A Self-Healing Farm for Distributed FX Simulation on Studio Workstations

Krishna S
Perfekta FX, London, United Kingdom, and India

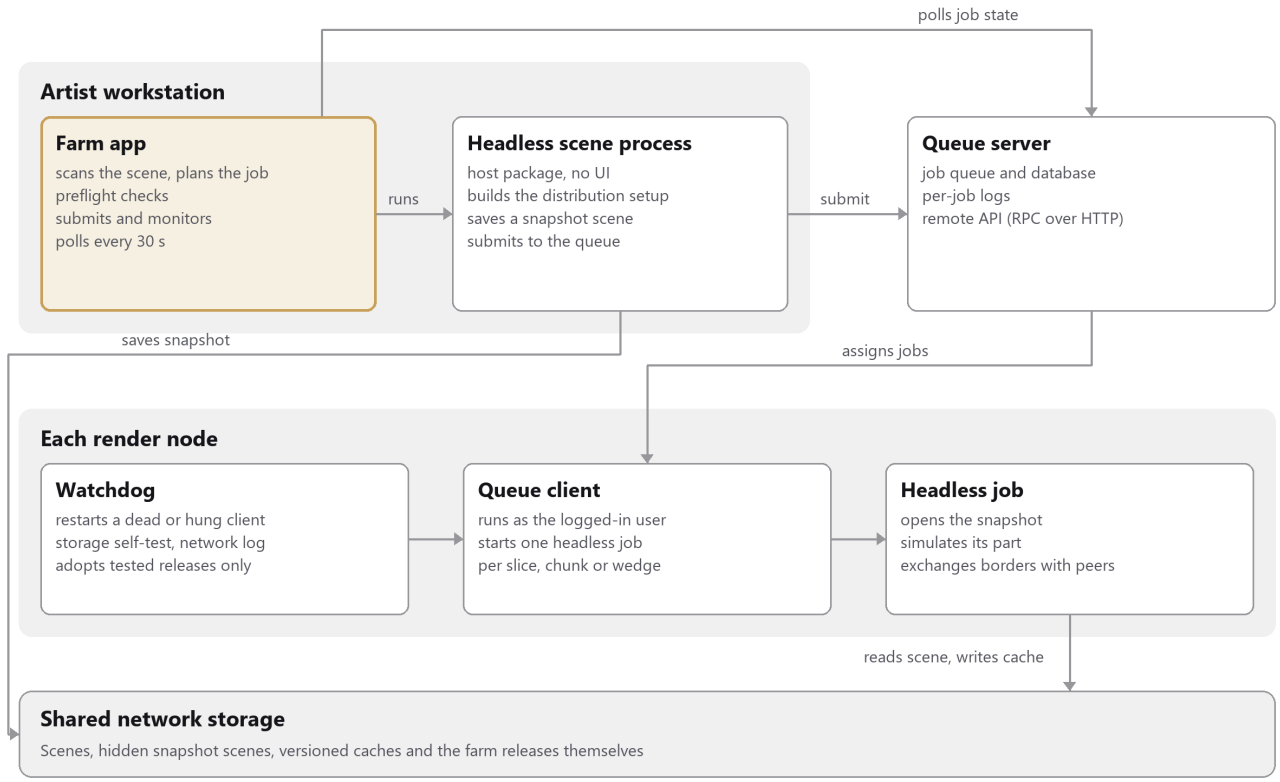

Fig. 1. System overview. A desktop app on the artist's workstation drives the host FX package's own headless interpreter to build the distribution setup and submit it to a queue manager. Every render node runs a watchdog that keeps its queue client alive, and all nodes read scenes and write caches through one shared network store.

Abstract. Small and mid-sized visual effects studios often own a handful of powerful workstations but no dedicated simulation farm. Distributing a simulation across them by hand is slow and error-prone: partition borders, tracker connections, cache paths and per-machine state must all be right for every submission. We describe a desktop tool that turns an FX scene into distributed farm jobs and watches them to completion. It adds no plug-ins and installs nothing on render nodes; it drives the host package's own distribution primitives through its headless scripting interpreter, the same way every time, so the design carries over to any FX package that exposes such an interpreter, a queue manager with a remote API and solver-level domain decomposition. The system offers five distribution methods (particle slices, volume slices, emission regions, frame splits and wedges), places particle-slice borders at particle-count quantiles, and records per-slice load and surface indices every frame. A watchdog on each node restarts failed queue clients, verifies shared-storage visibility and adopts new code only after it passes a self-test, while immutable versioned releases with a last-good fallback ensure that a faulty update cannot stop the farm. On a four-node cluster linked by 1 Gb Ethernet, frame-split caches and wedges scale almost one-for-one, whereas spatially sliced liquid simulations gain only 9 to 13% when moving to four nodes. Per-frame balance logs show that synchronisation, not load imbalance or particle traffic, dominates the cost. We therefore run simulations on a single node by default and reserve slicing for scenes that exceed one node's memory.

