Fluid Simulation with Finite Element Method

Philipp Erler*, Christian Hafner†
TU Wien

December 18, 2016

Contents

1 Introduction

The physical properties of fluids can be described with the Navier-Stokes equations. This text is a report on the practical project for computer graphics. The described system uses the finite element method to calculate the pressure that is required to keep the fluid divergence free. The results show that the current system essentially works but looses fluid through several issues.

2 Documentation

This section describes all functions of the fluid simulation with their interfaces and meaning. Basically, the simulation works as follows. At first, the initial state is loaded. Then, the free-slip boundary conditions are set meaning that velocities at fluid-solid boundaries normal to the interface are set to zero. In the main loop, external forces are applied, the distance and velocity fields are advected, and the pressure is calculated and subtracted from the velocity field.

2.1 Main Control

These functions handle the basic initialization and call flow.

*ph.erler@gmx.net
†christian.hafner@tuwien.ac.at

2.1.1 main.m

This is the entry point of the fluid simulation. This script calls the simulation and saves the results. Also, all parameters for the simulation are specified here in two structs, one for simulation settings and one for visualization settings. You can enable profiling by setting the variable 'enableProfiling' to true.

2.1.2 mainUnitTest.m

This is the entry point of the unit tests for the fluid simulation. The individual unit tests are sub-functions in this file. Currently there are tests for the gradient calculation, expanding blob, and boundary test.

2.1.3 runSimulation.m

function [divergence, fluidPixels] = runSimulation(simulationSettings, visualizationSettings)

- return divergence: vector(numFrames - 1) sum of divergence
- return fluidPixels: vector(numFrames - 1) num fluid pixels
- simulationSettings: struct with these fields:
  - xCells: grid cells in x direction
  - yCells: grid cells in y direction
  - cellSize: grid cell size in m
  - noSteps: number of simulation steps as integer
  - deltaTime: passed time per step in seconds
  - advectionAlgorithm: string, one of these 'semilagrange', 'maccormack'
  - advectionInterpolation: string, one of these 'spline', 'cubic', 'linear', 'nearest'
  - dissipation: friction within fluid
  - scenario: string as defined in loadScenario
- visualizationSettings: struct with these fields:
  - outputFolder: folder where the resulting video should be saved
  - gridCellSize: inflate cells to pixels
  - isobarWidth: width of the isobars (areas with similar distance)
  - drawVelocity: bool, display velocity field
  - distanceRefreshInterval: integer, distance field is reinitialized after n frames

This function performs the fluid simulation. It returns the quality measures, the number of fluid pixels and the sum of divergence. Additionally, it saves each frame as an image in the output folder. It loads the chosen scenario, applies the boundary condition and goes into the main loop. The main loop consists of the following steps:

1. apply external forces to the velocity field
2. delete velocities in solids
3. calculate pressure field
4. subtract pressure gradient
5. set air velocities
6. advect distance field
7. advect velocity field
8. re-initialize distance field
9. draw and save current frame
10. save quality measures

2.1.4 loadScenario.m

function [distanceField, velocityField, pressureField, obstacle, forceField] = loadScenario(simulationSettings)

- return distanceField: matrix(yNodes, xNodes), distances to interface
- velocityField: matrix(yNodes, xNodes, dim), velocity of fluid in m/s
- pressureField: matrix(yNodes, xNodes), pressure at each node
- obstacle: bool matrix(yNodes, xNodes), node is in obstacle (no fluid or air)
- forceField: matrix(yNodes, xNodes, dimension), force applied to fluid in m/s^2
- simulationSettings: struct with these fields:
  - xCells: grid cells in x direction
  - yCells: grid cells in y direction
  - cellSize: grid cell size in m
  - scenario: string, one of the construction functions below
  - deltaTime: passed time per step in seconds

This function loads the scenario specified in 'simulationSettings.scenario'. The content of this string must be the name of a scenario creation function, for example 'twoBlobs'. To add a new scenario, create a new function in this file and use its name in the 'simulationSettings' struct in the main.m. The scenarios currently available are the following:

- advectionTest: In this scenario, a quadratic blob moves constantly along a ruler (solid every 10th cell). Use 99x99 cells with a domain size of 1m. Then, the blob should move 0.4m (4 solid spaces) in 1s. Here, you can see nicely how much fluid loss is caused by the advection.
- expandingBlob: This scenario features a single blob with radial velocity field. The pressure projection should ensure that the blob volume stays constant, thus setting the velocity to approximately zero.
- restingBlob: This scenario contains a blob initially filling a basin with applied gravity, which shows the stability of the simulation.
- fallingBlob: In this scenario, a blob falls into a basin. This is the classic scenario for fluid simulations.
- fallingCells: This scenario is like fallingBlob but the blob is initially already at the bottom of the basin with a constant velocity down. In the first frame, you can observe the collision with the solid. Use a small grid like 9x9 cells to analyze the simplest fluid-solid collision.
- manyBlockers: This scenario features several layers with small holes. This scenario shows nicely how accurate the boundary handling is.
- funnel: This scenario contains a blob falling into a narrowing basin. It is helpful to see the handling of diagonal fluid-solid interfaces.
- singleBlocker: This scenario has a blob falling onto a blocker and then into a basin.
- twoCollidingBlobs: In this scenario, two blobs are colliding with mirror-inverted constant velocity.
- stretchingBlob: This scenario features a single blob whose left and right sides move towards the center. The velocity is a linear gradient.
- singleCollidingBlob: This scenario is like stretchingBlob but with constant velocity on the left and right.
- singleBlobInTrap: This scenario contains a blob moving to the bottom left into a basin.
- singleBlobInTrapWithBlocker: In this scenario, a blob is falling onto a blocker and into a basin.

2.2 Advection

These functions handle the fluid transportation within the domain.

2.2.1 applyForces.m

function [newVelocityField] = applyForces(velocityField, forceField, deltaTime)

- return newVelocityField: matrix(yNodes, xNodes, dim)
- velocity: matrix(yNodes, xNodes, dim)
- forceField: matrix(yNodes, xNodes, dimension), force applied to fluid in m/s^2
- deltaTime: float

Apply external forces to a given velocity field. Currently only used for a constant gravitational acceleration.

2.2.2 setAirVelocity.m

function [correctedVelocity] = setAirVelocity(velocity, distanceField, obstacle)

- return correctedVelocity: matrix(yNodes, xNodes, velocityInDim)
- velocity: matrix(yNodes, xNodes, velocityInDim)
- distanceField: matrix(yNodes, xNodes)
- obstacle: boolean matrix(yNodes, xNodes)

This function corrects velocities in the air. It copies the value of the nearest fluid cell. The path to the fluid cell cannot go through a solid node.

2.2.3 advectionCore.m

function [advectedSource] = advectionCore(targetCoords, source, interpolationAlgorithm)

- return advectedSource: matrix(yNodes, xNodes, [dim]), advected state of the source matrix, velocity or distance field
- targetCoords: matrix(yNodes, xNodes)
- source: matrix(yNodes, xNodes, [dim]), velocity or distance field
- interpolationAlgorithm: string, one of these: 'spline', 'cubic', 'linear', 'nearest'. 'linear' causes a major loss of fluid. 'spline' minimizes the loss.

This function advects a source value field to given target coordinates. This function takes care of the bounds by keeping the target coordinates within the value field size. The variable 'interpolationAlgorithm' is a string used for 'interp2'. Valid values are 'spline', 'cubic', 'linear', 'nearest'. Advection with linear interpolation causes a loss of fluid, even if a blob of fluid moves with constant velocity. Spline minimizes this loss but can make the simulation unstable. 'Linear' is usually the best, especially since MacCormack's method is made for (bi-/tri-)linear interpolation.

2.2.4 advectSemiLagrange.m

function [advectedSource] = advectSemiLagrange(simulationSettings, velocity, source)

- return advectedSource: matrix(yNodes, xNodes, [dim]), advected state of the source matrix, velocity or distance field
- simulationSettings: struct with these fields:
  - cellSize: grid cell size in m
  - deltaTime: time passed per step in seconds
  - dissipation: friction within fluid
  - advectionInterpolation: string, one of these: 'spline', 'cubic', 'linear', 'nearest'. 'linear' causes a loss of fluid. 'spline' minimizes the loss but introduces instability.
- velocity: matrix(yNodes, xNodes, dim) in m/s
- source: matrix(yNodes, xNodes, [dim]), velocity or distance field

Apply Semi-Lagrangian advection on source data with the velocity field. In short, this is a interpolation with target coordinates = gridCoordinates - velocityField * dt.

