

        bool is_empty()
        {
                return head == nullptr;
        }

        void clear()
        {
                while (!is_empty())
                {
                        dequeue();
                }
        }

private:
        State::ptr head;
        State::ptr tail;
};
```

3.3.9 Board.cpp

```
#include "Board.h"
#include <exception>

Board::Board(int x, int y, int tile_size)
{
        piece_count_ = 0;

        std::random_device rd;
        random_engine_ = std::mt19937(rd());
        clear_colors();
        x_ = x + tile_size;
```

```
        y_ = y + tile_size;
        tile_size_ = tile_size;

        piece_makers.push_back([](int x, int y, int rotation){return Piece::make_I(x, y, rotat
ion); });
        piece_makers.push_back([](int x, int y, int rotation){return Piece::make_J(x, y, rotat
ion); });
        piece_makers.push_back([](int x, int y, int rotation){return Piece::make_L(x, y, rotat
ion); });
        piece_makers.push_back([](int x, int y, int rotation){return Piece::make_O(x, y, rotat
ion); });
        piece_makers.push_back([](int x, int y, int rotation){return Piece::make_S(x, y, rotat
ion); });
        piece_makers.push_back([](int x, int y, int rotation){return Piece::make_T(x, y, rotat
ion); });
        piece_makers.push_back([](int x, int y, int rotation){return Piece::make_Z(x, y, rotat
ion); });

        current_piece_ = random_piece();
        next_piece_queue_.push_back(random_piece());
}

void Board::render(Window& window)
{
        clear_colors();
        render_tiles();
        render_live_piece();
        render_board(window);
}

void Board::render_tiles()
{
        for (int x = 0; x < BOARD_WIDTH; ++x)
        {
                for (int y = 0; y < BOARD_HEIGHT; ++y)
                {
                        if (tiles_[y * BOARD_WIDTH + x].occupied)
                        {

        colors_[y * BOARD_WIDTH + x] = tiles_[y * BOARD_WIDTH + x].color;
                        }
                }
        }
}

void Board::render_live_piece()
{
        Color color = current_piece_.get_color();
        auto tiles = current_piece_.get_tiles();
        for (int i = 0; i < PIECE_SIZE; ++i)
        {
                for (int j = 0; j < PIECE_SIZE; ++j)
                {
                        int x = current_piece_.get_x() + i - 2;
                        int y = current_piece_.get_y() + j - 2;

        if (x >= 0 && x < BOARD_WIDTH && y >= 0 && y < BOARD_HEIGHT && tiles[j * PIECE_SIZE +
i] != 0)
                        {
                                set_color(color, x, y);
                        }
```

```
                }
        }
}

void Board::render_board(Window& window)
{
        for (int x = 0; x < BOARD_WIDTH; ++x)
        {
                window.RenderRectangle(x_ + (x * tile_size_), y_ + (-
1 * tile_size_), tile_size_, tile_size_, Color::white());

        window.RenderRectangle(x_ + (x * tile_size_), y_ + ((BOARD_HEIGHT) * tile_size_), tile
_size_, tile_size_, Color::white());
        }

        for (int y = -1; y <= BOARD_HEIGHT; ++y)
        {
                window.RenderRectangle(x_ + (-
1 * tile_size_), y_ + (y * tile_size_), tile_size_, tile_size_, Color::white());

        window.RenderRectangle(x_ + ((BOARD_WIDTH) * tile_size_), y_ + (y * tile_size_), tile_
size_, tile_size_, Color::white());
        }

        for (int x = 0; x < BOARD_WIDTH; ++x)
        {
                for (int y = 0; y < BOARD_HEIGHT; ++y)
                {

        window.RenderRectangle(x_ + (x * tile_size_), y_ + (y * tile_size_), tile_size_, tile_
size_, colors_[y * BOARD_WIDTH + x]);
                }
        }
}

void Board::clear_colors()
{
        for (auto& tile : colors_)
        {
                tile = Color::black();
        }
}

void Board::set_color(Color color, int x, int y)
{
        if (x >= BOARD_WIDTH || y >= BOARD_HEIGHT || x < 0 || y < 0)
        {
                throw std::out_of_range("Tile must be inside the board");
        }
        colors_[y * BOARD_WIDTH + x] = color;
}

bool Board::test_collision(const Piece& piece) const
{
        auto tiles = piece.get_tiles();
        for (int i = 0; i < PIECE_SIZE; ++i)
        {
                for (int j = 0; j < PIECE_SIZE; ++j)
                {
                        int x = piece.get_x() + i - 2;
                        int y = piece.get_y() + j - 2;
```

```
                        if (tiles[j * PIECE_SIZE + i] != 0)
                        {
                                if (y >= BOARD_HEIGHT)
                                {
                                        return true;
                                }

        else if (x >= 0 && x < BOARD_WIDTH && y >= 0 && y < BOARD_HEIGHT && tiles_[y * BOARD_W
IDTH + x].occupied)
                                {
                                        return true;
                                }
                                else if (!(x >= 0 && x < BOARD_WIDTH))
                                {
                                        return true;
                                }
                        }
                }
        }
        return false;
}

/*
        Returns true if the action was legal
*/
bool Board::perform_action(Action action)
{
        if (action == Action::Rotate)
        {
                current_piece_.rotate_right();
                if (test_collision(current_piece_))
                {
                        current_piece_.rotate_left();
                        return false;
                }
        }
        else if (action == Action::Left)
        {
                current_piece_.move(-1, 0);
                if (test_collision(current_piece_))
                {
                        current_piece_.move(1, 0);
                        return false;
                }
        }
        else if (action == Action::Right)
        {
                current_piece_.move(1, 0);
                if (test_collision(current_piece_))
                {
                        current_piece_.move(-1, 0);
                        return false;
                }
        }
        return true;
}

/*
        Does a tick and returns how many rows were cleared in that tick, if any.
        Returns -1 if the game was lost this tick.
*/
```

```
int Board::tick()
{
        current_piece_.move(0, 1);
        bool collided = test_collision(current_piece_);

        if (collided)
        {
                while (collided)
                {
                        current_piece_.move(0, -1);

        collided = test_collision(current_piece_) && current_piece_.get_y() > -5;
                }

                if (imprint_live_piece())
                {
                        return clear_rows();
                }
                else
                {
                        return -1;
                }
        }
        return 0;
}

