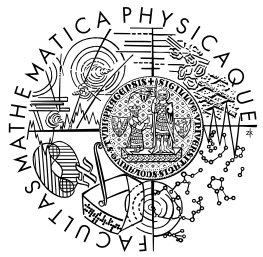

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Pavel Mikuš

Pattern recognition for in-game spell systems

Department of Software Engineering

Supervisor of the bachelor thesis:	Miroslav Kratochvíl, M.Sc.
Study programme:	Computer Science
Study branch:	Programming and Software systems

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Pattern recognition for in-game spell systems

Author: Pavel Mikuš

Department: Department of Software Engineering

Supervisor: Miroslav Kratochvíl, M.Sc., Department of Software Engineering

Abstract: Magic is a popular element in current computer games. Although most games spoil the sensation of magic as of something extraordinarily subtle by allowing the player to cast spells by simply hitting key combinations, several games require the player to finish a more complicated action before casting a spell: Drawing a complicated glyph that represents the spell is one of such actions. This thesis aims to provide a repurposable library that would allow simple implementation of structured glyph-drawing-based in-game spell systems. The thesis studies several relevant approaches to pattern recognition, describes a neural-network based method for recognition of various shapes and shape combinations, develops a system for describing the parameters and results of the used algorithm in terms of predefined spell shapes and their recognized combinations, and implements this approach in a library and an accompanying simple demonstrational game. The library and its parameters are benchmarked and systematically optimized.

Keywords: spell systems pattern recognition games neural networks shape combination

I would like to express my gratitude to Mgr. Miroslav Kratochvíl, the supervisor of this thesis, for his patient guidance, feedback and all the advice he has provided. I also want to thank my girlfriend for her support and encouragement during the time spent working on this thesis.

Contents

1. Introduction

Magic has always been a popular part of computer games. In many games, magic has its own lore and laws that make it systematical. Great examples of the complexity of magical spell systems include the games *Magicka* and *Magicka 2*, where the player casts spells by combining eight elements (as seen on fig. 1.1): For example, using only earth element results in a rock thrown at the enemy, but adding fire will create a classic fireball.

In books and movies, wizards can be seen performing complicated hand gestures or drawing complex shapes in order to cast spells. In games, on the other hand, casting magic is usually extremely easy, since players may often just push a few buttons to cast even the strongest spells. This spoils the feeling of magic as something extraordinary and secret, requiring years of study. Complicated magic systems allow the game developers to engage players in a different game experience, requiring focus, some amount of creative thinking, and possibly cooperation with other players.

In this thesis, we focus on the constructive kind of spell casting: While the majority of games simply binds spells to buttons, there are several that use some kind of different system. One class of such systems uses pattern recognition algorithms to translate the player's image-like (or, say, rune-like) input to a spell definition. However, as shown in the next section, these systems usually recognize only simple gestures or patterns, which is partly caused by the difficulties in implementing such a system[1]. The goal of this thesis is to create a comprehensive pattern recognition system that can be easily plugged into games to allow simple and robust recognition of player-drawn shapes and their translation to an arbitrary spell system.

[1] In this thesis we ignore other difficulties, which mainly include various marketing and accessibility issues.

Figure 1.1: *Magicka* spell interface. Image taken from Mag.

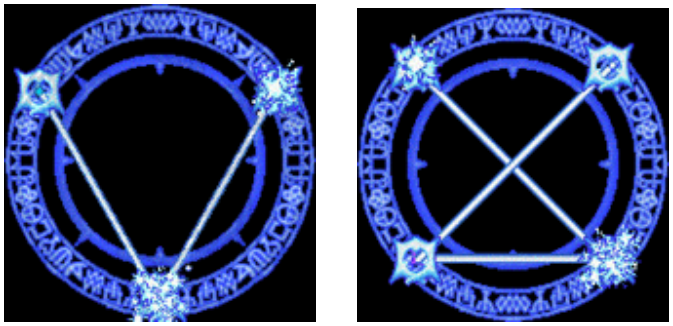

Figure 1.2: *Castlevania: Dawn of Sorrow* spell interface. Images taken from Cas.

1.1 Pattern recognition in current games

Symbols and gestures are usually an integral part of the magic and there were many attempts to bring them into video game environment. Howard [How14] provides a classification of existing gestural systems into three categories (which sometimes overlap and combination of these approaches is used):

Alternative controllers One such category are systems that utilize alternative controllers to mouse and keyboard, such as Wiimote or Kinect. A recent game from this category is *Fable: The Journey*, where the player casts spells by moving his hands: For example, push spell is cast by pushing into the air. Patterns made by the player are recognized using Kinect technology. While moves are simple in nature, such as waving sideways or back and forth, both hands at the same time can be used, which results in quite complex gestures.

Restricted drawing forms Another technique used in several games is to let player draw into a predefined grid, or through predefined points. Both *Castlevania: Dawn of Sorrow* and *Deep Labyrinth* take advantage of an Nintendo DS drawing interface that allows players to draw magic signs. In *Castlevania*, players draw signs by connecting glowing points in a circle in out-of-combat situations, as seen in fig. 1.2. *Deep Labyrinth* introduces special casting interface that consists of 3×3-grid, where the player connects dots.

The restriction to pre-existing following grid takes away a part of the creative freedom, but makes recognition algorithms much easier to implement and run: For example, only tracing the order of the connected points in *Deep Labyrinth* is sufficient to recognize the spell.

Figure 1.3: *Black & White* teleport and fireball gestures. Images taken from Ban.

Free-hand drawing One of the first games to integrate some form of pattern recognition of drawn shapes is *Black & White*. Using a mouse, players are able to cast miracles by drawing a specific pattern on the ground, as in fig. 1.3. The player can draw anything anywhere on the ground, and its up to the game logic to recognize if the result matches some of the miracle patterns. Alternatively, the player can still cast miracle by clicking on a button — presumably because a lot of players had trouble drawing the miracles.

A similar approach to recognition of player-drawn spells is used in *Arx Fatalis*. Players are drawing symbols into the air with a mouse, and a sequence of the drawn symbols represents some spell. While casting, game encodes the mouse moves into characters that represent the 8 directions. After player finishes the spell, a Levenshtein distance is calculated from each predefined spell sequence to the player's created sequence and the candidate with the lowest Levenshtein cost is returned as a matching spell.

1.2 Goals

The result of the thesis allows the players to draw complex spells, preferably more complex than in *Black&White* or *Arx Fatalis*. For that purpose, we create an algorithm that recognizes a set of basic symbols which can be arbitrarily combined to form new, very complex symbols.

Possibilities of such combinations include composition (symbol is made from other smaller symbols) and embedding (a symbol is put inside another symbol, or into a somehow important position relative to other symbol, e.g. on the top of a triangle). Several examples of symbols and their combinations is shown in fig. 1.4.

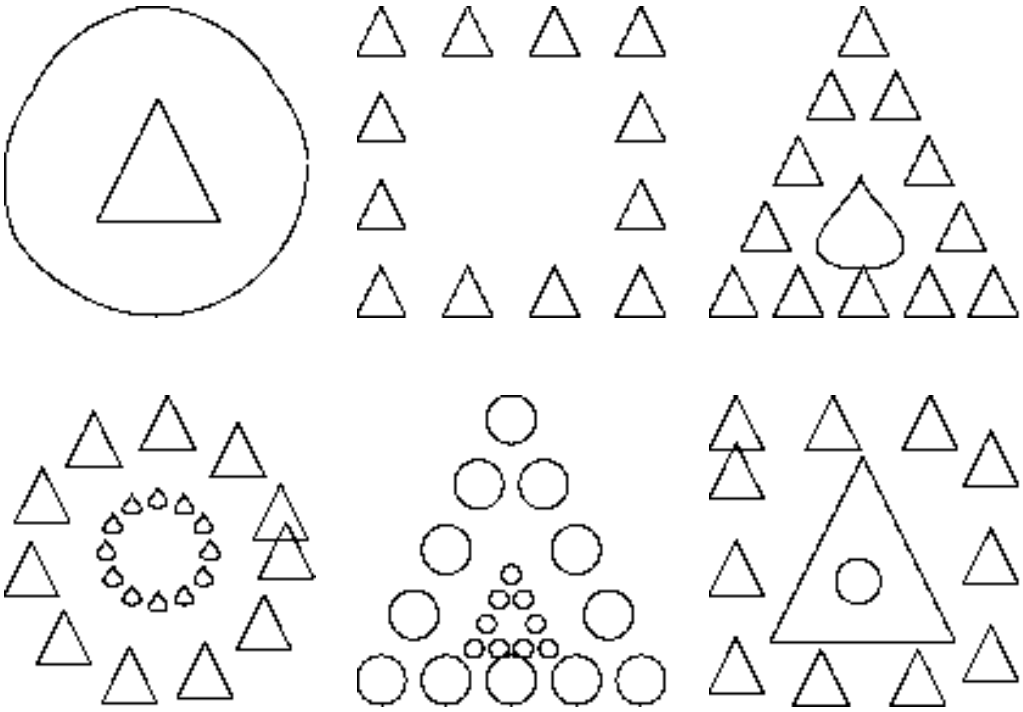

Figure 1.4: In the figure, embedded positions are always in the middle of the shape. From left to right: circle with embedded triangle; square composed of triangles; triangle composed of triangles, with water drop embedded; circle composed of triangles, with embedded circle composed of water drops; triangle composed of circles, with embedded triangle composed of circles; square composed of triangles, with embedded triangle, with embedded circle

We refrain from using any aiding structures (as e.g. in *Castlevania*) to retain generality of the system.

For the purpose of evaluation the work, we specify the following requirements on the resulting recognition system:

Durability against shape deformations Since we want to recognize hand-drawn shapes, our system has to be prepared for human-like imprecise drawing, especially when drawing with a mouse. However, drawing an exact line between a shape that should still be recognized and a shape that should be rejected is difficult and possibly subjective; we therefore only evaluate the performance on deformed shapes.

Extensibility We would like to offer an easy-to-use library that can be used in other projects. For this purpose, we need to let the users define their own shapes that should be recognized. The library should provide interfaces for both preparing custom shapes and using them in the pattern recognition.

Performance Our system should be applicable in demanding video games environment, prepared for the possibility of many players drawing their spells at once. To achieve this, we require fast, ideally parallelizable recognition technique.

Recognition of embedded shapes To allow players cast complex spells, we need to give them an ability to somehow encode multiple symbols in one spell. One possible way to do it are shape embeddings. Each defined shape can contain several areas, where other shapes might occur. These areas are then processed in our recognition system and classified.

Recognition of shape conglomerations For the purpose of this work, we consider shape conglomeration a group of shapes from the same shape class e.g. circle, arranged such that the whole conglomeration forms another shape. We can also look at it as taking the curves of the shape, sampling them uniformly, and then replacing all the samples with the pattern shape.

1.2.1 Approach

The thesis is divided into several steps: First chapter reviews the approaches and algorithms commonly used to solve the pattern recognition problem. We have closely examined three algorithms, namely the *Normalized cross-correlation*,*Shape matching and object recognition using shape contexts* and

Matching of Shapes Using Dynamic Programming. In the same chapter, we describe *Artificial neural networks* and place them in the context of pattern recognition. In chapter 3, we describe the implementation of the resulting neural-network-based recognition algorithm. The implementation is accompanied by a simple game prototype that is used to demonstrate the properties of the recognition system. Finally, we perform benchmarking of the performance of our algorithm, with respect to recognition success rate and speed.

In the appendix, we describe the interface of our algorithm and provide a guide for using it in another project. A simple explanation of demonstrational game usage is also included.

2. Pattern recognition

In this chapter, we describe several classes of algorithms for pattern recognition. In the first part, we review some of the algorithms, namely the *Normalized cross-correlation*, *Shape matching and object recognition using shape contexts* and *Matching of Shapes Using Dynamic Programming*. The second section is devoted to the *Artificial neural networks* in the context of pattern recognition, which are later used for the implementation of thesis goal.

Pattern recognition problem is a broad term for classification problems that are based on the similarity of the features of classified objects. Precise mathematical definition can be found in the work of Nieddu and Patrizi [NP00]. We will focus more on the shape recognition problems category, which is a form of pattern recognition. A shape can be typically defined as an equivalence class under the group of transformations, and the problem is then to algorithmically approximate the human-like visual pattern recognition. We define an image as a two-dimensional grid of pixels with values from the range $[0, 1]$ and we define the shape as a group of image instances where a human sees the same shape.

2.1 Algorithms classification

There are many of algorithms on for pattern recognition that can be categorized based on different criteria. Dougherty [Dou12] categorize the algorithms into following approach-based classes:

Statistical approach Algorithms in this category are based on the underlaying probability model. The shape class is determined by the features that can be automatically extracted from its members and the probability distributions of the shape belonging to each class. An example of such algorithm is the naive Bayess classifier.

Nonmetric approach This class contains decision tress, syntactic methods, and rule-based classifiers. The idea of these algorithms is that each shape can be decomposed into the simplest sub-patterns called primitives. These primitives are then viewed as a language and the shape class as a set of rules, from which the shape can be derived. However, the inference of the grammar rules from the training data and the detection of the primitives are difficult problems.

Cognitive approach Neural networks and support vector machines belong to this class. Neural networks are inspired by biological neural struc-

tures. They can be described as massively parallel computation systems that can learn complex input-output relationships.

At the same time, [Dou12] remark:

> However, in spite of the seemingly different underlying principles, most of the neural network models are implicitly similar to statistical pattern recognition methods.

[BL08] proposes in the work different classification based on the creation process of the classifier. The algorithms are divided into learning-based approaches and template-based approaches. In the learning-based approaches, pattern classifiers are obtained through training on pre-classified samples. In the template based approaches, patterns are described by templates and the recognition problem is transformed into searching for the best matching template for a given input image.

Another classification proposed by Ghafoor, Naveed Iqbal, and Khan [GNIK03] divides the algorithms into three classes based on the level of preprocessing:

- Algorithms that use pixel values directly, e.g. correlation-based methods. Example from this category is the normalized cross-correlation, described in section 2.2.
- Algorithms that use low-level features such as edges and corners, e.g. distance transform method, described in Ghafoor, Naveed Iqbal, and Khan [GNIK03].
- Algorithms that use high-level features such as identified objects or relation between the features, e.g. graph-theoretic-methods [BA83].

Given the amount of algorithms to choose from, we have decided to review only a few of them, namely the normalized cross-correlation, shape contexts object matching and matching of shapes using dynamic programming, and the general approach of using the neural networks, which we use in our algorithm implementation.

