Understanding and Implementing Level Set Methods

Department of computer science at Aarhus University

Christian P. V. Christoffersen (20050879)
cpvc@cs.au.dk

Sean Geggie (20052203)
geggie@cs.au.dk

Martin Have (20051456)
have@cs.au.dk

Peter Kristensen (20051866)
ptx@cs.au.dk

Mikkel Vester (20053229)
vester@cs.au.dk

January 22, 2010

Contents

Chapter 1

Introduction

When simulating physical phenomena one must choose a model which fits the phenomena being modeled. The level set method (LSM), is a method for modelling phenomena that can be described in terms of moving level sets. A level set in itself is a mathematical function that groups variables which have the same function value. It describes a parametric function in one dimension higher than its original domain, gaining the ability to describe more than one not colocated level set in the same structure. A two dimensional level set is known as a level curve or isocontour, which uses two dimensions to describe curves. We encounter isocontours in everyday life as pressure information in the weather forecast and terrain elevation on maps. A three dimensional level set is known as a level surface or isosurface, and is used, as suggested be it's name, to describe surfaces. In general, a level set in n dimensions is used to describe $n-1$ dimensional interfaces. Mathematically a level set is define as follows:

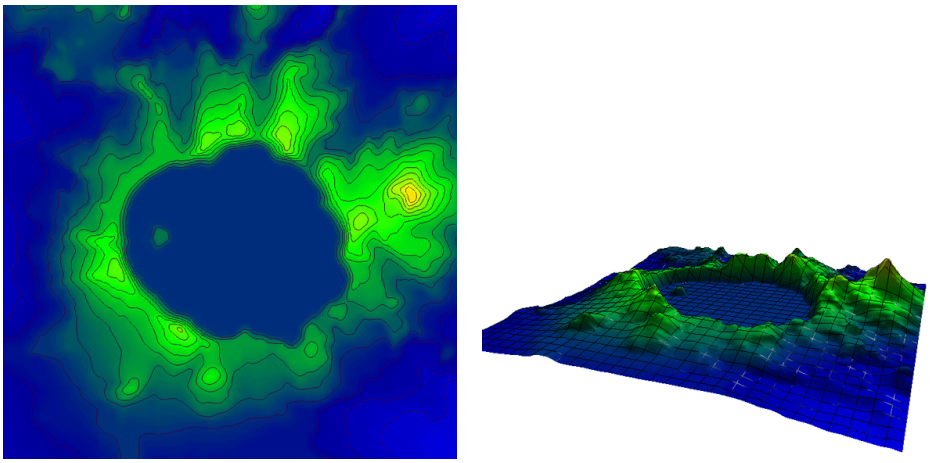
$$\{(x_1, ..., x_n)\} | f(x_1, ..., x_n) = c \tag{1.1}$$

In words this means that all values of x_i leading to a function value of c defines the level set c. The level set in its own right does not make a simulation. Unless it is evolved over times it stays the same. To evolve the level set over time, partial differential equations (PDEs) describing the physical phenomena needs to be solved. Typical phenomena modeled by the LSM includes: computational fluid dynamics (CFG), and fire and explosion simulation.

In this report, we will explain what the level set method is and what its applications are. Furthermore, we will provide code examples of how to implement the mathematical formulas.

In figure 1.1 on the following page we see two figures describing a two dimensional and three dimensional view of an island. Figure 1.1a on the next page describes an isocontour map of heights where the color indicates the height of each point. The brighter the color the higher the point. Figure 1.1b on the following page shows the corresponding 3d view.

For a level set representation there is an underlying function describing the topology. Each contour corresponds to a set of points in this function with the same value, thus we need a function which produces a field of values. In heightmaps, it would be a function from (x, y) coordinates to a height. In general, any function producing a field of values is sufficient.



(a) Two dimensional view of an isocontour map (b) Three dimensional view of figure 1.1a

Figure 1.1: Illustrations of heightmap, from: Intel Array Visualizer Gallery, [Int]

One function that has the desired characteristics is a function known as the signed distance function ϕ.

Outline of the report

In chapter 2 on the next page, we give an in depth look on the signed distance function, describe what kinds of mathematical operations we have in our toolbox and describe the important reinitialize function in 2.4 on page 8.

Chapter 3 on page 12 deals with evolving the interface. In 3.1 on page 15 we describe how to use an external velocity field, then in 3.2 on page 15 we discuss using internally generated fields.

We have individually made a number of extensions to the basic level set implementation. These extensions should show the versatility of the method and show some practical applications. In the final part of this paper, we look at these extensions. Each section has been written by the person responsible for it.

First, in chapter 4 on page 19 Mikkel Vester presents a technique called "narrow band" which increases efficiency by limiting the area of calculations. Then, Peter Kristensen talks in chapter 5 on page 22 about his use of GPU multi-kernel programming to significantly improve performance.

The final two chapters present practical applications of level sets. Martin Have has investigated and implemented a way of segmenting image data using level sets. He writes about his results in chapter 6 on page 27. In chapter 7 on page 32 Sean Geggie presents how to use level set methods for evolving an interface to write a rudimentary simulator of incompressible fluids.

Chapter 2

Signed distance field

In a level set context, the signed distance function is essentially a measure of, for each point in the domain how far that point is from the zero isocontour.

Since this function produces the straightforward euclidian distance, it increases linearly.

In order to use an implicit representation of the surface we use a signed distance field as the underlying function to the level set. We define the zero isocontour of this function to be the surface.

The level set method can use implicit functions which means that the function is defined in the entire plane and not only on the surface.

The function $\phi(x, y)$ is a signed distance function in all of $\mathbb{R}^n$, in our case $\mathbb{R}^2$. A signed distance function ϕ is a function that given a point on the plane, returns the distance to the surface. We have that $\phi(x, y) > 0$ if we are outside the object and $\phi(x, y) < 0$ when we are inside the object. And last, when $\phi(x, y) = 0$ we are on the interface or iso-surface. The iso-surface separates the inside and outside. Besides indicating whether we are inside or outside an object, it also indicates how far we are from the closest point on the iso-surface which is quite handy. For a picture of the above, see figure 2.1 on the following page.

Example

A simple example is to consider a circle and its equation:

$$x^2 + y^2 = r^2$$

It is defined in all points in $\mathbb{R}^2$ and is an example of an implicit function. Given a specific radius r, the equation of a circle defines an isocontour. If $r = 5$, then the isovalue is $c = 5^2 = 25$. For all the points (x, y) that evaluate to 25 gives us the isosurface. If the value is smaller then it is inside the surface, and outside when the value is greater. See figure 2.2 on the next page.

2.1 Cartesian grid

Since a computer has finite memory we need to come up with a way to store our representation. A simple way to do this is to partition the region into a grid where each square is of equal size. Normally

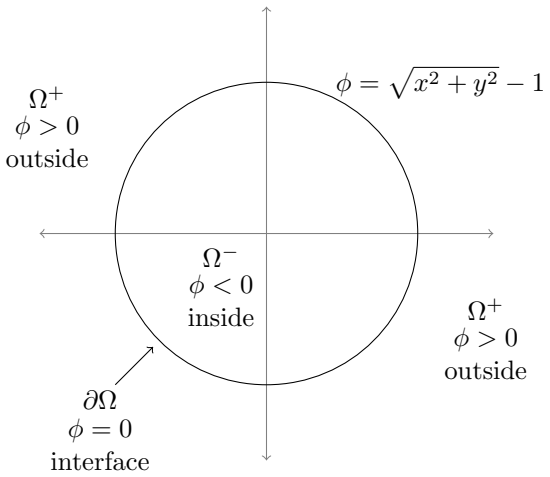


Figure 2.1: The figure is borrowed from [OF02]. A implicit function, defined in all of $\mathbb{R}^2$. We see that when we are inside the object then ϕ is less than zero, larger when we are outside and zero on the interface.

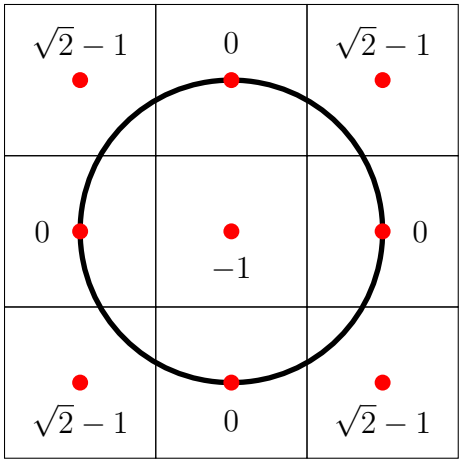


Figure 2.2: A circle, descritized into a cartesian grid. The value in each cell is the ϕ value described in this chapter.

In figure 2.2, we see how a plane has been descritized into a cartesian grid, showing a circle and the values of ϕ.

2.2 Initialization

In our solution we import the initial contours by loading a black and white image and analyse it to create the SDF matching it.