/* Returns number of rows cleared. */
int Board::clear_rows()
{
        int cleared = 0;
        for (int row = 0; row < BOARD_HEIGHT; ++row)
        {
                bool occ = true;
                for (int col = 0; col < BOARD_WIDTH; ++col)
                {
                        if (!tiles_[row * BOARD_WIDTH + col].occupied)
                        {
                                occ = false;
                                break;
                        }
                }
                if (occ)
                {
                        ++cleared;
                        for (int col = 0; col < BOARD_WIDTH; ++col)
                        {
                                tiles_[row * BOARD_WIDTH + col].occupied = false;
                        }
                        for (int y = row; y >= 1; --y)
                        {
                                for (int x = 0; x < BOARD_WIDTH; ++x)
                                {
                                        tiles_[y* BOARD_WIDTH + x] = tiles_[(y -
 1) * BOARD_WIDTH + x];
                                }
                        }
                }
        }
        return cleared;
}
```

```
/* returns true if the piece was imprintable. Ie: if this function returns false,
        the game is lost */
bool Board::imprint_live_piece()
{
        Color color = current_piece_.get_color();
        auto tiles = current_piece_.get_tiles();
        for (int i = 0; i < PIECE_SIZE; ++i)
        {
                for (int j = 0; j < PIECE_SIZE; ++j)
                {
                        int x = current_piece_.get_x() + i - 2;
                        int y = current_piece_.get_y() + j - 2;
                        if (x >= 0 && x < BOARD_WIDTH && y >= -
1 && y < BOARD_HEIGHT && tiles[j * PIECE_SIZE + i] != 0)
                        {
                                if (y == -1)
                                {
                                        return false;
                                }
                                tiles_[y * BOARD_WIDTH + x].occupied = true;
                                tiles_[y * BOARD_WIDTH + x].color = color;
                        }
                }
        }

        current_piece_ = next_piece_queue_.front();
        next_piece_queue_.pop_front();

        next_piece_queue_.push_back(random_piece());
        ++piece_count_;
        return true;
}


int Board::next_random(int min, int max)
{
        std::uniform_int_distribution<int> distribution(min, max);
        return distribution(random_engine_);
}

Piece Board::random_piece()
{
        int piece = next_random(0, piece_makers.size() - 1);

        while (true)
        {
                bool cont = false;
                int rotation = next_random(0, 3);

                int x = next_random(-2, BOARD_WIDTH + 1);
                //scratch that
                x = 5;
                //that too
                rotation = 0;

                Piece ret = piece_makers[piece](x, 0, rotation);

                auto tiles = ret.get_tiles();
                for (int i = 0; i < PIECE_SIZE; ++i)
                {
                        for (int j = 0; j < PIECE_SIZE; ++j)
```

```
                        {
                                int x = ret.get_x() + i - 2;
                                int y = ret.get_y() + j - 2;

        if (tiles[j * PIECE_SIZE + i] != 0 && (x < 0  || x >= BOARD_WIDTH))
                                {
                                        cont = true;
                                }
                        }
                }
                if (!cont)
                {
                        return ret;
                }
        }
}

int Board::get_piece_count() const
{
        return piece_count_;
}

PlayField Board::create_play_field() const
{
        PlayField ret(BOARD_WIDTH, BOARD_HEIGHT);
        for (int y = 0; y < BOARD_HEIGHT; ++y)
        {
                for (int x = 0; x < BOARD_WIDTH; ++x)
                {
                        ret.set(x, y, tiles_[y * BOARD_WIDTH + x].occupied);
                }
        }
        return std::move(ret);
}

const Piece& Board::get_current_piece() const
{
        return current_piece_;
}

const Piece& Board::get_next_piece(int i)
{
        while (static_cast<int>(next_piece_queue_.size()) <= i)
        {
                next_piece_queue_.push_back(random_piece());
        }
        return next_piece_queue_[i];
}
```

3.3.10 Piece.cpp

```
#include "Piece.h"

std::array<Piece::Settings, 7U> Piece::settings_;

/*
 For reference see: http://www.colinfahey.com/tetris/tetris_diagram_pieces_orientations_new.jp
g
 */
Piece::Piece( int rotation, int x, int y, int type)
```

```
        : rotation_(rotation % settings_[type].max_rotations)
        , x_(x)
        , y_(y)
        , type_(type)
{

}

void Piece::rotate_left()
{
        if (settings_[type_].reverse_rotate)
        {
                rotation_++;
        }
        else
        {
                rotation_--;
        }

        if (rotation_ < 0)
        {
                rotation_ = settings_[type_].max_rotations - 1;
        }
        rotation_ = rotation_ % settings_[type_].max_rotations;
}

void Piece::rotate_right()
{
        if (settings_[type_].reverse_rotate)
        {
                rotation_--;
        }
        else
        {
                rotation_++;
        }
        if (rotation_ < 0)
        {
                rotation_ = settings_[type_].max_rotations - 1;
        }
        rotation_ = rotation_ % settings_[type_].max_rotations;
}

void Piece::move(int dx, int dy)
{
        x_ += dx;
        y_ += dy;
}

std::array<int, PIECE_SIZE * PIECE_SIZE> Piece::get_tiles() const
{
        std::array<int, PIECE_SIZE * PIECE_SIZE> ret;
        int grid_x = 0;
        int grid_y = 0;
        int x_mod = 0;

        switch (rotation_)
        {
        case 0:
                for (int x = 0; x < PIECE_SIZE; ++x)
                {
```

```
                        for (int y = 0; y < PIECE_SIZE; ++y)
                        {

        ret[grid_y * PIECE_SIZE + grid_x] = settings_[type_].tiles[y * PIECE_SIZE + x] ? 1 : 0
;
                                grid_y++;
                        }
                        grid_y = 0;
                        grid_x++;
                }
                break;
        case 1:
                if (settings_[type_].sz_exception)
                {
                        x_mod = 1;
                        grid_x = x_mod;
                        for (int x = 0; x < PIECE_SIZE; ++x)
                        {
                                for (int y = 0; y < PIECE_SIZE; ++y)
                                {
                                        ret[x * PIECE_SIZE + (y)] = 0;
                                }
                        }
                }

                for (int x = x_mod; x < PIECE_SIZE; ++x)
                {
                        for (int y = PIECE_SIZE - 1; y >= 0; --y)
                        {

        ret[grid_y * PIECE_SIZE + (grid_x)] = settings_[type_].tiles[y * PIECE_SIZE + (x -
 x_mod)] ? 1 : 0;
                                grid_x++;
                        }
                        grid_x = x_mod;
                        grid_y++;
                }
                break;
        case 2:
                for (int x = PIECE_SIZE - 1; x >= 0; --x)
                {
                        for (int y = PIECE_SIZE - 1; y >= 0; --y)
                        {

        ret[grid_y * PIECE_SIZE + grid_x] = settings_[type_].tiles[y * PIECE_SIZE + x] ? 1 : 0
;
                                grid_y++;
                        }
                        grid_y = 0;
                        grid_x++;
                }
                break;
        case 3:
                for (int x = PIECE_SIZE - 1; x >= 0; --x)
                {
                        for (int y = 0; y < PIECE_SIZE; ++y)
                        {

        ret[grid_y * PIECE_SIZE + grid_x] = settings_[type_].tiles[y * PIECE_SIZE + x] ? 1 : 0
;
                                grid_x++;
```

