PixelJunk Shooterで使われたPLAYSTATION3の技術

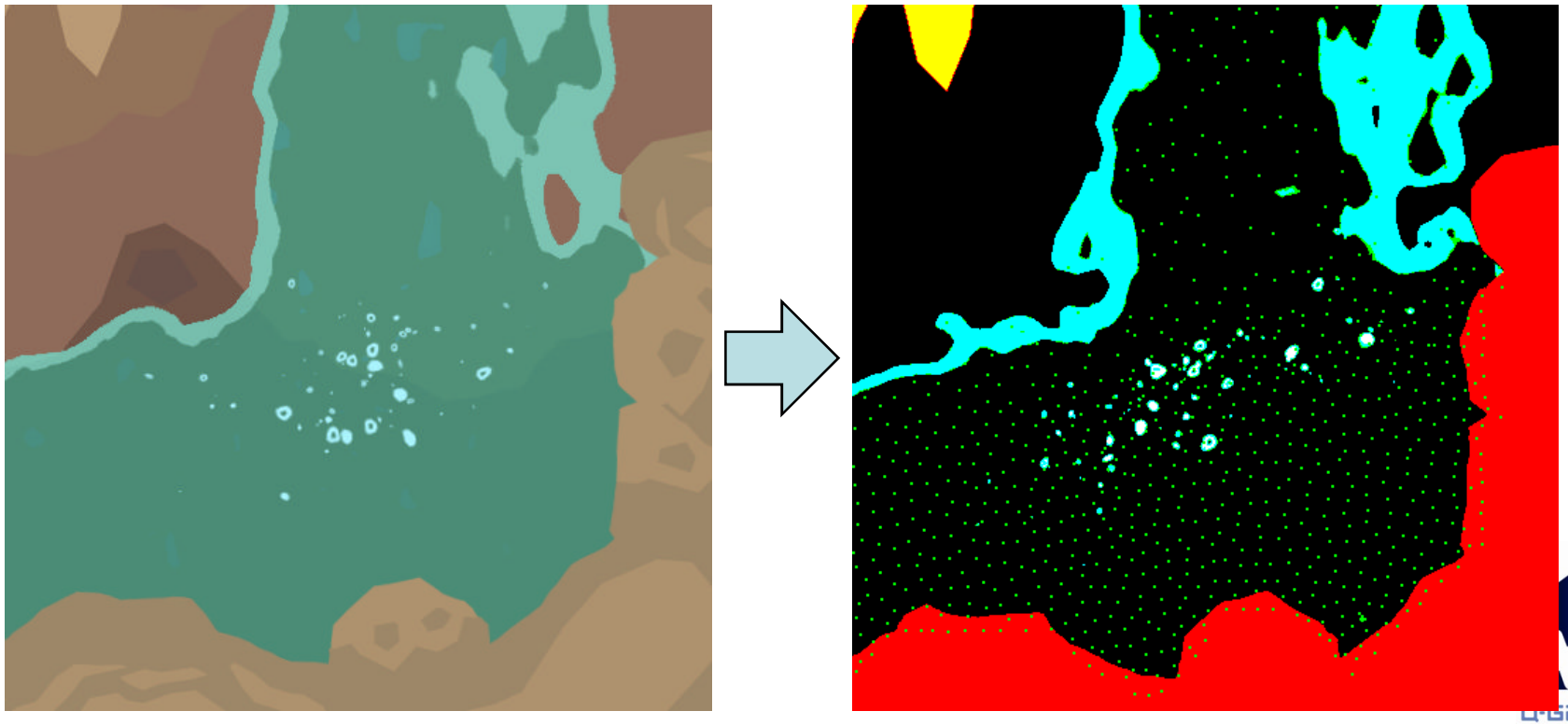
有限会社キュー・ゲームス　プログラマ　木下 直紀

概要

- SPUを用いたパーティクル流体シミュレーション
  - 並列パーティクルシミュレーションアルゴリズム
  - 異種流体の組み合わせによるゲームデザイン
  - 汎用衝突判定器としての利用
  - パーティクル流体レンダリング技術
- Distance fieldを用いたステージ衝突判定
  - リアルタイムSPU distance field計算アルゴリズム
  - リアルタイムGPU distance field計算アルゴリズム
- ステージエディタによるレベル編集事例
  - テンプレートによる地形デザイン

パーティクル流体シミュレーション

- フルイド(流体)をどうやって計算機で表現するか
  - パーティクルの集合体として計算
  - 既存手法 e.g. SPH (Smoothed Particle Hydrodynamics)



流体シミュレーションのゲームへの応用

- 既存手法の特徴(e.g. SPH):
  - 流体力学に基づいた計算モデル
    - Navier-Stokes偏微分方程式, etc.

- 目的:ゲームへの応用
  - 実装・制御の容易さ、高速性
    ＞＞＞＞＞＞物理的正確さ
  - PS3のSPUを使って計算する

- 今回開発した流体システム
  - シンプルな2Dパーティクル衝突シミュレーション
  - 32,768パーティクル@5SPU, 60FPS以上

出典: フリー百科事典『ウィキペディア(Wikipedia)』

ナビエ-ストークス方程式(- ほうていしき、Navier-Stokes e
形偏微分方程式であり、流体力学で用いられる。アンリ・ナビ
かれた。NS方程式とも略される。

しばしば用いられる条件である、非圧縮性流れのニュートン流
式をベクトル形式で表すと、

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \boldsymbol{u} + \boldsymbol{f}$$

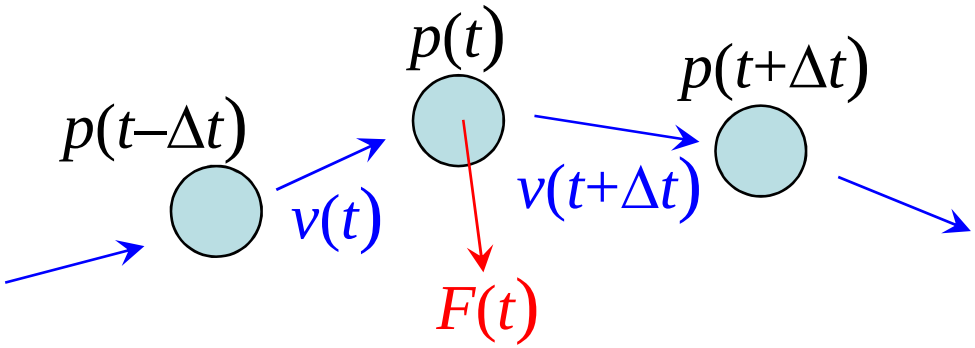
となる。左辺をまとめて $\mathrm{D}\boldsymbol{u}/\mathrm{D}t$ と書くこともある。ここで、時刻

- u - 速度ベクトル
- f - 単位体積当りの流体に加わる外力のベクトル
- ρ - 密度
- p - 圧力
- ν - 動粘性係数

Q-Games

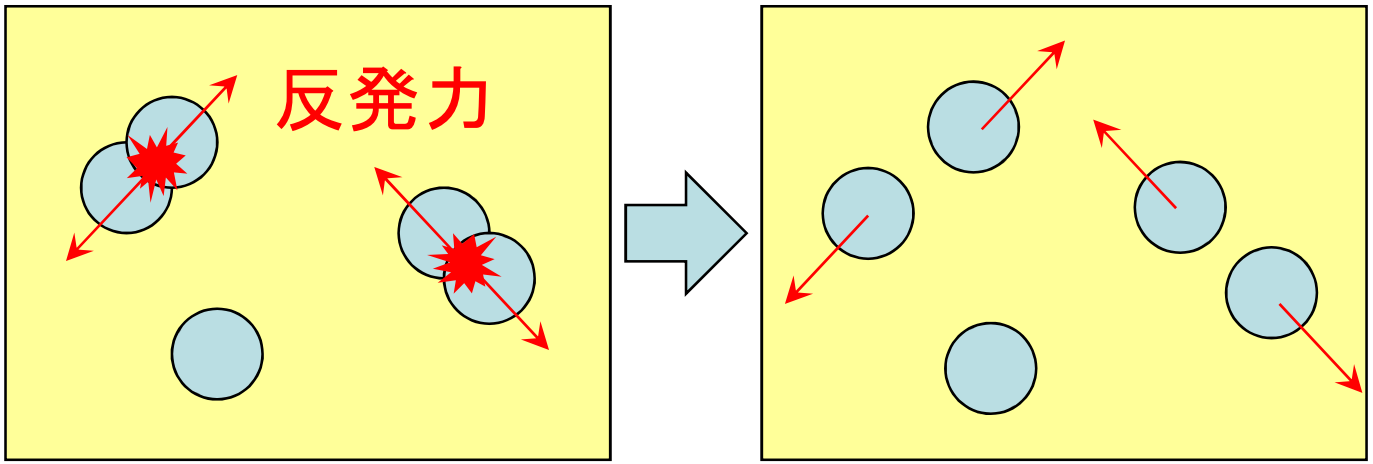
パーティクル単体シミュレーション

- 個々のパーティクルの物理シミュレーションには単純なVerlet integrationを適用
  - 時刻 t におけるパーティクルの位置 $p(t)$,速度 $v(t)$,外力 $F(t)$
  - パーティクルの質量 m
  - シミュレーション間隔 Δt
  - $p(t+\Delta t) = p(t) + v(t)\Delta t + F(t)\Delta t^2 / 2m$
  - $v(t) = (p(t) - p(t-\Delta t)) / \Delta t$



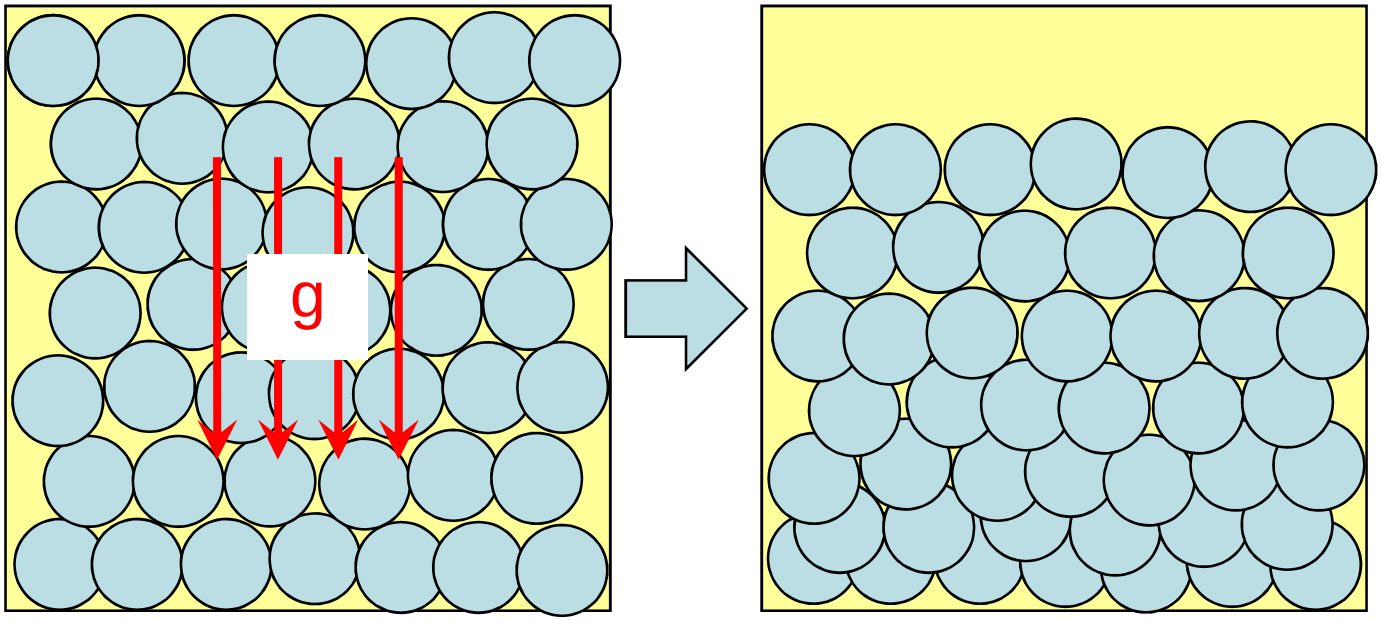
パーティクル衝突シミュレーション

- パーティクル同士が重ならないように、衝突したパーティクルの間に反発力を加える
  - パーティクル数を増やすと流体に近い挙動が得られる
  - 反発力はパーティクルの重なりの度合いに比例
- シンプルな計算モデル
  - パーティクルはすべて球体とみなす
  - 実装・並列化・高速化が容易



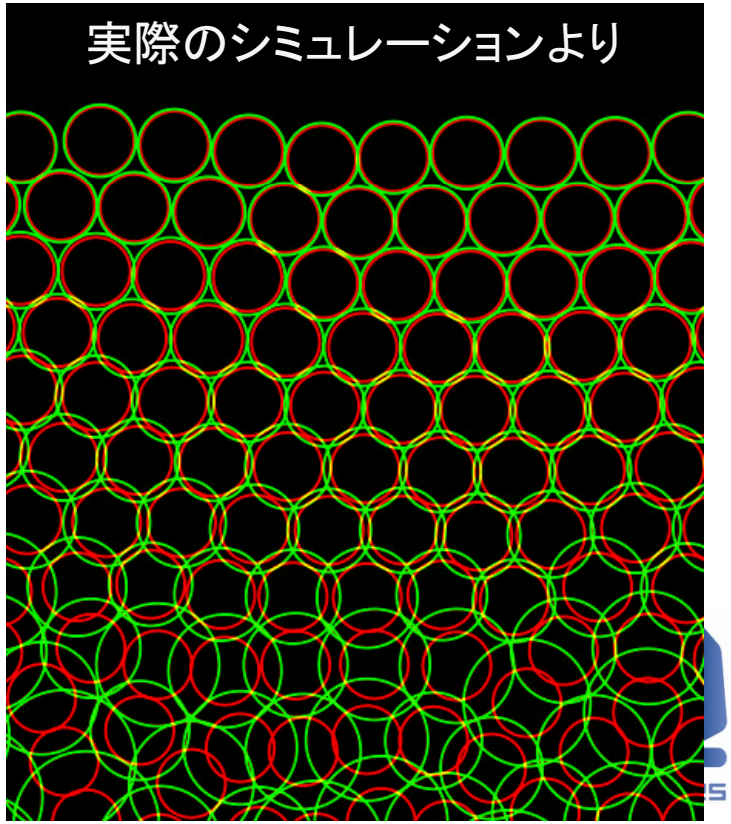
重力下でのパーティクルの挙動

- 液体の性質
  - 非圧縮性：圧力を加えても体積がほとんど不変
- パーティクルに重力を加える
  - 前述の最もシンプルな計算モデルでは、低い位置ほど上からの重力が蓄積され、圧縮されてしまう



