Flow Simulation using Obstacle Dependent Grids

Gergely Klár*

Department of Control Engineering and Information Technology Budapest University of Technology and Economics Budapest / Hungary

Abstract

In this paper we present a method to simulate visually plausible large scale flow of fluids or smoke while maintaining real-time speed. We define the simulation over a coarse grid which is refined with more detailed grids around moving obstacles where fine details may emerge. The detailed grids also act as fixed frames of reference to the surrounded obstacles to prevent the need for working with moving boundaries in the flow.

Keywords: flow simulation, real-time simulation, level-of-details, GPU computation, Navier-Stokes

1 Introduction

Modelling the flow of gases, fluids, and smoke has been a field of interest for engineering sciences for a long time. Computational fluid dynamics is a well developed field, but application of its findings for computer graphics started only in the last decade.

Flow simulation can be used to visualise a diverse range of natural phenomena including rivers, smoke or wind.

There are two main approaches to re-create fluids or smoke for computer graphics. The particle based approach animates a large number of elements based on their interactions with each other and the environment. On the other hand, the grid based approach simulates the dynamics of the fluid or smoke in a predefined region.

While these particle based methods are well suited for simulations where there are many interactions with the environment and the flow path is complex, grid based methods are particularly important for smoke simulation. The particle based simulation of smoke would require too many particles to produce pleasing effects.

The main driving dynamics of a flow are captured by the Navier-Stokes equations, which describe the motion of incompressible flows. These equations can be decomposed to four principal terms, including *advection*, *diffusion*, *pressure*, and *external forces*.

Advection is transport in a fluid. Objects immersed in a flow are carried by the flow's advection. Advection is based on the flow's velocity field, hence the velocity field is said to be self-advected, because the changes in velocity are caused by the velocity itself.

*g.klar@creativereboot.hu

The diffusion term accounts for the diffusion of momentum in the flow. This is defined in terms of the viscosity of the fluid. This term is usually dropped in gas or smoke simulations, yielding the so called Euler-equations.

The pressure term represents forces arising from compression and expansion appearing in the flow due to advection.

Every other force interacting with the fluid or gas falls in the category of external forces.

To be used in computational fluid dynamics, the original partial differential equation is redefined in a finite difference form over grids of scalar and vector quantities. The resolution of the grid defines the amount of details that can appear in the flow.

Our paper organized as follows. First we will discuss previous works about flow simulation in Section 2. This is followed by the overview of the used simulation technique and our new obstacle dependent grids method in Section 3. Implementation details are described in Section 4. We present experiment results in Section 5, conclude and discuss future works in Sections 6 and 7.

2 Previous Works

The main challenge in flow simulation for computer graphics is presenting realistic results while keeping the grid resolution as low as possible thus reducing the required computational time. Fedkiw et al. [1] presented a method called vorticity confinement to reinsert fine details lost due to numerical dissipation. This way the grid's resolution can be decreased without losing important visual features.

Increase in accuracy can be achieved by using a staggered grid in which vector quantities are represented at cell boundaries and not in cell centres [2].

Due to the complex nature of the computations involved, running simulations fast enough to allow the user to interact with the flow in real-time presents further challenges. The evolution of modern graphics hardware gave birth to methods and principles enabling partial differential equation (PDE) solving on the GPU at much higher rates. As Harris presented in his work [3], the driving dynamics of a flow defined by the Navier-Stokes equations can be efficiently and easily computed in GPU. The highly parallel nature of the computations involved makes the GPU a really suitable and fast platform. The use of

texture images as a discretized approximations of the vector and scalar fields is a key concept for PDE solving on GPU, thus for the principle of fluid dynamics simulation on the GPU as well.

Harris' work utilizes an implicit method based on Stam's Stable Fluids [5]. This method not only provides a stable simulation for arbitrary large time-steps and velocities, but the use of an implicit methods necessary in GPU implementations, because explicit methods require storing values to cells other than the one being processed, what is not possible on current GPUs.

3 Overview

In this section first we discuss an implementation of a timestep based solver for the Navier-Stokes equations. After this we present our new method what uses a modified version of this solver to simulate large scale flows using overlaying grids.

3.1 Flow simulation on the GPU

Our purpose in fluid and gas simulations for computer graphics is the visualization of the evolution of the flow. Opposed to some engineering applications, we are not looking for a stationary solution of the Navier-Strokes equations, but we are interested the changes of the velocity field.

