An overview of the implementation of Level Set methods, including the use of the Narrow Band method

Jonas Larsen
Thomas Greve Kristensen

December 9, 2005

1 Introduction

Level set methods are mathematical tools for transforming surfaces. These surfaces are described by a signed-distance-function, that given a point returns the distance to the surface. The surface separates the inside and the outside of some object, it is therefore often referred to as the *interface*. On a computer, one stores an implicit representation of the interface. That is, we store a value for each pixel, representing the distance from that pixel to the surface. Inside the object, this distance is negative, and outside it is positive, see figure 1 for a small example, with a circle with unit radius, centred in a 3×3 grid.

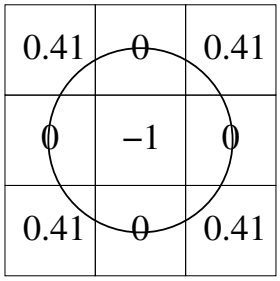

Figure 1: A simple circle, and its representation

The set of pixels in which the distance is 0, is called the *iso-surface*[KMJ]. That is, the iso-surface is the set S, where $S = \{(x, y) \in R^2 | \phi(x, y) = 0\}$.

The main idea is to let the values in the grid depend on time as a factor. That is, we let $\phi(x, y, t)$ detonate the distance from the surface in the point $(x, y) \in R^2$ at the time $t \in R^+$.

This adds a time dependent factor to the definition of the iso-surfaces. We therefore define the *level set formulation* of the iso-surface as being $S(t) = \{(x, y) \in R^2 | \phi(x, y, t) = 0\}$.

Assume we are stuck in time with a signed distance function. We are interested in knowing the new values in each of our points, if we were to move Δt forward in time. That is, we are interested in knowing what ϕ_{new} is, knowing ϕ_{old}.

The important thing to notice is, that if we can, by some insight, approximate $\frac{\partial \phi}{\partial t}$ in a point, we can calculate the new value by using Euler. Euler states that if $\frac{\phi_{new}(x_0,y_0) - \phi_{old}(x_0,y_0)}{\Delta t} \approx H(x_0, y_0)$ we must have that $\phi_{new}(x_0, y_0) \approx \phi_{old}(x_0, y_0) + \Delta t \cdot H(x_0, y_0)$.

Due to some mathematical voodoo, we actually know how to compute $\frac{\partial \phi}{\partial t}$ in our grid representation. What this voodoo is, will be covered later.

The calculation of H is called *solving the level set equation*.

In this paper we will be working with two main types of H, namely one type that advects a level set in the direction of a given vector field, and one that advects the level set in the direction of the normals with a given speed.

Picking a surface distance at time t_0 and solving equations thus allows for the calculation of the surface at a later timestep t_1. Every time we calculate this step forward in time, we risk corrupting our ϕ matrix, so that it is no longer descriping a signed distance function. It is therefore vital to re-normalize it to a signed distance function. How this is done in practice is covered later on.

The first part of this paper will cover how we solve a level set equation, how we reinitialize our signed distance function representation and what the Narrow band method is all about.

The second part of the paper will consider itself with our basic framework for working with level sets. This can be seen as a guide to how to write a level set solver, or at least how we have done it.

The third part descripes applications of our framework. In this part we will descripe how to generete exciting ϕ matrixes and how to perform the binary operations union, intersection and minus. We will look at different speed functions, including a speed function for morphing between different ϕ. Finally we will look at the detection of edges in images.

1.1 Solving

1.1.1 Advection by vector field

A possible solution to the equation $\frac{\partial \phi}{\partial t} = H$ is

$$H = -V \cdot \nabla \phi \tag{1}$$

where $\nabla \phi = (\frac{\partial \phi}{\partial x}, \frac{\partial \phi}{\partial y})$ and V is a vector field.

Solving the level set equation thus amounts to moving the interface in the direction of a vector field. This field can depend on a set of variables.

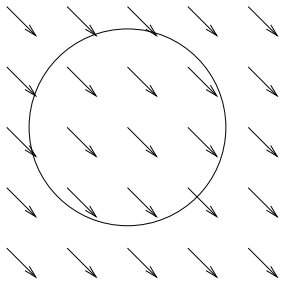

Figure 2: A simple vector field

As a simple example, consider the vector field only consisting of vectors of the form $(\frac{1}{2}; \frac{1}{2})$. This simple field moves a given interface downwards to the right. Vector fields can also be used to detect edges, as described later in this paper.

The idea is to pick a vector field and for each coordinate calculate $\nabla\phi$, dot this with V and see where it takes you. The main problem is, of course, to calculate $\nabla\phi = (\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y})$, from our grid representation.

