A Particle Level Set Library

Emud Mokberi* Petros Faloutsos†
University of California, Los Angeles

Abstract

This paper presents the implementation of a fast and accurate particle level set library. Level sets are very effective for representating dynamic surfaces and have been crucial to recent research within computer graphics, perhaps most significantly within fluid simulations. However, implementating a particle level set method is involved and requires significant effort. This paper aims to provide a publicly available library that implements a particle level set method as a blackbox. The users mainly need to provide a velocity field through which the level set is moved. The paper is self-contained. We first describe the particle level set method and the subtleties of a robust implementation. We then demonstrate the use and the effectiveness of the library with a few simple examples. The source code of the library and working examples are publicly available at our web site.

1 Introduction

The level set method was created by Osher and Sethian in 1988 and was introduced to the mainstream computer graphics community in the 2001 paper on water animation by Ron Fedkiw and Nick Foster [7]. The strength of the method lies in implicitly representing dynamic surfaces. This greatly simplifies many of the challenging problems one faces with explicit representations, such as merging and pinching of different surfaces. The main problem that the level set method suffers from is numerical dissipation. A lot of work has been done to solve this problem within the mathematics community, which mostly relies on improving the accuracy of the simulation to minimize the dissipation. The problem that this causes in computer graphics is that this improvement in accuracy comes at a high cost in computation time and memory. Further, although the bleeding caused by the numerical dissipation is slowed using these methods, it is not entirely stopped. The reason for the introduction of the particle level set method, combining Eulerian and Lagrangian methods, was to solve the problem of numerical dissipation. This hybrid method introduces marker particles to keep track of the implicit interface and uses them to correct any errors caused by dissipation. The full version of this method was introduced by Enright et al. in [5]. This method was further improved and optimized by Enright et al. in [1] by replacing the high order accuracy integration method of the level set with a fast first order accurate semi-Lagrangian integration, which was first introduced to the computer graphics community in 1999 by Jos Stam in his seminal paper [14].

The remainder of this paper is organized as follows: Section 2 describes the original level set method for background purposes. Section 3 describes the particle level set method and the subtleties that lead to a robust implementation. Section 4 describes the implementation of the library. Section 5 describes how to use the library. Section 6 shows a few examples for validation purposes. Section 7 describes further applications of the proposed library. Section 8 concludes the paper.

The library is publicly available at: www.magix.ucla.edu/software.html.

2 Level Set Method

Implicit functions are widely used in the mathematics and graphics communities for modeling complex dynamic surfaces such as the surface of water. The strength of the Level Set method is in modeling and animating implicit

*e-mail: emud@.ucla.edu
†e-mail:pfal@cs.ucla.edu

functions that dynamically change over time. The next section provides necessary background on implicit surfaces, followed by a detailed description of the Level Set method.

2.1 Implicit Surfaces

The distinguishing property of an implicit function is that the interface exists where the implicit equation evaluates to zero. In the 2D case where the interface is a curve, the curve exists where $f(x,y) = 0$. For example, if we consider the simple explicit equation of the line $f(x) = y$, the implicit equivalent would be $f(x,y) = x - y$ and the interface exists where $f(x,y) = 0$. For the purposes of computer graphics, the function is usually restricted to 3D. However, the beauty of level sets is that the methods that are developed for 1D and 2D functions can be trivially extended to handle higher dimensions. The reason for the popularity of level set method lies in part in the usefulness of implicit functions for certain applications. In particular, implicit formulations have certain important advantages over explicit or parametric formulations.

Given an space R^n, an interface Γ that exists in that space is of codimension one [13], which means that the space represented by the interface is R^{n-1}. For example, in 3D space, Γ is the surface of an object and represents a 2D space. Similarly in 2D space, Γ is a curve that represents a 1D space.

An explicit representation of a function only contains information about the interface itself. That is, given an explicit function, we can find the position of the interface, but that is it. We don't have any information about the rest of the space in which the interface exists. An implicit representation on the other hand contains information about the entire space of the function, including the position of the interface. For example, given the interface Γ representing a bounded region Ω in space, the implicit level set function $\phi(x,t)$ represents the space with the following properties,

$$\phi(x,t) > 0 \texttt{ for } x \notin \Omega, \tag{1}$$

$$\phi(x,t) = 0 \texttt{ for } x \in \Gamma, \tag{2}$$

$$\phi(x,t) < 0 \texttt{ for } x \in \Omega, \tag{3}$$

so the function doesn't just return the position of the interface Γ, but it specifies whether we are on the interface, inside the region bounded by Γ, or outside the region bounded by Γ.