2.2 Normalized cross-correlation

Following description is based on the work of [Lew01]. Cross-correlation is a method to measure the similarity between a template and a given area of an image, both in the form of a grid of pixel values. The term cross-correlation itself means a difference between two signals. It is one of the

oldest approaches to pattern and feature recognition and extraction, and still serves as a base for more complex algorithms.

It works by computing the distance between an image and the searched template. We can compute an Euclidean distance of a image f and a template t, where pattern is on a position $(u\,v)$, by comparing corresponding pixels of the image at $(x\,y)$ and the template shifted to $(x-u\,y-v)$.

$$d^2_{f,t}(u,v) = \sum_{x,y}[f(x,y) - t(x-u, y-v)]^2$$

When expanded, it gives us three terms. One of them is the cross correlation term $c(u,v)$.

$$\begin{aligned} d^2_{f,t}(u,v) &= \sum_{x,y} f(x,y)^2 - 2c(u,v) + \sum_{x,y} t(x-u, y-v)^2 \\ c(u,v) &= \sum_{x,y} f(x,y) * t(x-u, y-v). \end{aligned}$$

Other two terms express the energy (brightness) of the template and the image, respectively. We perform this computation for all possible values of u, v, looking for the lowest value. The we can then consider distances lower than a certain threshold to be a match, and the corresponding u, v terms give us the location.

Computing distance this way has several serious drawbacks. It is computationally demanding since we have to consider every position and every pixel appears in the computation many times, based on the size of the template. The method may also give false matches if the image energy changes a lot with the position. Also, the range of values of cross-correlation term depends on the size of the template and it is not invariant to scaling and rotation.

The performance of the method can be improved substantially if we use convolution. Convolution is an integral that expresses the amount of overlap of one function g as it is shifted over another function f, which is similar to the cross-correlation. For discrete real valued signals such as pixels of an image, they differ only in a time reversal in one of the signals. We can then apply the convolution theorem.

Convolution theorem [Wik18] states that when certain conditions are met, Fourier transform of a convolution is an element-wise product of Fourier transforms of input signals. From Convolution theorem follows that time domain or space domain convolution is equivalent to element-wise multiplication in the frequency domain. To compute convolution, we simply need to take element-wise multiplications of Fourier transforms of signals to be

convoluted. Cross-correlation differs in that we take the complex conjugate of the Fourier transform of the second signal.

Problems with the range of cross-correlation value and dependency on brightness can be fixed by using normalized cross-correlation, where the image and the template vectors are normalized to unit length. Other desired aspects, such as scale invariance, has been addressed in many algorithms using a cross-correlation method. However, they usually introduce some trade-offs and they may not achieve all the required properties together.

Normalized cross-correlation is defined as:

$$\gamma(u,v) = \frac{\sum_{x,y} f(x,y)[f(x,y) - \bar{f}_{u,v}][t(x-u,y-v) - \bar{t}]}{\{\sum_{x,y} f(x,y)[f(x,y) - \bar{f}_{u,v}]^2 \sum_{x,y}[t(x-u,y-v) - \bar{t}]^2\}^{0.5}}$$

where $\bar{t}$ is the mean of the feature pixel brightnesses and $\bar{f}_{u,v}$ is the mean of the values of $f(x,y)$ in the region under the feature [Lew01].

2.3 Shape matching and object recognition using shape contexts

Shape matching approaches that use shape contexts are usually based on the extracted features, which generally introduces a more robust recognition of deformed images. Shape context is usually a group of shape features and their relations and the shape class is then defined by their similarity. We review the algorithm by Belongie, Malik, and Puzicha [BMP02]. The approach attempts to find corresponding feature points of the image and the template, and then computes their distance as a sum of errors of corresponding points.

We treat an image as a possibly infinite set of points that for the shape and we assume that each shape is represented by a finite subset of its points. We extract from both image and template a certain number of feature points, 100 is recommended by the authors. These points do not need to be the key-points, such as maxima of curvature or corners. This allows us to use a simple extraction method like edge detection.

With feature points extracted, we need to find the corresponding points between the template and the tested image. To do so, Belongie, Malik, and Puzicha [BMP02] introduces a *shape context descriptor* 2.1, defined by the sample points and their relations. For each sample point p, we can create a set of vectors originating from p to all other feature points. Such set of vectors represents positions of other sample points relative to the origin point. The more sample points we choose, the more is this shape descriptor exact.

However, such descriptor might be too detailed and too sensitive to intraclass variations. In the paper, the authors presented a more robust and

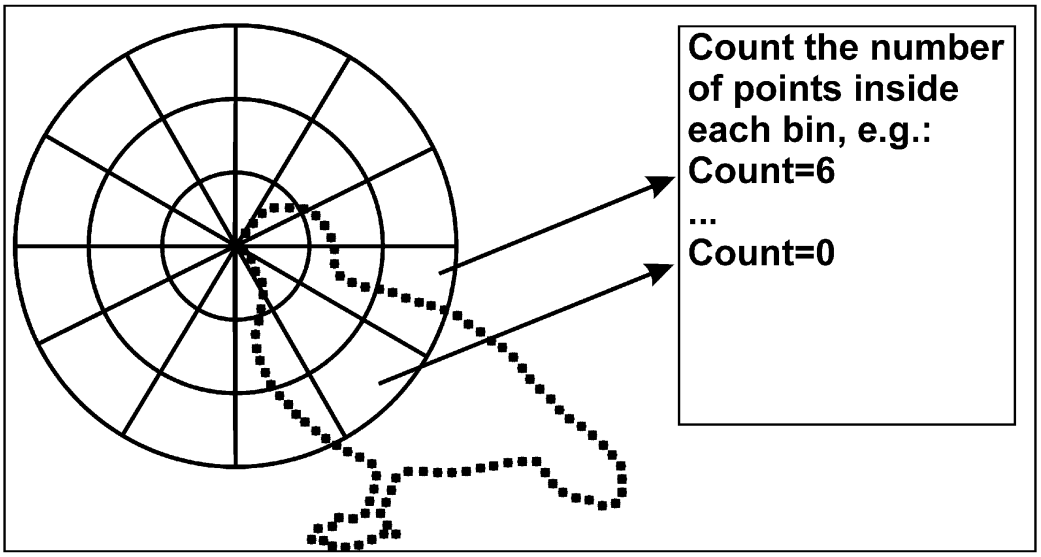

Figure 2.1: A scheme of representation of point sets using shape context. A shape is represented by a discrete set of points sampled regularly along the contours. For every point, a log-polar histogram—the shape context—is computed which approximates the distribution of adjacent point locations relative to the reference point. Image and description taken from Belongie, Malik, and Puzicha [BMP02].

compact shape descriptor. The idea is that for each point p, we compute coarse histogram by assigning other points to bins in polar coordinates with a center in p. Polar coordinates are more sensitive to points near p. We can then compute the cost of matching two points as

$$C_{ij} = C(p_i, q_j) = \frac{1}{2} \sum_{k=1}^{K} \frac{(h_i(k) - h_j(k))^2}{h_i(k) + h_j(k)},$$

where $h_i(k)$ and $h_j(k)$ represent a k-bin normalized histogram at p_i and q_j. Given a set of costs $C_{i,j}$ between all pairs of points p_i and q_j, we need to find the best alignment, which means that we want to minimize the total cost of matching

$$H(pi) = \sum_i C(p_i\, q_{pi(i)}).$$

This can be solved in $O(N^3)$ time using the Hungarian method[BMP02].

We can achieve scaling invariance by normalizing all radial distances by the mean distance, and rotation invariance is obtained when searching for the lowest cost among all permutations. According to the paper, this method is also robust against small geometric disturbances.

2.4 Matching of shapes using dynamic programming

A method proposed by Petrakis, Diplaros, and Milios [PDM02] introduces an algorithm which uses dynamic programming combined with high-level features extraction. Similarly to the previous algorithms, the method is based on computation of a distance between the template and the image, but in a different way.

The algorithm requires that the both shape and the template are represented as a sequence of convex and concave line segments, split by inflex points. The idea of the algorithm is to recursively merge segments using two grammar rules $CVC \rightarrow C$ and $VCV \rightarrow V$, where V denotes concave and C convex segment. Simultaneously, merging cost when the rules are applied is computed using a merging cost function, and the results are stored in the dynamic programming table. The merging cost represents a measure of dissimilarities between the merged segments of the shapes.

Rows and columns of the dynamic programming table represent the inflex points of the shape and the template, respectively. The cost $D(A, B)$ of matching shape A with shape B is defined as:

$$D(A, B) = min_T\{g(i_T, j_T)\},$$

where $\{g(i_T, j_T)\}$ is a cost of a complete match. The complete match is characterized by a complete path $((i_0, j_0), (i_1, j_1, ..., (i_T, j_T))$, i.e., a path that covers all segments of both shapes. In turn, $\{g(i_T, j_T)\}$ is defined as follows:

$$g(i_T, j_T) = min(i_w, j_w) \sum_{w=1}^{T} \phi(a(i_{w-1}|i_w), b(j_{w-1}|j_w))$$

Expression $\phi(a(i_{w-1}|i_w), b(j_{w-1}|j_w))$ represents the dissimilarity cost function defined as:

$$\begin{aligned}\phi(a(i_{w-1}|i_w), b(j_{w-1}|j_w)) = {} & \lambda\textsc{MergingCost}(a(i_{w-1}|i_w)) \\ & +\lambda\textsc{MergingCost}(b(j_{w-1}|j_w)) \\ & +\textsc{DissimilarityCost}(a(i_{w-1}|i_w), b(j_{w-1}|j_w))\end{aligned}$$

The first two terms in represent the cost of merging segments $a(i_{w-1}|i_w)$ in shape A and segments $b(j_{w-1}|j_w)$ in shape B, respectively, while the last term is the cost of associating the merged sequence $a(i_{w-1}|i_w)$ with the merged sequence $b(j_{w-1}|j_w)$. Each merging should be a recursive application of the grammar rules $CVC \rightarrow C$ and $VCV \rightarrow V$.

The lambda value controls merging tendency. With lower value, merging is more encouraged. For shapes with much detail, it is practical to set higher values, otherwise, these details may be lost during merging.

At the end of the computation, each field $F(i, j)$ of the dynamic table contains the minimal cost of merging the first $i - 1$ segments of the shape and $j - 1$ segments of the template. The lower the merging cost is, the more are the segments similar.

Since the algorithm assumes, that first segments of the shape and the template are aligned and match, we may need to run the algorithm for all possible starting points of shape if we don not know the first segment beforehand.

Differently to the previous algorithm, we now require the extraction of high-level features, we need to extract convex and concave segments in correct order. However, we obtain a matching algorithm that is independent of shape translation, scaling and rotation. We can also directly control the invariance to deformations using the lambda parameter.

2.5 Neural networks

Neural networks are mathematical models of computation inspired by a behavior of a biological nervous system [Bis96]. They have been successfully used to solve problems in many different areas, including image and pattern recognition. They can be used to solve problems, where we can not easily mathematically describe the solution given the instance of the problem, for reasons of either complexity or lack of existing mathematical model.

Neural networks have several advantages over the previous methods. They do not require feature extraction since they are able to learn it, so we can use pixel values directly. They also have a great ability to generalize, which can be used for the recognition of the shape compositions and embeddings. On the other hand, they have to go through a learning process before they can be used.

2.5.1 Neuron

Artificial neural networks are structures based on the parallel computational model of the biological neural systems. The basic network unit is a neuron (see section 2.5.1), which is characterized by its input and output connection weights, the activation function and a bias. The artificial neural network is built from a number of connected neurons, usually in a layered structure,

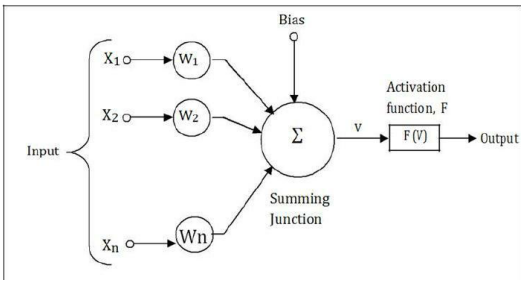

Figure 2.2: The model of a neuron unit. Image taken from Neu

and the network learning can be characterized as a process of altering the weights of the connections between neurons and the biases of the neurons.

Activation function characterizes the behavior of the neuron. When the values from the input connections are available, the activation function is applied onto the sum of the values and the result is passed further to other neurons. Two common examples of activation functions are the step function and the sigmoid function:

- $\textsc{Step}(x) = \begin{cases} 1 & (x \geq 0) \\ 0 & (x < 0) \end{cases}$
- $\textsc{Sigmoid}(x) = \frac{1}{1+e^{-x}}$

2.5.2 Artificial neural networks

An artificial neural network is a mathematical model, consisting of a set of neurons interconnected by connections. More exactly it is a set $M = (N, C, I, O, w, t)$, where:

- N is a finite set of neurons.
- $C \subset N \times N$ is nonempty set of oriented connections.
- $I \subset N$ is nonempty set of input neurons.
- $O \subset N$ is nonempty set of output neurons.
- $w : C \to R$ is weight function.
- $t : N \to R$ bias function.

In practice, multilayer neural networks are commonly used. Multilayer networks are networks, in which the neurons are organized in layers starting with the input layer and ending with the output layer, with hidden layers in between. For each neuron in this structure, its input connections originate only in the previous layer, and its output connections reach only to the following layer.

2.5.3 Learning process

Learning process of an neural network is an optimization problem, where we want to optimize the error function. Error function describes a difference between actual and expected output values for given set of training data. One of the popular error functions is mean square error function:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Learning process consists of showing inputs to the network and adjusting its connection weights based on the actual and correct output values in order to lower the MSE.

2.5.4 Learning algorithms

Back-propagation algorithm

Back-propagation is one of the most popular algorithms for neural networks training, and it serves as a base for many other algorithms.

The algorithm is based on the gradient descent method. In the first step, the input is evaluated and the mean square error of the output is computed. However, a different error function may be used. We can compute the gradient $\frac{\partial E}{\partial w_{ij}}$ for a given training sample using the chain rule

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial in_i}\frac{\partial in_i}{\partial w_{ij}} = a_j * \frac{\partial E}{\partial in_i} = a_j * \frac{\partial E}{\partial a_i} * \frac{\partial a_i}{\partial in_i},$$

where E is the error function, w_{ij} is the weight of the connection between the neurons i and j, in_i is the weighted sum of the inputs of the neuron i and a_i is the output value of the neuron i.

