

リアルタイムアプリケーションへ向けた
粒子法に関する研究（未）

Takahiro Harada

平成 21 年 4 月 9 日

目 次

第 1 章 序論	1
1.1 背景	2
1.2 関連研究	5
1.2.1 コンピュータグラフィックスにおける物理シミュレーション	5
1.2.2 粒子法	6
1.2.3 GPU を用いた高速化	7
1.3 目的	8
第 I 部 映像制作のための粒子法の開発	11
第 2 章 MPS 法流体シミュレーションにおける壁境界の改良	13
2.1 序論	14
2.2 Moving Particle Semi-implicit Method	14
2.2.1 支配方程式	14
2.2.2 非圧縮流体計算モデル	15
2.2.3 粒子間相互作用モデル	15
2.3 壁境界の計算モデル	16
2.3.1 粒子数密度	16
2.3.2 壁の圧力	17
2.3.3 ポアソン方程式	17
2.3.4 圧力項の計算	18
2.3.5 粘性項の計算	19
2.3.6 壁重み関数	20
2.4 距離関数の計算	22
2.5 結果	24
2.6 結論	27
第 3 章 SPH による流体シミュレーションにおける壁境界の改良	29
3.1 序論	30
3.2 関連研究	30

3.3	Smoothed Particle Hydrodynamics	31
3.3.1	支配方程式	31
3.3.2	SPH の原理	32
3.3.3	支配方程式の離散化	33
3.4	格子を用いた近傍粒子探索	36
3.5	壁重み関数	37
3.5.1	密度	37
3.5.2	粘性項	37
3.5.3	圧力項	39
3.5.4	壁重み関数の計算	40
3.6	距離関数の計算	41
3.6.1	手法	41
3.7	結果	43
3.8	結論	44
第 4 章	粒子から滑らかな表面を構築する手法の開発	47
4.1	序論	48
4.2	関連研究	49
4.3	手法	49
4.3.1	陰関数曲面の構築	49
4.3.2	表面の最適化	52
4.4	結果	53
4.5	結論	56
第 5 章	スライスグリッド	59
5.1	序論	60
5.2	関連研究	60
5.3	粒子法シミュレーションに求められる条件	62
5.4	スライスグリッド	63
5.5	結果	65
5.6	結論	71
第 II 部	粒子法シミュレーションの GPU を用いた高速化	75
第 6 章	近傍粒子探索の GPU を用いた高速化	77
6.1	序論	78
6.2	関連研究	79
6.3	ユニフォームグリッドを用いた近傍粒子探索の GPU での実装	79

6.3.1 データ構造	79
6.3.2 グリッド構築	79
6.4 スライスグリッドを用いた近傍粒子探索の GPU での実装 (shader)	82
6.4.1 データ構造	82
6.4.2 スライスのバウンディングボックスの決定	83
6.4.3 先頭番号の計算	83
6.4.4 値の格納と読み出し	84
6.5 スライスグリッドを用いた近傍粒子探索の GPU での実装 (CUDA)	85
6.5.1 データ構造の構築	85
6.5.2 CUDA を用いたデータ構造の構築	87
6.5.3 データへのアクセス	89
第 7 章 応用 (change name)	91
7.1 関連研究	92
7.1.1 流体シミュレーション	92
7.1.2 剛体シミュレーション	92
7.1.3 布シミュレーション	92
7.1.4 連成シミュレーション	94
7.1.5 GPU を用いた流体シミュレーション	95
7.2 Distinct Element Method	96
7.2.1 Distinct Element Method	96
7.2.2 実装	98
7.2.3 結果	98
7.3 Smoothed Particle Hydrodynamics	104
7.3.1 Smoothed Particle Hydrodynamics	104
7.3.2 実装	104
7.3.3 結果	104
7.4 粒子を用いた剛体シミュレーション	109
7.4.1 剛体計算	109
7.4.2 衝突計算	110
7.4.3 実装	112
7.4.4 結果	113
7.5 流体と剛体の連成計算	117
7.5.1 流体と剛体の連成計算	117
7.5.2 実装	117
7.5.3 結果	118
7.6 流体と布の連成計算	121

7.6.1	流体シミュレーション	121
7.6.2	布シミュレーション	121
7.6.3	流体と布の相互作用	121
7.6.4	実装	126
7.6.5	結果	127
7.7	弾性体シミュレーション	130
7.7.1	変形勾配テンソルを用いた弾性体計算	130
7.7.2	実装	131
7.7.3	結果	132
7.8	撃力を用いる剛体シミュレーション	136
7.8.1	広域衝突検出	136
7.8.2	衝突ペアの組への分解	138
7.8.3	狭域衝突検出	140
7.8.4	衝突解決	141
7.8.5	結果	141
7.9	木構造を用いた広域衝突検出	144
7.9.1	巡回履歴を用いた木構造の巡回	144
7.9.2	実装	145
7.9.3	結果	146
7.10	結論	150
第8章	複数のGPUを用いた粒子法の高速化	151
8.1	序論	152
8.2	関連研究	153
8.3	格子を用いた近傍粒子探索の効率化	155
8.4	計算の分割	156
8.5	データの管理	157
8.5.1	重複のない計算粒子の管理法	158
8.6	格子の選択	163
8.6.1	スライスグリッド	163
8.6.2	スライスグリッドへのソートの導入	163
8.6.3	ブロックトランジションソート	164
8.7	レンダリング	167
8.8	ベンチマーク	168
8.9	結果	168
8.10	結論	173
第9章	結論	177

図 目 次

2.1	壁境界と流体粒子.	16
2.2	壁重み関数.	19
2.3	曲率 κ と角度 θ	21
2.4	2 個のモデルから計算された距離関数. 距離関数は 3 次元 のデータであるが, ある軸に垂直な 1 断面での距離関数を それぞれ表示している. なおこの距離関数は図 2.5 の計算 に用いられた.	21
2.5	複雑な形状を壁境界として用いた計算例. 図 2.4 に用いた 境界の距離関数の 1 部を示す.	22
2.6	壁境界計算モデルの比較. 図左は壁重み関数を用いた計算 結果であり, 図右は壁粒子を用いた計算結果である. 図左 の壁重み関数を用いた計算では壁粒子は用いていないが, 描画のみ行なった.	23
2.7	水柱崩壊シミュレーションにおける壁境界計算モデルの比 較. 図左は壁重み関数を用いた計算結果であり, 図右は壁 粒子を用いた計算結果である.	25
2.8	水柱崩壊シミュレーションにおける壁境界計算モデルの比 較. 図左は壁重み関数を用いた計算結果であり, 図右は壁 粒子を用いた計算結果である.	26
2.9	水柱崩壊シミュレーションにおける壁重み関数を用いた計 算結果と, 壁粒子を用いた計算結果での流体の先端位置の 比較.	27
2.10	水柱崩壊シミュレーションにおける壁重み関数を用いた計 算結果と, 壁粒子を用いた計算結果での水柱の高さの比較. .	28
3.1	SPH のカーネル.	32
3.2	影響半径内における壁粒子の分布.	38
3.3	ドラゴンポリゴンモデルを壁境界として用いた自由表面流 れの計算結果.	38
3.4	ドラゴンポリゴンモデルから計算された距離関数.	42
3.5	ブッダポリゴンモデルを壁境界として用いた自由表面流れ の計算結果.	42

3.6 ガーゴイルポリゴンモデルを壁境界として用いた自由表面流れの計算結果.	45
4.1 液滴を水面に落としたシーン.	48
4.2 マーチングキューブ法での表面のパターン	50
4.3 テーブルによる表面のパターンの表現	50
4.4 マーチングキューブ法で穴が開く例	52
4.5 2次元での計算例. 図中の黒点は粒子位置を示したものである. (a) は濃度分布の表面を抽出したものであり, (b) から (i) の青線と緑線はそれぞれ $t = 1, 2, 3, 4, 5, 6, 7, 8$ での表面 V^t と T^t を示している. パラメータ c の値には 0.3 を用いた.	54
4.6 パラメータ t を変えることによる均等に並んでいる粒子から構築された表面への影響. $c = 0.3$ が用いられ, 図は左からそれぞれ $t = 3, 5, 7$ の結果を示している.	55
4.7 パラメータ c の変化による表面の変化. 青線は V^t であり緑線は T^t である. (a) から (e) に用いたパラメータ c はそれぞれ $c = 0, 0.2, 0.4, 0.6, 0.8$ であり, $t = 2$ での表面を示したものである.	55
4.8 3次元での計算例. 6個の粒子を用いており, (a) はそれらの位置を示したものである. それ以外の図は表面ポリゴンを示している. (b) から (j) の図はそれぞれ $t = 0, 1, 2, 3, 4, 5, 6, 7, 8$ での表面を示している. パラメータ c の値には 0.3 を用いた.	57
4.9 3次元流体シミュレーションへの応用例. 左列の図は濃度球を用いた結果であり, 右列は提案手法を用いた結果である.	58
4.10 本手法を大規模粒子法シミュレーション結果に適用した例.	58
5.1 空間固定されたユニフォームグリッドとスライスグリッド. それぞれメモリに確保される領域は青線で囲まれた領域内である.	64
5.2 空間に存在する格子とそのメモリ配置. ここでは2次元配列のメモリ上での配置を示す. メモリに確保されるボクセルはスライスを一方向に順に格納される.	65
5.3 3次元のDEMによるシミュレーションと, そのときのスライスグリッド. 図右列の緑色の領域がメモリに確保された領域	66

5.4	SPH による流体シミュレーションと、メモリ内を可視化したもの。図下において黒いテクセルは空のボクセルを示し、赤、黄、白色のテクセルはそれぞれ 1, 2, 3 個のデータが格納されている。	69
5.5	SPH による流体シミュレーションと、メモリ内を可視化したもの。図下において黒いテクセルは空のボクセルを示し、赤、黄、白色のテクセルはそれぞれ 1, 2, 3 個のデータが格納されている。	70
5.6	約 4,000,000 粒子を用いた広域の流体シミュレーション。	71
5.7	スライスグリッドと Octree のメモリを確保するボクセル数の比較。	73
6.1	粒子番号の格子への格納	80
6.2	ユニフォームグリッドとスライスグリッドを用いた場合におけるメモリに格納される格子をそれぞれ図左と図右に示す。灰色の領域内がメモリに格納される。	83
6.3	先頭番号の計算は、それぞれのスライスに線を用意し、それらを加算ブレンディングしながら描画することで行なわれる。	84
6.4	3 段階でのバウンディングボックスの計算	87
6.5	Prefix Sum	88
7.1	粒子間に設定されるバネとダンパー。	97
7.2	262,144 粒子を用いた回転する容器内の粉体のリアルタイムシミュレーション。	101
7.3	262,144 粒子を用いた粉体の排出のリアルタイムシミュレーション。	101
7.4	CPU と GPU による計算時間の比較。	102
7.5	検証計算の初期条件。	102
7.6	単精度浮動小数での GPU と CPU での計算精度の比較。	102
7.7	GPU での単精度浮動小数点を用いた場合と、CPU で倍精度浮動小数点を用いた場合の計算精度の比較。	103
7.8	水槽に液滴を落としたシミュレーション結果。	107
7.9	流体を水槽に流入させたシミュレーション結果。	107
7.10	高解像度でのダム崩壊のシミュレーション結果。	108
7.11	粒子を用いた形状表現。	110
7.12	粒子を用いた剛体シミュレーションにおいて用いられるデータ。	112
7.13	16,383 個のチェスのコマのシミュレーション	115

7.14	10,922 個のトーラスのシミュレーション	115
7.15	組合わさったトーラスのシミュレーション	116
7.16	11 個のグラスと流体のシミュレーション	119
7.17	剛体と流体の連成シミュレーション	119
7.18	2 種類のチェスの駒と流体の連成シミュレーション	120
7.19	布のモデル化に用いられた 2 種類のバネ.	122
7.20	二種類のグリッド. 灰色の点線は流体粒子が格納される格子であり, 橙色の点線は布のポリゴンが格納される格子である.	123
7.21	不連続な距離場 (図左) と連続な距離場 (図右).	123
7.22	頂点座標とポリゴンの接続情報のデータ構造. 頂点座標は頂点座標テクスチャに, 接続情報テクスチャには頂点番号を格納する.	126
7.23	ポリゴンにかかる力の計算. 粒子 p が頂点 v_0, v_1, v_2 から構成されるポリゴン t と衝突したとき, 流体粒子に働く力 \mathbf{F} が計算され, 粒子に働く力を格納するテクスチャに書き込まれる. またそれと同時に衝突しているポリゴンの番号 t も書き込まれる. メッシュにかかる力は流体粒子に働く力の反力であるので, 流体粒子にかかる力を 3 頂点に分配することによって計算する.	128
7.24	1 枚の布と流体の連成シミュレーション	129
7.25	2 枚の布と流体の連成シミュレーション	129
7.26	2 枚の布と流体の連成シミュレーション	129
7.27	計算体系	134
7.28	計算時間の比較	135
7.29	非対称な近傍粒子探索. 黒色のセルが灰色のセルを探索する. 例えばセル 4 は, セル 5, 6, 7, 8 を検索する.	137
7.30	衝突ペアの圧縮.	138
7.31	並列での速度の計算が可能な例.	138
7.32	衝突ペアのバッチへの分解.	140
7.33	距離閾値を用いた衝突検出.	141
7.34	計算結果	143
7.35	四分木の巡回. (説明追加)	145
7.36	要素数と木構造の巡回の計算時間	147
7.37	5,000 要素を用いた衝突検出	149
8.1	2 個のプロセッサを用いたときの計算の分割.	156
8.2	プロセッサ 1 からプロセッサ 2 へ送る粒子.	159

8.3 送信する粒子データ. 灰色の領域が p_0 の計算領域内のゴースト領域であり, 図左において p_0 から p_1 に送信する粒子を 2 色で示している. これらの粒子の時刻 t での配置を図右に示す.	161
8.4 ブロックトランジションソートの例.	166
8.5 5GPU を用いた並列でのデータ転送と逐次でのデータ転送のバンド幅の計測.	169
8.6 計算体系. 計算領域は 4 つの領域に分割され, 1GPU は 1 つの領域の計算を行なう.	169
8.7 500,000 粒子のシミュレーション.	170
8.8 2GPU を用いたときの計算時間.	171
8.9 4GPU を用いたときの計算時間.	172
8.10 転送されるボクセル数.	174
8.11 転送時間におけるデータの処理計算の占める割合.	174
8.12 表面粒子だけのレンダリング. 図上は表面粒子のみレンダリングされた結果であり, 図下はある面以前の粒子を描画しなかったもの.	175
8.13 リアルタイムシミュレーションの結果をオフラインでレンダリングした結果.	176

表 目 次

3.1	3種類のポリゴンモデルのポリゴン数と距離関数の計算時間.	42
3.2	総粒子数と1タイムステップにかかった計算時間.	44
3.3	1層の壁粒子数と総粒子数に占める壁粒子の割合.	44
5.1	スライスグリッドを用いた計算におけるユニフォームグリッドを用いた場合とスライスグリッドを用いた場合におけるメモリに格納されるボクセル数の割合(スライスグリッド/ユニフォームグリッド).	72
5.2	128^3 の計算領域における DEM のシミュレーションの計算時間(ミリ秒). ユニフォームグリッドを用いた場合のグリッド構築時間を (a) に示し, スライスグリッドを用いた場合を (b) に示す. また 1 タイムステップにかかった計算時間を (c), (d) に示すが, これらはそれぞれユニフォームグリッドとスライスグリッドを用いた計算時間である. また (e) にメモリに確保された最大のボクセル数を示す.	72
5.3	256^3 の計算領域における SPH のシミュレーションの計算時間(ミリ秒). (a), (b) はそれぞれユニフォームグリッド, スライスグリッドを用いた場合の 1 タイムステップの計算時間.	72
5.4	GPU(a) と CPU(b) でのスライスグリッドの構築の時間の比較.	72
7.1	計算時間の詳細(ミリ秒). Time(a), (b), (c) はそれぞれ格子の生成, 衝突検出, そしてその他の計算にかかる時間である.	101
7.2	GPU と CPU による計算時間の比較. Time(a) が GPU, Time(b) が CPU での計算時間である.	101
7.3	Time (a) は格子の生成にかかる計算時間であり, Time (b) は 1 タイムステップの計算とレンダリングにかかる計算時間(ミリ秒).	106