The other major difference between these two methods is how we represent the function in discrete space. Since we intend to work with these objects within a computer graphics framework, we need to consider how we represent its interface. When we are dealing with an explicit object, we make use of splines or triangles for representing its interface, or surface. For a rigid body, or nearly rigid body object, this explicit framework works fine, since the connectivity of the triangles or splines stay relatively constant. However, working with this methodology becomes non-trivial when we want to represent a surface that dynamically changes topology, such as water or fire. There is no easy way of dynamically merging and splitting the surface as it changes form.

Discrete representations of implicit functions are handled using a Cartesian grid that bounds the space that the function exists in. Then we simply represent the space by storing the value of as each grid node. To find the interface itself, we just need to interpolate between nodes that have negative values for ϕ and nodes that have positive values for ϕ. Since we are not representing Γ explicitly, no special case is needed for the merging and breakup of the interface. The only thing that is required as Γ is dynamically changed, is to update the values of ϕ at the nodes to represent the current state of the space.

2.2 Geometric Operations and Signed Distance Functions

Another advantage of the implicit level set representation of space is the simplicity of performing boolean operations and constructive solid geometry operations. For example, given ϕ_1 and ϕ_2, we can easily perform the following operations:

Intersection of Ω_1 & Ω_2	$\phi = \texttt{max}(\phi_1, \phi_2)$
Union of Ω_1 & Ω_2	$\phi = \texttt{min}(\phi_1, \phi_2)$
Difference $\Omega_1 - \Omega_2$	$\phi = \texttt{max}(\phi_1, -\phi_2)$

These properties proved very useful in Museth [11].

In order to use an implicit function in a computer graphics framework, we have to be able to determine the surface normal at the interface, without which shading the object would not be possible. Fortunately, the outward unit normal of Γ is simply

$$\mathbf{N} = \frac{\nabla\phi}{|\nabla\phi|}, \texttt{ where} \tag{4}$$

$$\nabla\phi = \left(\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) \tag{5}$$

We can approximate the gradient $\nabla\phi$ using second order accurate central differences as follows:

$$\frac{\partial\phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} \tag{6}$$

and similarly for y, and z. It should be noted that the normal, $\mathbf{N}$, can be calculated at any point in space not just the interface [13].

A subset of implicit functions is the signed distance function, which is basically an implicit function where at each point in space, the value of ϕ is the signed distance to the closest point on the interface, Γ. Using a signed distance function simplifies many of the level set methods that we use to work with implicit functions. An extra property of signed distance functions is that $|\nabla\phi| = 1$. As a consequence, the calculation of the unit normal vector no longer needs to be normalized.

2.3 Integration

Given an implicit function, a framework is needed for moving it through time and space. In the level set methodology, this is accomplished by taking a velocity field $\mathbf{u}$ as input, and evolving the level set function through

$$\phi_t + \mathbf{u}\cdot\nabla\phi = 0. \tag{7}$$

Because the topology of the interface can dynamically change, it is critical that the method used for updating the level set is accurate. In [5, 13] fifth order accurate HJ WENO integration (a class of Eulerian integration) was used to move the interface, along with third order accurate Runga Kutta (RK3) time integration. Even with these high order accurate integration methods, the level set is still very susceptible to numerical dissipation, which in part has to do with the coarse grids that are required to make this simulation practical in a computer graphics environment. This problem led to the development of the hybrid particle level set method, which uses explicit particles to correct the dissipation of the implicit surface. This will be discussed further in Section 3.

The initial implementation of this level set library was based on HJ WENO and RK3 integration which were thought necessary in order to obtain accurate solutions. The only problem with using this method to move the level set is that it is computationally expensive. HJ WENO requires a range of five nodes for its computation and RK3 integration in time basically means that the interface has to be updated three times before one time update is reached. This can quickly become expensive on a three dimensional grid. Further, because of the large number of nodes needed by the HJ WENO, using acceleration structures such as an octree[10] to better represent the interface becomes very difficult.

