PixelJunk™ Edenにおける植物制御に関する技術解説

PixelJunk™ Eden

有限会社　キュー・ゲームス

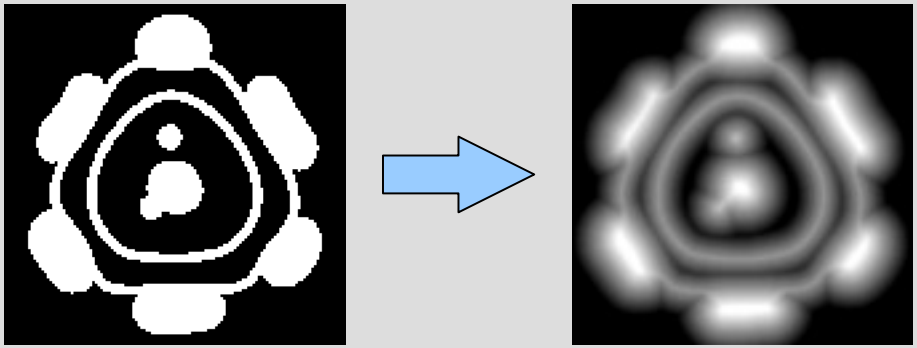
Jerome Liard
木下　直紀

セッション概要

- PixelJunk Edenに用いられている
  各種テクニック、チャレンジ、成功・失敗
  - Distance field、アンチエイリアシング、…
  - Picking buffer、衝突判定、…
  - 植物の表現、シミュレーション、…
  - Skinning、SPU v.s. GPU、…
  - シルクのシミュレーション、Verlet、…
  - Particle system、SPU、query prefetch、…
  - スクリプティング、GameMonkey、…

Distance fieldの活用

- (Signed) distance field/transformとは？
  - 各画素からオブジェクト(の境界)への距離を画素値で表現



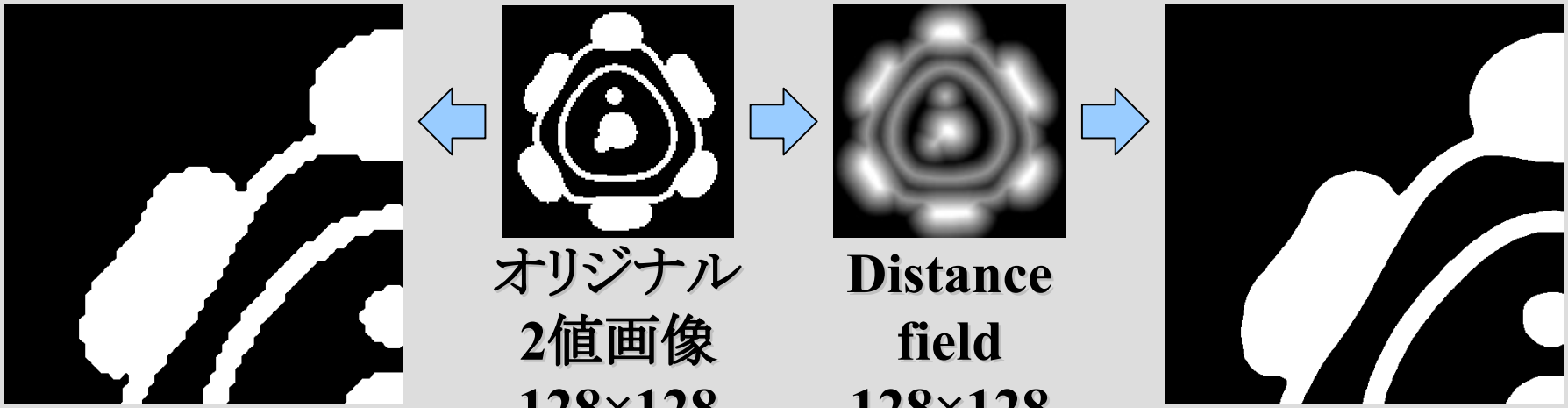
Input　　Output

白色で示された領域への距離をsigned distance fieldに変換した例

- Distance fieldの応用例
  - 拡大画像のエッジ改善
  - エッジ抽出、アンチエイリアシング

拡大画像のエッジ改善

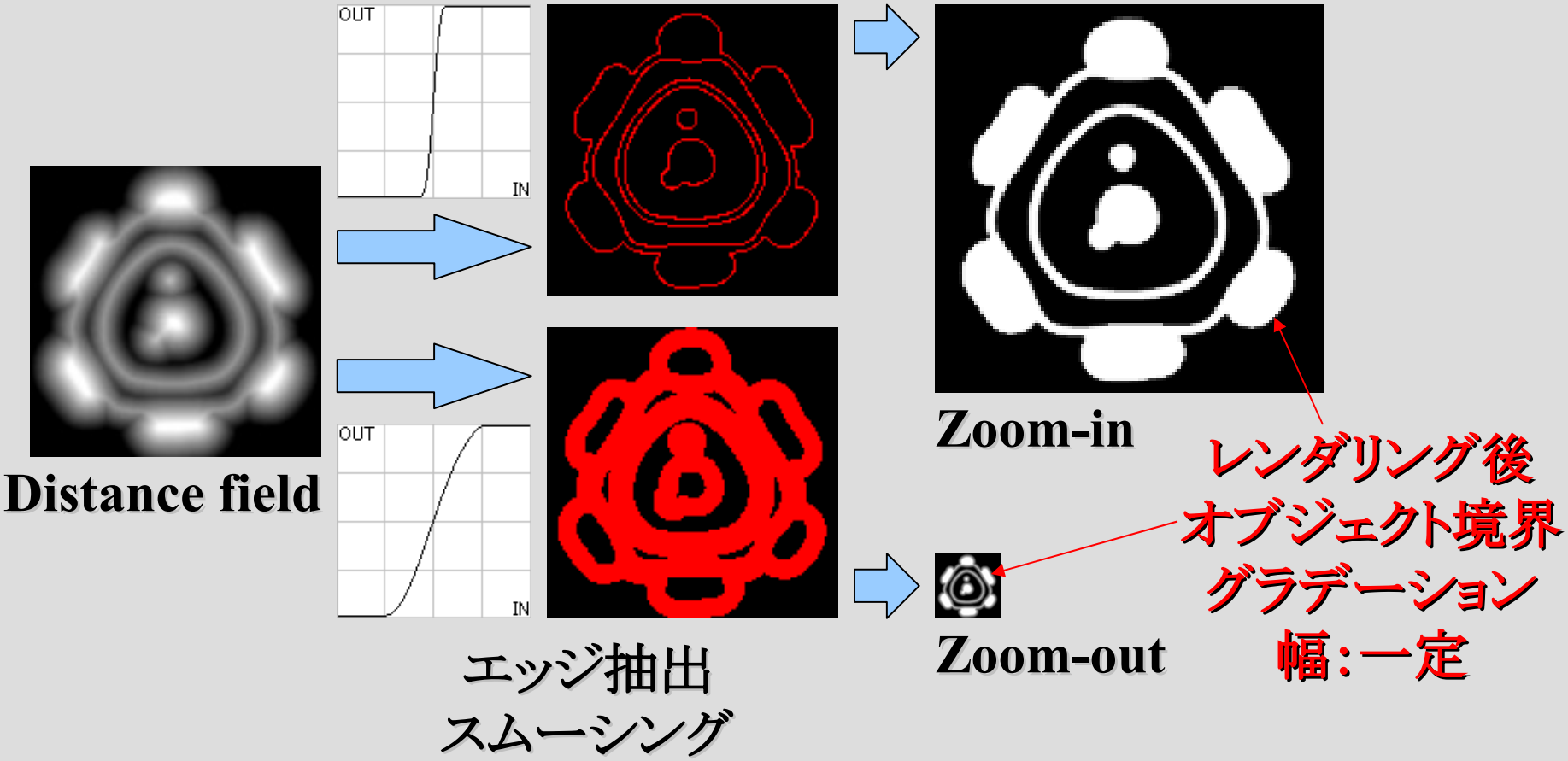
- 2値画像とそのdistance fieldを拡大して2値化
  - 拡大方法:線形補完
  - Distance fieldはオフラインで計算可能



- Chris Green. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. SIGGRAPH Course on Advanced Real-Time Rendering in 3D Graphics and Games, 2007.

Distance fieldによる アンチエイリアシング

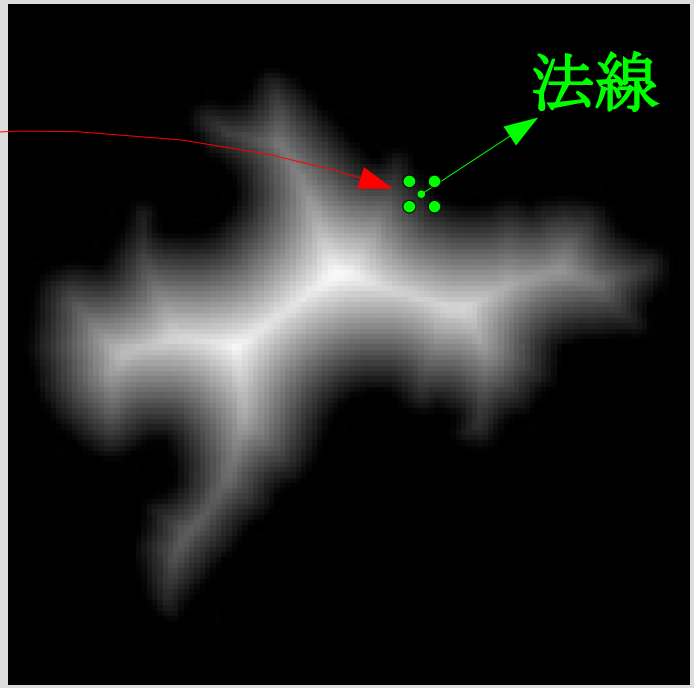
- ズームレベルに応じて、distance fieldテクスチャからアンチエイリアシングの適用幅を変化させる