7.4	CPU での計算時間(ミリ秒)と CPU と GPU での計算時間の比.	106
7.5	チェスの駒を用いた計算時間とフレームレート(ミリ秒).	114
7.6	トーラスを用いた計算時間とフレームレート(ミリ秒).	114
7.7	流体粒子数とフレームレート.	118
7.8	流体粒子数とフレームレート.	128
7.9	計算時間(ミリ秒).	134
7.10	計算時間(ミリ秒). なお Time(a),(b),(c),(d) はそれぞれスタックあり(CPU), 巡回履歴(CPU), スタックあり(GPU), 巡回履歴(GPU)を用いた結果.	147
8.1	計算時間の比較(ミリ秒).	168
8.2	レンダリングのフレームレートの比較(フレーム/秒). Solid では全ての粒子がレンダリングされており, surface では表面の粒子のみレンダリングされている.	174

第1章 序論

1.1 背景

近年、数値シミュレーションが工学以外の分野にも用いられるようになってきた。一例として映画などの映像制作である。我々の身近には実写ではないコンピュータグラフィックスを駆使した映像が多く存在するようになった。映像制作のため様々な研究者が現実的なライティングの計算が可能な手法を研究しており、それらの結果としてCGの映像は実写に近づいてきている。実写に近いライティングが可能になり、複雑なシーンも作成されるようになってきた。しかし今度は別の問題が浮上してきた。それは物体の挙動である。現実世界の中の物体の挙動はある物理法則に基づいているのだが、映像制作では物体の挙動も手作りであり、必ずしもその結果は物理法則に基づいているわけではないので、不自然に見えるという問題が生じる。そこでコンピュータグラフィックスの分野でも、数値シミュレーションを用いて物体の挙動の計算が行なわれるようになった。しかしコンピュータグラフィックスで必要な数値シミュレーションは工学分野で必要な数値シミュレーションとは性質が異なる。工学分野では計算精度に重点が置かれるのに対して、コンピュータグラフィックスの分野では計算精度よりも計算速度や計算の効率性などに重点が置かれる。そのため、コンピュータグラフィックスに適した数値シミュレーションの研究がコンピュータグラフィックスの分野において行なわれてきた。

映画などで数値シミュレーションがある程度用いられるようになってきた後、映像制作でもまた異なる分野に数値シミュレーションが使われ始めてきた。それはゲームなどのリアルタイムアプリケーションであり、ここでの数値シミュレーションへの必要条件は映画等での必要条件とは異なる。一般的に映画などの数値シミュレーションはある程度時間をかけて行なわれ、その結果の画像を連続再生することにより映像が作られる。そのためここで用いられる数値シミュレーションはオフラインシミュレーションと呼ばれている。しかしリアルタイムアプリケーションではユーザのインタラクションが一意に決まっておらず、このようにあらかじめ計算しておいた解を全ての状況において用いることはできない。ここで必要とされるのはあるユーザの入力を計算条件に取り入れその場で数値シミュレーションを行なうことである。リアルタイムアプリケーションでは一般的に30frame per second(fps)程度の周波数で画像が進行しているため、1枚の画像を製作するのにかけられる時間は30fpsでは約33ミリ秒である。この時間内に数値シミュレーションを行なわなければならない。しかしこの時間内でシーンのライティングなども行なわなければならないため、この時間を全て数値シミュレーションに用いることはできなく、さらに計算への必要条件は厳しくなる。このように映画などで目にするコンピュータグラフィックスとゲームなどで目にするコンピュータグラフィックスは類

似しているが、これらにおける数値シミュレーションへの必要条件は全く異なる。リアルタイムアプリケーションで用いられる数値シミュレーションはリアルタイムシミュレーションという。

現在、リアルタイムアプリケーションでは剛体シミュレーションは多く用いられている。剛体シミュレーションの計算コストは比較的低く、数百から数千の剛体ならば十分リアルタイムで計算可能であり、その程度の数の剛体があればゲームなどでは十分なことが多い。それに比べ流体シミュレーションなどは一般的に剛体シミュレーションに比べると計算コストが高く、リアルタイムに計算することが難しい。流体は計算空間を格子に分割して解く格子法が伝統的な方法であり、コンピュータグラフィックスの分野でも採用されてきた。しかし我々が慣れ親しんでいる水などの自由表面流れを計算する場合、格子法では流体の速度以外にその表面を追跡しなければならない。特にリアルタイムアプリケーションではその時間制約ゆえにオフラインシミュレーションほどの計算解像度を取ることができないため、オフラインシミュレーションに比べると低い解像度の計算格子で計算を行なわなければならない。しかしここに問題が発生する。それは格子法では移流を解く際に、数値拡散が発生してしまう。この数値拡散は格子が粗いと如実に現れる。さらに自由表面の追跡においても数値拡散が生じるため、低解像度の格子での計算では流体の体積が簡単に減少してしまう。これは工学的にも大きな問題であるが、映像制作においても大きな問題である。このような問題を抱えていることから、格子法の流体シミュレーションを自由表面流れをリアルタイムアプリケーションで実現することは困難である。

流体の計算手法として、格子法以外に粒子法がある。粒子法では計算空間を分割するかわりに流体を粒子の集合として表し、支配方程式を粒子間相互作用に書き下して粒子の挙動を計算する手法である。粒子法を用いた流体シミュレーションは格子法とは異なり、粒子自体が移動することによって移流計算を行なうため、移流による数値拡散が生じないという利点がある。これは粒子法の性質であり、計算解像度によらないため、低解像度の数値シミュレーションでも流体の体積が減少するという問題はない。さらに前述の通り粒子自体が流体の存在を表すため、自由表面流れの計算を行なう場合に自由表面の追跡を行なう必要もない。そのため、粒子法における流体シミュレーションはリアルタイムアプリケーションに適していると考えられる。しかしながら問題は存在する。それは粒子法を用いて流体シミュレーションを行なうときには十分な数の粒子を用いて計算しなければ視覚的に満足できる結果を得ることはできない。同じ計算要素数であれば粒子法は格子法よりも計算コストが高く、本研究の開始時点ではリアルタイムで計算できる粒子数が少なく、結果は十分と言える物ではなかつ

た。リアルタイムアプリケーションに適している可能性がある粒子法をどのように高速化し、視覚的に十分な粒子数をリアルタイムでシミュレーションするのかという課題が存在する。ここに本研究が位置しており、本研究では粒子法を高速化し、リアルタイムで計算可能な粒子数を増加させることによって、粒子法が”リアルタイムアプリケーションに適している可能性”があるという段階から、”リアルタイムアプリケーションに用いることができる”という段階に引き上げる。

もう1つの研究の背景として、現在のプロセッサテクノロジがある。最近まで Central Processing Unit(CPU) は周波数を年々増加させその演算性能を向上させてきた。CPU の周波数は 4GHz 近くまで到達したが、周波数增加による性能向上は突然終焉を迎える。プロセッサテクノロジは演算の効率性を向上させる方向に進み始めた。現在は周波数を増加させるのではなく、複数のプロセッサを 1CPU に搭載させ、並列で演算をすることにより演算性能を向上させるという方向に向かっている。CPU がこの方向に向かい始めたのはごく最近であるが、グラフィックスの描画専用プロセッサである Graphics Processing Unit(GPU) はその前から複数のコアを 1GPU 上に搭載し、並列で演算を行なうことによって高い演算性能を出すように設計されていた。そして現在でも 100 を超えるプロセッサが 1GPU 上に搭載されており、そのピーク演算性能は 1TFLOPS を超える。しかしこの GPU の高い演算性能は CPU が搭載している多くの自由度を削除し、プロセッサを単純化することで得られた物であり、GPU は CPU と同じ処理を高速に行なうことができるわけではない。GPU をグラフィックスの描画以外の用途に用いて様々なアプリケーションを高速化するという研究も行なわれてきた。並列度の高いプロセッサは GPU だけではない。Playstation 3 にも搭載されている Cell Broadband Engine(Cell BE) Architecture は 1 個の PPU と 8 個の SPU という計算ユニットが搭載されており、合計で 9 個のコアが搭載されている。このようにマルチコアプロセッサはすでに我々のすぐ近くに多く存在している。このようなマルチコアプロセッサの能力を引き出して利用するためには、従来の 1 プロセッサ用に開発していたアルゴリズムでは困難な場合が多い。そこでこれらにあわせたアルゴリズムの開発が必要である。

格子法の流体シミュレーションは多くの場合に、計算要素間の接続が固定されており、さらにそれぞれの要素の演算は独立しているため並列化しやすい。そのため GPU を用いて高速化することが容易である。数値シミュレーションでは格子法の流体シミュレーションの他にも布のシミュレーションなどが GPU を用いて高速化されてきたが、これらの共通点は要素間の接続が固定されており、計算中に動的に変化することはない。この観点から粒子法シミュレーションを見ると、ある粒子の物理量はその近

傍粒子の物理量から計算されるため、その粒子とその近傍の粒子間に接続が確立されていると考えることができる。しかし粒子法を用いた流体シミュレーションなどにおいては粒子が自由に動き回るため、ある粒子の近傍粒子も動的に変化する。すなわち、計算要素間の接続がそれぞれのタイムステップにおいて変化するということである。このような動的に要素間の接続が動的に変化するような問題を GPU で解いた研究はない。そこで本研究では粒子法を GPU を用いて高速化し、GPU は静的な接続を持つ計算以外の、粒子法のような動的に接続を変化させる数値シミュレーションにも用いることができることを示す。そして粒子法シミュレーションの計算を GPU 上で実装可能な並列な計算にして、GPU を用いて高速化することによってリアルタイムアプリケーションに用いることのできる粒子法シミュレーション手法を開発する。

1.2 関連研究

1.2.1 コンピュータグラフィックスにおける物理シミュレーション

コンピュータグラフィックスの分野において剛体シミュレーションは古くから研究されており、映像制作などへのオフラインシミュレーションからリアルタイムシミュレーションまで幅広く行なわれている。近年では Guendelman らが形状表現に符号付き距離関数を用い、さらに剛体が積み重なる場合でも安定に解くことができる手法を発表している [43]。しかし実際剛体シミュレーションに関してはアカデミックよりも産業業界の方が先進的である。剛体シミュレーションの様々なソフトウェアが開発されており、それらはこの論文で議論されているような多数の剛体が積み重なるようなシミュレーションもリアルタイムで安定に解くことができる。しかしゲームなどに用いられている剛体シミュレーションエンジンでも数千から数万の大規模な剛体シミュレーションはリアルタイムでは難しく、さらなる高速化については研究の余地がある。Havok は GPU で狭域衝突検出と時間積分を高速化した Havok FX を開発し、剛体シミュレーションを高速化したが、広域衝突検出は CPU で行なっているため、GPU の本来の性能を完全に引き出せていない可能性がある [16]。

布のシミュレーションに関しては布を粒子とバネで表現したシミュレーション手法が多く用いられてきた [103]。そして数値的に安定に解くために陰解法が導入された [11]。また粒子とバネによる布は布の性質がメッシュの解像度に大きく依存するため、Finite Element Method(FEM) を応用した手法も研究されている [33]。リアルタイムアプリケーション、特にゲームでは計算の速度だけでなく計算の安定性も重要であるため、どのような条件下でも安定に解くことができる Position-based の手法が開発された

[95]. 布のシミュレーションだけならばリアルタイムアプリケーションにも十分応用可能なレベルまで到達しているが、流体との連成などの問題のリアルタイムでの解法は研究されていない [44].

また流体シミュレーションは計算格子を用いるオイラー解法が多く研究されており、自由表面の追跡にはレベルセット法に関する研究が多い [37]. 計算時間が重要なコンピュータグラフィックスの分野では大きなタイムステップを刻めるセミラグランジュ手法が移流計算に多く用いられてきたが、数値拡散が激しいため、細かい流体の挙動の計算ができない [112]. そのため、より高精度の移流スキームである CIP, BFECC, MacCormak などが研究されて、流体のオフラインシミュレーションは高精度化に向かっている [24]. また同じように計算精度を上げるため非構造格子を用いた研究も行なわれており、動的にリメッシングする研究までも行なわれてきた [72]. さらに応用として流体と剛体の相互作用や粘弾性流体、薄膜との相互作用、混相流、2次元と3次元のシミュレーションとのカップリングなど様々な研究が行なわれてきている [19, 39, 44, 82, 67]. しかしこれらは全てオフラインシミュレーションであり、リアルタイムアプリケーションに用いることはできない.

1.2.2 粒子法

粒子法は計算対象を粒子として離散化し、支配方程式を粒子間相互作用にして解く手法である。大きな特徴として粒子間には固定された接続を必ずしも作る必要がないため、流体計算などでは粒子自体が自由に動くことができる。主な粒子法としては Smoothed Particle Hydrodynamics(SPH) と Moving Particle Semi-Implicit(MPS) 法がある [83, 76]. SPH は天文学の分野で開発された後、流体シミュレーションに適用され、様々な研究が行なわれている [89]. 基本的に SPH は圧縮性流体を解く手法であり、全ての項が陽的に解かれる。非圧縮流体を解くときには圧縮性を下げることによって非圧縮に近い状態にして解いている [25, 30]. また MPS 法は非圧縮流体を解くために開発され、圧力項を陰的に解き、それ以外の項は陽的に解く半陰解法である。粒子法では一般的に計算領域に壁境界が存在した場合に、その壁を粒子化し、固定粒子として解くことで境界の計算を行なっている。しかし、壁粒子を用いると境界の形状を正確に表現することができないという問題がある。また粒子法は界面を特別に追跡する必要がないため、自由表面流れの計算に向いているが、計算粒子から流体の界面を定義することが難しい。濃度球を用いて粒子から表面を構築する手法が多く用いられてきたが、表面に不自然な凹凸が残ってしまうという問題がある [14]. 様々な研究が行なわれているが流体の平な面などを表現することは難しい [129].