Because of this [4] used the fast first order accurate Semi-Lagrangian advection method, introduced by Stam[14] to the computer graphics community, to update the level set. Without performing an RK3 integration in time, they found that the accuracy of this method matched the accuracy achieved using the high order accurate methods. The reason for this is simply that the particles used to maintain the volume in [5], do such a good job that high order accurate integration of the level set itself is no longer necessary.

The basic idea of the semi Lagrangian method is as follows. Given the level set function and the velocity field $\mathbf{u}$, the updated value at a point $\mathbf{x}$ after one time dt integration is:

$$\phi(\mathbf{x}) = \phi(\mathbf{x} - \mathbf{u}(\mathbf{x})dt). \tag{8}$$

The only on the semi Lagrangian integration is that dt still has to satisfy a Courant-Friedrichs-Levy (CFL) condition which means that

$$dt\left(\frac{|u_x|_{max}}{\Delta x}+\frac{|u_y|_{max}}{\Delta y}+\frac{|u_z|_{max}}{\Delta z}\right)=c \tag{9}$$

where c is the CFL restriction that dt has to satisfy. In [4] they determined that a value of 4.9 for c gave satisfactory results. It should also be noted that the higher order accurate HJ WENO integration also has to satisfy a CFL condition, and in that case c was set to 0.5, which results in a much smaller time step, further slowing the update of the level set.

2.4 Reinitialization

One of the other inherent problems caused by moving the level set through a velocity field is that it ceases to be a signed distance function, and the values for ϕ no longer vary smoothly between grid nodes. This is however remedied by performing a reinitialization on the space, which sets the signed distance values based on the updated interface.

This is also where the CFL condition, Equation 9, comes into play. Since we know that the space will be re-initialized based on the interface, we only need to advect Γ and let the reinitialization function update the rest of the space based on the new Γ. Setting c in the CFL equation to 4.9, means that at any one timestep, the interface can move at most 4.9 cells within the Cartesian grid representation. Since, the semi Lagrangian method follows the velocity field in the negative direction for its updated values, we know that we only need to look within a band of five cells on each side of the interface, to locate the new Γ. Thus, the semi Lagrangian advection does just that, and lets the reinitialization function update the rest of the space.

If the level set is being advected using a fifth order accurate HJ WENO scheme, then the surface can be re-initialized using the same HJ WENO method to advect the interface in the normal direction with a velocity of one. In this method, the location of the interface after one timestep is where the $\phi = 1$ interface lies. This method can be repeated to find all the points at a signed distance of 2 from the interface and so on. However, this can quickly become very expensive. Fortunately, just as semi Lagrangian advection was shown to be very close in accuracy to the HJ WENO scheme,[5] also showed that a fast first order accurate fast marching method coupled with the semi Lagrangian particle level set method comes close to matching the accuracy achieved by using an HJ WENO reinitialization. Because of this, we use the $O(nlogn)$ first order accurate fast marching method to re-initialize Γ.

2.5 Fast Marching Algorithm

For reinitialization, we use the Fast Marching method invented by Tsitsiklis[15] and rediscovered by Sethian and Helmsen, Puckett, Colella, and Dorr. We start by initializing a band adjacent to the interface using linear interpolation and designate these as *done* points. We then initialize every point adjacent to the band as *close* points, and determine their signed distance using the *done* points' values taking into account the size of the grid cells. We then have a band of *done* points, a band of *close* points and consider every other point a *far* point. The *close* points are stored in a min heap based on their values, and the algorithm proceeds by changing the *close* points to *done* points one at a time always working with the one which has the smallest value, and hence is closest to the interface. Every time a point changes from *close* to *done*, the points adjacent to it either have their values updated if they were already *close* points, or they are assigned a value and changed from a *far* point to a *close* point. The key however is that only the *done* points are used to determine the value of a *close* point. That is, *close* points are not used to estimate the signed distance values of other *close* points. Thus, given that the interface Γ has been accurately updated using semi Lagrangian advection and error corrected using particles, we only use the accurate values near the interface, and the distance between cells to propagate the signed distance function away from Γ. One key to speeding up this process further is only re-initializing grid values within a finite limit around the interface. The reason is that we only need to keep the grid accurate enough to represent and move the interface. Since the semi-Lagrangian method only needs values within five cells of the interface to advect it through one timestep, re-initializing grid points within six cells of the interface is enough to ensure an accurate representation of the surface. Thus after each timestep, the grid is re-initialized, providing the next timestep with accurate signed distance values within six cells of the interface.