Distance fieldによる法線計算

- 衝突が検出されたオブジェクトのdistance field から衝突点近傍の4点をサンプル
- 4点の値から衝突点の法線ベクトルを計算

ゲーム画面

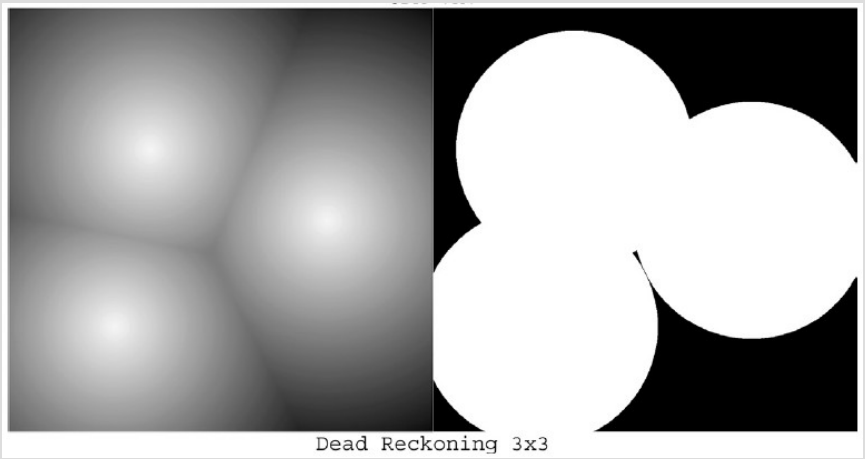


Distance field

Distance fieldの高速計算

- PixelJunk Edenでは全てビルド時に予め計算
  - とはいえnaiveなアルゴリズムでは
    入力画素数nに対し$O(n^2)$の計算時間
- “Dead reckoning”
  - 高速・高精度の近似計算
- G. Grevera. The “dead reckoning” signed distance transform. Computer Vision and Image Understanding 95(3), pp. 317–333, 2004.

	forward pass			backward pass		
Dead	$\sqrt{2}$	1	$\sqrt{2}$	-	-	-
Reckoning	1	*C*	-	-	*C*	1
3x3	-	-	-	$\sqrt{2}$	1	$\sqrt{2}$

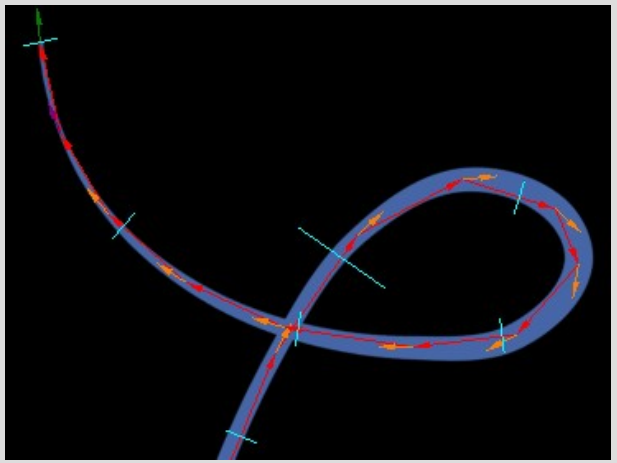

Dead Reckoning 3x3

ランタイムデータのビルドシステム

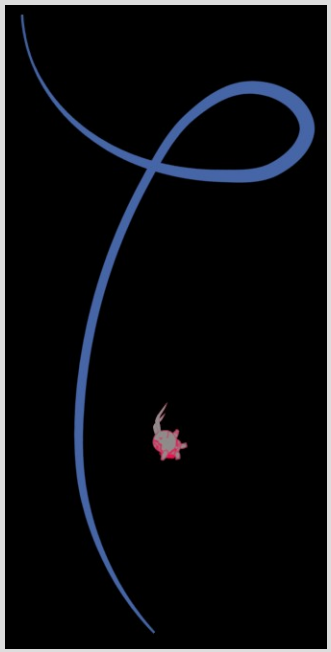
- GNU makeにより各種データ変換作業を自動化
  - TGA (original) → TGA (distance field) → DDS A8 (PC) → GTF (PS3)
  - WAV (original/PC) → AT3 (PS3)
- プロジェクトのバージョン管理：Subversion
  - リポジトリ肥大化防止のため、開発終盤まではランタイムデータはチェックインしない
  - チェックイン前はrsyncによりデータを同期

“植物”の表現

- シミュレーション単位：セグメント
  - 枝（branch）を微小区間に分割
  - 長さ、太さ、角度
- 描画時にリアルタイムで中間ポリゴンを補完（skinning）
  - PS3ではskinningにSPUを使用