This is where magical calculus kicks in. We can approximate this value in a point, using the upwind scheme. Given a point $(x, y) \in R^2$ and a vector $v = (v_x, v_y) = V(x, y)$, the upwind scheme dictates that

$$\frac{\partial\phi}{\partial x} \approx \begin{cases} \phi_x^- = \phi(x+1, y) - \phi(x, y) & if\ v_x < 0 \\ \phi_x^- = \phi(x, y) - \phi(x-1, y) & if\ v_x > 0 \end{cases} \tag{2}$$

...and the same for $\frac{\partial\phi}{\partial y}$. This gives us a usable gradient for each point. All that is left to do, is using the Euler method as mentioned previously, and voila.

1.1.2 Advection in the normal direction

A special case of advection by vector field is advection in normal direction. The idea is, given a point, to pick its normal and scale it with a speed a, and use this as a vector field.

As an example, consider the case where we chose a positive scalar to scale **all** the points with. This will expand the surface from the middle and outwards, e.g. expanding a circle. If we choose a negative scalar instead, we would collapse the surface.

The speed in a point can be dependent of a set of variables. In the remaining part of this paper, the calculations of the speed will be based on functions called *speed functions* or Γ functions, which can take a variety of parameters.

Assume we are given a point $(x, y) \in R^2$ and the speed a in the point. The vector v we are going to use would then be defined as $v = a \cdot n$ where $n = \frac{\nabla\phi}{|\nabla\phi|}$. That is, the normal in a point is the gradient scaled to unit length, as known from linear algebra.

If we use this v in the definition of H from advection by vector field we get

$$H = -v \cdot \nabla\phi = -a \cdot n \cdot \nabla\phi = -a \cdot \frac{\nabla\phi}{|\nabla\phi|} \cdot \nabla\phi = -a \cdot \frac{|\nabla\phi|^2}{|\nabla\phi|} = -a \cdot |\nabla\phi| \tag{3}$$

As you see, we now only need to find the length of the gradient. One way of doing this, is using equation 2, but this is not as precise as *Godunovs method*. This method dictates how to find $(\frac{\partial\phi}{\partial x})^2$ and $(\frac{\partial\phi}{\partial y})^2$, but it can not be used to find $\frac{\partial\phi}{\partial x}$ or $\frac{\partial\phi}{\partial y}$, which is obviously the reason why not to use it in the general vector field case...

When $(\frac{\partial\phi}{\partial x})^2$ and $(\frac{\partial\phi}{\partial y})^2$ are calculated, they can be used to find $|\nabla\phi|$. With this we can find $\frac{\partial\phi}{\partial t} \approx -a \cdot |\nabla\phi|$ which then again can be used to find ϕ_{new}, by using the Euler method.

So how do we calculate $(\frac{\partial\phi}{\partial x})^2$ and $(\frac{\partial\phi}{\partial y})^2$? Godunov dictates that

$$(\frac{\partial\phi}{\partial x})^2 \approx \begin{cases} max(max(\phi_x^-, 0)^2, min(\phi_x^+, 0)^2) & if\ a < 0 \\ max(min(\phi_x^-, 0)^2, max(\phi_x^+, 0)^2) & if\ a > 0 \end{cases} \tag{4}$$

where ϕ_x^- and ϕ_x^+ are defined as in (2).

That's all there's to it!

1.1.3 An upper limit to the size of Δt

The Courant-Friedrichs-Lewy (CFL) stability condition found in [KMJ] dictates, that the numerical wave must move at least as fast as the level set. This gives us an upper bound on Δt, and hence the step sizes we can perform. If we therefore would like to move our level set surface 10 time steps into the future, we might have to do it in e.g. 17 steps, each step with a different size.

But how does one calculate the maximum legal time step? When advecting by a vector field, one simple looks at all the vectors to find the one that moves the interface the most. The maximum time step Δt will then be one over the length of this vector, or to put it at bit more precise

$$\Delta t < \frac{1}{max\{|v|\}} \tag{5}$$

When advecting in the normal direction, this amounts to setting v to $a \cdot n$, and using this to find the maximum time step. We can find Δt using the following equation

$$\Delta t < \frac{1}{max\{|a \cdot n|\}} \tag{6}$$

A very strict way of making sure not to pick a too big step is to use the number of dimensions as a rough measure of how long away the speed function will take us. Since we are working in two dimensions, this gives us

$$\Delta t < \frac{1}{max\{2 \cdot |a|\}} \tag{7}$$

1.2 Re-normalisation

A signed distance function ϕ has the property, that its length $|\phi|$ is close to one at all time. This, however, is not always the case when we solve level set equations. There is nothing in the methods described above that guarantees this.

