THINK SERVICES
A DIVISION OF UNITED BUSINESS MEDIA LLC

GDC 10

Learn. Network. Inspire.

www.GDConf.com

Game Developers Conference® | **March 9-13, 2010** | Moscone Center | San Francisco, CA

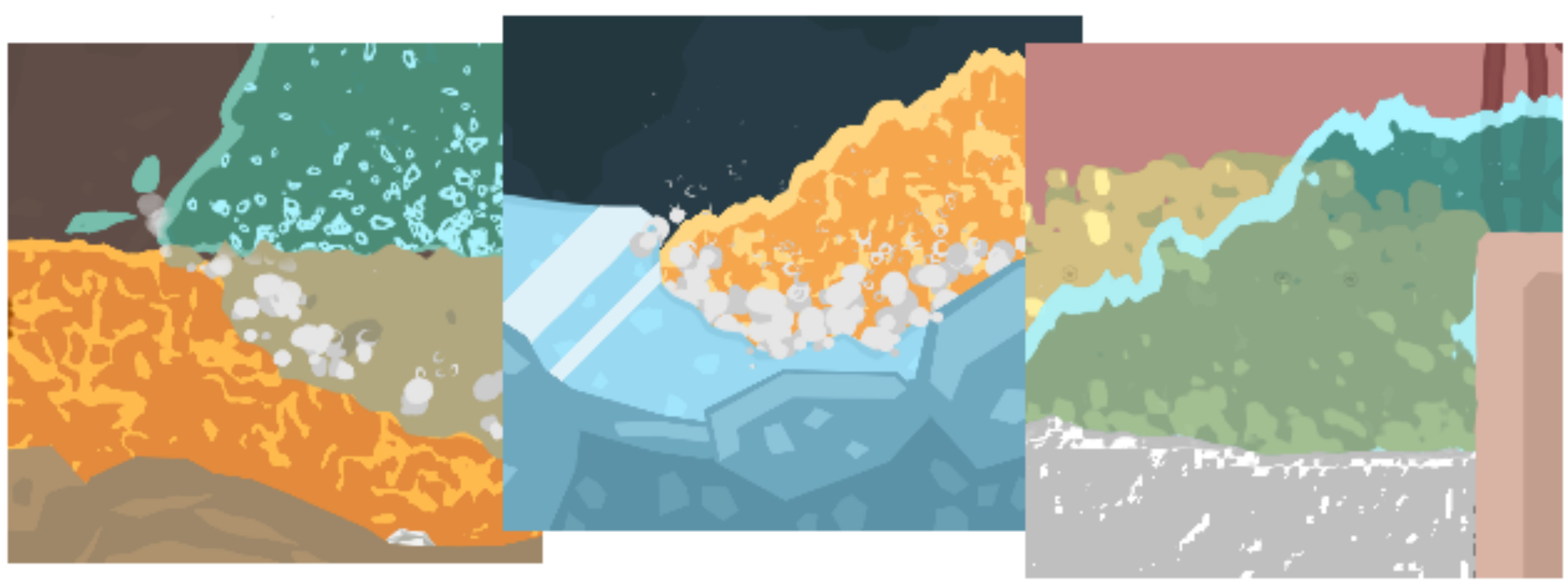
Game Developers Conference®
March 9-13, 2010
Moscone Center
San Francisco, CA
www.GDConf.com

Go With The Flow: Fluid and Particle Physics in PixelJunk Shooter

Jaymin Kessler
Q-Games
Technology Team
jaymin+gdc@q-games.com

Shooter overview

- Game designed around mixing of various solids, liquids, and gasses
  - Magma meets water, cools, and forms rock
  - Ice meets magma and melts
  - Magnetic liquid meets water to form a toxic gas, just like in real life
  - Lasers melt ice and rock into water and magma
  - Other cool effects like explosion chain reactions, and water turbulence



Video (for those who haven't played it yet)

Video (for those who haven't played it yet)

Overview

- SPU based fluid simulation
  - Parallel particle sim algorithms
  - Game design built around mixing of different fluids
  - Universal collision detection mechanism
  - Particle flow rendering
- Collision detection by distance field
  - Real-time SPU and GPU algorithms
- Level editing via stage editor
  - Topographical design via templates
  - Particle placement

Episode 1
Fluid Simulation

PIXELJUNK

Existing fluid simulation algorithms

- Smoothed particle hydrodynamics
  - Divide the fluid into particles, where each has a smoothing length
  - Particle properties are smoothed over smoothing length by a kernel function
  - Particles affected by other particles close by
  - SPH formulation derived by spatially discretizing Navier-Stokes equations
  - Used in astrophysics!

What we actually used

- Goal: practical application in-game
  - Ease of implementation
  - Rapid control response
  - Physical accuracy
  - Cater to the strengths of the SPUs
    - No SIGGRAPH framerates
- Fluid system developed for Shooter
  - 2D particle collision simulation
  - 32,768 particles running @ 60fps on 5 SPUs (could have done way more if needed ;))

Verlet integration

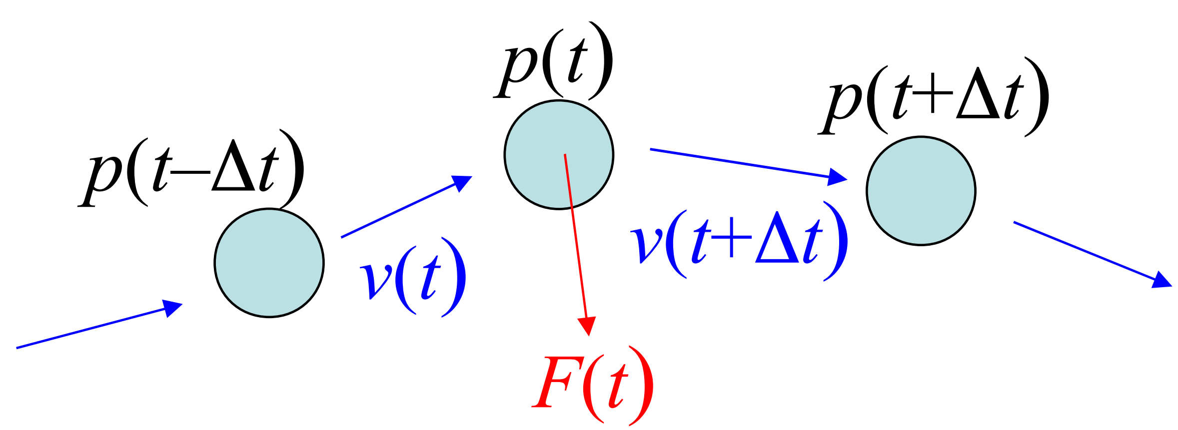
- The good
  - 4th order accurate (Euler is 1st)
  - Greater stability than Euler
  - Time-reversibility
- The bad
  - Bad handling of varying time steps
  - Needs 2 steps to start, start conditions are crucial
- Time-corrected verlet helps

The version we used

- Applied to elemental particle sim
- location p(t) as a function of time t against velocity v(t) and ext force F(t)
- For mass m and sim interval Δt

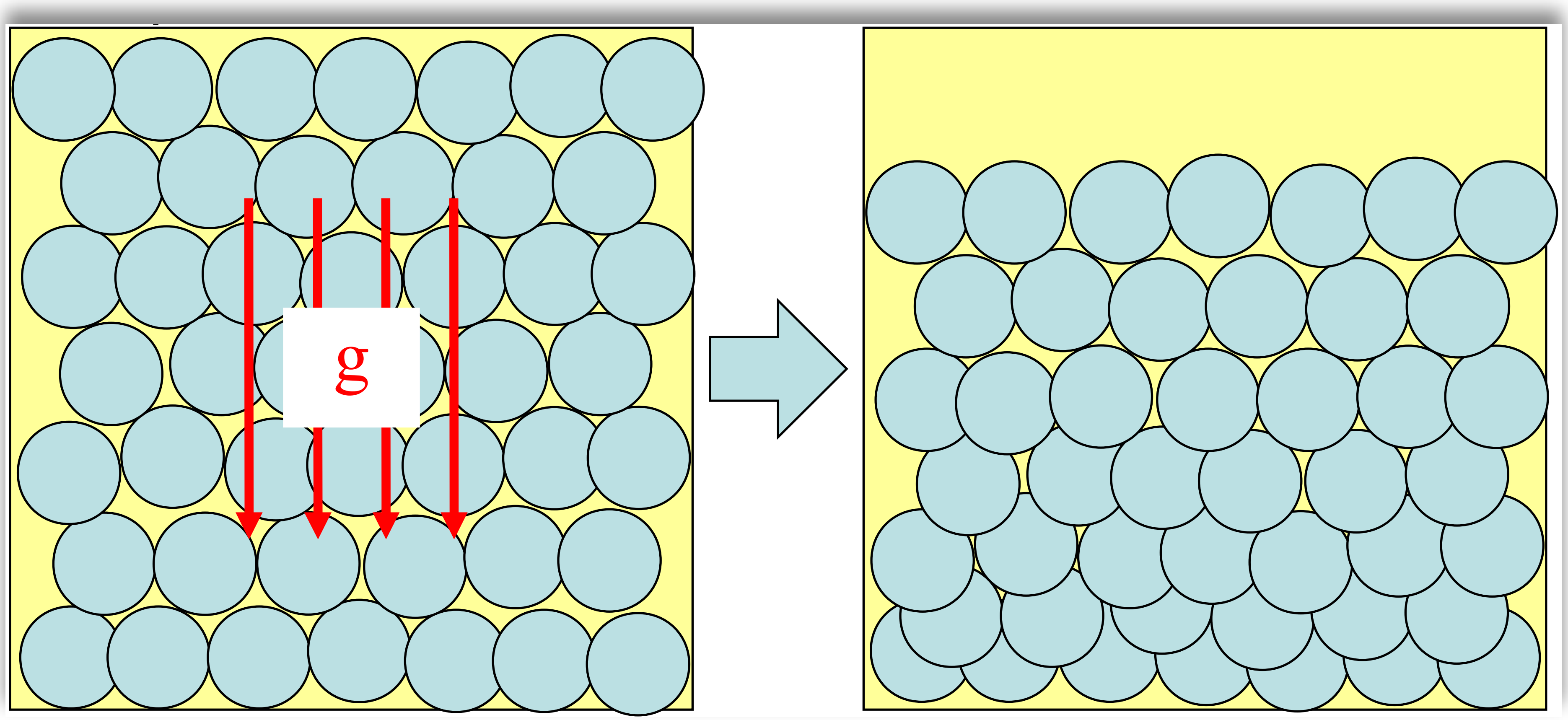
$$p(t+\Delta t) = p(t) + v(t)\Delta t + F(t)\Delta t^2 / 2m$$