CCS Concepts: Computing methodologies, Physical simulation; Computer systems organization, Distributed architectures; Software and its engineering, Software reliability.

Additional Key Words and Phrases: distributed simulation, render farm, domain decomposition, load balancing, fault tolerance, FX pipeline

Reference Format: Krishna S. 2026. A Self-Healing Farm for Distributed FX Simulation on Studio Workstations. Perfekta FX Technical Report, November 2026.

1 INTRODUCTION

Liquid, smoke, fire and particle simulations are among the most expensive steps in an FX pipeline. Large facilities run them on dedicated clusters with in-house schedulers. A smaller studio usually has something different: a few high-end artist workstations that sit idle overnight, a shared network store and the queue manager that ships with its FX package. The pieces for distributed simulation exist, but assembling them for one shot means inserting partition nodes, linking them to a tracker, setting slice numbers, versioning the cache and submitting, then noticing when a machine drops off. Each manual step is a chance for a silent error, and many of these errors only appear hours later.

This report describes how we turned that manual procedure into a tool. Our goals were:

- **Deterministic.** Build exactly the setup a careful artist would build by hand, identically every time, and be able to re-run it on the same scene.
- **Non-invasive.** No plug-ins and no extra software on any node; use only what the FX package and the operating system already provide. The artist's workfile is never modified.
- **Self-healing.** Nodes that fail should recover without anyone walking over to them.
- **Safe to change.** A broken edit to the farm's own code must never stop the farm.
- **Measured.** Every decision about how to split work should rest on logged numbers, not intuition.

Our contributions are a practical architecture that meets these goals (Section 2), five distribution methods chosen automatically by the kind of node being cached (Section 4), quantile-based placement of particle-slice borders with per-frame load and surface indices (Section 4.2), a monitoring and recovery layer that diagnoses failures from job logs (Section 6), per-node watchdogs with storage self-tests and clock-proof heartbeats (Section 7), a release mechanism with out-of-process checks and last-good fallback (Section 8), and measurements that explain why spatial slicing pays off little on commodity networks (Section 9).

2 SYSTEM OVERVIEW

Figure 1 shows the three parts that do the work. The *farm app* is a small Python desktop application on the artist's workstation. It scans the scene, shows the plan and tracks jobs. All interface updates pass through a queue drained every 200 ms by the UI thread, so the window never freezes while jobs are polled in the background. The *headless scene process* is the host package's own command-line interpreter, started by the app with no window. It opens the scene, adds the distribution setup, saves a snapshot and submits it. The app passes it a request as a JSON file, streams its log lines into an activity view and reads a JSON result when it exits, so a crash or host error comes back as a readable message with its traceback.

The *queue manager* (one server, one client per node) hands work to the render nodes, which open the snapshot from shared storage and write their part of the cache back to it. The app talks to the queue server through its remote API over HTTP to list clients, read jobs and logs, and submit, cancel, delete or reschedule jobs.

The design assumes only three capabilities of the host package: a headless scripting interpreter that can open, edit, save and submit scenes; a queue manager with a remote API; and solver support for domain decomposition with border exchange between processes. Any FX package that provides these can be driven the same way. A workstation can be both an artist's machine and a render node, and the queue server can also take render work.

3 LIFE OF A JOB

A job goes from a click to a finished cache in nine steps. The artist performs the first two; the app and queue manager do the rest.