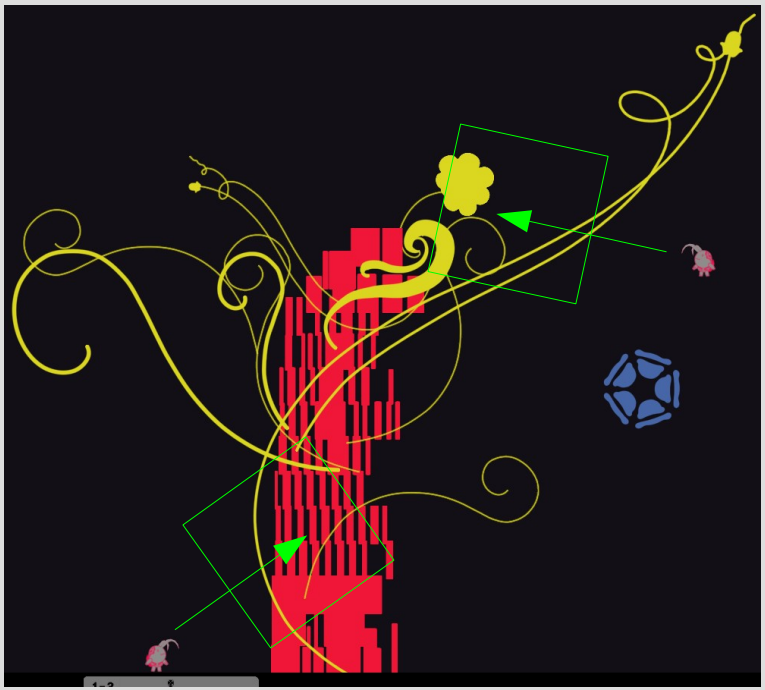
セグメント分割



植物の例

Pickingによる衝突判定

- 衝突判定専用のpicking bufferに低解像度で判定対象オブジェクトをレンダリング
- 描画色にオブジェクトID等の情報を埋め込み



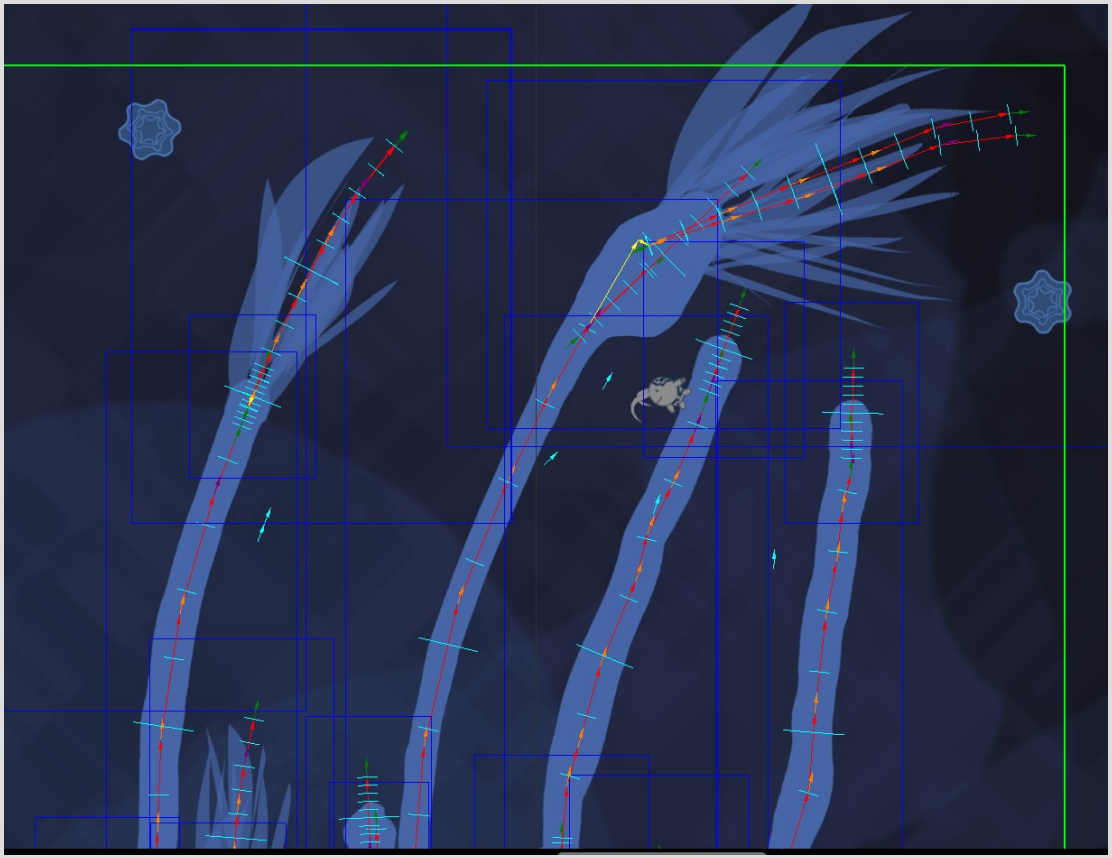
ゲーム画面

- Local picking
  - 全体をPicking bufferに描画せず、判定リクエスト点付近のみを個別に描画

Picking buffers

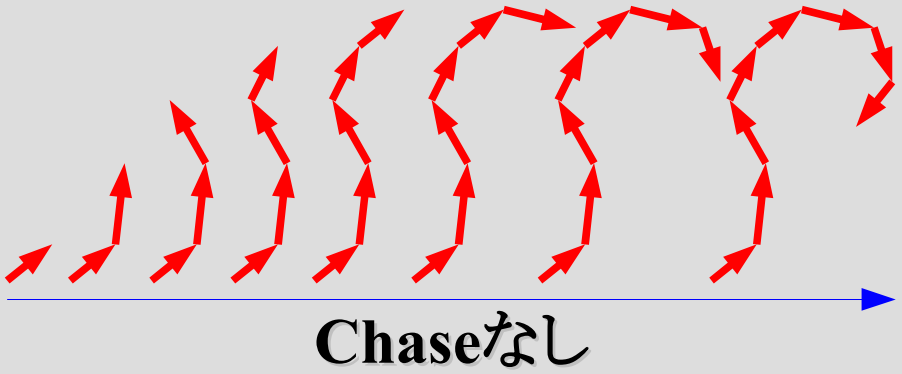
“植物”のシミュレーション

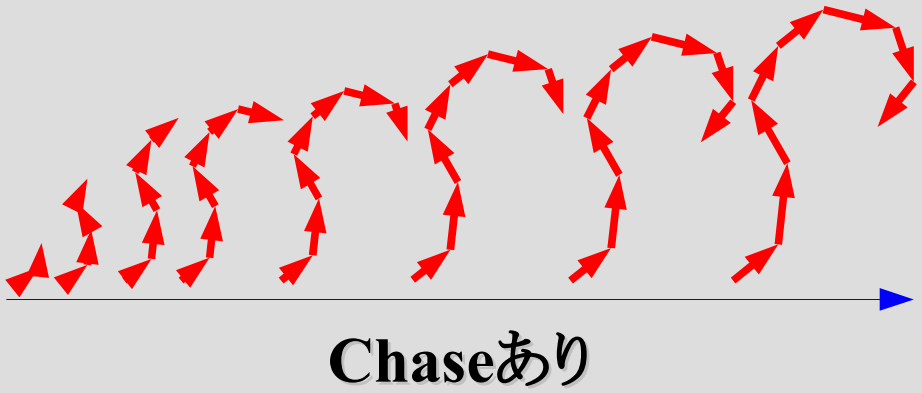
- 風や重力のある環境下で自然な動きを表現
- **K. Anjyo, Y. Usami, T. Kurihara. A Simple Method for Extracting the Natural Beauty of Hair. ACM Computer Graphics Vol.26 No.2 pp.111-120**



“植物”のシミュレーション

- シミュレーション単位：セグメント
  - 主なパラメータ：固さ、ダンピング係数
- パラメータはセグメントごとに指定
  - e.g. 根元は固く、先端はやわらかい
- **Exponential chase**
  - セグメントの成長速度を非線形に制御することでなめらかな成長アニメーションを表現

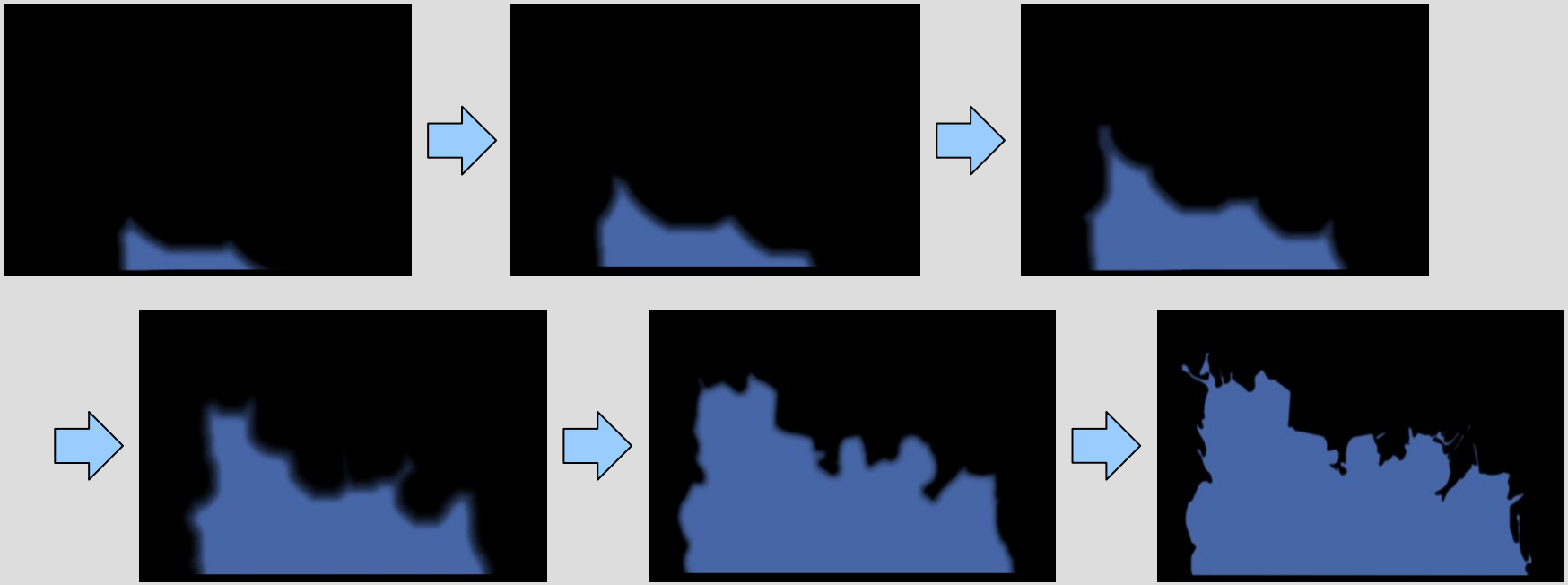




Depth mapによる成長アニメーション

- Distance fieldと似たアイデア
- 明→暗に向けて成長させる
  - 複雑なパターンのアニメーションも表現可能

Depth map texture



アーティストとの連携

- 完成イメージからテクスチャ素材への逆変換
  - アーティストが植物の完成イメージを作成
  - デザイナーがパーツ単位に分解
  - 植物の枝部分を専用ツールでトレース
  - テクスチャ直線化
  - セグメント情報出力
- 分解されたパーツはスクリプトで組み立て

完成イメージ

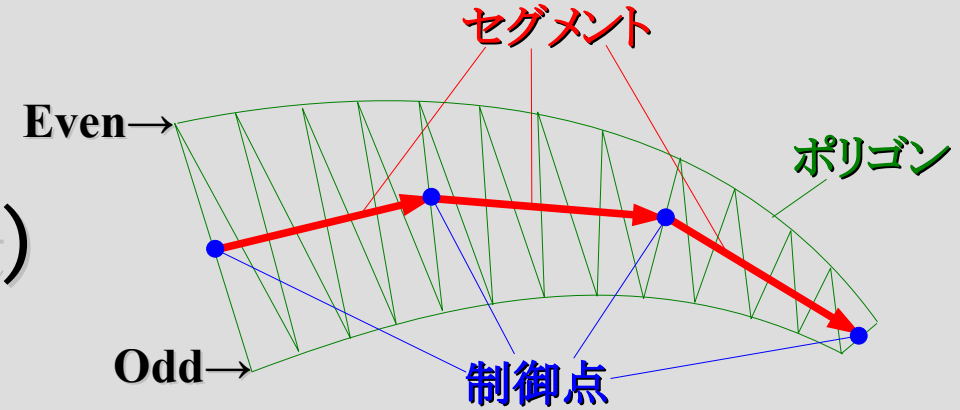

枝のトレース

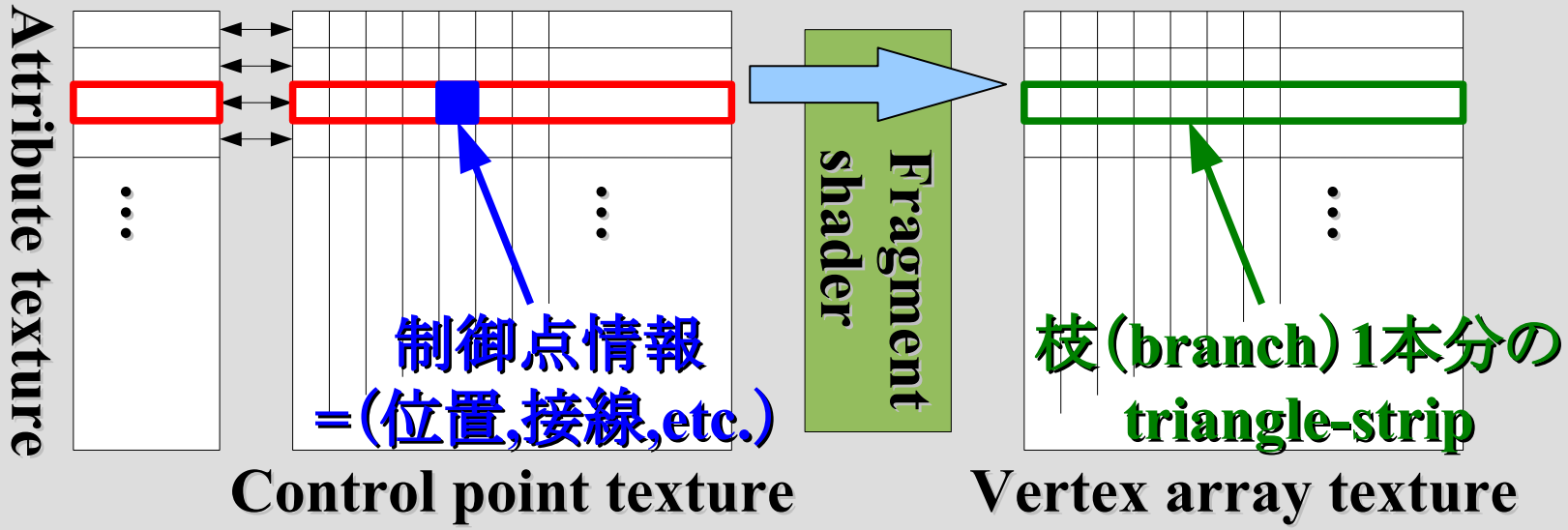
PCビルドの活用

- PCビルドとPS3ビルドを併用
- PCビルドのメリット
  - ビルド、起動が高速（データ転送が不要）
  - ステージ編集等の作業に適する
- PCビルドのデメリット
  - SPUが使用不可（PC専用の回避策を要する）
  - PS3ビルドと共存させての保守が煩雑
- 実装の異なる例
  - PCビルド：GPU skinning
  - PS3ビルド：SPU skinning