$$v(t) = (\, p(t) - p(t-\Delta t) \,) / \Delta t$$



Incompressibility of liquid

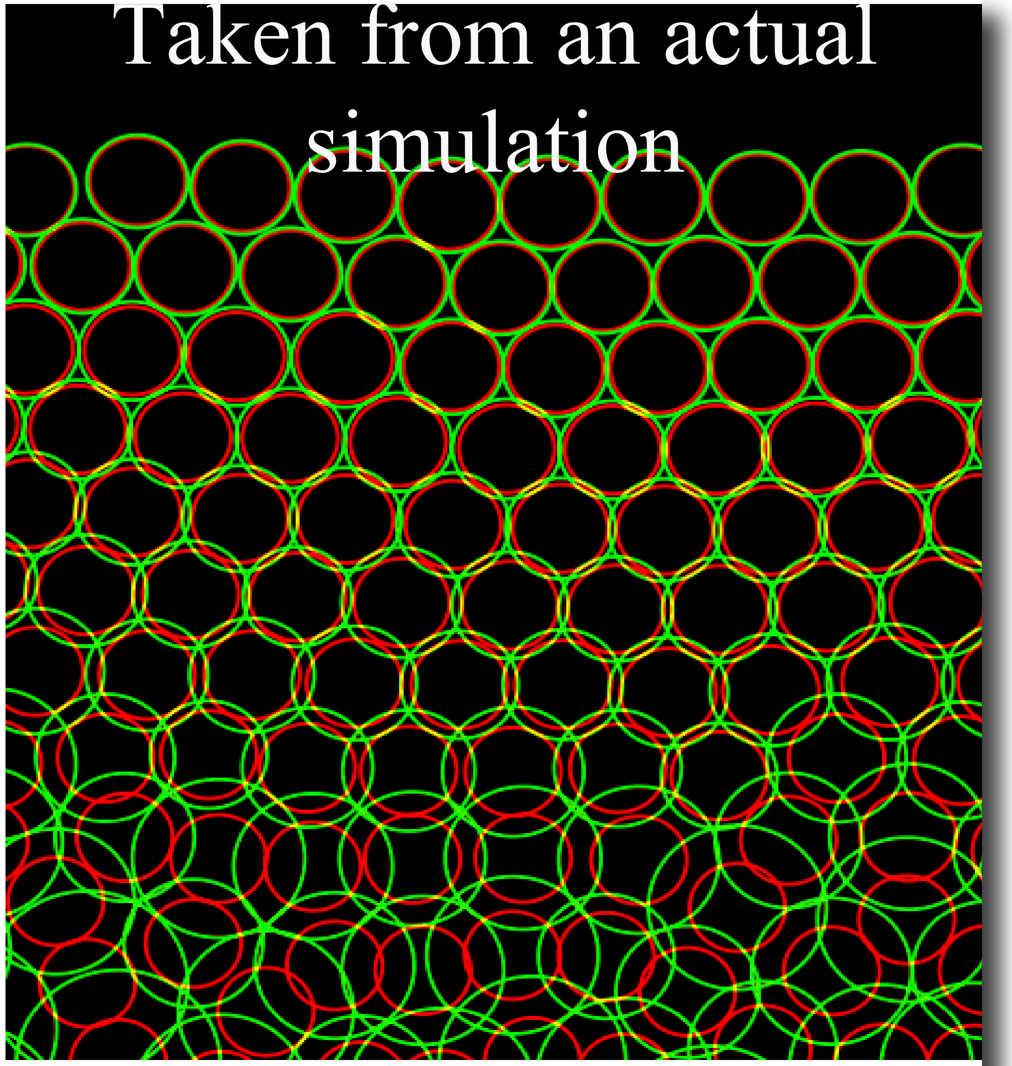
- Liquids don't compress or expand to fill volumes, but...
- In our model, mass and gravity can compress lower particles
- Don't worry! We have a fix

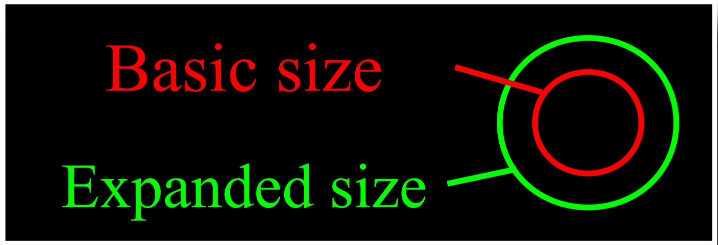


Maintaining the incompressibility of liquid

- Must maintain constant distance between particles
- Particles have an adjustable radius bias
- Each frame:
  - Calculate the desired radius bias
    - Based on max ingression of surrounding particles
  - Lerp from current bias to desired bias
    - Two different rates for expanding and contracting
    - Contraction ~4x faster than expansion

Maintaining the incompressibility of liquid



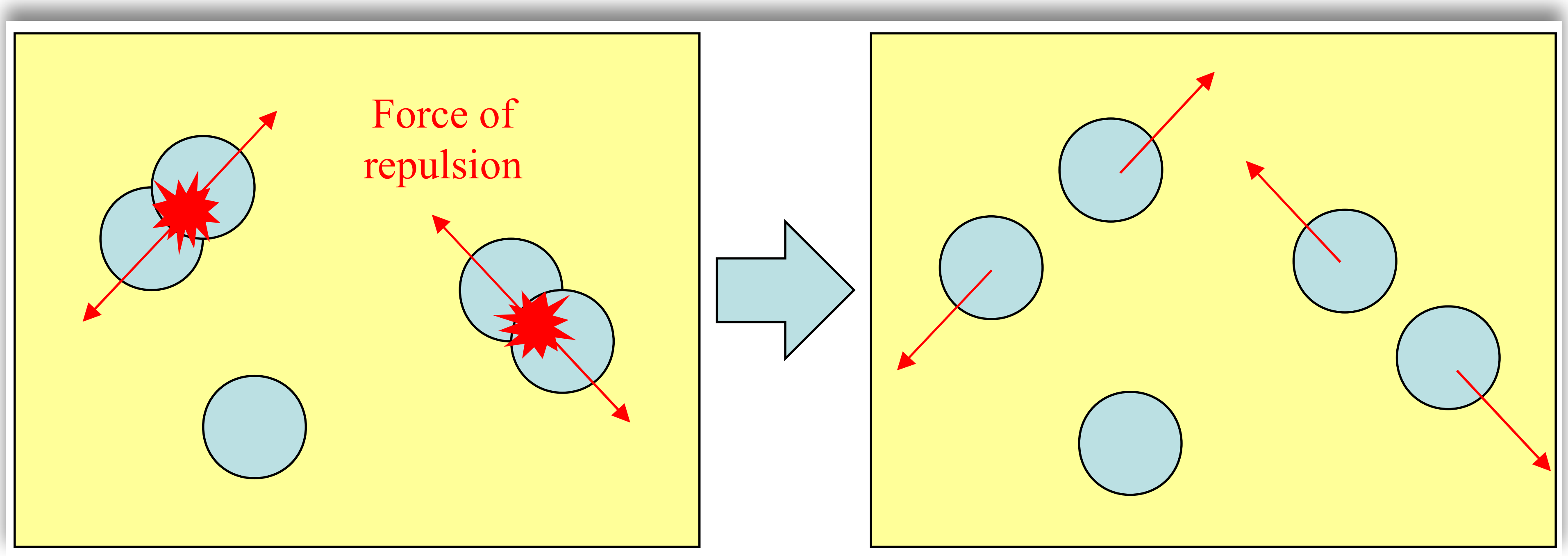


Keeping particles apart

- Add repulsive force in the space between colliding particles
- Force of repulsion proportional to the number of colliding particles
- Increasing number of particles creates fluid-like behavior

Keeping particles apart

- Simple-ish computation model
  - All particles perfectly spherical, but with varying radius size
  - Helped with ease of implementation

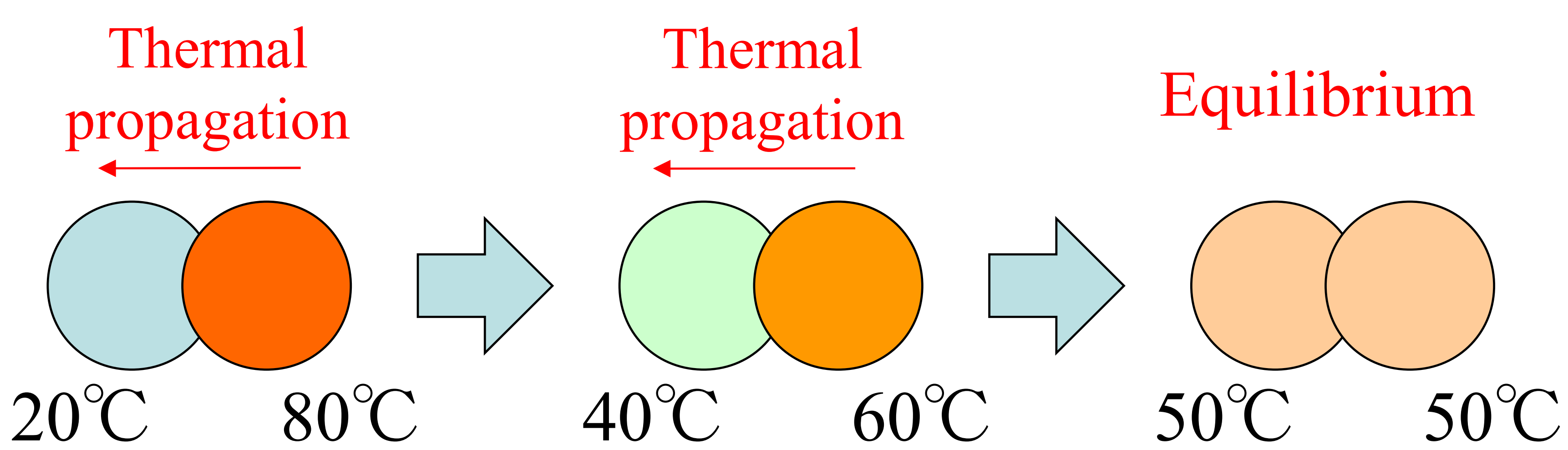


Not all particles created equal

- Different particle combinations have different force of repulsion values
- Different chemical reactions simulated when fluids mix
- Different mass
- Some have rigid bodies, others don't
- Particle types propagate heat differently
  - i.e. magma cools to form a rock-like solid

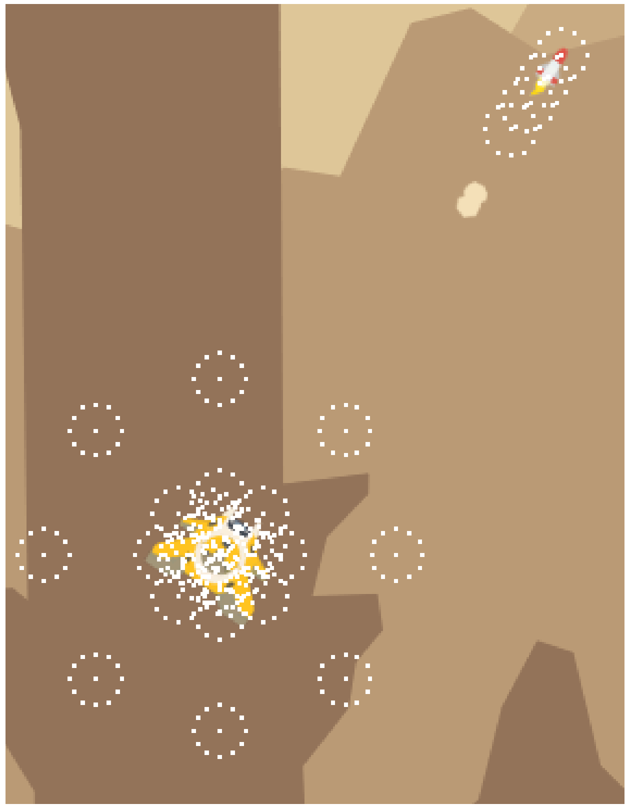
Heat propagation

- Each particle carried thermal data
- When particles collide, heat is propagated
  - Warmer particle to cooler one
  - Same algorithm we use for force of repulsion
  - Particle types have different thermal transfer values



One other (mis)use of the particle system

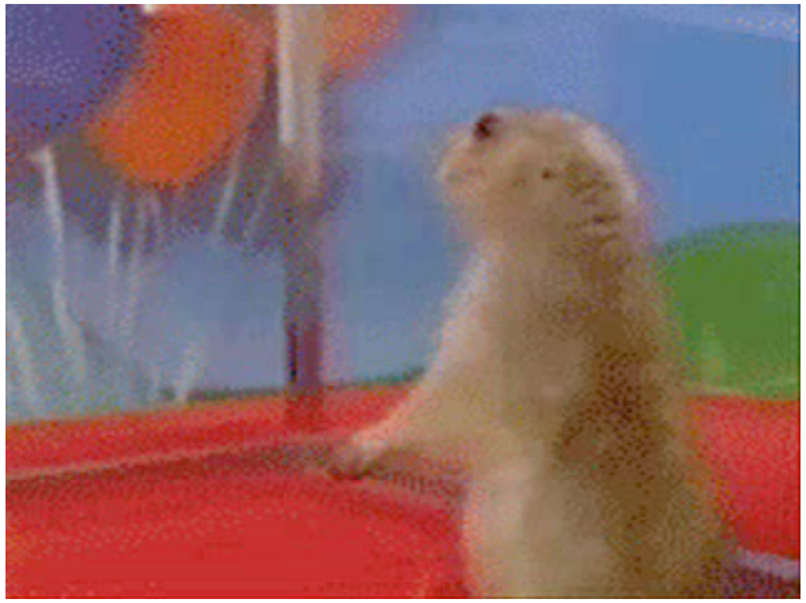
- In-game collision detection
- Characters, missiles, etc. are surrounded by special dummy particles (interactors)
  - Some benefits include
    - No need to write lots of different collision detection systems
    - Depending on the location of the interactor particle, pretty much any collision can be simply detected and tracked