1. **Scan.** The headless process walks every node in the scene and lists what can be distributed: liquid (FLIP) solvers, particle solvers, smoke and fire solvers, whitewater solvers and cache nodes. Sparse-grid smoke solvers that the host cannot partition are marked as single-node. For each cache it checks whether the output is written per frame and whether a simulation feeds it directly; this decides between a frame split and single-node or wedge execution. Each item receives a unique cache name.
2. **Choose.** The artist picks a row and checks the frame range. For simulations the app ticks one machine: the best idle node that is not the queue server and has passed its storage self-test. Ticking further machines slices the simulation, and more than two slices asks for confirmation. Several rows can be selected in order to form a chain.
3. **Preflight.** Before anything is written, the submit is blocked if the scene is not on shared storage, no machine is selected, or a selected machine is offline, unknown to the queue or has failed its self-test. Free licences are checked. Warnings are raised when slices would run on the queue server, when a busy machine would make the job wait, and when a smoke simulation uses a different slice count from its previous run.
4. **Set up.** The headless process builds the nodes for that kind of job (Section 4). For sliced particle simulations it first samples particle positions and places the borders. It picks the cache version and refuses to overwrite an existing version without explicit confirmation.
5. **Snapshot.** The prepared scene is saved as a hidden file next to the workfile. The workfile itself is never saved.
6. **Submit.** The headless process submits the job to the queue, and the app captures the returned job ids.
7. **Run.** The queue assigns child jobs to the chosen nodes. Each node opens the snapshot and writes its part of the cache. For sliced simulations a tracker job starts first and the slices connect to it.

8. **Monitor.** Every 30 s the app reads the job tree, counts finished frames on disk and reads the slices' statistics files, then updates progress, time per frame, ETA and balance (Section 6).
9. **Notify.** A desktop notification reports submission, completion, failure or cancellation, naming the cache version. A sliced job is flagged when its busiest slice is typically more than 10% above the average.

4 DISTRIBUTION METHODS

The app uses five methods, chosen by the kind of node (Table 1). Each builds the same nodes the host's own interactive tools would build, but without a UI, so it can be re-run on the same scene. Simulations run on one node by default and are sliced only when the artist selects more machines; a single-node simulation writes the same versioned files, as slice 0.

Table 1. Distribution methods and validation against a single-node run.

Method	Used for	Split	Peers talk	Tested result
Particle slices	Liquids, particles	N spatial slabs	Every substep	Particles identical to 1 node (20,000, 0 duplicates); liquid 2 slices within 2.2%
Volume slices	Smoke, fire	N slabs of the container	Every substep	2 slices within 0.75%; 3 slices drift up to ~6% by frame 48
Emission regions	Whitewater	N birth regions, full range	No	Within 1.8%, 0 duplicate ids
Frame split	Per-frame caches	Frame chunks	No	60 frames as 4 jobs of 15
Wedges	Cloth, rigid bodies, any cache	One cache per value	No	3 results in parallel

4.1 Particle slices

The simulation domain is cut into N slabs along one horizontal axis, never along gravity. Both horizontal axes are scored from a sample of particles, and the axis with better balance and fewer border particles is used; the log records the choice. Each node simulates the particles inside its slab. Particles within the exchange distance of a border are copied to the neighbour every substep, and the liquid pressure solve runs across all slabs together [Brackbill and Ruppel 1986; Zhu and Bridson 2005; Bridson 2015]. Before writing, each slice drops the ghost particles it does not own, and the loader that merges slices removes the NaN and infinite values that stitching can leave, which otherwise break downstream meshing and whitewater emission.

Border placement. We measured the host slicer's rule empirically: a segment of length l centred on t is divided into N equal regions and the two outer regions extend to infinity. We sample particle positions, from an earlier version of the same cache when one exists and otherwise from the sources, and place borders at particle-count quantiles. Arbitrary borders are realisable for two or three slices; for four or more, the equal-spacing constraint applies and we search for the best offset and spacing. Borders stay fixed during the simulation, because the host's dynamic rebalancing is incompatible with narrow-band field exchange. Recent work on optimal-transport partitioning addresses dynamic domains directly [Wretborn et al. 2026].

Reproducible links. Parameter links between the generated controls are written literally, without following existing references. Otherwise re-running the setup on an already distributed scene made the controls node reference itself, and slices waited indefinitely. The links are checked again before submission.