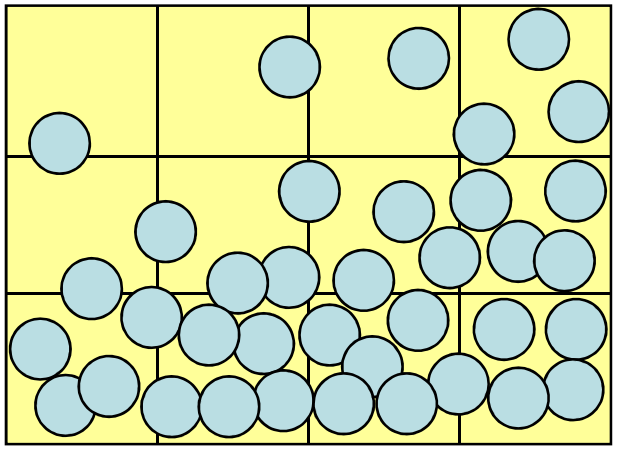
液体の非圧縮性を保証する

- 液体が圧縮されないようにするには
  - 隣接パーティクル間の距離を一定に保つ
  - 圧力の高い場所でパーティクルサイズを仮想的に膨張させる（半径バイアス）
  - 圧力は隣接パーティクルの「めりこみ具合」で測定
- 半径バイアスの更新
  - 毎フレームフィードバックを行う
  - 更新速度の調整が難しい
  - 膨張時より縮小時を速めに設定



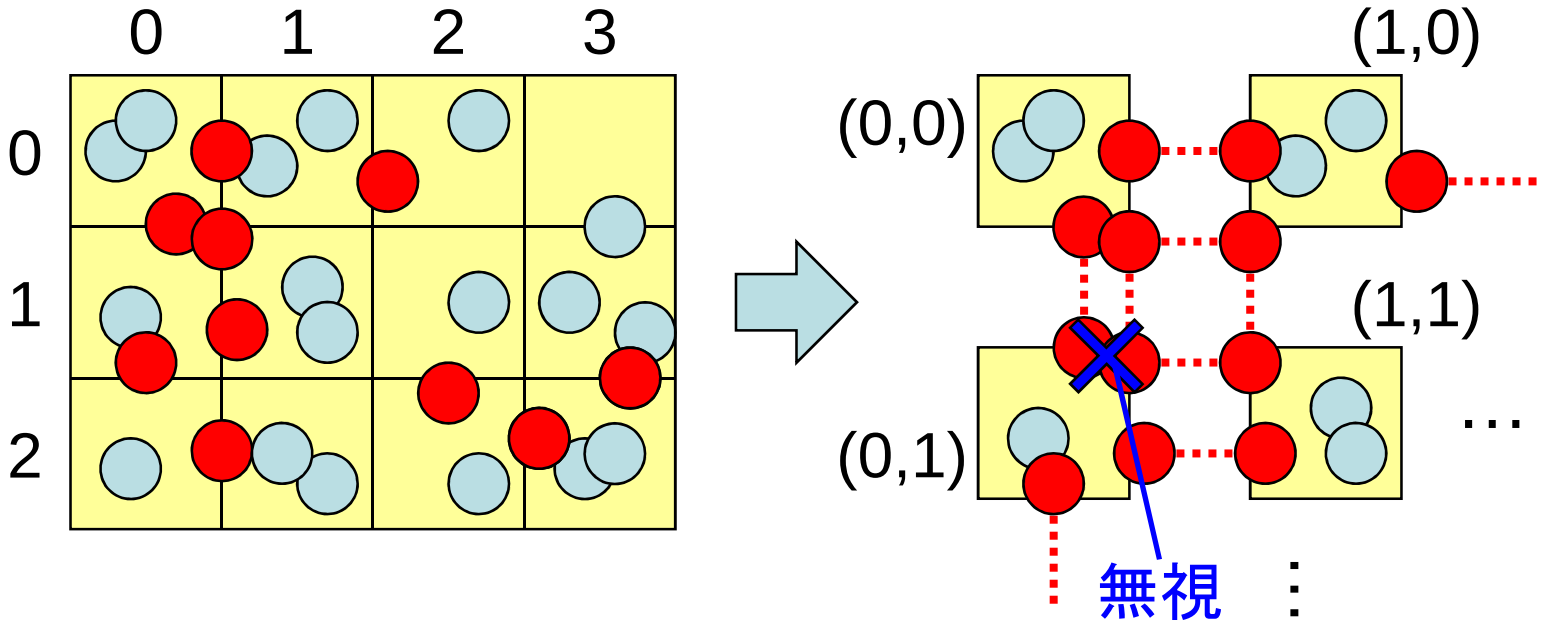
基本アルゴリズム（並列化前）

- アルゴリズムの目的
  - 衝突しているパーティクルのペアをすべて検出して処理
- 自明なアルゴリズム
  - n 個のパーティクルの総当り → 計算時間 $O(n^2)$
  - $n \fallingdotseq 30000$（目標値）
- グリッド分割によるアルゴリズム
  - パーティクルをグリッド上のセルに分割
  - セルごとに総当りで処理
  - k セルに分割 → パーティクルが均等に分布するとすれば、計算時間 $O(n^2/k)$, $k \fallingdotseq 1000$
  - セルごとの処理は並列化が容易



グリッド分割による衝突検出アルゴリズム

- セル境界上にあるパーティクルの扱い
  - 境界両側のセルに含めて処理
  - パーティクルが含まれるセルの範囲を記録しておく
  - 複数のセルに含まれるペアが重複処理されないように、ペアが含まれるセル範囲のうち最も左上のセルでのみ処理



SPUを用いたパーティクル処理の実装

- データ構造
  - パーティクルは配列としてメインメモリに格納
  - SPUからDMAでアクセスして処理
- SPU処理手順（1フレーム分ジョブチェイン）

  Job ①セルごとの衝突処理、結果をメモリに退避

  Job ②各セルで生じた反発力をパーティクルに統合

  Job ③地形との衝突判定処理（後述）、物理計算(Verlet)

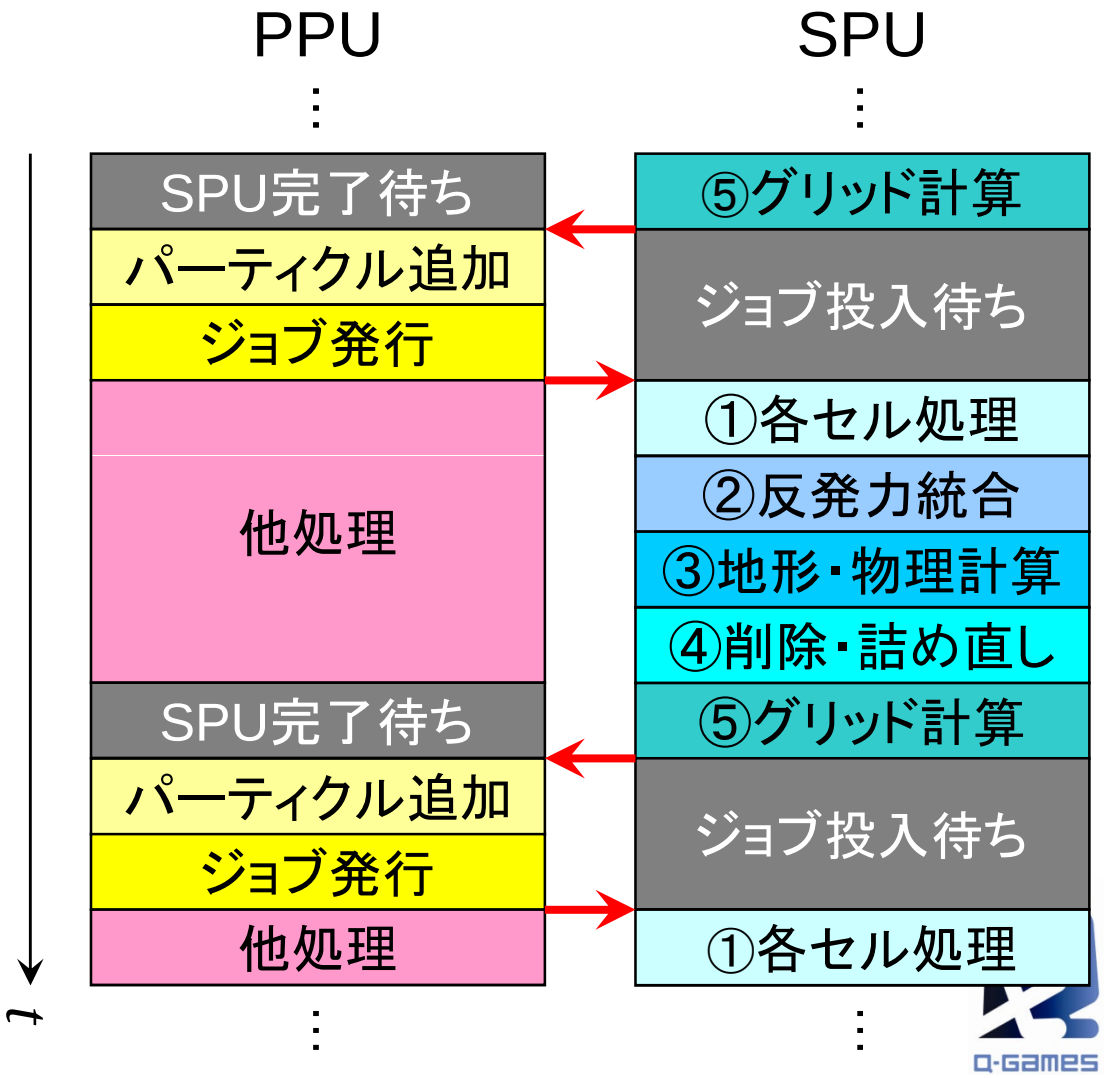
  Job ④不要パーティクルの削除、配列の詰め直し

  Job ⑤次フレームのためのグリッド構築、セルへの分配

  ①～⑤は直列、各ジョブ内で5個のSPUにより並列処理
- PPU処理
  - パーティクルの生成（ゲームシステムから）、ジョブ発行等

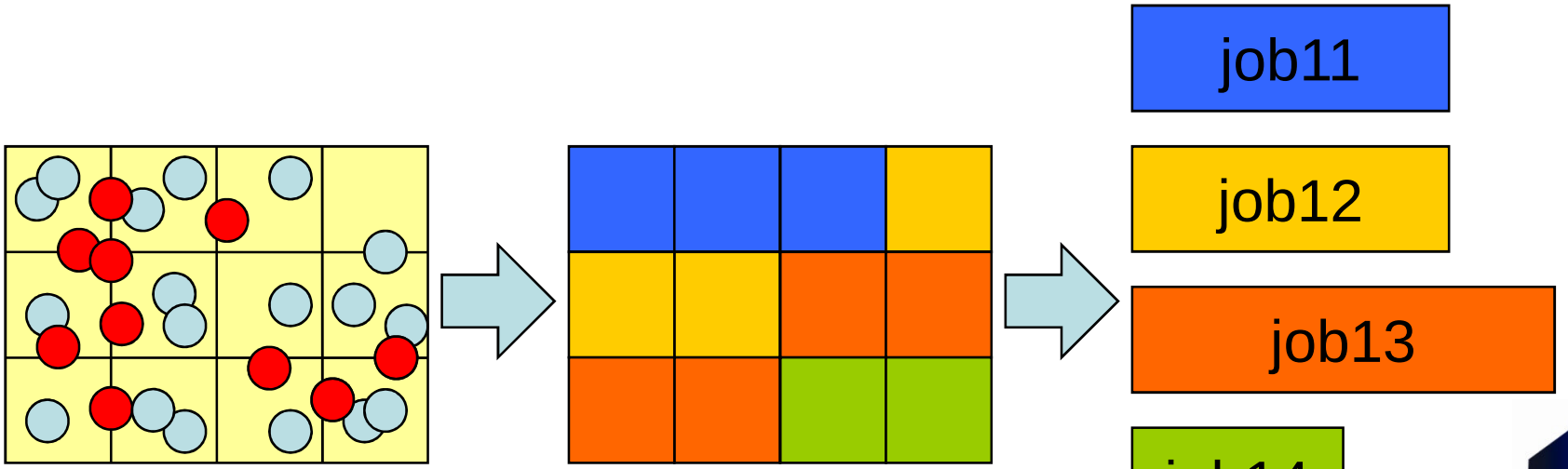
PPU／SPU並列処理

- できるだけ多くの処理をSPUに任せる
- 各SPUジョブもできるだけ複数のSPUに並列化する
- ゲーム側からフルイドシステムへの制御命令（削除等）はコマンドとして蓄積しておき、SPU停止中に同期処理する