Premoze らは MPS 法をコンピュータグラフィックスの分野に持ち込んだが、MPS 法は半陰解法であり陽解法の SPH と比べると計算コストが高いため、その後 MPS 法はあまりコンピュータグラフィックスの分野では研究されていない [102]。一方、SPH はその低い計算負荷のため、コンピュータグラフィックスの分野で比較的多く研究されている [94]。また粒子法は低解像度の計算でも流体の質量が保存性が良いため、リアルタイムアプリケーションへの応用を考えた研究も行なわれている。しかしこれらの研究で用いられてきた計算解像度ではリアルタイムアプリケーションにおいても不十分である。そこでより高速でより多くの粒子をリアルタイムで計算することのできる手法が必要とされている。また粒子間相互作用を計算するために、近傍粒子探索を行なわなければならないが、一般的にユニフォームグリッドを用いた手法が用いられている。しかしこの手法の欠点はメモリ効率が悪い。より高速に計算ができるようになるということは、より多くの粒子を同じ計算時間で扱えるようになるということであり、粒子数が大幅に増加した場合には、このメモリ効率の問題点が浮き彫りになってしまう。よって高速な粒子法シミュレーションでは、計算効率を落とさずによりメモリ効率の良い近傍粒子探索手法が必要になる。

1.2.3 GPU を用いた高速化

GPU はグラフィックスの処理に特化したプロセッサであり、複数の計算ユニットを内部に多数並べた構造になっており、それらが並列に処理を行なうことで高い演算性能を出すことができる。グラフィックスの主な処理は頂点の座標変換と、ピクセルのフィルである。コンピュータグラフィックスで用いられるモデルの頂点数は増加し続け、数千頂点ほどからなるモデルも特別ではなくなってきた。あるモデルを画面に表示するためには、モデル自体の座標系からワールド座標系に全ての頂点を座標変換しなければならない。この座標変換はベクトルと行列の乗算であり、これを多くの頂点に対して行なう。この処理を行なうのがバーテックスユニットである。ここでそれぞれの頂点の処理は他の頂点の処理に依存しないため、並列で行なうことが可能ならばより効率的である。また座標変換された頂点から形状が構成され、次の処理はこの形状に囲まれた領域のピクセルの色の計算である。この処理を行なうのがフラグメントユニットである。この計算もあるピクセルの計算は他のピクセルの計算に依存しないため、並列で処理した方が効率が良い。このようにグラフィックスの処理の多くが完全に互いに依存性のない処理であるため、GPU は多くの計算ユニットを並列に並べるという機構になっている。GPU の高い演算性能は、グラフィックス以外のアプリケーションにとっても有用であると考えられ、様々な研究が行なわれてきた [100]。物理シミュレーションに近い研究と

しては、Coupled Map Lattice(CML)の研究からはじまり、2次元の格子での流体シミュレーション、そして3次元の格子での流体シミュレーションも研究されてきた[62, 59, 24]。さらに布や、粒子間相互作用のない粒子のシミュレーションも行なわれてきた。しかしこれらの研究の共通点は計算要素間の接続が固定されている、もしくは存在しないということである。粒子法のように動的に接続が変化する問題に対しては解決策は見いだされておらず、GPUがそれらの計算に適しているかも未知である。

1.3 目的

本論文は大きく分類して3部にわかれる。まず第一部ではリアルタイムアプリケーションを含めた映像制作のための粒子法を開発する。具体的には以下の4手法を提案する。

- MPS法流体シミュレーションにおける壁境界の改良手法。この手法では半陰解法であるMPS法において壁境界を粒子として取り扱わずに壁形状からその粒子に与える影響を求めて計算の境界条件とする。
- SPHによる流体シミュレーションにおける壁境界の改良手法。この手法では陽解法であるSPHにおいて壁境界を粒子として取り扱わずに壁形状からその粒子に与える影響を求めて計算の境界条件とする。
- 粒子からの表面構築の改良手法。この手法を用いることによって計算粒子からメタボールを用いて計算する表面に後処理を行い、より自由度の高い流体表面を構築することができ、薄い膜や鋭い表面を構築することができるようになる。
- 近傍粒子探索を効率化するためのデータ構造であるスライスグリッド。この手法を用いることによって計算負荷をあまり増やさずにメモリ効率を上げ粒子法シミュレーションのボトルネックの1つである近傍粒子探索を効率化することができる。

また第二部ではGPUを用いた粒子法の高速化手法を提案する。粒子法シミュレーションをGPUで高速化するための鍵となる部分は粒子の近傍探索である。第二部ではまず近傍粒子探索のGPUを用いた高速化手法を提案する。そしてこの手法の様々な応用について述べる。本研究で行なった応用は以下のものである。

- DEMの高速化
- SPHの高速化

- 粒子を用いた剛体シミュレーションの高速化
- 流体と剛体の連成計算
- 流体と布の連成計算
- 弹性体シミュレーションの高速化
- 撃力を用いる剛体シミュレーションの高速化

そしてさらにこれらを拡張し複数の GPU を用いた粒子法シミュレーションの高速化手法について第三部で提案する。

第I部

映像制作のための粒子法の開発

第2章 MPS法流体シミュレーションにおける壁境界の改良

2.1 序論

粒子法は自由表面流れの解析をはじめとして様々な計算に応用されてきた[132]. 例えは、船舶への海水打ち込みの解析や[110, 114], マイクロ流体解析[57], 弹性体[133], 剛体[118, 46]の解析などである.

流体解析を行なう粒子法の主な手法としては Moving Particle Seim-Implicit(MPS)法[130, 131]とSmoothed Particle Hydrodynamics(SPH)[83, 89]がある. 圧力項を陰的に解くMPS法は、全ての項を陽的に解くSPHよりも計算コストは高い. 近年ではSPHでもMPS法のように陰的に圧力項を解く手法も研究されている[25, 30]. 粒子法では壁境界を扱うときは、一般的に壁境界を粒子に変換して計算を行なう. MPS法において壁粒子は2種類存在する. 1つは圧力を流体と同様に計算する壁粒子であり、もう1つは圧力計算粒子の粒子数密度の低下を防ぐために存在する粒子である. 計算体系にもよるが壁粒子は全体の粒子数の大きな割合を占め、圧力のポアソン方程式の行列を大きくしている. 壁粒子を圧力のポアソン方程式に組み込まずに計算することが可能ならば計算コストは下がり、計算の高速化が可能になる. その結果としてより高解像度の計算を行なうことができる. また壁粒子を用いるもう1つの欠点としては斜面を表現するときに滑らか斜面を表現できず、計算結果に影響を与えててしまう. 本研究では壁粒子を用いず、ポリゴンを壁境界として用いる境界の計算手法を開発する. 流体粒子の影響半径内に存在する壁の部分の寄与を計算し、それを圧力のポアソン方程式に組み込む. そして算出された圧力から流体の座標の補正を行なう[56, 137].

2.2 Moving Particle Semi-implicit Method

2.2.1 支配方程式

非圧縮流体の支配方程式は以下に示す質量保存を表す連続の式と、運動量保存を表すナビエストークスの式である.

$$\frac{D\rho}{Dt} = 0 \quad (2.1)$$

$$\frac{D\mathbf{u}}{Dt} = -\frac{1}{\rho}\nabla P + \nu\nabla^2\mathbf{u} + \mathbf{g} \quad (2.2)$$

ここで $\mathbf{u}, P, \rho, \nu, \mathbf{g}$ はそれぞれ流体の速度、圧力、密度、動粘性係数、重力加速度である.

2.2.2 非圧縮流体計算モデル

MPS 法では 2 段階で支配方程式を解く。まず圧力項以外の項を計算し、各粒子の仮の速度を求める。 \mathbf{u}_i^* を第 1 段階で圧力項以外から求めた粒子 i の仮の流速であるとすると圧力項による式は以下のようになる。

$$\frac{\mathbf{u}_i^{n+1} - \mathbf{u}_i^*}{\Delta t} = -\frac{1}{\rho} \nabla P_i^{n+1} \quad (2.3)$$

式 (2.3) の発散を取り $\nabla \cdot \mathbf{u}_i^{n+1}$ の項は発散が 0 であることを用いて以下のように式変形できる。

$$\nabla^2 P_i^{n+1} = \rho \frac{\nabla \cdot \mathbf{u}_i^*}{\Delta t} \quad (2.4)$$

MPS 法の密度の定義を用いて式 (2.4) は以下のように変形することができる。

$$\nabla^2 P_i^{n+1} = -\frac{1}{n_0} \frac{n^* - n_0}{(\Delta t)^2} \quad (2.5)$$

n^* は第 1 段階後に得られた粒子数密度であり n_0 は基準となる粒子数密度である。このポアソン方程式を解くことで圧力 P_i^{n+1} を求め、式 (2.3) により圧力項による速度の補正を行なう。

2.2.3 粒子間相互作用モデル

MPS 法では偏微分方程式を粒子間相互作用モデルを用いて離散化する。これらの粒子間相互作用を重み関数に基づいて計算する。重み関数は次のように定義する。

$$w(\mathbf{r}) = \begin{cases} \left(\frac{|\mathbf{r}_e|}{|\mathbf{r}|} \right) - 1 & (|\mathbf{r}| < |\mathbf{r}_e|) \\ 0 & (|\mathbf{r}| \geq |\mathbf{r}_e|) \end{cases} \quad (2.6)$$

\mathbf{r}_e は粒子 i の影響半径である。粒子 i の粒子数密度 n_i は影響半径内にある粒子 j の位置を用いて以下のように定義される。

$$n_i = \sum_{j \neq i} w(\mathbf{r}_{ij}) \quad (2.7)$$

勾配モデルとラプラシアンモデルはそれぞれ以下のように与える。

$$\langle \nabla \phi \rangle_i = \frac{d}{n^0} \sum_{j \neq i} \frac{\phi_j - \phi_i}{|\mathbf{r}_{ij}|^2} \mathbf{r}_{ij} w(\mathbf{r}_{ij}) \quad (2.8)$$

$$\langle \nabla^2 \phi \rangle_i = \frac{2d}{n^0 \lambda} \sum_{j \neq i} (\phi_j - \phi_i) w(\mathbf{r}_{ij}) \quad (2.9)$$

d は空間の次元数であり、 λ は統計的な分散の増加を解析解と一致させるための係数である。これらを用いて式 (2.2) を離散化して解く。

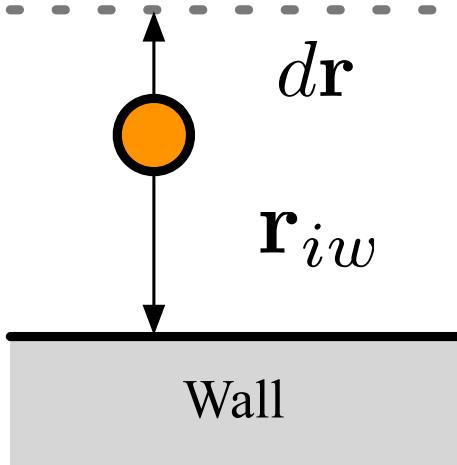


図 2.1: 壁境界と流体粒子.

2.3 壁境界の計算モデル

まず粒子数密度のモデル化を行なう。MPS法において壁境界が流体粒子に影響を及ぼすのは、密度、圧力項、そして粘性項である。そのため、新しい壁境界計算モデルでは、これらをモデル化しなければならない。以下でそれらのモデル化を順番に述べていく。

2.3.1 粒子数密度

近傍粒子に壁境界を含む場合の流体粒子 i の粒子数密度 n_i は、近傍の流体粒子による粒子数密度と壁境界による粒子数密度の寄与に分けられる。

$$\begin{aligned} n_i &= \sum W(\mathbf{r}_{ij}) \\ &= \sum_{j \in fluid} W(\mathbf{r}_{ij}) + \sum_{j \in wall} W(\mathbf{r}_{ij}) \end{aligned} \quad (2.10)$$

この壁境界による粒子 i の粒子数密度への寄与を $\sum_{j \in wall} W(\mathbf{r}_{ij}) = Z(\mathbf{r}_{iw})$ とし、壁重み関数とよぶ。壁重み関数は $|\mathbf{r}_w| \geq |\mathbf{r}_e|$ の時には 0 である。流体粒子の密度は、近傍の流体粒子の寄与を計算したあとに、この壁重み関数による値を足し合わせることで計算される。壁重み関数の具体的な求め方は 2.3.6 節において説明する。

2.3.2 壁の圧力

壁による圧力項はその物理的意味を考慮に入れてモデル化する。壁が流体粒子に及ぼす圧力による力は流体粒子と壁境界との距離が平均粒子間距離以下になったときに、流体粒子を平均粒子間距離に離すように動かせる。よって壁境界が流体粒子 i に及ぼす圧力による力は以下のようになる。

$$\begin{aligned} d\mathbf{r} &= -\frac{1}{\rho} \nabla P dt^2 \\ &= -\frac{dt^2}{\rho} \left(\frac{d}{n_0} \frac{P_{wall} - P_i}{|\mathbf{r}_{iw}|^2} \mathbf{r}_{iw} W(\mathbf{r}_{iw}) \right) \end{aligned} \quad (2.11)$$

なお $d\mathbf{r}$ と \mathbf{r}_{iw} はそれぞれ、壁近傍の流体粒子が圧力項によって移動すべきベクトルと、流体粒子座標から壁境界までのベクトルであり、図 2.1 に示すように定義し、 $|d\mathbf{r} - \mathbf{r}_{iw}|$ は初期粒子間距離である。式 (2.11) は壁による圧力勾配を粒子 i が受け Δt で、その粒子が壁から初期粒子間距離に離れる力である。式 (2.11) を変形すると流体粒子 i が近接している壁の圧力は以下のように求めることができる。

$$P_{wall} = P_i - \frac{\rho n_0 |\mathbf{r}_{iw}|^2}{dt^2 dW(\mathbf{r}_{iw})} \frac{d\mathbf{r}}{\mathbf{r}_{iw}} \quad (2.12)$$

これを用いて圧力のポアソン方程式を組み立て、圧力勾配による力の計算を行なう。

2.3.3 ポアソン方程式

式 (2.12) を用いて影響半径内に壁が存在する流体粒子 i の圧力のポアソン方程式を組み立てる。MPS 法における圧力のポアソン方程式は

$$\nabla^2 P = -\frac{\rho}{dt^2} \left(\frac{n^* - n^0}{n^0} \right) \quad (2.13)$$

であるが、影響半径内に壁が存在する場合は壁境界の寄与を考慮に入れなければならない。

まず式 (2.13) の左辺を考える。影響半径内に壁が存在する場合は式 (2.13) の左辺は流体粒子による圧力のラプラシアンと壁境界による圧力のラプラシアンに分解することができる。つまり

$$\nabla^2 P = (\nabla^2 P)_{fluid} + (\nabla^2 P)_{wall} \quad (2.14)$$

となる。

式(2.12)を用いて、影響半径内の壁の圧力が一定だとすると、壁境界による圧力のラプラシアンは以下のように表すことができる。

$$\begin{aligned} (\nabla^2 P)_{wall} &= \frac{2d}{\lambda n_0} (P_{wall} - P_i) Z(\mathbf{r}_{iw}) \\ &= \frac{2\rho}{\lambda dt^2} |\mathbf{dr}| |\mathbf{r}_{iw}| \end{aligned} \quad (2.15)$$

ここで $Z(\mathbf{r}_{iw}) = \sum_{j \in wall} W(\mathbf{r}_{ij})$ であり、流体粒子 i の影響半径内の壁の寄与を表す壁重み関数である。

また式(2.13)の右辺も流体粒子による粒子数密度の寄与と、壁境界による粒子数密度の寄与に分けられる。圧力項以外の項を解いたあとの粒子数密度 n^* のうち、流体粒子と壁境界による粒子数密度の寄与をそれぞれ n_{fluid}^*, n_{wall}^* とすると、式(2.13)の右辺は、

$$-\frac{\rho}{dt^2} \left(\frac{n^* - n^0}{n^0} \right) = -\frac{\rho}{dt^2} \left(\frac{n_{fluid}^* + n_{wall}^* - n^0}{n^0} \right) \quad (2.16)$$

と表される。 n_{wall}^* は、圧力項以外の項を解いて座標を更新したあとの粒子座標における壁の粒子数密度への寄与分であるため、式(2.10)と同様に $\sum_{j \in wall} W(\mathbf{r}_{ij})$ と求めることができる。

これらを用いて近傍に壁境界が存在する流体粒子の圧力のポアソン方程式を組み立てるが、式(2.15)は未知数の圧力がふくまれていない。すなわちこの式を用いると壁による圧力のラプラシアンは定数項にすることができるということである。壁の圧力のラプラシアンを式(2.13)に代入し、壁の圧力のラプラシアンを式(2.13)の右辺に移項すると、式(2.13)は以下のように変形することができる。

$$\nabla^2 P = -\frac{\rho}{dt^2} \left(\frac{n_{fluid}^* + n_{wall}^* - n^0}{n^0} \right) - \frac{\rho}{dt^2} \frac{|\mathbf{dr}|}{|\mathbf{r}_{iw}|} \mathbf{r}_{iw} \quad (2.17)$$