To generate the SDF we start by constructing two Cartesian grids of the same sizes as the image. One to hold the distances to the contour on the outside of the object, and one for the inside. The "outer grid" is populated with (0,0) if the pixel is dark and (∞,∞) if it's white, and opposite for the inner grid. This way each entry in the grid has a vector.

The next step is to calculate the actual distances to the iso-contour for each pixel. We first traverse the image pixel by pixel from top left corner to the lower right. For each row in the grid we go from left to right and for each pixel calculate the length of our vector in this entry against all the neighboring pixels vectors. The distance is the vector length of the vector created with the x,y values in the grid. If the distance in a neighbor, plus the offset to the current pixel, is smaller than the one already in the given pixel, it is substituted with the new and smaller one. This way we keep the 0 values inside the object, and use this value to propagate out to the pixels we visit afterwards.

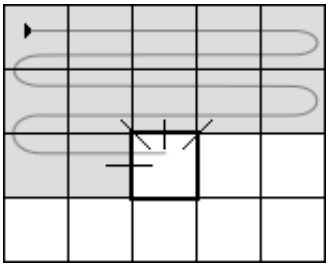


Figure 2.3: Building the Signed Distance Function table, pixel by pixel from an image.

When we reach the end of a row we go back again in the same row, and now check the same pixels, this time checking the pixel to the right of it, also filling out distances to the left of objects. The navigation in the grid can be seen in figure 2.3.

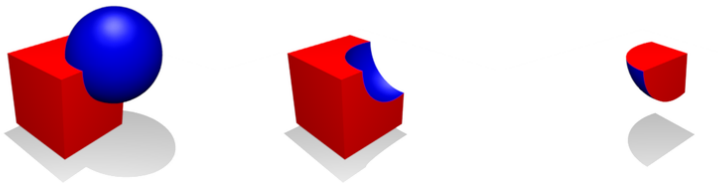
We now have distances below all objects in the image. If we now run the same pass again, just starting from the bottom and moving up, we have a vector at each pixel with the offset to the nearest contour.

Doing the same with the opposite starting values will create a distance grid with the values from inside objects in the image. Subtracting the distance of the vectors in the second grid from the first will give us an ϕ-value for each pixel, and we can now save these as our ϕ.

This SDF will have to be reinitialized numerous times afterwards to complete it, as the contour should be a smooth line instead of a 1-pixel wide line following pixels completely. This will be covered in section 2.4 on the following page. When the SDF has been reinitialized it will become as smooth as the one depicted in figure 2.2 on the previous page.

2.3 Constructive geometry operations

As our SDF is an implicitly defined, we can use simple Boolean operations on it. These functions, called Constructive Solid Geometry(CSG) can merge, subtract and find intersections. Given two different signed distance fields, these operations will generate another SDF matching the function. The three operations are shown in figure 2.4 and are explained below.



(a) Union (b) Subtract (c) Intersection

Figure 2.4: The three different CSG operations in three dimensions. Images borrowed from [oC]

Union The union Boolean operation merges two interfaces, combining everything inside the interfaces. This is done by maintaining everything where the ϕ-value is below 0. As every pixel contains the least distance to an interface, the only thing we have to do is to keep the lowest distance from both input fields and thereby always having the lowest distance to either of the interfaces.

$$\phi(x, y) = \min(\phi_1(x, y), \phi_2(x, y)) \tag{2.1}$$

Intersection To get the intersection between two interfaces we remove everything not contained in either of the interfaces. By keeping the maximum value from both fields, the pixels inside the first object will get the distance to the second object. In the intersection the biggest distance will be the one closest to the edge of the intersection as we are working with two negative numbers. As we are working with SDFs the result will also comply with the rules of an SDF.

$$\phi(x, y) = \max(\phi_1(x, y), \phi_2(x, y)) \tag{2.2}$$

Subtract The subtraction method is also simple. By subtracting the values from the second field from the first, we maintain the abilities of the SDF while removing the distances from it in the first. This results in a new SDF where the interfaces in the second field are removed from the first.

$$\phi(x, y) = \max(\phi_1(x, y), -\phi_2(x, y)) \tag{2.3}$$

2.4 Reinitialization

The main advantage of representing an implicit counture or surface as a signed distance field, is that the length of the gradient is one. This is also exactly what defines a signed distance field.

When constructing or manipulating a SDF defined on a Cartesian grid, the result is not always a SDF. So to enable further calculations or iterations of an algorithm the result must be turned into a new SDF that reflects the changes done by the calculations. This process is called *reinitialization* of the implicit function, and can be done several different ways.

Algorithms for reinitializing SDFs focus on reinitializing the whole domain, and at the same time keeping the zero level set as fixed as possible. This means that the process disrupts data outside the zero level set, which depending on the type of phenomena being modeled can be problematic. For the problems we are modelling this is not an issue and is therefore not of relevance here.

Before diving into the algorithms a good question is, how often must the implicit function be reinitialized? There is no good answer to this question, because it depends on how rapidly the contour is changing. In our work we have been reinitializing after every change, which ensures that this is not a source of error.

The two most used algorithmic approaches to reinitialization are: methods that geometrically calculate distances to the contour or surface, and PDE based methods that numerically approximate solutions to the Eikonal equation: $||\nabla\phi|| = 1$ [Bri08, page 89].

When initializing the implicit contour in section 2.2 on page 7 we used an algorithm that geometrically calculated the distance to the contour. So the natural choice should be to also use this algorithm for reinitializing, but because the algorithm does not calculate the SDF precise enough, we cannot use the algorithm when reinitializing. Furthermore if we had used this type of algorithm, we would have had to construct the contour before invoking the algorithm. Constructing the contour is very expensive, and is something that should be avoided at almost all cost when using the level set method. Instead we have chosen to use PDEs to solve the Eikonal equation.

The PDE way of reinitializing

When using the PDE way of reinitializing the level set, we solve the following PDE [OF02, page 65-66]:

$$\phi_t + S(\phi_0)(|\nabla\phi| - 1) = 0 \tag{2.4}$$

Where ϕ is the SDF being evolved as a PDE, and ϕ_0, is the initial SDF given as input to the reinitialization algorithm, which has not been altered by the process of solving the PDE. The function $S(\phi_0)$ gives the sign of the SDF, like the following function, described in [OF02, page 66]:

$$S(\phi) = \begin{cases} -1 & \text{if } \phi < 0, \\ 0 & \text{if } \phi = 0, \\ 1 & \text{if } \phi > 0. \end{cases} \tag{2.5}$$

When using this approach of reinitializing the SDF, the grid points in ϕ that are nearest to the contour is reinitialized first, and then propagated in the normal direction from the zero level set, hereby reinitializing the grid points in layers around the zero level set, reinitializing a new layer each iteration.

This algorithm is relatively slow if all grid points need to be reinitialized, because it only reinitializes one layer in each iteration when solving the PDE.

This means that the PDE, on a two-dimensional domain, must be iterated $\sqrt{width^2 \times height^2}$ times to make sure that the algorithm has reinitialized the whole domain.

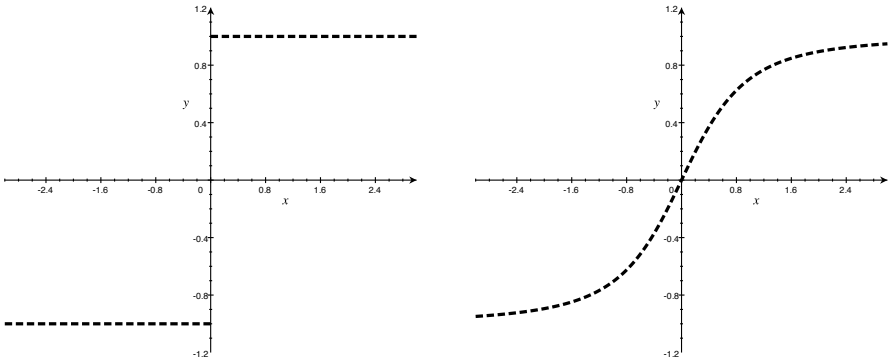
But because the algorithm has the property of reinitializing the SDF in layers it is of special interest in regards to performance when using a narrow band level set as described in section 4 on page 19. Here only three or four layers around the SDF needs to be reinitialized, making the algorithm ideal for the narrow band approach.

The sign function S

Because equation 2.4 is a hyperbolic PDE, we need to use a smeared out version of equation (2.5). One way of smearing the function is to use equation (2.6).

$$S(\phi_0) = \frac{\phi_0}{\sqrt{\phi_0^2 + (\Delta x)^2}} \tag{2.6}$$

The difference between equation (2.5) and equation (2.6) can more easily be seen by looking at plots of the two functions, as depicted in figure 2.5.