If we use the sigmoid activation function denoted g, the derivation is:

$$\frac{dg}{dx} = g(x)(1 - g(x))$$

which gives us:

$$\frac{\partial E}{\partial in_i} = a_i(1 - a_i)\frac{\partial E}{\partial a_i}.$$

We can put the equations together to obtain:

$$\frac{\partial E}{\partial w_{ij}} = a_j a_i(1 - a_i)\frac{\partial E}{\partial a_i}$$

There we have two possible cases for the neuron i:

1. i is an output neuron. Then we get:

   $$\frac{\partial E}{\partial a_i} = -(t_i - a_i)$$

   where the t_i is the expected output for the neuron i for the current input.

2. i is a hidden neuron. In this case we consider all neurons k that receive input from i. Since we are propagating backwards, we know the values $\frac{\partial E}{\partial a_k}$ for all k. Using chain rule again gives us:

   $$\frac{\partial E}{\partial a_i} = \sum_k \frac{\partial in_k}{\partial a_i}\frac{\partial E}{\partial in_k}$$

   and from combining the equations we get:

   $$\frac{\partial E}{\partial a_i} = \sum_k w_{ki} a_k(1 - a_k)\frac{\partial E}{\partial a_k}$$

Now we have defined gradient values $\frac{\partial E}{\partial w_{ij}}$ for all weights in a given neural network. We can then use the following rule to update the weights:

$$\Delta^t w_{ij} = -\epsilon\frac{\partial E}{\partial w_{ij}} + \alpha\Delta w_{ij}^{t-1}.$$

$\Delta^t w_{ij}$ is the change at time t for the weight w_{ij}. The value ϵ is called the learning rate. If the learning rate is too low, it will take much longer to train the network. If it is too high, the algorithm will cross large section in the search space and will probably oscillate.

In summary, the input is evaluated and the difference between the correct and actual output is computed using error function. The difference is propagated back through the network and weights are updated using the gradient descent method. The whole process is repeated until the desired error value is achieved.

Quick-prop algorithm

The quick-prop algorithm is an improved version of the backpropagation. It is based on independent optimization steps for each weight, rather than updating all weights at once. For the update computation, it also requires data from the last iteration, which increases space complexity. The modified Δ function of the quick-prop algorithm is:

$$\Delta^t w_{ij} = \Delta^{t-1} w_{ij} * \left(\frac{\bigtriangledown_{ij} E^t}{\bigtriangledown_{ij} E^{t-1} - \bigtriangledown_{ij} E^t}\right)$$

where $\Delta^t w_{ij}$ is the weight delta from t-th iteration and $\bigtriangledown_{ij} E^t$ denotes partial w_{ij}-derivation of error function E in t iteration.

2.5.5 Advanced structures of neural networks

There are several advanced techniques in the neural networks field that are commonly used. We describe two of them, the convolution networks and deep neural networks.

Deep neural networks All neural networks with more than two hidden layers can be considered deep neural networks. With the advantage of more layers, deep neural networks can outperform other methods of machine learning. The process of training such network is called *deep learning*.

More layers allow the network to make more complicated model abstractions over the data. However, there are many challenges to overcome. Deep neural networks have a large number of parameters and training them is demanding on computational power. They are also prone to overfitting, as the added layers of abstraction allow them to find possible unforeseen dependencies in the training data.

Convolutional neural networks Convolutional networks can be considered a special case of deep neural networks, because they usually have higher number of layers. They are well suited for pattern recognition directly from the pixels of the image because of their structure of layers with different purposes. For convolutional networks the convolutional layer is characteristic. This layer is essentially a filter, that performs feature extraction. The filter is composed from neurons that have shared weights, so that the filter performs the same operation on every part of the input. This in turn lowers the number of parameters

of the network. Usually, more than one filter is used on the input image, allowing the network to extract different features from the simple ones like edges and corners, to very complex features, such as a house or a tree.

Despite of the performance advantages, we have decided to use the classic feed-forward neural networks with two hidden layers. Because we want to allow the users train the network for their own defined shapes, we need to be able to perform the learning on the user side and offer a simple interface to do so. This does not correspond well with the advanced neural network structures, which may require certain costly hardware or complicated learning process. Learning of classic neural networks is cheap in comparison, and they can still perform quite well in the recognition tasks.

2.5.6 Data preparation for pattern recognition

Because of their generalization properties, neural network are successfully used in pattern recognition. They are able to approximate an arbitrary mapping between the input values and the output values [Bis96]. With this ability, we are not forced to do feature extraction and we can initialize the network with pixel values directly.

It is however recommended to apply several pre-processing transformations that can improve the generalization performance significantly.

Input normalization One of the common forms of pre-processing is input normalization. By applying a linear transformation we can arrange all of the inputs to have zero mean and unit standard deviation over the transformed training set. In practice, input normalization ensures that the inputs and target outputs stay in unit range, and we can expect that the weights should also be in unit range. We can then initialize the weights with suitable random number. Otherwise, we would have to find a solution, where the weight values differ distinctly.

Training with noise Another technique that improves generalization capabilities of the network is training with noise. It involves the addition of a random vector to the input vectors during training. A remark by Bishop [Bis96] states that:

> Heuristically, we might expect that the noise will 'smear out' each data point and make it difficult for the network to fit individual data points precisely, and hence will reduce

over-fitting. In practice, it has been demonstrated that training with noise can indeed lead to improvements in network generalization.

Data dimensionality reduction Data dimensionality reduction is a pre-processing method that allows the network to have fewer input neurons. It can take the the form of color information removal or pixel averaging, where we group several pixel to one block and use average of each block as an input. By lowering the number of inputs, we lower the number of parameters that the learning process has to optimize.

Feature extraction Some of the more complicated techniques use feature extraction as a pre-processing step. Simple geometric primitives extractions are common, such as extraction of lines with their lengths and angles.

3. Implementation

In the first part of this chapter we describe the recognition system implemented by this thesis, in the second part we describe the game prototype that demonstrates the usage of the algorithm. Implementation of the recognition system consists of two parts, one responsible for the actual recognition, and one for the training of neural network based on user-defined shapes. The usage of the framework consists of three steps:

1. Defining the shapes and setting up desired recognition parameters.
2. Training the neural network to classify the defined shapes.
3. Using the framework and the trained network to recognize patterns.

3.1 Data representations

We have developed several data classes that are crucial for the algorithm functionality. The `ImageLines` is a class that holds the input to the algorithm. The next class is a `ShapeDescriptor`, an abstract class whose implementations are used to describe shapes, and the last one is a `ShapeNode` class for the output data representation.

Image lines class We use the vector form in the interface. The shapes are defined as a vector of lines with the start and end point. The format of input is an instance of the `ImageLines` class, defined in the `ImageAnalyzer.h` header file. This class wraps a vector of lines defining the shape, accessible through `GetImageLines` method. The user is expected to fill the vector with the lines representing the two-dimensional image before passing it into the `Analyze` method. The class offers several other methods, used primarily by the recognition algorithm.

A line is represented by the instance of class `Line`, whose constructor accepts two `float2` type vectors, first containing the start coordinates of the line, and second containing the end coordinates. Internally, the coordinates are transformed into three-dimensional vectors of a homogeneous coordinate system.

Shape descriptors Shape descriptors are a key part of the system, as they are used both during training of the neural network and during analysis of image. They provide the information about the expected line positions and possible embedded shape locations to the algorithm.

Library user can define their own shapes by inheriting from the abstract class `ShapeDescriptor` defined in `ImageAnalyzer.h` file. The shape descriptors use the parametric-like expressions to define shapes. Abstract class functions, which should be defined by the user, take parameter `t` in the interval $[0, 1]$ as an argument, and return a point on the curve in a 2D space in the interval $[0, 1]^2$.

The curve does not have to be continuous, but it has to return a value for every parameter value in the range. Thread safety of some methods is also required. It is recommended to keep the shape complexity low, as some very small details may be lost in the data preparation for the training.

Output data format Output of the algorithm implementation is in the form of instance of `ShapeNode` class defined in `ImageAnalyzer.h` file. It represents a tree structure shown in fig. 3.1, where each node contains the recognized shape index and its pattern shape index. Embedded shapes are contained in a vector of child nodes. Indexes are numbers assigned to each descriptor during algorithm setup, except for the `unknown` shape index, which serves as a placeholder for unrecognized shapes. We do not differentiate between the empty space and any unrecognized shape.

3.2 Recognition process

The recognition process is depicted in fig. 3.3. It starts in the `Analyze` method, declared in the `ImageAnalyzer.h` file. It accepts an instance of `ImageLines` class as an argument, and returns `ShapeNode` instance describing the recognized elements in the image.

We have then decided to create the interface for the vector graphics form of input. We expect the input to the program to represent rather simple black and white shapes or their combinations, without any additional information like color or different shades. We also expect the input to be created on the computer in a game, rather than from a photo, and in this environment, it is common to use vector graphics.

The image lines are first normalized by normalization to the $[0 - 1]^2$ coordinate space. This transformation is essentially a rescaling, preserving both the length ratios and angles. Then, the top level shape is analyzed by rasterizing the lines into the pixel map which is then passed into the trained neural network.

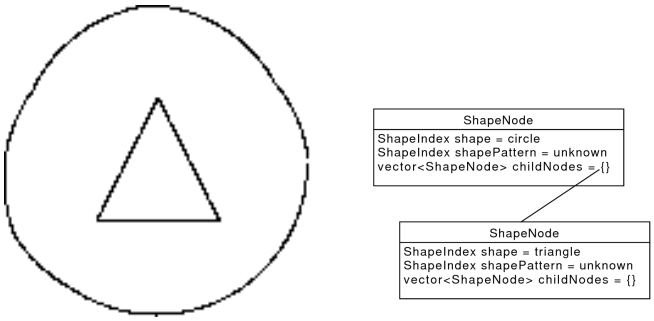

Figure 3.1: Input image with expected output. The embeddings form a tree hierarchy, with a single shape in the root and the embedded shapes as its sons. The `unknown` shape index marks both unrecognized shapes and empty space.

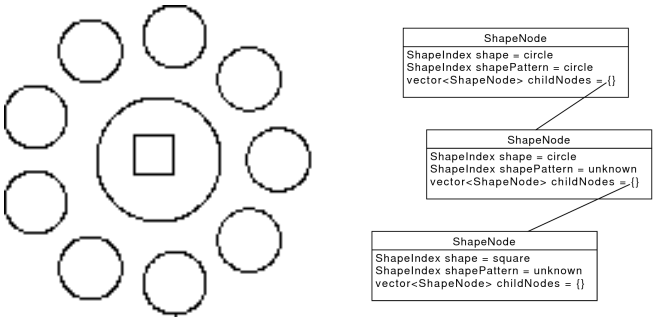

Figure 3.2: The output schema for a more complicated image.

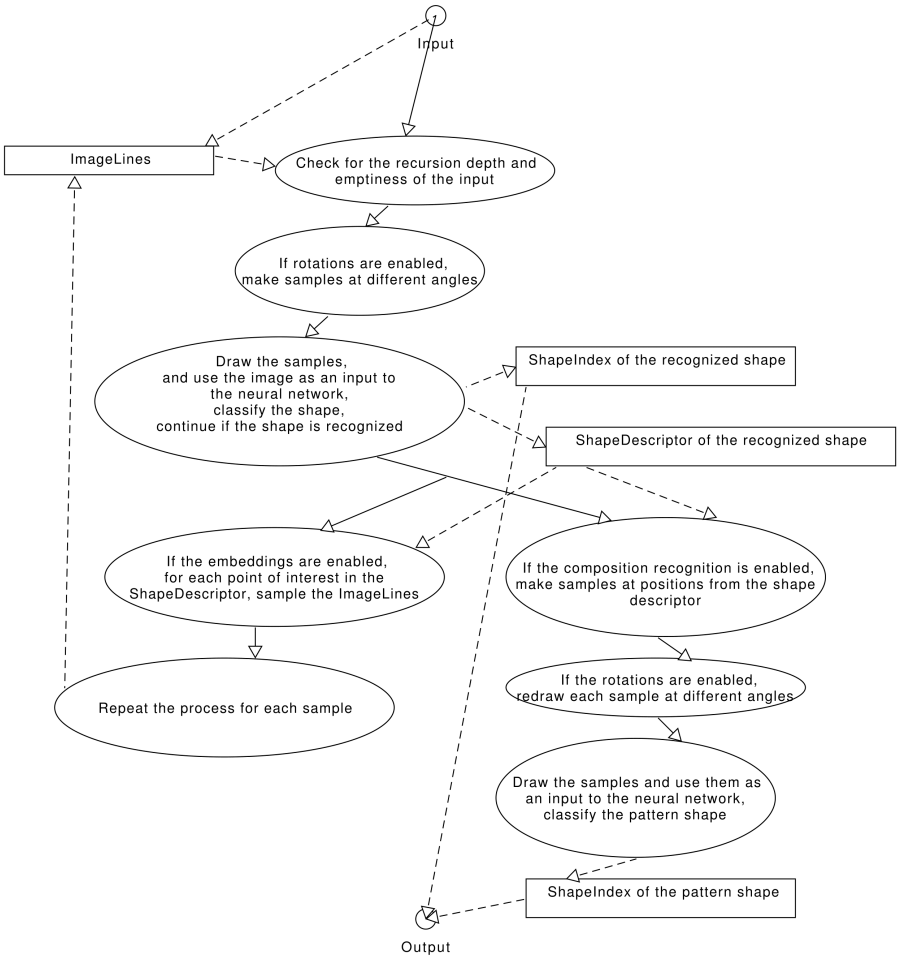

Figure 3.3: The figure shows the high-level schema of the algorithm. The dashed arrows show appearances of important data structures: `ImageLines`, `ShapeIndex` and `ShapeDescriptor`. The full lines describe the algorithm flow. The algorithm starts with the input, the output is built during the analysis. The algorithm stops when the shape is not recognized.

Generally, the highest neural network output considered to be the matched shape pattern, but only if the value is higher than a user-set constant. There are optionally several more steps involved, described below:

Rotated shapes If a recognition of rotated shapes is enabled, the image lines are rotated several times (the exact number is definable by user) by an appropriate angle value, re-drawn and analyzed by the network at each angle.

The network returns a vector of values, each output's value signaling the similarity to the shape assigned to the output, and the highest value is considered the matched shape. If there are several highest network outputs for different rotations with the same value, the first highest value among all outputs is considered. A recognized shape index is returned along with its matching rotation, in which it scored the highest network output.