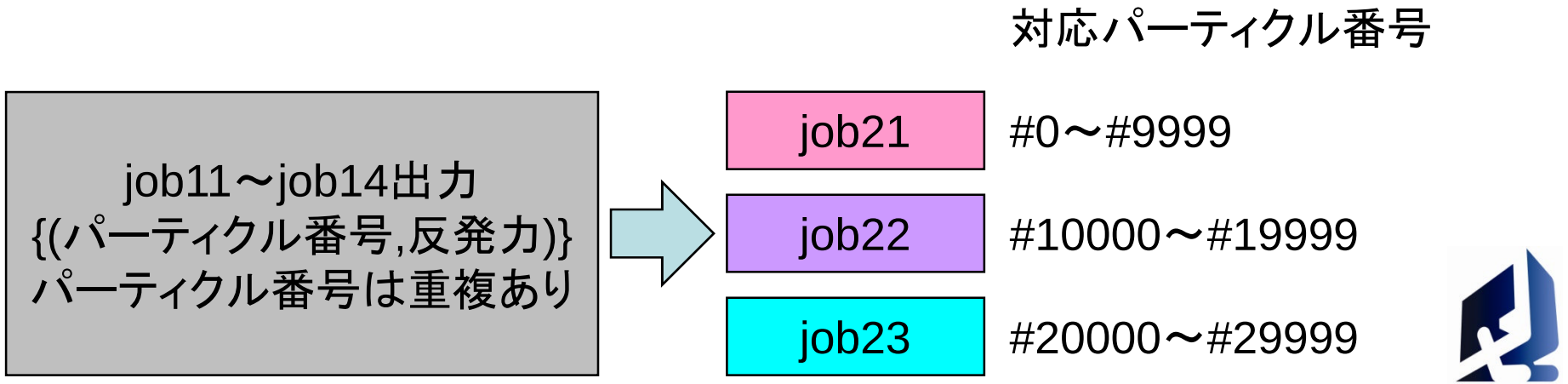
①セルごとの衝突判定

- グリッド上のセルごとにパーティクル同士の衝突を検出し、反発力の計算等の処理を行う
- 複数のセルをまとめて1個のジョブを発行
  - セル内パーティクル数の総和によってジョブを区切る
- 出力：セルごとに {(パーティクル番号,反発力)}



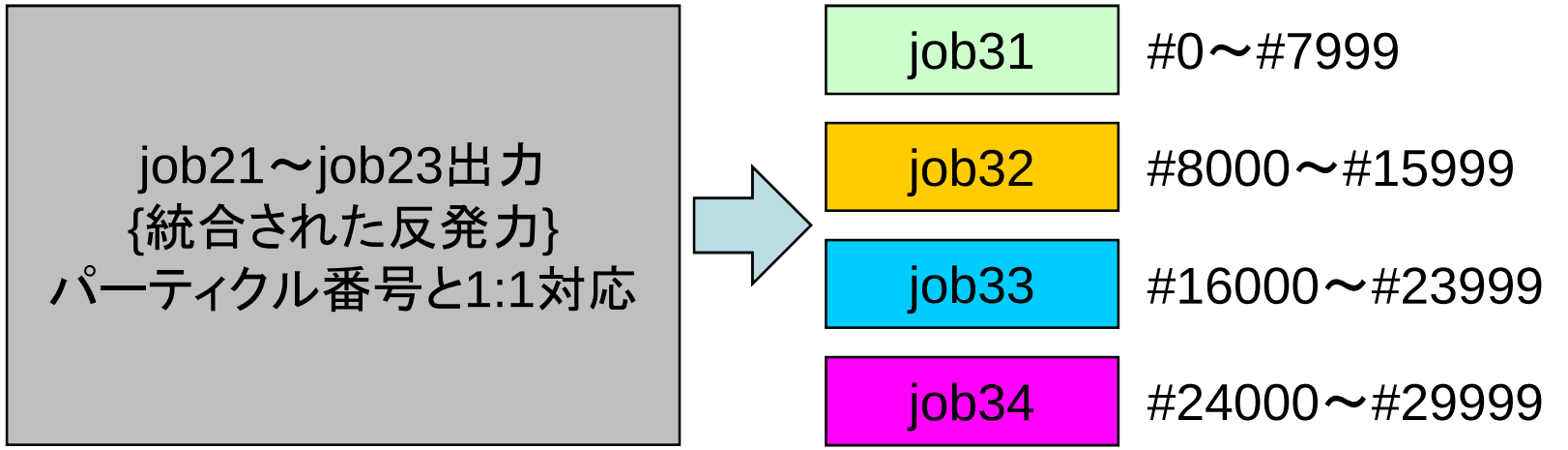
②各セルで生じた反発力をパーティクルに統合

- 1個のパーティクルが複数セルにまたがる場合
  - 各セルで生じた反発力を統合する必要がある
- 前ジョブの結果をパーティクル番号ごとに統合
  - 統合後の加速度を配列としてLSに保持するため、LS容量の制約からパーティクル番号の範囲でジョブを区切る
- 出力：パーティクルごとに統合された反発力



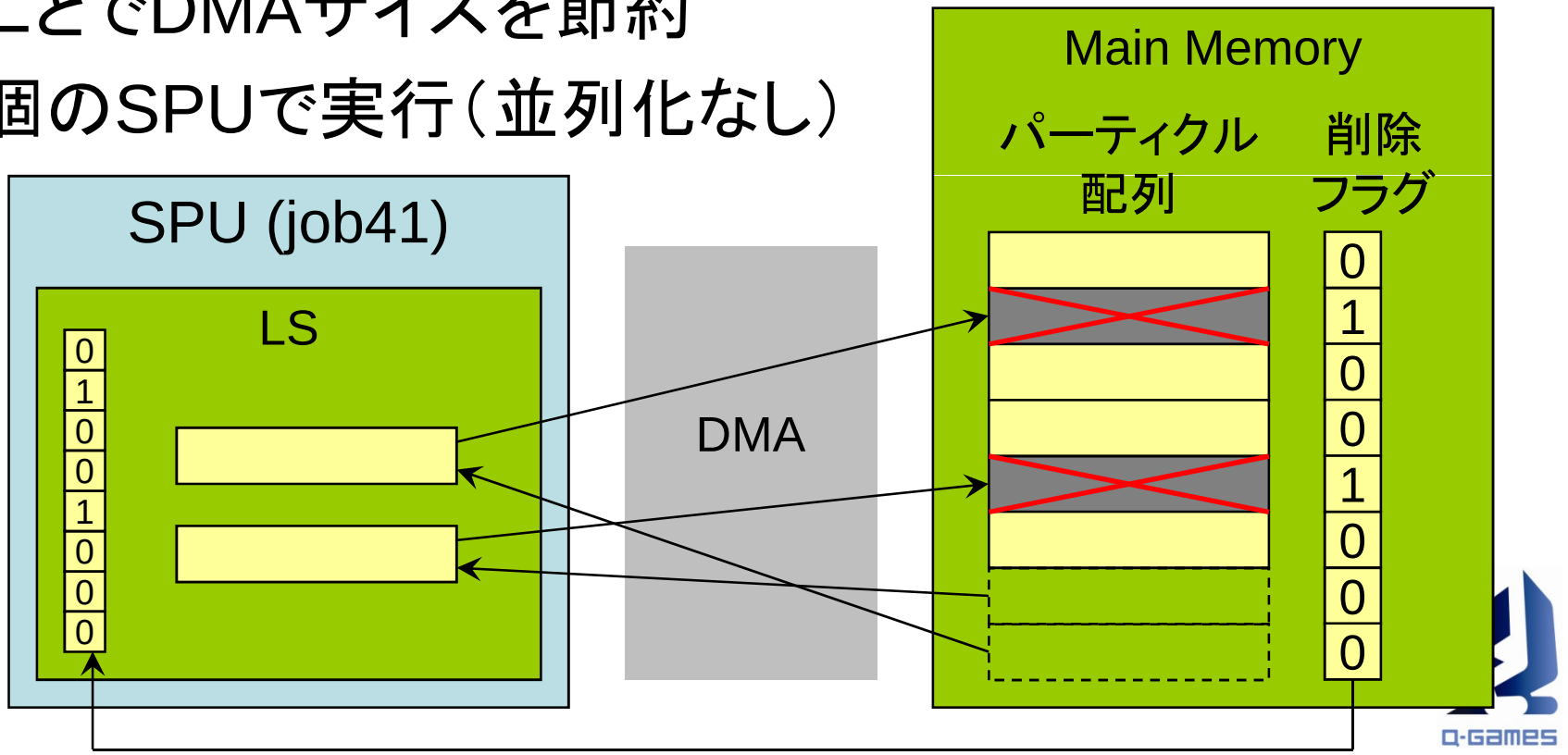
③地形との衝突判定、物理計算

- パーティクル単位のアップデート処理
  - Distance fieldによる地形との衝突判定処理（後述）
  - Verlet integrationによる各パーティクルの物理計算 etc.
  - パーティクル番号の範囲でジョブを区切る
- 出力：更新されたパーティクル情報、不要なパーティクルに対応する削除フラグ配列



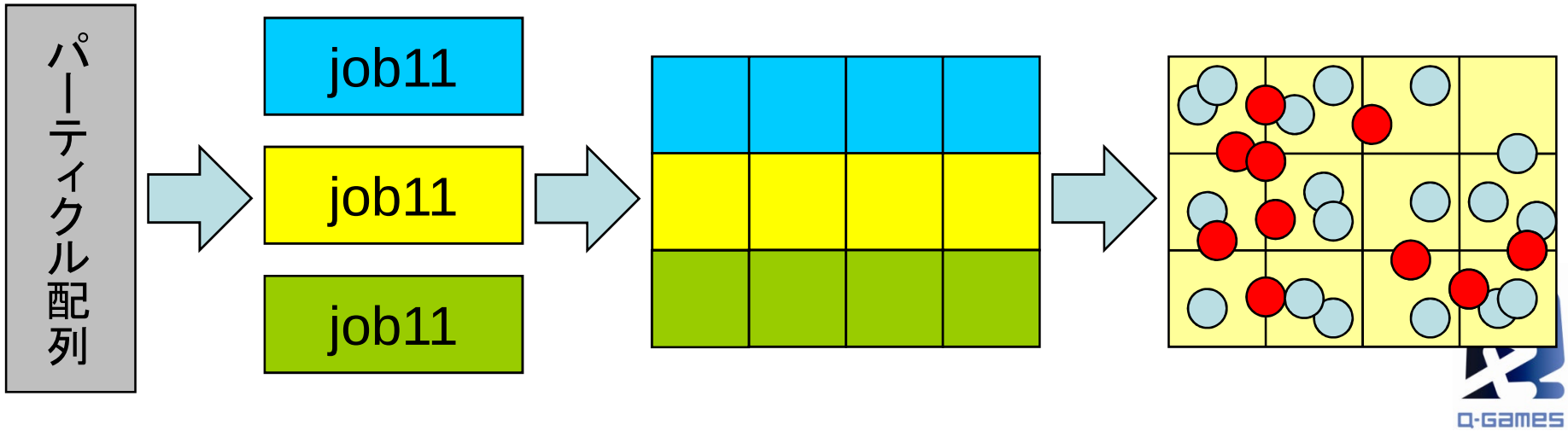
④不要パーティクルの削除、配列の詰め直し

- メインメモリ上ではパーティクルが配列に格納されているので、削除された要素を詰め直す
- 削除フラグをパーティクル配列と別の配列に格納しておくことでDMAサイズを節約
- 1個のSPUで実行（並列化なし）



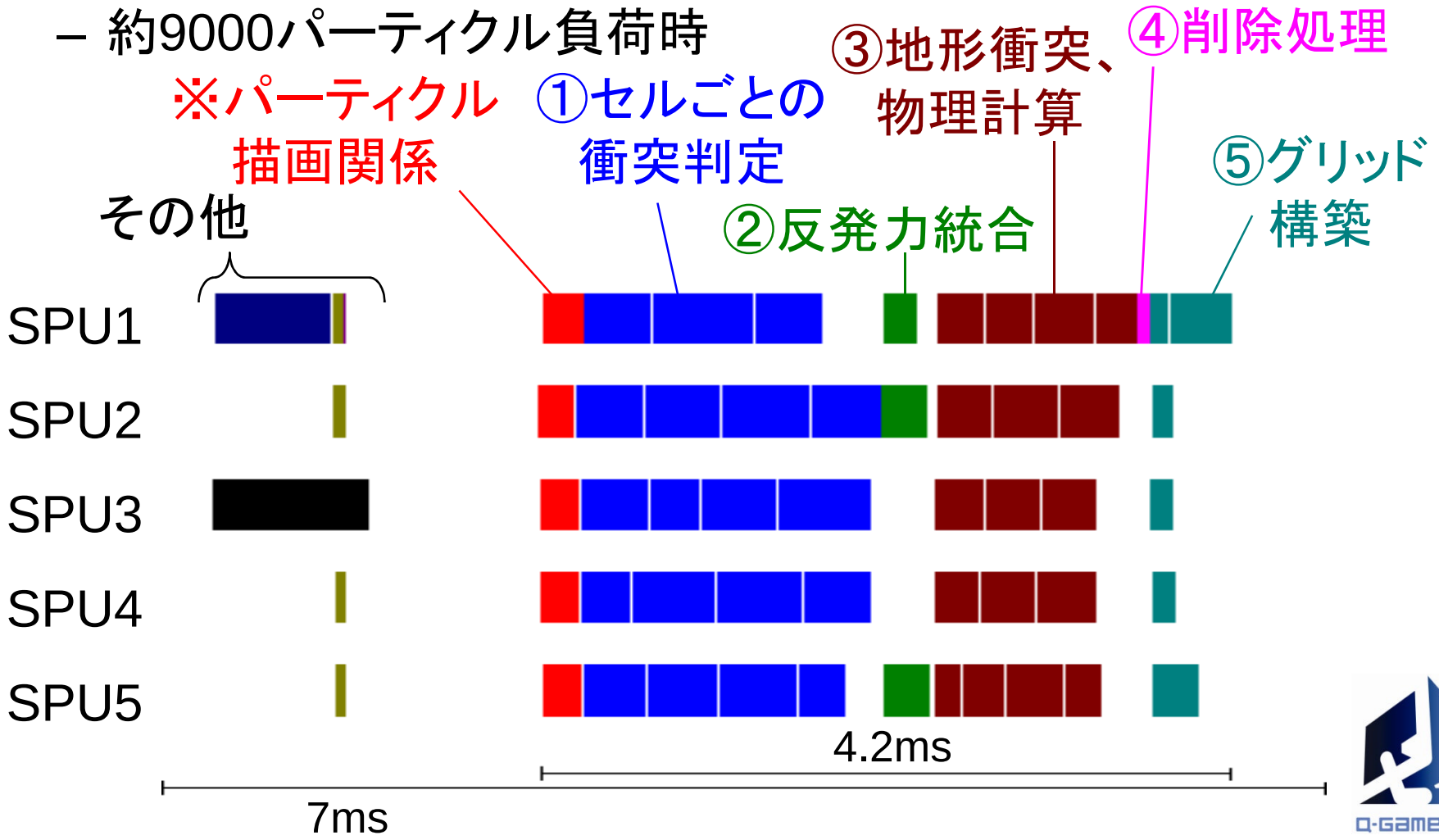
⑤パーティクルをグリッドのセルに分配

- グリッドの領域によってジョブを分割
  - 各ジョブは全パーティクルから、自分の受け持つ領域に含まれるものだけを取り出し、セルに分配する
- グリッドのデータ構造
  - セル要素数配列
  - セル内パーティクルリスト配列（パーティクル番号を格納）