This algorithm is extremely efficient and will reinitialize N grid points in $O(NlogN)$ time. It is however inherently first order. Initially, before the work done by [5], some effort was spent in attempting to implement a second order

accurate Fast Marching method. The main obstacle in implementing such a scheme results from the fact that the initial band adjacent to the interface is determined using linear interpolation, and extending this to second order is nontrivial. Chopp[3] uses a variant of Newton iteration to achieve a second order accurate interpolation and falls back to linear interpolation at points where this method fails. However, this proved to be unnecessary thanks to the creation of the hybrid particle level set method.

3 Particle Level Set Method

The main drawback of level set implicit surfaces represented with relatively coarse grids $(100x100x100)$ is their susceptibility to numerical dissipation, and inability to maintain areas of high curvature. When a level set is advected, the result is that all sharp edges get smoothed out and the level set looses volume. To fix this problem, [7] introduced the first version of the particle level set method. This involved introducing particles within an interior band of the interface, and using these particles to correct any volume loss that resulted from advecting the level set. In their method, the level set was updated with the velocity field $\mathbf{u}$ using Eulerian methods (HJ WENO), and the particles were separately updated using Lagrangian methods. When the interior particles escaped the interior boundary, it meant that some volume loss had occurred, and error correction was performed on the level set to maintain its volume. The problem that they still faced was that the level set would gain volume near the concave portions of the interface. Thus the full-blown particle level set method was introduced by [5] where particles where used within both an interior and exterior band of the interface, with the purpose of maintaining the interface, rather than preserving volume.

One thing that should be noted is that although particles are more accurately moved through space, they alone could not be used to represent a dynamic surface such as water or fire. The problem is that as the simulation is updated, the distribution of particles can become very uneven, resulting in unresolved areas where there simply aren't enough particles to properly represent a surface. The second problem inherent in a particle only method is that a surface still needs to be defined and doing so based on particles is non trivial. It should be noted that past work in fluid simulation has been done using particle only methods, without extracting a surface from the particles. The result is that the interface looks more like viscous sand than a smooth water surface. The elegance in the particle level set method is that it combines the accuracy of Lagrangian advection with the simplicity of the Eulerian level set surface representation.

3.1 Particle Initialization

When the initial surface is defined, particles need to be placed within a band of the interface. This is done by randomly placing particles inside of any grid cell that is within three cells [4] of the interface. The particle stores its position and its radius, which is used to perform error correction on the level set. The radius is set so that the boundary is just touching the interface:

$$r_p = \begin{cases} r_{max} & \texttt{if} \quad s_p\phi(\mathbf{x}_p) > r_{max} \\ s_p\phi(\mathbf{x}_p) & \texttt{if} \quad r_{min} \leq s_p\phi(\mathbf{x}_p) \leq r_{max} \\ r_{min} & \texttt{if} \quad s_p\phi(\mathbf{x}_p) < r_{min} \end{cases} \tag{10}$$

where s_p is the sign of the particle, set to -1 if $\phi(\mathbf{x}_p) < 0$ and $+1$ if $\phi(\mathbf{x}_p) > 0$, $r_{min} = 0.1 \times min(x,y,z)$, and $r_{max} = 0.5 \times min(x,y,z)$. In [4] they recommend that 16 particles be placed in each cell in 2D and 32 particles in each cell for 3D.

3.2 Particle Update

The particles are updated using standard Lagrangian update as follows:

$$\mathbf{x}_p(t) = \mathbf{x}_p(t-1) + dt\mathbf{u}_{t-1}(\mathbf{x}_p(t-1)). \tag{11}$$