(a) Plot of equation (2.5)

(b) Plot of equation (2.6), $\Delta x = 1$

Figure 2.5: Illustrations of S

Implementation

The implementation of how to solve the PDE in equation (2.4), uses the description of Godunov's scheme from [OF02, page 58] in the spatial dimensions, and a forward Euler in the temporal dimension, as described in formula (1.3) in [OF02, page 10].

```
Tex<float> phi0 = GetPhi(); Tex<float> phin = GetPhi();
for (unsigned int i = 0; i<iterations; i++) {
    for (unsigned int x = 0; x < width; x++) {
        for (unsigned int y = 0; y < height; y++) {
            float xy = phi(x, y);
            float phiXPlus = 0.0f;
            float phiXMinus = 0.0f;
            float phiYPlus = 0.0f;
            float phiYMinus = 0.0f;
            if (x != width-1) phiXPlus  = (phi(x+1, y) - xy);
            if (x != 0)       phiXMinus = (xy - phi(x-1, y));
            if (y !=height-1) phiYPlus  = (phi(x, y+1) - xy);
            if (y != 0)       phiYMinus = (xy - phi(x, y-1));

            float dXSquared = 0;
            float dYSquared = 0;
            float a = phi0(x,y);
            if (a > 0) {
                // formula 6.3 page 58
                float max = std::max(phiXMinus, 0.0f);
                float min = std::min(phiXPlus, 0.0f);
                dXSquared = std::max(max*max, min*min);
                max = std::max(phiYMinus, 0.0f);
                min = std::min(phiYPlus, 0.0f);
                dYSquared = std::max(max*max, min*min);
            } else {
                // formula 6.4 page 58
                float max = std::max(phiXPlus, 0.0f);
                float min = std::min(phiXMinus, 0.0f);
                dXSquared = std::max(max*max, min*min);
                max = std::max(phiYPlus, 0.0f);
                min = std::min(phiYMinus, 0.0f);
                dYSquared = std::max(max*max, min*min);
            }
            float normSquared = dXSquared + dYSquared;
            float norm = sqrt(normSquared);

            // Using the S(phi) sign formula 7.6 on page 67
            float sign = phi0(x,y) / sqrt(phi0(x,y)*phi0(x,y) + 1);
            float dt = 0.3; // A stabil CFL condition
            phin(x,y) = phi(x,y) - sign*(norm - 1)*dt;
        }
    }
    for (unsigned int y=0; y<height ; y++)
        for (unsigned int x=0; x<width; x++)
            phi(x,y) = phin(x,y);

}
```

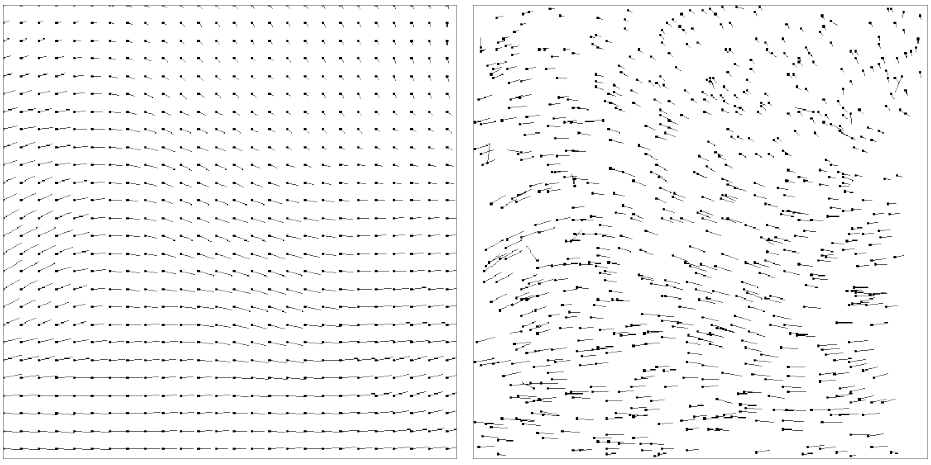
Chapter 3

Evolving the interface

3.1 Externally generated velocity field

The process of moving an implicit surface given by a signed distance function is known as *level set methods*. Level set methods are ways of influencing signed distance fields to move the implicit surfaces contained therein. This is done by solving certain equations of motion that we will describe in this section.

First we distinguish between Lagrangian and Eulerian representations of the surface. To understand the difference between the two viewpoints we imagine how we could measure the movement of e.g. a fluid. In the Eulerian viewpoint we would place measuring devices at fixed points in the fluid, and continues sample the velocity of the fluid, the measured value is taken as a avarage for an area. For convenience the measuring devices are almost always placed in a uniform grid, with square areas as illustrated in figure 3.1a.



(a) Eulerian (b) Lagrangian

Figure 3.1: The eulerian and the lagrangian viewpoints.

In the lagrangian viewpoint we let the measuring devices move along the current in the fluid and take measurements at different locations in the fluid. Here we imagine that the measuring device is part of the fluid or a particle

that the fluid moves around. Figure 3.1b shows how the current has moved the devices as time has progressed.

These two viewpoints are tightly coupled to the way the simulation data is represented and visuliazed. Lets say that we are interested in visualizing the surface of the substance. The Lagrangian data is keept as points in space and moved around by updating the position of the points. This can be visualized by imposing that the points define a surface and rendered as geometry e.g. triangles between the points. The eulerian viewpoint is seen as a 3D grid. Each cell in the grid has a density that describes how much substance the cell contains (0-100%).

In the Lagrangian representation, movement by a velocity field can be accomplished by solving the ordinary differential equation:

$$\frac{d\vec{x}}{dt} = \vec{V}\left(\vec{x}\right) \tag{3.1}$$

As discussed previously, however, using implicit surfaces by an Eulerian representation rather than an explicit Lagrangian representation, such as a polygon mesh, provides certain benifits. Nowhere is this more clear than when implementing moving surfaces. Moving a surface built from triangles presents a number of problems. First and foremost, it becomes necessary to determine whether the surface starts to overlap itself and what to do if it does. Clearly this is problematic when dealing with polygons, since there is no obvious or "natural" way of merging two polygons which have overlapped.

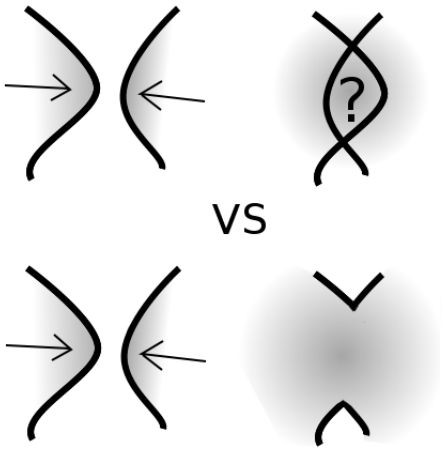


Figure 3.2: Two moving surface representations. Top: Explicit representation, merging is a problem. Bottom: Implicit surface by signed distance field, merging is automatic.

With implicit surfaces, this problem disappears, since an implicit surface is just that: implicit. If two "interior" areas of the implicit function "overlaps" it simply means that that area of the domain is in the interior of the function, and the interface will wrap around it as appropriate. See figure 3.2 for an illustration of the advantage of implicit surfaces.

The level set equation

We examine now how to evolve an implicit surface, or interface, by affecting the underlying implicit function with an externally generated velocity field. This process of convection in an Eulerian representation is defined by equation 3.2

$$\phi_t + \vec{V} \cdot \nabla\phi = 0 \tag{3.2}$$

This partial differential equation is referred to as the *level set equation* due to its central importance to level set methods. It describes the evolution of an implicit function ϕ by a velocity field $\vec{V}$. $\nabla\phi$, of course, is the gradient of the function. From this, we have

$$\vec{V} \cdot \nabla\phi = u\phi_x + v\phi_y \tag{3.3}$$

Where ϕ_x and ϕ_y are the spacial derivatives in the first two dimensions respectively and u and v are the two components of the velocity vector.

Since we are essentially only interested in moving the implicit surface or interface with the velocity field, it is sufficient for the field to contain values only in a band around the interface. For simplicity of implementation, however, we assume that the field is defined across the entire domain.

For concrete implementation purposes, to evolve an implicit surface in an n-dimensional domain, the velocity field is an n-dimensional cartesian grid of n-dimensional vectors. In our two-dimensional case, that means all velocity fields are double arrays of two-value vectors.

Upwind differencing

How then, do we numerically solve equation 3.2? As we know, the implicit function is discretized into a cartesian grid of cells with Δx representing the width and height of these cells in the theoretical continuous field. In our case this is a discretized signed distance field, each cell in the grid being of course a number representing the distance from the zero-isocountour. So too is the velocity field represented discretely with vectors, as mentioned previously.

Now, since motion takes place across time, we will need discretize our motion across time as well. We discretize time into steps of Δt. The n'th time step we denote t^n and the state of the cartesian grid of our signed distance field at that time as ϕ^n. Since the velocity also may change over time, this too is separated into discrete steps $\vec{V}^n$.

One way of discretizing equation 3.2 is the simple *forward Euler* method. Equation 3.4 shows how the time-dependant term ϕ_t of equation 3.2 is discretized by forward Euler.