SPURS jobchain (in words)

- Yes, we really used SPURS
- SPU jobchain:
  1) Collision detection and repulsive force calc
  2) Force unification (for multi-cell particles)
  3) Particle update (verlet)
  4) Particle deletion, only 1 SPU used
  5) Grid calc for the next frame
- PPU processing:
  - Particle generation
  - Jobchain building

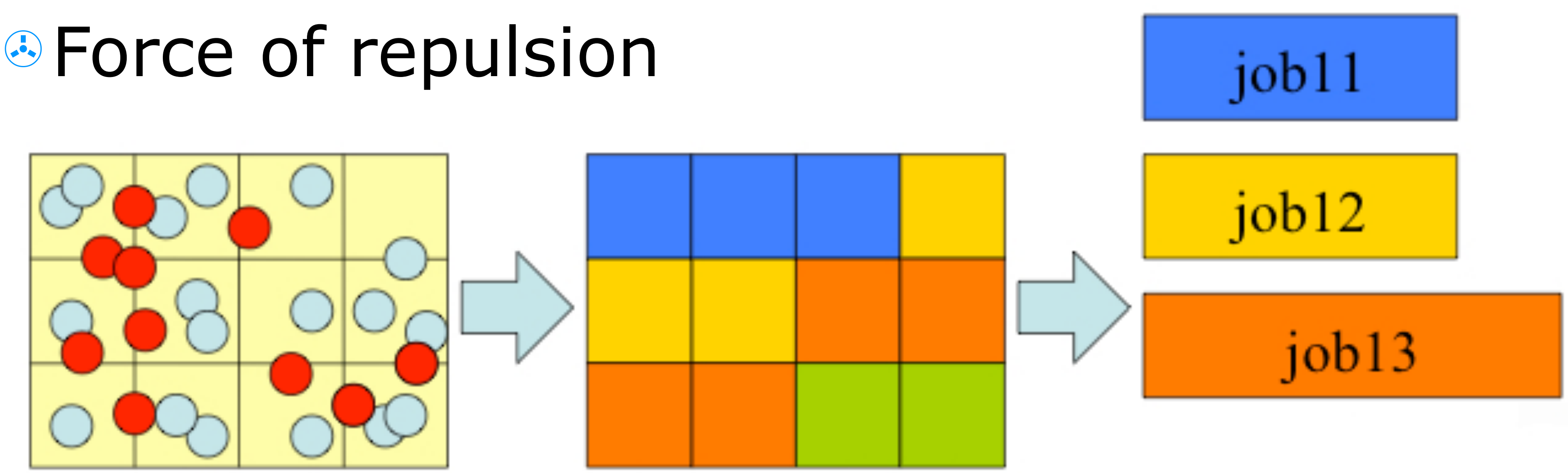
SPURS jobchain (in words)



- Yes, we really used SPURS
- SPU jobchain:
  1) Collision detection and repulsive force calc
  2) Force unification (for multi-cell particles)
  3) Particle update (verlet)
  4) Particle deletion, only 1 SPU used
  5) Grid calc for the next frame
- PPU processing:
  - Particle generation
  - Jobchain building

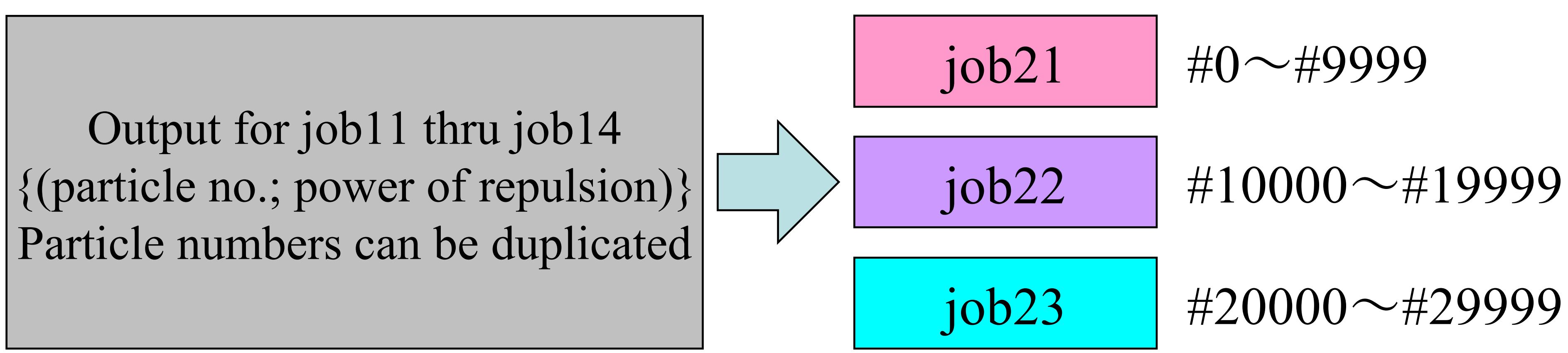
Collision job

- Collision detection between every particle in a cell
- Several cells pooled together to make one job
- Jobs are divided to help with load balancing
- Output
  - Particle number
  - Force of repulsion



Force unification job

- If a particle is processed in more than one cell, we have to unify the results
- Output: Unified force of repulsion, acceleration, and other info by particle number

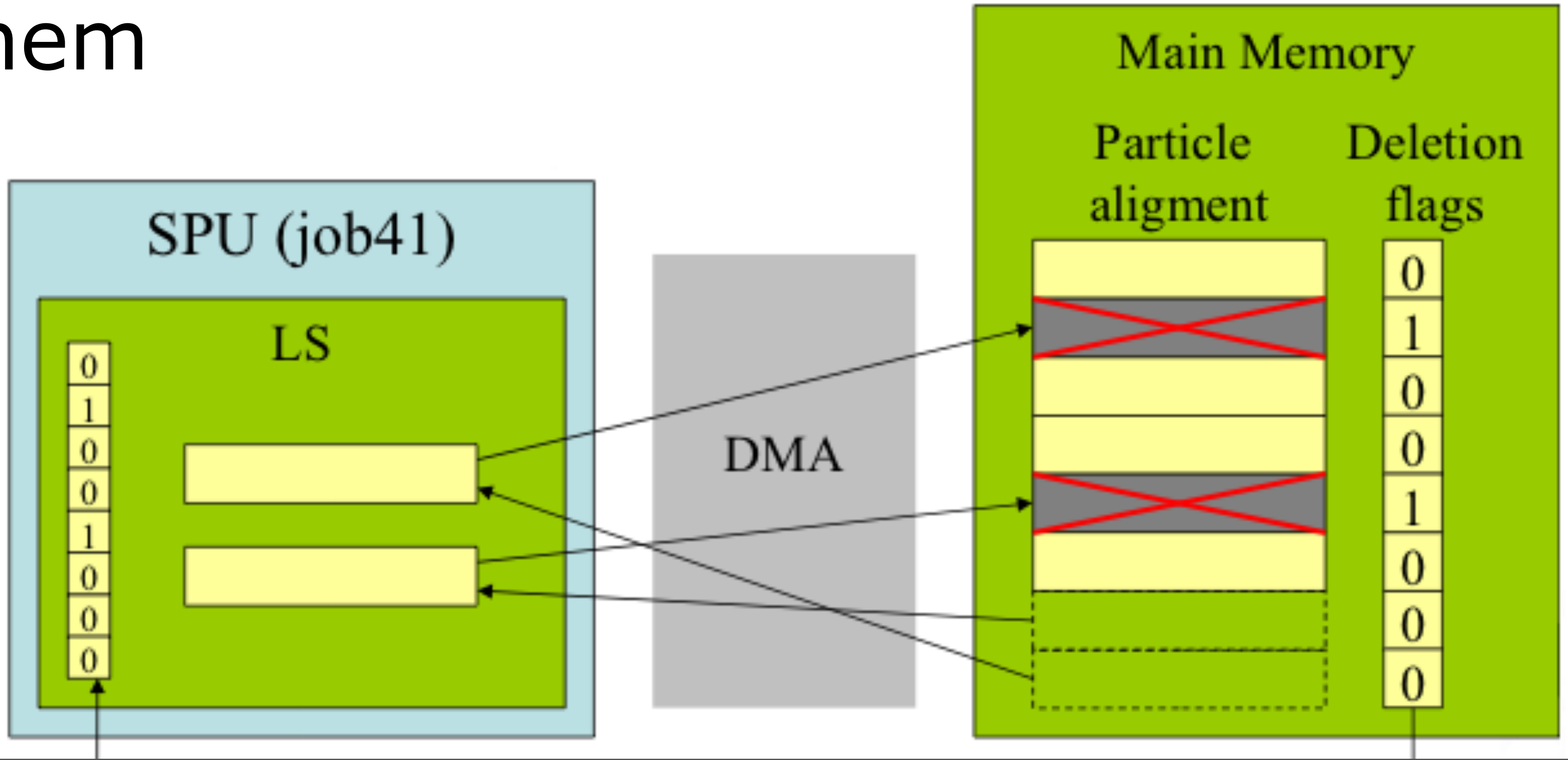


Update job

- This is the BIG one (in terms of code size)
- Particle physics calculations, including verlet integration
- PPU notification of interesting events
  - Like abrupt changes in fluid direction, triggering effects
- Output
  - Updated particle data
  - Particle deletion info
  - Various other flags

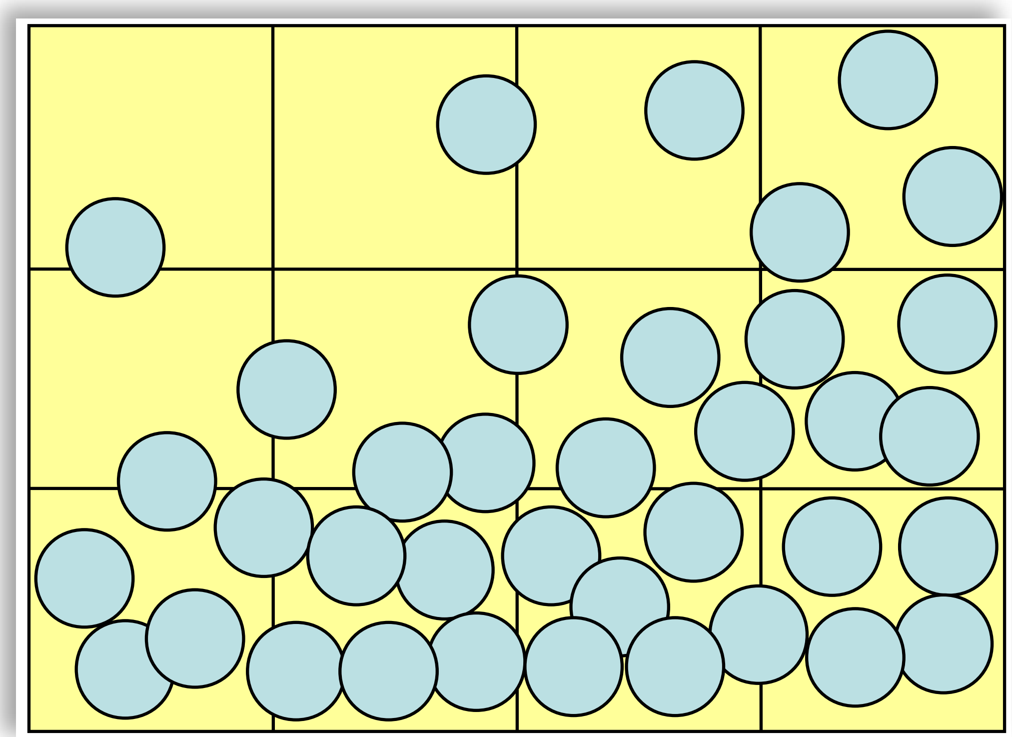
Particle delete job

- Only run on one SPU
- Very few particles deleted, around 10 per frame
- Take valid particles off the end, and overwrite deleted particles as we find them