2.2.5 advectMacCormack.m

function [advectedSource] = advectMacCormack(simulationSettings, velocity, source)

- return advectedSource: matrix(yNodes, xNodes, [dim]), advected state of the source matrix, velocity or distance field
- simulationSettings: struct with these fields:
  - cellSize: grid cell size in m
  - deltaTime: time passed per step in seconds
  - dissipation: friction within fluid
  - advectionInterpolation: string, one of these: 'spline', 'cubic', 'linear', 'nearest'. 'linear' causes a major loss of fluid. 'spline' minimizes the loss but introduces instability.
- velocity: matrix(yNodes, xNodes, dim) in m/s
- source: matrix(yNodes, xNodes, [dim]), velocity or distance field

Apply advection based on MacCormack's Scheme [1] on source data with the velocity field. This algorithm is much more precise than the normal Semi-Lagrangian advection, which results in more fluid conservation.

The calculation steps are in short:

1. normal semi-lagrangian advection from current state
2. backward semi-lagrangian advection from (1)
3. res = (1) + (currentState - (2)) / 2
4. clamp with values around (1)

2.3 Pressure Projection

These functions handle the pressure projection. They are meant to calculate the pressure from a velocity field, compute its gradient and then subtract it from the velocity field. This ensures the incompressibility of the fluid.

2.3.1 solvePressure.m

- return pressure: matrix(yNodes, xNodes)
- simulationSettings: struct with these fields:
  - cellSize: grid cell size in m
- velocity: matrix(yNodes, xNodes, dim)
- distanceField: matrix(yNodes, xNodes)
- obstacle: boolean matrix(yNodes, xNodes)

Calculate the pressure from a velocity field. This pressure solver uses the Finite Element Method (FEM). See 'Pressure Solve with Finite Elements' by Christian Hafner for the details. In short, the algorithm works like this:

1. for each fluid cell (mixed cells also count as fluid cells)
   (a) calculate local influence matrix and vector with Gauss quadrature
   (b) add local matrix to global influence matrix
2. remove air nodes from the global matrix and vector because the pressure in the air must be zero
3. calculate pressure by solving the matrix with the vector

2.3.2 subtractPressureGradient.m

function [newVelocityField] = subtractPressureGradient(velocityField, pressure, obstacle)

- return newVelocityField: matrix(xNodes, yNodes, dim)
- velocityField: matrix(yNodes, xNodes, dim)
- pressure: matrix(yNodes, xNodes)
- obstacle: boolean matrix(yNodes, xNodes)

Subtract pressure gradient from the velocity field.

2.3.3 calcGradient.m

function [gradient] = calcGradient(pressure, obstacle)

- return newVelocityField: matrix(yNodes, xNodes, dim)
- pressure: matrix(yNodes, xNodes)
- obstacle: boolean matrix(yNodes, xNodes)

Calculate the gradient of a pressure field with finite differences. Use the central difference within the fluid and forward / backward differences at fluid-solid interfaces.

2.3.4 enforceBoundaryCondition.m

function [newVelocityField] = enforceBoundaryCondition(velocityField, obstacle)

- return newVelocityField: matrix(yNodes, xNodes, dim)
- velocityField: matrix(yNodes, xNodes, dim)
- obstacle: boolean matrix(yNodes, xNodes)

This function ensures that the velocity in normal direction from the solids is zero. This is the free-slip boundary condition.

2.4 Utility Functions

2.4.1 cellsToNodes.m

function [nodes] = cellsToNodes(cells)

- return nodes boolean matrix(yCells + 1, xCells + 1)
- cells: boolean matrix(yCells, xCells)

Convert cells to nodes. Both are logical matrices. A single true cell incident to the node suffices to set the node to true. This prevents fluids and solids from being only a single node wide.

2.4.2 distanceFieldToFluidNodes.m

function [fluidNodes] = distanceFieldToFluidNodes(distanceField)

- return fluidNodes: boolean matrix(yNodes, xNodes)
- distanceField: matrix(yNodes, xNodes)

Convert a distance field to a boolean matrix denoting the fluid nodes. This function assigns a node to the fluid if its distance is greater or equal zero. The inverse function is 'fluidNodesToDistanceField'.

2.4.3 fluidNodesToDistanceField.m

function [distanceField] = fluidNodesToDistanceField(fluidNodes)

- return distanceField: double matrix(yNodes, xNodes)
- fluidNodes: boolean matrix(yNodes, xNodes)

Convert a boolean matrix denoting fluid nodes to a distance field. This function is the inverse of 'distanceFieldToFluidNodes'.

2.4.4 getFluidCells.m

function [fluidCells] = getFluidCells(distanceField)

- return fluidCells: boolean matrix(yCells, xNodes)
- distanceField: matrix(yCells, xNodes)

Convert a distance field to a boolean matrix. Fluid cells must contain at least one fluid node.

2.4.5 getInterfaceNodes.m

function [interfaceNodes] = getInterfaceNodes(distanceField, obstacle, onlyAir)

- return interfaceNodes: boolean matrix(yNodes, xNodes)
- distanceField: matrix(yNodes, xNodes)
- obstacle: boolean matrix(yNodes, xNodes)
- onlyAir: boolean scalar

Get nodes at the fluid-air interface. Fluid-solid interfaces don't count. If 'onlyAir' is true, only air nodes at the fluid-air interface are returned.

2.5 Visualization

These functions are used to create a video from the simulation results.

2.5.1 reinitDistances.m

function [newDistanceField] = reinitDistances(distanceField, obstacle)

- return newDistanceField: matrix(yNodes, xNodes)
- distanceField: matrix(yNodes, xNodes)

Re-initialize the distance field. Nodes at the interface keep their distance. Without this function, the interpolation in the advection would cause the distances to blend.

2.5.2 calcFrame.m

function [frame, fluidPixels] = calcFrame(visualizationSettings, distanceField, obstacle, velocity, pressureField)

- return frame: matrix(yNodes * gridCellSize, xNodes * gridCellSize, colorChannel) fluidPixels: scalar, number of fluid pixels
- visualisationSettings: struct with these fields:
  - gridCellSize: number of pixels the cell should cover
  - isobarWidth: width of the isobars (areas with similar distance)
  - drawVelocity: bool should display velocity field
  - drawPressure: bool should display pressure field
- distanceField: matrix(yNodes, xNodes)

- obstacle: boolean matrix(yNodes, xNodes)
- velocity: matrix(yNodes, xNodes, dim)
- pressureField: matrix(yNodes, xNodes)

Generate an image out of the distance field and obstacles using the visualization settings. If desired, the velocity and pressure field can be added as overlay.

2.5.3 saveImage.m

function saveImage(frame, noFrame, outputFolder)

- frame: matrix(yRes, xRes, colorChannel)
- noFrame: simulation step the frame belongs to
- outputFolder: the folder in which the images are to be saved

Save the given frame as an image in the output folder.

2.5.4 saveVideo.m

function saveVideo(visualizationSettings, deleteImages)

- visualisationSettings: struct with these fields:
  - outputFolder: folder in which the resulting video is to be saved
  - fps: frames per second of the video
  - outputFile: the name of the resulting video
- deleteImages: bool, delete or keep source images

Create a video out of all .png images in the output folder.

2.6 Prototyping

These functions are prototypes for a future implementation of the classical fluid simulation with finite differences and divergence solving.

2.6.1 jacobi.m

function [newPressure] = jacobi(oldPressure, divergence, obstacle)

- return newPressure: matrix(yNodes, xNodes)
- pressure: matrix(yNodes, xNodes)
- divergences: matrix(yNodes, xNodes)
- obstacle: boolean matrix(yNodes, xNodes)

Calculate the pressure to minimize the divergence with the Jacobi iteration.

2.6.2 calcDivergence.m

function [divergence] = calcDivergence(velocity, obstacle)

- return divergence: matrix(yNodes, xNodes)
- velocity: matrix(yNodes, xNodes, dim)
- obstacle: boolean matrix(yNodes, xNodes)

Calculate the divergence from a velocity field.