このポアソン方程式は、壁粒子を用いた場合と異なる。それは壁重み関数を用いた壁境界計算手法だと壁粒子の圧力を求める必要がないため、壁粒子の圧力の行が存在しない。よって圧力計算を行なう壁粒子を用いたポアソン方程式と比較すると小さな連立方程式になるため、計算コストが低くなる。

2.3.4 圧力項の計算

ポアソン方程式を解くことによって求められた圧力を用いて流体粒子の座標の補正を行なう。圧力勾配も流体粒子と壁粒子による寄与に分解することができる。

$$\nabla P = (\nabla P)_{fluid} + (\nabla P)_{wall} \quad (2.18)$$

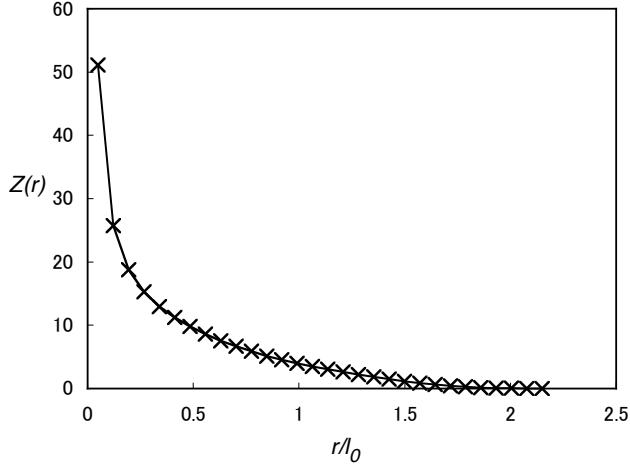


図 2.2: 壁重み関数。

壁による圧力勾配は式 (2.12) を用いて

$$\begin{aligned} (\nabla P)_{wall} &= \frac{d}{n_0} \left(\frac{P_{wall} - P_i}{|\mathbf{r}_{iw}|^2} \mathbf{r}_{iw} Z(\mathbf{r}_{iw}) \right) \\ &= \frac{\rho}{dt^2} \frac{d\mathbf{r}}{d\mathbf{r}_{iw}} \mathbf{r}_{iw} \end{aligned} \quad (2.19)$$

と計算することができる。ここで Z は影響半径内の壁境界の寄与を表す壁重み関数である。

2.3.5 粘性項の計算

粘性項の計算も以下のように流体粒子による寄与と壁境界による寄与に分けることができる。

$$\nabla^2 \mathbf{u} = (\nabla^2 \mathbf{u})_{fluid} + (\nabla^2 \mathbf{u})_{wall} \quad (2.20)$$

ここで壁境界の持つ速度が一定だとすると、壁境界の粘性項への寄与は、

$$\begin{aligned} (\nabla^2 \mathbf{u})_{wall} &= \frac{2d}{n_0 \lambda} \sum_{j \in wall} (\mathbf{u}_j - \mathbf{u}_i) W(\mathbf{r}_{iw}) \\ &= \frac{2d}{n_0 \lambda} (\mathbf{u}_j - \mathbf{u}_i) Z(\mathbf{r}_{iw}) \end{aligned} \quad (2.21)$$

と求めることができる。

2.3.6 壁重み関数

これまで壁粒子を用いない壁境界計算手法を述べてきたが、この手法を用いるためにはある粒子の影響半径内に存在する壁の寄与を計算する壁重み関数を求めなければならない。粒子*i*と壁までの距離と、壁の曲率が決まれば、影響半径内の壁の割合が決まるため、この壁重み関数はあらかじめ計算しておくことが可能である。

まず壁の曲率が0である場合を考える。粒子*i*から壁までの距離が $|\mathbf{r}_{iw}|$ であるときの壁重み関数はその距離に仮想的に壁を考え、壁内部に粒子を仮想的に配置する。そしてそれらの重みの和を取ることで壁重み関数を求める。粒子*i*から壁までの全ての距離に対してこの壁重み関数を計算することは不可能であるため、粒子*i*から有限個の距離 $r_0, r_1, r_2, \dots, r_n$ において、それぞれ仮想的に壁を生成し、壁重み関数を計算する。この距離以外の点に対しては、求めたこれらの点での壁重み関数を線形補間することによって求める。影響半径 $2.1l_0$ (ただし l_0 は初期粒子間距離)の場合において、その影響半径内の30個の位置での壁重み関数を求め、線形補間を行なった結果を図2.2に示す。

壁境界が曲率を持っている場合は曲率 κ が0であるときの壁重み関数の値を基準として、近似的に曲率を用いて計算する。まずは曲率 κ から影響半径内において壁が成す角度 θ を計算する。

$$\kappa = \frac{1}{R} = 2 \cos(\theta/2) \quad (2.22)$$

$$\theta = 2 \arccos(\kappa/2) \quad (2.23)$$

なお角度 θ は図2.3に示すように定義する。そして θ を用いて曲率を考慮した壁重み関数 Z_{cur} は、 $\kappa = 0$ の時の壁重み関数 Z を用いて、

$$Z_{cur}(\mathbf{r}) = \frac{\theta}{\pi} Z(\mathbf{r}) \quad (2.24)$$

と求める。

粒子から壁までの距離

壁重み関数を用いた壁境界計算手法を用いるためには、ある粒子から壁までの距離を求めなければならない。粒子から壁までの距離とはすなわち、壁を構成する全てのポリゴンまでの距離のうちの最小値である。この計算において各粒子から壁を構成する全てのポリゴンまでの距離を求めるのは、計算コストが高いため、本研究では距離関数を導入した。距離関数とは形状を表現するための1手法である。ここで S を壁境界とすると、距離関数 D は空間内の全ての点 \mathbf{p} に対し、

$$D(\mathbf{p}) = \min(d(\mathbf{p}, \mathbf{q}) | \mathbf{q} \in S) \quad (2.25)$$

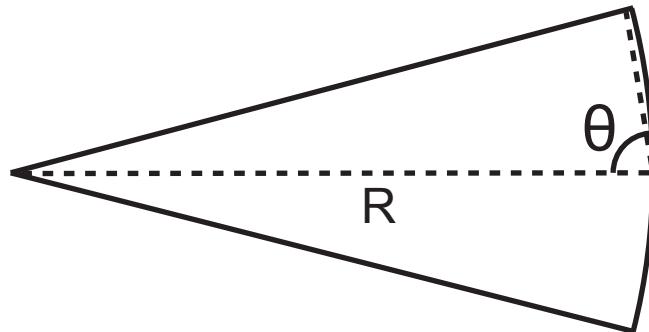
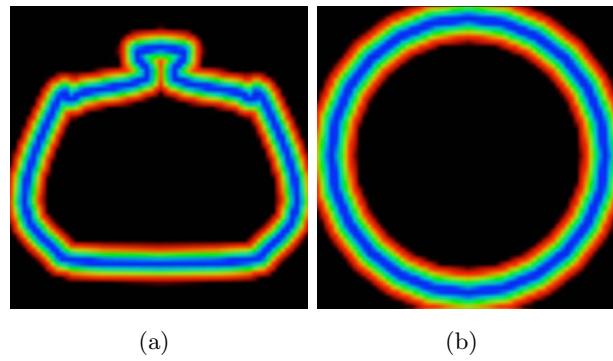
図 2.3: 曲率 κ と角度 θ .

図 2.4: 2 個のモデルから計算された距離関数。距離関数は 3 次元のデータであるが、ある軸に垂直な 1 断面での距離関数をそれぞれ表示している。なおこの距離関数は図 2.5 の計算に用いられた。

と定義される。ここで $d(\mathbf{p}, \mathbf{q})$ は \mathbf{p}, \mathbf{q} の距離である。距離関数の具体的な計算手法は後述する。

曲率の計算

距離関数は空間のある点から境界までの距離を容易に求めることができるだけでなく、その点での曲率も容易に求めることができるという特徴を持つ。ある位置 \mathbf{x} における平均曲率 κ は距離関数 D を用いて、

$$\kappa = \frac{1}{2} \nabla^2 D(\mathbf{x}) \quad (2.26)$$

と求めることができる。

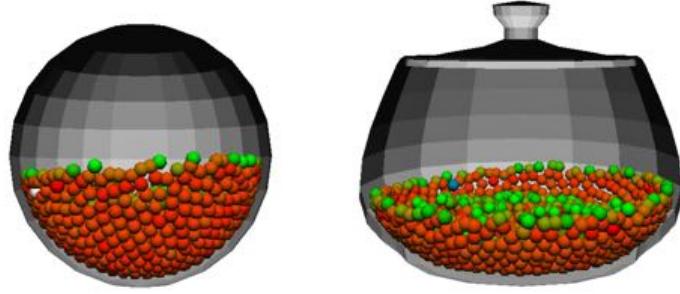


図 2.5: 複雑な形状を壁境界として用いた計算例。図 2.4 に用いた境界の距離関数の 1 部を示す。

2.4 距離関数の計算

まず計算領域に 3 次元格子を用意し、その格子点において距離関数を求める。境界全体から定義される距離関数は式 (2.25) からわかるように境界がいくつかの小境界に分解されるとき、それらの小境界の距離のうち最も小さな値を選択することで求めることができる。すなわち境界がポリゴンからできているときは、全てのポリゴンからの距離を計算し、そのうちの最小値を求めて境界全体から定義される距離関数を求めることができる。よって 1 個 1 個のポリゴンによる距離関数をそれぞれ求め、各格子点でそれらの最小値を取ることで境界全体からの距離を求めた。

空間のある点での距離関数は、その点から境界上の最近点までの距離であるため、1 個のポリゴン上での最近点はそのポリゴンの頂点、辺、そして面の 3 つのうちのいずれかの上に存在する。距離関数はポリゴンから影響半径の距離までの格子点で必要であり、全ての格子点において求める必要はない。そのため、まずは 1 個のポリゴン i の XYZ 軸方向の最小値 $(x_{i,min}, y_{i,min}, z_{i,min})$ と最大値 $(x_{i,max}, y_{i,max}, z_{i,max})$ を求め、距離関数を求めるバウンディングボックスを定義する。壁境界が流体粒子に影響を及ぼす距離を ϵ とすると、距離関数を計算する領域 $\mathbf{r} = (x, y, z)$ は、 $x_{i,min} - \epsilon < x \leq x_{i,max} + \epsilon, y_{i,min} - \epsilon < y \leq y_{i,max} + \epsilon, z_{i,min} - \epsilon < z \leq z_{i,max} + \epsilon$ となる。このバウンディングボックス内においてまず、頂点からの距離を計算する。そして辺からの距離を計算するが、辺からの距離を計算する領域はバウンディングボックス内でも、その点から辺に下ろした垂線の足が辺上にある領域のみである。同様に面からの距離も計算するが、この計算を行なう領域も、点から下ろした垂線の足が面上にある領域のみである。

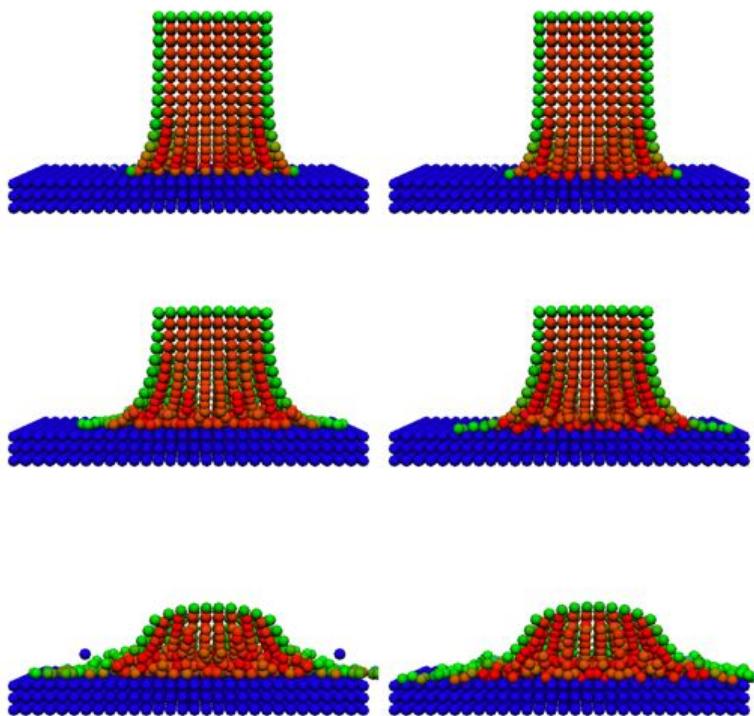


図 2.6: 壁境界計算モデルの比較. 図左は壁重み関数を用いた計算結果であり, 図右は壁粒子を用いた計算結果である. 図左の壁重み関数を用いた計算では壁粒子は用いていないが, 描画のみを行なった.

2.5 結果

まず、2つの形状のポリゴンデータから距離関数を計算したものを図2.4に示す。距離関数は3次元空間内で計算されているが、この図では3次元の距離関数をZ軸に垂直な1平面上での2次元の距離関数を表示しているものであり、境界からの距離に応じてセンター表示されている。距離が大きくなると青から赤に色が変わっている。この距離関数を境界条件として用いて内部に流体粒子を配置し、計算を行なった結果を図2.5に示す。本手法を用いると、このように複雑な形状を持った壁境界条件を用いても計算できていることがわかる。壁粒子を用いた計算手法だと、斜面である境界を正しく計算することができない。それは境界は全て壁粒子に変換されるため、斜面は粒子の大きさの凹凸がある面として計算されてしまうからである。しかし本手法を用いるとそのような問題点は解決できる。壁重み関数は境界からの距離の関数であるため、境界からの距離が同じ点ならば同じ値になる。そのため、傾斜を持つ境界も凹凸のある平面ではなく、スムーズな傾斜を持つ境界として計算することができる。

次に本手法と壁粒子を用いた壁境界計算手法を比較することで検証を行なった。まず曲率0である平面上に水柱を置き、崩壊の計算を行なった。図2.6に結果を示す。図2.6では水柱をZ軸に垂直な平面で切断した断面を示している。この計算では距離関数は壁境界の形状が簡単であるため、格子点上に定義される値ではなく、平面の境界からの距離をそれぞれの粒子において求めて用いた。水柱の先端位置と高さは壁粒子を用いた計算とほぼ一致した。

次に定量的評価を行なうために直方体の容器内における水柱の崩壊計算を行なった。図2.7の左に示す図が本手法を用いた計算結果であり、右は壁粒子を用いた計算結果である。この計算において、水柱の先端位置を計測して比較したものを図2.9に示す。本手法を用いた場合の方が先端位置の進む時間が若干遅かったが、ほぼ同じ結果が得られた。次に水柱の高さを計測して比較したものを図2.10に示す。計算開始から時間がたつにつれて本手法を用いた計算結果の方が、水柱の高さが高くなっている。この結果は図2.9の先端位置の結果と矛盾のないものになっており、水柱が進む速度が遅いため、水柱の高さが高くなっている。図2.6や図2.7の表示の粒子の色は粒子数密度を表している。大きな粒子数密度を持つ粒子は赤、小さい粒子数密度を持つ粒子は青で描画されている。

図2.9、2.10での計算結果の違いの原因は、距離関数を格子点上で求め、それらを線形補間して任意の点での距離関数を求めていることに起因すると考えられる。格子点上の距離関数を補間して格子点間の距離関数を求めるときまらないことがある。曲率が0でない境界部分での距離の誤差により、図2.9や図2.10の差が生じたと考えられる。この解決方法とし