In the original particle level set method [5, 13], the particles were updated using RK3 time integration, to match the RK3 time integration of the level set update. In the new semi-Lagrangian particle level set method [4] the level set is updated using first order time integration. However, since the particles are being relied on to accurately represent the

position of the interface, they are updated using a more accurate second order Runga Kutta (RK2) time integration, also known as the midpoint rule [13]. This simply involves performing two updates and then averaging as follows:

$$\mathbf{x}_p(t+1) = \mathbf{x}_p(t) + dt\mathbf{u}_t(\mathbf{x}_p(t)), \tag{12}$$

$$\mathbf{x}_p(t+2) = \mathbf{x}_p(t+1) + dt\mathbf{u}_{t+1}(\mathbf{x}_p(t+1)), \tag{13}$$

$$\mathbf{x}_p(t+1) = \frac{\mathbf{x}_p(t) + \mathbf{x}_p(t-1)}{2}. \tag{14}$$

3.3 Error Correction

To enable error correction, we define a level set value for each particle as follows:

$$\phi_p(\mathbf{x}) = s_p(r_p - |\mathbf{x} - \mathbf{x}_p|). \tag{15}$$

Whenever a particle escapes the interface by more than its radius, it is used to perform error correction on the interface. Error correction is performed using the positive particles to create a temporary grid ϕ^+ and the negative particles to create a temporary grid ϕ^-. For each positive escaped particle, we find ϕ_p for each corner of the cell that contains the particle. The value for each corner is then set to

$$\phi^+ = \max(\phi_p, \phi^+).$$

For each negative escaped particle, we similarly find ϕ_p for each corner of the cell that contains the particle. The value for each corner is then set to

$$\phi^- = \max(\phi_p, \phi^-).$$

The level set is then reconstructed using ϕ^+ and ϕ^- by choosing the value with minimum magnitude at each grid node:

$$\phi = \begin{cases} \phi^+ & \texttt{if} \quad |\phi^+| \leq |\phi^-| \\ \phi^- & \texttt{if} \quad |\phi^+| > |\phi^-| \end{cases} \tag{16}$$

It should be noted that since the fast marching method effects the position of the interface, error correction is used both after the level set has been updated and again after it has been re-initialized.

3.4 Re-Samping and Re-initialization

After the interface is corrected, it can be helpful to re-sample the radius of the particles based on the new interface. However, in our tests we found that this re-sampling did not visibly affect the result of the simulation. A problem that can be caused by re-sampling is that the level set correction method is not without errors, and the interface position might not be absolutely accurate after each and every timestep. When the radii are re-sampled, these errors are then introduced into the particles. Without re-sampling, we allow for the possibility that the errors will be corrected in the following timesteps based on the still accurate particle radii. Due to this, even though [4] recommends this method and our level set library supports it, we do not re-sample the radii in the simulations. Another factor to consider is that based of the input velocity field, the distribution of the particles within the space can become uneven, which is the main problem with using a particle only method. [5] hndled this problem by re-initializing the particles every 20 timesteps. The re-initialization function basically deletes all existing particles, and distributes a new set of particles based on the current position of the interface, ensuring that there is a uniform distribution of particles near the interface. Even though doing this is critical when the distribution of particles is bad, it does have a negative side effect, which is much similar to the problem caused by re-sampling the particles. The problem is that any inaccuracies in the interface representation are now transferred to the new set of particles. Because of this the tests done in [4] are done without re-initializing the particles, which results in a more accurate representation of the interface. Thus, if the simulation does not cause the particles to be unevenly distributed, there is no reason to re-initialize. If however, the topology of the interface is changing drastically through the simulation, the gains of re-initialization outweigh the errors that it introduces.

3.5 Summary

To sum up, the level set is represented using a Cartesian grid, where the interface is located at the = 0 level set. Particles are initialized within a band of the interface. Then for each simulation step:

1. A velocity field is taken as input and used to determine a CFL timestep.
2. The velocity field is used to update the level set through a fast first order accurate semi-Lagrangian advection.
3. Particles are updated using the velocity field using a RK2 time integration.
4. Error correction is done on the interface using the particles.
5. The grid is re-initialized using the fast marching method.
6. Since the fast marching method can affect the position of the interface, error correction is again performed on the interface using the particles.
7. If necessary, the particles are re-initialized.

4 Implementation

One of the most interesting aspects of the level set method is that it extends almost trivially from 2D to 3D. This is extremely helpful as debugging computational code in 3D is no small challenge. Thus the library was created and debugged in 2D and then extended to 3D free from bugs. Each version of the library consists of five main objects, those being LevelSet, ParticleSet, Particle, FastMarch, and a Container class. The ParticlesSet keeps one list for the particles instead of keeping track of the particles within each cell. The global constants that are used by the library include the grid size in each dimension, limits for particle radii, and limits for extent of semi-Lagrangian integration and re-initialization. Finally, the cell size can also be changed, but remains the same for all dimensions.