3 Experiments

This section describes all performed experiments. Five scenarios are compared using different grid resolutions and different time steps.

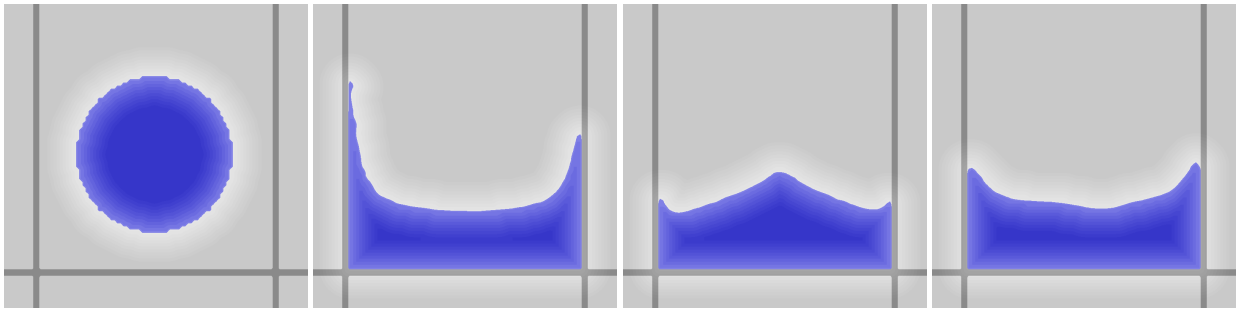
3.1 Scenarios

This section describes all simulated scenarios and how to reproduce the results. The scenarios are: falling blob, single blocker, many blockers, funnel, and resting blob. All scenarios are calculated with the following settings:

- Domain size = 1 m
- Simulation time = 10 s
- Advection algorithm = MacCormack's
- Advection interpolation = spline
- Grid cell size = 800 / numCells
- Isobar width = 400 / numCells
- Distance refresh interval = 1

3.1.1 Falling Blob

In this scenario, a blob of fluid falls into an empty rectangular basin. When the blob collides with it, the fluid gushes to the bottom left and right corners of the basin. The velocities there are very high leading, meaning several cells can be passed in a single frame. The pressure reaches its maximum in these corners making the fluid splash in the air with even higher velocities. Afterwards, the fluid flows back to the middle, again to the borders and so on, while slowly coming to rest. When all movement has mostly stopped, instabilities occur. Figure 1 shows frames from the first 1.25 seconds.

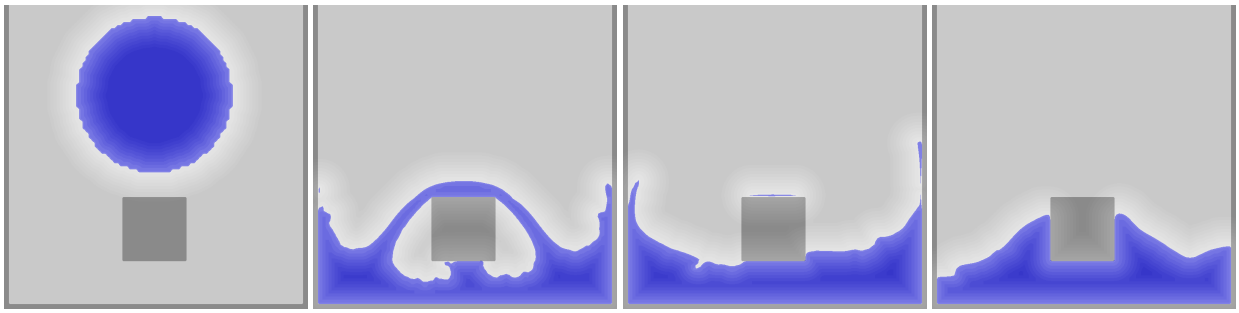


(a) Frame 0 (b) Frame 50 (c) Frame 100 (d) Frame 150

Figure 1: Falling blob scenario with 100x100 nodes at 120 FPS until 10 s.

3.1.2 Single Blocker

In this scenario, a single block falls onto a square blocker splitting into two streams. These streams flow to the bottom left and right corner and merge again in the middle. Again, instabilities take over when the fluid almost comes to a rest. Figure 2 shows frames from the first 1.25 seconds.

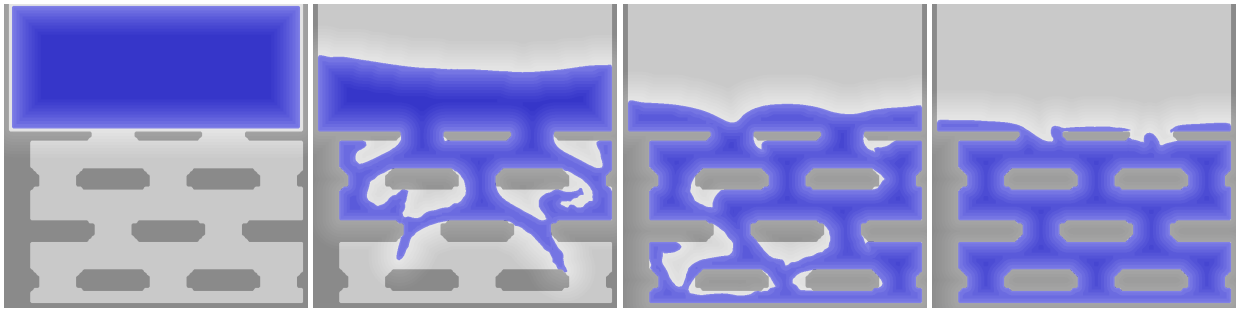


(a) Frame 0 (b) Frame 50 (c) Frame 100 (d) Frame 150

Figure 2: Single blocker scenario with 100x100 nodes at 120 FPS until 10 s.

3.1.3 Many Blockers

In this scenario, the upper part is initially filled with fluid. The lower part contains air and interleaving solid slabs in multiple rows. The fluid has to flow around many corners at high speeds. This results in a very heterogeneous velocity field. Figure 3 shows frames from the first 1.25 seconds.

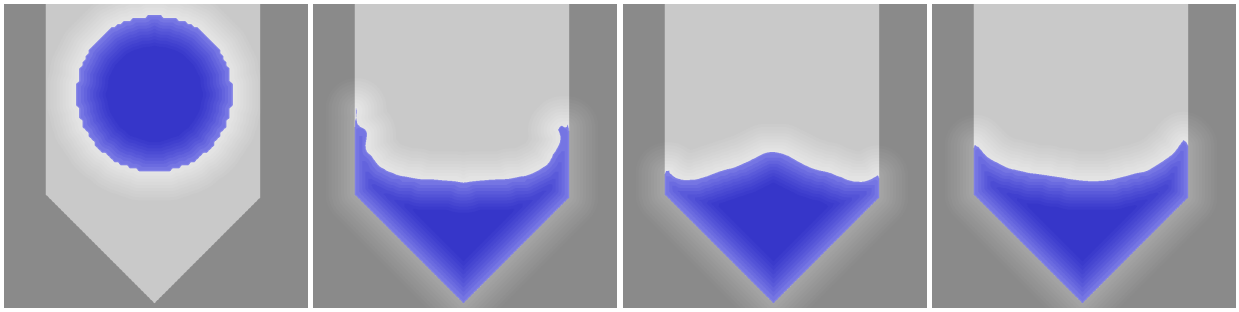


(a) Frame 0 (b) Frame 50 (c) Frame 100 (d) Frame 150

Figure 3: Many blockers scenario with 100x100 nodes at 120FPS until 10s.

3.1.4 Funnel

This scenario is like the falling drop scenario, but with a triangular basin instead of a rectangular one. It shows how accurately diagonal fluid-solid interfaces can be simulated. The result is similar to the falling drop: a fluid blob falls down, gushes around and comes mostly to a rest. Figure 4 shows frames from the first 1.25 seconds.