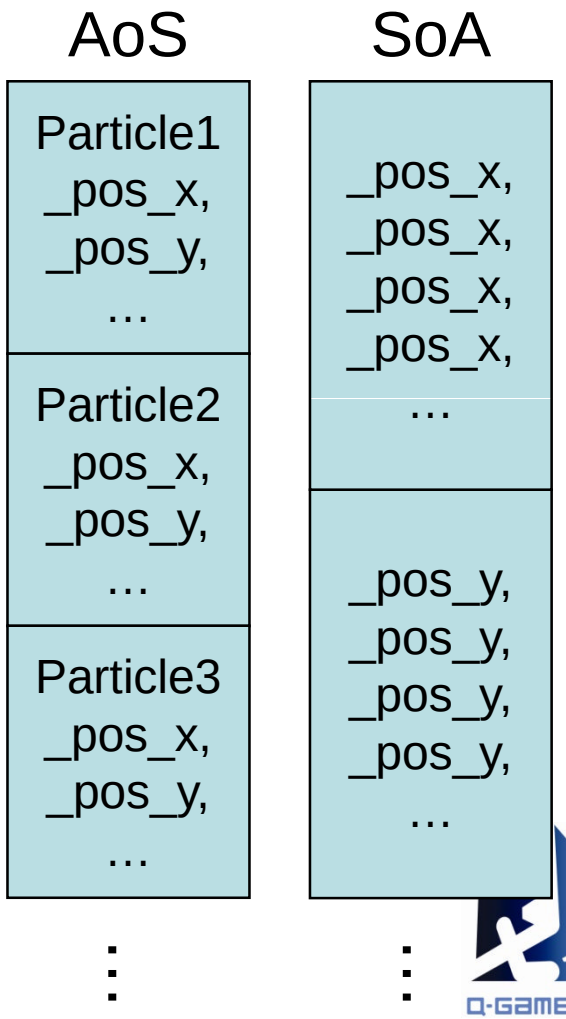
実際のSPU使用状況

- SPUパケット監視による計測結果（1フレーム分）
  - 約9000パーティクル負荷時



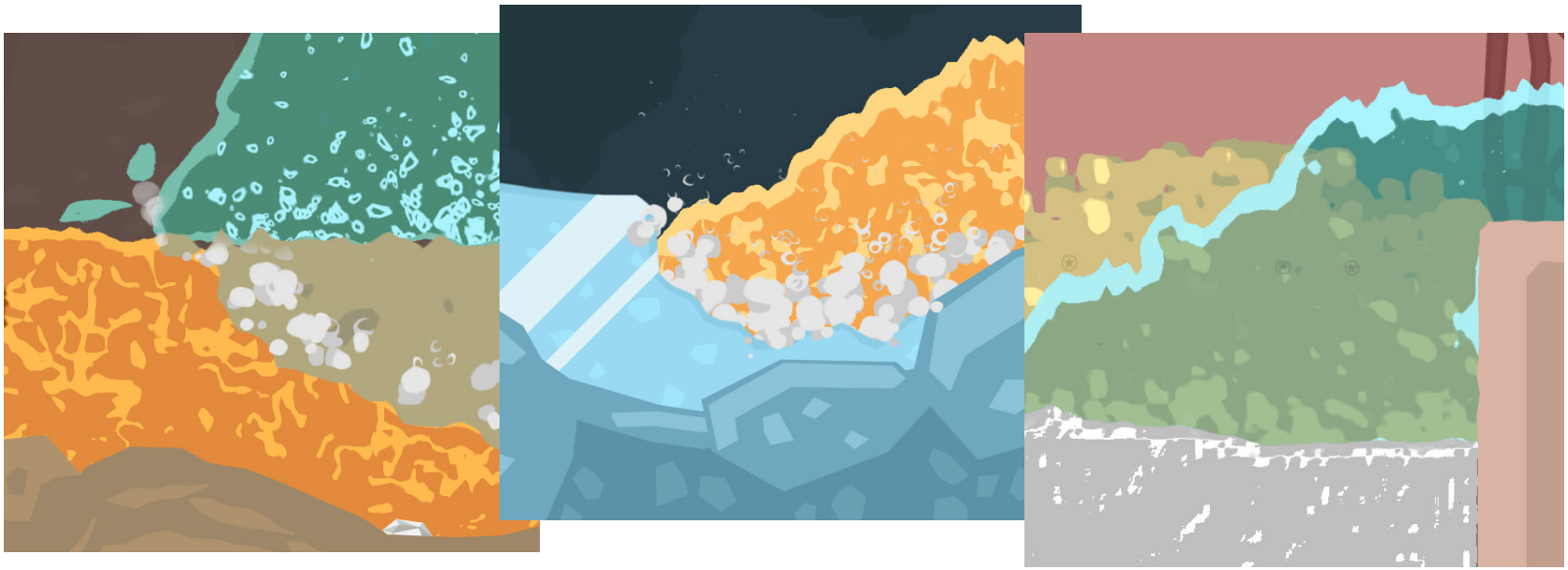
SPUコード最適化

- AoS (Array of Structure)よりSoA (Structure of Array)
  - e.g. パーティクルデータの場合：
  - X座標,Y座標等の最小のデータ単位でそれぞれ配列を構成する
  - SPUのベクトル命令を用いて、複数のパーティクルを同時に処理できる
  - SPUはスカラー処理が苦手！
- 分岐命令削減、パイプライン効率
  - Cell：分岐予測なし
  - Cell：アウトオブオーダー実行なし
  - ループアンロール
  - ソフトウェアパイプライン



異種流体の組み合わせ

- 多種の固体・液体・気体を組み合せたゲームデザイン
  - マグマに水をかけると冷却されて個体の溶岩になる
  - 氷にマグマをかけると溶ける
  - 液体金属に水をかけると毒ガスが発生する etc.
- パーティクル間の反発力以外の相互作用が必要

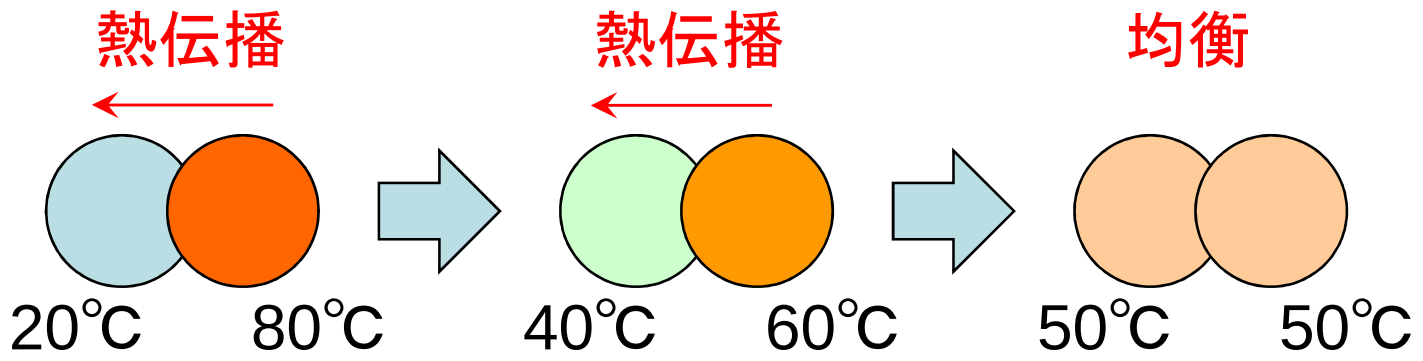


パーティクル衝突検出時の処理

- 反発力を発生させる
  - パーティクルが重ならないようにして非圧縮性を保つ
  - タイプのペアごとに反発力の大きさを設定→比重の概念
- 他にも・・・
- パーティクルの熱を伝播させる
  - 応用：マグマが冷却されると固体の溶岩に変化する、等
- 接触している他パーティクルの種類を記録
  - 応用：液体混合時の化学反応シミュレーション等
- 特定タイプのパーティクルペアの衝突をPPUに通知
  - 応用：衝突検出用の特殊パーティクルを使用して、キャラクターと液体の接触判定を行う、等
  - 汎用衝突検出器としてのパーティクルシステムの利用

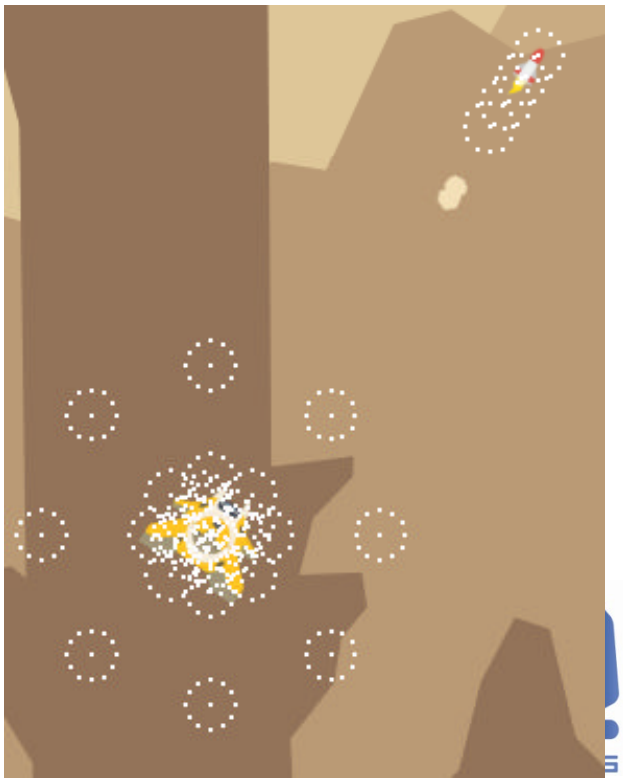
流体の熱伝播システム

- パーティクルごとに温度情報を持たせる
  - 温度によって流体の挙動を変える等ゲームでの応用が可能
- パーティクル衝突時の熱伝播
  - 温度の高いパーティクルから温度の低いパーティクルへ熱を移動させる
  - 反発力の計算と同じアルゴリズムで各セルの熱伝導を統合
  - パーティクルの種類ごとに熱伝導率を設定



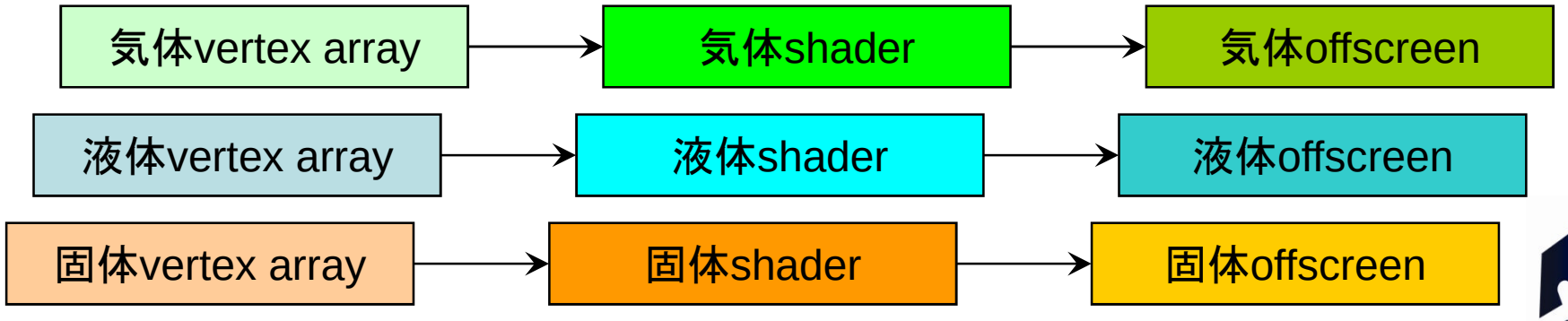
汎用衝突判定器としてのパーティクル流体システム

- PixelJunk Shooterでは、ゲーム中のほとんどの衝突判定処理にパーティクル流体システムを応用
  - キャラクターやミサイル等に衝突判定専用のダミーパーティクル（インタラクタ）をくっつける
  - インタラクタとの衝突はSPUでの衝突判定処理時にPPU側に通知される
- メリット
  - 衝突判定処理を個別に実装する必要が無い
  - インタラクタパーティクルの配置により、どんな形状の衝突判定も容易に実現可能