```
                        }
                        grid_x = 0;
                        grid_y++;
                }
                break;
        }
        return ret;
}

Color Piece::get_color() const
{
        return settings_[type_].color;
}

int Piece::get_x() const
{
        return x_;
}

int Piece::get_y() const
{
        return y_;
}

Piece Piece::make_O(int x, int y, int rotation)
{
        setup_O();
        return Piece(rotation, x, y, 0);
}

void Piece::setup_O()
{
        if (!settings_[0].setup)
        {
                settings_[0] = Settings({ {
                        0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0,
                        0, 1, 1, 0, 0,
                        0, 1, 1, 0, 0,
                        0, 0, 0, 0, 0
                        } },
                        Color::make_from_bytes(255, 255, 0),
                        1, false, false);
        }
}

Piece Piece::make_I(int x, int y, int rotation)
{
        setup_I();
        return Piece(rotation, x, y, 1);
}

void Piece::setup_I()
{
        if (!settings_[1].setup)
        {
                settings_[1] = Settings({ {
                                0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0,
                                0, 1, 1, 1, 1,
                                0, 0, 0, 0, 0,
```

```
                                0, 0, 0, 0, 0
                        } },
                        Color::make_from_bytes(0, 255, 255),
                        2, false, false);
        }
}

Piece Piece::make_S(int x, int y, int rotation)
{
        setup_S();
        return Piece(rotation, x, y, 2);
}

void Piece::setup_S()
{
        if (!settings_[2].setup)
        {
                settings_[2] = Settings({ {
                        0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0,
                        0, 0, 1, 1, 0,
                        0, 1, 1, 0, 0,
                        0, 0, 0, 0, 0
                        } },
                        Color::make_from_bytes(191, 255, 0),
                        2, true, false);
        }
}

Piece Piece::make_Z(int x, int y, int rotation)
{
        setup_Z();
        return Piece(rotation, x, y, 3);
}

void Piece::setup_Z()
{
        if (!settings_[3].setup)
        {
                settings_[3] = Settings({ {
                        0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0,
                        0, 1, 1, 0, 0,
                        0, 0, 1, 1, 0,
                        0, 0, 0, 0, 0
                                } },
                                Color::make_from_bytes(255, 0, 0),
                                2, true, false);
        }
}

Piece Piece::make_L(int x, int y, int rotation)
{
        setup_L();
        return Piece(rotation, x, y, 4);
}

void Piece::setup_L()
{
        if (!settings_[4].setup)
        {
```

```
                settings_[4] = Settings({ {
                        0, 0, 0, 0, 0,
                        0, 0, 0, 1, 0,
                        0, 1, 1, 1, 0,
                        0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0
                                } },
                                Color::make_from_bytes(255, 127, 0),
                                4, false, true);
        }
}

Piece Piece::make_J(int x, int y, int rotation)
{
        setup_J();
        return Piece(rotation, x, y, 5);
}

void Piece::setup_J()
{
        if (!settings_[5].setup)
        {
                settings_[5] = Settings({ {
                        0, 0, 0, 0, 0,
                        0, 1, 0, 0, 0,
                        0, 1, 1, 1, 0,
                        0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0
                                } },
                                Color::make_from_bytes(0, 0, 255),
                                4, false, false);
        }
}

Piece Piece::make_T(int x, int y, int rotation)
{
        setup_T();
        return Piece(rotation, x, y, 6);
}

void Piece::setup_T()
{
        if (!settings_[6].setup)
        {
                settings_[6] = Settings({ {
                        0, 0, 0, 0, 0,
                        0, 0, 1, 0, 0,
                        0, 1, 1, 1, 0,
                        0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0
                                } },
                                Color::make_from_bytes(255, 0, 255),
                                4, false, false);
        }
}

int Piece::get_rotation() const
{
        return rotation_;
}
```

```
int Piece::get_max_rotations() const
{
	return settings_[type_].max_rotations;
}

void Piece::set(int x, int y, int rotation)
{
	x_ = x;
	y_ = y;
	rotation_ = rotation;
}
```

3.3.11 PlayField.cpp

```
#include "PlayField.h"

PlayField::PlayField(int w, int h)
	:w_(w)
	,h_(h)
	,tiles_(h * w, false)
	, cleared_rows_(0)
{

}

void PlayField::set(int x, int y, bool occupied)
{
	int index = index_of(x, y);
	if (index != -1)
	{
		if (occupied == true)
		{
			occupied = true;
		}
		if (tiles_[index] != occupied)
		{
			tiles_[index].flip();
		}
	}
}

bool PlayField::get(int x, int y) const
{
	int index = index_of(x, y);
	if (index != -1)
	{
		return tiles_[index];
	}
	return false;
}

bool PlayField::test_collision(const Piece& piece) const
{
	auto tiles = piece.get_tiles();
	for (int i = 0; i < PIECE_SIZE; ++i)
	{
		for (int j = 0; j < PIECE_SIZE; ++j)
		{
			int x = piece.get_x() + i - 2;
			int y = piece.get_y() + j - 2;
			if (tiles[j * PIECE_SIZE + i] != 0)
			{
```

```
                                if (y >= h_)
                                {
                                        return true;
                                }

        else if (x >= 0 && x < w_ && y >= 0 && y < h_ && tiles_[y * w_ + x])
                                {
                                        return true;
                                }
                                else if (!(x >= 0 && x < w_))
                                {
                                        return true;
                                }
                        }
                }
        }
        return false;
}

bool PlayField::imprint(const Piece& piece)
{
        auto tiles = piece.get_tiles();
        for (int i = 0; i < PIECE_SIZE; ++i)
        {
                for (int j = 0; j < PIECE_SIZE; ++j)
                {
                        int x = piece.get_x() + i - 2;
                        int y = piece.get_y() + j - 2;
                        if (x >= 0 && x < w_ && y >= -
1 && y < h_ && tiles[j * PIECE_SIZE + i] != 0)
                        {
                                if (y == -1)
                                {
                                        return false;
                                }
                                tiles_[y * w_ + x] = true;
                        }
                }
        }

        cleared_rows_ += clear_rows();
        return true;
}

/* Returns number of rows cleared. */
int PlayField::clear_rows()
{
        int cleared = 0;
        for (int row = 0; row < h_; ++row)
        {
                bool occ = true;
                for (int col = 0; col < w_; ++col)
                {
                        if (!tiles_[row * w_ + col])
                        {
                                occ = false;
                                break;
                        }
                }
                if (occ)
                {
```