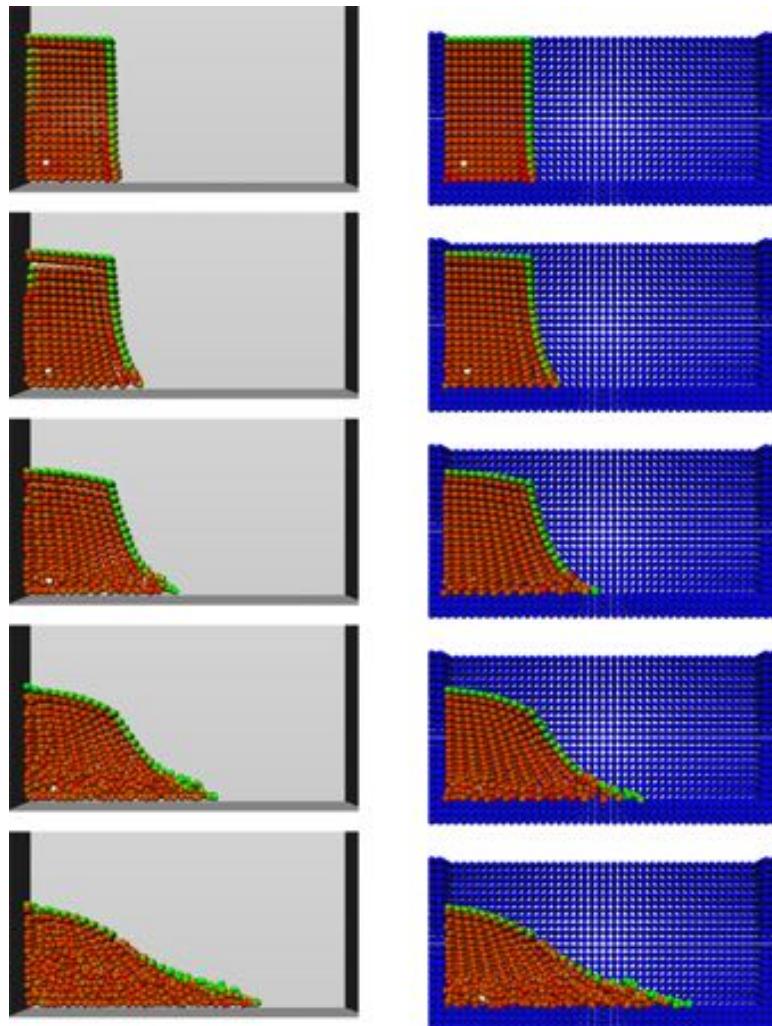


図 2.7: 水柱崩壊シミュレーションにおける壁境界計算モデルの比較。図左は壁重み関数を用いた計算結果であり、図右は壁粒子を用いた計算結果である。

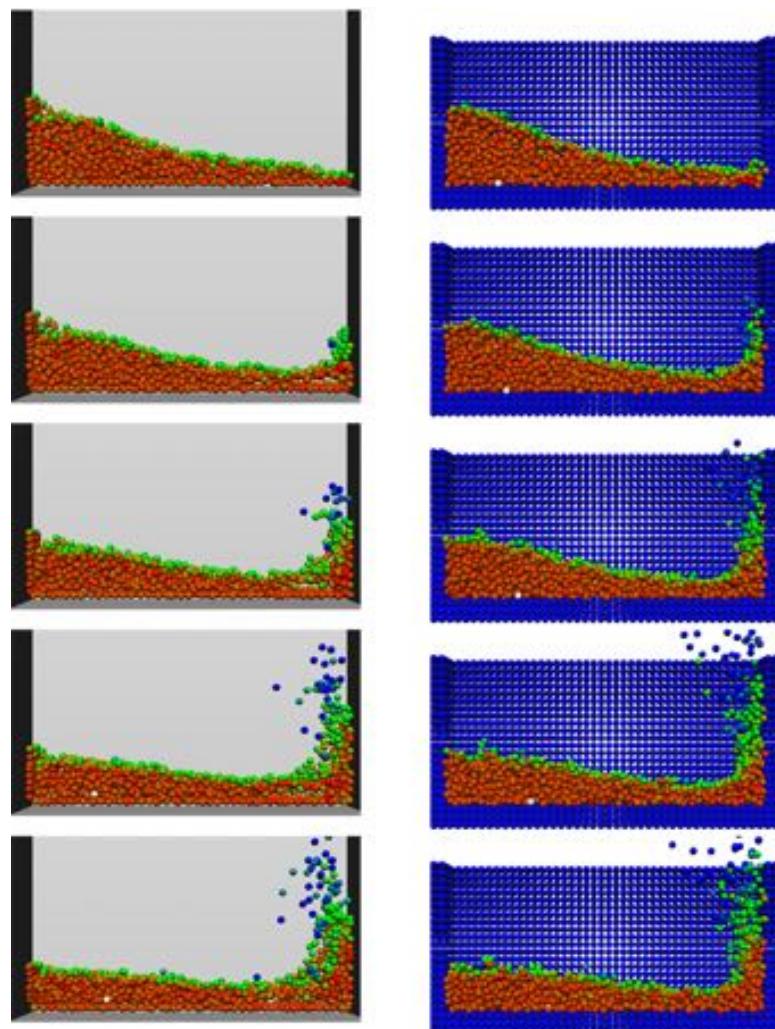


図 2.8: 水柱崩壊シミュレーションにおける壁境界計算モデルの比較。図左は壁重み関数を用いた計算結果であり、図右は壁粒子を用いた計算結果である。

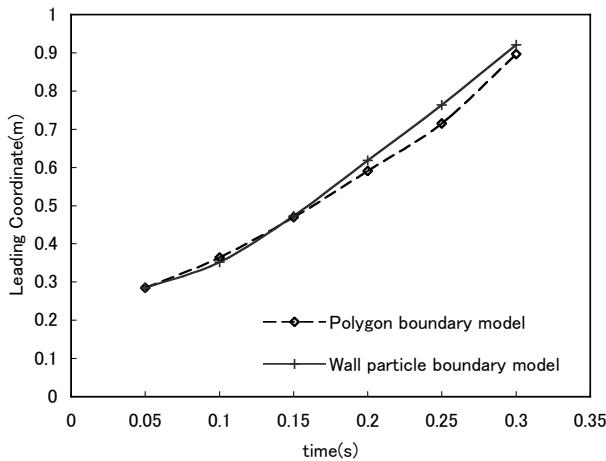


図 2.9: 水柱崩壊シミュレーションにおける壁重み関数を用いた計算結果と、壁粒子を用いた計算結果での流体の先端位置の比較。

ては格子点上で距離関数を計算し、それらを線形補間するのではなく、格子点上で最近点までのベクトルを計算し保持しておき、それらを線形補間して、任意の点での最近点までのベクトルを求め、その大きさを計算することによって距離関数を計算することによって改善できる可能性がある。

2.6 結論

本研究では壁粒子を用いない壁境界計算モデルを開発した。複雑な形状をした境界を持つ計算を行なった。計算モデルの検証のために、壁粒子を用いたダム崩壊の計算と比較を行なった。計算結果はほぼ一致したが、わずかながら差は存在したが、考察により、この計算結果の差は格子点上に定義される距離関数の線形補間に起因するものであると考えられ、この点の改良が今後の課題である。

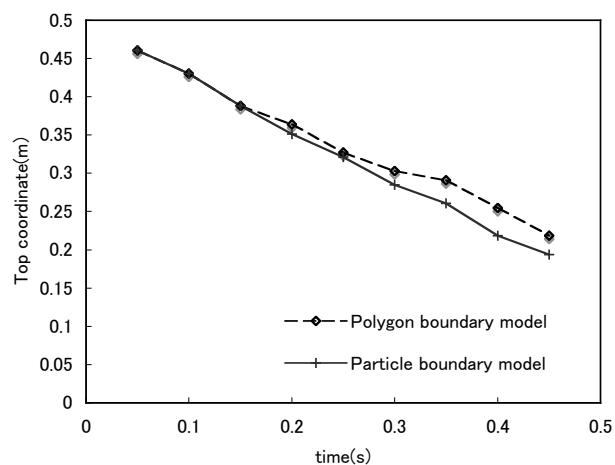


図 2.10: 水柱崩壊シミュレーションにおける壁重み関数を用いた計算結果と、壁粒子を用いた計算結果での水柱の高さの比較。

第3章 SPHによる流体シミュレーションにおける壁境界の改良

3.1 序論

流体は我々の身近に多く存在している。流体の挙動は複雑であるため、コンピュータグラフィックスで流体のアニメーションを作成するときに、流体の支配方程式を解き流体挙動を計算することが行なわれている。流体の解法は大きく分類してオイラー解法とラグランジュ解法が存在する。オイラー解法は空間に固定された計算格子を用いる。一方ラグランジュ解法は格子を用いず、互いに接続されていない粒子を用いて計算を行なう。粒子を用いて計算する手法は粒子法と呼ばれている。この手法は粒子自体が流体を示すので自由表面の扱いが容易であり、また粒子自体が移流するため、移流項の計算における数値拡散が起こらないという特徴がある。流体を計算する粒子法はMoving Particle Semi-implicit法(MPS法)[130, 131, 76]とSmoothed Particle Hydrodynamics法(SPH法)[83]がある。MPS法は非圧縮流体を解くことが可能であり、工学分野において広く用いられている。またSPH法は圧縮流体を解き、MPS法に比べ計算コストが低く、コンピュータグラフィックスにおいて広く用いられている。圧縮性を弱くすることで、SPH法ではほぼ非圧縮な流体を取り扱うこともできる。粒子法では流体は粒子として表され計算され、また流体以外の壁境界も一般的に粒子として表される。しかし壁粒子を用いると境界の形状を正確に表すことができないなどの問題がある。

本論文ではSPH法における粒子を用いない壁境界計算モデルを提案する。本手法は壁境界に粒子ではなくポリゴンを用いる。壁粒子を用いないため、壁粒子を生成する必要がなく、壁境界を正確に表すことができる。粒子法においては壁粒子の占める粒子数が流体の粒子数を含めた全粒子数に占める割合が大きいが、本手法を用いるとその壁粒子を生成する必要がないため、壁粒子を用いる場合と比べると同じ計算を行なうときの粒子数を大幅に削減することが可能である。そのため、同数の流体粒子を用いた計算をした時と比較すると、本手法を用いた場合は計算時間も削減できる。また提案する壁境界計算モデルでは壁境界との計算を簡単な数回の補間計算のみによって求めることができるので、全体の計算コストに占める壁境界の計算時間は小さい[49, 53, 51]。

3.2 関連研究

コンピュータグラフィックスにおいて、計算格子を用いるオイラー解法を用いた自由表面流れの解法では、レベルセット法が界面追跡手法として多く用いられている[37]。しかしオイラー解法を用いると移流計算の際に数値拡散が起きるため、質量損失が起こってしまう。自由表面流れでの質量損失は流体が消えたように見えるため、映像制作などでは大きな問題と

なる。この問題点を克服するためレベルセット法をラグランジュ粒子と組み合わせたパーティクルレベルセット法が開発された[31]。それ以外にも流体と剛体の相互作用[19]、粘弾性流体[39]、薄膜との相互作用[44]、混相流[82]、2次元計算と3次元計算のカップリング[67]なども研究されている。構造格子を用いると壁境界の計算精度が下がるためOctreeを用いたり[81]、非構造格子を用いた研究も行なわれている[72]。

またラグランジュ解法である粒子法に関してMPS法[76]は計算力学の分野において多く研究されており、剛体との相互作用[110]、混相流[57]などの研究も行なわれている。コンピュータグラフィックスの分野においてはPremozeらがMPS法を用いた計算を行なった[102]。SPH法は天文学の分野で開発された手法であったが、流体計算にも用いられており、弾性体とのカップリング[96]や混相流[97]の計算も行なわれている。粒子法は粗い計算解像度での計算でも質量が保存するのでリアルタイムの計算に向いている[94]。

粒子法では一般的に壁境界も粒子として表されて計算される。Müllerらはラグランジュメッシュで弾性体を表し、メッシュと流体粒子を相互作用させる手法を開発した[96]。この手法はメッシュ表面に粒子を発生させて粒子間相互作用を計算し、メッシュに運動量を伝達した。この手法では毎タイムステップごとにポリゴン上に仮想粒子を発生させなければならなく、またポリゴンの大きさによって生成される粒子の密度が変化してしまい、壁付近において粒子数密度一定の条件を必ずしも満たすとは限らないという問題点がある。

3.3 Smoothed Particle Hydrodynamics

3.3.1 支配方程式

非圧縮流体の支配方程式は質量保存を表す連続の式と、運動量保存の式である。

$$\frac{D\rho}{Dt} = 0 \quad (3.1)$$

$$\frac{D\mathbf{U}}{Dt} = -\frac{1}{\rho}\nabla P + \nu\nabla^2\mathbf{U} + \mathbf{g} \quad (3.2)$$

ρ, \mathbf{U}, P, ν はそれぞれ流体の密度、速度、圧力、動粘性係数であり、 \mathbf{g} は重力加速度である。

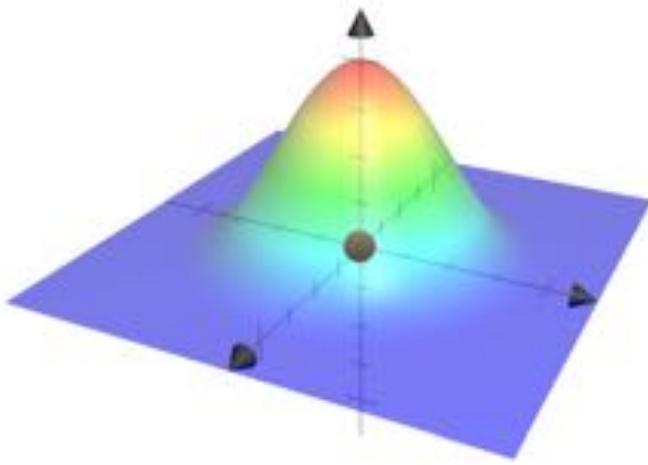


図 3.1: SPH のカーネル.