The Navier-Stokes equations describe the change of the velocity field with the following partial differential equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu\Delta\mathbf{u} + \mathbf{F}, \tag{1}$$

$$\nabla\cdot\mathbf{u} = 0, \tag{2}$$

where ρ is the density of the fluid, and ν is the kinematic viscosity. The terms on the right-hand side of the first equations are called *advection*, *diffusion*, *pressure*, and *external forces* terms.

A timestep based solution to these equations is presented in details in Harris' work [3]. Harris uses the *operator splitting* technique to define operations that result in an aproximation of the velocity field for the next timesteps.

The defined operators are advection ($\mathbb{A}$), diffusion ($\mathbb{D}$), application of external forces ($\mathbb{F}$) and projection ($\mathbb{P}$). Using these notations, computation of the velocity field in the next timestep $\mathbf{u}_{n+1}$ from the current velocity field $\mathbf{u}_n$ is described by the following equation:

$$\mathbf{u}_{n+1} = \mathbb{P}\circ\mathbb{F}\circ\mathbb{D}\circ\mathbb{A}(\mathbf{u}_n).$$

Each of these operators corresponds to a term on the right-hand side of equation (1), and is implemented as a fragment shader program.

The advection operator computes an approximate solution for the advection term by using the implicit method described in Stam's Stable fluids [5]. The resulting operator for r scalar or vector field advected by velocity field $\mathbf{u}$ is:

$$\mathbb{A}r(\mathbf{x}) = r(\mathbf{x} - \Delta t\cdot\mathbf{u}(\mathbf{x})).$$

The diffusion operator is defined to approximate the diffusion term of equation (1). The implicit form of the solution for this equation yields a Poisson equation:

$$(\mathbf{I} - \Delta t\cdot\nu\nabla^2)\mathbf{u}_{n+1} = \mathbf{u}_n, \tag{3}$$

where Δt is the timestep, and $\mathbf{I}$ is the identity matrix.

Harris uses a Jacobi iteration implemented in shader program to solve Poisson equations. The shader program computes $x_{i,j}^{(k+1)}$ for each cell of a discretized scalar or vector field x using:

$$x_{i,j}^{(k+1)} = \frac{x_{i+1,j}^{(k)} + x_{i-1,j}^{(k)} + x_{i,j+1}^{(k)} + x_{i,j-1}^{(k)} + \alpha b_{i,j}}{\beta},$$

where b is the right-hand side of the Poisson equation, α and β are constants. This general form of the iteration allows us to use the same shader with different constants to solve other Poisson equations.

For solving equation (3), x is $\mathbf{u}_{n+1}$, b is $\mathbf{u}_n$, $\alpha = \frac{(\Delta x)^2}{\nu\Delta t}$, and $\beta = 4+\alpha$, where Δx is the horizontal or vertical distance of the cell centres in the discretized grid. We work with equidistant grids, so these two distances are the same.

To account for the boundary conditions, when an (i',j') index refers to a cell outside of the fluid, the cell in the centre is (i,j) and $(i',j') \neq (i,j)$, then $x_{i,j}$ is used in the iteration in place of the respective $x_{i',j'}$ terms.

The operator for external forces includes all the application specific forces that affect the flow. Such forces can be gravity, buoyancy, or forces representing user interactions.

The projection operator corresponds to the pressure term of equation (1), but using the Helmholtz-Hodge decomposition theorem it can be show, that this term projects $\mathbf{w} = \mathbb{F}\circ\mathbb{D}\circ\mathbb{A}(\mathbf{u}_n)$ to a divergence-free velocity field $\mathbf{u}_{n+1}$, thus the solution for the next timestep satisfies equation (2).

The Helmholtz-Hodge decomposition theorem is also used to find a scalar field q for which $\nabla q = \frac{1}{\rho}\nabla p$. Using the theorem to get q for vector field $\mathbf{w}$ we get:

$$\nabla^2 q = \nabla\cdot\mathbf{w}. \tag{4}$$

This is also a Poisson equation, hence the aforementioned Jacobi iteration can be used to find a solution for q. For this the Jacobi iteration used with $x = q$, $b = \nabla\cdot\mathbf{w}$, $\alpha = -(\Delta x)^2$ and $\beta = 4$.