Composed shape When the top level shape is recognized, its shape descriptor is used to sample the theoretical curve points via parameter t uniformly on interval $[0-1]$. If the recognition of rotated shapes is enabled, we transform the sample by an inverse rotation to the matched rotation, to align the recognized shape with the shape descriptor. Then, for each sampled point, this point is expanded into a small square, the original image is clipped to this square, discarding the lines outside, and the result is analyzed.

Recognized pattern shapes are counted over all samples, and if the count of matches for any shape is high enough, the top level shape is recognized as a composed shape created from a number of smaller shapes of the type with the highest count. Pattern shapes that form the composition are not tested for embedded shapes.

Embedded shape To recognize embedded shapes, we use the shape descriptor of the shape recognized previously, and focus on the user-defined embedded shape locations. For each embedded shape location defined in the shape descriptor, we clip the image lines to that area, and recursively run the `Analyze` method on the result. If the recognition of rotated shapes is enabled, we transform the an area by inverse rotation to the matched rotation, to align the recognized shape with the shape descriptor.

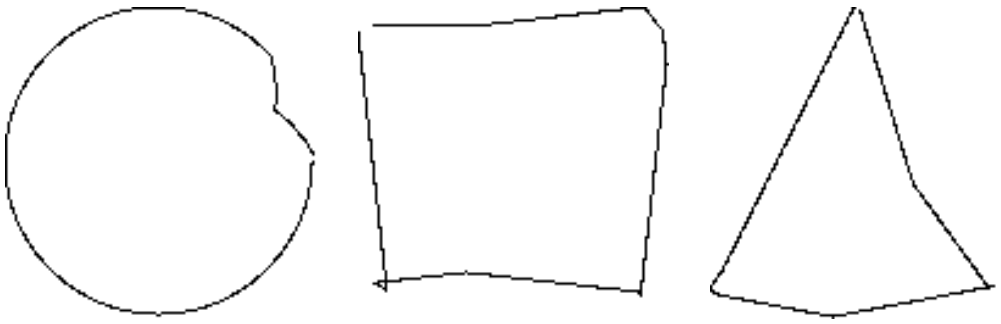

Figure 3.4: Examples of generated deformations of the shapes.

3.3 Neural network preparation

We use the *Fast Artificial Neural Network Library* [Fan] to train and use neural networks. This open source software is easy to use and install while having a lot of options in terms of network architecture and training. Thanks to its simple interface, we were able to develop an automatized training system for the networks.

3.3.1 Training data

Training the neural network requires a training dataset. Since we want to allow the extensibility in the form of user defined shapes, we have two options. Either to let the user somehow make his own dataset, which is not very convenient, or generate the data procedurally and try to approximate human-like drawing.

We decided for the second option. We generate the data based on the shape descriptors registered by the user and on the algorithm settings. To generate a data sample for the neural network, we take the shape descriptor and create a `ImageLines` instance from it.

Based on the algorithm settings, it will be either a deformed simple shape instance or an instance of a compound shape. In the case of creating a deformed simple shape, shape curve is sampled many times using the shape descriptor, and the points are connected by lines.

To approximate the deformations each point sample can be moved in the direction perpendicular to the line between the original position of the last point, and the original position of the the transformed point. Examples of deformations created by this procedure can be seen in fig. 3.4.

The transformation offset is linearly magnified and then reduced, resulting in a heuristics that try to approximate some of the humans like deformations

of drawn shapes. Then, the line between the previous and new point is added to the image instance.

In the case of the shape conglomeration, the shape curve is sampled less frequently and each point is replaced by rescaled image lines of a randomly chosen shape, but this shape is the same for the whole composition. See fig. 1.4 for examples.

3.3.2 Training

Training is done using the generated data from the previous methods and the *FANN* library. First, the training data and test data are generated,The size of the test data is one-third of the size of training data. Then the network is repeatedly trained on the training data until it reaches the user set up MSE, or until it stops improving.

Internally, the training starts with a high MSE target value, which is then gradually lowered, and in between the improvement ratio is checked. We use the resilient backpropagation (rprop) algorithm from the FANN library, because we achieved best results with this algorithm in terms of the improvements stability. In the neural network, we use an *Elliot* activation function [1].

3.4 Game prototype

The game, which demonstrates the usage of this pattern recognition work, has been developed using the *Urho3D* [Urh] game engine in version 1.7. Urho3D is, as stated by authors, a free, lightweight, cross-platform 2D and 3D game engine implemented in C++ and released under the MIT license.

3.4.1 Game description

In the game, the player controls the character from a top down view perspective. The player can move and attack the enemies, and, most importantly, cast spells by drawing.

Spells of the game are represented by totem objects — each time the player casts spell, a totem is created. These totem have different effects on the characters. They can take a form of positive effects that affect the player's character, like healing, or negative effects that affect the enemies, like

[1] The Elliot activation function is defined as $\text{ELLIOT}(x) = \frac{\frac{x \cdot s}{2}}{1+|x \cdot s|} + 0.5$, where s is the steepness parameter.

burning. The task for the player is to survive and destroy all the enemies in the game.

We now describe some of the constructs of the game implementation:

Drawable texture `DrawableTexture` is a component attached to ground in the game. It allows player to draw shapes onto the ground. This component handles the lines sampling and glowing shape line nodes, depicted in fig. 3.6.

Characters The player's character and the enemies 3.5 are represented by a composition of several components: the animated mutant model component, the health component and movement component. The enemies have also an AI component attached, while the player's character has an controller component attached which causes it to listen to the keyboard controls, and the caster component, which allows the character to extract drawn shape from the drawable texture, like in fig. 3.7. the caster component also transforms the shape into an `ImageLines` instance for the recognition.

SpellSystem The `SpellSystem` component is a initialization component for the recognition algorithm. It is created only once for the game and the algorithm setup is situated in the constructor. It holds the shape indexes and is responsible for the algorithm output parsing and construction of the spell.

Spell An instance of a node with the spell component is created each time the player casts a spell. It is a temporary class, serving as a placeholder for the totem object. It invokes the algorithm's `Analyze` method in a separate thread and waits for the result. This way, the game does not freeze while the algorithm analyzes the image. The place of the totem is marked by a green fire, as shown in fig. 3.8, while the image is analyzed.

Totem The core of the game are totem objects (see fig. 3.9) and all the spell effects are represented by them. Each totem has a duration for which it exists and an area of effect around it, where the characters are influenced by its presence. The totem can have several effects to it attached.

Effects Effects are components attached to the totems. When a character is inside an area of effect of the totem, the totem applies all of its effects on the character. However, the effects can also be of an single activation type, in which case they are activated only once when the

totem is created. An example of such effect is an effect that increases the totem duration.

States When an effect with duration is applied onto the character, a state component is attached to it, as shown in fig. 3.10. For example, when an over-time healing effect is applied onto a character, the healing state component is attached to the character for a given duration.

3.4.2 Spell system

The game spell system consists of several parts. The first is the `DrawableTexture` component, which is attached to the ground entity. This component is able to track players drawings and store them as lists of points, each list representing one continuous line.

When the player presses the button to cast the spell, the `Caster` component attached to their character extracts the lines in a rectangular area around the character and transforms them into the vector of lines. Then the instance of `Spell` class defined in the `SpellSystem.h` file is created. There, the `ImageAnalyzer::Analyze` method is finally called in a different thread and when the analysis is done, the result is parsed and the spell is cast.

Figure 3.5: There are three mutants in the image, one is controlled by the player, others are controlled by AI. The player starts drawing the spell on the ground by pushing the left mouse button. The mouse position is projected in the `Caster` component to the game surface and added to the `DrawableTexture` component which creates a new line.

Figure 3.6: The blue glowing line in the image represents the drawing. While the player holds the mouse button, the mouse positions are repeatedly sampled and projected to the surface. The projection points are then added to the line and the space between the points is filled with a blue glow dots to show the line.

Figure 3.7: The image shows a finished square shape. When the player releases the mouse, the current line is ended. It is possible to continue drawing other shapes. However, when the player casts spell, all lines in a large area around the character are consumed and used for the spell. The extraction of the lines from `DrawableTexture` and conversion into `ImageLines` class is done in the `Caster` component.

Figure 3.8: When the player casts a spell, the extracted lines are removed and a node with the `Spell` component is created to handle the spell. The node is represented by green glowing orb and it marks the place where the totem will be created. It invokes the `ImageAnalyer::Analyze` method with the lines in another thread to avoid the freezing while the analysis is performed.

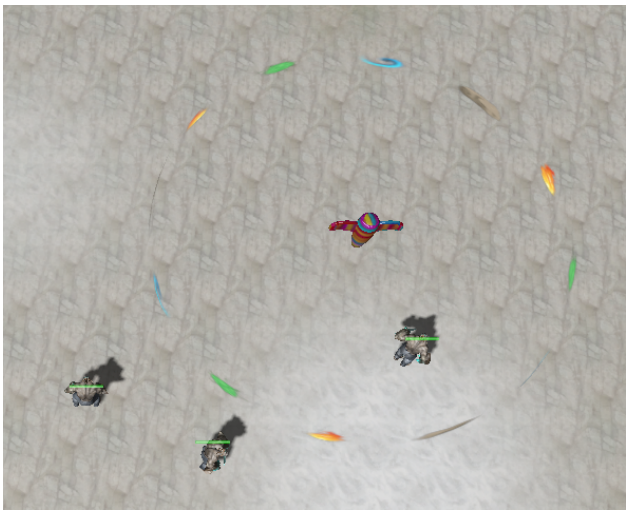

Figure 3.9: When the analysis is finished, the `SpellSystem` component parses the output. It creates a node with the `Totem` component, and for each shape, the corresponding effect is added to the totem. The totem is then placed on the spot of the `Spell`. The totem's area of effect is marked by a rotating circle of elements.

Figure 3.10: The recognized shape was square. Since square represents fire, the totem has the `FireEffect` component attached to it. When the characters are detected inside the area of effect, the totem applies all of its effect on them. In the figure, the `FireEffect` is applied to the enemy inside the circle, causing it to have `FireState` component attached. This component makes the enemy burn, periodically decreasing its health.

4. Results

In this chapter we evaluate the learning performance of the resulting library. In the first section we describe the process of the optimization of the neural network parameters, in order to achieve the best performance. In the next sections we optimize the parameters of the algorithm and evaluate its performance. For the purposes of evaluation, we have decided to use fixed four shape descriptors, namely the square, circle, triangle and water drop shape descriptors throughout this chapter. If the number of shape descriptors changes, the same method can be applied to re-establish the modified optimal parameters. Examples of the expected inputs of embedding and composition can be seen in fig. A.5 and fig. A.10.

4.1 Neural networks performance

For the correct functionality of the algorithm, it is required that the network is able to recognize the shapes. There are numerous parameters for the neural networks, like the training algorithm choice, architecture and activation function. Because it is not feasible to test every combination of the parameters, we have tested the influence of only a several of these parameters.

4.1.1 Setup

For all the tests we have fixed the following parameters like this:

Network architecture All tested networks have a layered structure, with two hidden layers. However, we have tested the influence of different sizes of the layers.

Training algorithm The networks are trained using resilient backpropagation algorithm from the FANN library.

Activation function The neurons of the network use the Elliot activation function, a faster version of the sigmoid activation function.

Shape descriptors We have used four basic shapes descriptors: square, circle, triangle and a water drop throughout the the whole chapter.

Data The training data and the test data were generated by the developed generator. The training data consisted of 150 000 images, where each shape had the same amount of examples, and 50 000 images of random

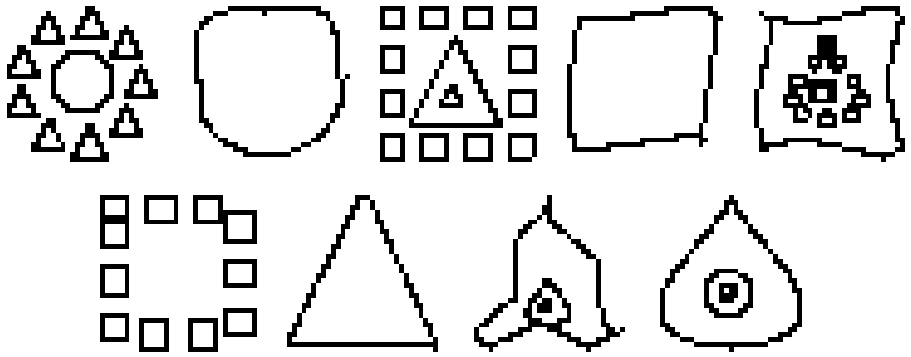

Figure 4.1: 9 examples of auto-generated training data. The resolution of each image is 32*32 pixels.

data. The test data consisted of 40 000 images of examples of shapes, and 20 000 images of random data. Examples of the data are shown in fig. 4.1.

4.1.2 Tested parameters

We have tested the combinations o the following parameters:

Layer size We have tried several values for both layers. For the first hidden layer, we have tried 100, 200 and 300 neurons, and for the second hidden layer 10, 20 and 30 neurons. These values were chosen heuristically from the range recommended by Karsoliya [Kar12].

The value of MSE We have trained the networks at different levels of precision, with the target MSE values of 0.1, 0.07, 0.04 and 0.01.

Algorithm settings The neural networks were trained for different combinations of algorithm settings, with embeddings and composition turned off and on.

We trained the networks with all the combinations of the parameters above, which gave us a total of 150 trained neural networks to evaluate.

4.1.3 Evaluation

The networks were evaluated in the following scenario. For each of the combinations of embeddings and composition being turned on or off, the dataset

of 1000 images was generated, with the same amount of examples for each shape. Then, the network was assigned a score computed as a sum of scores over all examples. Score per example was assigned by the following rules, in order to mimic the actual usage in the algorithm, where we consider only the maximum output:

1 If the maximum of the outputs of the network is in the corresponding output neuron of the shape that is in the example, which means, that the network classified the shape correctly.

2 If the maximum of in the correct output neuron and is higher than 0.7.

-1 If the maximum is in the wrong output neuron which corresponds to the different shape than the one in the image, and the value is higher than 0.7.

0 Otherwise.

These rules mimic the usage of the network in the algorithm, where only maximum values above 0.7 are considered. The value 0.7 was chosen because the scores of the networks differ more with a numbers closer to 1, and the plots are more clear. From the rules it follows that each network could achieve a score in range from -1000 to 2000 on the dataset of 1000 samples.