3.3.2 SPH の原理

SPH は流体を粒子群として近似して計算するシミュレーション手法である。式(3.2)を粒子で解くには、まず空間における物理量 $\phi(\mathbf{r})$ を計算する必要がある。SPH では、空間のある座標 \mathbf{x} での物理量 $\phi(\mathbf{x})$ は、物理量 $\phi(\mathbf{r})$ を空間において積分して求める。

$$\phi(\mathbf{x}) = \int \phi(\mathbf{r}) W(\mathbf{x} - \mathbf{r}) d\mathbf{r} \quad (3.3)$$

ここで W はカーネルである。カーネル W は正規化されており

$$\int W(\mathbf{x} - \mathbf{r}) d\mathbf{r} = 1 \quad (3.4)$$

という条件を満たすものである。

SPH では粒子に物理量を保持させ、その物理量が粒子の周辺に図 3.1 のように分布していると考える。この図は 2 次元平面上に粒子が存在するとして考え、分布している物理量を高さで表現している。そして式(3.3)の積分は、つまり座標 \mathbf{x} での物理量 $\phi(\mathbf{x})$ は周囲の粒子の持つ物理量 ϕ_j の和として求められる。よって粒子を用いて計算するときには式(3.3)は以下のようになる。

$$\phi(\mathbf{x}) = \sum_j m_j \frac{\phi_j}{\rho_j} W(\mathbf{x} - \mathbf{x}_j) \quad (3.5)$$

ここで m_j, ρ_j は粒子 j の持つ質量と密度であり、 $m_j = \rho_j \Delta \mathbf{r}_j$ が成り立つ。また \mathbf{x}_j は粒子 j の座標である。一般的にカーネル W は \mathbf{x} から遠ざ

かると 0 になるものが用いられるため、近傍の粒子の物理量の総和だけを取る。式(3.5)を用いることである座標での物理量を求めることができる。

例えば \mathbf{x} での密度は、以下のように求められる。

$$\rho(\mathbf{x}) = \sum_j m_j W(\mathbf{x} - \mathbf{x}_j). \quad (3.6)$$

流体の運動方程式を解くためには、物理量の他に物理量の勾配を計算する必要がある。 \mathbf{x} での物理量 $\phi(\mathbf{x})$ の勾配 $\nabla\phi(\mathbf{x})$ は、式(3.3)の勾配を取り

$$\nabla\phi(\mathbf{x}) = \int \nabla\phi(\mathbf{r})W(\mathbf{x} - \mathbf{r})d\mathbf{r} \quad (3.7)$$

となるが、ここで部分積分を用いて

$$\int \nabla\phi(\mathbf{r})W(\mathbf{x} - \mathbf{r})d\mathbf{r} = \int \nabla(\phi(\mathbf{r})W(\mathbf{x} - \mathbf{r}))d\mathbf{r} + \int \phi(\mathbf{r})\nabla W(\mathbf{x} - \mathbf{r})d\mathbf{r} \quad (3.8)$$

と変形させることができる。右辺第1項の体積分はガウスの定理により面積分に置き換えられ、またカーネル W は \mathbf{x} から遠い表面だと 0 なので右辺第1項は 0 になる。式(3.7)と式(3.8)より、物理量の勾配は

$$\nabla\phi(\mathbf{x}) = \int \phi(\mathbf{r})\nabla W(\mathbf{x} - \mathbf{r})d\mathbf{r} \quad (3.9)$$

となる。

粒子の持つ物理量の総和で近似すると式(3.9)は以下のように表される。

$$\nabla\phi(\mathbf{x}) = \sum_j m_j \frac{\phi_j}{\rho_j} \nabla W(\mathbf{x} - \mathbf{x}_j) \quad (3.10)$$

このように物理量の勾配の計算を行なうときには、粒子の持つ物理量ではなくカーネルを微分すればよい。

3.3.3 支配方程式の離散化

圧力項

流体の挙動を解くためには圧力項と粘性項を、粒子の保持している物理量を用いて計算しなければならない。圧力項は式(3.10)を用いると以下のように表せる。

$$\mathbf{f}_i^{press} = - \sum_j m_j \frac{p_j}{\rho_j} \nabla W_{press}(\mathbf{x}_i - \mathbf{x}_j) \quad (3.11)$$

ここで p_j は粒子 j の圧力である。しかしこの式を用いると粒子 i と粒子 j の間の力は非対称になってしまい、作用反作用の法則に反する。そこで粒子間の力を対称にするために粒子 i, j の圧力を求めて、以下のようにモデル化する。

$$\mathbf{f}_i^{press} = - \sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W_{press}(\mathbf{x}_i - \mathbf{x}_j) \quad (3.12)$$

粘性項

粘性項を解くためにはラプラシアンを計算する必要があるため、ラプラシアンを粒子上で離散化しなければならない。ここではまずその離散モデルを作る。

粒子 i での値 ϕ_i は粒子 j での値 ϕ_j を用いると、1次までの泰ラー展開で以下のように表せる。

$$\phi_i = \phi_j + \nabla \phi_j \cdot (\mathbf{x}_i - \mathbf{x}_j) \quad (3.13)$$

よって粒子 j の位置での $\nabla \phi_j$ は

$$\nabla \phi_j = (\phi_j - \phi_i) \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|^2} \quad (3.14)$$

と求めることができる。

ラプラシアンは勾配の発散である ($\nabla^2 \phi = \nabla \cdot (\nabla \phi)$)。また発散は勾配の式 (3.10) と同様に求まり、その発散の式に式 (3.14) を代入すると、

$$\begin{aligned} \nabla^2 \phi_i &= \sum_j m_j \frac{\nabla \phi_j}{\rho_j} \cdot \nabla W(\mathbf{x}_i - \mathbf{x}_j) \\ &= \sum_j m_j \frac{\phi_j - \phi_i}{\rho_j} \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|^2} \cdot \nabla W(\mathbf{x}_i - \mathbf{x}_j) \end{aligned} \quad (3.15)$$

と求まり、ここで $\nabla W_{vis}(\mathbf{x}_i - \mathbf{x}_j) = \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|^2} \cdot \nabla W(\mathbf{x}_i - \mathbf{x}_j)$ とおくと式 (3.15) は

$$\nabla^2 \phi_i = \sum_j m_j \frac{\phi_j - \phi_i}{\rho_j} \nabla W_{vis}(\mathbf{x}_i - \mathbf{x}_j) \quad (3.16)$$

となる。この式を用いると粘性項は

$$\mathbf{f}_i^{vis} = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla W_{vis}(\mathbf{x}_i - \mathbf{x}_j) \quad (3.17)$$

と計算することができる。

SPH の原理では物理量の空間微分はカーネルを微分することで求める。しかしカーネルの微分によってラプラシアンを計算し、粘性項をモデル化すると Müller et al. が論じているように非対称な力となってしまう [94]。そのため、粒子 i, j の速度の差を取っている。ここで示した定式化は微分演算子を粒子間で計算し、それらを重み付けして求める形になっている。この考え方は Moving Particle Semi-Implicit(MPS) 法のものであり、この定式化は MPS 法と SPH の両方の考え方を用いた定式化になっている。MPS 法の定式化の詳細については参考文献に譲る [130, 131]。

状態方程式

理想気体の状態方程式から粒子座標での圧力は以下のように計算する。

$$p = k(\rho - \rho_0) \quad (3.18)$$

ここで ρ_0 は初期粒子配置における密度である。非圧縮流体を正確に解くためには、質量保存の式を解かなければならぬが、SPH では質量保存の式を解かないため、完全な非圧縮流体を再現することができない。そこで SPH では k を大きくして圧縮性を下げることによってほぼ非圧縮な流体を計算している。

この圧力を用いると問題が起こる部分が存在する。初期の粒子間距離で均等に粒子が並んでいるときには、空気の領域に粒子が存在しないため、自由表面付近の粒子は近傍粒子が少ない。よってこの状態で粒子座標における密度を式 (3.6) によって求めると自由表面付近の粒子で計算される密度は小さなものとなってしまう。圧力は密度に比例する物理量としてモデル化するので、自由表面に垂直な方向に圧力勾配ができる、自由表面付近で粒子が凝集してしまう。そこで ρ_0 より小さい密度から計算される圧力を p_0 とすることによって初期の粒子間距離で均等に粒子が並んでいるときでも自由表面付近でそのような問題を避けることができる。

カーネル

それぞれの項の計算にはカーネルが必要であり、まず密度の計算に用いられるカーネルは以下のように定める。

$$W(\mathbf{r}) = \begin{cases} \frac{315}{64\pi r_e^9} (r_e^2 - |\mathbf{r}|^2)^3 & 0 \leq |\mathbf{r}| < r_e \\ 0 & r_e \leq |\mathbf{r}| \end{cases} \quad (3.19)$$

ここで r_e は影響半径であり、粒子 i からこの距離以内に存在する粒子のみ粒子 i に影響を与える。本来の SPH ではこのカーネルを微分して圧

力項や粘性項に用いるカーネルを求めるが、式(3.19)のカーネルの中心付近では傾きが0になっているため、圧力項にこのカーネルの勾配を用いると粒子が接近するに従って力が弱くなってしまう。粒子が接近するにつれて強い力で反発させるようにする必要がある。そこで圧力項のカーネルは以下のものを用いる。

$$\nabla W_{press}(\mathbf{r}) = -\frac{45}{\pi r_e^6} (r_e - |\mathbf{r}|)^2 \frac{\mathbf{r}}{|\mathbf{r}|} \quad (3.20)$$

また粘性項には以下のカーネルを用いる。

$$\nabla W_{vis}(\mathbf{r}) = \frac{45}{\pi r_e^6} (r_e - |\mathbf{r}|) \quad (3.21)$$

これらについても式(3.19)と同様に $r_e \leq |\mathbf{r}|$ の条件を満たすものについては値を0にする。

3.4 格子を用いた近傍粒子探索

粒子位置での物理量を計算するためにはその粒子から影響半径内に存在する近傍粒子を探さなければならない。しかしSPHでは粒子が移流することによって移流項の計算を行なうので、粒子の相対位置関係は常に変化していく。そのため、毎タイムステップで近傍粒子を探索しなければならない。粒子1個に対して全粒子 n 個から近傍粒子を探索すると計算コストは $O(n^2)$ となってしまう。そこでまず計算領域のXYZ軸方向の最大値と最小値を求め、領域全体を覆う格子を用意する。格子の構成要素をボクセルと呼ぶ。すると計算領域内のある点はどれか1つのボクセルに内包されるようになる。すなわち粒子もいずれかのボクセルに入っていることになる。

格子と粒子番号を関係づけることによって近傍粒子探索を効率的に行なうことができるようになる。近傍粒子を探索する前に、全ての粒子の番号を粒子が存在しているボクセルに格納して、粒子番号を格納した格子を作成する。このデータ構造を用いると、ある粒子が入っているボクセルに格納されている値を読み出すことによって、その粒子と同じボクセルに存在している粒子がわかる。よって近傍粒子を全粒子から探索する必要はなくなる。粒子 i の近傍粒子は粒子 i の番号が格納されているボクセルとその周囲のボクセルに限定されるため、粒子 i の格納されているボクセルだけでなく、そのボクセルの周囲のボクセルに格納されている粒子番号を読み出すことで近傍粒子を見つけることができる。そのため計算コストは $O(n)$ に改善される。ボクセルの一辺の長さは影響半径とすると、3次元計算では 3^3 個のボクセル内を探索すれば良い。ある計算領域での計算を

考えると、ボクセルの一辺の長さが長くなればなるほど、必要なボクセルの総数は減り、少ないメモリで実現できるが、1ボクセルに格納される粒子数が多くなるため、近傍粒子探索の効率が下がる。またボクセルを小さくすればするほど、1ボクセルに格納される粒子数は減るが、多くのメモリを必要してしまう。そこで影響半径の大きさと1ボクセルの1辺の長さを同じにするのが、効率的である。

3.5 壁重み関数

本章では壁粒子を用いない壁境界計算モデルについて述べていく。SPH法で支配方程式を解くときに壁粒子は流体の密度、粘性項、圧力項に関係する。そこで壁粒子を用いない壁境界計算モデルではこれらを新しくモデル化しなければならない。以下これらのモデル化を順に示していく。

3.5.1 密度

流体の密度は式(3.6)のように表されるが、粒子*i*の近傍に壁粒子が存在すると仮定して、流体粒子の寄与と壁粒子の寄与を分離する。

$$\begin{aligned}\rho_i(\mathbf{r}_i) &= \sum_j m_j W(\mathbf{r}_{ij}) \\ &= \sum_{j \in fluid} m_j W(\mathbf{r}_{ij}) + \sum_{j \in wall} m_j W(\mathbf{r}_{ij})\end{aligned}\quad (3.22)$$

右辺第1項が流体粒子からの寄与、第2項が壁粒子からの寄与である。壁粒子の質量*m_j*は全て一定であり、重み関数*W*では距離の絶対値のみが用いられるので、図3.2のように壁粒子が粒子*i*から壁におろした垂線に垂直に均一に並んでいると仮定すると、粒子*i*から壁までの距離| \mathbf{r}_{iw} |が決まれば、粒子*i*の近傍に存在する壁粒子の配置は一意に決まる。つまり壁粒子の寄与は以下のように粒子*i*から壁までの距離| \mathbf{r}_{iw} |の関数とすることができる。

$$\rho_i(\mathbf{r}_i) = \sum_{j \in fluid} m_j W(\mathbf{r}_{ij}) + Z_{wall}^{rho}(|\mathbf{r}_{iw}|)\quad (3.23)$$

この $Z_{wall}^{rho}(|\mathbf{r}_{iw}|)$ を密度の壁重み関数と呼ぶ。

3.5.2 粘性項

密度と同様に粘性項も流体粒子による寄与と壁粒子による寄与に分解する。粒子*i*に働く粘性力 \mathbf{F}_i^{vis} は、流体粒子から受ける粘性力 $\mathbf{F}_{i,fluid}^{vis}$ と壁

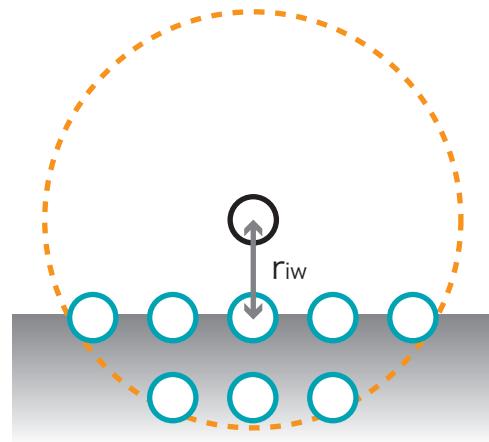


図 3.2: 影響半径内における壁粒子の分布.

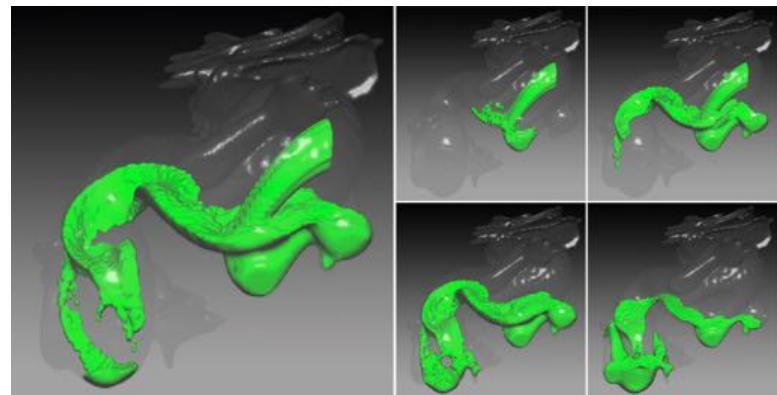


図 3.3: ドラゴンポリゴンモデルを壁境界として用いた自由表面流れの計算結果.