4.1 Grid Structure

A Grid object is used to represent the 3D Cartesian grid that is used by the LevelSet, and simply stores the grid as a 1D array. Besides the traditional operators, the grid has one level of buffer cells to simplify boundary checks, and has methods for setting the boundary to satisfy among others Neumann and Dirichlet conditions. The difference between the way the library represents the grid and those used by [5] is that the value of phi is store at each cell corner, instead of the center of the cell, which is how a MAC grid is typically implemented. Recent research done by [10] improves the representation of the interface by using an octree structure, and this change in the representation of the grid was done to simplify this possible extension of the library.

4.2 ParticleSet

There are some nuances to the creation of the particles that should be noted. First, particles are placed in a cell if any corner of the cell has an absolute distance of $2h$ from the interface, where h is the cell size in each dimension. Further, the library does not create positive and negative particles. Once it is determined that a cell is within the limit, particles are randomly placed in that cell. The particles themselves are responsible for keeping track of whether they are *inside* (-1) particles or *outside* (1) particles, and are set simply based on what the sign of the value of ϕ is at their position at the time of their creation. Finally, if any corner of the cell is within $0.5h$ of the interface, that cell is given twice as many particles, since those particles will be most critical in detecting error.

4.2.1 Random Number Generator

The pseudo-random number generator used for creating the particles is the *Mersenne Twister* developed by Makoto Matsumoto and Takuji Nishimura. Detailed information about this generator is available at http://www.math.keio.ac.jp/ matumoto/emt.html.

4.3 Interpolation

Because the level set function is stored as values on a Cartesian grid, we need to use interpolation between grid nodes to determine the actual position of the surface. Even though tri-linear interpolation is the default and suitable for most simulations, the library uses a monotonic cubic interpolation scheme developed by [6]. The problem with normal cubic interpolation is that it is susceptible to overshooting the data. This occurs if the four points surrounding the interpolation point are not monotonic. This new method forces the four points to be monotonic before interpolation. It should be noted that there is a typo in the paper outlining this method. The equation for a_3 is missing a factor of 2 in front of Δk. Finally, even though this method improves the accuracy for determining the value of ϕ, it is substantially more computationally expensive, and should only be used when the additional accuracy is necessary.

4.4 Visualization

Although the level set method easily extends from 2D to 3D, its visualization does not. In 2D, OpenGL is used to visualize the level set, either by creating points, or quadrilaterals. In 3D visualization can be handled in two ways. The first method is necessary if the engine requires an explicit surface representation for rendering. In this case, the level set has to first be triangulated using the marching cube algorithm [9]. This involves creating a grid over the space of the level set, marching through this grid and creating triangles whenever the sign of the level set for all eight nodes of a cell do not match. This triangulated mesh can then be passed to OpenGL, or any other rendering engine. For this implementation, we have used a marching cube method implemented by Lucio Flores for the MAGIX lab. The library also supports exporting this mesh to PovRay for high quality rendering through a template created by Ethan Drucker for the MAGIX lab. Finally, the second and perhaps better way in which visualization can be handled is through the use of a ray-tracing engine that supports intersections with signed distance functions. This is the method used in [5, 10, 12]. The problem with the marching cubes algorithm is that besides being relatively slow, it can misrepresent the surface, if small-scale details are present.

5 Using the library

The use our library one has to instantiate LevelSet, ParticleSet, and Velocity objects. The Container class, which is included with the library, provides an easy to use interface for the library. The Container object constructs the LevelSet and ParticleSet based on the desired grid dimensions and cell size. Before actual simulation, the LevelSet has to be initialized by the user, meaning a grid has to be passed in indicating where the initial interface lies. The included MakeSphere function, which creates an interface representing Zalesal's disc, is an example of this. The Grid class has to be used in setting the initial values for the interface. Once the LevelSet is initialized, the ParticleSet is seeded with particles and the simulation can then take place. Currently, the velocity field used by the Container is a constant vorticity field, and hence there is no velocity update-step in the example. The LevelSet and ParticleSet are updated at each step of simulation based on the velocity field. In order to achieve this, the user must create a class called Velocity (currently included) with a GetVelocity function, which accepts a position and returns the velocity at that position. This object is then passed to the LevelSet and ParticleSet objects and is the interface through which the velocity lookup occurs.

In terms of actual code here is what someone needs to do to use our particle levelset implementation.