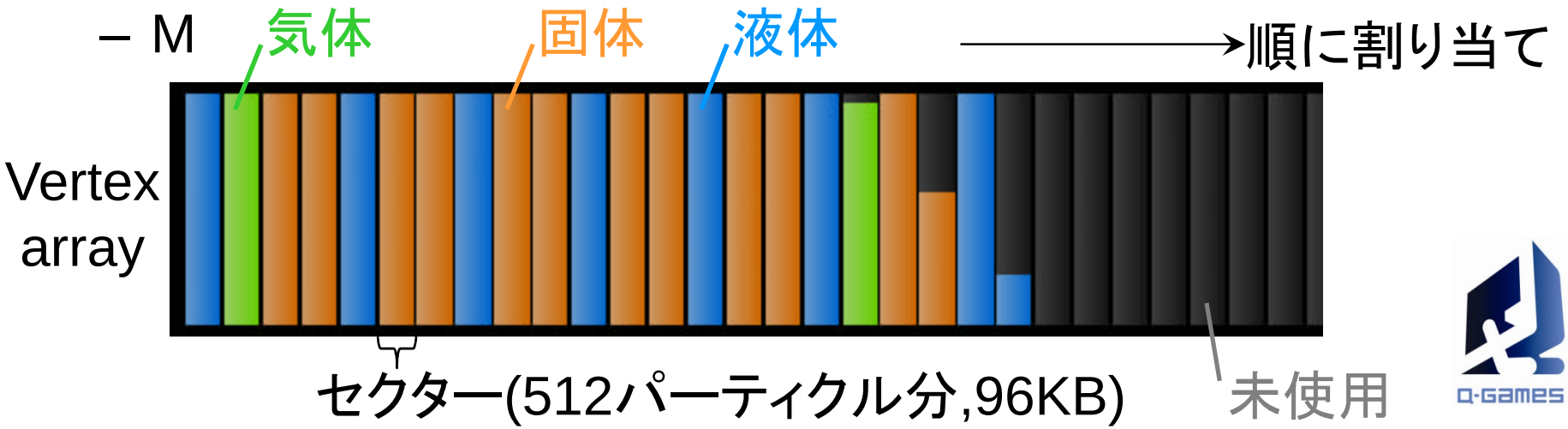
パーティクルタイプごとのvertex array構築

- パーティクル描画時に必要なvertex array
  - 大きく分けて固体、液体、気体の3種類
  - それぞれ異なるオフスクリーンバッファに描画
  - 種類ごとに異なるvertex arrayを用意する必要あり
- パーティクル数上限：約30000
  - 3種類、各30000個分のvertex arrayを確保するのは無駄
  - ひとつのvertex arrayを状況に応じて共有する



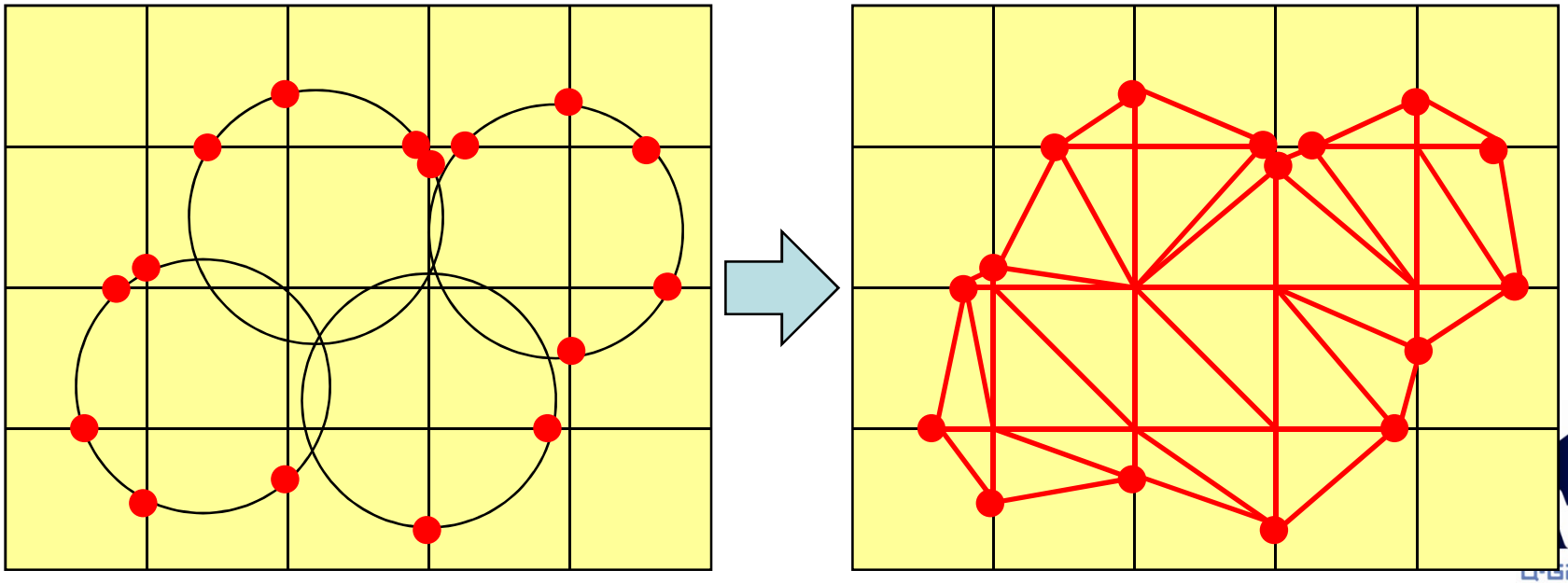
Vertex arrayの共有

- パーティクルvertex array構築
  - パーティクル番号の範囲で5個のSPUジョブに分割
  - SPU内で数十KB単位でvertex arrayを構築し、DMA転送
- Vertex arrayを64個のセクターに分割
  - SPUからメインメモリへの転送時に、パーティクルの種類（固体、液体、気体）ごとにセクターを割り当て
  - セクター割り当て処理はatomic DMAで排他制御
  - M



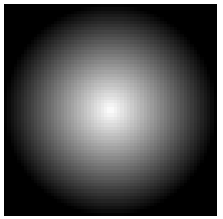
液体のレンダリング

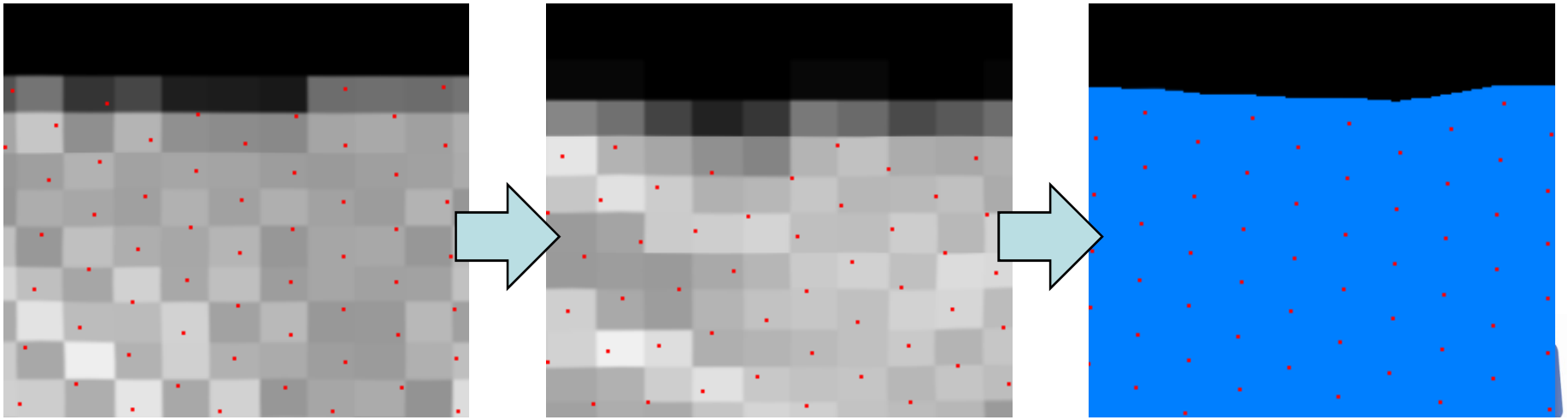
- パーティクルの集合をなめらかな流体として描画
- 既存手法の例：Marching square/cube
  - パーティクル球体境界とグリッドとの交点を検出、連結して、流体をポリゴンメッシュとして表現する手法
  - グリッドを細かくしないとなめらかに見えない。特許（終了？）



画像処理ベースのレンダリング

- レンダリング手順
  - 低解像度のオフスクリーンバッファに、中心が明るくなっているテクスチャ用いてパーティクルを描画
  - オフスクリーンバッファにブラーをかける
  - バイリニアフィルタを用いてスクリーン解像度まで拡大
  - 一定値以上の明るさの部分のみ抽出して液体色を描画

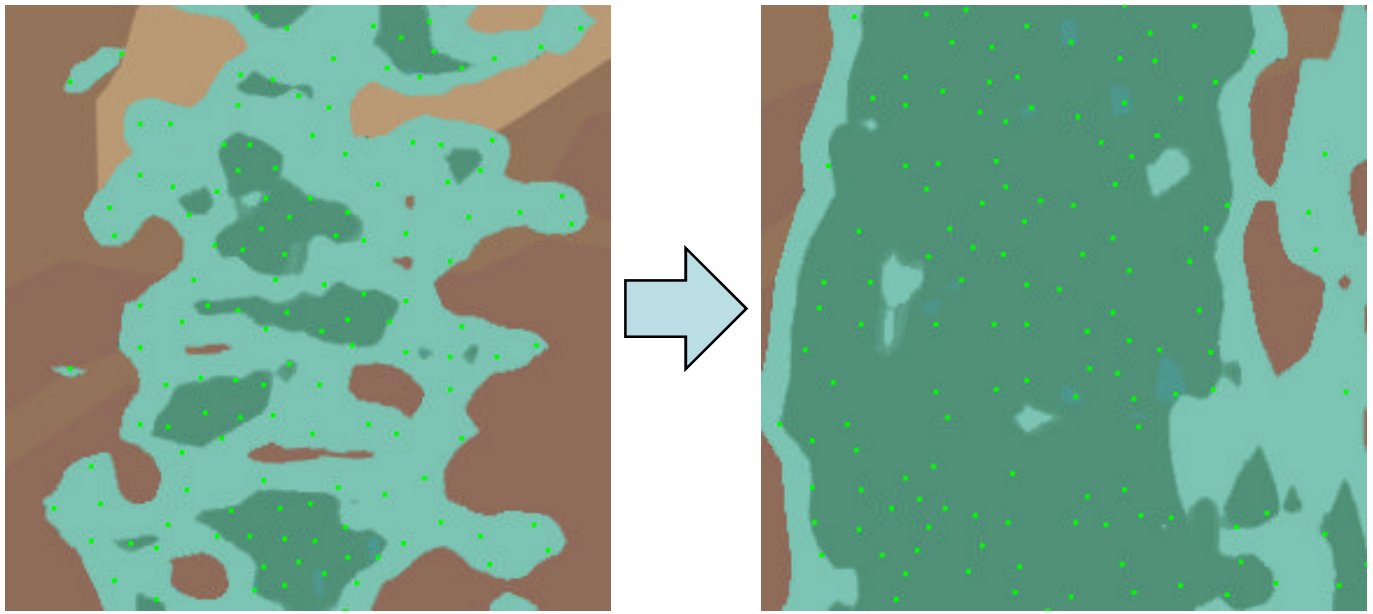




低解像度オフスクリーン　　ブラー適用後　　バイリニア拡大＋ステップ

液体の残像効果

- 液体が自由落下する場面など
  - パーティクル間の距離が広くなり、液体の連続感が失われる
- レンダリング時、オフスクリーンバッファを完全にクリアせず、少しずつ減衰させる
  - 残像効果によって動きのある場所でも連続感が得られる



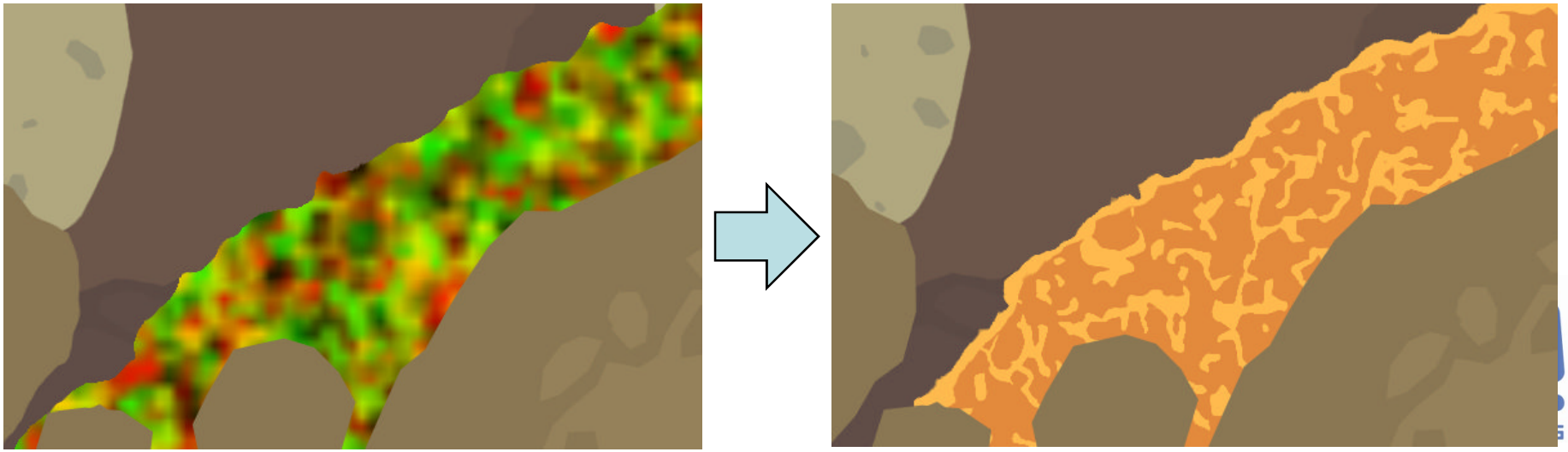
液体レンダリングのブラシアップ

- 水面の検出、アンチエイリアシング
  - オフスクリーンバッファから液体をレンダリングする際にスムーズステップ関数を使用→アンチエイリアシング
  - 異なる2段階の閾値を用いて水面を検出、着色
- 激しい動きの検出
  - 液体パーティクルの速度・方向の急激な変化を検出（SPUアップデートジョブ）
  - イベントとしてPPUに通知
  - 気泡などのエフェクトを発生