The projection operator is more complex than the ones before and requires several steps to compute $\mathbf{u}_{n+1} = \mathbb{P}\mathbf{w}$. These steps are: computing the divergence of $\mathbf{w}$, solving the Poisson equation (4) with Jacobi iteration, and subtracting the gradient of q from $\mathbf{w}$.

These steps are implemented either as shader programs, or as repeated application of a shader program in case of the Jacobi iteration.

The application of these operators results in a new divergence-free velocity field that describe the state of the fluid in the next timestep. This velocity field is then used again with the operator to continue the simulation.

3.2 Obstacle Dependent Grids

Although the speed-up gained by GPU implementation is remarkable, the attainable scale of the flow while keeping the simulation real-time is limited. Simulation of large scale flows with fine details requires high computation times when using ordinary methods.

To enable larger scales of simulated flow whilst keeping computational costs low, we define a level of detail hierarchy over the scalar and vector fields used in the equations.

Let's denote the coarse grid covering the whole field as the main grid, and denote the smaller, but refined grid as the sub-grid. The sub-grid overlays a portion of the main grid, but in such manner that several cells of the main grid are covered by the refined grid as shown in Figure 1. If the proportion of the grids were such that the sub-grid covers only a few of the main grid's cells, it would lead to sharp changes at the edges of the sub-grid.

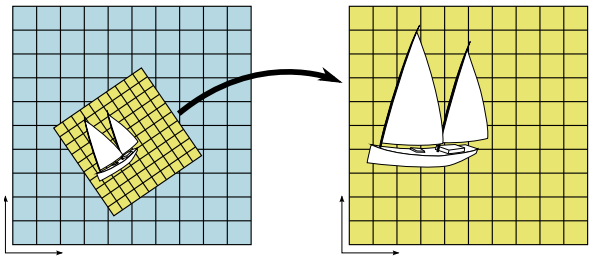

Figure 1: Relation of the main grid and the sub-grid.

The sub-grid is defined to surround the obstacles that might cause fine detailed changes in the flow. The obstacle should be fully contained in the sub-grid and additional space is required for the details to emerge between the obstacle boundaries and the sub-grid's edges. The exact placement and alignment of the obstacle in the sub-grid is arbitrary but immutable during the simulation.

The sub-grid is fixed to the object it contains, and its frame of reference follows the objects, should it be moving. This will result in a difference between the grid's and the sub-grid's frame of reference, that is addressed during computations. The benefit of fixing the frame to the object is that this way we can tackle the need to work with moving obstacles and the boundary conditions have to be calculated only once.

The result of the prior constraints is that only one object or objects moving together should be contained by a single grid. Allowing multiple, independently moving obstacles in the same grid would be contradicting to our aim to avoid the need of working with moving obstacles.

Admittedly, multiple obstacles can be enclosed by separate refined grids, but these grids should never overlap during the simulations.

Complementary Velocities

The flow in the sub-grid emerges from the difference between the frames of references. Zero velocities in respect to the main grid's frame are non-zero velocities in a moving sub-grid's frame. We address this difference by adding complementary velocities to the sub-grid. Our aim is to modify the sub-grid's velocities to counter the motion and to keep zero velocity parts of the grids still and non-zero velocity parts moving the proper direction. We have to deal with linear and circular motion of an obstacle separately, because circular motion could be approximated with linear motion only if the motion's centre is far off from the sub-grid.

In case of linear motion the complementary velocity is uniform over the sub-grid, and equals the inverse of the obstacle velocity, rotated and scaled to the sub-grid. Figure 2 illustrates this relation between the obstacle's velocity and the complementary velocities.

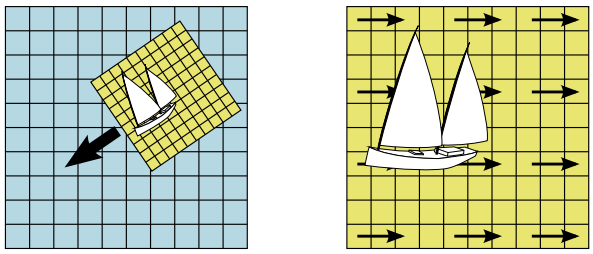

Figure 2: Linear motion of an obstacle (left) and the required complementary velocities (right).