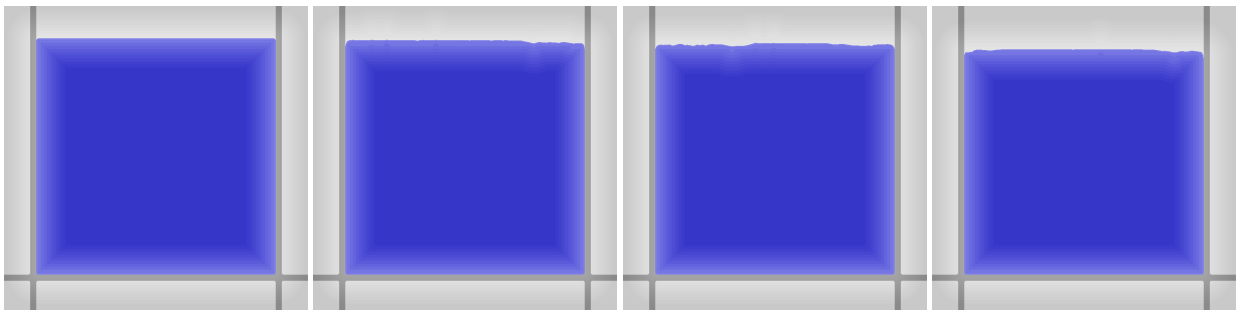


(a) Frame 0 (b) Frame 50 (c) Frame 100 (d) Frame 150

Figure 4: Funnel scenario with 100x100 nodes at 120 FPS until 10 s.

3.1.5 Resting Blob

This scenario features a rectangular blob in a rectangular basin. The blob has no initial movement but gravity applies to it at any time. Shortly after the start, the blob starts to become irregular at the fluid-air interface. This instability seems to have a limit. The fluid volume significantly decreases over time. Figure 5 shows frames from the first 7.5 seconds.



(a) Frame 0 (b) Frame 300 (c) Frame 600 (d) Frame 900

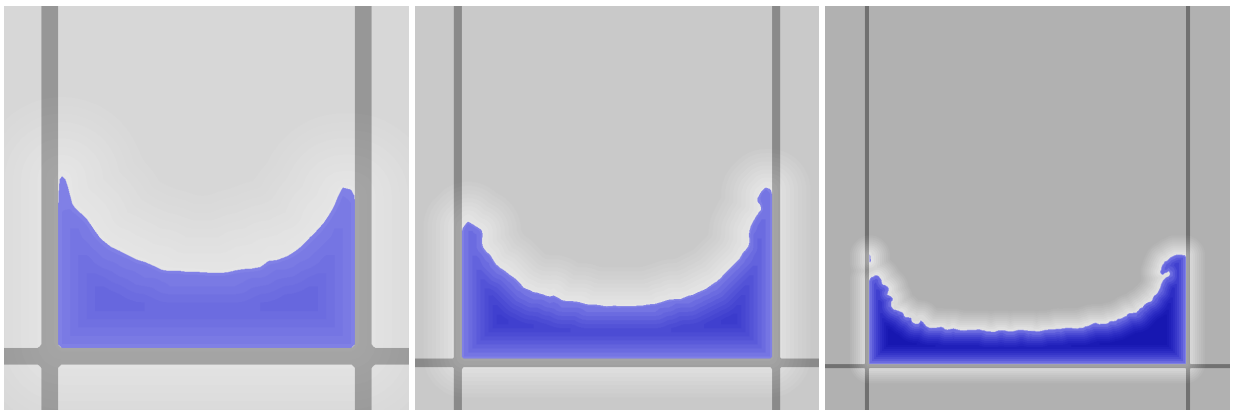
Figure 5: Resting blob scenario with 100x100 nodes at 120 FPS until 10 s.

3.2 Influence of Resolution

A higher resolution of the grid enables more and finer details but in most cases also leads to more fluid loss.

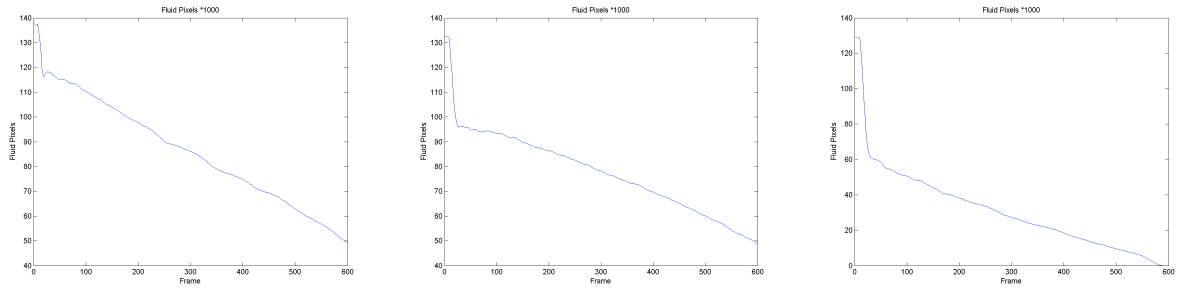
3.2.1 Falling Blob

Figure 6 shows the situation in frame 35 with different resolutions. The fluid pixel data in Figure 7 shows that the impact causes a major loss of fluid. This amount is greater for higher resolutions. After the blob hits the ground, the fluid disappears through the usual instability.



(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 6: Falling blob scenario frame 35 at 60 FPS until 10 s.

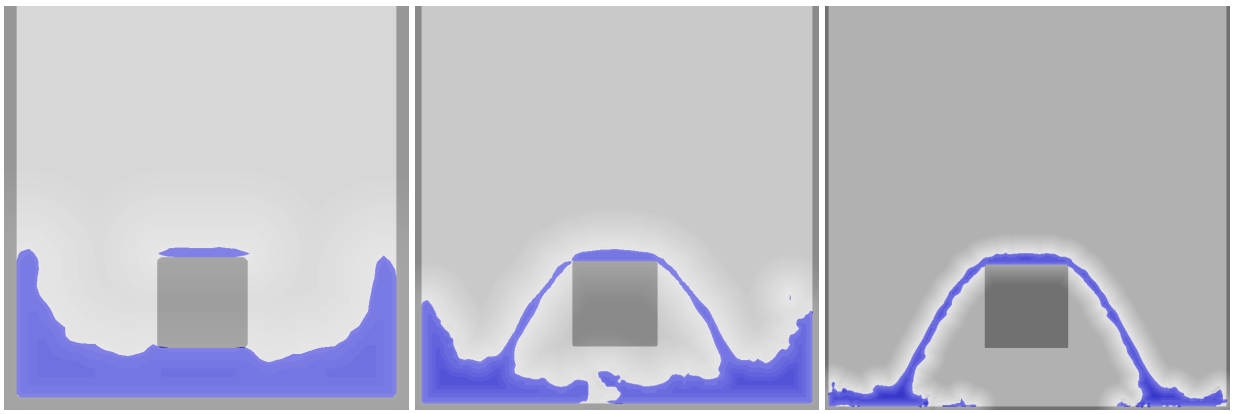


(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 7: Fluid pixels of falling blob scenario at 60 FPS until 10 s.

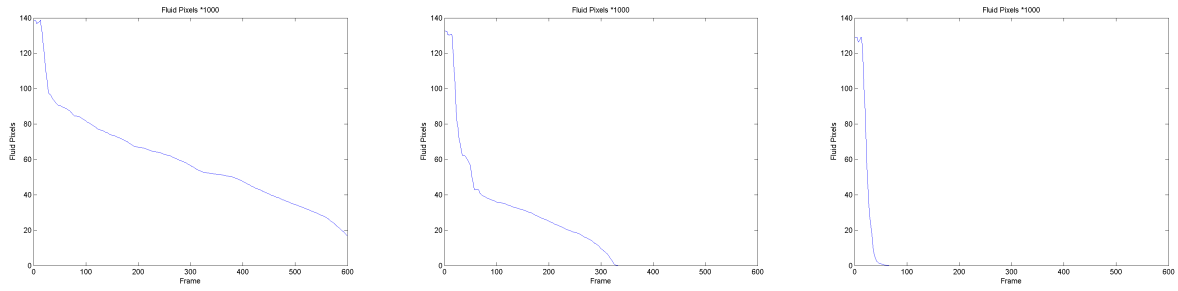
3.2.2 Single Blocker

Figure 8 shows the situation in frame 30 with different resolutions. The fluid pixel data in Figure 9 shows that the loss here is much greater than in the falling blob scenario. The impact phase is about the same length but causes more losses.



(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 8: Single blocker scenario frame 30 at 60 FPS until 10 s.