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + \vec{V}^n \cdot \nabla\phi^n = 0 \tag{3.4}$$

This is a first-order accurate method for time discretization, which [OF02] suggests is adequate, based on practical experience. Expanding the gradient as in 3.3 we get

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + u\phi_x^n + v\phi_y^n = 0 \tag{3.5}$$

To calculate this equation, then, we must find the spatial derivatives in the x and y directions. For this we can again use a first-order difference method. However, we need to pick a direction in which to calculate the derivative. That is, do we use

$$D^+\phi \approx \frac{\phi_{i+1} - \phi_i}{\Delta x} \quad \text{or} \tag{3.6}$$

$$D^-\phi \approx \frac{\phi_i - \phi_{i-1}}{\Delta x} \tag{3.7}$$

These being forward and backwards difference, respectively. Naturally, we choose by examining the velocities given for the cell in question in the velocity field and take the difference in the direction of change. Not surprisingly, the term "Upwind differencing" is derived from this way of sampling in the direction of change. Although it might seem natural to simply use a central difference, according to [OF02], this is unstable with forward Euler time discretization.

The method, then, for each cell in the grid of the implicit function is as follows:

- Look up the velocity in the corresponding cell in the velocity field.
- Calculate the appropriate forward/backwards difference
- From these, arrive at the partial spatial derivative
- Store the new cell value

When this has been done for the entire grid, overwrite it with the new values. Essentially, we are "collecting" the values that need to be written to the current cell of the grid in the direction they come from via the velocity grid.

To ensure stability of this method, [OF02] recommends limiting the time-step according to the *Courant-Friedrich-Lewy* condition (CFL for short) which can be written as

$$\Delta t < \frac{\Delta x}{\max\{|u|\}} \tag{3.8}$$

By enforcing this, we increase the guarantee that small errors are not amplified over time. The effects of an unstable method are commonly referred to as "exploding", for obvious reasons.

Finally, [OF02] recommends methods such as an essentially nonoscillatory (ENO) way of computing more accurate spatial differences and total variation diminishing (TVD) Runge-Kutta for further accuracy in temporal difference. In short, ENO is a polynomial forward or backward difference function and Runge-Kutta is essentially a multiple-step version of the simpler forward Euler method we use.

The above method, then, is the general solution to the problem of convecting an implicit surface by an externally generated velocity field. Using it, we may arbitrarily define our field of motion to produce motion in any way we wish.

3.2 Internally generated velocity field

In contrast to section 3.1, this section describes motion using a internally generated velocity field.

Motion in the normal direction

One of the most simple things we can do, is to move the interface in the normal direction $\vec{N}$ using a constant a, efficiently scaling the iso-surface.

From equation 3.2 on page 14 we have:

$$\phi_t + \vec{V} \cdot \nabla\phi = 0 \tag{3.9}$$

Since $a\vec{N}$ has the same direction as $\nabla\phi$, we have the following:

$$\begin{aligned} a\vec{N} \cdot \nabla\phi &= a\frac{\nabla\phi}{|\nabla\phi|} \cdot \nabla\phi \\ &= a\frac{|\nabla\phi|^2}{|\nabla\phi|} = a|\nabla\phi| \end{aligned}$$

With corresponds to this level set equation:

$$\phi_t + a|\nabla\phi| = 0 \tag{3.10}$$

At this point, the point of the signed distances field is clear, if ϕ is a SDF, then $|\nabla\phi| = 1$. This means that we can solve (3.2) very simply using the forward euler method (see equation 3.6).

This corresponds to the following code:

```
for(unsigned int x=0; x<width; x++) {
    for(unsigned int y=0; y<height; y++) {
        phi(x,y) += -a;
    }
}
```

A sample is shown in figure 3.3 where the AU logo is expanded with $a = 1.0$ for 6 and 20 iterations.

(a) The original logo (b) After 6 iterations (c) After 20 iterations

Figure 3.3: The AU logo after expanding with $a = 1.0$ for 6 and 20 iterations

Mean-Curvature

Another interesting thing we can do with a level set, is to move the interface in the normal direction with a velocity proportional to its curvature (this is called mean-curvature). The effect of this is to soften or sharpen the interface.

To do this we need the following velocity field:

$$\vec{V} = -b\kappa\vec{N} \tag{3.11}$$

Where κ is the curvature, and $b > 0$ is a constant describing the speed of the motion.

This corresponds to the level set equation:

$$\phi_t - b\kappa|\nabla\phi| = 0 \tag{3.12}$$

This is a parabolic equation, so to discretize it we need to use a new approach.

We start by exploring the curvature κ. It is defined as:

$$\kappa = \nabla \cdot \left(\frac{\nabla\phi}{|\nabla\phi|} \right) \tag{3.13}$$

When ϕ is a SDF, then $|\nabla\phi| = 1$, which simplifies our equation to: $\kappa = \nabla^2\phi$. Which is the laplacian operator: $\kappa = \Delta\phi$. In this context, $\Delta\phi$ is defined as:

$$\Delta\phi = \phi_{xx} + \phi_{yy} \tag{3.14}$$

ϕ_{xx} and ϕ_{yy} can be solved using central difference:

$$D_x^+ D_x^- \phi \approx \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \tag{3.15}$$

Again the forward Euler method can be used, but we have to use a small time step according to [OF02, page 44]:

$$\Delta t \left(\frac{2b}{(\Delta x)^2} + \frac{2b}{(\Delta y)^2} \right) < 1 \tag{3.16}$$

If we respect these, then the implementation is straight forward:

```
for(unsigned int x=1; x<width-1; x++) {
    for(unsigned int y=1; y<height-1; y++) {
        const float dx = 1.0;
        float phi_xx = (phi(x+1,y) - 2*phi(x,y) + phi(x-1,y))/(dx*dx);
        float phi_yy = (phi(x,y+1) - 2*phi(x,y) + phi(x,y-1))/(dx*dx);

        float kappa = phi_xx + phi_yy;
        phi(x,y) += kappa * a; //mean curvature
    }
}
```

(a) The original logo (b) After 20 iterations (c) After 100 iterations

Figure 3.4: The AU logo after mean-curvature with $a = 0.4$ for 20 and 100 iterations

This is demonstrated in figure 3.4. The logo becomes less sharp, and if continued long enough, it will disappear.

Morph

Another simple way of evolving the interface is a technique called "morphing". Given two signed distance fields, we can "morph" one into the other simply by using the difference in fields in place of velocity of change. This is a simple idea which can yield visually impressive results.

The operation of morphing ϕ_1 to ϕ_2 is captured in this equation:

$$\phi_t = \phi_1 + (\phi_2 - \phi_1) \cdot -a \tag{3.17}$$

Figure 3.5 shows the intuition behind the equation. The parts of the field that are "inside" one curve and "outside" another yield higher differences than the parts that are inside or outside both curves. Depending on the sign of a, the equation adds a term to ϕ_1 that increases or decreases its values in correspondance to the difference of the fields. The result of this is that the level sets of one field are evolved to resemble the curves of the other.

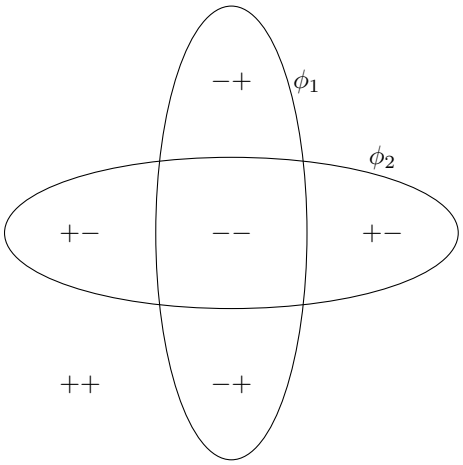


Figure 3.5: Morphing two SDFs

The implementation is simply as follows:

```
for(unsigned int x=0; x<width; x++) {
    for(unsigned int y=0; y<height; y++) {
        phi(x,y) += (phi(x,y) - phi2(x,y)) * -a; //morph
    }
}
```

Chapter 4

Narrow-band

When working with level sets you almost always work with small objects in a much bigger context. This is due to the fact that our level set is defined in on a Cartesian grid, and we have to manipulate all grid cells(from here on referred to as the workspace) to make sure the SDF is always welldefined, meaning the length of the gradient is equal to one. This is a time consuming process as described in section 2.4 on page 8. A solution to make the processes faster is to only work on the area of the SDF just beside the contour of interest. This method is called the narrow band method and is described in [AS95]. A narrow band of the Aarhus University(AU) logo can be seen in figure 4.1 where (a) shows the input figure given to the program, and (b) showing the narrow band generated from the interface of the input.

(a) The input (b) The Narrow band

Figure 4.1: Visualizing the narrow band of the AU logo.