```
                        ++cleared;
                        for (int col = 0; col < w_; ++col)
                        {
                                tiles_[row * w_ + col] = false;
                        }
                        for (int y = row; y >= 1; --y)
                        {
                                for (int x = 0; x < w_; ++x)
                                {
                                        tiles_[y* w_ + x] = tiles_[(y - 1) * w_ + x];
                                }
                        }
                }
        }
        return cleared;
}

int PlayField::index_of(int x, int y) const
{
        return y * w_ + x;
}
```

3.3.12 Solver.cpp

```
#include "Solver.h"
#include "EvaluationFunctions.h"

Solver::Solver(std::vector<std::pair<evaluation_function, double>> evaluations)
        :current_piece_count_(-1)
        , level_(0)
        , score_(0.f)
        , running_(true)
        , lines_cleared_this_level_(0)
{
        evaluations_ = std::move(evaluations);
}

Solver::Solver()
        :current_piece_count_(-1)
        , level_(0)
        , score_(0.f)
        , running_(true)
        , lines_cleared_this_level_(0)
{
        evaluations_.emplace_back(EvaluationFunction<0>(), 1.0f);
        evaluations_.emplace_back(EvaluationFunction<1>(), 1.0f);
        evaluations_.emplace_back(EvaluationFunction<2>(), 1.0f);
        evaluations_.emplace_back(EvaluationFunction<3>(), 1.0f);
        evaluations_.emplace_back(EvaluationFunction<4>(), 1.0f);
        evaluations_.emplace_back(EvaluationFunction<5>(), 1.0f);
        evaluations_.emplace_back(EvaluationFunction<6>(), 1.0f);
}

void Solver::update(Board& board)
{
        if (running_ == false)
        {
                return;
        }

        if (board.get_piece_count() != current_piece_count_)
        {
```

```
                auto& current_piece = board.get_current_piece();
                if (!board.test_collision(current_piece))
                {
                        current_piece_count_ = board.get_piece_count();
                        start_search(board);
                }
        }

        play_recorded_actions(board);
}

bool Solver::is_done() const
{
        return !running_;
}

double Solver::get_score() const
{
        return score_;
}

void Solver::play_recorded_actions(Board& board)
{
        if (action_recording_.size() == 0)
        {
                handle_tick(board.tick());
                return;
        }
        else
        {
                auto& top_frame = action_recording_.front();
                if (top_frame.size() == 0)
                {
                        handle_tick(board.tick());
                        action_recording_.pop_front();
                }
                else
                {
                        board.perform_action(top_frame.front());
                        top_frame.pop_front();
                }
        }
}

void Solver::start_search(Board& board)
{
        PlayField play_field = board.create_play_field();

        std::vector<Piece> piece_queue;
        piece_queue.push_back(board.get_current_piece());
        for (int i = 0; i < 1; ++i)
        {
                piece_queue.push_back(board.get_next_piece(i));
        }
        StateArray states = build_states(play_field, piece_queue);
        auto& start_piece = board.get_current_piece();
        auto start = states[start_piece.get_x()][start_piece.get_y()][start_piece.get_rotation
()][0];
        auto best = search(states, play_field, play_field, 0, piece_queue, std::vector<Piece>(
));
        action_recording_ = std::move(make_recording(best, start));
```

```
}

State::ptr Solver::search(StateArray& states, PlayField& original_play_field, PlayField& play_
field, int depth, const std::vector<Piece>& piece_queue, std::vector<Piece>& locked_pieces)
{
        if (depth == piece_queue.size())
        {
                return nullptr;
        }

        StateQueue queue;
        const Piece& current = piece_queue[depth];

        queue.enqueue(states[current.get_x()][current.get_y()][current.get_rotation()][depth])
;
        State::ptr best_state;
        double best_state_value;

        while (!queue.is_empty())
        {
                auto state = queue.dequeue();
                const Piece& current = state->piece;

                Piece move_left = current;
                Piece move_right = current;
                Piece rotate_right = current;
                Piece move_down = current;
                move_left.move(-1, 0);
                move_right.move(1, 0);

                rotate_right.rotate_right();
                move_down.move(0, 1);

                if (current.get_max_rotations() != 1)
                {

        add_state_to_queue(states, queue, state, play_field, rotate_right, depth);
                }
                add_state_to_queue(states, queue, state, play_field, move_left, depth);
                add_state_to_queue(states, queue, state, play_field, move_right, depth);

                if (!add_state_to_queue(states, queue, state, play_field, move_down, depth))
                {
                        PlayField next_play_field = play_field;
                        next_play_field.imprint(current);
                        locked_pieces.push_back(current);

        State::ptr next_search = search(states, original_play_field, next_play_field, depth +
1, piece_queue, locked_pieces);

                        if (next_search == nullptr)
                        {


        double eval = evaluate_play_field(original_play_field, next_play_field, locked_pieces)
;
                                if (best_state == nullptr || best_state_value > eval)
                                {

        best_state = states[current.get_x()][current.get_y()][current.get_rotation()][depth];
                                        best_state_value = eval;
```

```
                                }
                        }
                        locked_pieces.pop_back();
                }
        }
        return best_state;
}

Solver::StateArray Solver::build_states(const PlayField& play_field, const std::vector<Piece>&
 piece_queue) const
{
        int w = play_field.get_width();
        int h = play_field.get_height();
        int d = 4;
        auto ret = StateArray(w, h, d, piece_queue.size());

        for (int z = 0; z < d; ++z)
        {
                for (int y = 0; y < h; ++y)
                {
                        for (int x = 0; x < w; ++x)
                        {
                                for (int s = 0; s < piece_queue.size(); ++s)
                                {
                                        ret[x][y][z][s] = std::make_shared<State>();
                                        ret[x][y][z][s]->visited = false;
                                        ret[x][y][z][s]->piece = piece_queue[s];
                                        ret[x][y][z][s]->piece.set(x, y, z);
                                }
                        }
                }
        }
        return std::move(ret);
}

bool Solver::add_state_to_queue(StateArray& states, StateQueue& queue, State::ptr prev_state,
const PlayField& play_field, const Piece& piece, int depth)
{
        int x = piece.get_x();
        int y = piece.get_y();
        int z = piece.get_rotation();