4.1.4 Results

Figures 4.2 to 4.5 show the score of the networks achieved on different evaluation datasets. The datasets contain 1000 samples each, but differ in the techniques used in the images. The first dataset contains only simple shapes, without embeddings or conglomeration, the second contains embeddings only, the third composition only, and finally the fourth contains both embeddings and composition combined.

There is a visible gap on the y-axis between 0.07 and 0.04 in all graphs. This is caused by the a sudden improvement when the network was trained to reach MSE under 0.07. The network usually improved even a little more, finishing the training with a MSE around 0.04 or lower.

Size of the network did not show any visible influence on the network performance. It therefore seems that even the network with the lowest number of neurons is sufficient for this setup, although more neurons might be needed for networks that need to recognize more basic shape descriptors.

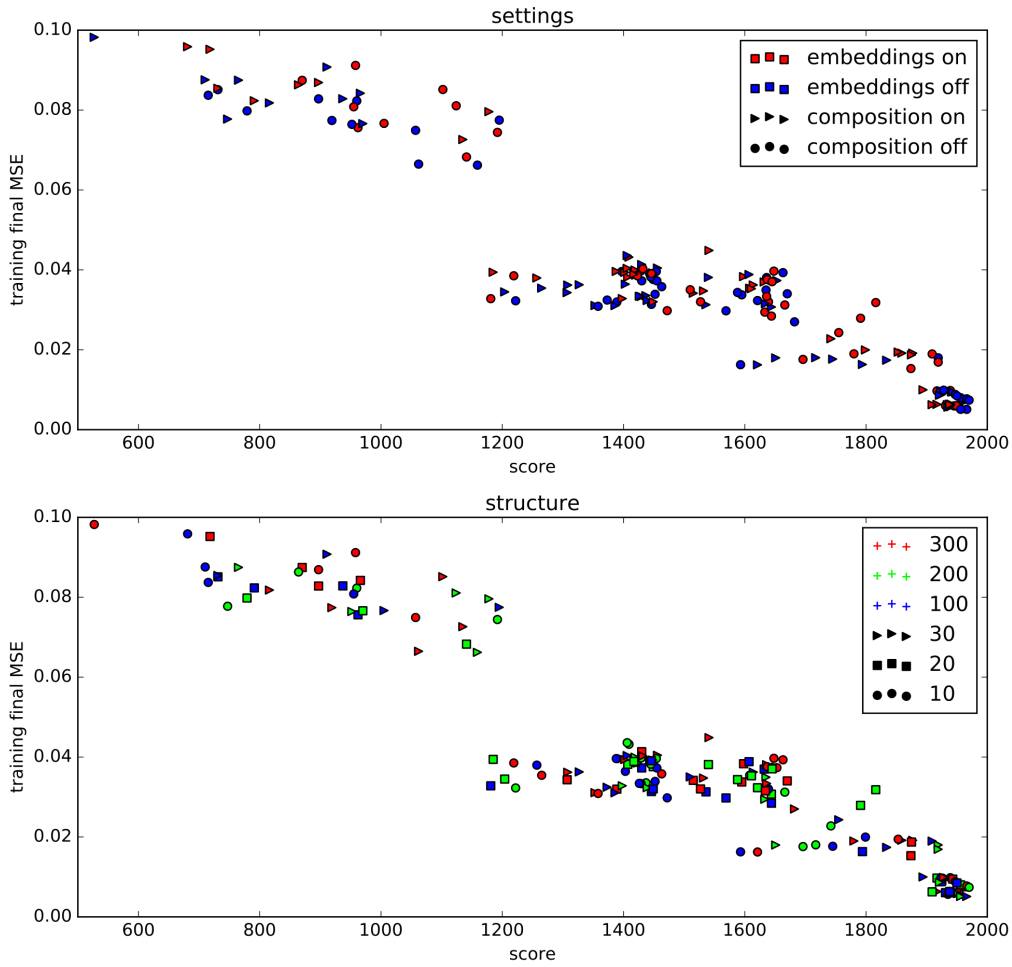

Figure 4.2: The plot shows results on a dataset containing only a simple shapes, without embeddings or conglomeration. Each point represents a single network. The x-axis shows achieved score according to the rules (see section 4.1.3), and the y-axis shows the achieved MSE of the network during training. The first graph shows by color whether was trained to recognize embeddings, and by shape whether the network was trained to recognize conglomerations. The second graph shows the structure of the network. The color marks the number of neurons in the first hidden layer and the shape marks the number of neurons in the second hidden layer.

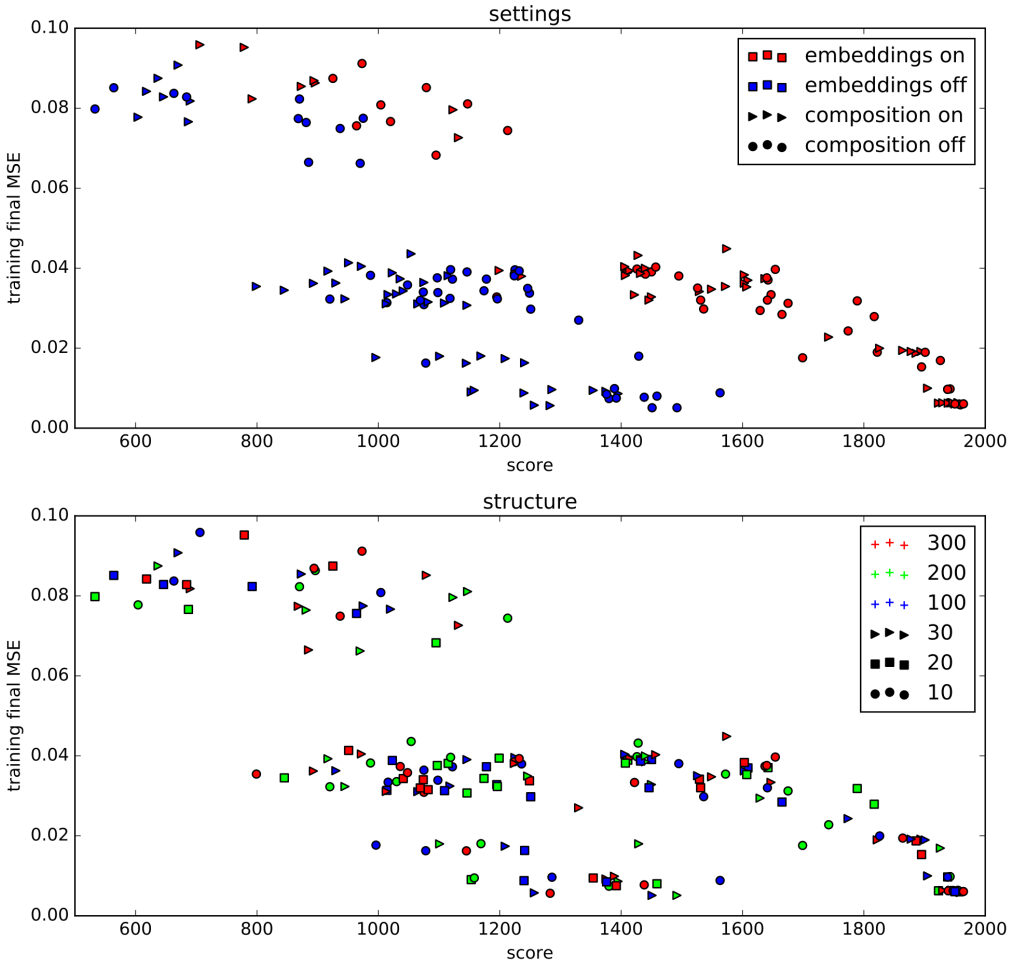

Figure 4.3: The plot shows result on a dataset containing embedded shapes. Each sample contained shape with an embedded shape. The plot follows the same format as in fig. 4.2.

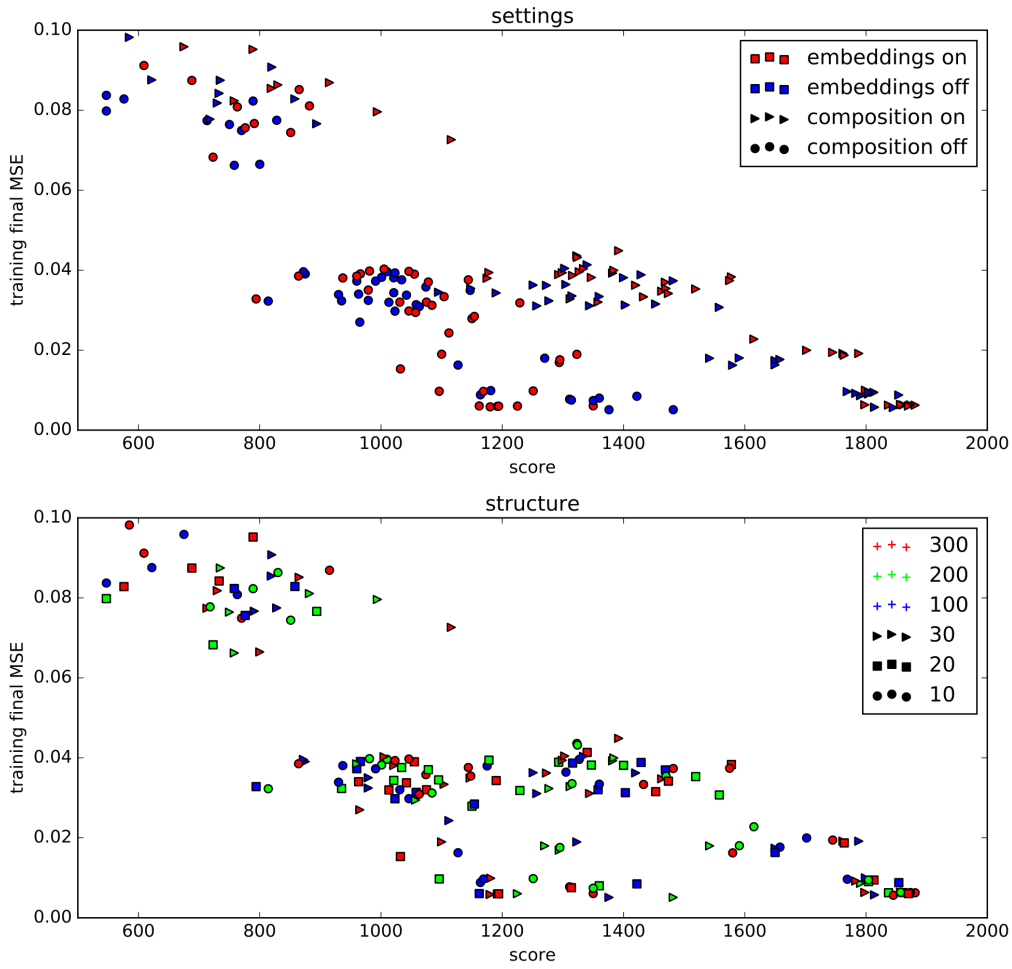

Figure 4.4: The plot shows result on a dataset containing composition of shapes. Each sample contained shape composed from a number of instances of a pattern shape. The plot follows the same format as in fig. 4.2.

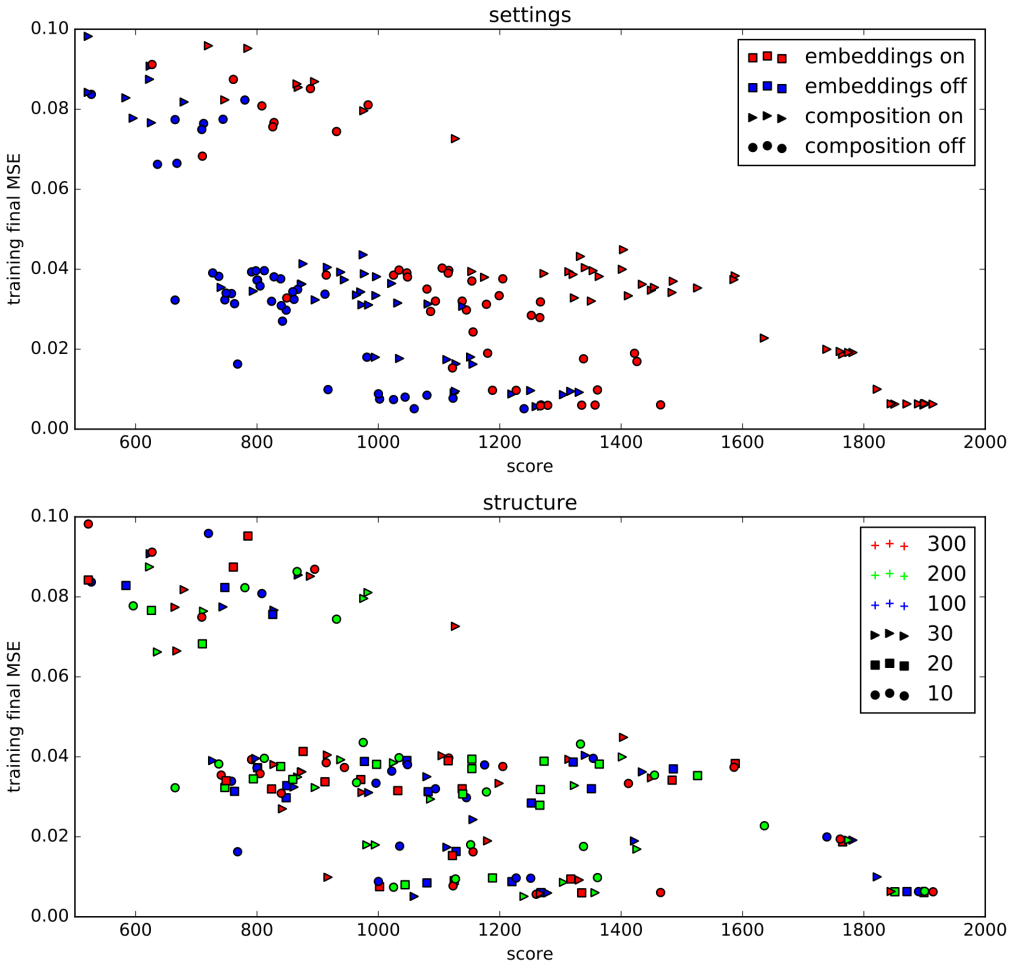

Figure 4.5: The plot shows result on a dataset containing combination of embedded and composed shapes. Each sample contained a shape composed from a number of instances of a pattern shape and with an embedded shape. The plot follows the same format as in fig. 4.2.