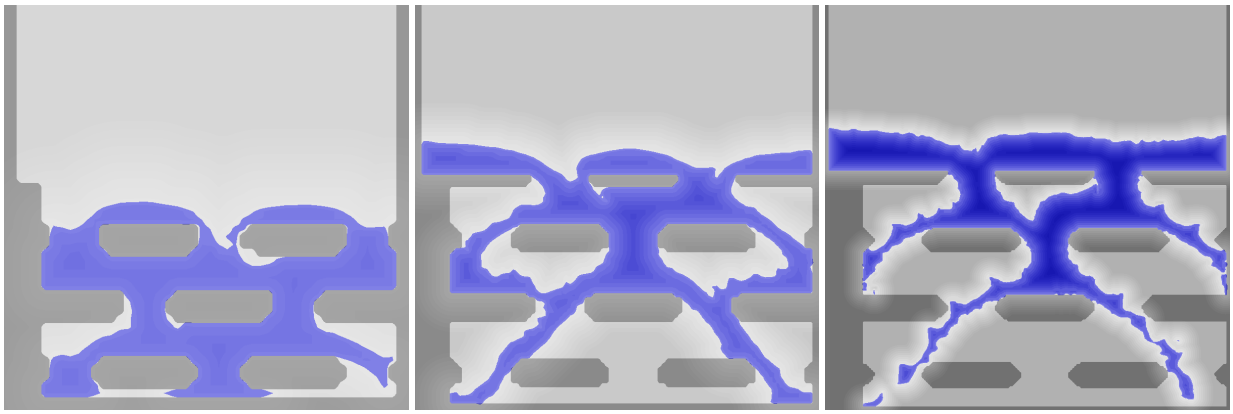


(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 9: Fluid pixels of single blocker scenario at 60 FPS until 10 s.

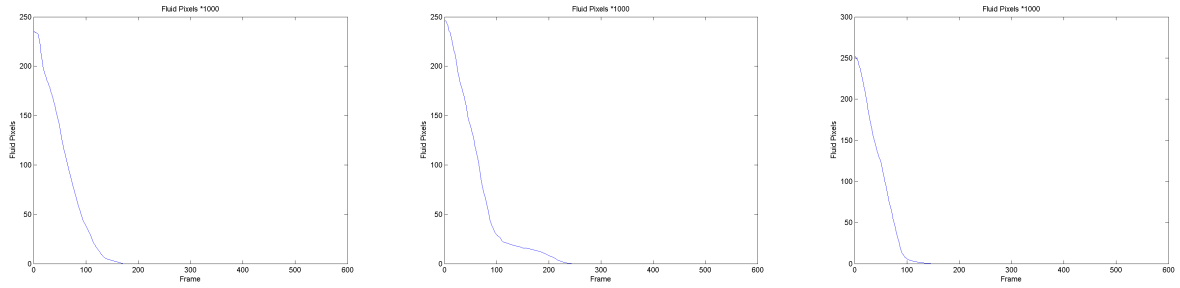
3.2.3 Many Blockers

Figure 10 shows the situation in frame 50 with different resolutions. The fluid pixel data in Figure 11 shows that more fluid is lost in this scenario than in most others.



(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 10: Many blockers scenario frame 50 at 60 FPS until 10 s.



(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

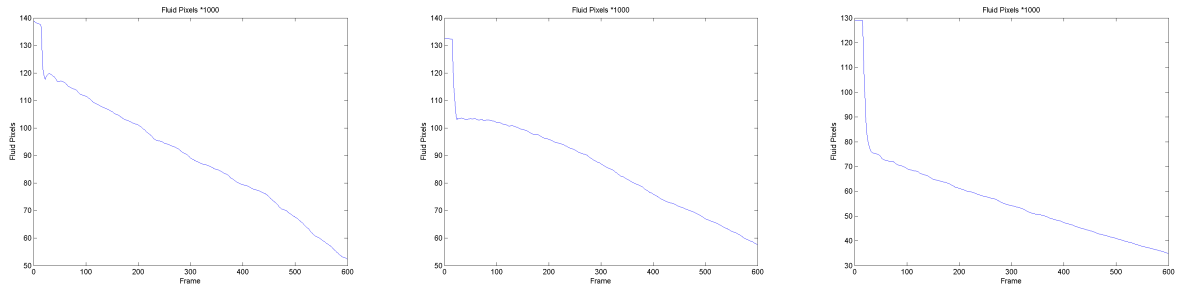
Figure 11: Fluid pixels of many blockers scenario at 60 FPS until 10 s.

3.2.4 Funnel

Figure 12 shows the situation in frame 30 with different resolutions. The fluid pixel data in Figure 13 shows that this example loses much more fluid through instability, despite being similar to the falling blob. The fluid pixel graph for 200x200 nodes shows exponential loss when the fluid is almost resting. We can conclude that the loss of fluid through instability is mostly linear.

(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 12: Funnel scenario frame 30 at 60 FPS until 10 s.

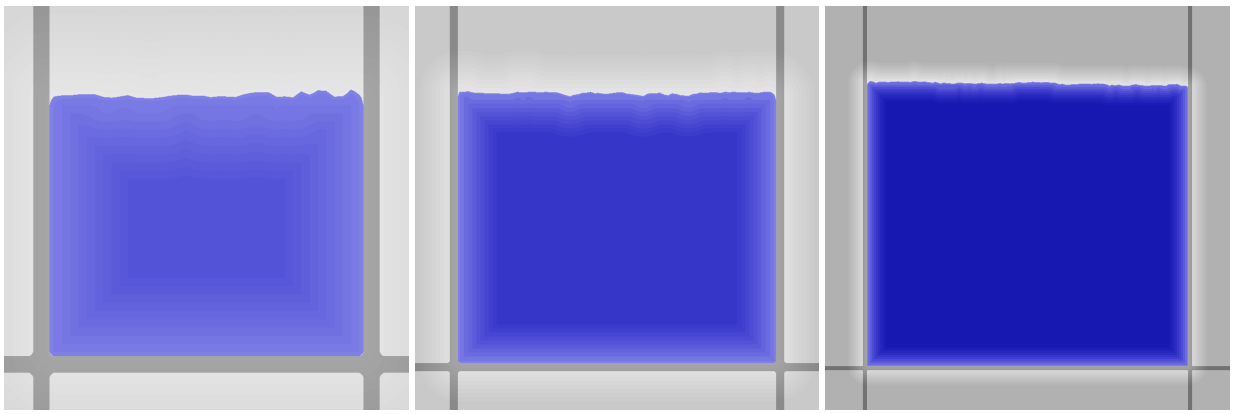


(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 13: Fluid pixels of funnel scenario at 60 FPS until 10 s.

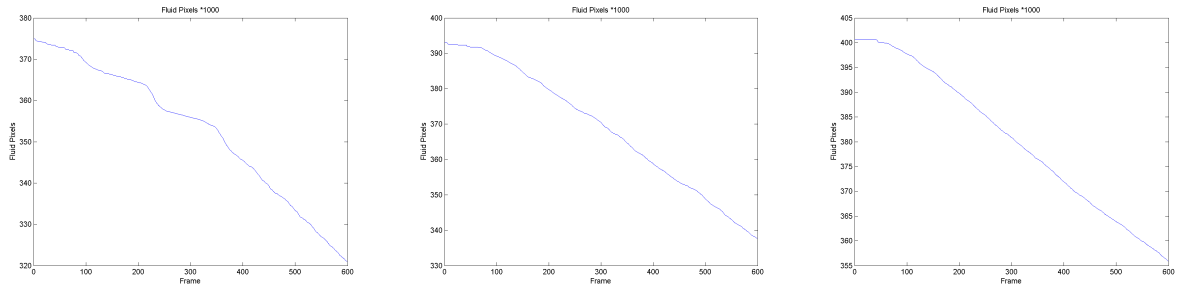
3.2.5 Resting Blob

Figure 12 shows the situation in the last frame 599 with different resolutions. The fluid pixel data in Figure 13 shows that the loss of fluid is approximately linear over time. This is the only scenario in which a higher resolution leads to less loss of fluid. We can conclude that instability appears independently from the resolution but its effect is weaker. Therefore, the main reason for higher losses through higher resolutions are that movements cover more cells in the same time.



(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

Figure 14: Falling blob scenario frame 599 at 60 FPS until 10 s.