4.2 Load and surface indices

While a sliced simulation runs, every slice writes a small statistics file per frame containing the particles it owns, the particles within the exchange distance of a border, and the seconds the frame took. From these the app derives two numbers for slice i, where n_i is owned particles, $\bar{n}$ their mean over slices and b_i border particles:

$$L_i = n_i / \bar{n} - 1, \quad S_i = b_i / n_i$$

The load index L measures excess work: at +0.25 a slice has 25% more particles than average and every other node waits for it at each substep. The surface index S is the share of a slice that must be copied to a neighbour every step; it grows with the number of slices. The job view shows the busiest slice against the average, and a balance window plots both indices per frame alongside each node's network traffic. Before a run, the scan predicts S for two, three and four slices; for our test scene it predicted 4%, 9% and 13%.

4.3 Volume slices

For smoke and fire the container is cut into N slabs along its longer horizontal side. The overlap, in voxels, is an expression of fluid speed, voxel size, frame rate and substeps, and density, velocity, temperature, heat and fuel are exchanged at the borders. Each node writes its slab and the loader splices them. Two slices matched a single-node run within 0.75%, but three slices drifted up to about 6% by frame 48. Because the split changes the result, the app warns when a simulation is submitted with a different slice count from its last run, so look development and final renders use the same count.

4.4 Emission regions

Whitewater particles do not affect one another, so the spray can be split by where it is born rather than where it travels. The emission field is divided into N equal regions along the source's longer horizontal axis, and each part simulates its region over the full frame range on one node, with no communication. Two details proved essential: any NaN velocity voxels, which stitched liquid slices can produce, are set to zero before emission, as they otherwise turned most particles into NaN; and particle ids are made unique as $id \times N + part$, so the merged result supports motion blur and retiming. Parts matched a single-node run within 1.8% with no duplicate ids.

4.5 Frame splits, wedges and chains

Frame split. For per-frame caches the frame range is cut into chunks, by default the range divided by the number of machines. The queue hands chunks to free nodes, so a faster node simply takes more. The original cache node is left unchanged: a separate output driver references its path.

Wedges. A wedge index steps from 0 to N−1 and the chosen parameter reads the *k*-th value from a list. The output name carries the wedge number and a manifest records the value each wedge used. Cloth, soft-body and rigid-body simulations run on one node or as wedges.

Chains. Several steps, for example liquid sim, mesh, whitewater source and whitewater, can be submitted as one job. The snapshot contains a dependency graph with one fetch per step and a wait between steps, so each step starts when the previous one finishes. Simulations run on one machine over the whole range, frame-splittable steps are split, and earlier cache steps switch to reading from disk. In our test the second step wrote its first file 20 s after the first step wrote its last.

5 VERSIONS AND SNAPSHOTS

Every submission writes a new cache version and a hidden snapshot that records exactly what was simulated. Caches follow the pattern `geo/<scene>.<cache>/vN/<scene>.<cache>.vN.<slice>.<frame>`; wedges add `w<k>`. A blank version means the next free one, and a typed version that already exists stops the submit with an overwrite question. The cache name is the simulated object's name when unique; when names clash, the parent's name is added, so two solvers never share files.

The snapshot's file name necessarily differs from the workfile's, which would change the scene-name variable inside every path. Before saving, the app therefore replaces that variable with the workfile's name in every string parameter and expression, so inputs and outputs resolve exactly as in the workfile. As an alternative, artists can save a new numbered scene version and cache under that name.

6 MONITORING, DIAGNOSIS AND RECOVERY

Every 30 s the app checks each tracked job. It reads the job and its children from the queue, ignoring helper jobs. For simulations it counts frames on disk: a frame is done when every slice's file for it exists, and progress, seconds per frame and ETA follow from that count. A child marked abandoned, because its client stopped reporting, receives 300 s of grace, since slow clients often report again and the queue re-activates the job. A failed child's log is read through the queue's API and matched against known errors (Table 2), so diagnosis works from any node. Automatic retries are capped at twice the number of child jobs.

Table 2. Diagnosed failures and the automatic response.