        if (play_field.test_collision(piece))
        {
                return false;
        }

        auto& state = states[x][y][z][depth];
        if (x == prev_state->piece.get_x() &&
                y == prev_state->piece.get_y() &&
                z == prev_state->piece.get_rotation())
        {
                return true;
        }
        if (state->visited)
        {
                return true;
        }

        state->visited = true;
        state->predecessor = prev_state;
```

```
        queue.enqueue(state);
        return true;
}

double Solver::evaluate_play_field(const PlayField& from, const PlayField& to, const std::vect
or<Piece>& piece_queue) const
{
        double total = 0;
        for (auto& p: evaluations_)
        {
                total += p.first(from, to, piece_queue) * p.second;
        }
        return total;
}

void Solver::handle_tick(int tick)
{
        if (tick == -1)
        {
                running_ = false;
        }

        double score_multiplier = 0.f;
        if (tick == 1)
        {
                score_multiplier = 0.1f;
        }
        else if (tick == 2)
        {
                score_multiplier = 0.3f;
        }
        else if (tick == 3)
        {
                score_multiplier = 0.6f;
        }
        else if (tick >= 4)
        {
                score_multiplier = 1.0f;
        }
        lines_cleared_this_level_ += tick;
        if (lines_cleared_this_level_ >= 10)
        {
                level_++;
                lines_cleared_this_level_ -= 10;
        }
        score_ += score_multiplier * (level_ + 1);
}

Solver::Recording Solver::make_recording(State::ptr state, State::ptr start) const
{
        Recording ret;

        State::ptr current = state;
        while (current != nullptr && !current->predecessor.expired() && start != current)
        {
                ret.emplace_front();
                State::ptr prev = current->predecessor.lock();
                auto current_piece = current->piece;
                auto prev_piece = prev->piece;

                        //get in rotation
```

```
                while (current_piece.get_rotation() != prev_piece.get_rotation())
                {
                        current_piece.rotate_left();
                        ret.front().emplace_front(Board::Action::Rotate);
                }

                //move to the right x
                while (current_piece.get_x() < prev_piece.get_x())
                {
                        current_piece.move(1, 0);
                        ret.front().emplace_front(Board::Action::Left);
                }
                while (current_piece.get_x() > prev_piece.get_x())
                {
                        current_piece.move(-1, 0);
                        ret.front().emplace_front(Board::Action::Right);
                }

                current = prev;
        }

        return std::move(ret);
}
```

3.3.13 SolverEvolver.cpp

```
#include "SolverEvolver.h"
#include "EvaluationFunctions.h"
#include "picojson.h"
#include <fstream>

SolverEvolver::SolverEvolver(int population_size, double combination_probability, double mutat
ion_probability)
        :population_size_(population_size)
        , combination_p_(combination_probability)
        , mutation_p_(mutation_probability)
        , steps_(0)
{
        std::ifstream file("population.json");
        picojson::value v;
        std::string err;
        err = picojson::parse(v, file);
        if (err.empty())
        {
                auto arr = v.get("population").get<picojson::array>();
                for (int i = 0; i < population_size && i < arr.size(); ++i)
                {
                        auto genome = arr[i].get<picojson::array>();
                        population_.push_back(Individual());
                        if (genome.size() == population_[i].genome.size())
                        {
                                for (int j = 0; j < population_[i].genome.size(); ++j)
                                {
                                        population_[i].genome[j] = genome[j].get<double>();
                                }
                        }
                        else
                        {
                                population_[i].genome = random_genome();
                        }
                }
        }
```

```
        for (int i = population_.size(); i < population_size; ++i)
        {
                population_.push_back(Individual());
                population_[i].genome = random_genome();
        }

        initate_population();
}

SolverEvolver::Genome SolverEvolver::random_genome()
{
        Genome ret;
        for (int i = 0; i < ret.size(); ++i)
        {
                ret[i] = random_double(0.0f);
        }
        return std::move(ret);
}

double SolverEvolver::random_double(double middle)
{
        return rand_.NextDouble(middle - 10.f, middle + 10.f);
}

void SolverEvolver::update(Window& window)
{
        bool no_update = true;
        for (auto& p : population_)
        {
                if (!p.solver->is_done())
                {
                        no_update = false;
                        p.solver->update(*p.board);
                        p.board->render(window);
                        break;
                }
        }
        if (steps_ % 50 == 0)
                //window.PrintScreen("screen_" + std::to_string(steps_) + ".bmp");
        ++steps_;
        if (no_update)
        {
                evolve();
        }
}

void SolverEvolver::evolve()
{
        sort_by_fitness();

        std::vector<Individual> child_population;
        while (child_population.size() < population_size_)
        {
                Individual& c0 = select();
                Individual& c1 = select();

                if (rand_.NextBool(combination_p_))
                {
                        crossover(c0.genome, c1.genome);
                }
```

```
                mutate(c0.genome);
                mutate(c1.genome);

                child_population.push_back(c0);
                child_population.push_back(c1);
        }
        population_ = child_population;
        initate_population();
}

SolverEvolver::Individual& SolverEvolver::select()
{
        //tournament selection
        auto rind = [this]() { return rand_.NextInt(0, population_.size() - 1); };

        int best = rind();

        for (int i = 0; i < population_.size() - 1; ++i)
        {
                int curr = rind();
                if (population_[curr].solver->get_score() > population_[best].solver-
>get_score())
                {
                        best = curr;
                }
        }
        return population_[best];
}

void SolverEvolver::sort_by_fitness()
{
        std::sort(population_.begin(), population_.end(), [&](const Individual& g0, const Indi
vidual& g1) {return g0.solver->get_score() > g1.solver->get_score(); });
}

void SolverEvolver::crossover(SolverEvolver::Genome& t0, SolverEvolver::Genome& t1)
{
        Genome g0 = t0;
        Genome g1 = t1;
        size_t len = g0.size();

        for (size_t i = 0; i < len; i++)
        {
                t0[i] = rand_.NextInt(0, 1) == 1 ? g0[i] : g1[i];
                t1[i] = rand_.NextInt(0, 1) == 1 ? g0[i] : g1[i];
        }
        t0 = g0;
        t1 = g1;
}

void SolverEvolver::mutate(SolverEvolver::Genome& t)
{
        for (size_t i = 0; i < t.size(); i++)
        {
                if (rand_.NextBool(mutation_p_))
                {
                        t[i] = random_double(t[i]);
                }
        }
}
```