(a) 50x50 nodes (b) 100x100 nodes (c) 200x200 nodes

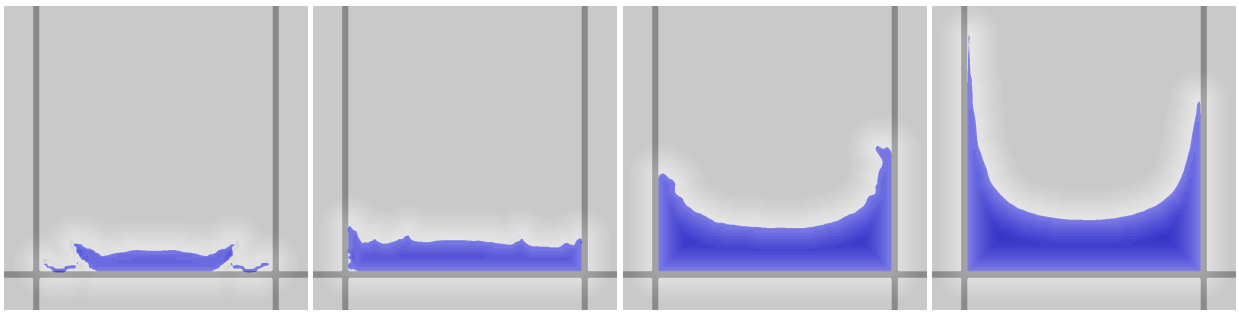
Figure 15: Fluid pixels of resting blob scenario at 60 FPS until 10 s.

3.3 Influence of Time Step

The graphs make it obvious that shorter time steps result in less fluid loss. Consequently, more details are preserved.

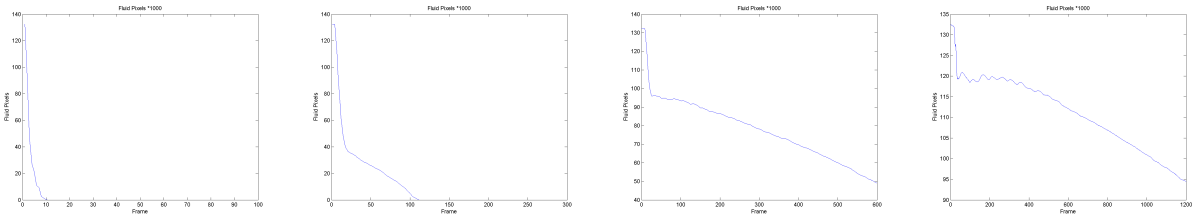
3.3.1 Falling Blob

Figure 16 shows the state of the simulation after 0.5 seconds. Obviously, a higher frame rate leads to more fluid conservation and an overall higher quality. Figure 17 shows the change in the amount of fluid pixels over time. At 10 FPS, the entire fluid is lost within 1 second. At 30 FPS, it is after 2 seconds. At 60 FPS, about 37% of the fluid remain after 10 seconds. 120 FPS provides the best result with about 70% remaining at the end.



(a) Frame 5 at 10FPS (b) Frame 15 at 30FPS (c) Frame 30 at 60FPS (d) Frame 60 at 120FPS

Figure 16: Falling blob scenario with 100x100 nodes until 10 s.

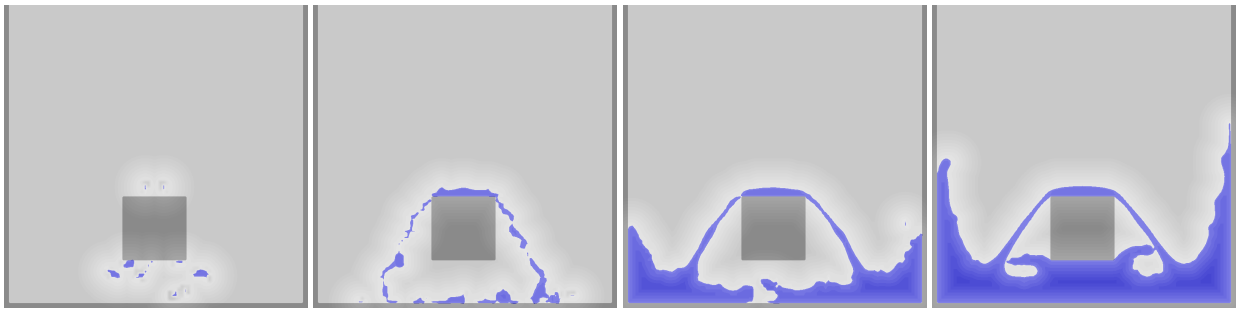


(a) 10 FPS (b) 30 FPS (c) 60 FPS (d) 120 FPS

Figure 17: Fluid pixels of falling blob scenario with 100x100 nodes until 10 s.

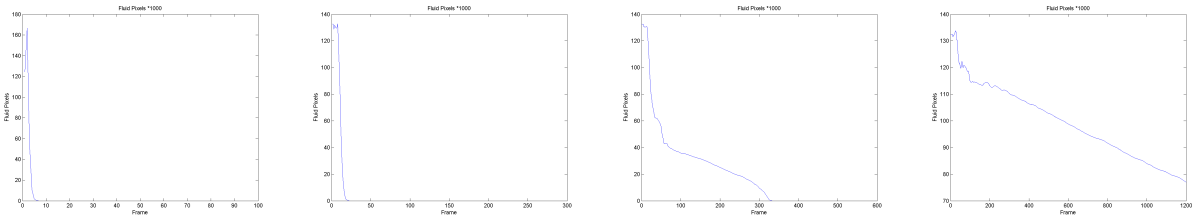
3.3.2 Single Blocker

Figure 18 shows the state of the simulation after 0.5 seconds. The effect of the frame rate on this scenario is similar to the falling blob scenario. Figure 19 shows that all fluid is lost within about 0.8 seconds at 10, 1.2 seconds at 30 FPS, and 5.5 seconds at 60 FPS. About 57% of the initial fluid remains at 120 FPS after 10 seconds.



(a) Frame 5 at 10 FPS (b) Frame 15 at 30 FPS (c) Frame 30 at 60 FPS (d) Frame 60 at 120 FPS

Figure 18: Single blocker scenario with 100x100 nodes until 10 s.

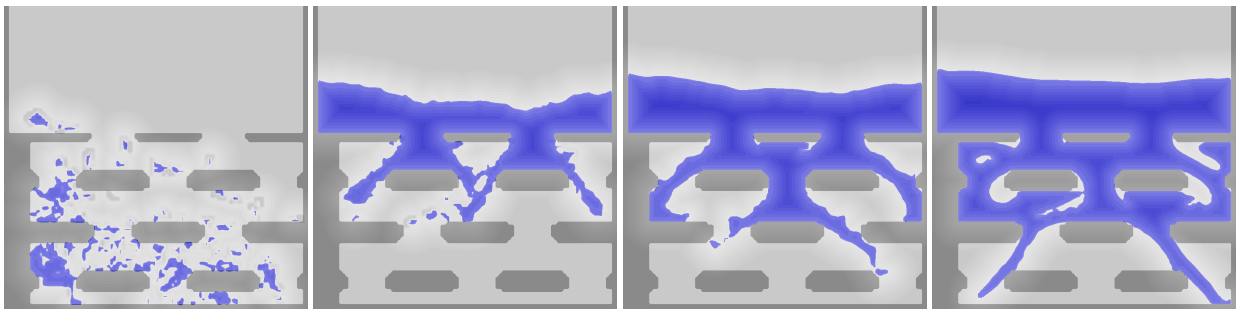


(a) 10 FPS (b) 30 FPS (c) 60 FPS (d) 120 FPS

Figure 19: Fluid pixels of single blocker scenario with 100x100 nodes until 10 s.