Symptom in log	Diagnosis	Action
Shared path not found	Node's client cannot see the share (elevated session)	Names the node and the fix
Folder already exists	Two slices created one folder at once	Reschedule
Tracker connection refused	A slice lost the tracker	Reschedule
Missing output driver	Job points at a missing node	Rescan and resubmit
Licence error	No free licence	Explain
Out of memory	A node ran out of RAM	Suggest larger particle separation or more nodes
Abandoned past grace	Usually a firewall blocking slice traffic	Point to the firewall rule
Anything else	Last error line in the log	Show under the job

Stopping a sliced simulation. Slices wait for one another, so a single dead slice leaves the others hanging. When a slice fails permanently the app cancels the parent and every live child, waits 2 s and deletes whatever is still alive. Deletion matters: the queue's cancel skips abandoned jobs, and an abandoned slice that later reported again would otherwise revive the cancelled simulation. Each workstation keeps its own job list on disk, so jobs survive closing the app, and reopening it resumes monitoring.

7 SELF-HEALING NODES

No node can start programs on another, so every node guards itself. A watchdog starts at logon, waits for shared storage to appear and keeps a single instance through a named mutex. Every 60 s it finds the queue client processes and whether each runs elevated, asks the server for the node's last heartbeat, and decides: no client process means *not running*, so it clears the stale process file and starts a client; a live process with no heartbeat for 300 s means *hung*, so it kills and restarts it. At least 120 s pass between restarts so that a client still starting is not killed again. It also warns when an elevated client or two clients are running, because an elevated client cannot see the user's mapped storage and two clients accept jobs they cannot both run. The client is started as the logged-in user, detached from the watchdog so that it outlives it.

Clock-proof heartbeats. Heartbeat times are stamped by the server's clock. One node whose clock ran 12.5 hours fast saw every heartbeat as 45,000 s old, restarted its client every two minutes and killed running slices. Ages are now measured against the newest heartbeat on the server whenever the local clock is implausible (behind, or more than 30 minutes ahead).

Storage self-test. A second client running under a system or administrator account cannot see mapped storage, and the watchdog cannot see that process either. Every three hours, while the node is idle, the watchdog therefore sends two tiny jobs to its own node that report the account they run under and check that a known file on shared storage is visible. A failing node is excluded from submission, the artist is told how to fix it, and it is re-

checked every ten minutes. This caught a misconfigured node within seconds of the feature going live.

Network log. A background thread reads the network adapters' byte counters every second and writes a line per minute with the average and peak throughput in each direction. On 1 Gb Ethernet the practical ceiling is about 110 MB/s.

8 SAFE UPDATES

The farm never runs the files people edit. Developers work in a development copy; published releases are immutable, numbered folders; a one-line pointer names the release every node should run; and a small, unversioned loader picks the release. Publishing runs four checks: every source file compiles; the engine imports in a fresh interpreter and passes its self-test; the app imports without opening a window; and the headless host process runs the scene tool's self-test, which also confirms that every node type it creates exists. Only then is the copy staged, checked again, renamed to the next release number and the pointer switched in one atomic write. A rollback points every node at an older release within a minute.

At load time the loader returns the release in the pointer if it passes a check in a separate process, otherwise the node's last good release, otherwise the newest release that passes. Watchdogs follow the same rule and restart onto a new release only after a self-test in a separate process; a failed release is logged, reported and not retried. We tested a broken release in the pointer (nodes stay on the old one), a half-copied release folder (ignored), an empty or garbled pointer (last good release runs), a syntax error in the loader (not restarted onto, picked up once repaired) and a broken development copy (publishing refuses).

9 PERFORMANCE

Our cluster has four nodes: three with 24 cores and 128 GB of RAM and one with 28 cores and 64 GB, linked by 1 Gb Ethernet to a network store. Each slice uses all cores of its node, so a node runs one slice. Table 3 and Figure 2 summarise the measurements.

Table 3. Measured wall-clock times.

Test	Nodes	Time	Change
Liquid, 240 frames	1	29.7 min	
Liquid, 240 frames	2	29.5 min	1% faster
Liquid, 240 frames	4	27.0 min	9% faster than 1
Production water, 51 frames, 8M particles	2	248 min	
Production water	4	216 min	13% faster than 2
Production water, baseline	1	~5.7 min/ frame	2 nodes: 4.9 min/ frame, ~15% saving
Test liquid, 110k particles, 60 frames	1	0.64 s/ frame	
Same, equal-width slices	2	1.15 s/ frame	slower than 1 node
Same, quantile borders	2	0.99 s/ frame	14% faster than equal widths

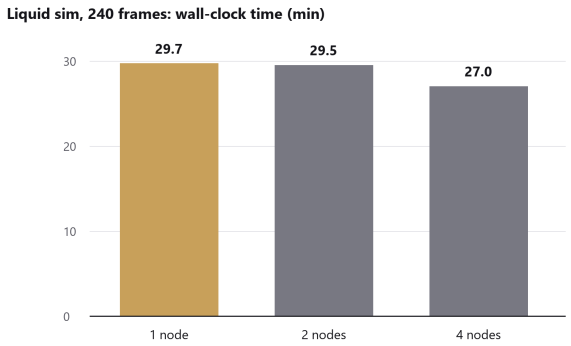

Fig. 2. A 240-frame liquid simulation sliced over one, two and four nodes. Four nodes are only 9% faster than one.