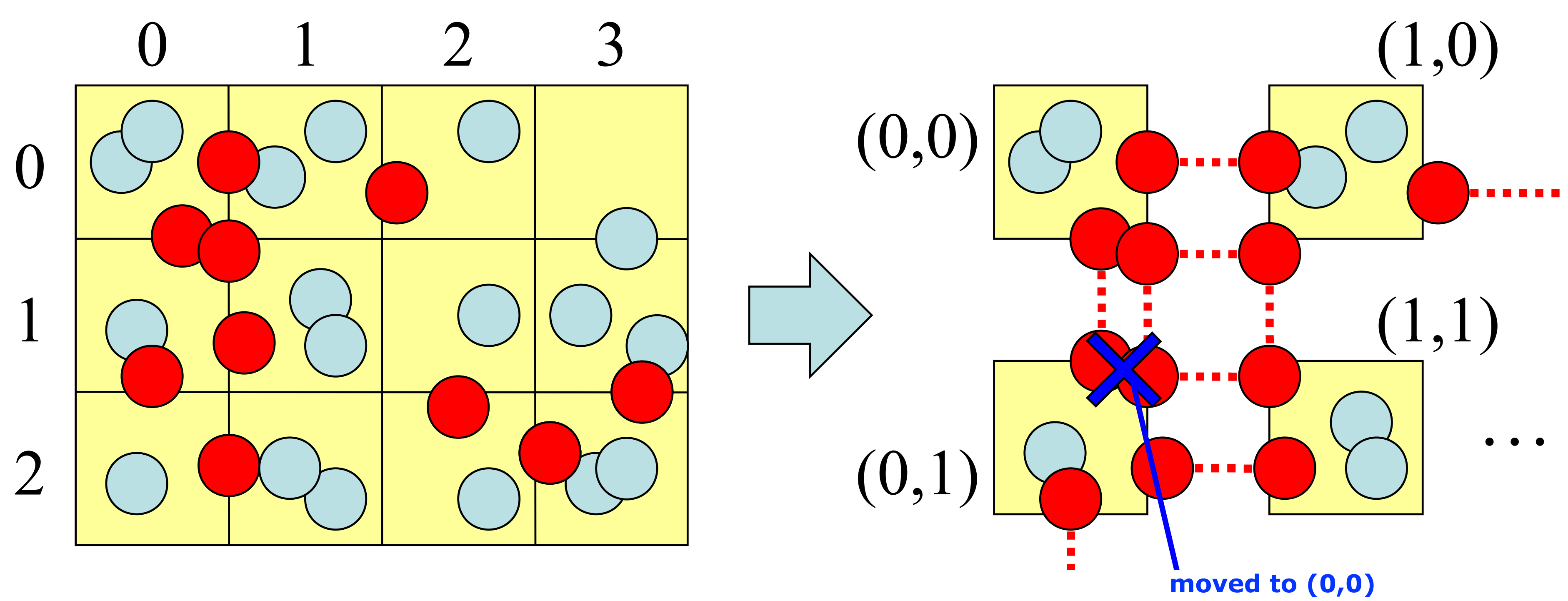
Particle grid division

- Used to parallelize workload
- $O(n^2/k)$ for $k \approx 1232$ cells, or a 44x28 grid (better than $O(n^2)$)
- Multiple cells per job
- But what if a particle is on the border between two cells?

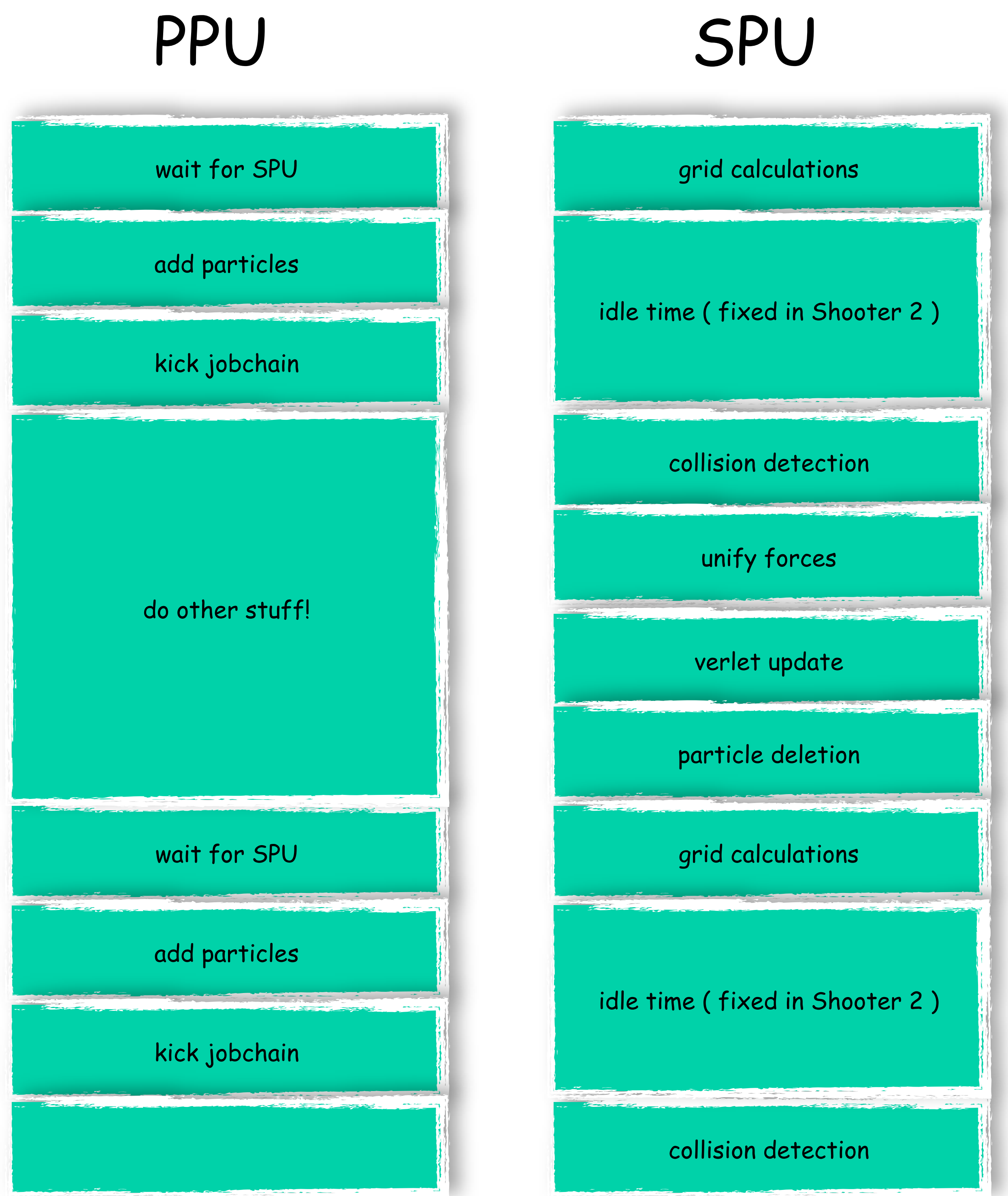


Particle grid division

- Particles are processed (hit test) with every cell they touch
- In the next phase, we unify all forces acting on a particle
- After merge, the particle belongs to the upper left most cell it touches

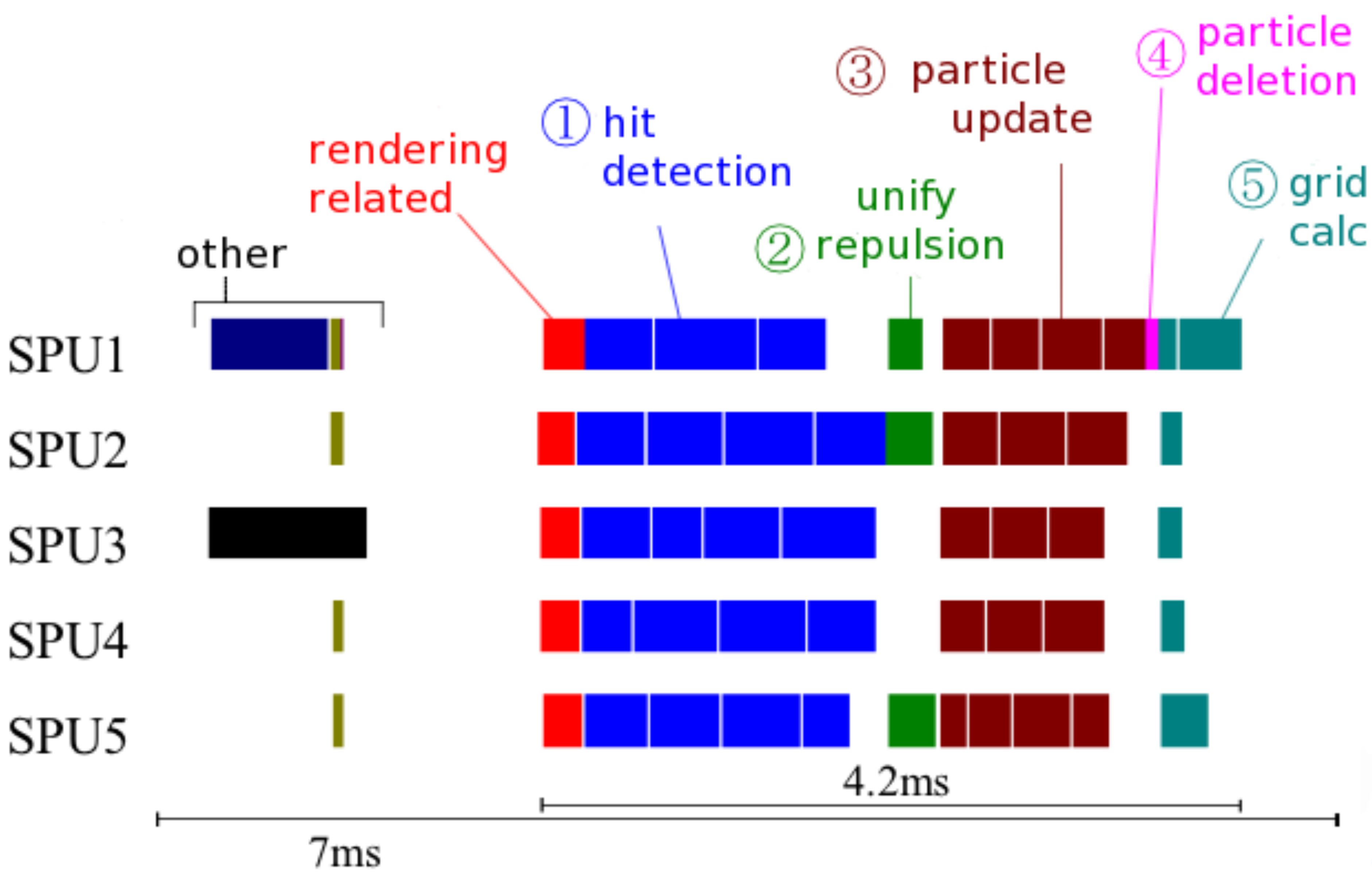


SPURS jobchain (in pictures)



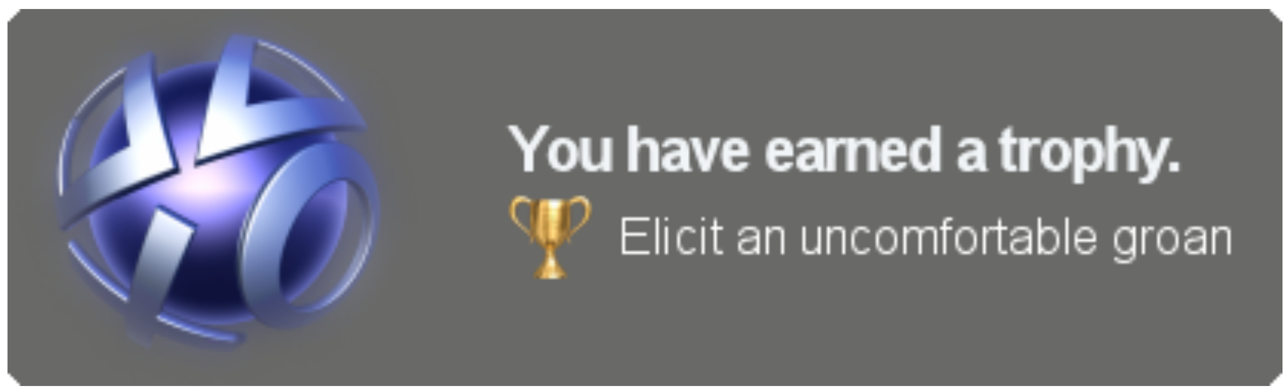
SPURS jobchain (in-profiler)