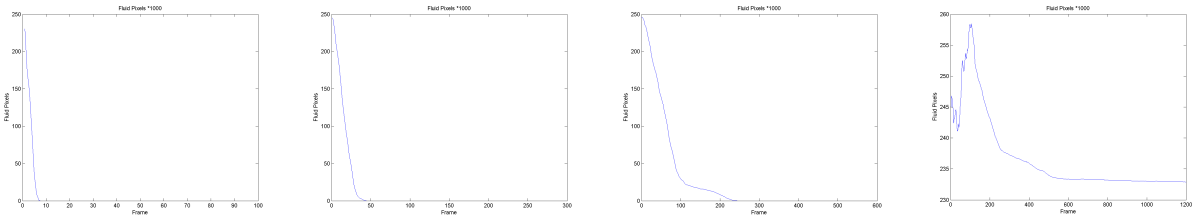
3.3.3 Many Blockers

Figure 20 shows the state of the simulation after 0.5 seconds. Figure 21 shows that the entire fluid is lost within about 0.9 seconds at 10, 1.4 seconds at 30 FPS, and 4 seconds at 60 FPS. About 95% of the initial fluid remains at 120 FPS after 10 seconds.



(a) Frame 5 at 10 FPS (b) Frame 15 at 30 FPS (c) Frame 30 at 60 FPS (d) Frame 60 at 120 FPS

Figure 20: Many blockers scenario with 100x100 nodes until 10 s.

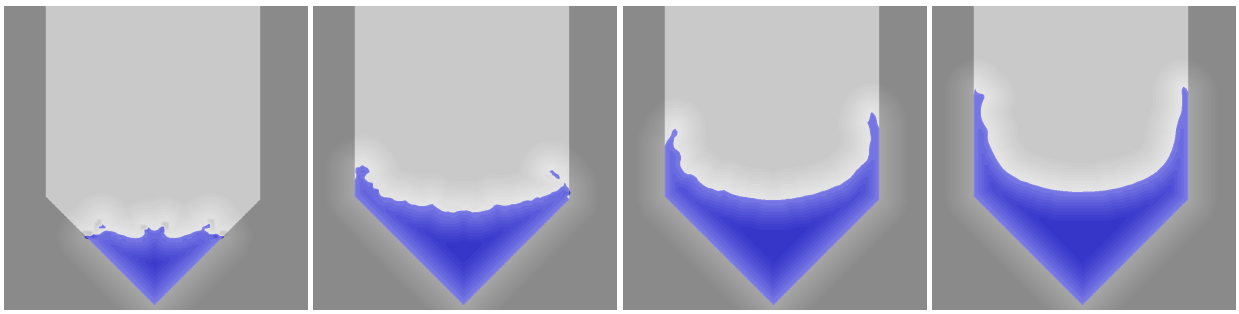


(a) 10 FPS (b) 30 FPS (c) 60 FPS (d) 120 FPS

Figure 21: Fluid pixels of many blockers scenario with 100x100 nodes until 10 s.

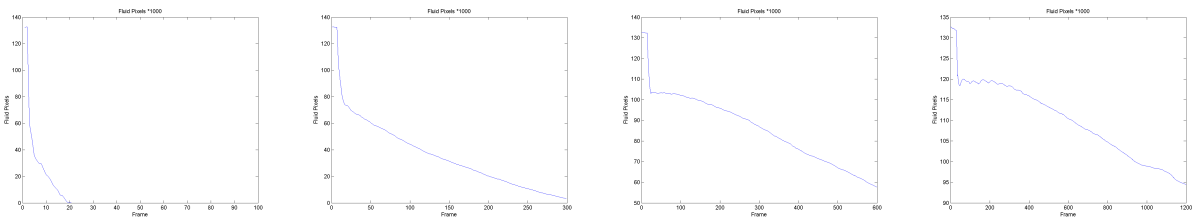
3.3.4 Funnel

Figure 22 shows the state of the simulation after 0.5 seconds. Figure 23 shows that all fluid is lost within about 2 seconds at 10 FPS. A bit of fluid remains after 10 seconds at 30 FPS. About 45% of the initial fluid remains at 60 FPS and 70% at 120 FPS.



(a) Frame 5 at 10 FPS (b) Frame 15 at 30 FPS (c) Frame 30 at 60 FPS (d) Frame 60 at 120 FPS

Figure 22: Funnel scenario with 100x100 nodes until 10 s.

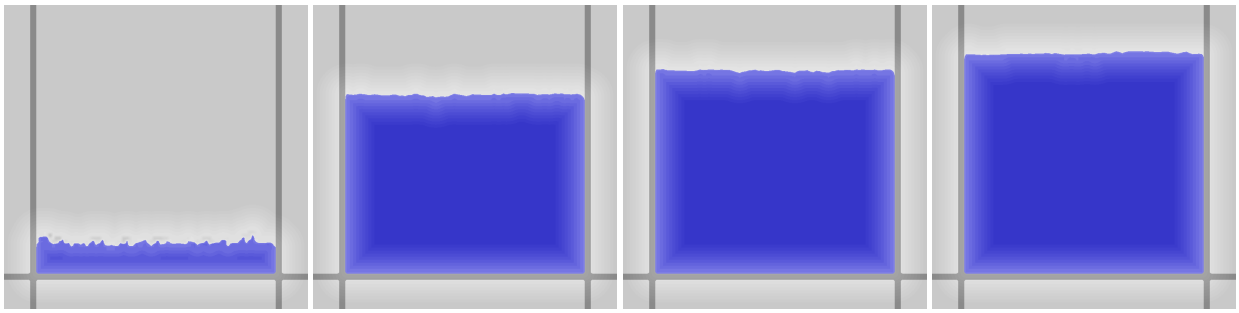


(a) 10 FPS (b) 30 FPS (c) 60 FPS (d) 120 FPS

Figure 23: Fluid pixels of funnel scenario with 100x100 nodes until 10 s.

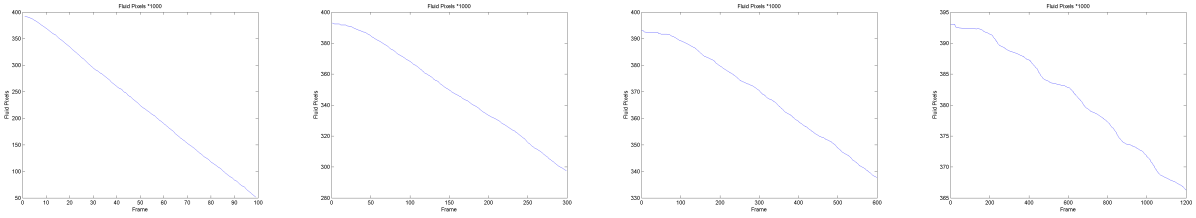
3.3.5 Resting Blob

Figure 24 shows the result in the end, after 10 seconds. Figure 25 shows that at all frame rates, at least some fluid is preserved. The remaining amounts are 12% at 10 FPS, 78% at 30 FPS, 87% at 60 FPS, and 93% at 120 FPS.



(a) Frame 99 at 10 FPS (b) Frame 299 at 30 FPS (c) Frame 599 at 60 FPS (d) Frame 1199 at 120 FPS

Figure 24: Resting blob scenario with 100x100 nodes until 10 s.



(a) 10 FPS (b) 30 FPS (c) 60 FPS (d) 120 FPS

Figure 25: Fluid pixels of resting blob scenario with 100x100 nodes until 10 s.

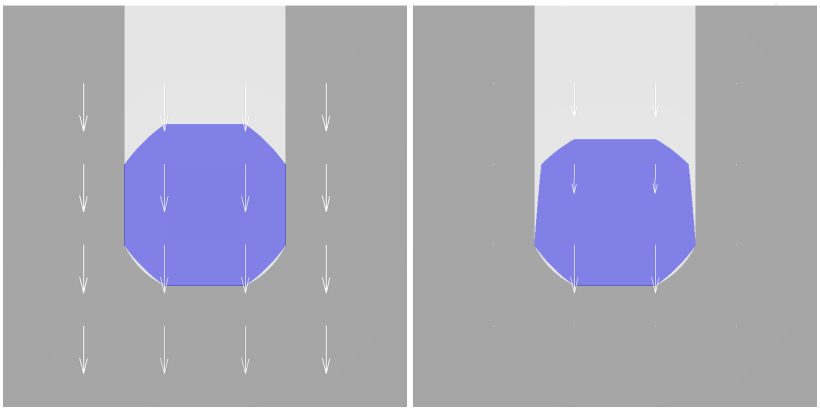
4 Discussion

The results from the experiments concerning resolution and frame rate are, in short, listed here. A higher resolution leads to more details, while also losing more volume - except in the resting blob scenario. A higher frame rate leads to less loss of volume. All simulations suffer from loss of fluid to some extent. The fluid almost comes to a rest, although no dissipation is applied. This is the result of numerical dissipation. Resting fluid does not stay stable, i.e. movement appears out of nowhere.