What is needed is a so-called *re-normalisation*. In our implementation, this is done by solving the *Eikonal Equation* $\frac{\partial \phi}{\partial t} = Sign(\phi)(1 - |\nabla \phi|)$ where $Sign$ is a continuous sign function[1]. In our implementation $Sign(\phi) = \frac{\phi}{\sqrt{\phi^2 + |\nabla \phi|}}$.

If you examine this equation, you will notice that it is similar to the equation for advection in the normal direction. Therefore, the same method can be used for solving it! We simply use $-Sign(\phi)$ as the speed a, and solve it using Godunov and Euler.

This equation should be solved enough times to ensure that the system is reasonable stabile around the interface. The idea is, that if this is done for all

[1] If it was not, we would have no method for solving it

points, it will give the points just beside the interface a reasonable value. In the next iteration the next layer will have a reasonable value and so on, and so on.

1.3 The Narrow Band method

Imagine you have to expand a very small interface in a big big world. If you were to do this by hand, you would probably not calculate the values out on the border of the world, since they will not be used in the nearest future. In the same matter, you would no longer calculate the distance in the points you have already passed, which you are no longer interested in.

The *Narrow band method* is invented by a man as lazy as you, and is described in [ea99]. The idea is, only to solve the level set equation in a narrow band γ pixels wide around the interface[2].

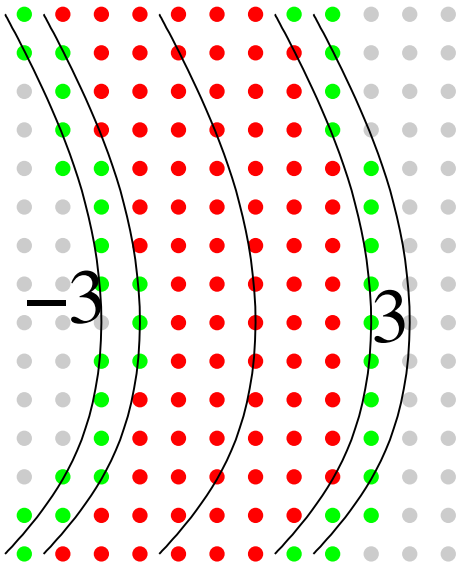

Figure 3: A narrow band

All values outside this band are assumed to have a value of $\pm\gamma$, until they are close enough for the gamma-band to suck them in. To ensure this we use a so-called safe-tube. The safe-tube is defined as all the points that are not in the gamma-band, but who do have a neighbour that is. When a point enters this magical safe-tube, its value is set to $\pm\gamma$, depending on the sign of its lucky neighbour, who is small enough to get into the gamma-band. The gamma-band together with the safe tube is called the narrow band.

Please notice, that this ensures that the value in the point can be used by e.g. the Godunov scheme to calculate the length of the gradient. It might not

[2]Hence the name

be precise, but it is precise enough[3], and after the re-normalisation it will be even more precise.

As you might have guessed, or already know, we only need to re-normalise as many times as this gamma-band is wide, divided by the maximum step size in our CFL condition. In our case, we have chosen a γ of three.

If we start out with such a gamma-band and a safe-tube, we first solve our level set equation in the gamma-band. When we are done, we re-normalise in the gamma-band. We can now estimate the values in the safe-band better, and the values in it is therefore updated. How this is done is described in the next section.

We now have to rebuild our gamma-band and safe-tube. This can be done in a naive way, and in a smart way. We have obviously chosen the latter, and how it is done, is described in section 2.3.n

[3] . . . after all, we are working on finite arithmetics

2 Framework

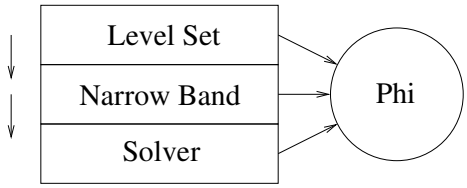

Figure 4: The general flow in the program

In general, the framework is implemented in three main layers. In the top, we have the level set, which uses the lower level to perform the desired actions in the correct order. In the next layer, we have the narrow band, which stores all the points, and manage which points are to be manipulated. In the bottom layer, we have the solvers. There is a solver for advection by vector field, and a solver for advection in normal direction. The solvers solve the level set equation in the given points, and return the result. This result can then be used by the narrow-band to update the values in the gamma-band.

2.1 Phi

ϕ holds distances to the interface, recall that we only sample in a finite number of grid points. ϕ give access to manipulate the value in the individual grid points.

A ϕ can be asked to renormalise itself, using the same methods as the narrow band.

The specialised versions of ϕ is described in 3.1.

Implementation The distances is held in a 2D array. The only class that uses `renormalize` is the `ImagePhi`. Details about the implementation of `renormalize` can be found in the description of the narrow band.