~9000 particles

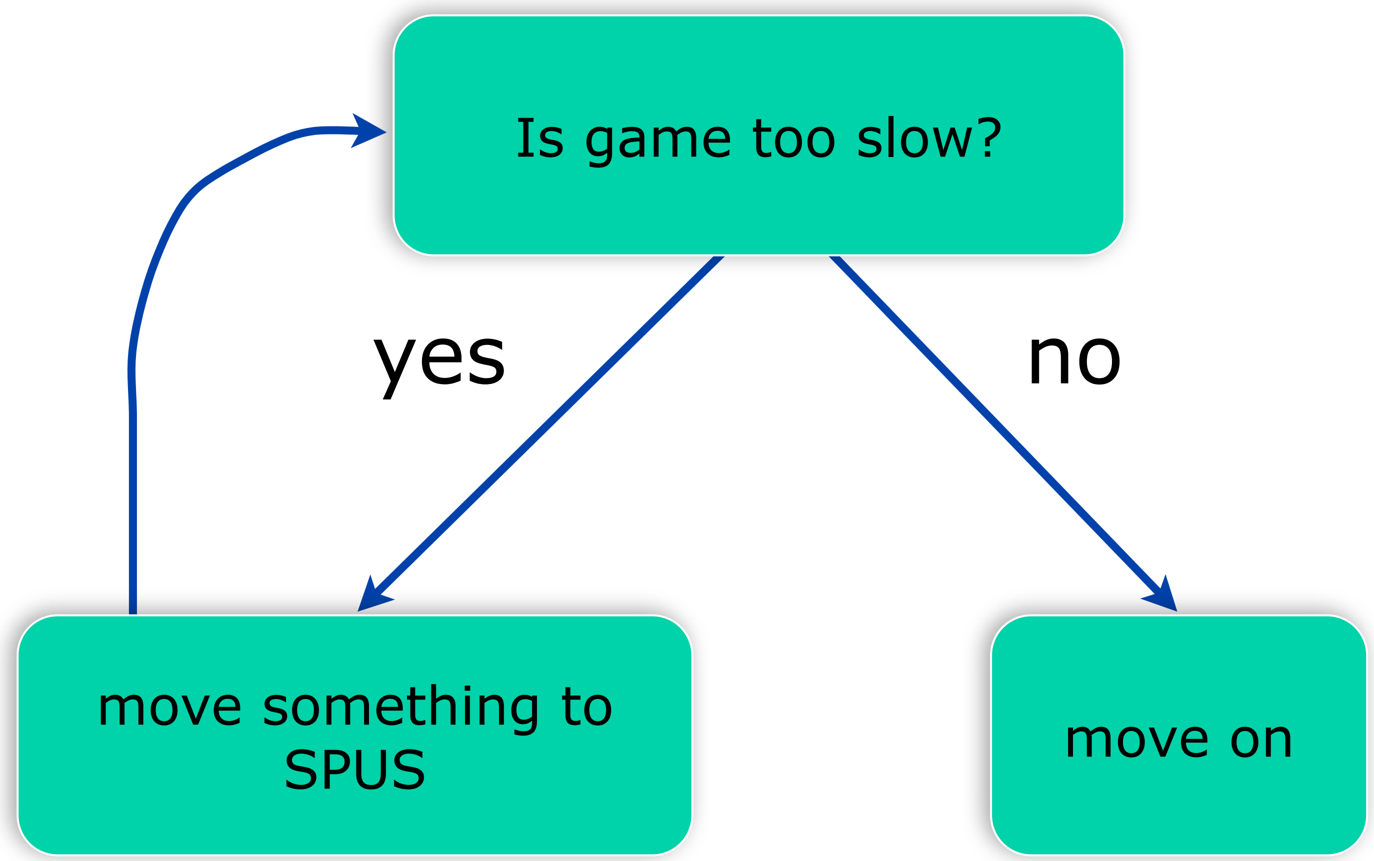


Painfully obvious optimizations

- Heavy use of SoA
  - big win even when converting to and from in the same job
- Avoid scalars (especially multiple writes) like the plague
  - Or don't read/write the same buffer in the same loop
- As branch-free as possible
- Software pipelining and unrolling
  - But less LS left for particles
- Favor intrinsics over asm :(
  - Dylan's inline asm site
  - Possible through compiler communication


You have earned a trophy.
Elicit an uncomfortable groan

Christer’s© algorithm



Episode 2

Rendering

Fluid rendering

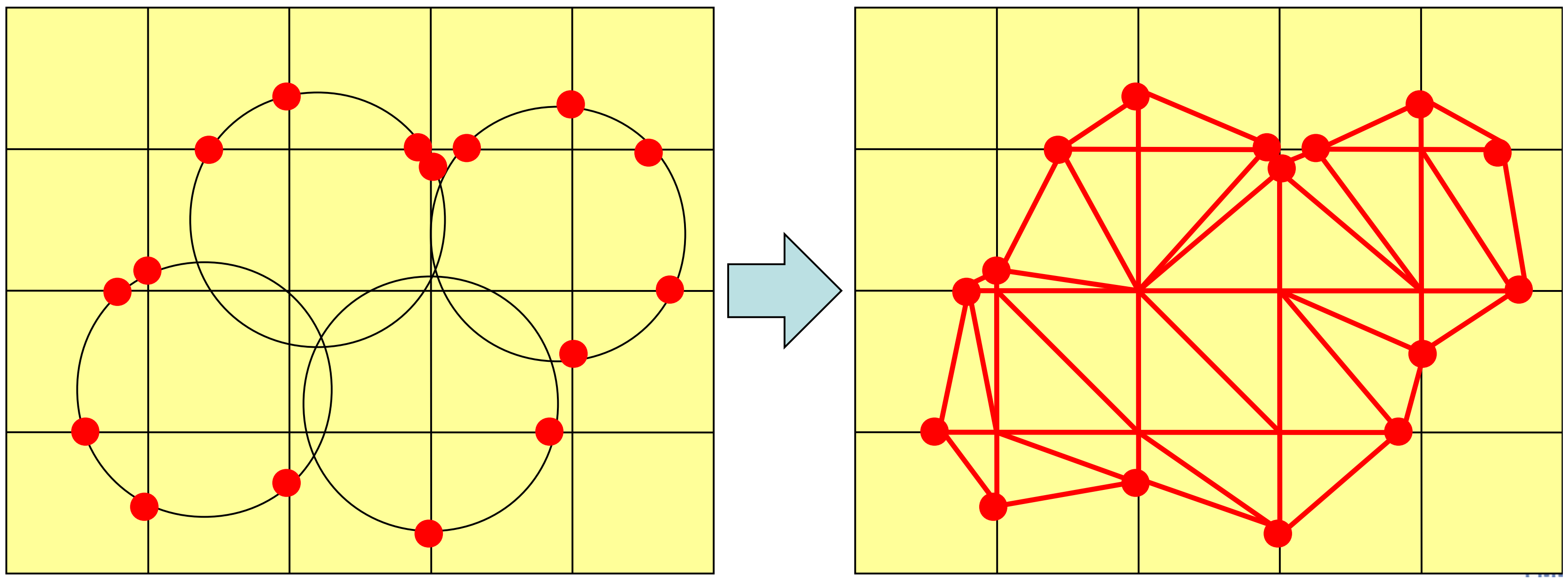
- Render particles in a vertex array
  - Three basic particle types: solid, liquid, and gas
  - Each is rendered to a different offscreen buffer
  - A vertex array is required for each particle type
- Upper particle limit is approx 30,000
  - three different vertex arrays for three particle types with 30,000 particles each is a waste
  - One vertex array can be used for all three types

Fluid rendering

- Vertex array built on the SPUs
  - 1~5 SPUs used depending on the num particles
  - Lists built in LS and DMA'd to main memory
- The vertex array is 64 sectors
  - Each sector contains one particle type
  - Max 512 particles per sector
  - Atomic DMA to coordinate shared list updates

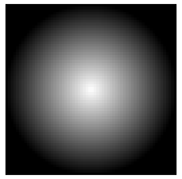
Fluid rendering

- Grouped particles rendered as a smooth flowing fluid
- Existing example: marching square/cube
  - Related particles depicted as a polygon mesh
  - The grid has to be fine, or liquid movement isn't smooth
- Currently patented
  - Not by us

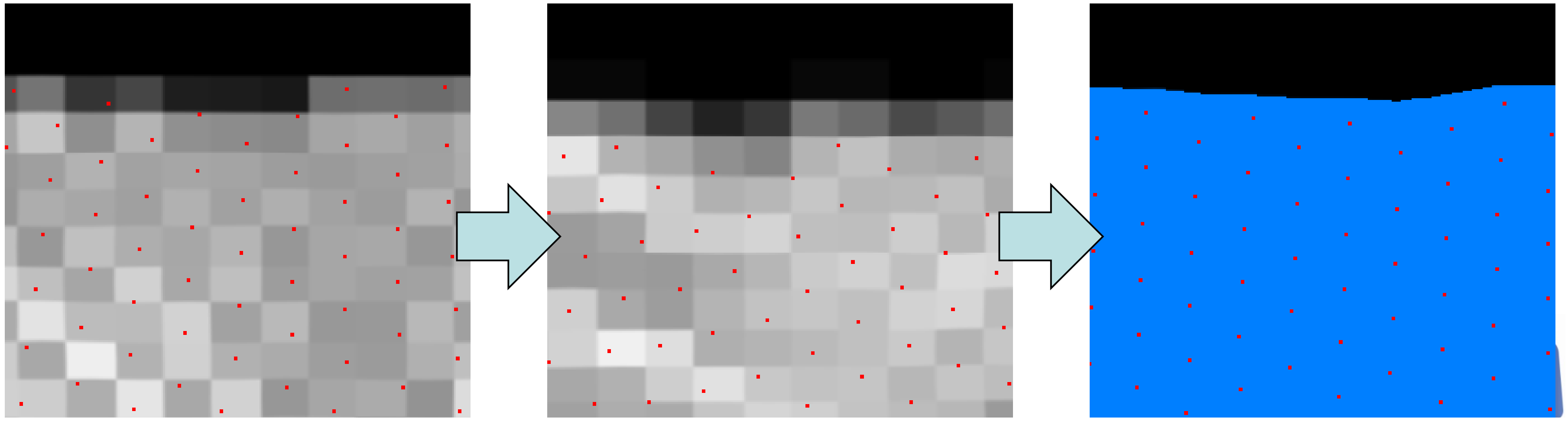


Fluid rendering

- Render particles to a low-res offscreen buffer with a luminance texture



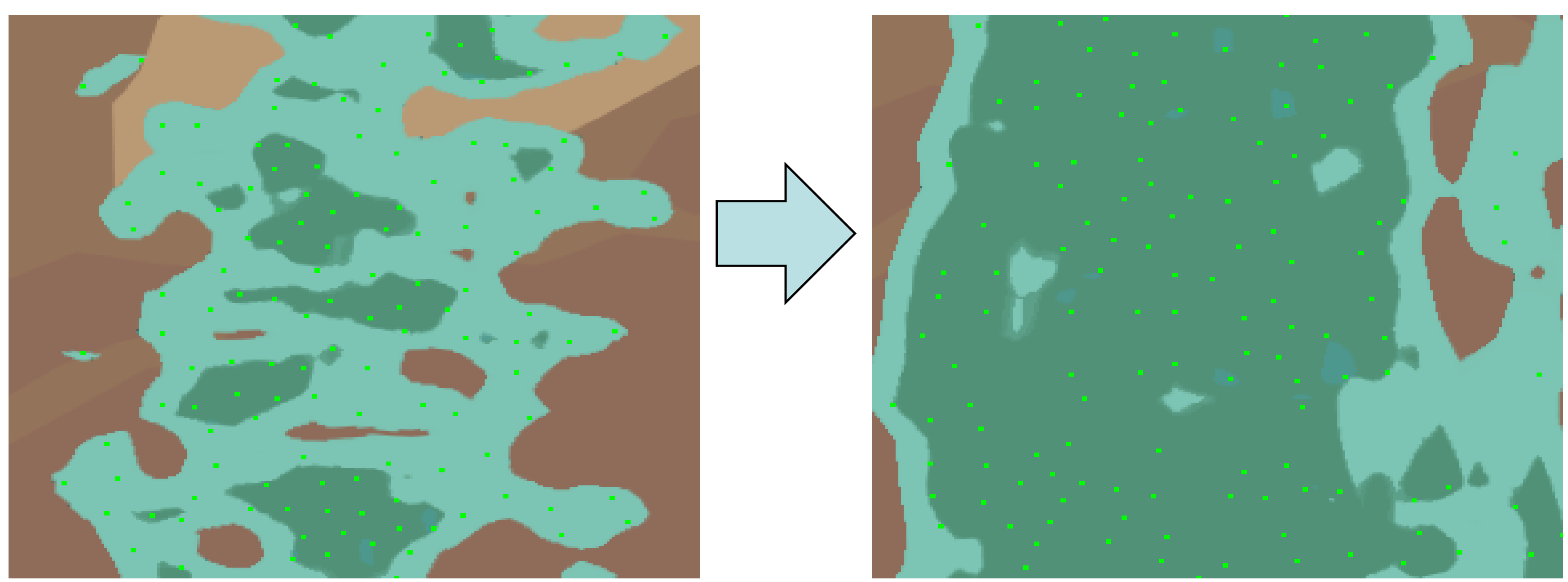
- Blur the offscreen buffer
- Scale up with bi-linear filter
- Use the resulting brightness to color the liquid