In case of circular motion the complementary velocity is derived from the tangential velocity for each cell of the sub-grid as shown in Figure 3. Section 4.2 details the computations of this velocity.

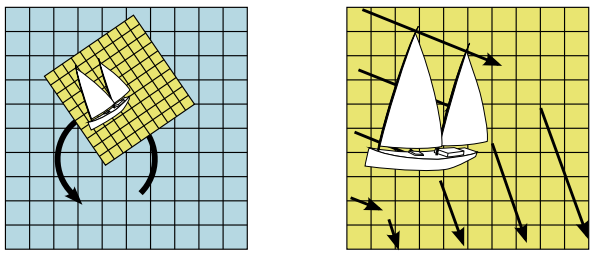

Figure 3: Circular motion of an obstacle (left) and the required complementary velocities (right).

4 Implementation

The GPU implementation follows the GPGPU concepts described in Harris' work [4]. The grids acting as vector and scalar fields are represented using textures with texels

corresponding to the cells centres of the grids. Since each texel is a four dimensional vector, vector field calculations can be done with ease and each texture can be used to store up to four scalar fields. Figure 4 demonstrates a texture containing values of a two dimensional velocity field.

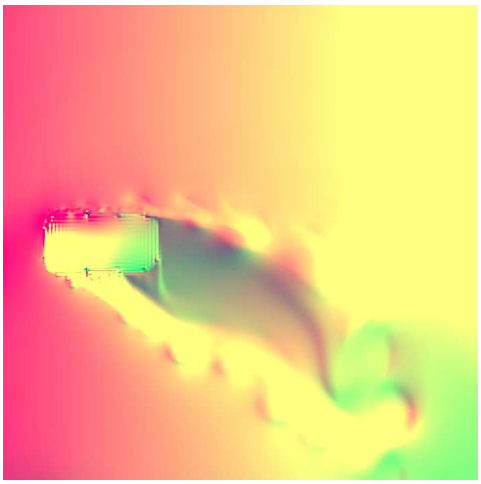

Figure 4: Velocity field texture example.

The computation steps are done on the GPU by fragment programs. To invoke the computations, a full-screen quad is rendered with a texture set as render target. The results will be stored in the specified render target. The fragment program calls will happen on a per-pixel basis, therefore the quad's texture coordinates have to be set up so that each texel maps perfectly to a single pixel. Failing to do so will result in interpolated values getting stored in the result texture. To prevent this, a half pixel sized offset should be applied to the quad's texture coordinates.

The GPU provides features on the hardware that would require additional calculations in a CPU implementation. These features include bilinear interpolations for texture reads for addresses between texel centres. Exploiting this feature yields more precise numerical simulation. Handling of texture addresses outside a texture's range can also be done automatically by the hardware, so there is no need to define separate cases for computations of inner and boundary cells.

4.1 Simulation Steps

Each simulation step consists of the following operations: simulation step on the main grid, then simulation step on the sub-grid, and finally sub-grid to main grid feedback. Listing 1 details these operations. Each of these operations are described in the following sections.

The computations on the main grid are independent from the sub-grid, therefore the advection and projection operations can be implemented with the programs described by Harris [3]. Similarly, we do not have to take the main grid into account for the projection operation of the

```
for each advectant of the main grid:
  advect(advectant);
advect(velocity);
project();

for each advectant of the sub-grid:
  advectOnSubGrid(advectant);
advectVelocityOnSubGrid(subGridVelocity);
correctVelocities();
projectOnSubGrid();

updateTime();
updatePosition();

feedback();
```

Listing 1: Detailed simulation step

sub-grid. Therefore projections of the main grid and the sub-grid differ only in parameters.

The advection of velocity differ from advection of other carried quantities in the sub-grid. The sub-grid's advections depend on the main grid, and velocities read from the main grid have to be converted to the sub-grid's frame of reference and be modified by the complementary velocities.

In each simulation step the sub-grid's velocities have to be readjusted after advection, because advection corrupts the complementary velocity components.

4.2 Velocity Correction

At each change of the obstacle's velocity, angular velocity or rotational centre the complementary velocities have to be readjusted. This is done by subtracting the old, then computing and adding the new complementary velocities to the sub-grid's values.