2.2 Level set

The main part of the level set method is to repeatedly transform the pixels in the level set. This is done as described in 1 by first solving the level set equation, this is done by asking the narrow band to apply the solver on all pixels in the γ-band. Secondly the narrow band is asked re-normalise all the pixels in the γ-band and in the safe tube. This is done by first re-normalising the pixels in the γ-band a number of times, the goal of the re-normalisation is to turn ϕ into a signed distance function. We re-normalise three times using the maximum

time step[4] , this is done to propagate the changes through the entire γ-band. Since the re-normalisation moves pixels in the γ-band closer to the interface, it is necessary to rebuild the safe tube. This is done by using Manhattan city distance.

Finally the narrow band is asked to rebuild itself.

The level set is responsible for doing morphologic operations, opening and closing, as well as different ways of moving the level set forward in time.

2.2.1 Implementation

The level set has a pointer to the narrow band and ϕ.

Safe translation Since we cannot make time steps larger than our CFL condition, we have to split a long steps into several smaller steps.

Opening The idea of opening is to first contract the interface for a certain amount of time, and afterwards expand the interface for the same amount of time. Opening removes bumps and spikes on the interface.

```
Solver oldSolver = nb.getSolver();
Solver normalDirectionSolver =
       new NormalDirectionSolver(new ContractingSpeedFunction());
nb.setSolver(normalDirectionSolver);
safeTranslationInTime(d);

normalDirectionSolver =
       new NormalDirectionSolver(new ExpandingSpeedFunction());
nb.setSolver(normalDirectionSolver);
safeTranslationInTime(d);

nb.setSolver(oldSolver);
```

Closing Closing of holes in interfaces is doing the same operations as opening, but in reverse order.

2.3 Narrow band

The narrow method is described in section 1.3. The narrow band is responsible for doing different translations in all these pixels that are close enough to the interface for us to be concerned about them. One of the main jobs of the narrow band is to keep this collection of pixels updated along with the movement of the interface.

[4]It does not make any sense at all to re-normalise three times using the maximum time step. However, it works! This is probably because small time steps are followed by small time steps, and thus the interface is not moving enough for the inaccurate values to be a problem. Instead we should use another CFL condition.

Since the narrow band has information about all the interesting pixels in the level set, computations that involve all pixels are send through the narrow band.

Implementation The narrow band is represented as two lists, one holding the x values and one holding the y values. To decide if a pixel is in the γ-band or in the safe tube we use a integer matrix. The values in the matrix are 0, 1 and 2. 0 meaning outside the narrow band, 1 meaning in the safe tube and 2 meaning that the pixel is in the γ-band. The mask can also be implemented as a list. The only reason we use a matrix is that our first implementation used it.

2.3.1 Solve

Asks the solver to transform all the pixels in the γ band, one by one.

Implementation We iterate through all the pixels in the narrow band and ask the solver for the new distance of those pixel with mask 2. All these new distances are stored in a new ϕ.

2.3.2 Find the maximum time step

If a time step causes the change in distance of a pixel to be greater than the grid size, will make the level set unstable, it is necessary to make a prediction on the maximum time step. The maximum time step is calculated by finding the reciprocal of the maximum advection over all pixels in the γ-band. The narrow band ask the solver for the advection of each pixel in the γ-band.

Implementation The solver iterates over all pixel in the narrow band and ask the solver for the advection of those with mask = 2. The maximum time step is calculated as:

$$\frac{1}{advection_{max} \cdot 2} \tag{8}$$

2.3.3 Re-normalisation

2.3.4 Rebuild bands

Rebuilding the bands is done in two stages: First the γ-band is rebuild, and then a new safe tube is added.

2.3.5 Implementation

We have chosen to implement the rebuilding of the narrow band in what we thought was the most intuitive way. By construction of the narrow band, all pixels in the new γ-band are already in the old narrow band. To construct the new γ-band we simply run through the pixels in the old narrow band and add those whose updated distance is smaller than γ to the γ-band. The new

members of the γ-band are all prior members of the safe tube that was moved within γ-distance when updating the city distance. To make a new safe tube, we run through the new γ-band and remember all the neighbour pixels that are not already in the γ-band. Notice that all new pixels in the safe tube are either neighbours of pixels from the old safe tube, that entered the γ-band, or pixels that just moved out of the γ-band.

2.4 Solver

The solver is responsible for doing operations on a single given pixel. These operations include finding the length of the gradient, finding the advection and solving the level set equation.

We have done two types of solvers; a normal direction solver and a vector field solver. They have the same functionality, but the computations differ quite a bit as described in 1.1. The vector field solver depend on actually finding the the gradient, whereas the normal direction solver only uses the length of the gradient.

2.4.1 Finding the gradient

The upwind finite difference scheme is used to find the gradient.