4.1 Idea of the Narrow Band Method

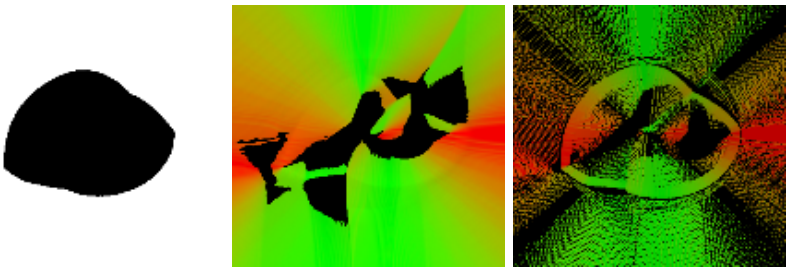
The basic idea of the method is to narrow down the amount of grid cells we are working on, to include as few as possible without loosing accuracy. This is done by keeping a band of γ calls around the interface so we only update the SDF only where $-\gamma \leq \phi(x, y) \leq \gamma$. When we solve the level set equation within the narrow band, we use the values of the neighboring cells. On the edge of the band we cannot be sure the values of their neighbors are valid as they aren't in the narrow band, and this is where the outer band, also called an safety band, comes in to play. All the neighbors on both sides of our narrow band is added to this set. The only job of the outer band is to supply the inner band with information when calculating on the border of the band. Because we know how big γ is, it is safe to assume that the distance from the outer

band to $\phi = 0$ is more or less γ. Setting the value of all cells in the outer band to γ is therefore safe if we promise to reinitialize the band, after solving the level set equation, to make the length of the gradients one. This will make the whole narrow band including the outer band satisfy the requirements of the SDF and we only have to reinitialize inside the band itself. Hence the program flow is changed a little. We first evolve the interface by solving the level set equation on the SDF. We then recalibrate the narrow band to fit the newly generated SDF and then at the end reinitialize inside the narrow band.

Solve level set equation
Update narrow band
Reinitialize

Figure 4.2: Program flow with narrow band

To decide the size of γ, and thereby the width of the narrow band, we have to look at the maximum distance the interface can move in a time step. In our implementation this boundary is at one cell per time step, therefore a narrow band of size 3 around the interface is sufficient. As seen in figure 4.1 on the preceding page the band reaches just outside the interface border as displayed in red. The green outline around the red band is the outer safety band.



(a) The output (b) Without narrow band (c) With narrow band

Figure 4.3: A morph between two figures (AU-logo to a circle) showing the gradient map with and without the narrow band method. In the gradient map we have also shown places where the length of the gradient is different from zero, here depicted with the color black.

As seen in figure 4.3, the narrow band method is powerful. Using the method we have significantly lowered the number of grid cells needed to visit while working with the SDF, simultaneously we have reduced the number of times needed to reinitialize. If we only reinitialize once it is clear that the method without narrow band is more correct overall, though it contains large amounts of wrong data around the interface, making calculations faulty. The narrow band version is clearly wrong in the majority of the cells, but around the interface we see that the narrow band is in effect and the one reinitialisation per time step almost covers our needs.

4.2 Implementation

To represent our Narrow band we need a data structure to hold the new information. Normally we traverse the workspace linearly and know which cells have been updated, and which we are going to visit next. In the narrow band approach we have no linearity and, the narrow band can be divided into many detached objects in the workspace. Therefore we need another way to traverse it than run through (x,y) in the height and width of the workspace. Our solution is to keep every grid cell in a vector with (x,y) coordinates, and append the vectors of the cells inside the narrow band to a list. This list will then contain the complete narrow band, both inner and outer band. To distinguish the two bands from each other we are using a two dimensional matrix holding a 1 if the said cell is in the inner band, and a 2 if it is in the outer, 0 means the cell is in neither. We also maintain an integer containing the number of cells inside the narrow band.

To build the narrow band, the technique is simply to traverse the SDF and check the distance to an interface. If we are within the γ range of it, we are inside the narrow band, and the cell is added to the set of cells inside the band. Simultaneously we add the said cell to the matrix telling which of the bands it is contained in. For simplicity we can in the same traversal check if it should be in the outer band if it isn't inside the inner band. As it does not matter if we take too much in the outer band, we here just check if it is in γ range + 2 cells. We take more than needed, but we are on the safe side. The type of the cells in the outer band is of course added to the type matrix. This is done as shown by this code:

```
for (unsigned int x=0; x<width; x++)
    for (unsigned int y=0; y<height; y++) {
        float phiVal = fabs((*phi)(x, y));
        if (phiVal <= narrowBandWidth) {
            //Part of the inner band
            narrowBand[narrowBandSize++] = (Vector<2,int> (x,y));
            nbType(x,y) = 1;
        } else if (phiVal <= narrowBandWidth + 2.0f) {
            //Part of the outer(safety) band
            narrowBand[narrowBandSize++] = (Vector<2,int> (x,y));
            nbType(x,y) = 2;
        } else {
            nbType(x,y) = 0;
        }
    }
```

4.3 Discussion

Testing the narrow band method we got an significant speedup. We did not get to implement a faster version for rebuilding the narrow band as explained in [PMO+99]. In this version of the narrow band they rebuild the narrow band inside the old narrow band. This can be done as the interface only moves one cell per time step. Therefore we can build the new inner band inside the old narrow band. The new outer band can then be found be traversing the neighbors of the new inner band.

Chapter 5

CUDA

Improving the performance of our algorithms can be done in many ways, but one of the more obvious ones is using parallel computing!

Currently, the most accessible way to run programs in parallel is using the graphics computation unit (GPU). Modern GPUs are very powerful, and major manufactures have released software development kits (SDK) for utilising the GPU for general purpose computation.

One of these SDKs is nVidias CUDA[nVi]. The only thing needed to use CUDA is a nvidia graphics card that is relatively new (a few years tops), and the free SDK found at the CUDA website.

As GPUs were developed to render graphics, they are optimized to work on spatially coherent data. This makes many of our algorithms a natural target, as we often only need information about neighbouring data points.

5.1 Threads

The CUDA programming model is centered about data parallel programs. This means that you spawn a thread for each element in your data, which runs the same program. In our case, this means spawning a thread for each pixel. Luckily our algorithms are already in this format, just with two `for`-loops iterating over the pixels.

Most of our algorithms uses this pattern:

```
for (unsigned int i = 0; i<iterations; i++) {
    for (unsigned int x = 0; x < width; x++) {
        processPixel();
    }
}
```

Which is easily translated into:

```
    const dim3 blockSize(32,16,1);
    const dim3 gridSize(width/blockSize.x, height/blockSize.y);
    processPixel<<gridSize,blockSize>>();
```

The grid and block size are telling CUDA how many threads to spawn. A grid contains many blocks, and each block contain many threads. In this sample, each block have 32×16 threads, and the grid have $\frac{width}{32} \times \frac{height}{16}$

blocks. If width and height are divisible with 32 and 16, this corresponds to $width \times height$ threads.

In CUDA 2.3, a block can contain no more than 512 threads, hence the block size of $32 \cdot 16 = 512$.

5.2 Memory

Memory in CUDA is divided in 3 parts. The Per-thread local memory than only a single thread can access. Per-block shared memory which is accessible to every thread in the same block and the global memory that every thread can access. This can be seen in figure 5.1. Here the global memory is separated into global, constant and texture memory, and local memory is separated into local memory and registers.

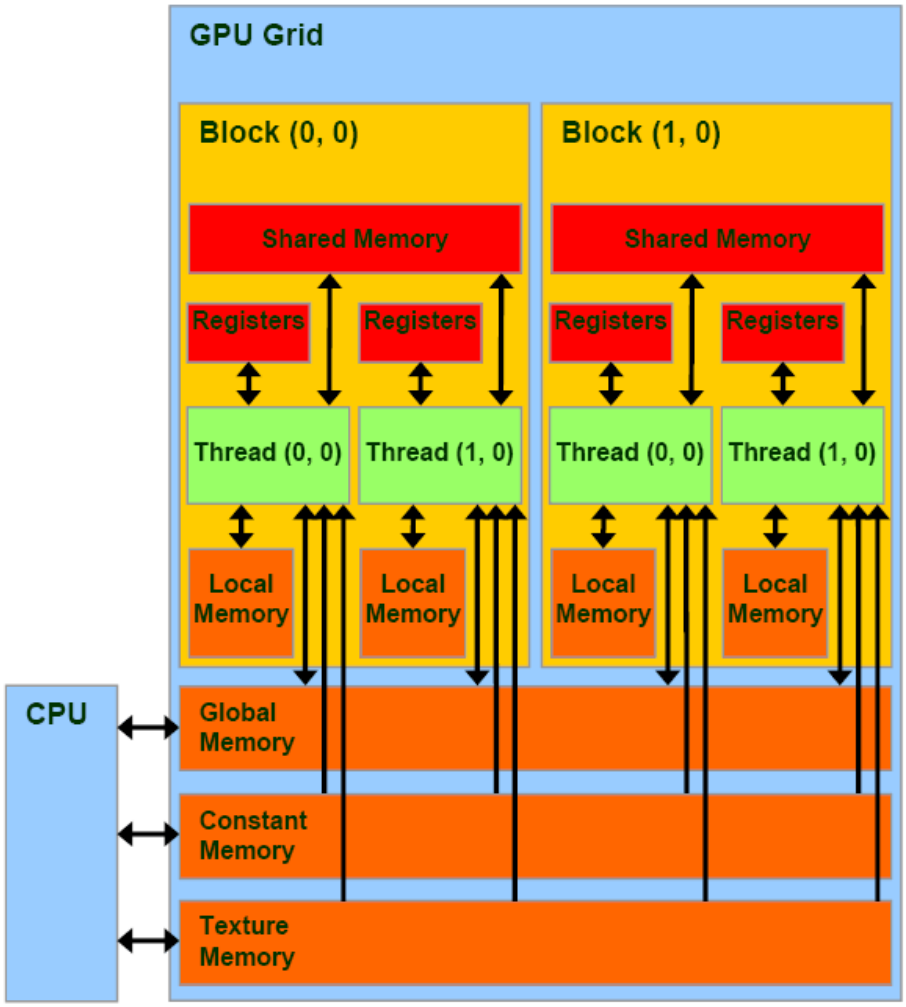


Figure 5.1: The CUDA Memory Model (from [nVi])

The difference between the shared and global memory is the speed. The

shared memory is much faster, but also much smaller (typically 16 KB). Its also inaccessible from threads in different blocks.