GPU skinning

- Fragment shaderでskinningを処理
  - PCビルドで実装
  - 保守性の問題
  - 精度の問題（half float）
  - PS3ではSPUを使用





SPU skinning

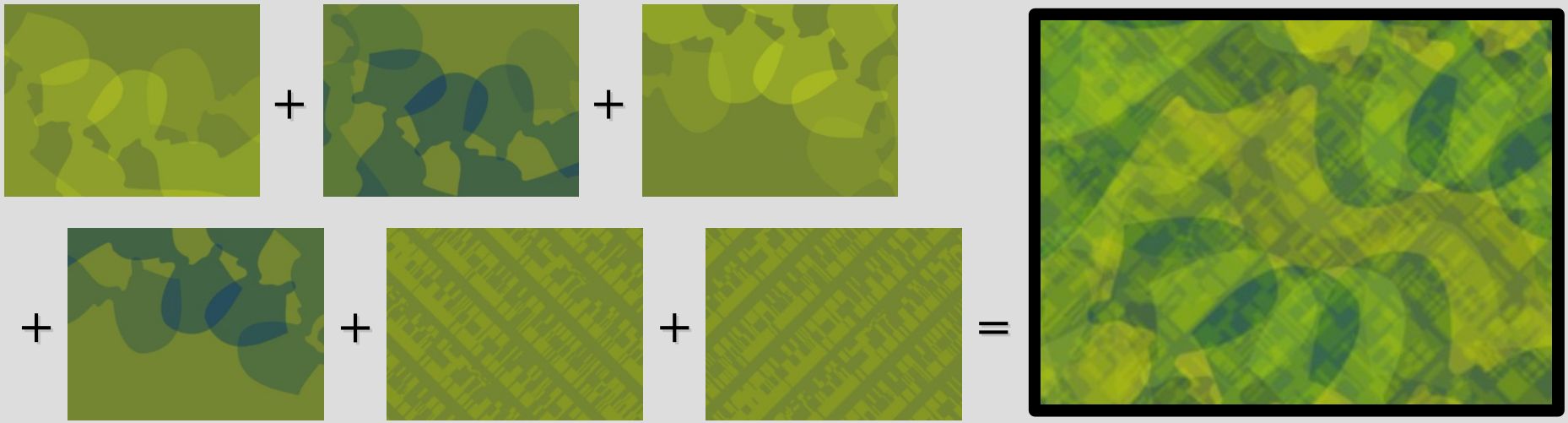
- Job system
  - シェーダのように、ジョブごとに1個の定義ファイルを作成
  - 入力クラス定義
  - 出力クラス定義
  - SPU mainコード C++でそのまま記述
  - Jobシステムが自動的に処理

```
%% ← job system tag
// 入力クラス
struct Input { ... };
%%
// 出力クラス
struct Output { ... };
%%
// C++ code for SPU
...
JOB_MAIN()
{
   ... // C++ SPU main
}
```

skinning.job
(ジョブ定義ファイル)

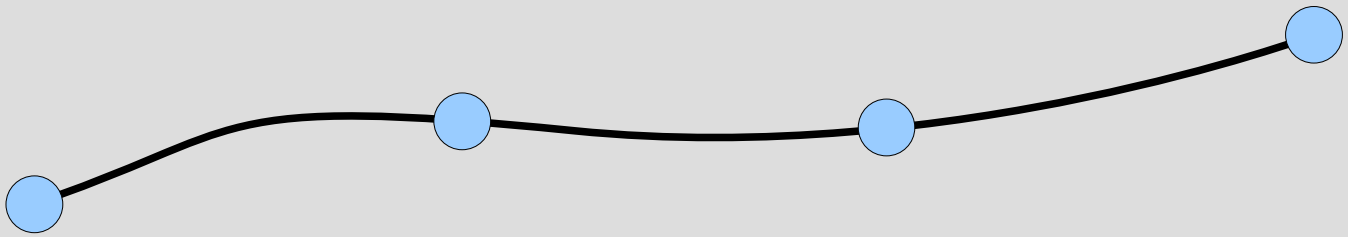
BGエフェクトの負荷軽減

- 巨大なポリゴンを多数重ね合わせたBG表現
  - 詳細な解像度は要求しない
- 小型（512×256px）オフスクリーンに描画→拡大
  - 容易に大幅な負荷軽減が可能
  - e.g. 7.7ms → 1.5ms



シルクのシミュレーション

- 結合されたノード集合として表現
  - ノード間の単純な制約式により
    シルクの動きをシミュレート
  - 物理的厳密性より安定性、
    計算量の少なさを重視した方法
- **Verlet integration**
  - 安定かつ単純な物理計算

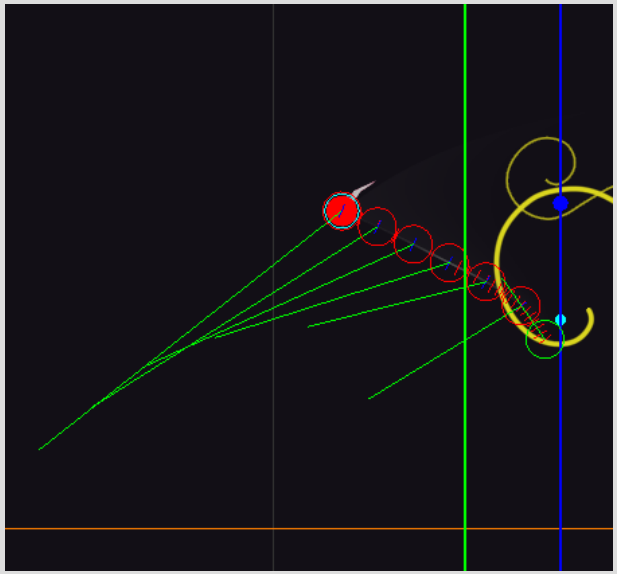


Euler integration

- ノードの持つ物理量
  - 時間 t における
    位置 $x(t)$, 速度 $v(t)$, 加速度 $a(t)$, 力 $f(t)$
  - 質量 m
- 微小時間 Δt 後の状態計算
  - $x(t+\Delta t) = x(t) + v(t)\Delta t$
  - $v(t+\Delta t) = v(t) + a(t)\Delta t$
  - $f(t) = m \cdot a(t)$

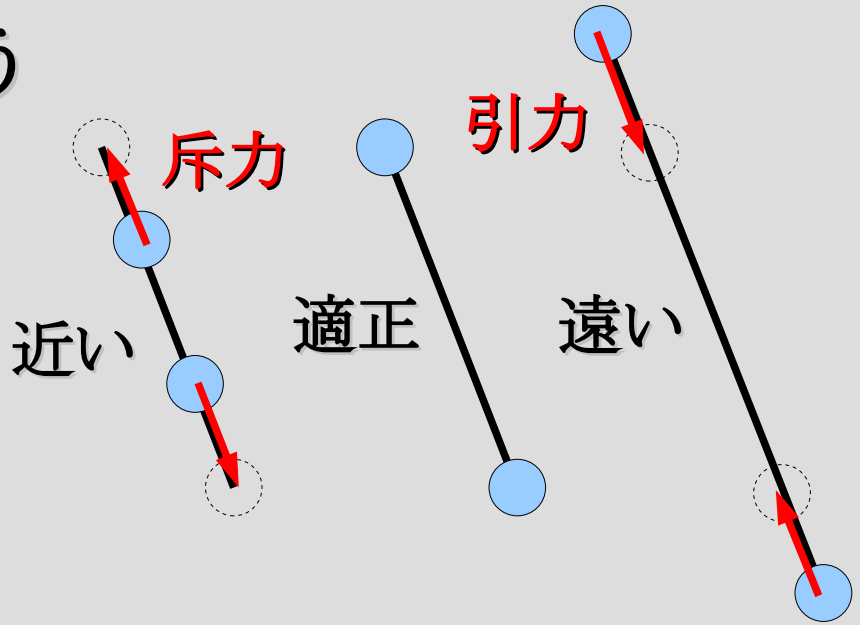
Verlet integration

- ノードの持つ物理量
  - 時間 t における
    位置 $x(t)$, 加速度 $a(t)$, 力 $f(t)$
  - 質量 m
- 微小時間 Δt 後の状態計算
  - $x(t+\Delta t) = 2x(t) - x(t-\Delta t) + a(t)\Delta t^2$
  - $f(t) = m \cdot a(t)$
- 各ノードは速度 v の情報を保持せず、直前の位置 $x(t+\Delta t)$ から求められる
  - 位置と速度の同期が外れることがなく、安定