2.4.2 Finding the length of the gradient

The normal direction solver finds the gradient by using Godunovs method. Since the vector field solver knows how to find the gradient, all it has to do is compute it and find the length.

Implementation Normal direction: Godunovs method is implemented pretty straight forward. It uses the speed function to find a.
Vector field: The length of the gradient, a vector, is computed.

2.4.3 Advection

The solver can tell us how far a a pixel will move if $t = 1$ is used.

$$|\phi_{new}(x, y) - \phi_{old}| = |\Gamma(x, y)| \cdot |\nabla\phi(x, y)| \tag{9}$$

Implementation The implementation is straightforward. the only tricky part is, that do not solve the equation at the border of the grid, as this would read values outside the grid.

2.4.4 Solve - Vector field

To find the change in distance of the pixel by using the vector field methods, you can simply dot the gradient with the vector from the vector field and multiply

it by t. Updating a pixel is therefore done in the following way:

$$\phi_{new}(x,y) = \phi_{old}(x,y) - t \cdot \overrightarrow{\nabla\phi_{old}(x,y)} \bullet \overrightarrow{V(x,y)} \tag{10}$$

Implementation The following three lines are all it takes to find the new distance in a pixel (x, y).

```
float[] grad = gradient(phi, x, y);
float[] v = V.getVector(x, y, phi);
return (phi.get(x, y) - t*(grad[0]*v[0] + grad[1]*v[1]));
```

2.4.5 Solve - Normal direction

The normal direction solver is moving a pixel in the direction of the gradient. The distance to move is given by the speed function multiplied by t.

Implementation To move a pixel the normal direction solver first calculate the norm of the gradient. Secondly the speed function is asked for the speed in that pixel. Since a positive value means that the pixel is moving closer to the interface, we use the follow equation to find the new distances.

$$\phi_{new}(x,y) = \phi_{old}(x,y) - \Gamma(x,y) \cdot |\nabla\phi(x,y)| \cdot t \tag{11}$$

2.4.6 Speed function

The basics of speed functions are described in 1.1.2. The speed functions are used by the normal directions solver to decide the speed of change in a pixel. More specific speed functions are described later on.

Implementation We have chosen to let the all speed functions implement the following interface:

```
public float getSpeed(int x, int y, Phi phi);
```

That means that we cannot let the speed depend on parameters like time. We would probably make that more flexible if we were to reimplement the code.

3 Applications

There are many different applications of the level set method. In this part of the paper, we will briefly describe some of the more usable methods and our implementation of them in our framework.

3.1 Phi functions

The grids representing our level sets have to be initialised somehow. We have used two primary initialisation procedures, which we will now cover briefly.

3.1.1 Circle

The simplest of forms to represent in a level set. As you very well know, a point (x_1, y_1) has the distance $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$ to the point (x_0, y_0), and if we define a circle with centre in (x_0, y_0) and with radius r, the distance from a point (x_1, y_1) to the circle is $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2} - r$. Please notice, that this means, that the points **inside** the circle have a negative distance, as we desire.

Implementation As mentioned earlier, our ϕ functions are implemented as a grid containing the distances for every pixel. We let the class `CirclePhi` extend the ordinary `Phi` class, but change the constructor slightly, so that it calculate the grid distances. To initialise this grid, we simply run over all the coordinates, and calculate the distances.

```
public CirclePhi(int width, int height, float radius,
                 float centerX, float centerY)
{
 super(width, height);
 for(int x = 0; x < width; x++)
  for(int y = 0; y < height; y++)
  {
   float length = (float) Math.sqrt((x - centerX)*(x - centerX) +
                                    (y - centerY)*(y - centerY));
   grid[x][y] = length - radius;
  }
}
```

3.1.2 Loading a raw image

First of all, it is important to know what we mean by a "raw image". A raw image, is an image where everything that is not a part of the shape is coloured white, and everything that is a part of the shape is coloured in some other colour.

Converting these images to a signed distance function is a bit more tricky than making, say, a circle. But it is in no way impossible.

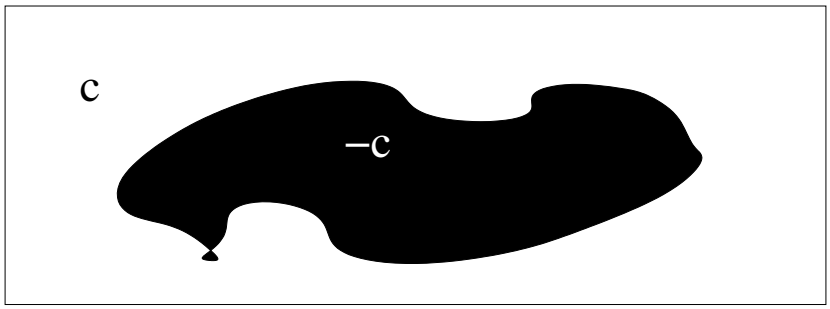

Figure 5: The image initialised with the maximum distances

Implementation This is implemented in the class `ImagePhi`. The first thing we do, is initialise every point of the grid with the greatest possible distance in the image c. c is the length from the upper left corner to the lower right corner.