This make it challenging to utilise the whole GPU, as the algorithms needs to be rethought.

One way to do this could be letting each thread fetch it's value (ϕ in our case) into the shared memory. Then if the threads are organized in block where neighbouring threads are in the same block, each thread can fetch the neighbouring pixels from the shared memory.

Such an optimization creates new challenges, as the threads near the border cannot cross over to the next blocks shared memory. The solution is to pad the area around the edges of the blocks, so if a pixel is on the border, its run by both blocks. This makes us run a few more threads than we have pixels, but it increases the speed as we can exploit the shared memory.

A more simple type of optimization, is to use texture memory. GPUs often need fast access to textures, so it have a texture cache optimized for 2D spatial locality. This means that fetching data from a texture will make fetching the neighbouring pixels faster.

5.3 Implementation

After a quick time profiling, we found that `Reinitialize` is where our program spends most of its time, so this was the first to be converted into CUDA.

The following is a very naive conversion. The code is almost the same as the original in section 2.4, and there are no clever usage of shared memory or other optimizing tricks.

```
#define GetPhi(phi,x,y,w) phi[x+w*(y)]

__global__ void reinit(float *phi,float* phi0, float* phin,
                       unsigned int width, unsigned int height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;

    if (x > width || y > height)
        return;

    float xy = GetPhi(phi,x,y,width);

    float phiXPlus = 0.0f;
    float phiXMinus = 0.0f;
    float phiYPlus = 0.0f;
    float phiYMinus = 0.0f;
    if (x != width-1) phiXPlus  = (GetPhi(phi,x+1, y,width) - xy);
    if (x != 0)       phiXMinus = (xy - GetPhi(phi,x-1, y,width));

    if (y !=height-1) phiYPlus  = (GetPhi(phi,x, y+1,width) - xy);
    if (y != 0)       phiYMinus = (xy - GetPhi(phi,x, y-1,width));

    /* GetPhi(phin,x,y,width) = phiYPlus; */
    /* return; */


    float dXSquared = 0;
    float dYSquared = 0;
    float a = GetPhi(phi0,x,y,width);
    if (a > 0) {
```

```
        // formula 6.3 page 58
        float _max = max(phiXMinus, 0.0f);
        float _min = min(phiXPlus, 0.0f);
        dXSquared = max(_max*_max, _min*_min);

        _max = max(phiYMinus, 0.0f);
        _min = min(phiYPlus, 0.0f);
        dYSquared = max(_max*_max, _min*_min);
    } else {
        // formula 6.4 page 58
        float _max = max(phiXPlus, 0.0f);
        float _min = min(phiXMinus, 0.0f);
        dXSquared = max(_max*_max, _min*_min);

        _max = max(phiYPlus, 0.0f);
        _min = min(phiYMinus, 0.0f);
        dYSquared = max(_max*_max, _min*_min);
    }

    float normSquared = dXSquared + dYSquared;
    float norm = sqrt(normSquared);

    // Using the S(phi) sign formula 7.6 on page 67
    //float sign = phi(x,y) / sqrt(phi(x,y)*phi(x,y) + normSquared);
    float sign = GetPhi(phi0,x,y,width) /
        sqrt(GetPhi(phi0,x,y,width)*GetPhi(phi0,x,y,width) + 1);
    float t = 0.3; // A stabil CFL condition
    GetPhi(phin,x,y,width) = GetPhi(phi,x,y,width) - sign*(norm - 1)*t;


}
```

Coping the data, and starting the threads are done in the following code:

```
void cu_Reinit(float* data,
               unsigned int w,
               unsigned int h,
               unsigned int iterations) {
    float* phiData;
    float* phi0Data;
    float* phinData;

    cudaMalloc((void**)&phiData, sizeof(float)*w*h);
    cudaMalloc((void**)&phi0Data, sizeof(float)*w*h);
    cudaMalloc((void**)&phinData, sizeof(float)*w*h);
    cudaMemcpy((void*)phiData,(void*)data, sizeof(float)*w*h,
               cudaMemcpyHostToDevice);
    cudaMemcpy((void*)phi0Data,(void*)data, sizeof(float)*w*h,
               cudaMemcpyHostToDevice);
    cudaMemcpy((void*)phinData,(void*)data, sizeof(float)*w*h,
               cudaMemcpyHostToDevice);


    CHECK_FOR_CUDA_ERROR();

    const dim3 blockSize(32,16,1);
    const dim3 gridSize(w/blockSize.x, h/blockSize.y);

    for (unsigned int i=0;i<iterations;i++) {
        reinit<<<gridSize,blockSize>>>(phiData,phi0Data,phinData,w,h);
        float* tmp = phiData;
        phiData = phinData;
        phinData = tmp;
```

```
        cudaThreadSynchronize();
        CHECK_FOR_CUDA_ERROR();
    }

    cudaMemcpy((void*)data,(void*)phiData,
               sizeof(float)*w*h,cudaMemcpyDeviceToHost);
    CHECK_FOR_CUDA_ERROR();
    cudaFree(phiData);
    cudaFree(phi0Data);
    cudaFree(phinData);
}
```

5.4 Results & Conclusion

The results in table 5.1 are taken from a system with a 1.8 Ghz Intel Core 2 Duo CPU, 4 GB RAM and a 512MB nVidia GeForce 9600M GT. The time is an average of about 100 iterations of the algorithm.

Algorithm	CPU (μs)	GPU (μs)	Speedup ($\times$)
Reinitialization	417825	136675	3.0570697
- with textures	-	100006	4.1779993

Table 5.1: GPU vs. CPU comparison

The results shows a significant speedup. Using a quite naive implementation the speedup is easily tripled on a inexpensive consumer graphics card.

Using textures to cache lookup, we gain even more performance, going from 3x to 4x. If we'd had more time more optimization techniques could have been applied. E.g. using shared memory which most likely would have improved performance even more.

A visual ilustration of the sppedup can be seen in figure 5.2. Here the AU logo is morphed into a circle, and after 30 seconds the GPU version is way ahead of the CPU version.

(a) Original AU logo (b) Circle (c) GPU version (d) CPU version

Figure 5.2: Comparision of CPU vs. GPU on morph

Chapter 6

Segmentation

(a) (b) (c) (d) (e)

Figure 6.1: A segmentation method in progress showing the segmentation at iteration 75, 150, 225, 250 and 275.

Segmentation is an incredibly important area of interest when it comes to Medical Imaging. Segmentation is the problem of partitioning a digital image into multiple segments that are more meaningful and easier to analyse. Typically one would like to locate the boundaries in a picture such as lines, curves, etc.

The result of a segmentation is a set of segments that covers the entire picture. All pixels in each segment shares properties based one how the picture is segmented. It could be color or intensity. Adjecent regions are significantly different based on these characteristics.

A technique is to initially start inside the object you want to segment and then expand it like a balloon until the surface reaches the edge of the contour.

To illustrate segmentation in a level set model, I have implemented two different algorithms which are described in sections 6.2 on the following page and 6.3 on page 29.

6.1 Implicit vs. Explicit representation

Since segmentation techniques normally are used to locate organs in MR scans or measure the volume of tissue, eg. from real people, it is very important that the segmentation is correct and that it is fast. Therefore, we have to convince ourselves that our technique can find the contour in images even though they can contain a lot of noise and artefacts.

In an explicit representation, we have the problem that when we only represent the surface, we run into trouble when segmenting artefacts as can be

seen in figure 6.2. The problem is that the segmentation can not figure out to skip over the artefacts which resolves in a segmentation that never terminates. There exist algorithms that try to skip the artefacts, but they are prone to failure.

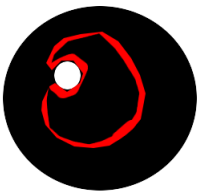


Figure 6.2: We see how the segmentation, using an explicit representation has trouble with artefacts in the picture. The surface is about to wrap around it self creating an endless loop around the artefact.