Why so little. The balance logs rule out the obvious causes. In the production shot the busiest slice was typically only 3% above average (0.7% with two slices and 1.7% with four in the 240-frame test), and only 0.5 to 1.9% of particles sat near a border. Neither uneven slices nor particle traffic explains the result, and writing the cache, about 300 MB every four minutes, is not the limit either. The likely cost is synchronisation: the distributed pressure solve waits for every slice several times per frame, and four slices have more borders than two. Quantile borders still help, cutting the two-slice time by 14% relative to equal widths, but on this network they cannot overcome the synchronisation cost for small scenes.

Policy. Simulations run on one node by default, and slicing is reserved for scenes that do not fit in one node's memory. For throughput, one node runs each simulation and the remaining nodes take other simulations, wedges or frame-split caches, which scale almost one-for-one.

10 LIMITATIONS AND FUTURE WORK

Sparse-grid smoke solvers that the host cannot partition run on one node. Volume slicing with three or more slices drifts a few percent from a single-node result late in a simulation. Each slice and each wedge consumes a licence of the host package, and a node must be switched on and logged in for its client to run. The app must stay open to deliver notifications, although jobs continue without it. Chains run across nodes only when the host's work-item messaging is allowed through each node's firewall.

The next step is to measure before buying faster networking: every node now logs its traffic, and a four-slice production run will show whether the links approach their ceiling. If they do not, a faster network will not help, and effort is better spent on dynamic rebalancing that remains compatible with narrow-band exchange.

11 CONCLUSION

A studio does not need a dedicated cluster or new software to distribute FX work reliably. Driving the host package's own primitives deterministically, guarding every node with a watchdog and refusing to run untested code turned a fragile manual routine into a farm that recovers on its own. Just as important, logging balance, borders and traffic

every frame let us replace intuition with evidence: on commodity networks, run simulations whole and split everything else.

A LESSONS FROM DEPLOYMENT

Problem	Cause	Fix
Jobs could not find shared files	Client started elevated or by a boot-time task	Storage mapping in client config; boot task disabled; watchdog starts client as user
Both slices ran on one node	Dedicated tracker and CPU count let the queue pack slices	All cores per slice, no dedicated tracker
One slice nearly empty	Slicer placed off the fluid	Measured the slicer rule; centre on the bounds
Slices hung on a re-run	Controls node linked to itself	Literal links, checked before submit
Clients shown as abandoned, 40 to 60 s delays	Server name resolved to IPv6 first; queue listens on IPv4	IPv4 addresses everywhere
A cancelled simulation came back	Cancel skips abandoned jobs	Cancel, then delete leftovers
Slices killed every two minutes	Node clock 12.5 h fast	Heartbeat age against server time
Whitewater became NaN; duplicate ids	Stitched slices give NaN velocities; parts number ids from 0	NaN velocities zeroed; ids unique per part
Jobs failed on a node that looked healthy	Hidden second client under a system account	Storage self-test every 3 h
One bad edit could stop every node	Nodes ran the edited files	Tested, versioned releases with last-good fallback

REFERENCES

J. U. Brackbill and H. M. Ruppel. 1986. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics* 65, 2, 314 to 343.

Robert Bridson. 2015. *Fluid Simulation for Computer Graphics* (2nd ed.). CRC Press.

Joel Wretborn, Marcus Schoo, Noh-hoon Lee, Christopher Batty and Alexey Stomakhin. 2026. A practical partitioner for distributed simulations on sparse dynamic domains using optimal transport. *ACM Transactions on Graphics*.

Yongning Zhu and Robert Bridson. 2005. Animating sand as a fluid. *ACM Transactions on Graphics* 24, 3, 965 to 972.