「流れ」の表現　液体の動きを可視化する

- 定常的な液体の流れ
  - 液体とともに流れるものがないと「流れ」が見えない
- 液体に模様をつける
  - パーティクルごとに固有のランダムなUV値を付与
  - 別のオフスクリーンに固有UV値をRG色として描画
  - RG=(0.5,0.5)に近い部分を模様として別色で描画

0.0～1.0の2次元実数値

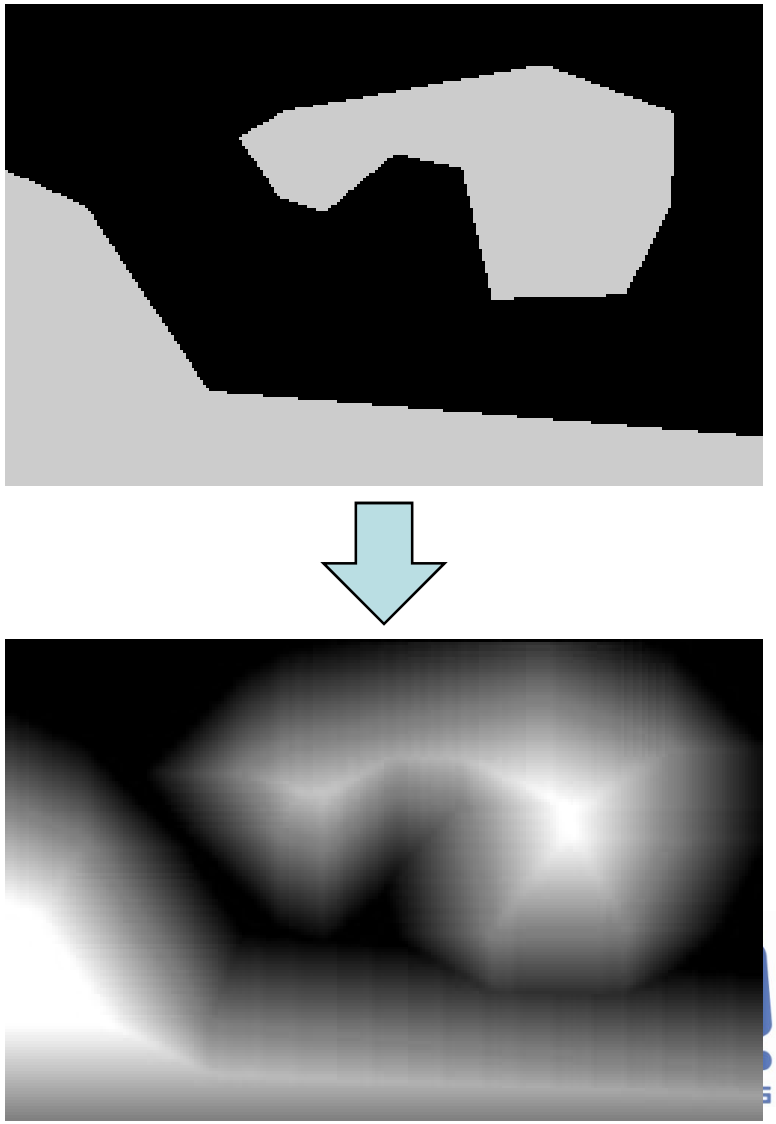


かげろう･屈折の表現

- 水中やマグマの熱気による屈折表現
  - シーンを一旦すべてオフスクリーンバッファに描画
  - スクリーンへの描画時に、水中やマグマの近傍を歪ませる
  - 歪みの量や方向は、液体パーティクルに付与された固有UV値を描画したオフスクリーンを参照して決定する
  - テクスチャのフィードバック処理により、熱気の伝播をシミュレート

Distance field

- Distance fieldとは
  - 入力画像(binary image)
    - 壁：白
    - 空間：黒
  - 出力画像(distance field)
    - 壁にめり込んでいるところ：明
    - 壁の境界線：中間色(0.5)
    - 壁から遠いところ：暗
- Distance fieldの応用例
  - フォント等の拡大画像改善
  - 地形等の衝突判定処理

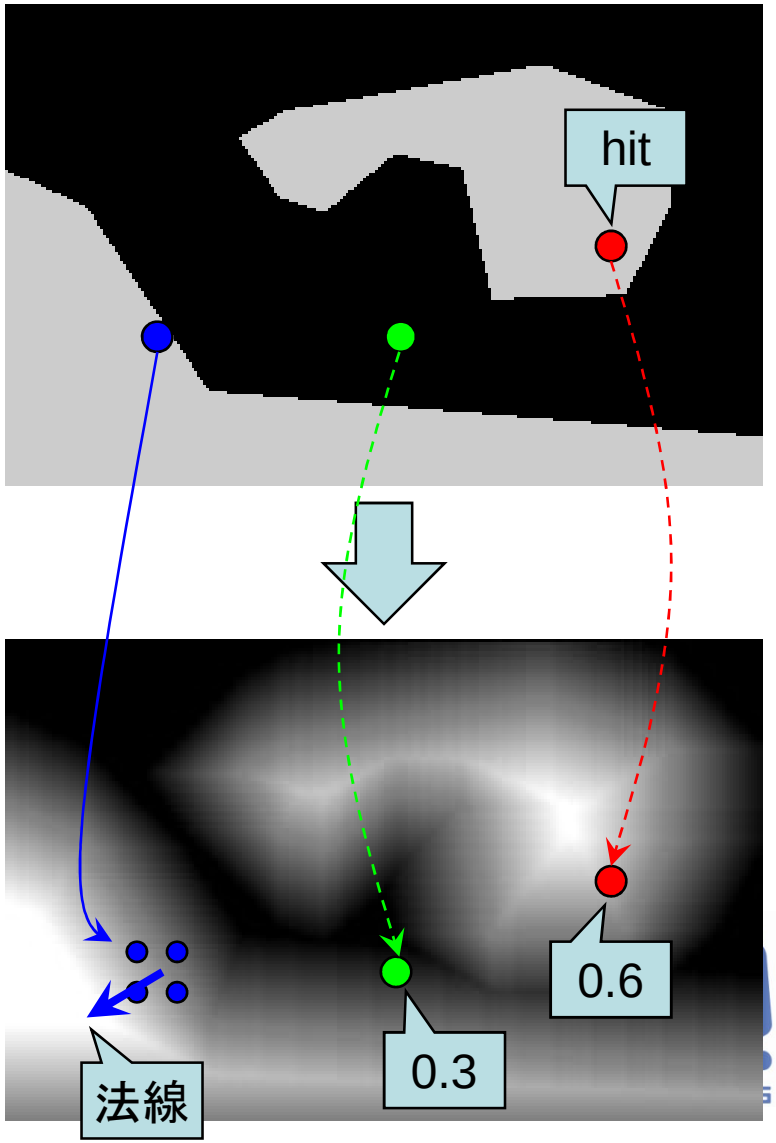


Distance fieldによる地形衝突判定

- 地形(壁,障害物)衝突判定
  - キャラクターの位置に対応するdistance field画素値 v を取得

    $v > 0.5 \rightarrow$ 壁に衝突している

    $v \leqq 0.5 \rightarrow$ 衝突していない
- 衝突位置の法線計算
  - 衝突地点近傍4点のdistance field画素値を取得
  - 画素値の増加する方向(gradient)＝壁に近づく方向



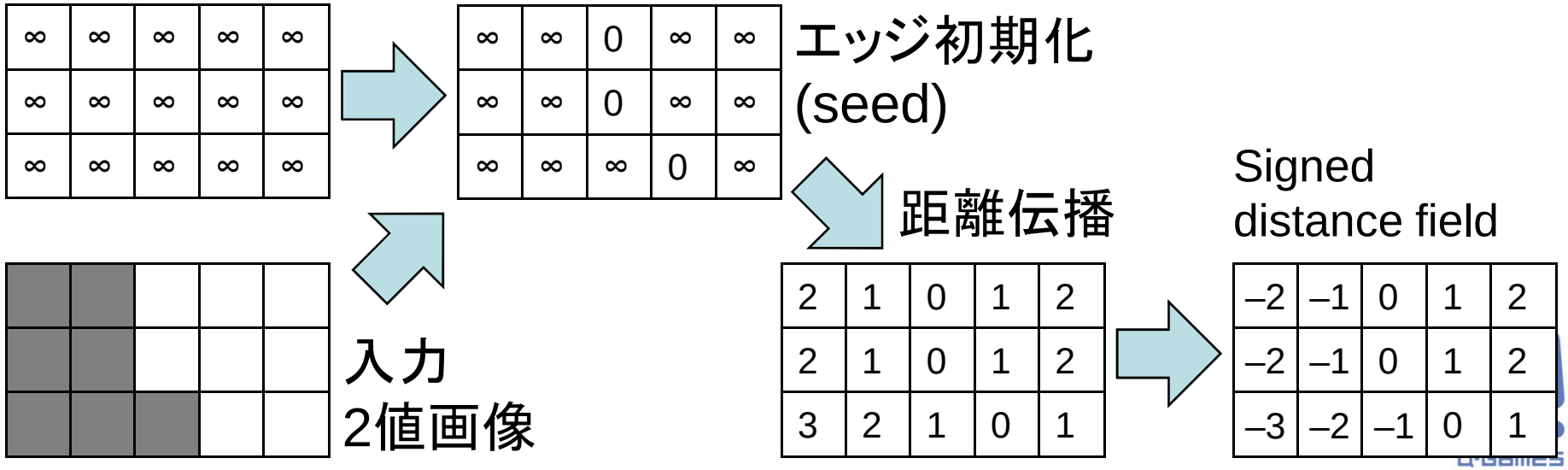
Distance fieldの計算アルゴリズム

- 逐次計算型
  - Chamfer distance algorithm (CDA)
    - マンハッタン距離ならこれで十分
  - Dead reckoning algorithm (DRA)
    - CDAに若干の改良を加えたもの
    - CDAと同じ計算量でもより正確（メモリを多く消費する）
    - G. J. Grevera. The “Dead Reckoning” Signed Distance Transform. J. CVIU 95(3), pp.317–333, 2004.
- 並列計算型
  - Jump flooding algorithm
    - GPUでの実装に適する
    - G. Rong and T. S. Tan. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. In Proc. of ACM i3D, 2006.
- いずれも計算結果は厳密ではない（誤差を生ずる）

PixelJunk Shooterで使用したアルゴリズムはマンハッタン距離CDAのSPU実装 256×256, 1ms

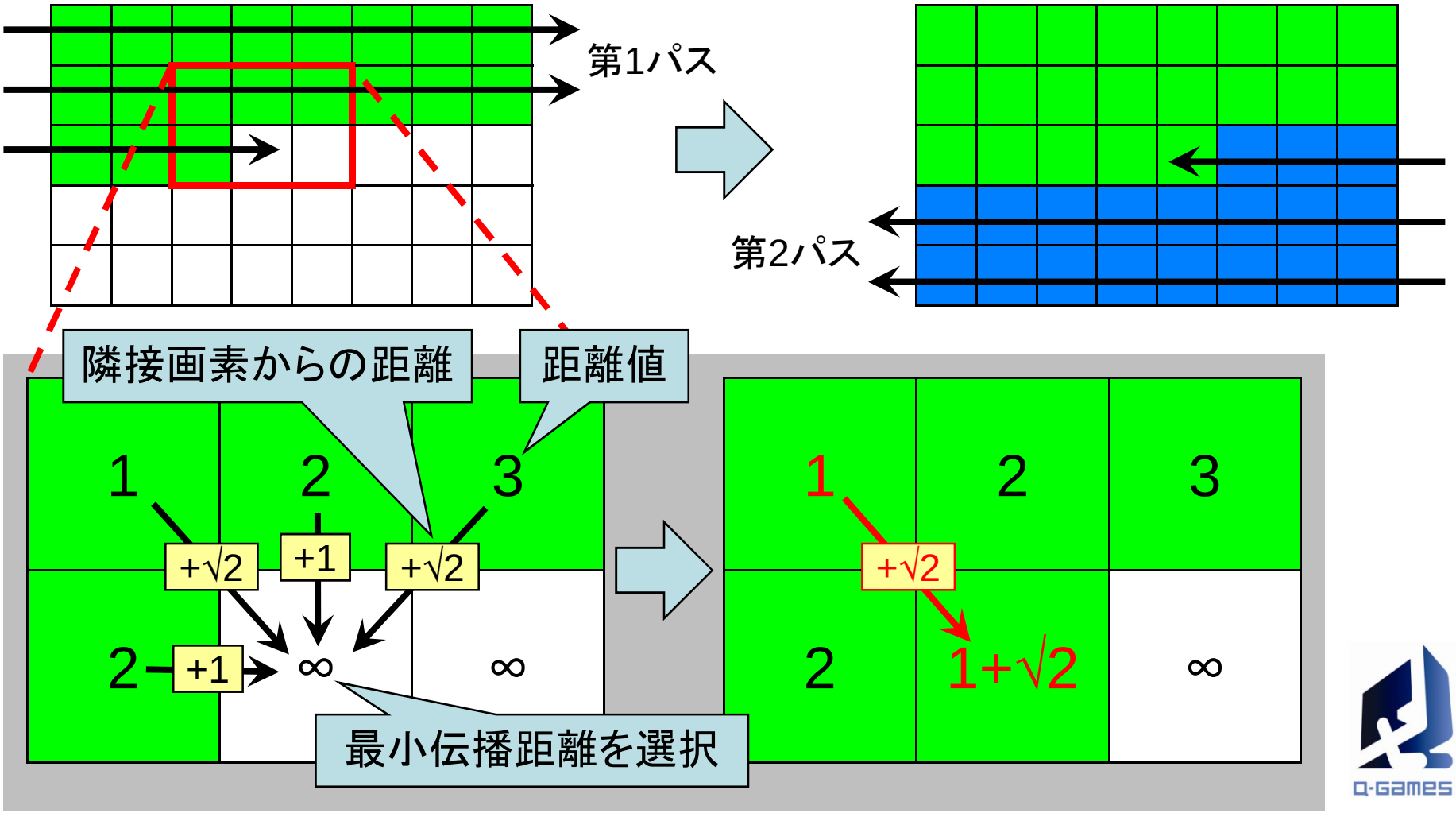
Chamfer distance algorithm (CDA) 概要

- アルゴリズムの流れ
  - 全ピクセルを距離∞で初期化
  - 入力2値画像のエッジ部分を距離0に設定(seed)
  - 距離伝播
  - オブジェクト外側（入力の黒領域）の値を反転
    - Signed distance field … 距離値符号により内外判定可能