Low-res off screen After blur added Bi-linear filter and step

Cohesiveness

- Free falling liquid causes particle distances to increase
  - Liquid mass loses cohesiveness
  - Opposite problem as compression
- Solution: don’t fully clear the buffer
  - Image lag effect maintains cohesiveness, even in motion



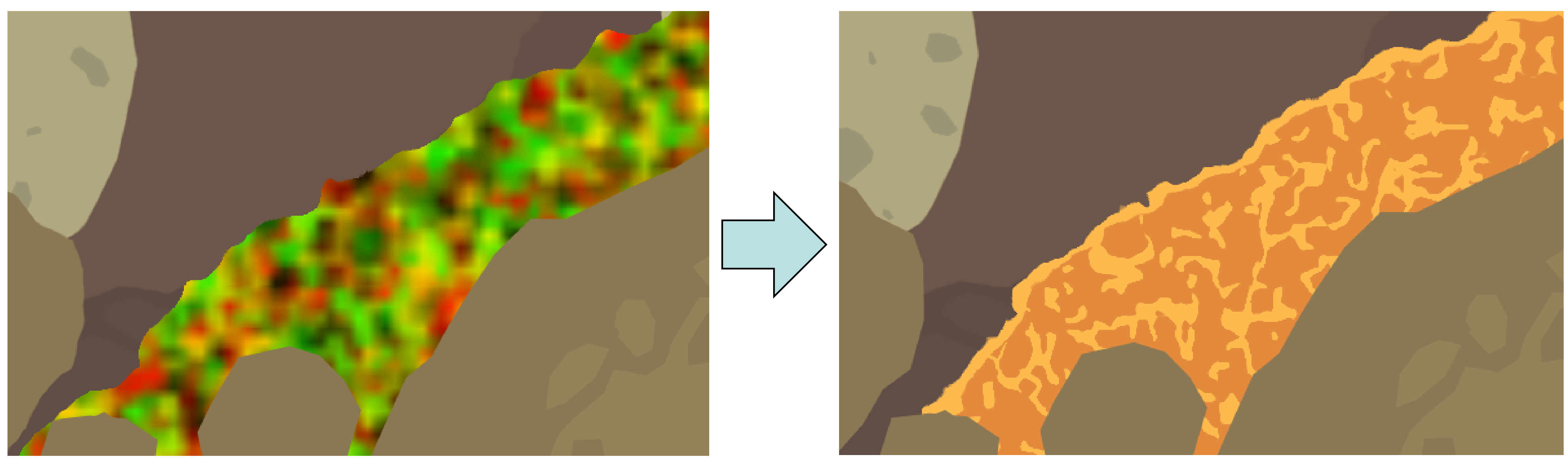
Water surface AA

- When rendering liquid to offscreen buffer, use a smooth step function
- Two thresholds used for water surface and for tinting

SPU update job detects sudden changes in liquid speed and direction, and notifies the PPU to add foam effects

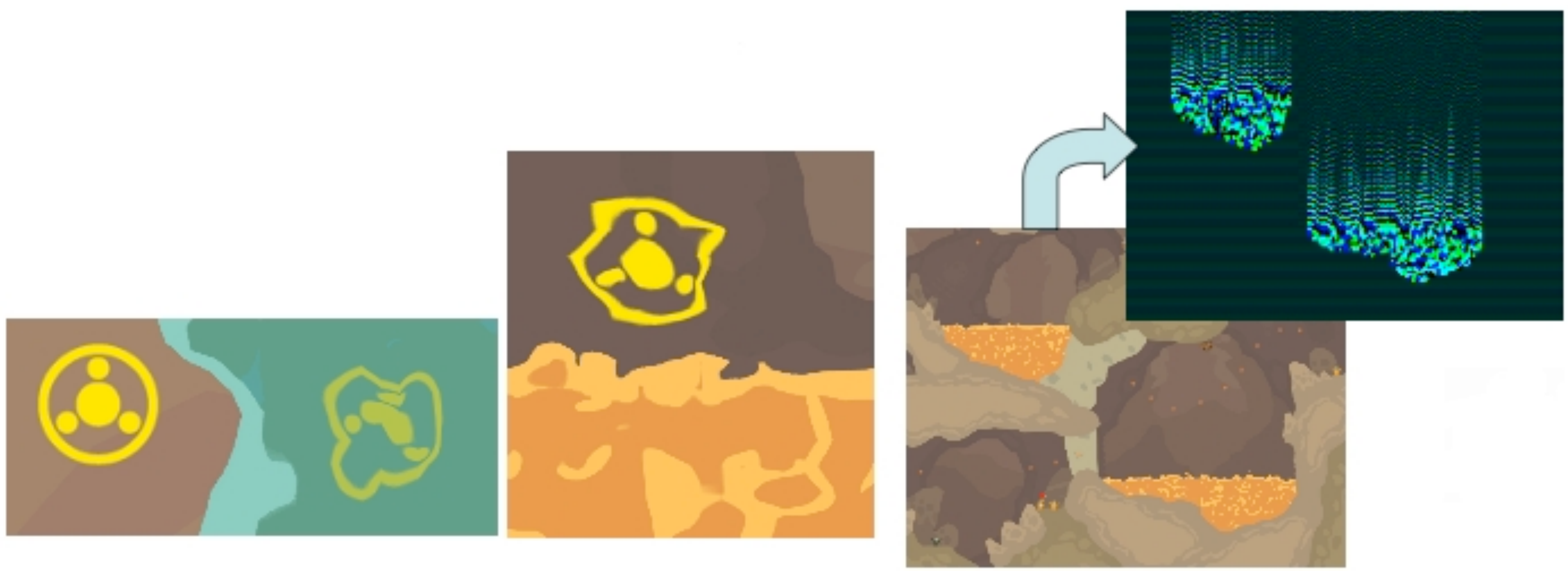
Depicting movement

- Create a flow pattern to show movement
  - Each particle gets a fixed random UV value [0..1]
  - UV value converts to RG value
  - Use a different color where RG is 0.5f, 0.5f



Refraction

- From water and from magma heat
- Ping-pong between offscreen buffers (tex feedback processing)
- Degree and direction depends on particles fixed UV

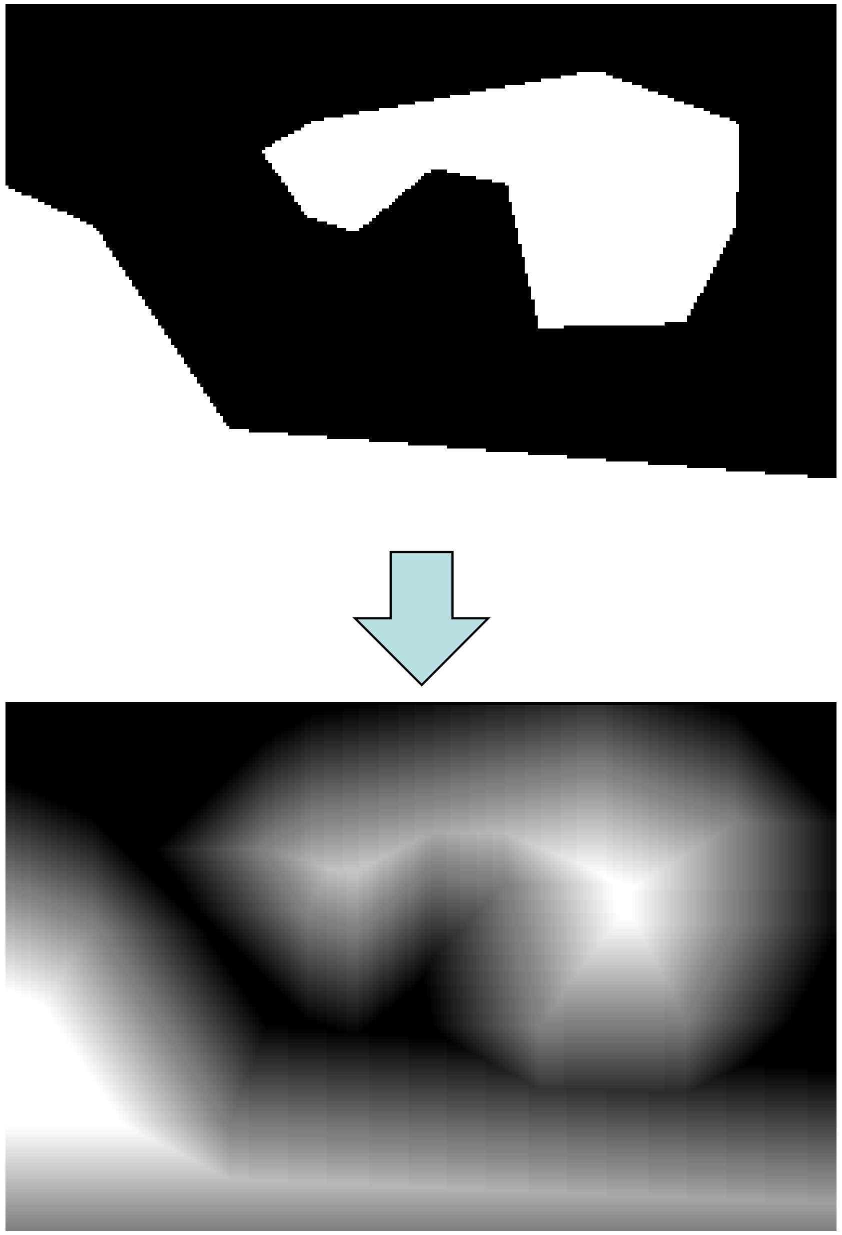


Episode 3
Distance Transform

Distance field

- How does it work?
  - Binary input image
    - Walls are white
    - Space is black
  - Output image
    - Wall core is bright
    - Wall boundary is 0.5f
    - Gets darker as you move away from wall
    - 2 distance transforms: static and dynamic
- Sample uses
  - Wall collision detection
  - Making enlarged fonts look better