Let $\mathbf{w}$ denote the complementary velocity. This vector $\mathbf{w}$ is simply the scaled and rotated inverse of the obstacle velocity for linear motion. Because $\mathbf{w}$ is uniform over the sub-grid, advection does not affect this component of the velocities, so there is no need for correction after advection during linear motion. Vector $\mathbf{w}$ is computed using the following equation:

$$\mathbf{w} = \text{rotate}(-\mathbf{v}/scale, -\phi),$$

where $\mathbf{v}$ is the obstacle's velocity, $scale$ is the sub-grid/main grid side ratio, and ϕ is the angle of the sub-grid's and main grid's axes. Because handling linear motion is quite straightforward, our further discussion focuses mainly on circular motion.

In case of circular motion, vector $\mathbf{w}$ is not uniform over the sub-grid, but changes with the distance from the centre of rotation in each cell. Because our flow dynamics' advection term uses an implicit method to transport velocities, vector $\mathbf{w}$ has to be defined for cell position $\mathbf{p}$ in the

grid so that $\mathbf{p}-\mathbf{w}$ has the same distance from the centre of rotation as $\mathbf{p}$ does.

To achieve this, instead of using the tangential velocities derived from the angular velocity we use the following equation. Let $\mathbf{c_r}$ denote the centre of rotation, ω the angular velocity, Δt the timestep of the simulation, and let $\mathbf{r} = \mathbf{p} - \mathbf{c_r}$.

$$\mathbf{r}' = \text{rotate}(\mathbf{r}, -\omega\Delta t),$$
$$\mathbf{w} = (\mathbf{r} - \mathbf{r}')\frac{1}{\Delta t}.$$

Note that $\mathbf{w}$ is not tangent to the circle of motion, therefore not parallel to the tangential velocity. Tracing back along $\mathbf{w}$ from $\mathbf{p}$ by Δt, as it happens in advection, yields a point on the circle.

Since advection of a field $\mathbf{u}$, either scalar or vector, at point $\mathbf{p}$ is defined as $\mathbf{u}_{new}(\mathbf{p}) = \mathbf{u}(\mathbf{p} - \Delta t\mathbf{v}(\mathbf{p}))$, the quantities if $\mathbf{u}$ will be carried along perfectly in circles defined by the centre of rotation if it is not affected by other velocities.

The advection carries the velocity from the back-traced cell to the one being processed. The magnitudes of the original and the back-traced velocities are the same, but their directions are different. To account for this, the velocities of the sub-grid have to be rotated toward the centre of the circular motion as if it were centripetal acceleration. This rotation is defined by the following equation:

$$\mathbf{w}' = \text{rotate}(\mathbf{w}, \omega\Delta t).$$

This step can be viewed as rotating the velocity field's vectors to follow the rotation of the grid.

4.3 Advection

The main difference between the advections on the sub-grid and on the main grid is that the sub-grid's advection takes the main grid's values into account too. Whenever the implicit scheme's back-traced position is not inside the sub-grid, the corresponding position of the main grid is used.

In Listing 2, we show the fragment programs used for advection. In both cases first a check is made to see if the processed cell is inside the obstacle. If it is, then zero value is returned. Next, if the back-traced position is inside the sub-grid, then the value in given position multiplied by the dissipation term is returned.

If the back-traced position is outside of the sub-grid, the corresponding main grid position is computed using `local2world`, and the value of the main grid's advected field is read from the resulting position. While this value can be used directly if the advected quantity is not the velocity, but further steps are needed if it is. The function `chordVelocityAt` returns the complementary velocity for circular motion, as described in the previous section. Additionally the velocity have to be converted to the sub-grid's frame of reference, what is done by the `worldV2localV` function.

```
float4
  psAdvectVelocityOnSubGrid(vsOutput input)
  : COLOR0
{
  float b = tex2D(boundary, input.tex).w;
  float2 pos;
  float4 ret;
  if (b == 1.0)
    ret = 0;
  else
  {
    pos = input.tex - timestep *
      tex2D(velocity, input.tex);
    float2 posC = clamp(pos, 0,1);
    if (posC.x == pos.x && posC.y == pos.y)
      ret = dissip*tex2D(advectant, pos);
    else
    {
      pos = local2world(pos);
      ret = tex2D(mainGridVelocity, pos);
      ret.xy = worldV2localV(ret.xy);
      ret.xy += chordVelocityAt(pos,
        centre, angularVelocity);
    }
  }
  return ret;
}

float4
  psAdvectOnSubGrid(vsOutput input)
  : COLOR0
{
  float b = tex2D(boundary, input.tex).w;
  float2 pos;
  float4 ret;
  if (b == 1.0)
    ret = 0;
  else
  {
    pos = input.tex - timestep *
      tex2D(velocity, input.tex);
    float2 posC = clamp(pos, 0,1);
    if (posC.x == pos.x && posC.y == pos.y)
      ret = dissip*tex2D(advectant, pos);
    else
    {
      pos = local2world(pos);
      ret = dissip*tex2D(mainGridAdv, pos);
    }
  }
  return ret;
}
```