```
void SolverEvolver::initate_population()
{
	int x = 0;
	int y = 0;
	for (auto& c : population_)
	{
		std::vector<std::pair<Solver::evaluation_function, double>> evaluations;
		evaluations.emplace_back(get_evaluation<0>(c.genome));
		evaluations.emplace_back(get_evaluation<1>(c.genome));
		evaluations.emplace_back(get_evaluation<2>(c.genome));
		evaluations.emplace_back(get_evaluation<3>(c.genome));
		evaluations.emplace_back(get_evaluation<4>(c.genome));
		evaluations.emplace_back(get_evaluation<5>(c.genome));
		evaluations.emplace_back(get_evaluation<6>(c.genome));
		c.solver = std::move(std::make_unique<Solver>(evaluations));
		c.board = std::move(std::make_unique<Board>(x, y, 16));
		x += 16 * 12;
		if (x > 800)
		{
			y += 16 * 22;
			x = 0;
		}
	}

	picojson::object obj;
	picojson::array arr;
	for (auto& c : population_)
	{
		picojson::array genome;
		for (int i = 0; i < c.genome.size(); ++i)
		{
			genome.push_back((picojson::value)c.genome[i]);
		}
		arr.push_back((picojson::value)genome);
	}
	obj.insert(std::make_pair("population", (picojson::value)arr));

	picojson::value val = (picojson::value)obj;
	std::string str = val.serialize();
	std::ofstream file("population.json");
	file << str;
}
```

3.3.14 main.cpp

```
#include <SDL.h>
#include "Window.h"
#include "Board.h"
#include "Timer.h"
#include "SolverEvolver.h"

int handle_input(Board&, Window&);

void render_piece(Piece& piece, Window& window)
{
	Color color = piece.get_color();
	auto tiles = piece.get_tiles();
	for (int i = 0; i < PIECE_SIZE; ++i)
	{
		for (int j = 0; j < PIECE_SIZE; ++j)
		{
			int x = piece.get_x() + i;
```

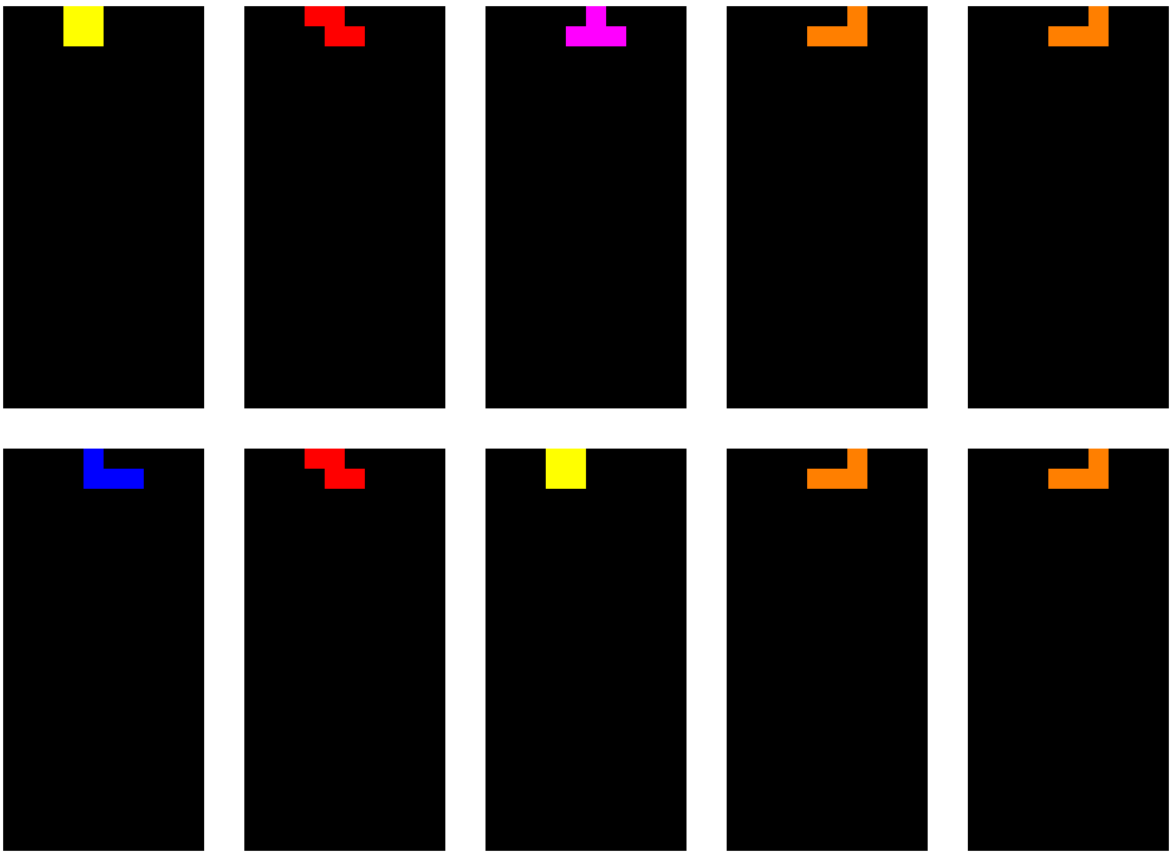
```
                        int y = piece.get_y() + j;
                        if (tiles[j * PIECE_SIZE + i] != 0)
                        window.RenderRectangle(x * 16, y * 16, 16, 16, color);
                }
        }
}

int main(int argc, char *argv[])
{
        Window win("Tetris!", 5 * 16 * 12, 2 * 16 * 22);
        SolverEvolver solver(10, 0.5f, 0.05f);

        float rot = 0.0f;

        Timer timer;
        timer.Start();
        int tick_time = 1000;

        while (win.Open())
        {
                win.PollEvents();
                win.Clear();

                solver.update(win);
                win.Display();