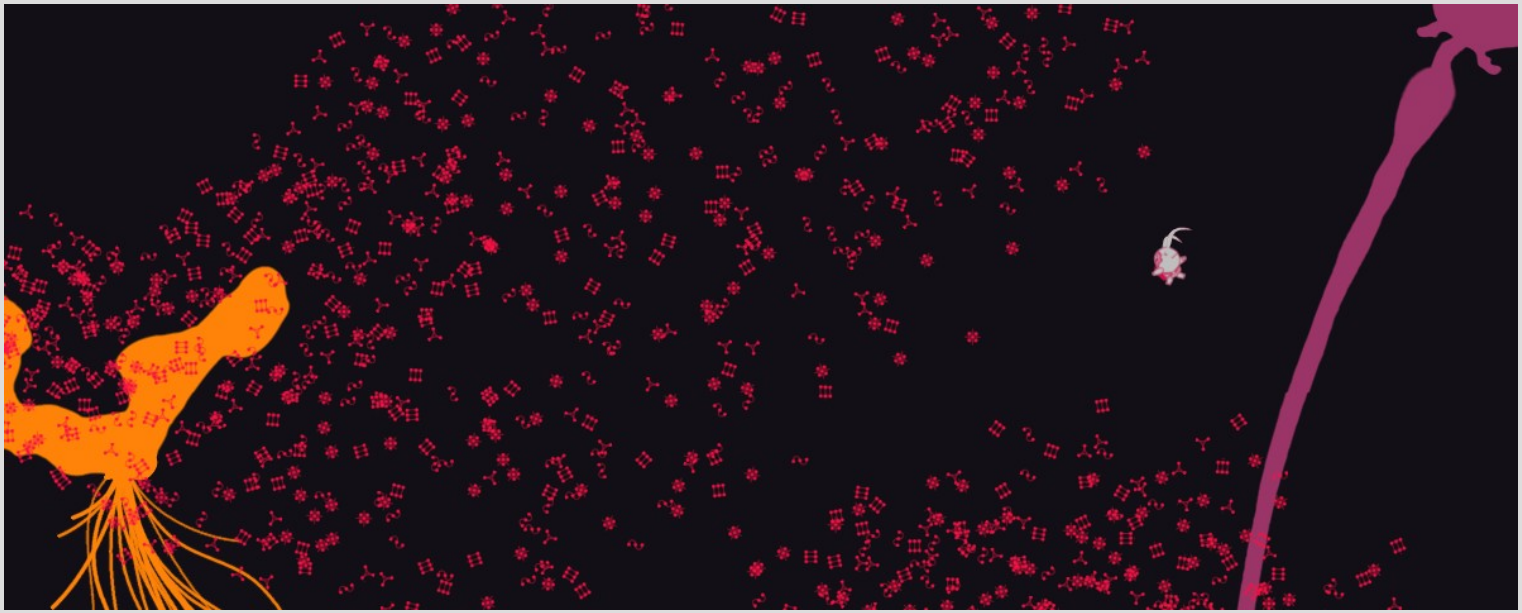
ノード間の制約

- ノード間の距離が適正距離になるように
  ノードに引力、斥力を加える
- 追加要素
  - ノード間の最大距離
  - ノードの最高速度
  - スイングの中心へ向かう
    加速度のダンピング
  - …
- 得られた知見
  - **Physicsよりtricks？**



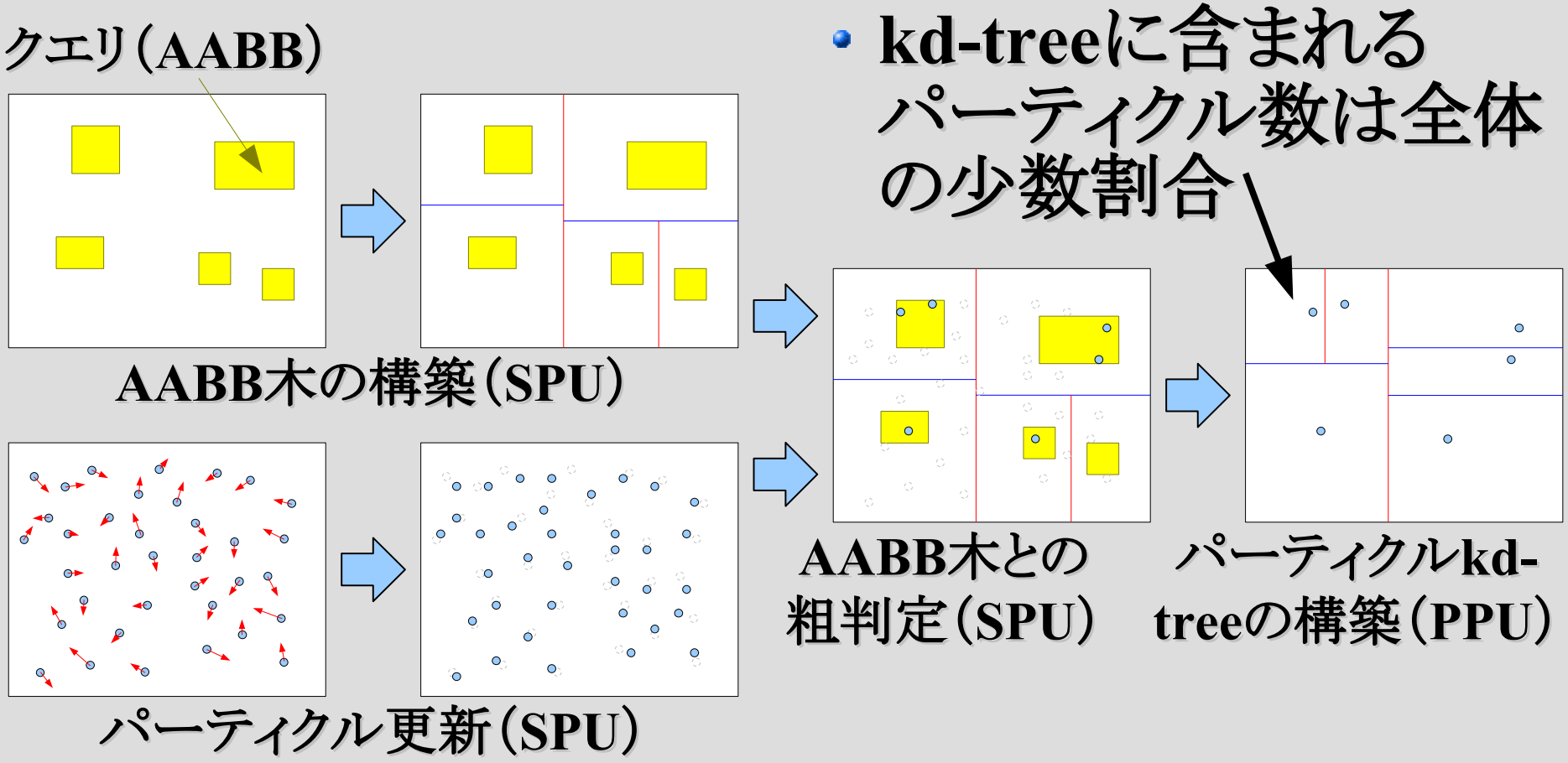
Particle system

- SPU 1基の一部を利用し、最大10240個のパーティクルを数ミリ秒で更新
  - 重力、風、その他外力
  - 衝突判定のための粗判定（query prefetch）
  - “コイン”にも同じシステムを利用



Query prefetchによる粗判定

- フレーム開始時に衝突判定を行いたい領域をあらかじめ指定(query prefetch)



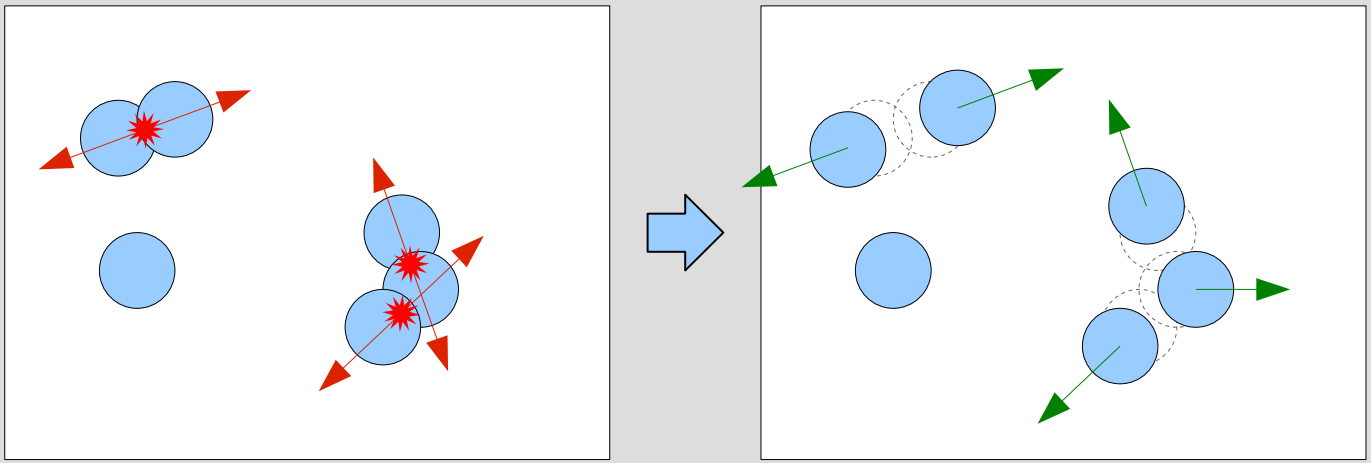
- kd-treeに含まれるパーティクル数は全体の少数割合

SPUによるパーティクル並列計算

- PixelJunk Edenのパーティクルシステム
  - 個々のパーティクルの物理計算
  - 外部からの衝突判定クエリ
  - SPU 1個のみでの処理
- より複雑なパーティクルシステムに向けて
  - 数万パーティクル間の衝突判定・相互干渉
  - パーティクルと地形の衝突判定
  - SPUによる並列処理、SPURSによる実装