Listing 2: Modified advections implemented as fragment programs

```
float4
  psTransformVelocity(vsOutputPass input)
  : COLOR0
{
  float2 v = tex2D(velocity, input.tex);
  v -= chordVelocityAt(input.tex,
    relCentre, angularVelocity);
  v = localV2worldV(v);

  return float4(v,0,0);
}
```

Listing 3: Fragment program used for velocity transformation before feedback

4.4 Feedback

Because every level of the hierarchy on the whole depends on an other one, efficient feedback of the simulated quantities is crucial.

Feedback from the sub-grid to the main grid can be done with ease by exploiting the fact that the simulation takes place on the GPU. For feedback, for each cell of the main grid the corresponding cells of the sub-grid should be determined, their values filtered, for example, by averaging, and should be registered back to the main grid. Fortunately this task can be committed to the graphics hardware by rendering a full-screen quad for the main grid and then rendering a scaled and properly aligned quad for the sub-grid. This can be done by a simple pass-through fragment program for the scalar fields such as inks or smoke densities. The render target's resolutions should be the same as the main grid's. The new data is the result of the current simulation step and should be used in the subsequent step for the main grid.

However, in case the advected field is the velocity field itself, an additional step is needed to transform the sub-grid's velocities to the main grid's frame of reference. For this step we render the velocity field to a temporary texture using the fragment program shown in Listing 3. First the complementary velocity is subtracted from the sub-grid's velocity then it is converted to the main grid's frame of reference using the `localV2worldV` function, what is the inverse of `worldV2localV` used during advection.

4.5 Numerical Diffusion

The discrete nature of the methods results in diffusion of the advected quantities, particularly of velocities. This leads to a serious problem if the surrounded obstacle is making circular motion. Considering only the complementary velocities, back-tracing along a velocity vector will mark a point of the same distance away from the centre of the rotation as the original point, but this new point will coincide with a cell centre only in the rarest cases in real use. Off-centre values read from a texture will be interpolated from the ones surrounding it by the graphics hardware. While this is appealing in general and improves numerical accuracy, in this very case the interpolation will result in velocities that will cause the advected quantities to spiral inwards.

To resolve this problem, the advected velocities should be boosted by a small amount, using the dissipation term in advection. The exact boosting rate heavily depends on the other parameters of the simulations, but they can be found easily by experimenting.

5 Results

The main achievement of our work is that large scale flows with relatively small moving elements can be simulated without a demanding simulation of large scalar and vector fields while preserving fine details.

To evaluate the performance of our obstacle dependent grids method we made several experiments. First, to determine a reference value for the frame rate, we ran the simulation on a single, 512×512 grid. This simulation ran at 7 frames per second. Then we ran the experiment using hierarchic grids of several size ratios and resolutions. The size ratio defined the resolution of the sub-grid, so the sub-grid could be considered as a portion of the grid of the reference experiment.

The experiments consisted of simulating a scenario with a single fine grid, then making the same simulation, but this time using hierarchical grids. We tested the hierarchical grids method with different size ratios, while examining the frame rates of the simulation.

Ratio	Sub-grid	Main grid	FPS
1/2	256×256	256×256	19
1/2	256×256	128×128	28
1/4	128×128	256×256	32
1/4	128×128	128×128	60

Table 1: Performance results.

The observed results are displayed in Table 1. The columns of the table represent, from left to right, (i) the size ratio of the main grid and the sub-grid, (ii) the resolution of the sub-grid, which directly arises from the size ratio, (iii) the resolution of the main grid, and (iv) the observed frame rate.