        }
        return 0;
}
```

3.4 Körexempel

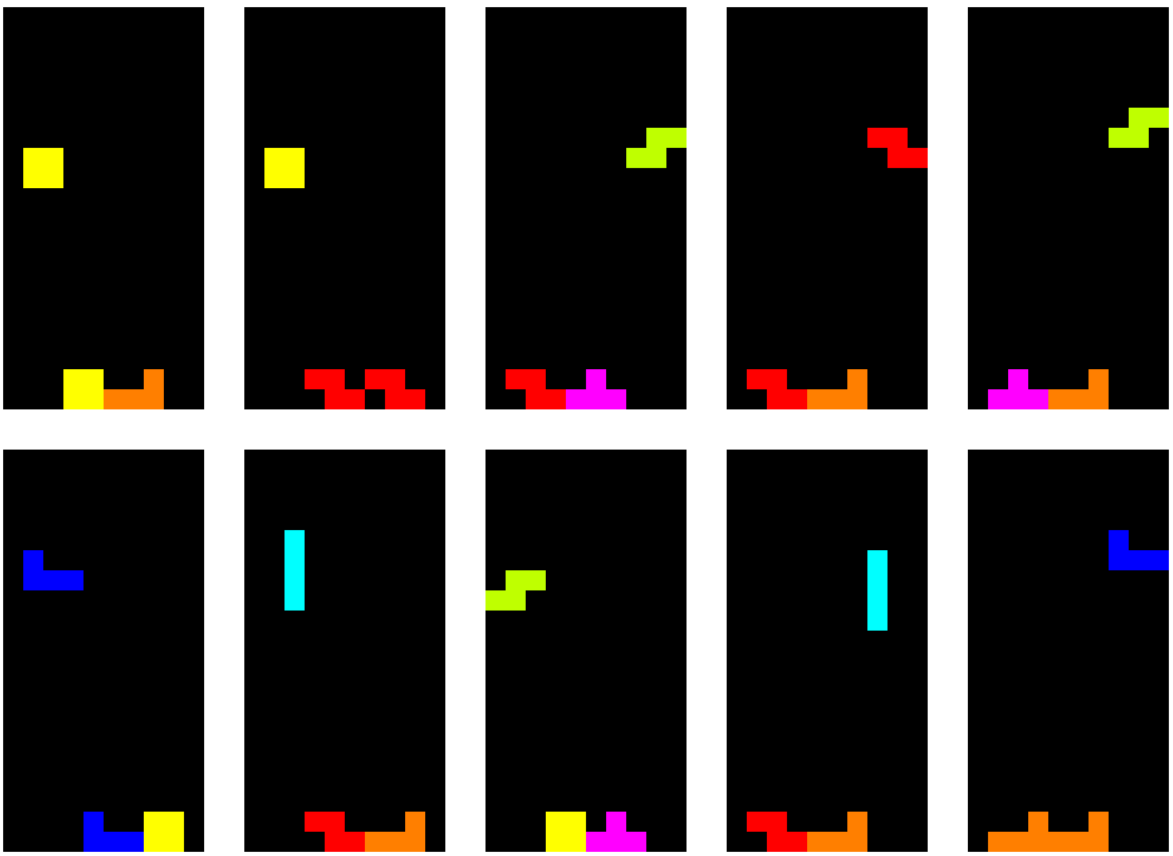
Körexempel börjar med en AI som har tränats i flera timmar redan, där den individ med högst fitness har vikterna:

1. Totala antalet rensade rader: 128,15456933807582
2. Totala låshöjden: 26.605806578882039
3. Totala antalet väggceller: 3.988116541877389
4. Totala antalet kolonhåll: 109.55691834446043
5. Totala antalet kolonövergångar: -33.212596164084971
6. Totala antalet radövergångar: 27.446223678998649
7. Stackhöjd:-6.1861635837703943

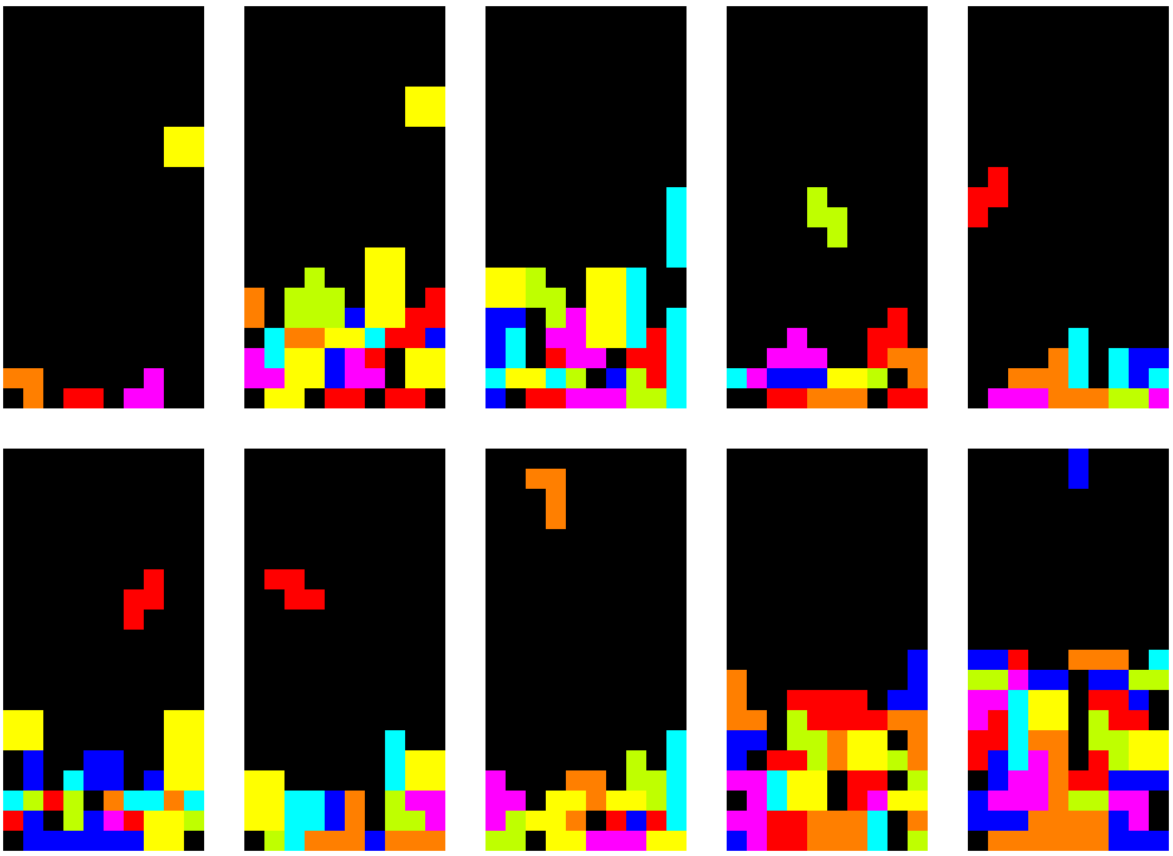
Ett urval av skärmdumpar från en omgång där alla individer syns I fönstret samtidigt (individen med högst fitness visas längs upp till höger) visas nedan.



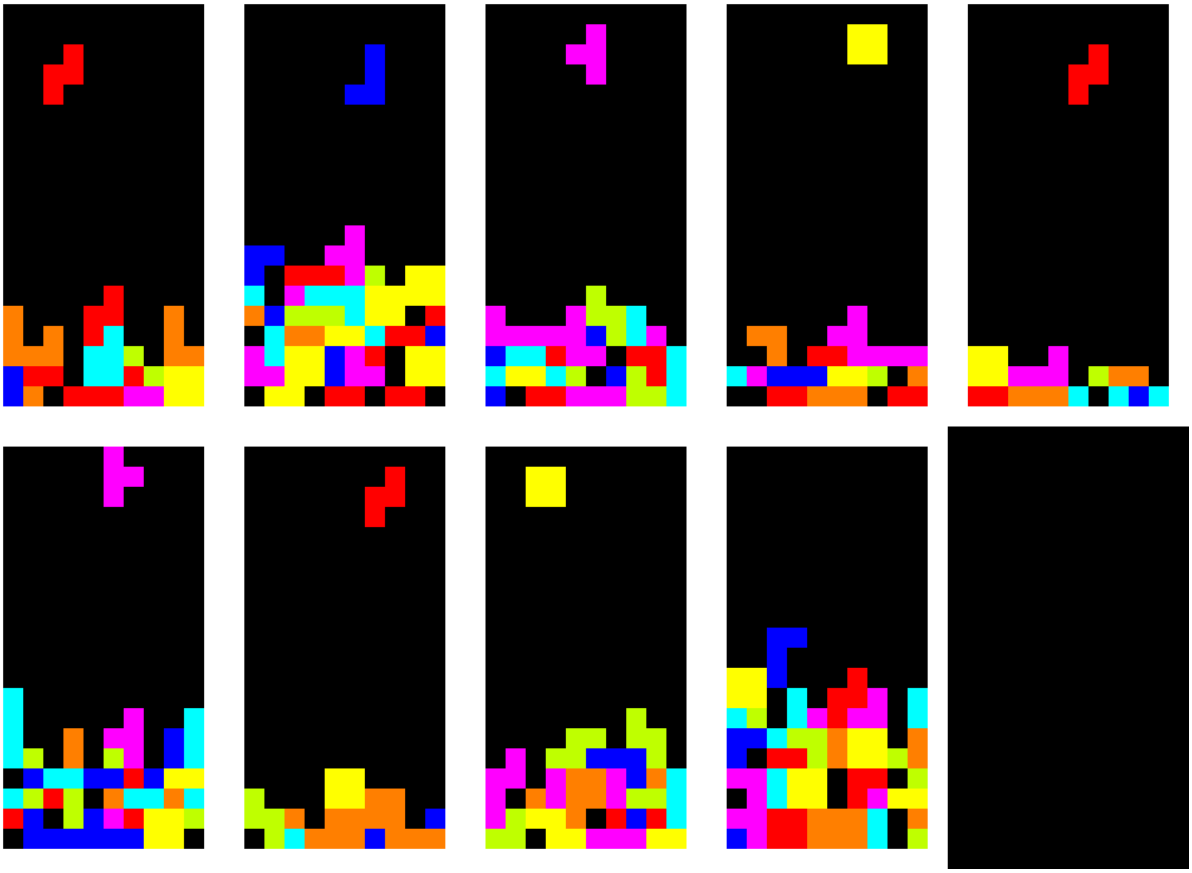
Figur 20.



Figur 21.



Figur 22.



Figur 23. Ett skede där en av individerna har förlorat.

Det syns inte tydligt att AI:n lyckas rensa rader i körexemplet, det är svårt att visa det i stillbilder, men AI:n rensar faktiskt rader under körexempel.

3.5 Analys

Att använda en evolutionär algoritm för att bestämma vikterna för evalueringsfunktionen ger fördelen att dessa vikter inte behöver skrivas in manuellt, något som hade varit svårt att göra då det borde vara svårt, om inte omöjligt, att veta vilka vikter som vore optimala. Det gör också att det går att skriva godtyckliga evalueringsfunktioner eftersom att algoritmen kan sortera bort mindre bra funktioner själv.

En fråga värt att ställa är hur bra AI:n blir. Designen håller inte reda på poängen som AI:n får, men det hade varit interessant att se hur höga poäng AI:n får. En annan fråga är om den evolutionära strategin fungerar. Vid körning märks det väldigt tydligt att AI:n blir bättre för varje eon som går.