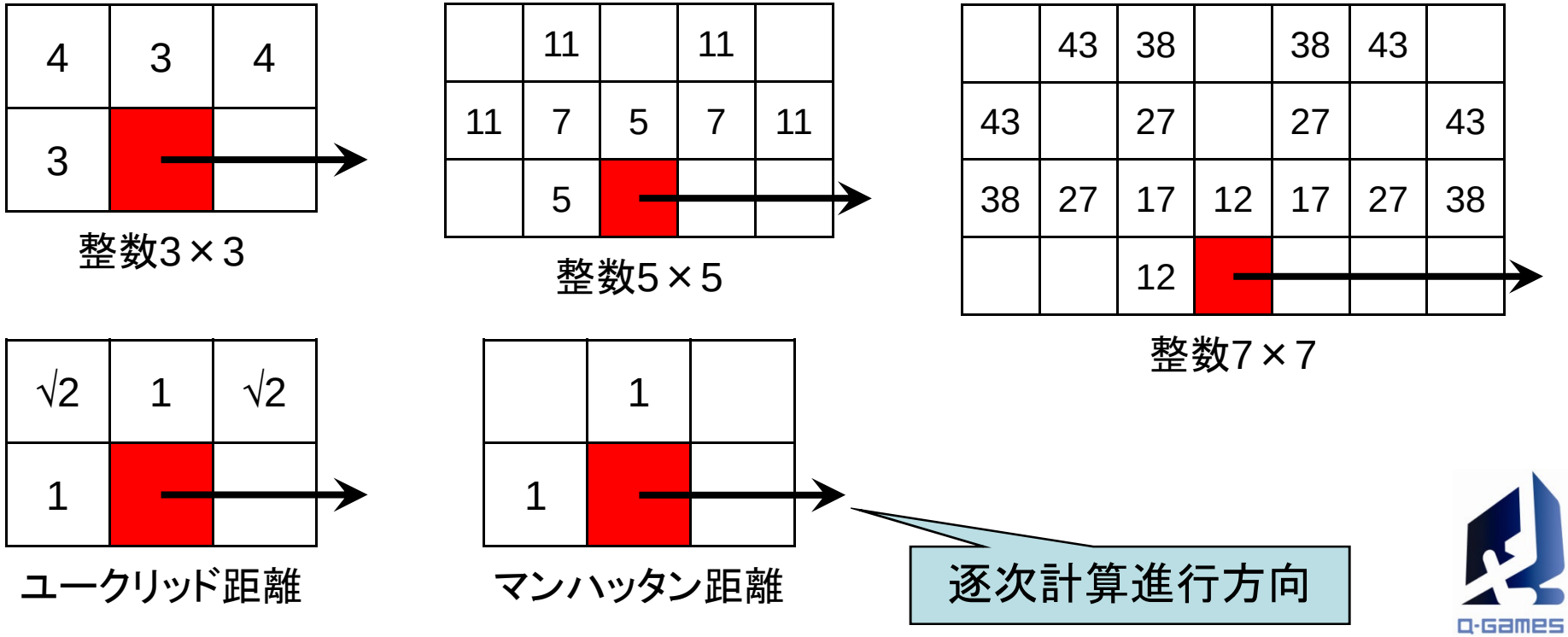
Chamfer distance algorithm (CDA) 距離伝播

- 逐次計算による距離伝播（×2パス）



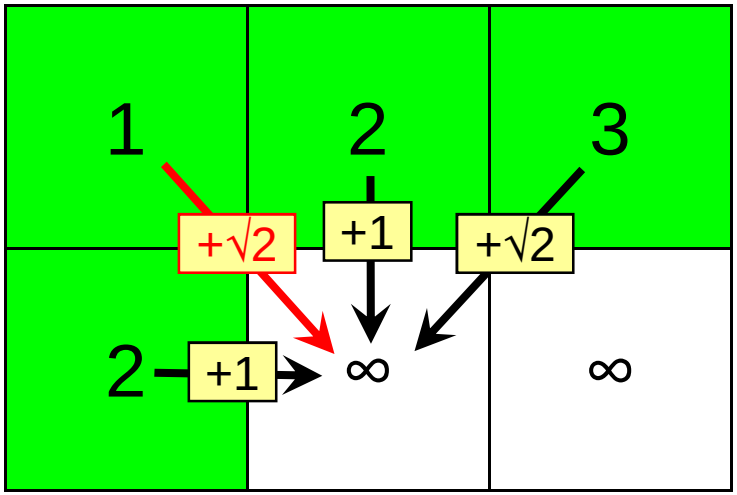
CDAの距離伝播ウィンドウ

- ウィンドウ：距離伝播に参照する範囲と加算距離
  - 正確なユークリッド距離ウィンドウより、整数値ウィンドウを用いたほうが実は誤差が少ないことがわかっている
  - 大きなウィンドウを使うほど正確な計算結果が得られる

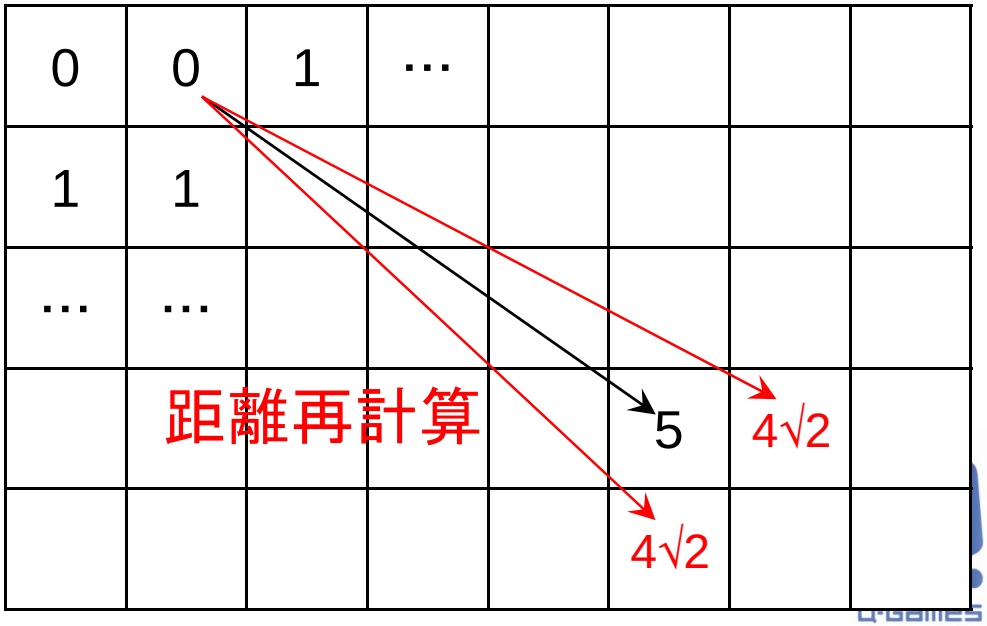


Dead reckoning algorithm

- CDA：距離だけを伝播
  - 各伝播ステップの計算誤差が蓄積される
- 距離=0の伝播元地点を各セルに記憶
  - 伝播ウィンドウは3×3ユークリッド距離を使用
  - 距離更新時に伝播元地点からの距離を再計算



最小伝播元隣接点を決定



SPU distance transform

- 距離空間
  - 計算が容易なマンハッタン距離を使用

$$\mathrm{Dist}((x_1,y_1),(x_2,y_2)) = |x_2 - x_1| + |y_2 - y_1|$$

  - 多少誤差はあるが使用に問題なし
- アルゴリズム
  - Chamfer distance algorithm
  - 原理的に並列化不可能なため1SPUで処理
  - 256×256リアルタイム計算、約1ms
    - ベクトル命令を用いたSPU向け最適化を実施
  - 512×512でも2ms程度で計算可能
    - SPUでのデータ使用時の容量制限から、最終的には使用せず

Jump flooding algorithm

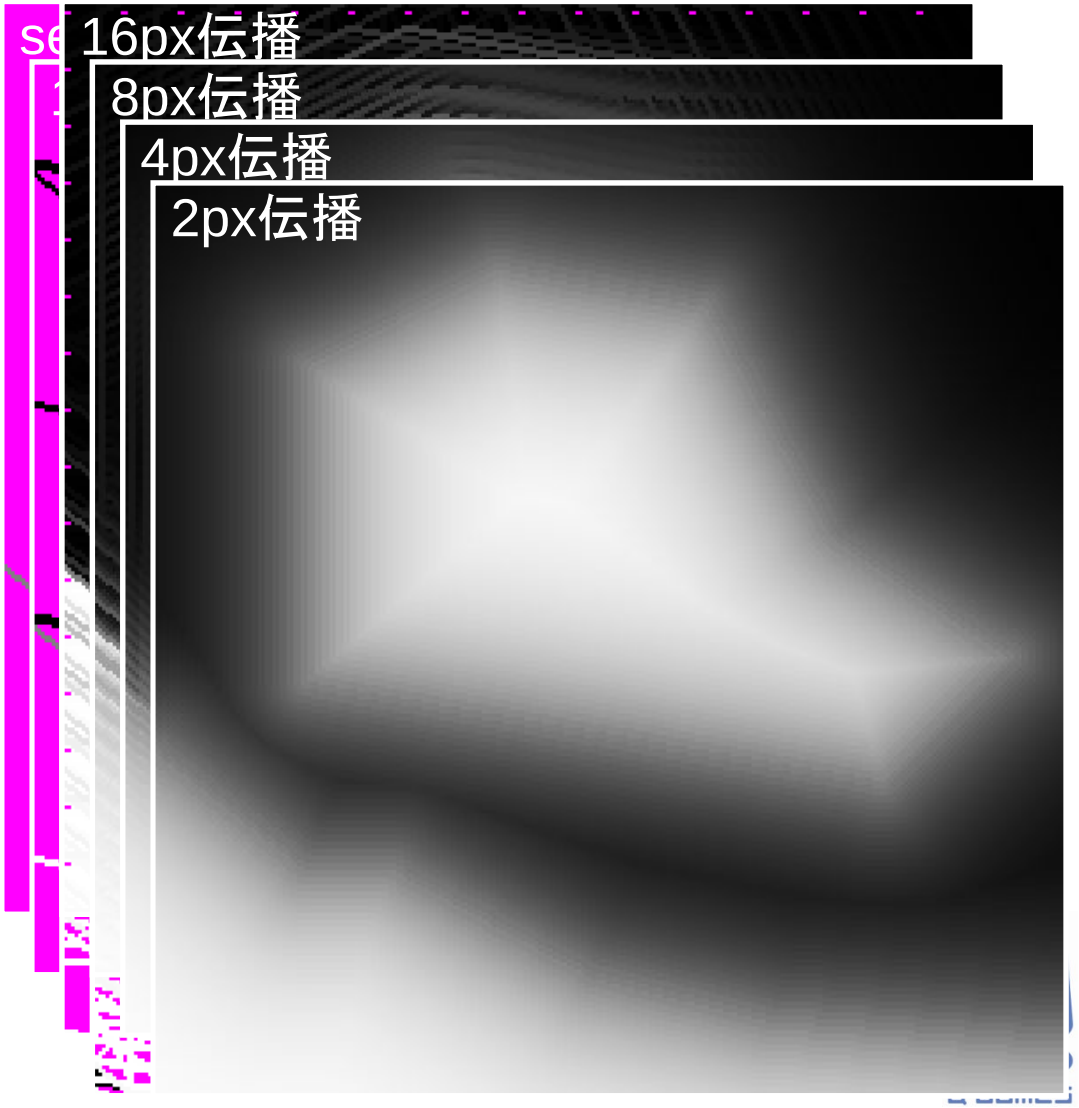
- Chamfer distance等の逐次計算アルゴリズム
  - 各ピクセルの計算が隣接ピクセルの計算結果に依存
  - 並列計算ができない
  - $n \times n$ ピクセル画像に対する計算量 $O(n^2)$
- 並列計算可能なアルゴリズム：Jump flooding
  - アイデア：長距離の大まかな距離伝播を先に計算し、徐々に近距離の細かい距離伝播にシフトしていく
  - 伝播距離をパスごとに1/2倍にする
  - パス数は $\log_2(n)$, 計算量 $O(n^2 \log_2(n))$
  - パス内画素単位で並列計算可能（GPU処理に適する）