In an implicit representation, the problem vanishes since we look at the larger picture and not just the boundary of the current segmentation. Because we use a level set to solve the problem, when our algorithm reaches an artefact the solver simply goes around it and merge at the other side.

6.2 Algorithm 1 - Moving in the normal direction

In this algorithm I have been inspired by the balloon algorithm from [BMS97]. To segment a part of an image, we start with a small area inside the area we want to segment and grow it in the normal direction if we have not reached the boundary yet. We know if we have hit the boundary if the value of the pixel is smaller/larger than a specified treshold we define. If we have crossed the border (again based on the threshold value), we shrink that point of the surface by going in the reverse direction of the normal.

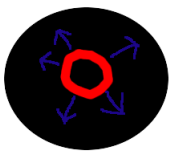


Figure 6.3: In the figure, we see how the segmentation grows in the normal direction

Basically we solve the following equation:

$$\phi_t + a|\nabla\phi| = 0 \tag{6.1}$$

which is easily achieved using the code below:

```
for(unsigned int x=0; x<width; x++) {
    for(unsigned int y=0; y<height; y++) {
        [...]
        if (picture(x,y) > threshold) {
            phi(x,y) += a;
            growth += -a;
        } else {
            phi(x,y) += -a;
            growth += a;
        }
    }
}
[...]

if (growth / ``number of pixels moving'' < tTreshold) {
    done = true;
}
```

We want to be able to terminate the segmentation when we have found the correct area. We do this by looking at the zero iso-surface and check how much the surface is moving. When it slows down we know that we have found the area we want to segment.

This is a quite simple algorithm which surprisingly produces good results. On a picture of dimensions 512 x 512 it stops after approximately 250-300 iterations which is quite good. We could do this many times faster if we choose to implement the reinitialization step on the GPU via CUDA.

In order to make the segmentation algorithm stop when it reaches the boundary, we calculate the following factor:

$$\text{factor} = \text{growth} \ / \ \text{number of pixels moving on iso-surface}$$

Which is a number that goes towards zero when the iso surface stops moving. To see this, think about what happens when we have reached the boundary. since about half of the iso-surface is increased and the other half is going to be decreased the factor should be approximately zero. `tThreshold` is set to 0.03 through experiments.

6.3 Algortihm 2 - Edge detection

Algorithm 2 is more advanced and tries to find the edges beforehand to increase the likelyhood that we segment the correct part, see [MBZW02]. It builds a series of images and solves the following equation:

$$\frac{\partial\phi}{\partial t} - \text{grad}(D) \cdot \text{grad}(\phi) = 0 \tag{6.2}$$

t where image D has the edge information with an edge denoted as a one and a nonedge as a zero.

To compute image D we have to go through a number of steps. First, we compute an image A where every pixel is the norm of the gradient in the original image. Secondly, we compute an image B where every pixel is the

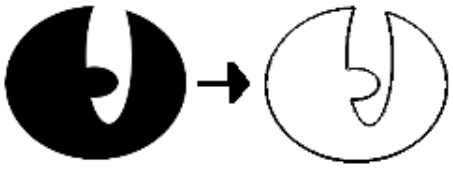


Figure 6.4: We see the result of computing the final image in the preprocessing step where the algorithm calculates the edges. On the left, we see the original picture we want to segment. On the right, the final image we use in the segmentation algorithm is showed. The final picture is generated by spotting zero crossings and by that creating a picture containing the edge information we need.

gradient in image A dotted with the normal in the original image. Compute an image C where every pixel is the absolute value of the gradient in the original image dotted with the normal. With this information, it is now possible to calculate the zero crossings. A zero crossing is defined to be either that one of the neighbouring pixels have a different sign than the current pixel, or that with the value of the current pixel is zero. And finally, if the corresponding value in image C is larger than some specified value then it is also a zero crossing. The final image, D, is calcutated by setting all pixels with an edge to one and all non-edges to zero. We need to run the reinitialization method on the image to make sure that all distances are correct. In this particulary instance, we need to do a thousand iterations. These calculations are all computed as a preprocessing step before we iteratively solve the level set equation (6.2 on the previous page).

To solve the level set equation we implement the following code:

```
for(unsigned int x=0; x<width; x++) {
    for(unsigned int y=0; y<height; y++) {
        phi(x,y) +=  (gradD(x,y) * gradPhi(x,y)) * time;
    }
}
```

Where time is the factor:

$$\frac{\Delta x}{\max\{|\text{gradient}(x,y)|\}}$$

When solving the level set equation, for every pixel we get a vector that goes away from the iso-surface and points at the closest edge in the normal direction.

6.4 Conclusion

I have implemented two algorithms for segmenting pictures where the first is a simple algorithm that grows only based on the normal of the iso-surface and stops its segmentation when the iso-surface encounters pixelvalues that cross the threshold specified.

The second algorithm is more advanced and makes good use of the information from the gradient by growing in the direction of the edges and the normal.

Due to time constraints I have only tested the algorithms on grayscale pictures and also not on real medical data, but nonetheless I still get results that should scale to real data.

Chapter 7

Simulating fluids using level sets

We have previously discussed the advantages of using level set methods to evolve interfaces by representing them implicitly rather than explicitly. One area in which this proves particularly useful is in the simulation of physical phenomena. We look at one such example in this section: the simulation of fluids. The basic idea is to let the fluid be represented by the zero-level isosurface of a signed distance field. By solving a set of equations we arrive at a velocity field, with which the interface can be evolved as discussed in the section on generating motion by externally generated velocity fields, section 3.1 on page 15.

7.1 Fluid equations

The basis of our fluid simulator is the *incompressible Navier-Stokes equations* as presented in [Bri08], a common approach to fluid simulation. These are a set of equations which describe the motion of an incompressible fluid. Incompressibility is a central quality of fluids. Simply put, it means that the fluid cannot grow or shrink in size. Naturally this property does not quite hold for real-world fluids, especially if we include gases in the definition, as fluids in the real world do compress to some degree. For graphical simulation purposes, however, it is adequate to assume complete incompressibility.

Here, then, is how the incompressible Navier-Stokes equations are usually written:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \Delta p = \vec{g} + \nu \nabla \cdot \nabla \vec{u}, \tag{7.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{7.2}$$

We will be breaking these equations down into some simpler parts that can be calculated discretely and numerically without too much hassle. For now, the variables of the equations: $\vec{u}$ is the field of velocities that affect the fluid, ρ is the density of the fluid, p tracks how much pressure is present across the fluid, $\vec{g}$ is the downwards force of gravity acting upon the fluid and ν is a term known as viscosity, which basically says something about how easily the fluid deforms.

Equation 7.1 is called the *momentum equation* and deals (not surprisingly) with the momentum of the fluid. That is, how the fluid continues to move once

in motion. Naturally, this equation also encompasses motion derived from external forces like gravity.

The incompressibility of the fluid is captured by equation 7.2 on the preceding page. This equation is derived from looking at the normal component of the velocity at the boundary of a given fluid. In basic terms, it says that the amount of fluid flowing out of the fluid region is equal to the amount flowing in. This measure of change is called divergence, and by this restriction we seek to remove it with as much numerical accuracy as possible. When evolving the interface with the resulting velocity field, the incompressibility constraint ensures that the interface is never moved in a way that produces a larger or smaller fluid volume (or area, in the case of our two dimensional implementation).

Looking at the equations, solving the entire scheme at once seems daunting. Thankfully, [Bri08] presents us with a method for breaking them into smaller, more managable parts. By using this technique, called splitting, we arrive at the following parts:

$$\frac{Dq}{Dt} = 0 \text{ (advection)}, \tag{7.3}$$

$$\frac{\partial \vec{u}}{\partial t} = \vec{g} \text{ (body forces)}, \tag{7.4}$$

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = 0 \tag{7.5}$$

$$\text{such that } \nabla \cdot \vec{u} = 0 \text{ (pressure/incompressibility)}, \tag{7.6}$$

Notice that the momentum equation has been split into three seperate parts. The first, advection, ensures that velocities are passed around the velocity field properly. The second simply applies external forces. The third and fourth adjust the velocity field under the incompressibility constraint, and thus the velocity field is created and maintained across steps. To formalize the preceding into an algorithm:

- Advect velocities around the velocity field
- Add external forces
- Adjust the velocity field for pressure
- Use the resulting velocity field to move the fluid

As mentioned, we use a signed distance function to represent the position and layout of the fluid. The zero-level isocontour is the surface of the fluid and naturally divides the domain into two sections: all values above zero represent empty space not currently occupied by the fluid while everything below zero represents the interior of the fluid.

Our natural inclination at this point would be to represent the discrete velocity field as an identically sized double array (or triple, depending on dimensionality) of values. Each cell would hold the velocity of the fluid at that point, which seems the simplest way of storing it. However, to ease certain portions of later sections of the simulation we have chosen instead the "Marker and Cell" (or MAC for short) grid which [Bri08] credits to Harlow and Welch.