Ett problem är att eftersom att AI:n måste spela färdigt för att kunna bli bättre så kommer det finnas en gräns då det tar längre och länger tid för AI:n att bli bättre. En lösning på det problemet är att ge AI:n en sekvens av block som är utvecklad för att vara svår att (eller t.o.m. omöjlig) att spela en längre tid. Ett exempel på en sådan sekvens är att varva Z-block och S-block, en sådan sekvens kommer garanterat att leda till förlust. På detta sätt tvingas AI:n att bli klar med spelet, oavsett hur bra AI:n är. För att förhindra att AI:n blir speciellt bra på en viss sekvens, men dålig på ett vanligt spel, borde AI:n tränas med olika sådan sekvenser.

Även efter att ha tränat AI:n ett antal timmar verkar den inte ha ”lärt sig” mer avancerade tekniker (ex. T-spin, som kan användas för att täppa igen svårtåtkomliga hål), även om den borde kunna utföra sådana manövrar. Det är svårt att veta vad det beror på, om det är tetris-simuleringen som AI:n spelar, hur AI:n kommer fram till alla möjliga tillstånd eller om det beror på att fitness-funktionerna som finns inte är tillräckliga för detta. Det kan även bero på att AI:n inte har tränats tillräckligt länge. Det skulle kunna gå att uppmuntra till sådana manövrar genom att analysera AI:ns drag och belöna sådana manövrar, men det vore svårt skriva en analys för att upptäcka dess.

Tyvärr har det funnits begränsat med tid att träna AI:n och det hade varit interessant att se hur bra AI:n hade kunnat bli efter ännu längre tid, eller om den alls hade blivit bättre. Det finns säkert flera sätt att utöka denna AI på också.

4 Slutsats

Jag tycker att det har varit intressant att få möjlighet att använda flera olika sorters AI för att lösa problem. Jag blev speciellt intresserad av Self-Organising maps och möjliga användningsområden för sådana.

Att låta en AI utveckla kod tyckte jag också var intressant, och jag undrar vilka andra problem som hade kunnat lösas med en sådan metod. Förslagsvis problem där en sådan metod hade varit en fördel. Man kan tänka sig en motståndare i ett spel som utvecklas under spel med spelaren på ett liknande sätt.

Att låta en AI spela Tetris var också givande och har fått mig att fundera på hur andra pusselspel kan lösas med samma principer. Jag har också funderat på på vilket sätt lösningen kan förbättras, exempelvis genom att göra AI:n övertygande mänsklig.