Jump flooding計算例

- Jump flooding
  - パスごとに伝播距離を1/2倍していく

:未定義領域

256×256入力画像

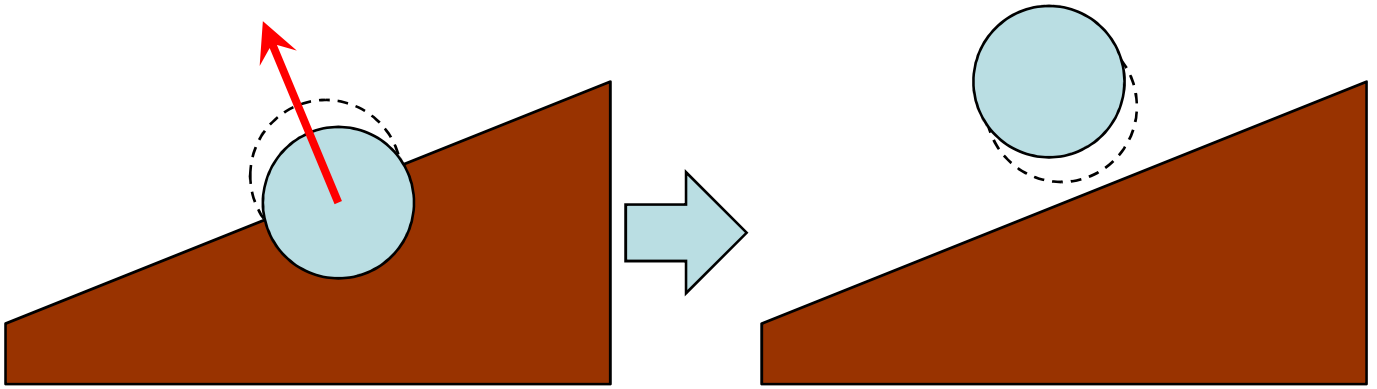


GPU distance transform

- PS3ビルド
  - Chamfer distance algorithm, SPU実装
- PCビルド（開発過程で使用）
  - SPU使用不可→GPUで代用
- Jump flooding algorithm, GPU実装
  - 256×256, 距離伝播8パス, 5～6ms
  - 8パスを1フレームで計算するのは厳しい
  - 4パス×2フレーム等に分割して計算
- GPU vs SPU
  - GPU：テクスチャを介してデータをやり取りする必要がある
  - GPU：純粋な描画関係処理の利用時間を圧迫する
  - SPU：DMAによるデータ通信、分岐等の複雑な処理が可

パーティクル流体システムの地形衝突判定処理

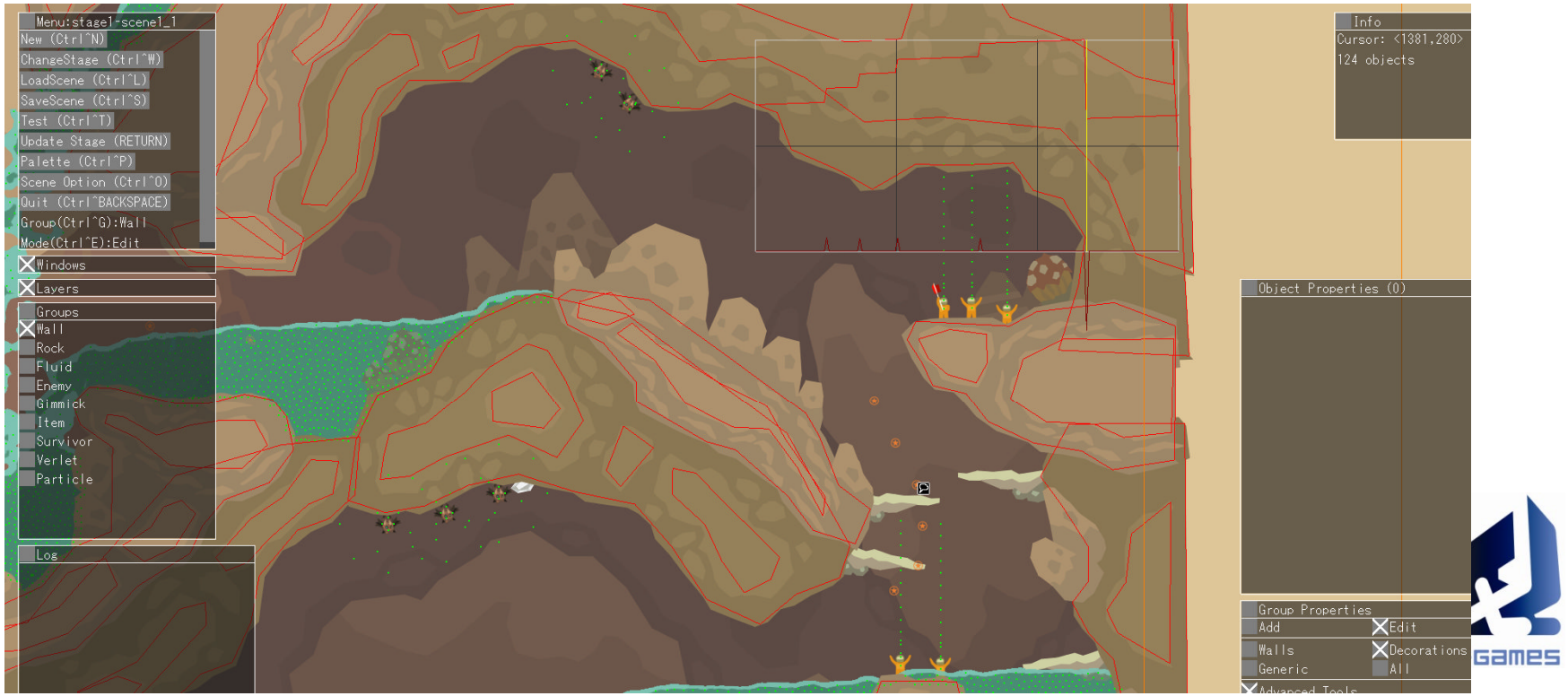
- Distance fieldを用いた衝突判定
  - Distance fieldは256×256=65,536px
  - 1px=1バイト整数値として64KB=SPUに格納可能
- 地形との衝突検出時
  - めり込み量に応じて、法線方向への反発力を加える



- ちなみに
  - PixelJunk Edenではdistance fieldはオフライン計算のみ

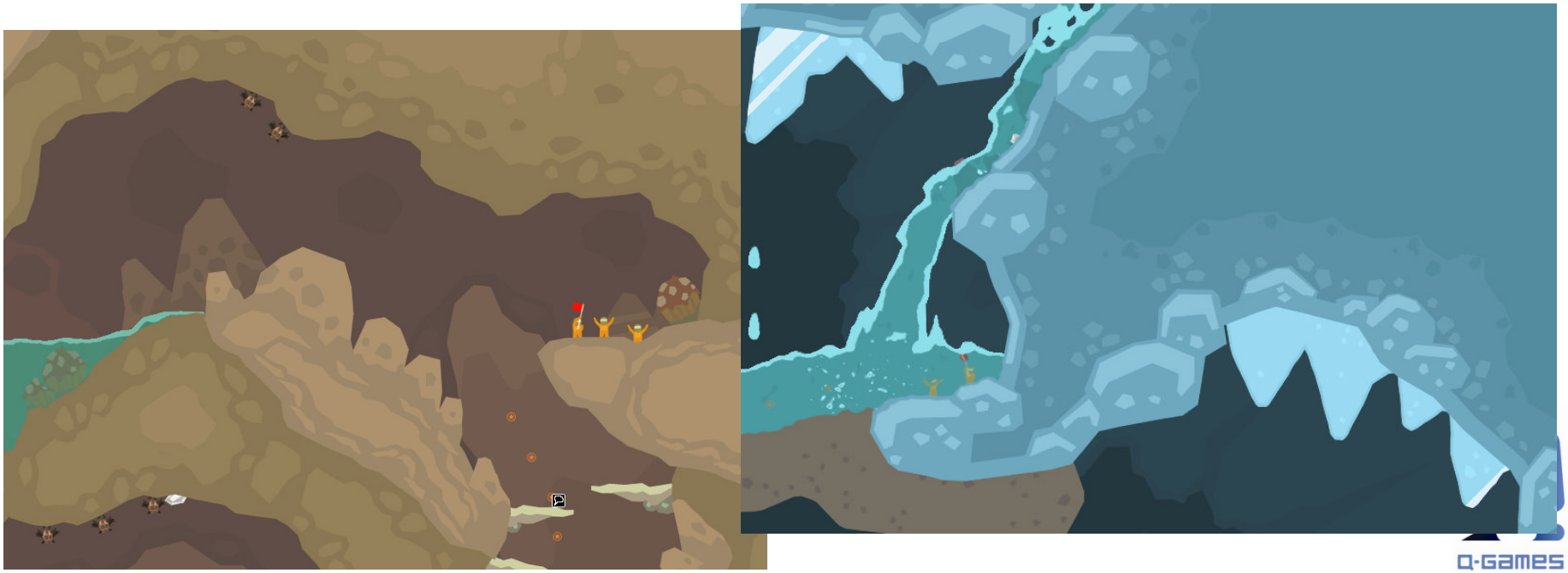
ステージエディタ

- 機能
  - 地形編集、キャラ・アイテム配置等一般的な機能
  - 流体編集、流体シミュレーション実行／停止
  - テンプレートによる地形デザイン機能



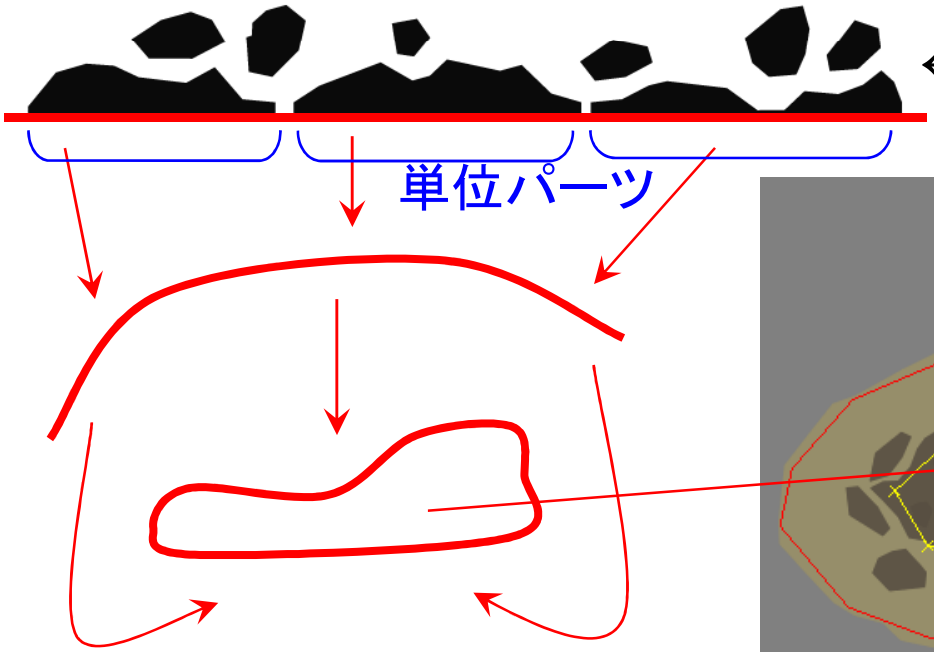
テンプレートを用いた地形デザイン

- 地形の（アート面での）デザイン
  - 地中の小石配置パターンや地表の凹凸パターン等
  - ステージごとに統一されたデザイン／コンセプト
  - すべてデザイナが手書きするには労力が必要

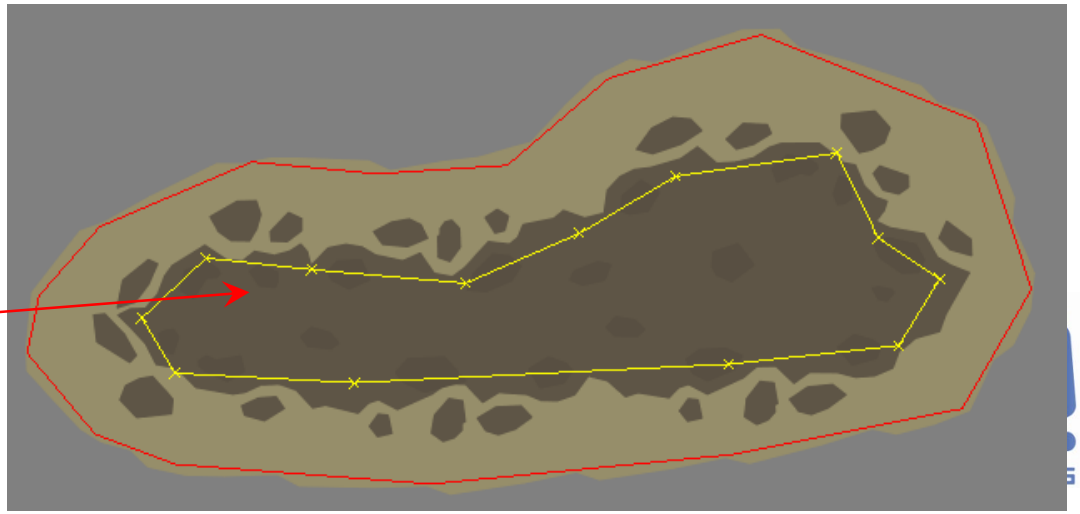


デザインのテンプレート化と利用

- パターンのテンプレート化
  - 地形デザインに必要なパターンをデザイナが作成(SVG)
  - レベル作成者はテンプレートを使用して地形をデザイン
  - テンプレートはいくつかのパーツに分けておく（手動）
  - ループや反転を用いてランダムかつシームレスに繋ぐ（自動）



←テンプレート：ベクタ形式なので
拡大縮小にも強い



GameMonkeyスクリプト形式のデータ管理

- GameMonkeyスクリプト
  - PixelJunkシリーズの高レベル部分の実装に用いられている
  - Lua系言語（Luaは各種ソフト、ゲームに採用実績）
- ステージデータ等の保存形式にスクリプト形式を使用
  - ステージ保存時にスクリプトを自動生成
  - データ読み込みはスクリプトのコンパイル＆実行
  - テキスト形式のため、可読性・保守性等に優れる
  - 実行時に使用されるハッシュテーブル形式のデータ構造をそのまま読み込むことができる
  - データ形式を独自に規定する必要がない

まとめ

- パーティクル流体シミュレーションシステム
  - 32,768particles@5SPU, 60FPS
  - 熱伝導機能、衝突検出情報のフィードバック機能 etc.
  - 汎用衝突検出器として利用
- Realtime distance field
  - 地形との衝突検出に利用
  - Chamfer distance, Dead reckoning, Jump flooding, etc.
  - CDA, 256×256, Manhattan@1SPU, 1ms
- レベル編集関連
  - テンプレートを用いた地形デザイン
  - GameMonkeyスクリプト形式のデータ管理
- Q&A?

END

- PixelJunk Shooter
  - 有限会社キュー・ゲームス
  - Q-Games, Ltd.

- ありがとうございました