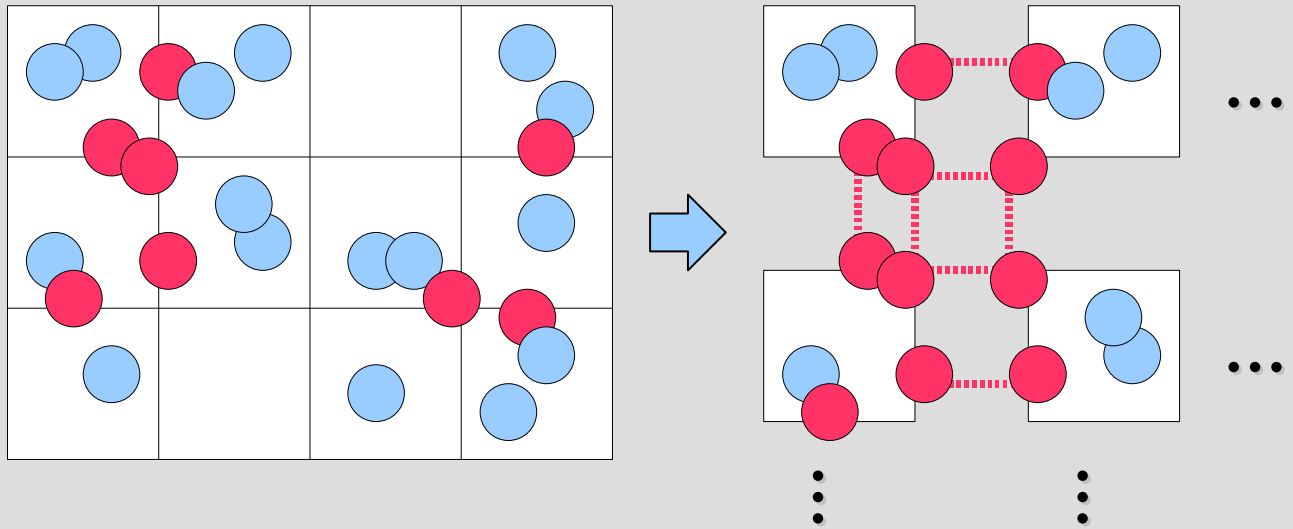
パーティクルの相互干渉

- 衝突したパーティクルの間に反発力を加えて位置が重ならないようにする
  - 単純なVerlet integration
- パーティクル数:数千～数万以上
  - 60FPS



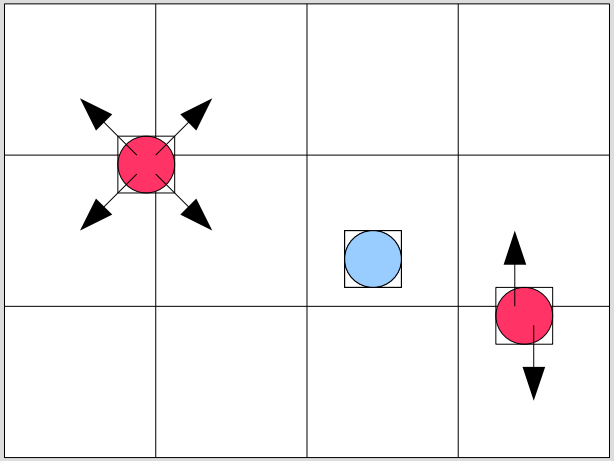
基本アルゴリズム（並列化前）

- 単純なアルゴリズム：総当り
  - nパーティクル：$O(n^2)$
- グリッドベースのアルゴリズム
  - グリッド状のセルに分配し、セルごとに総当り
  - 並列処理にも適用が容易
  - kセル：$O(k \cdot (n/k)^2) = O(n^2/k)$
  - $k \sim n_{max}$
    $\rightarrow O(n)$

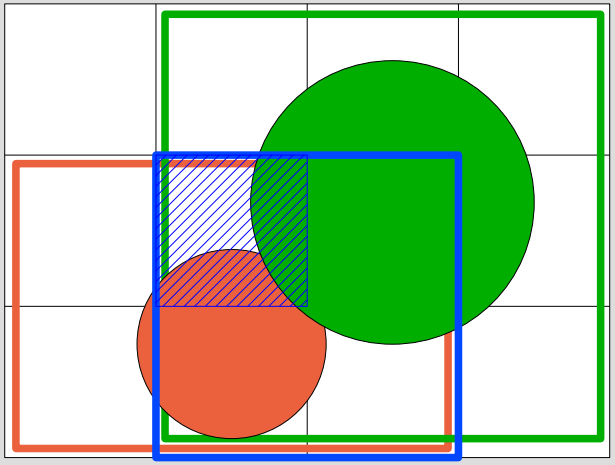


パーティクルのセルへの分配

- (a)各パーティクルの外接矩形が重複するセルすべてに同一パーティクルを分配
  - 各セルでの衝突による反発力を後で統合
- (b)同一ペアが複数のセルに存在する場合
  - 重複セルの内最も左上のセルのみ計算



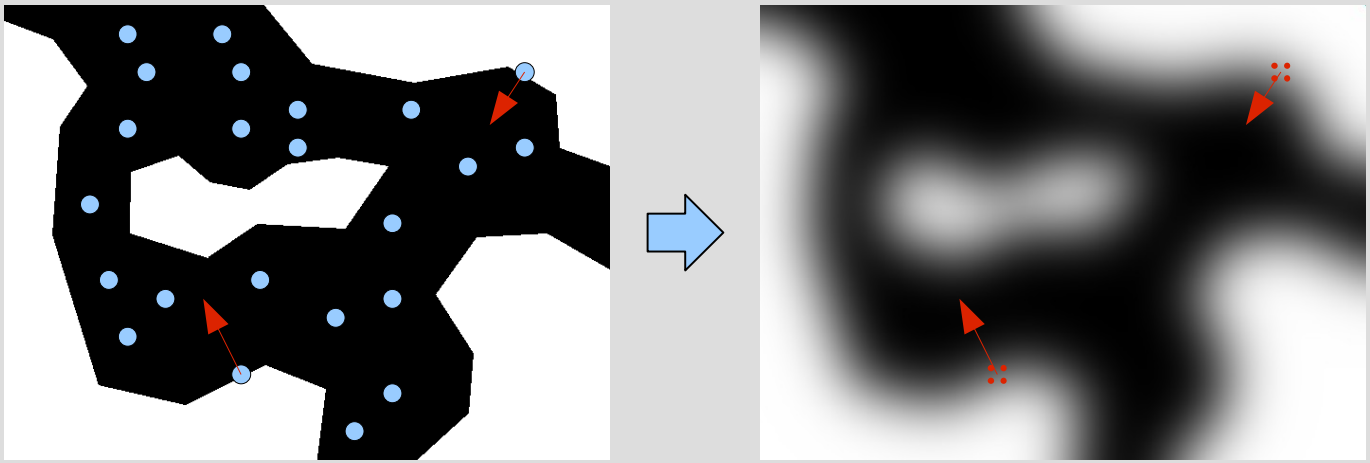
(a)



(b)

地形との衝突判定

- 地形との衝突判定はdistance fieldを用いて実現
  - 地形の複雑さによらず各パーティクル定数時間での衝突判定が可能
  - Distance fieldの解像度は地形の複雑さと計算精度とトレードオフ

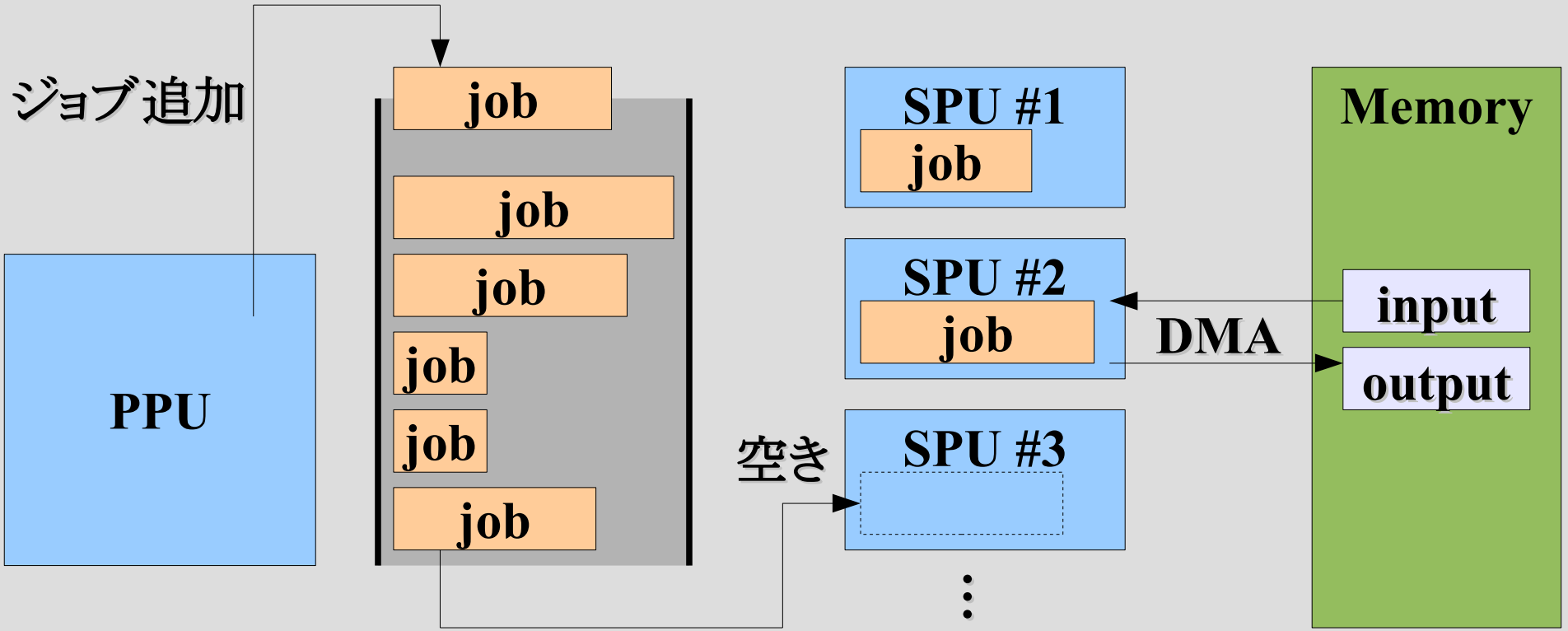


PLAYSTATION®3のCPU
Cell Broadband Engine

- PPE 1基とSPE 8基のヘテロ構造
- PPE(PPU)
  - PowerPC互換汎用コア
- SPE(SPU)
  - 自由に使えるのは6基
  - キャッシュメモリなし
  - 高速ローカルメモリ(LS) 256KB
  - SPE単体でのDMA制御
  - 128ビット計算
  - 分岐予測機構なし