Next we set the distance in all the points inside the shape described by the raw image to $-c$.

Then we re-normalise in all the points 30 times. This has the effect, that the distances around the edges are probably calculated. Remember that re-normalisation ensures, that `Phi` represents a signed distance function, with proper distances.

And that is it, there is actually nothing more to it. Here is the code for calculating the extreme grid

```
public ImagePhi(File file, int width, int height)
{
 super(width, height);
 BufferedImage image = null;
 try{ image = ImageIO.read(file); }
 catch(IOException ex)
 {
  System.out.println("Filen " + file + " findes ikke");
  System.exit(0);
 }
 float c = (float) Math.sqrt(width * width + height * height);
 int w = Math.min(image.getWidth(), width);
 int h = Math.min(image.getHeight(), height);
 for(int x = 0; x < w; x++)
  for(int y = 0; y < h; y++)
  {
   int farve = image.getRGB(x, y);
   if(farve == -1) grid[x][y] = c;
   else grid[x][y] = -c;
  }
 reinitialize(width, height, 30);
```

```
}
```

The code for reinitialisation is stored in `Phi`, as it might come in handy in other situations. The most important thing to notice is, that the reinitialisation uses the sign in the grid to calculate the norm using Godunov.

3.2 Boolean operations

Given two matrixes, each representing its own ϕ function, it is extremely easy to perform binary operations like union, difference and intersection. These operations can be expressed locally in each point by looking at how long the points are from the interfaces.

Given two signed distance functions ϕ_A and ϕ_B, the following equations demonstrates how to combine them.

- Union: $\phi_\cup(x_0, y_0) = min(\phi_A(x_0, y_0), \phi_B(x_0, y_0))$
- Intersection: $\phi_\cap(x_0, y_0) = max(\phi_A(x_0, y_0), \phi_B(x_0, y_0))$
- Difference: $\phi_-(x_0, y_0) = max(\phi_A(x_0, y_0), -\phi_B(x_0, y_0))$

It is worth noticing, that one should rebuild the narrow band after combining the functions. It is also worth noticing, that the ϕs do **not** have to be accurate everywhere. It is a sufficient condition, that they are accurate around their narrow bands. Remember, that all the values outside the band are assumed to have numerical values greater than γ, and they are clamped to such a value upon entering the safe band anyway.

3.2.1 Implementation

The implementations are made as methods in our `Phi` super class. These methods take a `Phi` object, which it uses to change the object on which the method is invoked.

As an example, consider the following code implementing the union of two `Phi` objects.

```
public void union(Phi phi)
{
 for(int x = 0; x < getWidth(); x++)
  for(int y = 0; y < getHeight(); y++)
   if(get(x, y) > phi.get(x, y))
    grid[x][y] = phi.get(x, y);
}
```

Whenever these methods are called, one should **always** remember to rebuild the narrow band!

3.3 Simple speed functions

As previous mentioned, speed functions, used to advect an interface in the normal direction, can depend on any number of parameters. There are, however, two extremely simple speed functions. These are the speed function for expanding **any** given interface, and the speed function for collapsing **any** given interface.

The expanding speed function is simply 1 at all time, in any point. This will give us a constant expansion along the normals.

Similarly the collapsing speed function is simply -1 at all time, in all points.

3.3.1 Implementation

The implementation is very simple. Here is the code for getting the speed in a point when we want to expand the interface

```
public float getSpeed(int x, int y, Phi phi) { return 1; }
```

3.4 Morphing

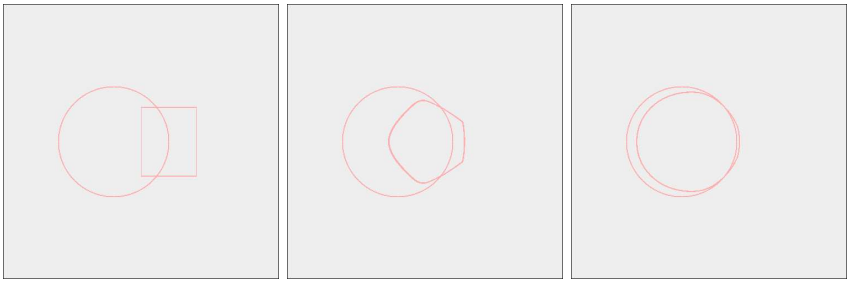

Figure 6: An example of a box morphing into a circle

A very cool effect in computer graphics is the morphing of objects from one shape to another. This can very easy be achieved in level sets, using advection in the normal direction. If we have two **overlapping** `Phi` objects, we can define a speed function from the primer to the latter by

$$\frac{\partial \phi}{\partial t} = (\phi - \phi_{target} \cdot |\nabla \phi|) \tag{12}$$

It is worth noticing we have changed the sign, as compared to [KMJ], the reason being we have a negative sign inside the interface, and a positive sign outside.