The loss is usually higher when a blob hits an obstacle, causing turbulence. When the fluid comes to rest, some of it is lost almost constantly.

This loss seems linear and independent on movement. Therefore, let us call this kind of loss 'static'. It is caused by the pressure projection, which does not completely counteract the force of gravity. To be more exact, the calculated pressure of the fluid at horizontal fluid-solid interfaces is lower than the pressure in the solid.

Figure 26 shows the falling cells scenario with a width of 6x6 nodes. This is a minimalistic example consisting of a blob at the bottom of a basin. The blob is made of four nodes and initially moves down with a velocity of one node per frame. Here, the velocity is 12m/s. The blob collides with the solid in the first frame, which should stop the blob immediately.



(a) Frame 0 (b) Frame 1

Figure 26: Initial state and first frame of the falling cell scenario.

Figure 27 shows some calculation steps in the first frame. As you can see, the gradient in x direction is not exactly zero. This might be caused by numerical issues in the pressure calculation and can explain the observed instabilities. Furthermore, the resulting velocity field shows that the fluid is not fully stopped by the collision's pressure. It continues moving down with about 20% of the initial velocity.

```
v0(:,:,1) =

     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0

v0(:,:,2) =

     0     0    12    12     0     0
     0     0    12    12     0     0
     0     0    12    12     0     0
     0     0    12    12     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
```

(a) Initial velocity field

```
     0         0         0         0         0         0
     0   -8.0000         0         0   -8.0000         0
     0  -10.6667  -10.6667  -10.6667  -10.6667         0
     0  -18.0000  -19.0000  -19.0000  -18.0000         0
     0  -21.3333  -23.3333  -23.3333  -21.3333         0
     0         0         0         0         0         0
```

(b) Pressure in first frame 1

```
g(:,:,1) =

  1.0e-14 *

     0     0         0         0     0     0
     0     0         0         0     0     0
     0     0   -0.1776   -0.1776     0     0
     0     0   -0.3553   -0.3553     0     0
     0     0         0         0     0     0
     0     0         0         0     0     0

g(:,:,2) =

     0     0         0         0     0     0
     0     0    5.3333    5.3333     0     0
     0     0    9.5000    9.5000     0     0
     0     0    8.3333    8.3333     0     0
     0     0         0         0     0     0
     0     0         0         0     0     0
```

(c) Pressure gradient in first frame

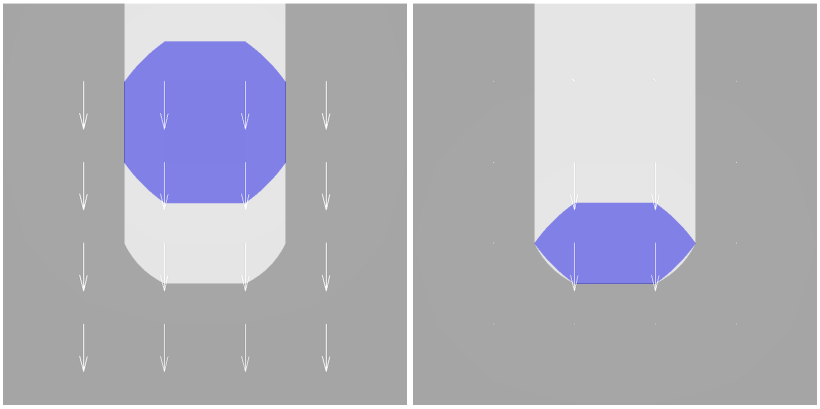
```
v1(:,:,1) =

  1.0e-14 *

     0     0    0.1776    0.1776     0     0
     0     0    0.1776    0.1776     0     0
     0     0    0.1776    0.1776     0     0
     0     0    0.3553    0.3553     0     0
     0     0         0         0     0     0
     0     0         0         0     0     0

v1(:,:,2) =

     0     0    2.5000    2.5000     0     0
     0     0    2.5000    2.5000     0     0
     0     0    2.5000    2.5000     0     0
     0     0    3.6667    3.6667     0     0
     0     0         0         0     0     0
     0     0         0         0     0     0
```

(d) Velocity after pressure projection and with set air velocities

Figure 27: Calculation steps in the first frame of the falling cell scenario. The gradient g of pressure p is subtracted from the velocity field v0 resulting in v1 after setting the air nodes' velocities.

The resting blob scenario is the only one that is dominated by 'static' loss. There, we can see that 'static' loss is decreased by increased resolution. In the other scenarios, a higher resolution leads to greater loss. Therefore, there must be a different kind of loss, 'dynamic' loss, which is mostly caused through movement. When we investigate the losses e.g. in the falling blob scenario, we find that the fluid depletes much faster when the blob hits the ground for the first time. In other words, the higher the velocities, the higher the loss. In this phase, we find parts of the fluid traveling with two nodes per frame or even more. This suggests that we face a situation similar to the bullet-trough-paper problem from the domain of rigid body physics. Instead of a body moving through a very thin body in one time step, we have fluid moving from different spaces to a single space or into solids without a chance to calculate pressure.

We can modify the falling cell scenario to illustrate this problem. The blob is created above the bottom of the basin with a distance of one node. The initial velocity is doubled so that the blob moves down at two nodes per frame. Figure 28 shows the result: the lower two nodes of the blob disappear into the solid. The upper two nodes are now colliding with the solid and the resulting pressure slows them down. A similar problem would occur if the bottom solid were fluid. This source of loss is resolved through small enough time steps.



(a) Frame 0 (b) Frame 1

Figure 28: Initial state and first frame of the modified falling cell scenario.

5 Outlook

To summarize the issues of the current implementation: The two main sources of fluid loss are 'static' and 'dynamic'. A higher resolution reduces 'static' but increases 'dynamic' losses. A smaller time step reduces 'dynamic' losses. 'Static' loss is probably caused by numerical issues leading to instability and inaccuracy in the pressure calculation. 'Dynamic' losses are caused by high velocities or big time steps. The question is how can we solve these problems.

The instability could be decreased by reducing numerical issues, for example by re-shaping formulas or using even more accurate data types than the IEEE double. Of course, there can always be a calculation or implementation error the author failed to notice.

The inaccuracy of the pressure projection might be a basic problem of the finite element method. It might be possible to reduce it by making the boundaries more accurate. Specifically, the nodes at fluid-air interfaces could be set exactly on the interface as defined in the distance field. One could also use more sampling points on the interface. Unfortunately, this approach might mess with subtracting the pressure gradient on the grid. Another issue that may cause inaccuracies is that solid nodes, which are neighbors only in a 8-connected grid but not in a 4-connected one, are treated like 4-connected neighbors. Another, probably more difficult idea is to add some kind of volume preservation part the the Navier-Stokes equations.

The 'dynamic' losses can be reduced when either the time step or the velocities are taken as variables. Changing the velocities seems like an unclean solution but may be easy and effective in practice. For example, one could prevent fluid from moving into undesired locations by setting the movement target in front of these locations. Simply put, shorten the velocity vector if it is too long. Another solution for this problem could be to allow such movement but calculate a pressure offset out of the predicted collision volume. This could be done by calculating the divergence of the colliding fluid parts. Fluid would be re-created at the place where it disappeared. Unfortunately, the disappearing and re-appearing of fluid will probably cause more distortion and instability.

On the other hand, changing the time step might be problematic because it can increase the computation time immensely. This often is a deal-breaker, especially for real-time applications. If you decide to take this approach, you can shorten time steps whenever there are nodes with velocities higher than one node per step. However, to get rid of all 'dynamic' losses, one would need a step for every bit of fluid arriving at a solid. Papers like 'Stable Fluids' by Jos Stam can give more ideas [2]. The proposed approach uses the Fast Fourier Transformation to accelerate the simulation.

6 References

[1] Selle, Andrew, et al. "An unconditionally stable MacCormack method." Journal of Scientific Computing 35.2-3 (2008): 350-371, http://physbam.stanford.edu/~fedkiw/papers/stanford2006-09.pdf.

[2] Stam, Jos. "Stable fluids." Proceedings of the 26th annual conference on Computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co., 1999.