It can also be observed from the figures that the tendency to overtrain is not present even with very low MSE. It is therefore preferable to aim for a lower MSE values during training, as that clearly improves the achieved score.

The key result, visible in fig. 4.2, is that training the network to recognize the embeddings and composition effects decreases only slightly the performance on the simple shapes if the network is trained for a low final MSE.

4.2 Algorithm parameters optimization

In this section, we test and evaluate the recognition algorithm performance. In the evaluation, we use the best network trained for both embeddings and composition, since it was observed from the results that the network trained for both cases performs almost as well as networks trained for only one case. We attempt to optimize the algorithm parameters (described in appendix A.1.2) for the recognition of the used shape descriptors: square, circle, triangle and water drop. It may be possible that the parameters would have different optimal values for different shape descriptors.

4.2.1 Composition recognition

To measure the precision of recognition of composed shapes we have prepared 100 `ImageLines` instances of composition examples. Each instance represents a shape composed of instances of another shape. We have also setup the algorithm this way:

- `COMPOSED_SHAPES_ENABLED = true;` — Indicates that the algorithm should search for the pattern shape.
- `EMBEDDED_SHAPES_ENABLED = false;` — Indicates that the areas of interest defined by `ShapeDescriptors` are not examined for embeddings.
- `ROTATION_ENABLED = false;` — Indicates that the shapes should not be analyzed at different rotations.

This setting causes the algorithm to search for the pattern shape of the possible composition, but to ignore embeddings locations and to not match different rotations.

We have tested different combinations of settings of these variables:

1. `COMPOSITION_SAMPLES_COUNT`

2. COMPOSITION_SAMPLES_LIMIT

3. COMPOSITION_WINDOW_SIZE

Again, see appendix A.1.2 for detailed parameter explanation.

For each test instance we have compared the correct pattern shape with the recognized pattern shape.

From the results in fig. 4.6, we can see that the success rate grows with the increasing number of samples. The algorithm has more chances to hit the area with the shape, rather than the area between the shapes. Generally the chance to recognize at least one pattern shape is higher.

We can also see that the red and black colors representing the smaller sizes of the sample window perform better with the higher sample count, while being faster at the same time. The performance improvement for the smaller sample window tells us that the network does not recognize the shapes if the surrounding of the area covered by parts of other shapes. The speed difference is caused mainly by rasterization algorithm that needs to process more lines that are found in larger windows.

There is also a sharp drop in success rate when increasing the sample limit to more than one. This means that the network has trouble recognizing even a single pattern shape from the shapes that form the composition. There are two factors that affect the recognition of the pattern shape:

The sampling window follows the ideal path from the shape descriptor, but the real shape is usually more or less deformed, which means that the window will hit only a part of the pattern shape.

The pattern shapes can be of various sizes which is hard to approximate by a single size of the sampling window. And if even the window hits the whole pattern shape, there is a very high chance of hitting also the noise from other shape patterns, or from embeddings.

Both factors make the recognition very difficult for the network. The network is rarely able to recognize even a single pattern shape correctly, and after 40 samples, the success rate does not improve much and stays somewhere between 40% and 60%.

It is clear that the algorithm becomes gradually slower with the increasing number of samples. The total time will also increase substantially if the rotation is turned on, because each composition sample is tested for all rotations.

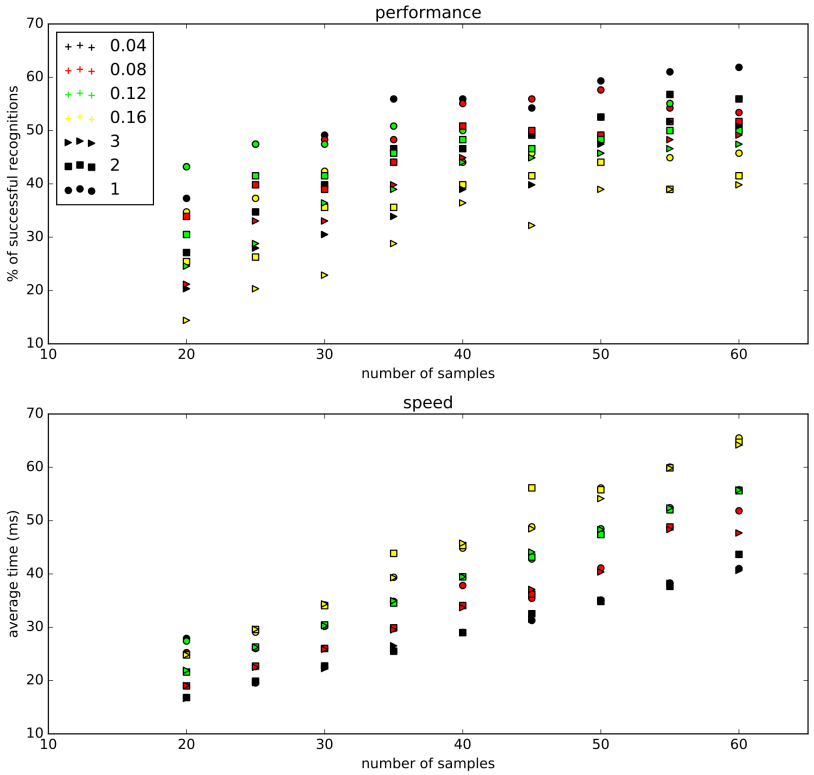

Figure 4.6: The x-axis shows the values of `COMPOSITION_SAMPLES_COUNT`, which is actually a count of samples made around the shape contour. The point color denotes the `COMPOSITION_WINDOW_SIZE`, and the point shape denotes the `COMPOSITION_SAMPLES_LIMIT` according to the legend. Each object of the plot then represents a combination of parameters settings and the performance and speed achieved with the parameters over the test samples. The y-axis of the first plot shows the successful recognitions percentage over the test samples, and in the second plot, the y-axis shows an average computation time per one sample.

4.3 Rotation

Rotation invariance is achieved by redrawing the image at different angles and returning the best match from the rotated samples. We tested the influence of the `ROTATION_SAMPLES_COUNT` variable on the precision and speed of the algorithm.

In the results fig. 4.7 we can see that the precision of simple data shapes reaches almost 100% at 15 samples. However, the recognition of the composed shapes is much worse. At 15 samples, it reaches about 70% success rate and barely improves for the higher sample values. At the same time we know from fig. 4.5 that the network is able to recognize composed shapes quite successfully.

From the result data, we conclude that the main cause of the problem are false matches of different shapes. For example, a square composed from circles can, at 45 degrees rotation, closely resemble a circle, and the network assigns the circle a higher match value than the square at the correct rotation.

We can also observe from fig. 4.7, that the speed of the recognition of composed shapes is much slower and the gap in the performance is increasing with the samples count. This is caused by the fact that composed shapes contain a lot more lines than the simple shapes and each of these lines adds to the transformation time and to the drawing time.

We can deduce that setting `ROTATION_SAMPLES_COUNT` higher than 15 is rather unnecessary.

4.4 Embeddings

The ability of the algorithm to recognize embeddings depends more on the shape descriptor definition rather than on the algorithm settings.

The library user can define the points of interest by their position and size, and these two parameters form a rectangular area where the algorithm will search for embedded shape.

It is then recommended to set the size of this area lower, to avoid fragments of the top shape appearing in this area, which might cause the network to not recognize the embedded shape. If the embeddings are supposed to appear inside composed shapes, the area should be even smaller, otherwise the fragments of the pattern shapes will appear inside.

Embedding matching has a simple influence on the algorithm performance. Every point of interest defined in the `ShapeDescriptor` is analyzed and if the embedded shape is found, the algorithm runs recursively on the area of the embedded shape.

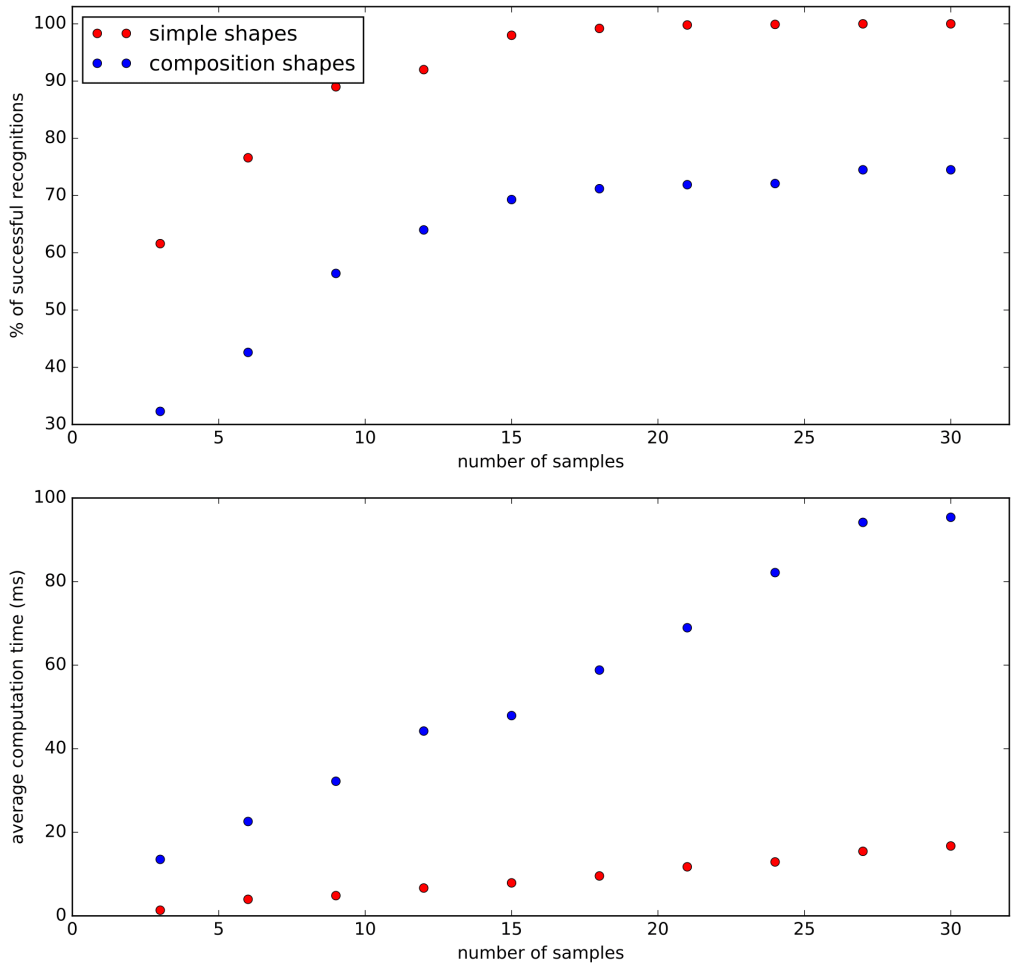

Figure 4.7: The x-axis shows the values of `ROTATION_SAMPLES_COUNT`. Each dot represents a test run over the test data. The red dots represent results over the test data with simple shapes, and the blue dots represent results over the test data with composed shapes. The y-axis of the first plot shows success recognition rate and in the second plot the y-axis shows an average computation time per one sample.

Conclusion

The goal of this thesis was to develop a recognition algorithm that recognizes structured combinations, such as compositions and embeddings. We have reviewed the literature for pattern recognition and found out that although the topic of recognition of shapes is covered well, approaches to recognition of shape combinations and modifications that this thesis targets are rare. From several reviewed algorithms we have chosen the artificial neural networks as a base for our recognition algorithm.

We have developed a recognition library that allows users to define their own shapes, train the appropriate neural network and use the library for recognition of structures made from the defined shapes. For the purpose of the neural network learning, we have also developed a shape images generator that attempts to approximate human imprecise drawing.

We have then used the library in a game prototype to demonstrate its functionality. The game supports several defined shapes and their combinations in the form of embeddings and composition, which all systematically map to different spell effects.

In the last chapter, we have benchmarked the performance of our library and optimized the algorithm parameters for the used shape descriptors. The results suggest that while the success rate of the recognition of composed and embedded shapes is high, the recognition process of the pattern shape of the composition still needs improvements.

4.5 Future work

There are several further directions that might be worth exploring:

- One of the main problems is the noise from the surroundings introduced when extracting the area of embeddings and pattern shapes. Heuristics, like extracting only continuous segments of lines or using neural networks with stronger generalization power, might be used to improve the recognition.

- Performance of the algorithm might be an issue in more demanding game environments. Apart from the fact that the analysis of several images can be performed in parallel, the algorithm can be even further parallelized. The large number of relatively simple repeated computations is especially well suited for being run on GPU.

- Developing a playable, gamer-targeting game that uses the library would be a great final step of the development. The provided game prototype is arguably not very well suited for the gamers (which will find it boring after several spells), nor for the library: The game currently forces the players to draw shape during combat and while moving the character, resulting time stress discourages the player from trying out more complex shapes.

  For example, a strategy game might be a better candidate. Instead of building the base with player's units, players could create them by drawing the correct shape. The advanced shape structures would then create buildings with stronger units.

  Using an input controller that is more suitable for drawing than a mouse might also greatly enhance the player experience.

Bibliography

[BA83] Horst Bunke and Gudrun Allermann. "Inexact graph matching for structural pattern recognition". In: *Pattern Recognition Letters* 1.4 (1983), pp. 245–253.

[BL08] Xiang Bai and Longin Jan Latecki. "Path Similarity Skeleton Graph Matching". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 30.7 (July 2008), pp. 1282–1292. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2007.70769. URL: http://dx.doi.org/10.1109/TPAMI.2007.70769.

[BMP02] S. Belongie, J. Malik, and J. Puzicha. "Shape Matching and Object Recognition Using Shape Contexts". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 24.4 (Apr. 2002), pp. 509–522. ISSN: 0162-8828. DOI: 10.1109/34.993558. URL: http://dx.doi.org/10.1109/34.993558.

[Ban] Black and White fireball and teleport gestures. URL: https://vignette.wikia.nocookie.net/blackandwhite/images/7/77/Gestureteleport.png/revision/latest/scale-to-width-down/180?cb=20110810005636.

[Bis96] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, 1996. ISBN: 0198538642.

[Bmp] *EasyBMP*. URL: https://github.com/aburgh/EasyBMP (visited on 07/19/2018).

[Cas] Castlevania magic seals. URL: https://vignette.wikia.nocookie.net/castlevania/images/a/ab/Dawn_of_Sorrow_-_Magic_Seal_1.gif.