1. Initialize the system. In our example the Container object defined in main.cpp holds the initial grid, a level set object and the particles. Function Container::MakeSphere() initializes the level set to correspond to the classic disk (sphere in 3D).

2. Provide an object Velocity2/3D that implements a function GetVelocity() and which is passed to the level set. This function takes a Cartesian position and returns the value of the velocity field at that position.

3. Implement a velocity update function, for example a water simulator.

4. During simulation, call the velocity update function and then Container::Update() that updates the levelSet.

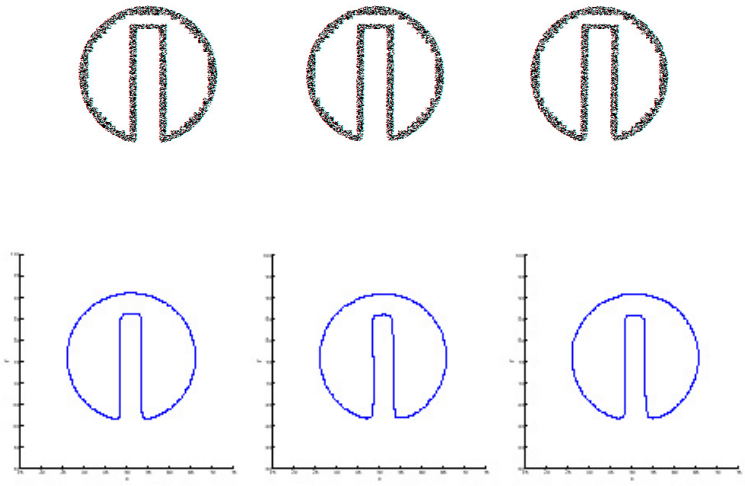

Figure 1: Comparison with the original paper.Top row: from [4], bottom row: our implementation.

Here is the how the Update function looks:

```
void Container::Update()
{
	//THE VELOCITY GRID WOULD BE UPDATED HERE. THIS
	//EXAMPLE IS USING A CONTANT VELOCITY GRID WHICH
	//CREATES A RIGID BODY ROTATION OF THE IMPLICIT SURFACE
	lset.Update(grid,dt); // update the level set
	pset.Update(grid,dt); // update the particles
	lset.Fix(pset);       // fix the level set based on the particles
	lset.ReInitialize();  // reinitialize the level set
	lset.Fix(pset);       // fix it again (see Section 3.5).
	//See Section 3.4 in the paper for activating the lines below
	//pset.Resample(lset); // resample the particles
	//count++;
	//if(count % 20 == 0) pset.Reseed(lset);
}
```

The code is self-explanatory and documented. Our particle levelset library is available at www.magix.ucla.edu/software.html for non-commercial use.

6 Tests

In order to verify the accuracy and efficiency of the library, a rigid body rotation of Zalesak's disk was performed in a constant vorticity velocity field, and was compared to the results found in [4]. In Figure 1 the top row shows results from tests done by [4]. From left to right, the figures represent the original disk, the disk after one rotation, and the disk after two rotations. These computations were performed on a 100×100 grid with a disk radius of 15 grid cells. The bottom row shows results of the same tests from our level set library. All the specifications for our test match those performed by [4], including grid size, disk size, and timestep. The area loss in both tests is $\sim 1\%$ after one revolution and $\sim 0\%$ after the second revolution.

To appreciate the particle level set method, the tests in Figure 2 were done leaving out some step from the simulation. The top row is with level set advection only, without re-initialization or particles. The middle row is with level