3.4.1 Implementation

Morphing is just implemented as a fancy speed function. This speed function needs to know ϕ_{target} at all points. Please notice that ϕ_{target} has to stay a signed distance function in all points at all time.

The speed depends on ϕ_{target}, and the ϕ we are moving. `target` is therefore given at the construction time of the object.

```
public class MorphingSpeedFunction implements SpeedFunction {
 Phi target;
 public MorphingSpeedFunction(Phi target) { this.target = target; }
 public float getSpeed(int x, int y, Phi phi)
 { return phi.get(x, y) - target.get(x, y); }
}
```

3.5 Morphological operations

When processing real life data, one is often faced with uneven interfaces or surfaces with holes and pikes in them, where there should be none.

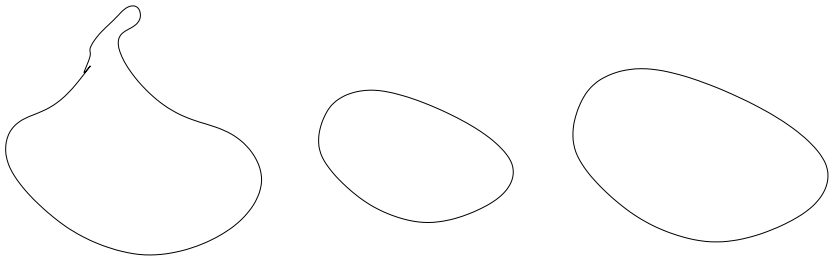

Figure 7: Removing a pike

Holes in surfaces can be removed by expanding the interface in the normal direction forward in time, and then collapsing it again. This operation is called a *morphological closing*.

Pikes out of surfaces can be removed by collapsing the surface, and then expanding it again. This operation is called a *morphological opening*.

3.5.1 Implementation

The idea is to translate the interface forward in time using one of the two simple speed functions and advection in the normal direction. How far into time we are going to translate it, differs from data set to data set, but we can not always make the translations in big steps. Since we are restricted by a CFL condition, we must take many small steps.

This is easily implemented. We have chosen to implement it in our `Level Set` class.

```
public void safeTranslationInTime(float time)
{
 while(time > 0)
 {
  float maxTimeStep = nb.maxTimeStep(phi);
  step(Math.min(time, maxTimeStep));
  time -= maxTimeStep;
 }
}
```

This translation method can then be used to perform a morphological opening on the surface. This is also implemented in the `Level Set` class.

```
public void opening(float d)
{
 Solver oldSolver = nb.getSolver();
 Solver normalDirectionSolver =
  new NormalDirectionSolver(new ProportionalSpeedFunction());
 nb.setSolver(normalDirectionSolver);
 safeTranslationInTime(d);
 normalDirectionSolver =
  new NormalDirectionSolver(new ExpandingSpeedFunction());
 nb.setSolver(normalDirectionSolver);
 safeTranslationInTime(d);
 nb.setSolver(oldSolver);
}
```

The morphological closing implementation is completely similar.

3.6 Canny edge detection

Another problem with real life data is, that it is often filled with noise and gaps. There are level set methods for finding interfaces corresponding to shapes in these corrupted images.

We detect shapes in four steps. First we remove general noise from image. Next we find the edges of the shapes in the image. Then we calculate the distance to the clearest edge in every point of our image. Lastly, we grow a simple interface outwards, until it hits the edges, and covers the gaps, of the shape we wish to analyse.

If you have an image that is completely without noise (probably because you drew it in some drawing program) you do not need to remove noise from it, and the first step of this procedure can thus be ommited.

When the image is nice and smooth, we are ready to find the edges in the image. This is described in the implementation part of this section.

With these edges, it is easy to calculate the distance to the nearest edge from every point. We use Manhattan city distance to approximate this distance. This is also covered in the implementation.

This image can then be used to find **the direction to the nearest edge**, just by finding the derivatives in the image. These derivatives will point from the nearest edge to the point we are standing in. If we negate such a derivative, we have a vector field pointing towards the edges. We can use this vector field in our advection by vector field scheme. We simply create a figure that is smaller than the shape in the image, and place the shape inside the shape. We then advect the figure by the vector field scheme, thus growing outwards until it hit the edges, where the vector field is zero. The ingenious thing is, that around the holes in the shape, the derivatives will point in opposite directions, and pull the interface out into a straight line. Please consult figure 8 for a clarification.