SPURS
(SPU Runtime System)

- SPUベースのタスク管理システム
  - SPU自身によるタスク、コンテキスト管理
  - PPUにかかる負担が少ない



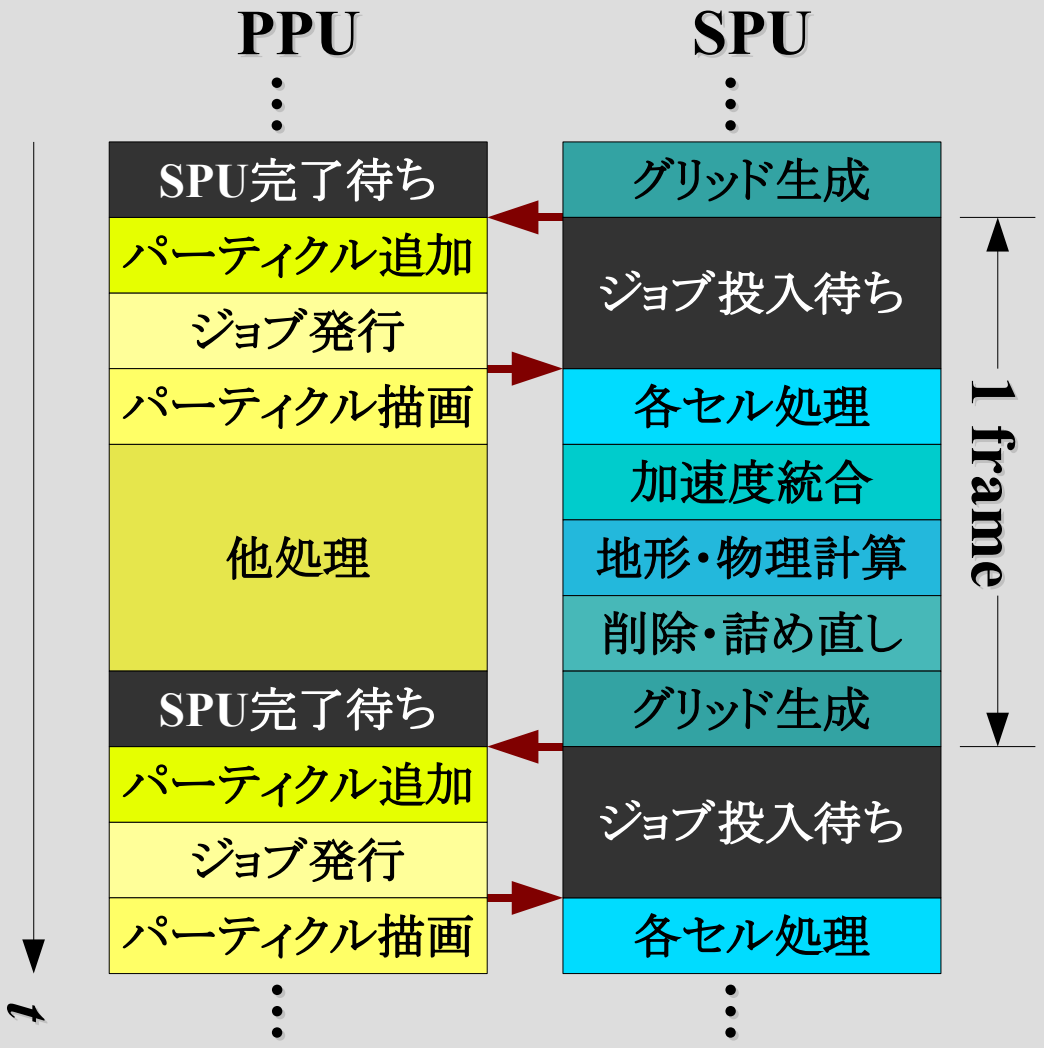
パーティクル処理の流れ

- (1)新しいパーティクルを配列に追加(PPU)
- (2)パーティクルをグリッドのセルに分配
- (3)セルごとの衝突判定
- (4)各セルでの加速度をパーティクルに統合
- (5)地形との衝突判定、物理計算
- (6)不要パーティクルを配列から削除(詰め直し)

- 実際には(3)(4)(5)(6)(2)を1フレーム分のSPUジョブチェインとして登録
  - グリッド構成後でないとPPUが(3)以降のジョブを発行できないため

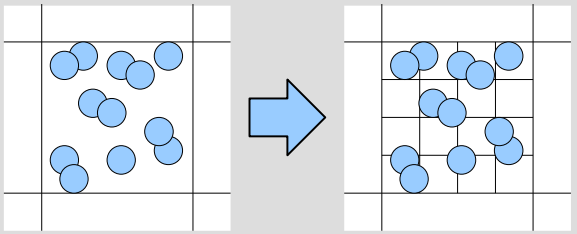
PPU／SPU並列処理

- できるだけ多くの処理をSPUで行う
- SPU処理はできるだけ複数のSPUで並列化する
- パーティクルの処理と描画の並列化のため、パーティクル配列をダブルバッファリングする必要がある

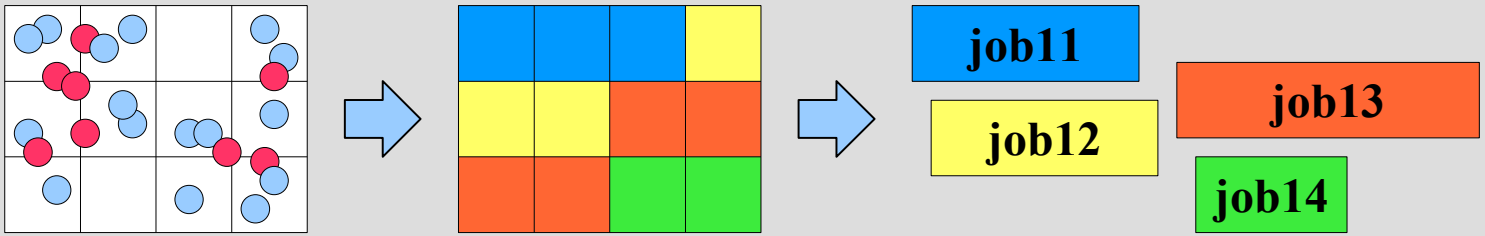


セルごとの衝突判定

- グリッド上の各セルごとにパーティクル間の衝突判定と反発力の計算を行う
  - 各セルをSPU側でさらにサブグリッドに分割することで高速化できる場合もある

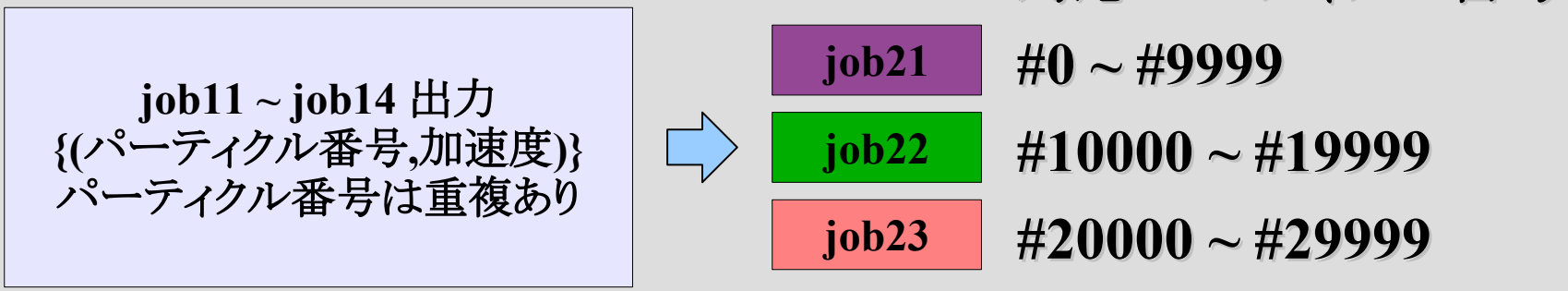


- 複数のセルをまとめて1個のジョブを発行
  - セル内パーティクル数の総和でジョブを区切る
  - 出力:セルごと {(パーティクル番号, 加速度)}



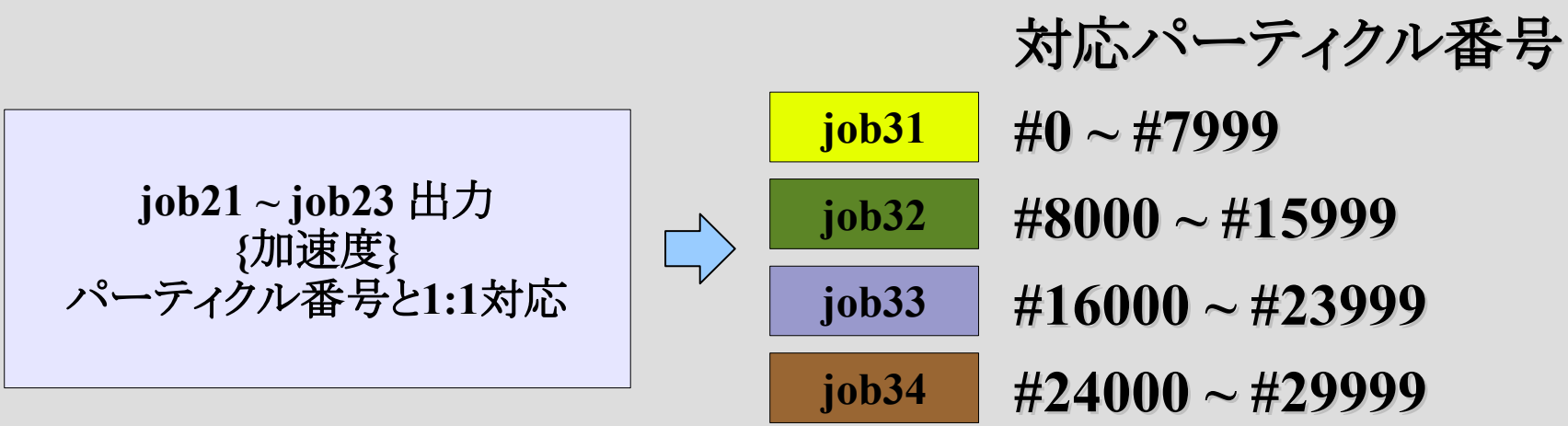
各セルでの加速度をパーティクルに統合

- 1個のパーティクルが複数のセルに分配された場合、セルごとに出力される加速度を個々のパーティクルに統合（加算）する必要がある
- 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統合ジョブを区切る