[Dou12] Geoff Dougherty. *Pattern Recognition and Classification: An Introduction*. Springer, 2012.

[Fan] *Fast Artificial Neural Network Library*. URL: https://github.com/libfann/fann (visited on 07/10/2018).

[GNIK03] Abdul Ghafoor, Rao Naveed Iqbal, and Shoab Khan. *Image Matching Using Distance Transform*. June 2003.

[How14] Jeff Howard. *Game Magic: A Designer's Guide to Magic Systems in Theory and Practice*. A K Peters/CRC Press, 2014.

[Kar12] Saurabh Karsoliya. "Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture". In: *International Journal of Engineering Trends and Technology* 3.6 (2012), pp. 714–717.

[Lew01] J.P. Lewis. "Fast Normalized Cross-Correlation". In: 10 (Oct. 2001).

[Lin] *linalg.h*. URL: https://github.com/sgorsten/linalg/ (visited on 07/19/2018).

[Mag] Magicka elements. URL: https://steamuserimages-a.akamaihd.net/ugc/558737235139743061/3E5FA655FD7F4B57C90D2221FF6DD0D9129C457B/.

[NP00] Luciano Nieddu and Giacomo Patrizi. "Formal methods in pattern recognition: A review". In: *European Journal of Operational Research 120* (2000).

[Neu] Neuron model. URL: https://www.researchgate.net/profile/Md_Shafiul_Alam3/publication/285400200/figure/fig5/AS:301898910453776@1448989717451/A-single-ANN-neuron-with-its-elements.png.

[PDM02] Euripides G. M. Petrakis, Aristeidis Diplaros, and Evangelos Milios. "Matching and Retrieval of Distorted and Occluded Shapes Using Dynamic Programming". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 24.11 (Nov. 2002), pp. 1501–1516. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2002.1046166. URL: http://dx.doi.org/10.1109/TPAMI.2002.1046166.

[Urh] *Urho3D*. URL: https://urho3d.github.io/ (visited on 06/15/2018).

[Wik18] Wikipedia contributors. *Convolution theorem — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Convolution_theorem&oldid=833285853. [Online; accessed 18-July-2018]. 2018.

A. User guide

A.1 Recognition system

The interface of the library is located in two headers: `ImageAnalyzer.h` and `Training.h` in the corresponding namespaces `ImageAnalyzer.h` and `Training.h`, respectively.

A.1.1 Creation of shape descriptor

For both the training and for the recognition shape descriptors are required. The shape descriptors have several rules, that need to be followed by the user in their implementation. The intended purpose of the descriptor is to describe a single 2D shape in a $[0, 1]^2$ rectangular area. The descriptor is expected to return the exactly same shape every time and all of its methods should return the same values when called with the same parameters.

It is also required, that all of the shape descriptor methods are thread safe. It is then recommended to design the shape descriptor as a class with a constant inner state. Not following these rules may result in an unexpected and dysfunctional behavior of the system.

The user can create a shape descriptor by including the `ImageAnalyzer.h` header file and inheriting from the class `ShapeDescriptor`. This class contains several abstract methods, that should be implemented:

`GetName()` Returns the name of the shape a string. This is only for debugging purposes and it does not have to be unique or even constant. However, it is recommended to return a constant unique name among the descriptors.

`GetPoint(float t)` Function overload with one parameter `t` should return a point on the contour of the shape, based on the `t` parameter. It is recommended to normalize the parameter into $[0, 1]$ range by `NormalizeParam` function.

`GetPoint(float last_t, float t, float & point)` Function overload with three parameters, that allows the descriptor to describe noncontinuous curves. The *point* parameter passed by a reference should be filled with the same value as from `GetPoint(t)` as it describes the point of the curve. Then the function should return true if the curve is continuous on the interval $[last\_t, t]$, otherwise false.

`GetPointsOfInterest()` Function, that describes places, where an embedded shape may appear. It returns a vector of `float3` type, where the first two numbers denote the top left corner of the square area, and the third number is the size of the square. It is important for the correct functionality that the points of interest do not overlap with the shape curve, and that they are substantially smaller than the parent shape.

Examples of shapes descriptors can be found in the `ExampleShapeDescriptors.h` file. Be careful to return the points only in the normalized range $[0, 1]^2$.

A.1.2 Algorithm properties

There are several variables then can be set up and influence the functionality of the software. They can be found in the `ImageAnalyzer` namespace. Some of them are used both in the training and in the recognition. It is necessary then for the user to be consistent and use the same settings for the recognition as they used for the training.

`DEBUG_IMAGE_SAVE` Boolean variable with default value false. Only for debugging purposes. If true, the images created during the recognition are saved as BMP files onto the hard disk into folder `debug/`, but only in the initial rotation of the image.

`DEBUG_IMAGE_SAVE_SIDE_SIZE` Integer variable that controls the size of the debug images.

`COMPOSED_SHAPES_ENABLED` Boolean variable with default value true. If false, recognition algorithm recognizes the whole shape but does not search for the pattern shape that might compose it. During training, the generating algorithm does not produce compositions of the shape, so the network is not trained to recognize the composed shapes.

`COMPOSITION_SAMPLES_COUNT` Integer variable describing the amount of samples taken when searching for the pattern shapes. This variable has no effect when composed shapes are disabled. Otherwise, the interval $[0, 1]$ is sampled uniformly and each sample is passed into the `GetPoint` method of the descriptor. The returned point is then expanded into the sampling window and checked for pattern shape. Values greater than zero are supported.

`COMPOSITION_WINDOW_SIZE` Float variable that describes the size of the sampling window used during pattern shape matching. The actual window

is a square with a side size of 2*COMPOSITION_WINDOW_SIZE. This variable has no effect when composed shapes are disabled. Values greater than zero are supported.

COMPOSITION_SAMPLES_LIMIT Integer variable describing the number of minimum pattern shape matches. If the count of the pattern shape matches is lower, the pattern shape is not recognized. This variable has no effect when composed shapes are disabled. Values greater than zero are supported.

EMBEDDED_SHAPES_ENABLED Boolean variable with default value true. If false, recognition algorithm ignores the GetPointsOfInterest method of the shape descriptor, where embedded shapes might be, and recognizes only the top level shape and its composing shape. During training, the generating algorithm will not generate embedded shapes, and the network will not be trained to filter out these locations, where the embeddings could appear.

ROTATIONS_ENABLED Boolean variable with default value true. If false, recognition algorithm will not test different shape rotations for the best match, but will use the initial rotation. This variable does not have an impact on training since rotation recognition is not a direct task of the neural network.

ROTATION_SAMPLES_COUNT The amount of samples created when matching rotated shape. It directly determines the rotation step angle size, which is in degrees 360/ROTATION_SAMPLES_COUNT. This variable has no effect when rotations are disabled. Values greater than zero are supported.

DEBUG_OUPUT Integer variable with default value 1. Controls the amount of debug info of the recognition system. If set to 0 or lower, no debug output is produced. If set to value 1, it prints recognized shape with its matching rotation and its composing shape to the standard output, for the top level shape and each embedded shape. If set to 2 or higher, produces the same output as with value 1, but also all network outputs from the analysis are printed for all the rotations.

SHAPE_VALUE_LIMIT Float value that controls the recognition threshold. If the maximal network output is lower than this value, the shape is not recognized. Recommended range of values is $[0, 1]$.

RECURSION_LIMIT Int value that controls the recursion depth of the algorithm. Values greater than zero are supported.

`IMAGE_SIDE_SIZE` Integer variable, controls the size of the images that are created anytime during algorithm. Every time there is an instance of `ImageLines` class that should be analyzed by the network, the lines are drawn into the square pixel map of a size set by this variable and then serialized as an input to the network. The network input layer size has to be the second power of this value. The default value is 32, which means that the neural network has an input layer of size 1024 neurons, and the images produced in the algorithm are pixel maps of $32 * 32$ pixels.

A.1.3 Training phase

Training the neural network consists of several steps. The first step is to set up the variables controlling the behavior of the recognition and training algorithms. By including the `Training.h` file, the user can access the `ImageAnalyzer` namespace for setting up the variables, and the `Training` namespace for the training functions. After setting the variables, it is necessary to register the desired shape descriptors through function the `Training::RegisterShapeDescriptor`. This function takes a unique pointer to the shape descriptor as its parameter, and returns an instance of the `ShapeIndex` class. This class is only a wrapper over integers but it works as an identifier of the shape in the algorithm and the calls to the `ImageAnalyzer::Analyze` will return the `ShapeIndex` for each recognized shape. The next step is to call the `Training::Train` function. This function has several parameters:

`std::string name` Describes the name of the neural network. When the network is created and trained, it is saved under the current working directory into the file with the name `name` with the extension `.net`.

`vector<unsigned int> networkStructure` This parameter describes the layers of the network. Each number represents number of neurons in a single layer. The first number describes the size of the input layer, while the last number describes the size of the output layer. The user can set any number of layers higher than two, and arbitrary sizes of layers, apart from the first and last layer. The first layer has to be equal to the second power of `IMAGE_SIDE_SIZE`, and the last layer size has to be equal to the number of registered shape descriptors.

`boolean generateData = true` Parameter that controls whether the training algorithm should generate the training data. The algorithm looks for training data in the current working directory in the file

`training.data` and for test data in file `test.data`. If the parameter is true, the algorithm will generate these two files, possibly overwriting them. If the parameter is false, the algorithm uses the files found, or ends with error, of the files are not found.

`float targetMSE = 0.01` The MSE that the network should achieve on the `test.data`. The training stops if the network achieves MSE lower than this value, or if the network does not improve anymore.

`int dataSize = 300 000` The parameter describing the number of generated training samples in the `training.data`. The size of the generated test data is one third of this value. if the parameter `generateData` is false, then this parameter is ignored.

after the training process finishes, the network is ready to be used in the recognition algorithm.

A.1.4 Recognition phase

When the neural network is ready, the user can set up the recognition system and use it in a game. The recognition system is used through interface located in the `ImageAnalyzer` namespace in `ImageAnalyzer.h` file. The namespace contains definition of the algorithm properties, as well as definition for classes that the user is supposed to use, namely the `ShapeDescriptor` class, the `ImageLines` class and the `ShapeIndex` class.

First, it is necessary to load the neural network into the algorithm, using `ImageAnalyzer::LoadNetwork` function, which takes string describing the path to the network as an argument. Then, the user has to register the same descriptors used for the training of the loaded network again, but this time using the `ImageAnalyzer::RegisterShapeDescriptor`, which takes the corresponding `ShapeIndex` instance together with the descriptor.

Alternatively, the `ShapeIndex` instance can be created from the order number of the corresponding shape descriptor when registered for the training. It is also necessary to set up the algorithm properties the same way they have been set up during network training. When these steps are done, the user can repeatedly call the `ImageAnalyzer::Analyze` function to determine the shape hierarchy in the provided `ImageLines` instance. The `Analyze` function is thread safe, so many calls can be done at the same time. However, the set up has to be done synchronously and only once, before the first call of `Analyze` function.

A.2 Game

The game is a simple prototype. When the game is started the character controlled by player is located in the middle of the screen. The character can move over the surface plane, but falling of the edge results in death. There will be also numerous enemies around. initially, all the enemies are harmless and the player has control over the AI activity. This feature was added because it was too difficult to draw more complicated spells without dying. The goal of the game is to destroy all the enemies, either trough usage of totems or character's attacks.

A.2.1 Controls

The game uses both keyboard and mouse with the following mappings:

W, A, S, D	The player can control the character using the keys W,A,S,D to move the character up, left, down and right respectively.
E, R	The keys E and R can be used to turn off, respectively on, the AI of the enemies.
Q	Using the Q key and pointing the mouse over the enemy, the players character starts moving towards the enemy and attacking it.
Left mouse button	The player can draw shapes on the ground using the left mouse button.
Right mouse button	By pushing the right mouse button, the player invokes spell casting.
Mouse wheel	Mouse wheel can be used to zoom in/zoom out.

A.2.2 Spells

The spells in the game are represented by totems. When the player casts a spell, a totem is created with corresponding effects on it, described by the rules below. The game supports four shapes: circle, square, triangle and water drop. Each of these shapes maps to a different effect, based on its position in the drawing. We distinguish between 3 positions of shapes: the root position, the embedded shape and the pattern shape.

- The root position is assigned to the biggest shape, i.e. the shape that forms the drawing.
- The embedded shape is a shape that is located in the center of another shape.

- The pattern shape is a shape that forms a composition of another shape.

Relations between the shape position, shape and effect are described by the following rules:

The root shape mappings:

Circle The *Shield* effect A.1 is added to the totem. If the players character is inside the area of effect of this totem, its shield is slowly recharged.

Square The *Fire* effect A.2 is added to the totem. The fire effect causes all enemies near the totem to burn, decreasing their health periodically.

Triangle The *Freezing* A.3 effect is added to the totem. The freezing effect causes the enemies to move slowly.

Water drop The *Healing* A.4 effect is added to the totem. When the players character is nearby, it is healed over time.

The embedded shape mappings:

Circle The *Reach* effect A.5 is added to the totem, extending its area of effect twice.

Square The *Power* effect A.6 is added to the totem. The power effect causes all the attacks of the player do substantially more damage to the enemies.

Triangle The *Durability* effect is added to the totem. The durability effect extends the duration of the totem twice.

Water drop The *Distraction* effect A.7 is added to the totem. Enemies under the distraction effect start attacking the totem instead of the player.

The pattern shape mappings:

Circle The *DefenseWall* effect A.8 is added to the totem. When the totem with the defense wall is created, it creates a wall around it.

Square The *Explosions* effect A.9 is added to the totem. This effect causes the enemies to explode, slowing them and reducing their health.

Triangle The *Tower* effect A.10 is added to the totem. The totem with the tower effect starts shooting at any enemy that appears in its area of effect.

Water drop	The *Madness* effect is added to the totem. Enemies under the madness effect start attacking randomly each other.[1]

[1] While this effect is supported in theory, in reality it is nearly impossible to draw, since the pattern shape is always classified as an circle rather than a water drop, due to the imperfections of the algorithm.