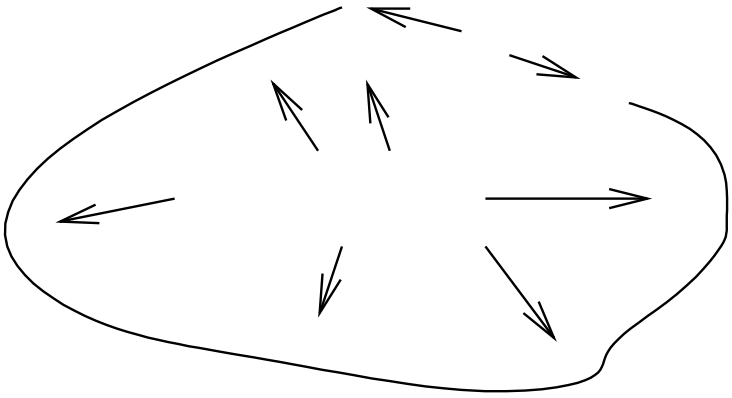

Figure 8: The derivatives in a shape with a hole in it

This has the effect, that the derivatives will be very close to zero vectors, and the interface stops moving, filling the gaps with straight lines. Brilliant, is it not?

Another strategy is creating a large interface, that you are absolutely sure covers the entire shape, and collapse it. The result is the same, unless of course, there a several disjoint shapes in you image.

3.6.1 Implementation

To get rid of the noise, one would typically use an image filter. In this project we have used a Gaussian filter as described in ([Ima02]) to clear out the noise. This filter simply makes a weighted approximation of the value in each pixel, by looking at their neighbourhood. The pixel value in each point is weighted using a kernel of size $2 \cdot k + 1 \times 2 \cdot k + 1$ where k is chosen in advance.

$$G_{i,j} = \frac{1}{2 \cdot \sigma^2 \cdot \pi} \cdot e^{\frac{(i-k-1)^2+(j-k-1)^2}{2 \cdot \sigma^2}} \tag{13}$$

and the new value in each point (x, y) is simply

$$\sum G_{i,j} \cdot I_{x-i+k,y-j+k} \tag{14}$$

where $I_{x,y}$ is the value in the original image.

As mentioned, the next step is to find the edges of the shape(s) in the image. This is done in a series of steps, as described by Michael Bang Nielsen in a mail[5]:

- Compute an image, A, where every pixel is the norm of the gradient in the smoothed image. The derivative of x in a pixel is defined as the value in the pixel to the left minus the value in the pixel to the right, divided by 2. The derivative of y is found in a similar manner.
- Compute another image, B, where every pixel is the gradient at the corresponding position in image A dotted with the normal from the corresponding position in the image.
- Compute an image, C, where every pixel is the numerical value of the gradient in the image dotted with the normal.
- Now you have to look for zero-crossings in image B. Basically a pixel is a zero crossing if any of its neighbouring pixels have a different sign, or if the value of the pixel itself is zero. A pixel is classified as an edge if it is a zero-crossing and if the value at the corresponding pixel in image C is higher than some threshold. We have chosen 10^7 as our threshold.

Please notice that we need the integer value in the pixels.

The only question that remains is, how do we compute the distance to the nearest edge in every point? We have chosen to do it the easy way, using city distance. We create a new image where all points on edges gets the value 0, and all other points get the highest possible value, that is, the distance from the top left corner to the bottom right corner. We then update all the distances using the Manhattan city distance scheme. We first update the values from the top left corner down to the bottom left. We then update them the other way. This guarantees, that all the distances are a correct approximation, in the city distance scheme.

Next we just use these distances to compute a vector field. The implementation is straight forward. Please remember, that one has to negate the vector field, so that is does not point away from the edges, but onto them.

```
public float[] getVector(int x, int y, Phi phi) {
 float dxG = (D[x + 1][y] - D[x - 1][y]) / 2;
 float dyG = (D[x][y + 1] - D[x][y - 1]) / 2;
 float[] v = new float[2];
 v[0] = -dxG;
 v[1] = -dyG;
 return v;
}
```

... easy and simple.

[5]The following is taken *directly* from that mail, and edited to explain the tricky parts

References

[ea99] Peng et. al. A pde-based fast local level set method. *Journal of Computational Physics 155*, 1999.

[Ima02] *Computer Vision - A modern Approach*. Prentice Hall, 2002.

[KMJ] Ross T. Whitaker Sean Mauch Ken Museth, David E. Breen and David Johnson. Algorithms for interactive editing of level set models.