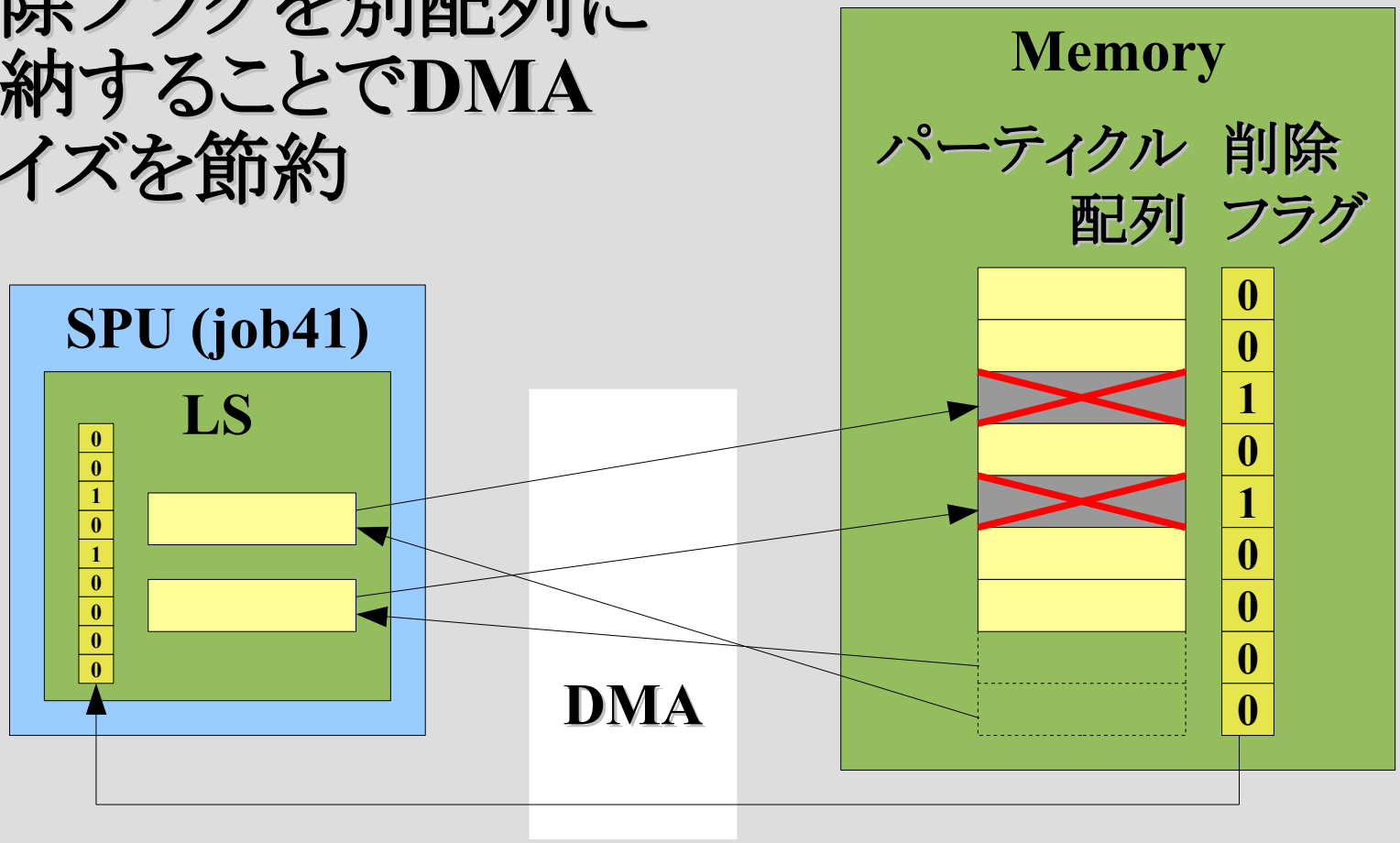
地形との衝突判定、物理計算

- Distance fieldによる地形との衝突判定処理と、Verlet integrationによる物理計算を行い、パーティクルの座標を更新する
  - Distance field全体をLSに格納する必要がある
- パーティクル番号の範囲でジョブを区切り並列化



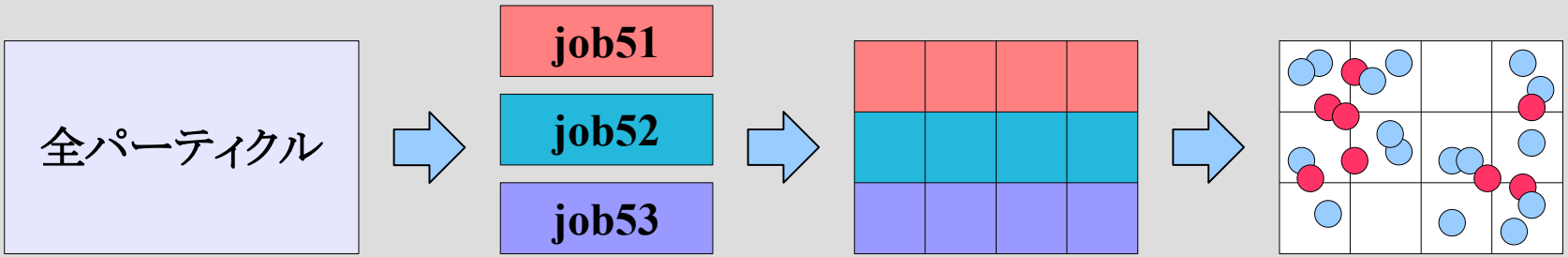
不要パーティクルの削除、詰め直し

- 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、不要なった要素を詰め直す
- 削除フラグを別配列に格納することでDMAサイズを節約



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
  - 各ジョブはそれぞれの対応する領域に含まれるパーティクルを選択し、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番号リスト配列

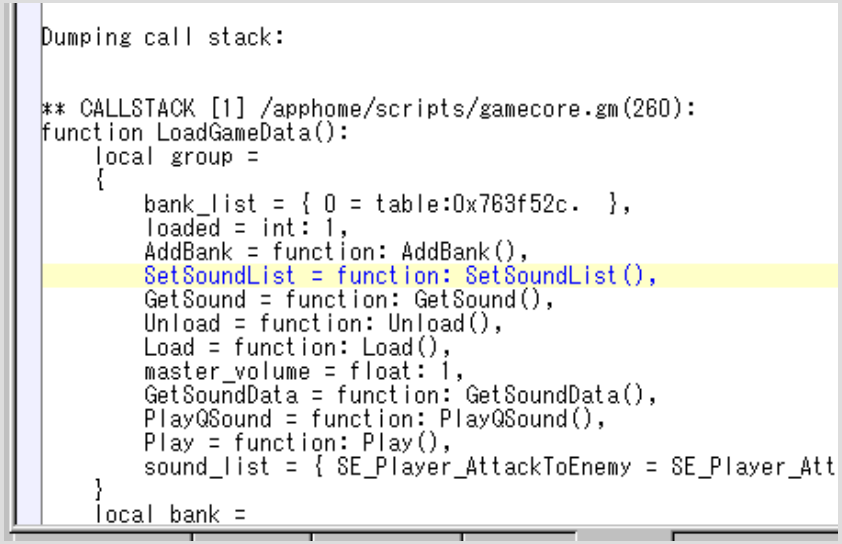


スクリプトの活用

- ゲームの大部分をスクリプトにより実装
  - **Lua系スクリプト言語GameMonkeyを使用**
- メリット
  - ビルド作業を高速化、生産性向上
  - ステージデータ等のコード埋め込みが容易
  - 安全性：参照とGCを用いたメモリ管理
  - 使いやすい“thread”
  - プログラマでなくても扱いやすい
- スクリプトは遅い？
  - コードの局所性
  - 頻繁に実行される箇所のみC++で記述

スクリプトの活用

- GameMonkeyの独自カスタマイズ
  - e.g. vector型の非GC対象化
- プロファイラの活用
  - スクリプトの指定範囲の実行時間、関数呼び出し回数、GC負荷等が容易に測定できる
  - グラフによる時系列可視化
- デバッグ機能
  - コールスタック表示
  - 変数のダンプ機能等



GameMonkeyにおける“thread”

- スレッドの切り替えタイミングをプログラマが制御
  - 完全な直列実行
  - 同期の問題がない
  - 各スレッドには1フレームに1回制御が移る
- forkも

```
...
fork {
  sleep(1.0f);
  DoSomething();
}
...
```

```
global UpdateObj
  = function(obj)
{
  while(1) {
    obj.draw();
    obj.move();
    yield();
  }
};
...
b = MakeBullet();
thread(UpdateObj, b);
e = MakeEnemy();
thread(UpdateObj, e);
...
```

“thread”の使用例

まとめ

- **SPUはかなり強力**
  - 最小限のチューニングで大幅な高速化
  - 高速なローカルメモリ、並列化
- **RSXのチューニング**
  - **HDR, fragment shader, ...**
- できるだけ多くの仕事をスクリプトでまかなう
  - 生産性の向上＞わずかな速度の犠牲

PixelJunk™ Eden

有限会社　キュー・ゲームス