から受ける粘性力 $\mathbf{F}_{i,wall}^{vis}$ に分けることができる。

$$\begin{aligned}\mathbf{F}_i^{vis} &= \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla W_{vis}(\mathbf{r}_{ij}) \\ &= \mathbf{F}_{i,fluid}^{vis} + \mathbf{F}_{i,wall}^{vis}.\end{aligned}\quad (3.24)$$

ここで壁粒子の速度を 0 とすると壁から受ける粘性力 $\mathbf{F}_{i,wall}^{vis}$ は

$$\begin{aligned}\mathbf{F}_{i,wall}^{vis} &= \mu \sum_{j \in wall} m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla W_{vis}(\mathbf{r}_{ij}) \\ &= -\mu \sum_{j \in wall} m_j \frac{\mathbf{v}_i}{\rho_j} \nabla W_{vis}(\mathbf{r}_{ij}) \\ &= -\mu \mathbf{v}_i \sum_{j \in wall} m_j \frac{1}{\rho_j} \nabla W_{vis}(\mathbf{r}_{ij})\end{aligned}\quad (3.25)$$

と計算される。

我々が取り扱っている流体は非圧縮流体である。SPH 法では連続の式を解いていないため、完全な非圧縮を再現することができないが、疑似非圧縮を再現することができる。すなわち流体内部の粒子の密度はほぼ一定である。壁粒子を用いた場合は壁粒子の密度もほぼ一定になる。また重み関数 ∇W_{vis} では距離の絶対値のみ用いられる。そこで式 (3.25)において、壁粒子の密度 ρ_j が一定だと仮定すると、全ての壁粒子の $m_j \frac{1}{\rho_j}$ が定数になる。すなわち粘性項における壁粒子の寄与は以下のように粒子 i から壁までの距離 $|\mathbf{r}_{iw}|$ の関数とすることができます。

$$\mathbf{F}_i^{vis} = -\mu \mathbf{v}_i Z_{wall}^{vis}(|\mathbf{r}_{iw}|)\quad (3.26)$$

この $Z_{wall}^{vis}(|\mathbf{r}_{iw}|)$ を粘性項の壁重み関数と呼ぶ。

3.5.3 圧力項

非圧縮流体を粒子法で解くときには、圧力項は粒子数密度を一定にするように働く力となる。そのため壁が流体粒子に及ぼす圧力による力 \mathbf{F}_i^{press} は壁と流体粒子の距離を初期粒子間距離 d に保とうとする力にする。粒子 i が壁から $|\mathbf{r}_{iw}|$ の距離に存在するとき、圧力による力 \mathbf{F}_i^{press} は粒子を $d - |\mathbf{r}_{iw}|$ の距離だけ $\mathbf{n}(\mathbf{r}_i)$ の方向に押し戻す。つまり圧力による力 \mathbf{F}_i^{press} は

$$\begin{aligned}\mathbf{F}_i^{press} &= m_i \frac{\Delta \mathbf{x}_i}{dt^2} \\ &= m_i \frac{(d - |\mathbf{r}_{iw}|) \mathbf{n}(\mathbf{r}_i)}{dt^2}\end{aligned}\quad (3.27)$$

とモデル化することができる。ここで $\mathbf{n}(\mathbf{r}_i)$ は粒子 i の位置 \mathbf{r}_i における距離関数の勾配ベクトルであり、このベクトルは粒子 i に最も近い壁の法線ベクトルである。

3.5.4 壁重み関数の計算

式(3.26)と式(3.23)での壁重み関数の計算は粒子 i から壁までの距離 $|\mathbf{r}_{iw}|$ のみに依存する関数であるため、この距離がわかれば計算することができます。壁重み関数は粒子から壁までの距離にのみ依存する関数なので前計算しておく、流体計算を行なうときには前計算しておいたデータを参照するだけでよい。壁重み関数は壁内部に存在する粒子位置で重み関数を積分しなければならないため、幾何学的には求めることはできない。そこで図3.2に示すように壁内部に粒子を配置し、それらによる重み関数を数値積分することで求める。前計算では粒子の影響半径内の数点で壁重み関数を計算しておく。そしてそれ以外の点においては計算された値を線形補間して近似値を求めた。

壁重み関数を前計算しておくことで、粒子から壁までの距離に対応する値を参照するだけで壁の寄与を計算できるようになったが、粒子から壁までの距離の計算は壁境界を構成する全てのポリゴンまでの距離計算を行ない、それらのうちの最小値を求める必要があるため、ポリゴン数が増加すればするほど計算コストが高くなる。そこで距離関数を用いると、この計算も前計算で構築したデータを参照するだけで済むようになる。距離関数とは計算領域に境界が定義されたときに、その境界までの最短距離を表す関数である。つまり粒子 i における壁までの距離 $|\mathbf{r}_{iw}|$ は距離関数である。流体計算において壁境界が不動ならば壁の距離関数は変化しないため、壁の距離関数もあらかじめ計算しておくことができる。これによってそれぞれの粒子の計算の際に、壁までの距離を計算する必要はなく、計算された距離関数を参照するだけで距離 $|\mathbf{r}_{iw}|$ を求めることができる。距離関数の計算手法については次章で述べる。

圧力項の計算では粒子 i から壁までの距離以外に、粒子 i に最も近い壁の法線ベクトルが必要となる。ある点において距離関数の勾配を取り正規化したベクトルは、その点から最近点の境界への方向を示すベクトルである。そしてこのベクトルはその最近点での境界の法線ベクトルとなっている。壁境界が変化しないため、計算領域内のある位置から最も近い壁の法線ベクトルも不变である。そのため、この法線ベクトルもあらかじめ計算しておく、流体計算時には値を参照するだけで求めることができる。

3.6 距離関数の計算

距離関数の計算は計算領域内の点から境界までの最近点を計算し、そこまでの距離を求める。境界上の最近点は3次元の問題だと面か辺か点のいずれかになる [4]。境界を構成するポリゴン数が多い場合、距離関数の計算コストは特に高くなる、本研究では距離関数の近似計算を行なった。

3.6.1 手法

前述のように計算領域内の点での境界上の最近点は境界の面か辺か点のいずれかになるが、境界を構成するポリゴンが十分細かく境界から距離が近い点では最近点は面上にあることが多い。本研究で必要な距離関数は境界から粒子の影響半径内の領域のみであるため、最近点は全て面上に存在すると近似して距離関数を求めた。

まずある1ポリゴンによる距離関数を考える。このポリゴンを最近点とする計算領域内の点はこのポリゴンの近辺にしか存在しないため、ポリゴンを内包する立方体を生成する。そしてこの立方体内部でポリゴンまでの距離を計算する。隣り合ったポリゴンでも同様の処理を行なうと距離計算を行なう立方体が重複する部分が出てくる。距離関数は境界上の最近点までの距離なので、重複した部分においては距離の比較を行ない、小さい値を保持する。この処理を全てのポリゴンに対して行なうことで距離関数を計算できる。

この距離関数計算手法はGraphics Processing Unit (GPU) を用いて高速化することが可能である。まず3次元格子にあたるテクスチャを用意し、その上にポリゴンを内包する立方体をレンダリングしていく。まずピクセルシェーダで距離関数を計算し、この値を出力する。そして最小値を求める処理は深度テストを用いて行なうことができる。このように求めた距離関数をZ軸で8枚にスライスしたものを図3.4に示す。距離関数の値によってコンター表示している。

計算された距離関数は3次元格子点上の値として格納した。本研究では 128^3 の3次元格子を用いた。そして流体粒子の座標における距離関数の評価は、その粒子を囲む8格子点での距離関数から線形補間することで求めた。本手法を用いた距離関数の計算時間を表3.1に示す。約20万ポリゴンで構成されているドラゴンポリゴンモデルでも本手法では1秒未満で距離関数を計算できている。

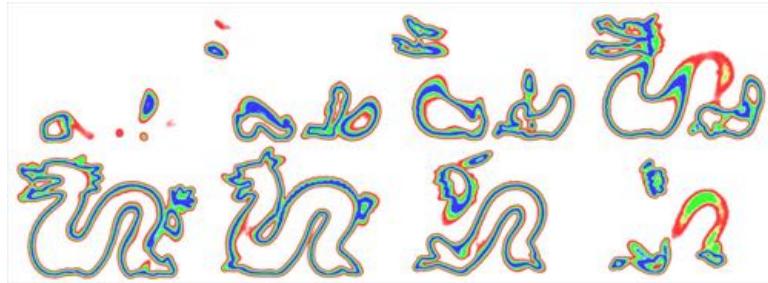


図 3.4: ドラゴンポリゴンモデルから計算された距離関数.

表 3.1: 3種類のポリゴンモデルのポリゴン数と距離関数の計算時間.

Model	Number of polygon	Time
Gargoyle	500,000	2,172
Dragon	202,520	906
Buddha	32,328	297

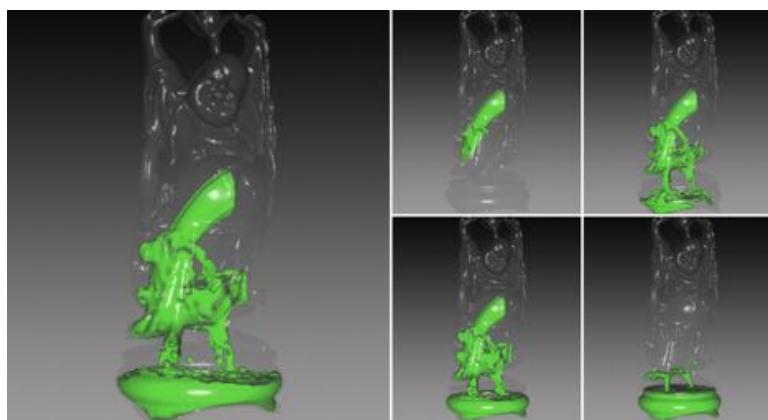


図 3.5: ブッダポリゴンモデルを壁境界として用いた自由表面流れの計算結果.

3.7 結果

本手法を Pentium 4 3.6GHz の CPU, 3.0GB の RAM を搭載した PC 上で C++ を用いて実装した。そして様々なポリゴンモデルを壁境界として用いて計算した。本論文ではいくつかの計算結果を示す。ここで示す計算結果は最大粒子数はいずれも 20,000 である。計算は全て固定壁境界を用いており、この壁の距離関数と勾配ベクトルは前計算によって求めた。そのため提案している壁境界計算モデルは計算時には各粒子位置における距離関数と勾配ベクトルを計算された値から補間して求めた。また壁重み関数も前もって計算によって求めており、粒子位置での値も補間して求めるため、壁境界の計算コストは低い。粒子法では壁境界を正確に表すためには粒子間距離を小さくする必要があるが、そうすると全粒子数が膨大になってしまう。オイラー解法では壁境界での計算解像度をあげるためにアダプティブグリッドが用いられているが、粒子法ではまだそのような手法は開発されていない。しかし本手法を用いることで壁境界は粒子の計算解像度によらず正確に表すことができる。

図 3.6 はガーゴイルモデル内部に流体を流入させた結果を示す。計算結果から表面を抽出して構築したメッシュを透明にレンダリングした結果である。このモデルは 2 枚の薄い羽を持っている。このような細かい部分も滑らかな壁境界として計算できている。図 3.3 はドラゴンモデル内部に流体を流入させた結果である。図 3.5 にブッダモデル内部に流体を流入させた結果を示す。このモデルは粒子径よりも小さな構造を持っている。そのような部分には流体粒子は流入することはできない。これは粒子法自体の限界である。これらのモデルを壁境界として用いたときの計算時間を表 3.2 に示す。

壁境界形状が複雑になればなるほど壁境界の粒子を生成することも難しくなってくる。壁粒子を用いる手法では壁に接している流体の粒子数密度を正確に計算するため、その粒子の影響半径内には壁粒子を配置しなければならない。これにより大抵の場合壁粒子を数層取らなければならない。

ここで単純な計算体系を用いて計算時間を測定し比較した。 $20 \times 50 \times 20$ 粒子で構成されている水柱と内部に $60 \times 50 \times 20$ 粒子入れることができる水槽を用いた。この水柱を構成する粒子数は 20,000 であり、壁に 3 層粒子を取ると水槽を構成する壁粒子の数は 36,096 である。すると本手法で計算に用いる粒子数は 20,000 であるのに比べ、壁粒子を用いる場合は 56,096 粒子用いなければならなく、全粒子数の 64 % を壁粒子が占めている。20,000 粒子を用いた計算時間は 1 タイムステップあたり 133.4 ミリ秒であったのに比べ、56,096 粒子用いた場合は 356.4 ミリ秒であった。このように総粒子数の増加は計算時間の増加に直結するため、本手法を用いることで計算時間の削減が可能であった。図 3.6, 3.3, 3.5 のモデルでそれぞ

表 3.2: 総粒子数と 1 タイムステップにかかった計算時間.

Model	Fluid	Time
Gargoyle	20,000	309.4
Dragon	20,000	281.2
Buddha	20,000	296.8

表 3.3: 1 層の壁粒子数と総粒子数に占める壁粒子の割合.

Model	Wall	Total	Ratio
Gargoyle	26,688	46,688	0.572
Dragon	18,582	38,582	0.482
Buddha	12,084	32,084	0.377

れ壁粒子を生成した時の総粒子数における壁粒子の割合を表 3.3 に示す。ここでは入力としたポリゴンモデルが複雑であり、壁粒子を複数層生成するのが困難であったため、1 層だけ生成した。ポリゴンモデルの変換には Dong らの手法を用いた [29]。数層の壁粒子を生成する場合には 1 層だけの場合より粒子数が多くなる。1 层だけの壁粒子を生成した場合でも壁粒子数は総粒子数の約半分を占めていた。

本手法では壁の平均曲率が 0 であると仮定しているが、本研究の結果からわかるように大きい平均曲率を持つ壁境界でも十分良い結果が得られている。これは大部分の壁の平均曲率に比べ平均粒子間距離が小さく、粒子の影響半径内の壁の平均曲率がほぼ 0 と近似できているからであると考えられる。そのため平均粒子間距離が大きくなってしまうとこの近似が難しくなると考えられる。

3.8 結論

本論文では SPH 法における壁粒子を用いずにポリゴンモデルを用いる壁境界計算手法を開発した。本手法では壁境界条件を粒子ではなく壁重み関数と距離関数で表す。壁粒子を用いる場合と総粒子数を比較し、本手法を用いることで総粒子数の削減ができる事を示した。本研究での壁境界計算モデルは壁の平均曲率を 0 と仮定しているが、今後の課題として平均曲率を考慮したより正確な壁重み関数の計算手法の開発が挙げられる。

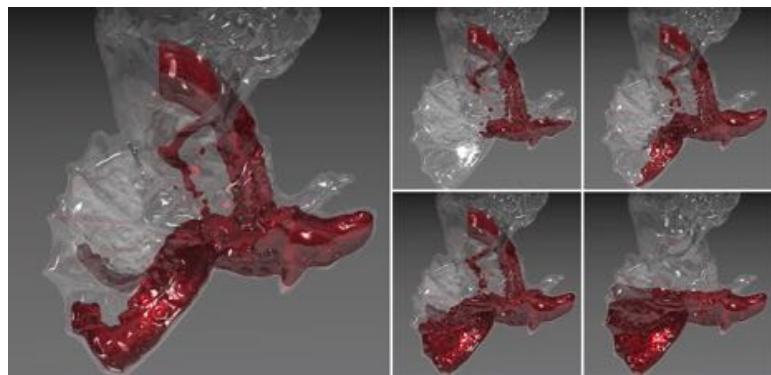


図 3.6: ガーゴイルポリゴンモデルを壁境界として用いた自由表面流れの計算結果。

第4章 粒子から滑らかな表面を構築 する手法の開発

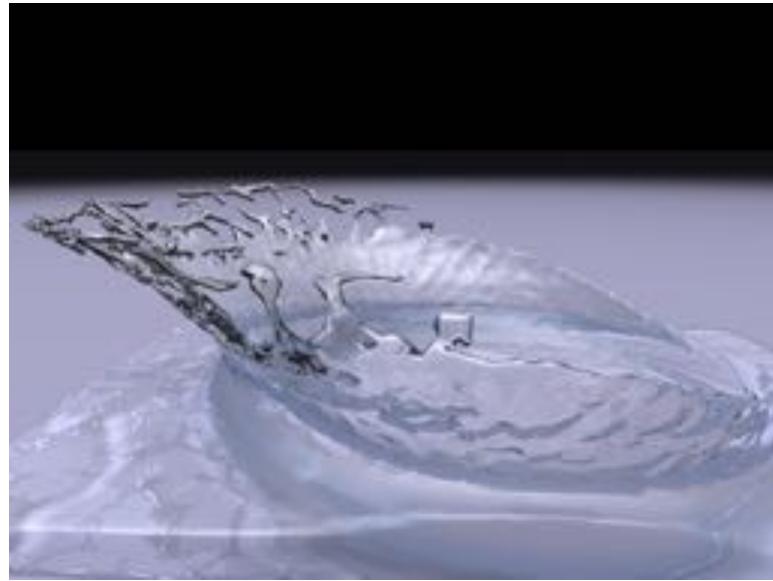


図 4.1: 液滴を水面に落としたシーン。

4.1 序論

粒子法は移流によって数値拡散が起きないため、質量損失を起こすことはない。粒子自体が流体を表すため界面追跡を行なう必要がなく自由表面流れを容易に計算することができる。一般的に粒子法の計算結果をレンダリングするときには計算粒子に濃度球を割り当て陰関数曲面を構築して表面を求める。しかしこのように表面を求めるとき図 4.1 に示すような薄い膜や鋭いエッジを表現するのが困難であった。

本論文では粒子から薄い膜や鋭いエッジを持つ表面を構築する手法を提案する。本手法はまず粒子に濃度分布を与え陰関数曲面を構築する。そして構築した表面を処理することによって表面を作り出す。濃度分布を与えて表面を構築する手法では濃度分布の関数を変えることしか、表面の形状を制御することはできなかったが、本手法ではそれらに加えてパラメータを操作することによって制御することができる。よって粒子から構築することのできる表面の形状の自由度が高くなり、様々な表面を作り出すことができる。本論文ではまず少ない粒子によって作成することのできる表面を 2 次元と 3 次元の例で示す。最後に粒子法の 1 つである Smoothed Particle Hydrodynamics(SPH) を用いた流体計算結果に本手法を適用した例を示すことによって手法の有用性を示す [136, 50]。

4.2 関連研究

Blinn が開発した濃度球を用いて複数の粒子から表面を構築する手法は粒子からの表面抽出に多く用いられてきた [14]. Müller らは SPH を用いて流体のリアルタイムシミュレーションを行ない、粒子に濃度球を割り当てる粒子法計算結果から表面を構築した [94]. Premonze らは非圧縮流体計算手法である Moving Particle Semi-Implicit (MPS) Method を用いてシミュレーションを行なったが、計算粒子から表面の生成が困難であるため、Level Set 関数を用いて表面の構築を行なった [76][102]. また Zhu and Bridson は計算粒子から凹凸の少ない陰関数曲面を構築する手法を開発した [129]. しかしこの手法でも鋭いエッジを構築することは困難であり、それらを構築するためにはやはり多数の計算粒子を用いる必要がある.