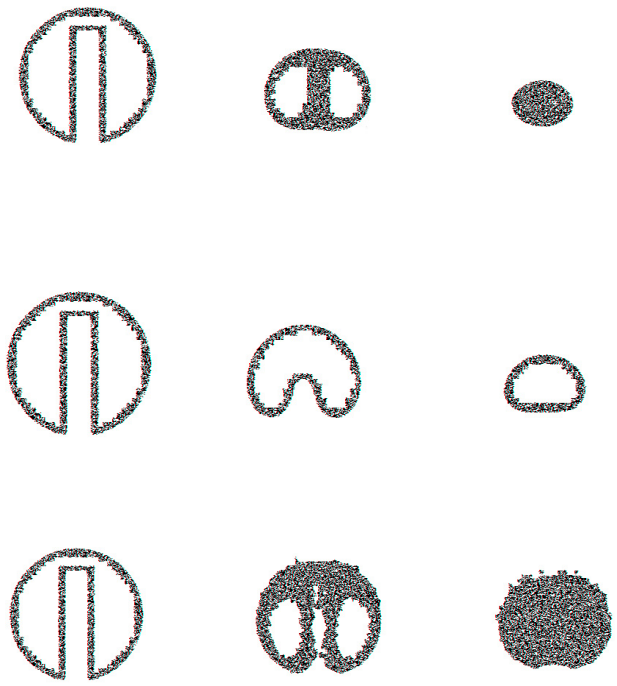

Figure 2: Two dimensional example demonstrating the effect of different stages of the particle level set method.

Figure 3: Three dimensional sphere rotating in a constant vorticity field.

set advection and re-initialization, and finally, the bottom row is with level set advection and error correction using particles, without re-initializing the level set. The specifications of the grid and the disk and number of rotations are the same as above for all images. We can see that ommiting any of these steps results in loss of accuracy and the disk changes shape.

Finally Figure 3 shows the same test run on the 3D level set library and visualized using the marching cubes algorithm and OpenGL. The images represent, in clockwise order from top left, the original sphere, after one full rotation, after two full rotations, and after ten full rotations. The grid size is $100 \times 100 \times 100$ with a radius of 15 grid cells for the sphere. The timestep is set to satisfy a CFL condition of 4.9, and there are 32 particles used within each cell. The proposed library is able to maintain the shape with almost minimum loss of shape.

7 Applications

The particle level set method has been an integral part of recent research in computer graphics. The power of using an implicit representation of objects with powerful and simple geometric tools has proven to be helpful within many sections of computer animation. It has been used to create complex CSG operations in [11], improve friction and folding of cloth in [2] and [1], and used for large-scale rigid body collision detection of nonconvex objects in [8]. Its contributions have been perhaps most significant within fluid simulation where it has been combined with the Navier-Stokes equations and the stable fluids method developed by Jos Stam [14] to create complex water animations in [5], and realistic fire simulations in [12]. It should be noted that this research has been so ground breaking that they were used to create special effects within major motion pictures within a couple years of being first introduced to the research community. Applications of the fluid simulation include ”Terminator 3” and ”The Day After Tomorrow and the cloth simulation was used in ”Harry Potter and the Chamber of Secrets.”

8 Conclusion

The level set method presented in this paper is a powerful tool for simulating dynamic objects within computer graphics animations. We hope that the availability of this tool will help the community with the implemention of complex simulations. [1]

References

[1] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics (SIGGRAPH 2002)*, 21:594–603, 2002.

[2] R. Bridson, S. Marino, and R. Fedkiw. Simulation of clothing with folds and wrinkles. In *In ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2003.

[3] D. Chopp. Some improvement of the fast marching method. *SIAM Journal of Scientific Computing*, 23:230–244, 2001.

[4] D. Enright, F. Losasso, and R. Fedkiw. A fast and accurate semi-lagrangian particle level set method. *Computer and Structures*, 2004.

[5] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of ACM SIGGRAPH 2002)*, pages 736–744, 2002.

[6] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In *Proceedings of ACM SIGGRAPH 2001*, pages 15–22, 2001.

[7] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proceedings of ACM SIGGRAPH 2001*, pages 20–30, 2001.

[8] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACm Transactions on Graphis (SIGGRAPH 2003)*, 22(3):871–878, 2003.

[9] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. In *Proceedings of ACM SIGGRAPH 1987*, pages 163–169, 1987.

[10] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23:455–460, 2004.

[11] K. Museth, D. Breen, R. Whitaker, and A. Barr. Level set surface editing operators. In *Proceedings of ACM SIGGRAPH 2002*, pages 330–338, 2002.

[12] D. Q. Nguyen, R. Fedkiw, and H. W. Jensen. Physically based modeling and animation of fire. In *Proceedings of ACM SIGGRAPH 2002*, pages 821–728, 2002.

[13] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, New York, 2002.

[14] Jos Stam. Stable fluids. In *Proceedings of ACM SIGGRAPH 1999*, pages 121–128, 1999.

[15] J. N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40:1528–1538, 1995.

[1] The code is available for non-commercial use at www.magix.ucla.edu/software.html.