An example of simulated smoke disturbed by an object can be seen in Figure 5. Both the main grid's and the sub-grid's resolution is 256×256. The main grid's side is five times longer than sub-grid's, therefore the same simulation without using hierarchical grids would require processing of grids of size 1280×1280.

6 Conclusions

In this paper we presented a method to improve the performance of fluid simulations when the simulated domain

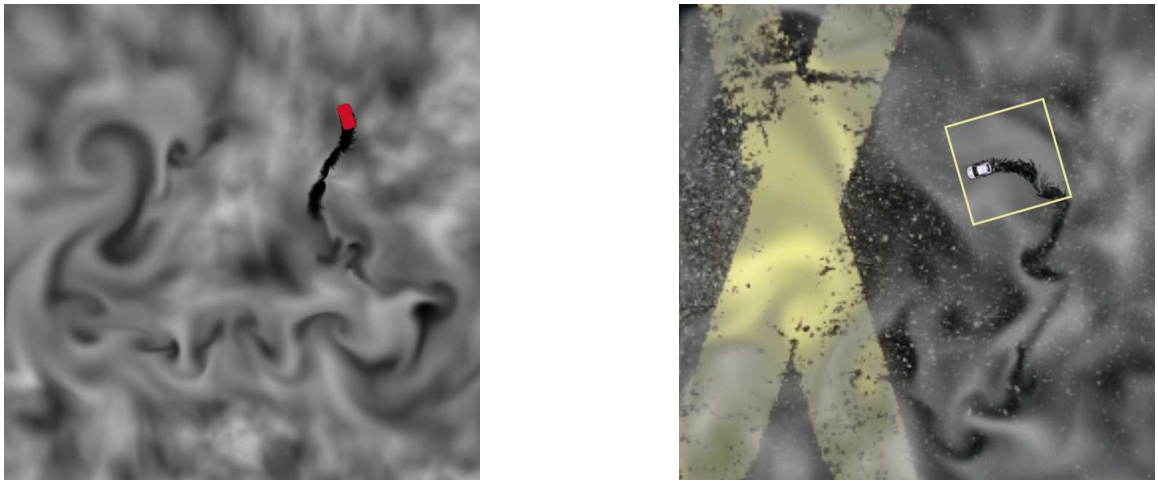

Figure 5: Smoke animated using hierarchical grids. Left: the smoke density field during simulation, the obstacle is marked with red. Right: simulation composed to a simple scene with the area covered by the sub-grid indicated.

is significantly larger, than the obstacles causing disturbances in the flow. We established our method on the assumption that fine details will emerge near these obstacles, and these details dissipate further away from the obstacles. Using this assumption we formulate the simulation using a low resolution grid for the whole domain and a high resolution grid for the the proximity of the obstacles.

Our experiments showed that using this method resulted in notable faster simulations rates while losing relatively small amount of detail.

7 Future work

The algorithms presented in this paper are defined for two dimensional flows but their main ideas are independent from this constraint.

Simulation of three dimensional flows is of much interest nowadays and due to its high computational cost simplification methods are worth being researched.

Utilizing this technique for three dimensional flows would provide significantly more performance gain than for two dimensional flows. Providing means to reduce the required grid resolution in two dimensions would reduce the solution time and space requirements by the second power, but doing likewise in three dimensions would reduce by the third power.

For today's hardware this would not only make a difference on the highest achievable speed and resolution, but it would determine the usability of three dimensional flow in applications.

8 Acknowledgements

This work has been supported by the National Office for Research and Technology (Hungary), by OTKA, and by the Croatian-Hungarian Action Fund.

References

[1] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM Press. http://doi.acm.org/10.1145/383259.383260.

[2] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. ISBN 0-89871-398-6.

[3] Mark Harris. Fast fluid dynamics simulation on the GPU. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM Press. http://doi.acm.org/10.1145/1198555.1198790.

[4] Mark Harris. Mapping computational concepts to GPUs. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM Press. http://doi.acm.org/10.1145/1198555.1198768.

[5] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. http://doi.acm.org/10.1145/311535.311548.

[6] L. Szirmay-Kalos, Gy. Antal, and F. Csonka. *Háromdimenziós grafika, animáció és játékfejlesztés*. ComputerBooks, 2003. ISBN 9-636-18303-1.

[7] Stefan Turek. *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*. Springer, Berlin, 1999. ISBN 3-540-65433-X.