Distance transform

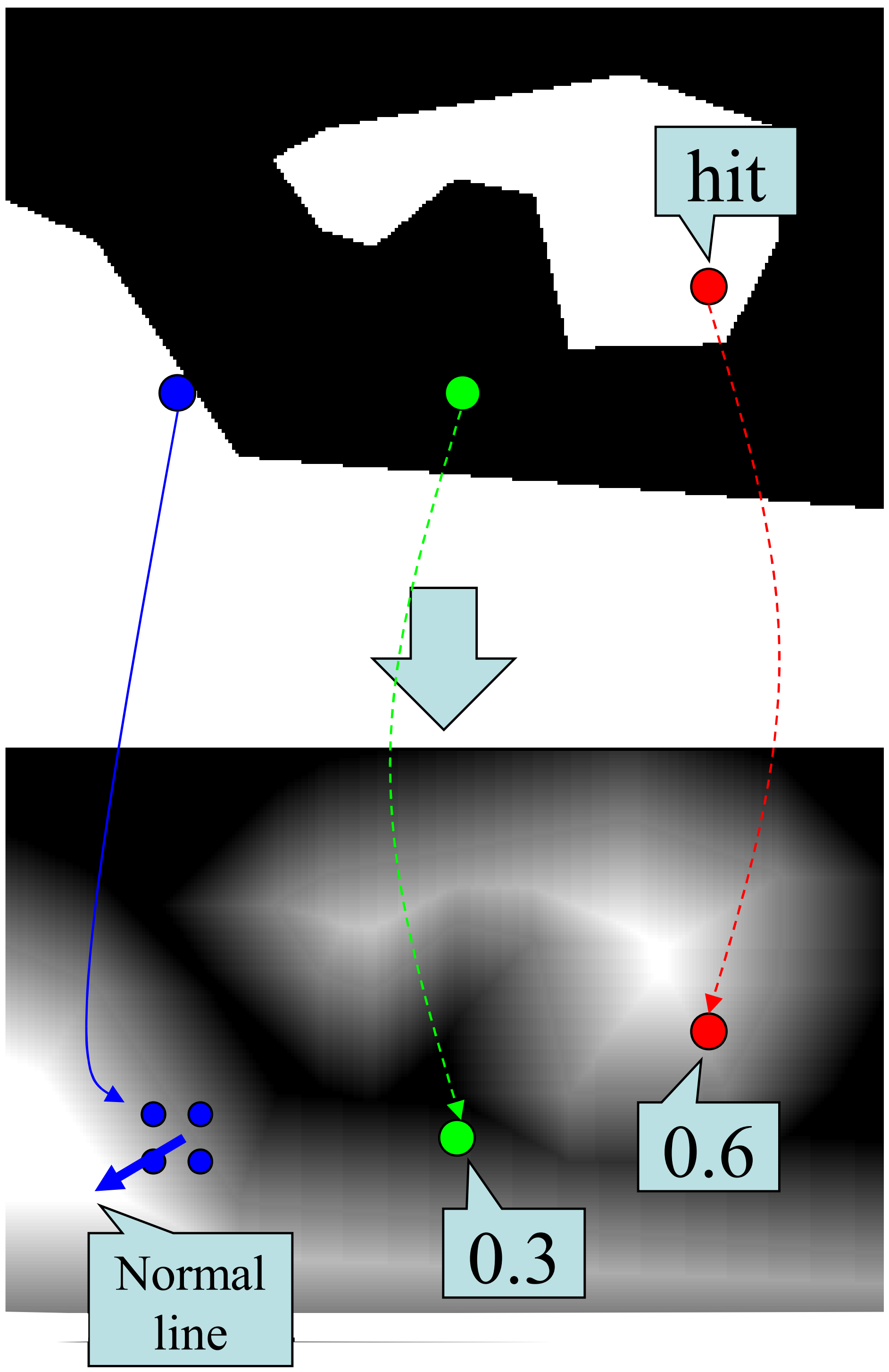


Using distance transform for wall collision

- Look up character's pos in the distance field
  - > 0.5f ⇒ collision
  - ≤0.5f ⇒ no collision
- Moving away from a collision
  - Get 4 distance field values near collision
  - Look at gradient to move away from the wall

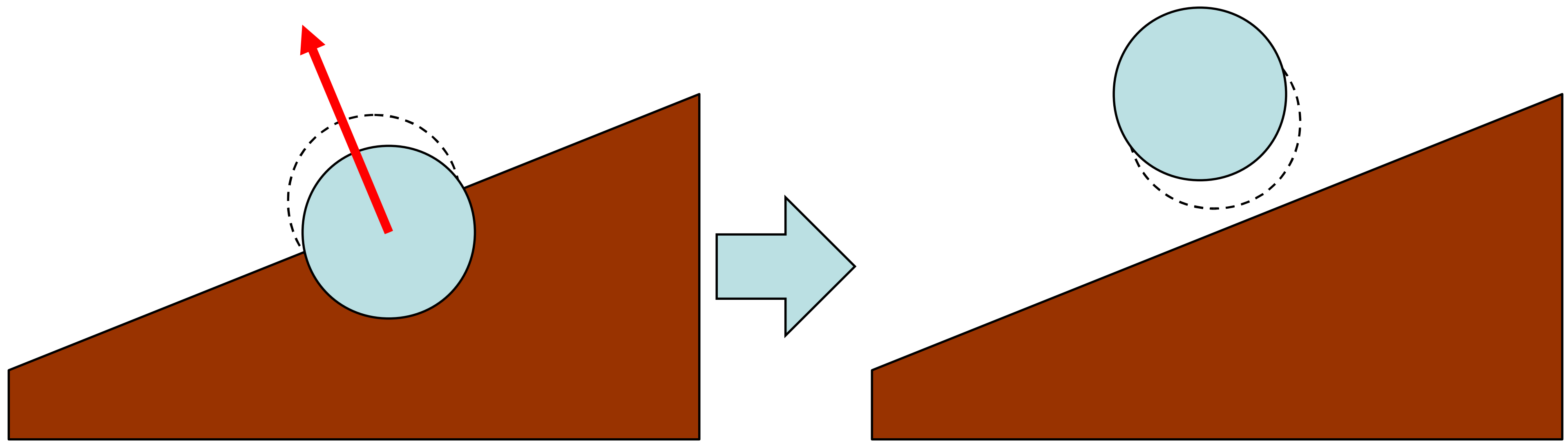
Game Developers Conference®
March 9-13, 2010
Moscone Center
San Francisco, CA
www.GDConf.com

Using distance transform for wall collision



Using distance transform for wall collision

- When detecting collision with the ground
  - The force of repulsion is applied in line with the collision surface
  - Proportional to the collision force



Distance transform in-game

Distance transform algorithms

- $O(n^2)$ Chamfer distance
  - Used with Manhattan distance
  - 1ms for 256x256 on one SPU
  - Also had a 512x512 version
- Dead reckoning
  - A little more accurate
- Jump flooding
  - Implementable on GPU
  - 6ms *GASP*
- Parallel versions exist, but...

Chamfer distance algorithm

- Two passes (forward and back)
- Propagate distance to closest wall
  - Forward pass looks at upper and left neighbors
  - Backwards pass looks at lower and right neighbors
  - The larger the window, the more accurate
  - We went with a 3x3 window

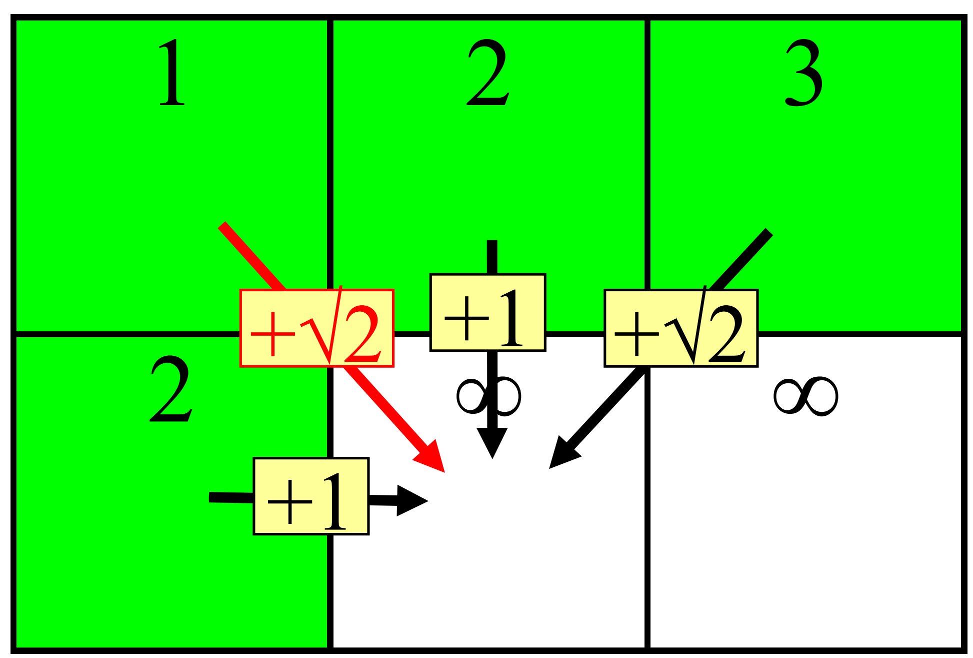
Chamfer distance algorithm (unsigned)

∞	∞	0	∞	∞
∞	∞	0	∞	∞
∞	∞	∞	0	∞

Initial state

2	1	0	1	2
2	1	0	1	2
3	2	1	0	1

Desired result

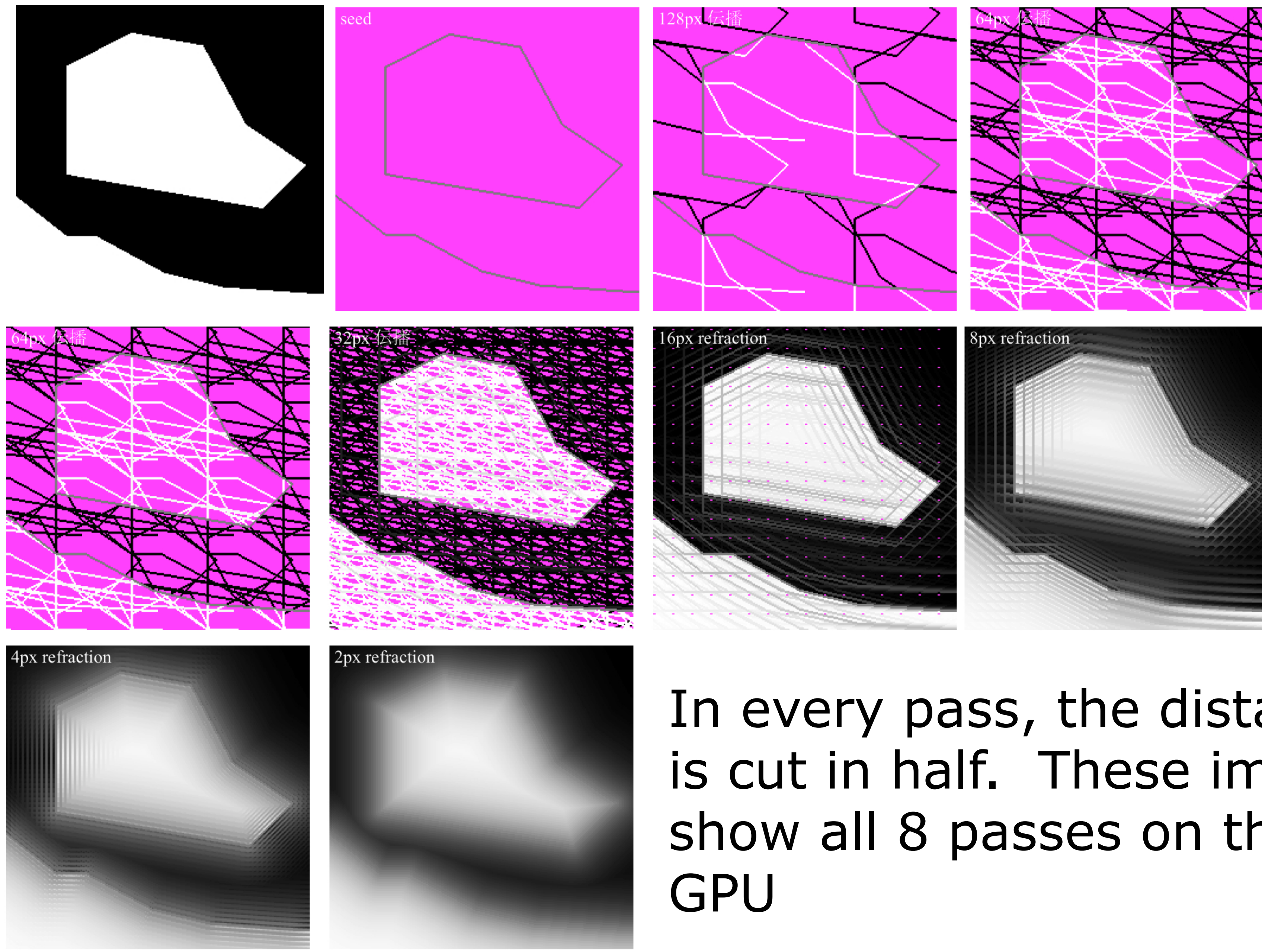


Learn. Network. Inspire.