7.2 Marker and cell grid

For a two-dimensional fluid domain, the velocities clearly need to be two-dimensional as well. With the MAC grid we chose not to simply store the complete velocity vector of the fluid at the center of each cell of the fluid. Instead, we store the only the normal components of the velocity at the *edges* of cells. As figure 7.1 illustrates, each edge of cell in the signed distance field has

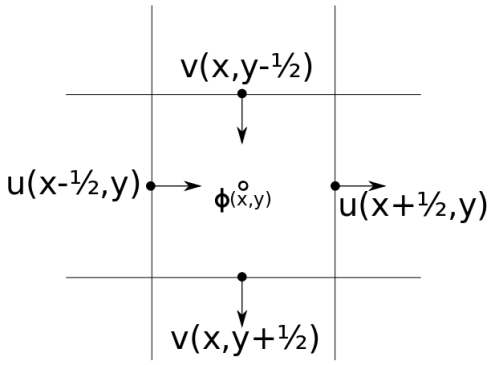


Figure 7.1: The MAC grid stores velocities at the edge of cells, rather than the center

associated with it a value indicating the component of the velocity vector in the corresponding direction. In practical terms, we store two grids in addition to the discretized signed distance field, one for each spatial dimension of the velocity field.

The purpose of this initially unintuitive representation becomes apparent when need to calculate central differences in the velocity field. With a "traditional" velocity field, computing the signed distance field we would sample velocities at either side of the cell in question, ignoring the value stored at the cell itself. Since our field may contain very thin feature-variation, this method of central differencing may accidentally ignore "sharp" features of no more than a cell's width.

With the MAC grid, when central differencing we can sample the "half-values" at the edges of cells, thereby efficiently avoiding this problem at no additional cost.

The MAC grid also needs to be able to handle values being taken between grid points. Not only at the cell edges, but at arbitrary locations in the grid, particularly at grid centers. We will see why in this next section.

7.3 Advection and forces

With the grids thus in place, we may begin our algorithm. Referring to the overview, the first step is the advection of the velocity field. In this step we allow motion to carry on, retaining momentum as the fluid moves. The implementation of this step is quite intuitive.

Recall equation 7.3 on the preceding page which describes the advection step. With our grid-based representation of the fluid it seems obvious to use an Eulerian solution to the equation. We won't go into that, however, since

it proves to be conditionally unstable. Instead we use something called the *semi-Lagrangian* method. We have previously discussed the difference between Eulerian and Lagrangian viewpoints. As the name suggests, this method is a mix of the two, taking primarily from the Lagrangian.

Essentially, we see each cell in the grid as a newly-moved particle in a Lagrangian system. To update a given cell, $\vec{x}_G$, we first look up the velocity at the grid center. Tracing this velocity backwards, we find where the "particle" originated from and take the value from that location to write to the current cell. The method of backtracing can be as simple as a single step of forward Euler, which has been discussed previously. For increased stability we use a second-order Runge-Kutta method, as recommended in section 3.1 of [Bri08]. With Runge-Kutta we first take a half-step against the direction indicated by the current cell to get an intermediate position: $\vec{x}_{mid} = \vec{x}_G - \frac{1}{2}\Delta t\vec{u}\,(\vec{x}_G)$. We then use the velocity there to arrive at the final value.

After advecting, adding a force such as gravity is as simple as modifying both components of the MAC grid in the desired direction. After this is done, we are ready for the pressure calculations.

7.4 Incompressibility

We arrive now at the part of the simulator that most gives rise to fluid-like behavior of the interface.

Recall that we want to solve equation 7.5 on page 33 while satisfying equation 7.6 on page 33. We use u and v as the horizontal and vertical components of velocity in the mac grid, using half-indices to indicate whether we are sampling at the top, bottom, left or right edge of the cell in question. Thus, the horizontal component at cell (i, j) is designated $u_{i-\frac{1}{2},j}$. The formulas for the pressure update are as follows:

$$u^{n+1}_{i+\frac{1}{2},j} = u_{i+\frac{1}{2},j} - \Delta t\frac{1}{\rho}\frac{p_{i+1,j} - p_{i,j}}{\Delta x}, \tag{7.7}$$

$$v^{n+1}_{i,j+\frac{1}{2}} = v_{i,j+\frac{1}{2}} - \Delta t\frac{1}{\rho}\frac{p_{i,j+1} - p_{i,j}}{\Delta x} \tag{7.8}$$

Notice in particular how the pressure is calculated at cell centers by using the velocity at cell-edges for the central difference. This is the central purpose of the MAC grid.

How, then, do we calculate the pressure? This portion of the simulation is mathematically simple, but quite complex in implementation, since we are interested in solving a large linear system of equations. Fortunately, a lot of the groundwork has been done for us already. We leave the actual solving of the system to an imported mathematical library and focus simply on properly formulating the problem.

We may write the problem of finding the pressures in the grid in the following simple matrix-vector form:

$$Ap = b \tag{7.9}$$

p is what we are interested in finding, the domain-wide grid of pressures. A is a matrix of coefficients to these pressures: A has a row for each fluid-filled cell,

with each of these rows containing non-zero values only in the cells corresponding to the cell in question and its six neighbours. In this way, Ap represents the unknown pressure for each cell in the grid. The right hand side of the equation, b, is another domain-sized matrix containing the negative divergences of each cell. Recall that we are trying to minimize divergence across the fluid. This is in fact the entire point of the pressure update. The pressures we are trying to find are exactly the pressures that, when added to the velocity field, will keep the field divergence free. We won't go into the details of the construction of the A-matrix. It is technically straightforward since we know exactly where our fluid currently is.

The b matrix on the right hand side of the equation is constructed from the component velocities as follows:

```
scale = 1 / dx
loop over i,j where phi(i,j) < 0
   rhs(i,j) = -scale * (u(i+1,j)-u(i,j)
                        +v(i,j+1)-v(i,j));
```

Of course, we are only interested in the cells which contain fluid, which in our representation is everything on the negative side of the zero-level isosurface.

Finally, we relegate the solving of this system to an implementation of the Preconditional Conjugate Gradient method. This is an iterative method, the precise implementation of which lies somewhat outside the scope of this paper. In short, the result of applying this method to our constructed linear system yields the grid of pressures. Applying this to the velocity field should purge it of divergence, thereby completing this step of the algorithm.

Finally, using this divergence free velocity field, we advect the values of the signed distance field. To this end, we use the same Runge-Kutta method as described in the advection of the velocity field. Of course, having done this we need to reinitialize the signed distance field, as it likely no longer lives up the the requirements described in chapter 2 on page 5.

Having done this, we are ready to start a new iteration afresh, and thus we may iteratively evolve our interface in full accordance with the Navier-Stokes equation.

7.5 Conclusion

While there are other methods for the simulation of fluids, the mathematical simplicity of the equations used here should show that the level set method lends itself particularly well to this sort of physical simulation. Other similar methods use particles to track the surface of the fluids, but these can prove expensive and relatively inaccurate.

Care needs to be taken, however, when using level sets. If features arise in the fluid which are smaller than the size of a grid-cell, they may disappear complettely as the grid is unable to represent them. Therefore, to preserve a natural-seeming motion one should make sure that no accuracy smaller than the cell size is required.

An implementation of the level set based fluid simulator has been incorporated into the joint level set project. At the time of writing, however, the implementation is quite unstable. Currently, little regard is being given to the size of the time-step for each iteration. This causes the simulation to "blow

up" before any useful simulation has been done! Fixing the time-step to something sensible is the obvious next step for the algorithm. Section 3.3 of [Bri08] describes a method of limiting it to maximize stability.

Bibliography

[AS95] D. Adalsteinsson and J.A. Sethian. A fast level set method for propagating interfaces. *Journal of Computational Physics*, 1995.

[BMS97] R. Bowden, T. Mitchell, and M. Sahardi. Real-time dynamic deformable meshes for volumetric segmentation and visualization. In *Proc. BMVC*, volume 11, pages 310–1319, 1997.

[Bri08] R. Bridson. *Fluid simulation for computer graphics*. AK Peters Ltd, 2008.

[Int] Intel. `http://www.intel.com/cd/software/products/apac/zho/perflib/226296.htm`.

[MBZW02] K. Museth, D.E. Breen, L. Zhukov, and R.T. Whitaker. Level set segmentation from multiple non-uniform volume datasets. In *Proceedings of the conference on Visualization'02*, page 186. IEEE Computer Society, 2002.

[nVi] nVidia. `http://www.nvidia.com/object/cuda_home.html`.

[oC] Wikipedia on CSG. `http://en.wikipedia.org/wiki/Constructive_solid_geometry`.

[OF02] S. Osher and R.P. Fedkiw. *Level set methods and dynamic implicit surfaces*. Springer Verlag, 2002.

[PMO+99] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang. A PDE-based fast local level set method. *Journal of Computational Physics*, 155(2):410–438, 1999.