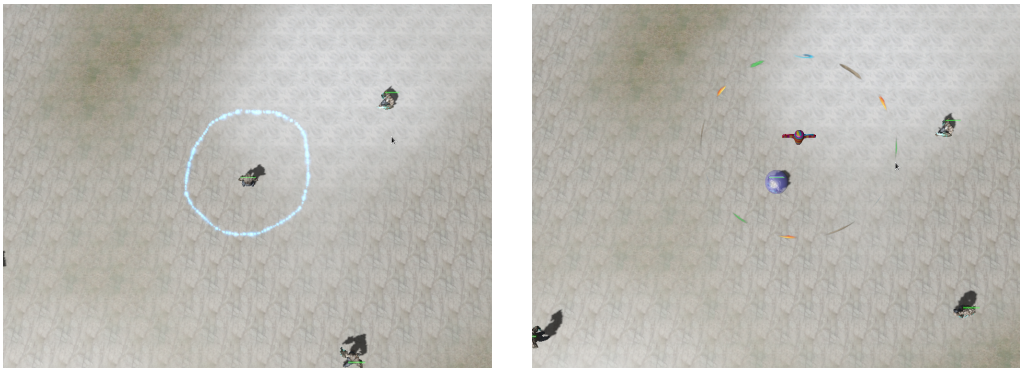

Figure A.1: The figures show a drawn circle and the created *Shield* effect totem. The player's character has a shield recharging on it.

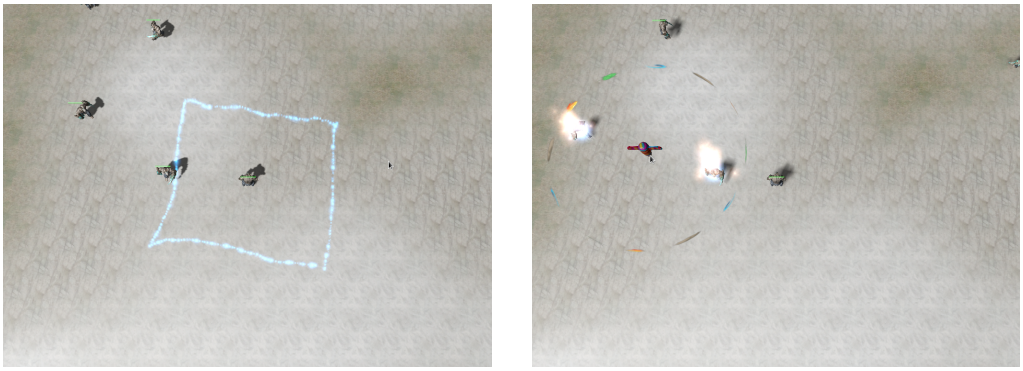

Figure A.2: The figures show a drawn square and the created *Fire* effect totem. The enemies in the area of effect of the totem are burning.

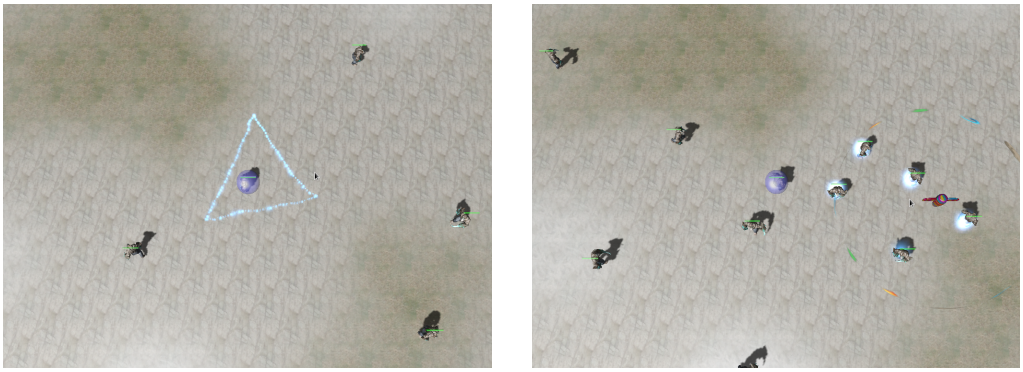

Figure A.3: The figures show a drawn triangle and the created *Freezing* effect totem. The enemies in the area of effect of the totem are slower.

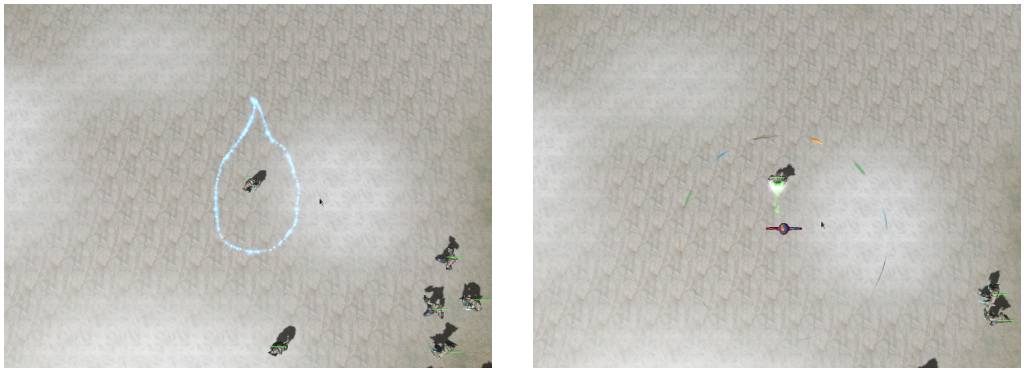

Figure A.4: The figures show a drawn water drop and the created *Healing* effect totem. The player's character is healed in the presence of the totem.

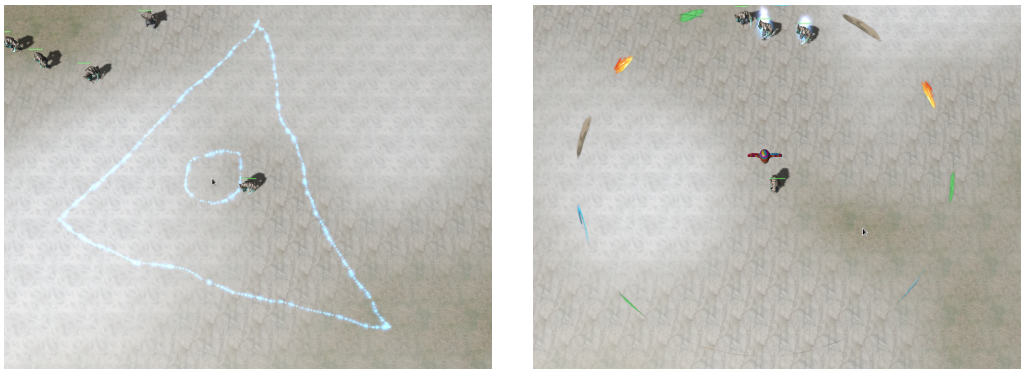

Figure A.5: The first figure shows a triangle with an embedded circle. The second figure shows the created totem has the are of effect increased twice.

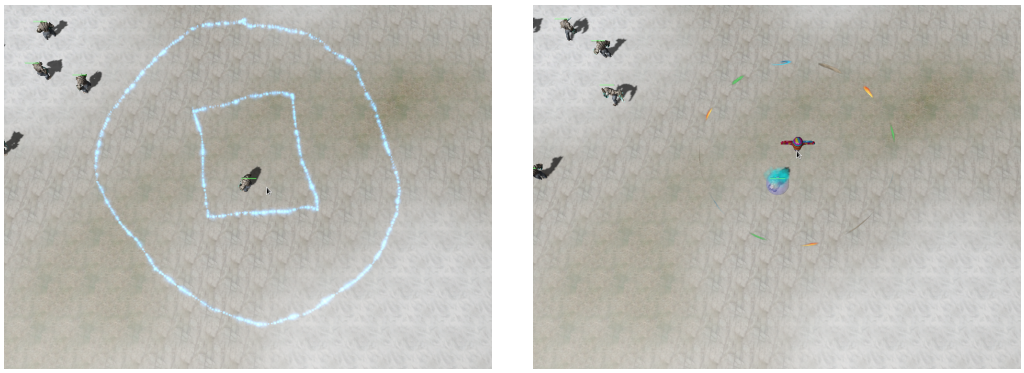

Figure A.6: The first figure shows a circle with an embedded square. The second figure shows the created totem shields the character and grants it power state, increasing its attack.

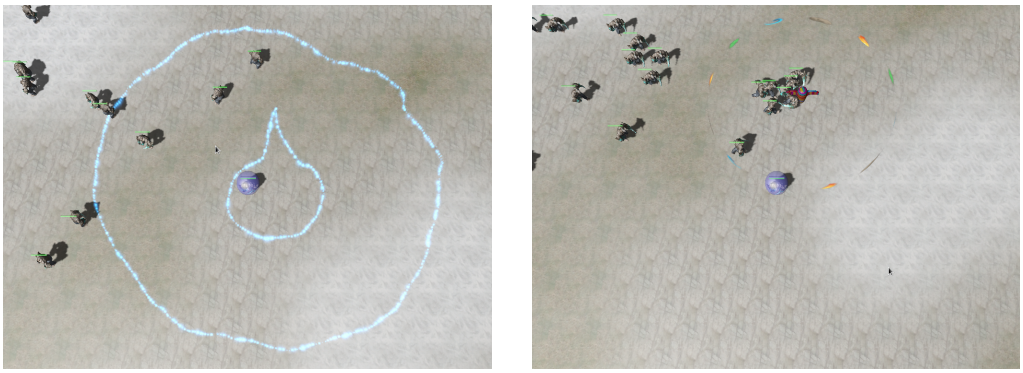

Figure A.7: The first figure shows a circle with an embedded water drop. The second figure shows the created totem shields the character and its *Distraction* effect lures the enemies.

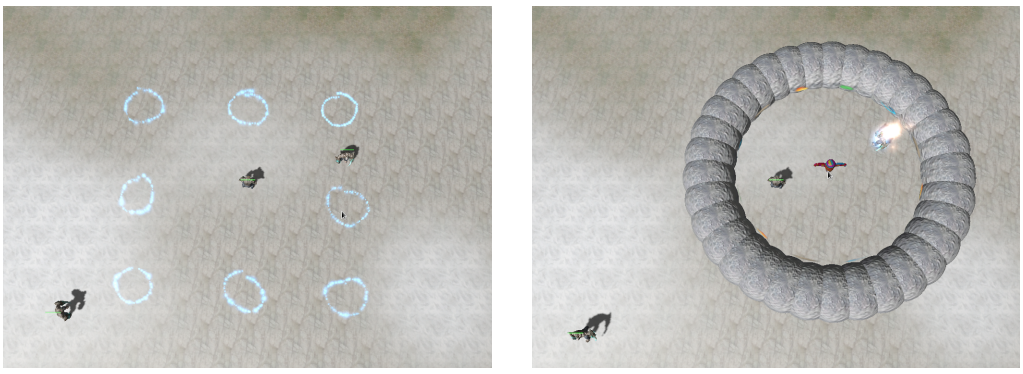

Figure A.8: The first figure shows a square composed from circle pattern shapes. The second figure shows the *WallEffect* caused by the circle pattern shape.

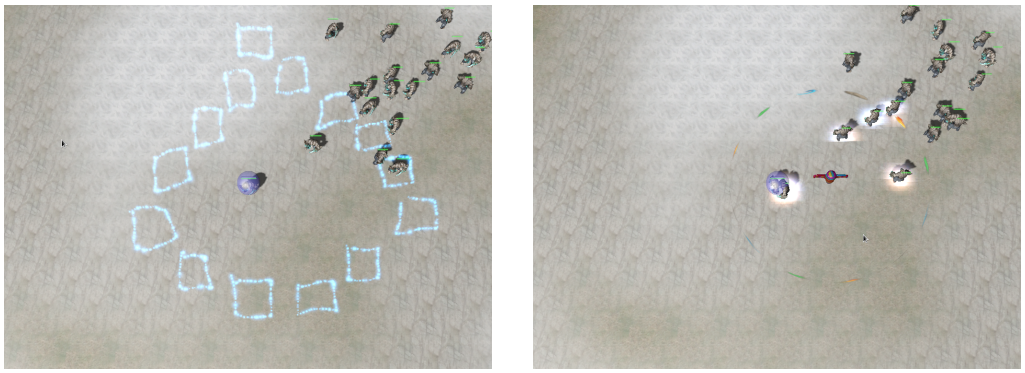

Figure A.9: The first figure shows a water drop composed from square pattern shapes. The second figure shows an explosions effect caused by the square pattern shape.

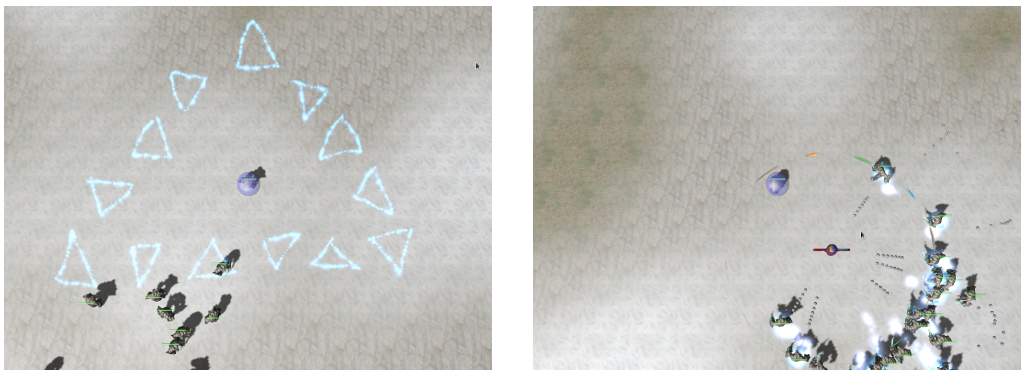

Figure A.10: The first figure shows a triangle composed from triangle pattern shapes. The second figure shows the *Tower* effect caused by the triangle pattern shape.

B. Third party software

The following libraries and projects has been used in the implementation:

Urho3D **[Urh]** Urho3D is a game engine and the game prototype has been implemented in it.

Fast Artificial Neural Network Library **[Fan]** The algorithm is built around the *Fast Artificial Neural Network Library* [Fan], using its simple interface to train and use the neural networks.

EasyBMP **[Bmp]** The *EasyBMP* [Bmp] has been used to save the debug images of the algorithm to the hard disk in the `bmp` format.

linalg.h **[Lin]** The *linalg.h* [Lin] is a library for linear algebra computations, which has been used for several matrix transformations in the algorithm.

Attachments

Attachment A — the Enclosed CD

On the CD attached to this thesis we enclose the source codes of the implemented software together with the source codes of its dependencies *Urho3D* [Urh] engine and *Fast Artificial Neural Network Library* [Fan].

The electronic version of this thesis is also enclosed.