Jump flooding

- Unlike DRA and CDA*, parallel processing is possible
- Works on GPU
- $\text{Log}_2(n)$ passes
- $O(n^2 \log_2(n))$ calculation
- Rough idea
  - Compute an approximation to the Voronoi diagram of a given set of seeds in a 2D grid

Jump flooding in action



In every pass, the distance is cut in half. These images show all 8 passes on the GPU

Platform comparison

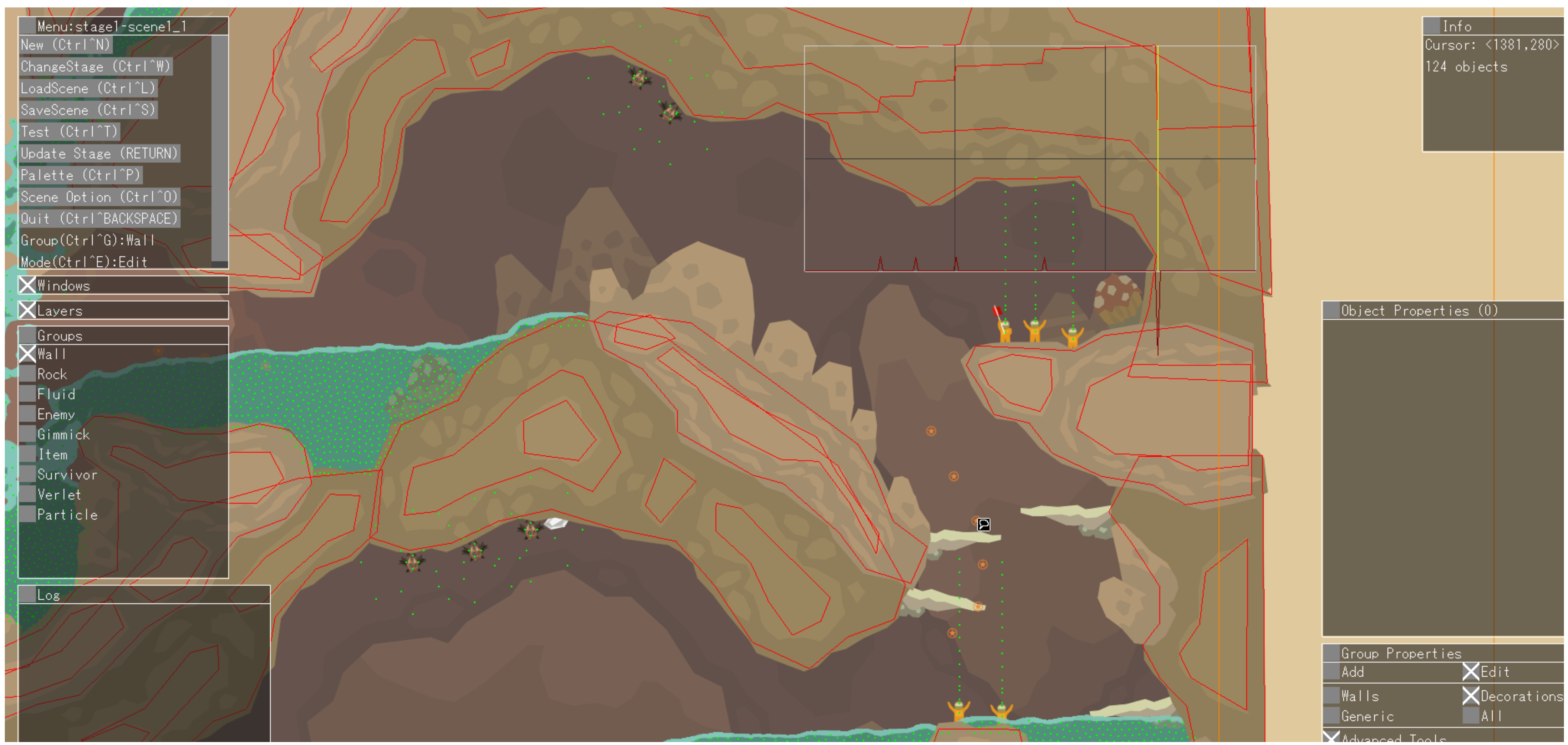
- PS3
  - CDA: 1ms for 256x256 on one SPU
- PC
  - Jump flooding: 8 passes required
  - 5~6ms was too much time, so we split it up and did 4 passes per frame
- SPU vs GPU
  - SPU: more complex processing possible
  - GPU: awkward to program, and is already so busy with rendering

Episode 4
Editor

Editor overview

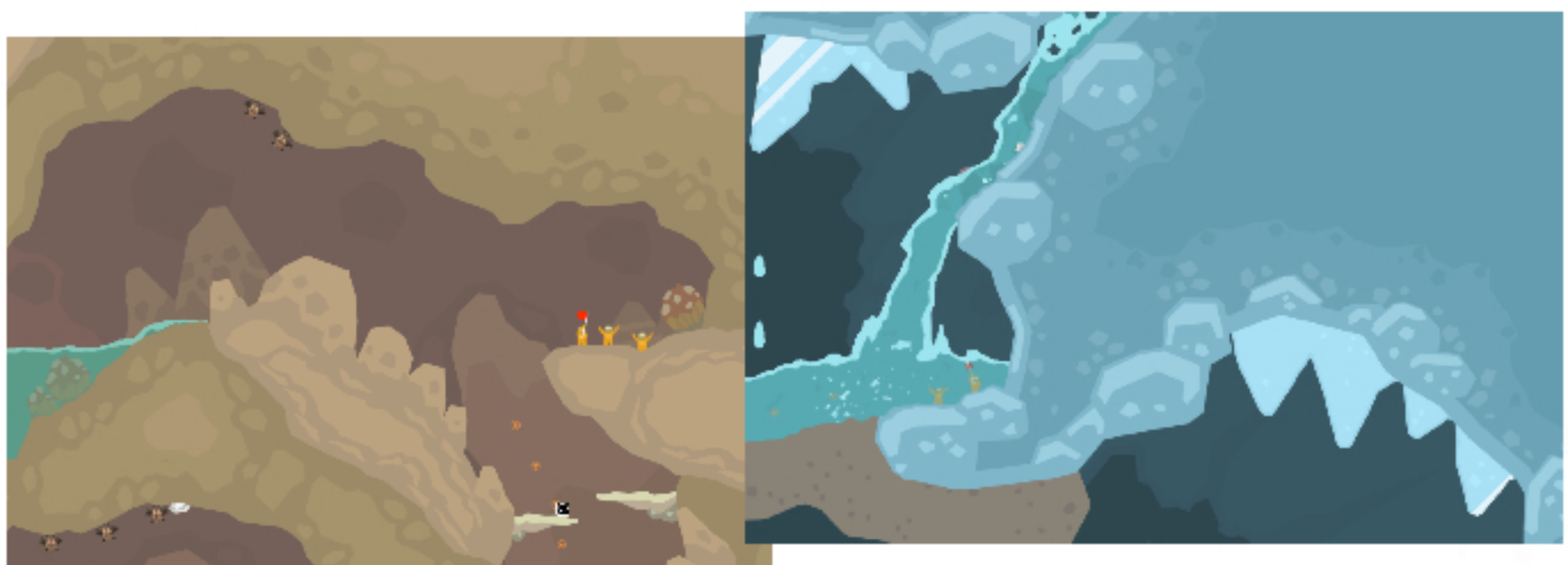
- Functions
  - Wall editing
    - Based on templates
    - Had procedural generation, but didn't use it
  - Placing items, characters, etc
  - Turning things on and off
    - Wall, rock, fluid, enemies, gimmicks, items, survivors, verlet update, particles
  - Fluid editor
    - Flow simulation
    - Execution/cancellation
    - Various debugging visualizations

Editor overview



Topographical design

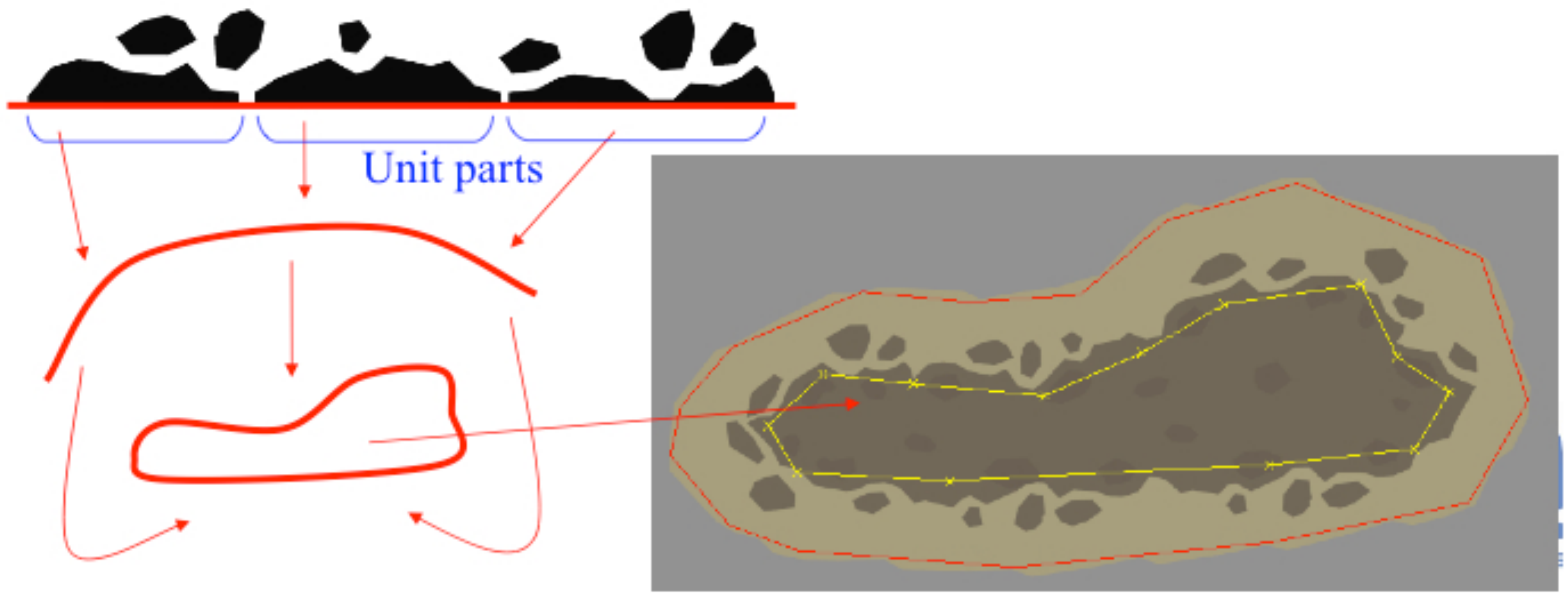
- Different patterns for different things
  - Different sized rocks, walls, ice, snow
- Each stage had unified design concepts
- Designers still have to hand-draw their levels
  - One of the reasons it would be hard to release a level editor on PS3



Game Developers Conference®
March 9-13, 2010
Moscone Center
San Francisco, CA
www.GDConf.com

Pattern templates

- Designers create patterns for wall decorations
- The level creator uses the templates to design the walls
- Templates broken into several parts
- Using randomized loops and reverses, joints are automatically made seamless
- Vector format for nice scaling



GDC 10
Learn. Network. Inspire.
THINK SERVICES

Conclusion

- Fluid simulation system
- 32,768+ fluid particles @ 5SPU, 60FPS
  - Heat transmission, constant distance maintenance, etc
- Universal collision detection system
- Real-time distance field
- CDA, 256×256, Manhattan@1SPU, 1ms
  - Used for collision detection
  - Also abused for ??? in Shooter 2
- Note to self: if time left over, have that “only on PS3” discussion I promised everyone on Twitter