4.3 手法

本手法は流体表面を 2 段階の操作によって構築する。1 段階目は濃度球を用いて粒子から表面を構築し、2 段階目では得られた表面を処理する。

4.3.1 陰関数曲面の構築

陰関数表現とは空間の点 \mathbf{x} においてある関数 $\phi(\mathbf{x})$ を定義し、 $\phi(\mathbf{x}) < 0$ である部分を物体の外側、 $\phi(\mathbf{x}) > 0$ である部分を物体の内側、そして $\phi(\mathbf{x}) = 0$ である部分を物体の表面として表現するものであり、この表面を陰関数曲面と呼ぶ。まず 1 個の粒子について考える。1 個の粒子 i に空間分布を持った密度関数 $W_i(\mathbf{r})$ を定義する。この関数は例えば以下のような関数を用いることができる。

$$W_i(\mathbf{r}) = \begin{cases} \frac{r_e}{|\mathbf{r}|} - 1 & 0 \leq |\mathbf{r}| < r_e \\ 0 & r_e \leq |\mathbf{r}| \end{cases} \quad (4.1)$$

ここで \mathbf{r}_i は粒子 i の座標であり、 $W_i(\mathbf{x})$ は \mathbf{x} における密度を表している。この密度関数は一例である。そしてある値 T を持つ部分が表面だとして、陰関数曲面は

$$\phi(\mathbf{x}) = W_i(\mathbf{r}_i - \mathbf{x}) - T \quad (4.2)$$

と表すことができる。そして粒子が複数個存在する場合はそれらの密度の総和を取り

$$\phi(\mathbf{x}) = \sum_{i \in Fluid} W_i(\mathbf{r}_i - \mathbf{x}) - T \quad (4.3)$$

と、陰関数曲面を求め、この $\phi(\mathbf{x}) = 0$ の部分を流体の表面とする。

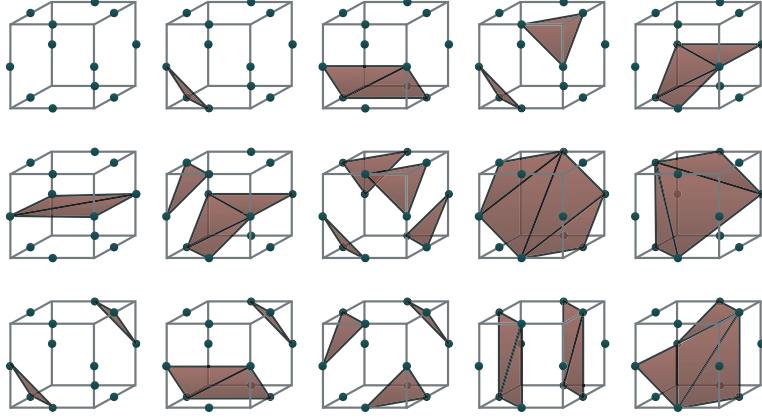


図 4.2: マーチングキューブ法での表面のパターン

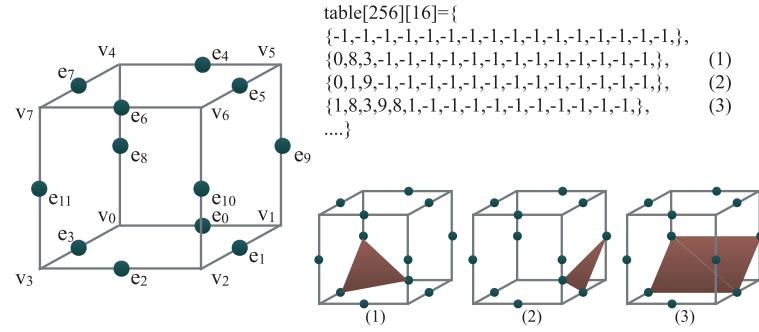


図 4.3: テーブルによる表面のパターンの表現

陰関数 $\phi(\mathbf{x})$ を計算するときに、計算領域を格子で分割しておき、その格子点上で $\phi(\mathbf{x})$ の値を求める。この計算はそれぞれの粒子に対してその粒子の影響半径内に存在する格子点を探査し、それぞれの格子点の座標と粒子の座標を用いて密度 $W_i(\mathbf{r}_i - \mathbf{x})$ を計算して、全粒子の値を足し合わせる。この計算も流体計算の近傍粒子探索と同様にその格子の周辺に存在する粒子のみを探す必要があるため、計算領域を覆う格子を用意して、その中に粒子番号を格納したデータ構造を作成することによって効率化することができる。

このようにして計算領域内での陰関数を求めたが、求まった関数は空間中の格子点上の値でしかないので、ポリゴンを描画するレンダラーを用いることはできない。そこでこの陰関数からマーチングキューブ法を用いて表面のポリゴンを抽出する [80]。計算領域内のある 1 個の立方体を考えると、この立方体の 8 個の頂点にそれぞれ陰関数による値が割り当てられ

ており、それらを用いると頂点は物体内部と外部の2種類に分類され、表面がどの辺上に存在するかを求めることができる。頂点はそれぞれ内部か外部の2通りでしかないため、立方体のどの辺上に表面が存在するかは限られたパターンにしかならない。1頂点あたり2通りの組み合わせがあるため、立方体の頂点の内部と外部の組み合わせは $2^8 = 256$ 通りになる。よって全ての立方体においてどの辺上で物体の内外が変化しているかを条件分岐で探す必要はなく、この256通りの辺の分割パターンを用意しておくことで、そのテーブルをパターンで参照するだけで、表面が存在する辺を求めることができる。これらのパターンの中には回転すると同じ分割パターンになるものが多く存在する。すると立方体の分割パターンは図4.2に示す15通りになる。

実装する際にはまず図4.3のように頂点と辺に番号をつけて、どの辺上を表面が通っているかのテーブルを用意する。図4.3にこのテーブルの一部を示す。256通りのパターンであり、それぞれは15個の要素を持っている。この15個の要素1つ1つは一辺に対応しており、どの辺を結んで表面ができるかを示している。例えば図の(1)は0,8,3番の辺 (e_0, e_8, e_3) を結んで表面ができるということを示している。頂点の内外からこれらのテーブルのどのパターンを参照するかを計算する必要がある。これは i 番目の頂点 (v_i) での $\phi(\mathbf{x})$ が0以下であったならば(物体の外部ならば) 2^i を加算する。例えば v_0 だけが0以下であるならばこの値は 2^0 となり、1番目の0,8,3,…を参照する。このようにして立方体内での表面のポリゴンを求めることができる。

ここでわかっているのはどの辺上に表面が存在するかである。辺の両端の表面での値が $+d, -d$ ならば頂点は辺の中点に存在するが、一般的には表面が全て辺の中点を通るわけではない。そこでその表面と辺との交点の位置を求めなくてはならない。2つの頂点上での値を ϕ_0, ϕ_1 、それらの座標をそれぞれ $\mathbf{x}_0, \mathbf{x}_1$ とすると、交点の座標 \mathbf{p} は

$$\mathbf{p} = \frac{|\phi_0|\mathbf{x}_0 + |\phi_1|\mathbf{x}_1}{|\phi_0| + |\phi_1|} \quad (4.4)$$

と求め、表面ポリゴンを構成する頂点の座標を求めることができる。

レンダリングを行なうときにはポリゴンのそれぞれの頂点での法線ベクトルが必要になる。頂点での法線ベクトルはその頂点を共有しているポリゴンの法線ベクトルの平均として計算することもできるが、正規化された陰関数の勾配 $-\nabla\phi(\mathbf{x})/|\nabla\phi(\mathbf{x})|$ が法線ベクトル \mathbf{n} になっている。陰関数の勾配 $\nabla\phi(\mathbf{x})$ は

$$\nabla\phi(\mathbf{x}) = \left(\frac{\phi(\mathbf{x} + \Delta\mathbf{x}) - \phi(\mathbf{x} - \Delta\mathbf{x})}{2d}, \frac{\phi(\mathbf{y} + \Delta\mathbf{y}) - \phi(\mathbf{y} - \Delta\mathbf{y})}{2d}, \frac{\phi(\mathbf{z} + \Delta\mathbf{z}) - \phi(\mathbf{z} - \Delta\mathbf{z})}{2d} \right) \quad (4.5)$$

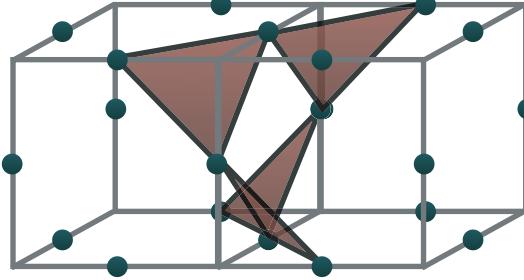


図 4.4: マーチングキューブ法で穴が開く例

と求めることができる。なお $\Delta\mathbf{x} = (d, 0, 0)^T$, $\Delta\mathbf{y} = (0, d, 0)^T$, $\Delta\mathbf{z} = (0, 0, d)^T$ である。

ただし、マーチングキューブ法にはアルゴリズム上ポリゴンが閉じずに穴が開いてしまう場合があるという問題点がある。例えば、図 4.4 に示すような値の分布になっている場合は必然的に穴が開いてしまう。これはキューブの解像度が粗いためであると考えられ、キューブの大きさを小さくすることで解決できる場合がある。

4.3.2 表面の最適化

この表面を構成する頂点が n 個あり、これらを $V^0 = \{\mathbf{v}_0^0, \mathbf{v}_1^0, \mathbf{v}_2^0, \dots, \mathbf{v}_n^0\}$ とする。2段階目ではまず頂点それぞれに対して以下のように頂点 \mathbf{v}_i^t の周囲に存在する粒子の座標の重み付け和を取り、それらの重みの総和で割ることで頂点 \mathbf{t}_i^t を計算する。

$$\mathbf{t}_i^t = \frac{\sum_j w'(\mathbf{v}_i^t - \mathbf{r}_j) \mathbf{r}_j}{\sum_j w'(\mathbf{v}_i^t - \mathbf{r}_j)} \quad (4.6)$$

ここで重み関数 w' は w 同じものを用いた。

このように全ての頂点について $T^t = \{\mathbf{t}_0^t, \mathbf{t}_1^t, \mathbf{t}_2^t, \dots, \mathbf{t}_n^t\}$ を求め、頂点 \mathbf{t}_i^t と頂点 \mathbf{v}_i^t を用いることによって頂点 \mathbf{v}_i^{t+1} を計算する。

$$\mathbf{v}_i^{t+1} = (1 - c)\mathbf{v}_i^t + c\mathbf{t}_i^t \quad (4.7)$$

ここで c は 0 から 1 の間の値を取るパラメータであり、この値を調節することによって表面の鋭さを調整することができる。表面をレンダリングするためには各頂点での法線ベクトルを求める必要がある。計算された頂点の法線ベクトルは濃度値の勾配を取ることによって求めることができないた

め、頂点 \mathbf{v}_i^t の法線は 2 次元の場合は頂点を共有する辺、3 次元の場合は共有する面の法線の和を取り、正規化して求める。

このように求めた \mathbf{v}_i^{t+1} を用いて式(4.6)と式(4.7)を再度計算することで \mathbf{v}_i^{t+2} を得ることができる。この操作を繰り返すことによって表面の形状をより粒子配置に密着した表面にすることができる。また本手法では c と t の 2 個のパラメータを用いて表面形状を制御することができる。

4.4 結果

まず 2 次元で計算した例を図 4.5 に示す。6 個の粒子を配置し、それから陰関数曲面の表面を抽出したものが(a)である。それ以外の図で示しているのは緑色の曲線が T^t であり、青色の曲線が V^t である。(b) は $t = 1$ での表面であり、(c) は $t = 2$ での表面を示したものである。中段、下段に行くに従って t が増加していく。反復を繰り返すことによって、表面が粒子座標に近づき、鋭い表面も表現できるようになっている。本手法は t を増加させることにより、表面上のある点は空間に分布している粒子座標の重み付け重心に収束していく。図 4.5 の流体の端の粒子の周りの表面は濃度球を用いた場合は広く広がっているが、その表面の近傍を見ると 1 個の粒子だけ近くに存在しているので、その表面で計算される粒子の座標の重み付け重心はほぼその粒子の座標になる。よって t を増加させることによって表面はその粒子の座標に収束していく。また図 4.7 では式(4.7)での c の値を変化させたときの表面の変化を示している。それぞれ $t = 2$ の表面、つまり 2 回反復を行なった後の表面である。(a) は $c = 0$ の表面を示している。 $c = 0$ なので表面は変化しない。すなわちこの表面は陰関数曲面を抽出したときの表面である。図 4.7 の右に行くに従って c の値は大きくなり、 V^t は T^t に近づいていく。これらの 2 つの図からわかるように、反復回数 t を変え、パラメータ c の値を変えることによって様々な表面を構築することができる。濃度球のみを用いた表面構築手法では、重み関数を変えることによってしか表面の形状を変えることができなかつた。しかし、本手法では t, c の 2 つのパラメータを調節することによって様々な表面を表現することができるようになった。提案手法は粒子が均等に並んでいる平たい表面にも影響を与える。本手法がそのような表面をどのように変化させるかを示したものが図 4.6 である。この図から本手法を用いると良いパラメータが選択されたときは平な表面を作成することができるが、同時にパラメータによっては表面をとがったものにすることもある。

また図 4.8 には 3 次元での計算例を示す。3 次元計算では陰関数曲面を構築した後に、まず Marching Cubes を用いて表面ポリゴンを抽出した[80]。この例でも 6 個の粒子を用い、これらの配置は左上の図に示すよう

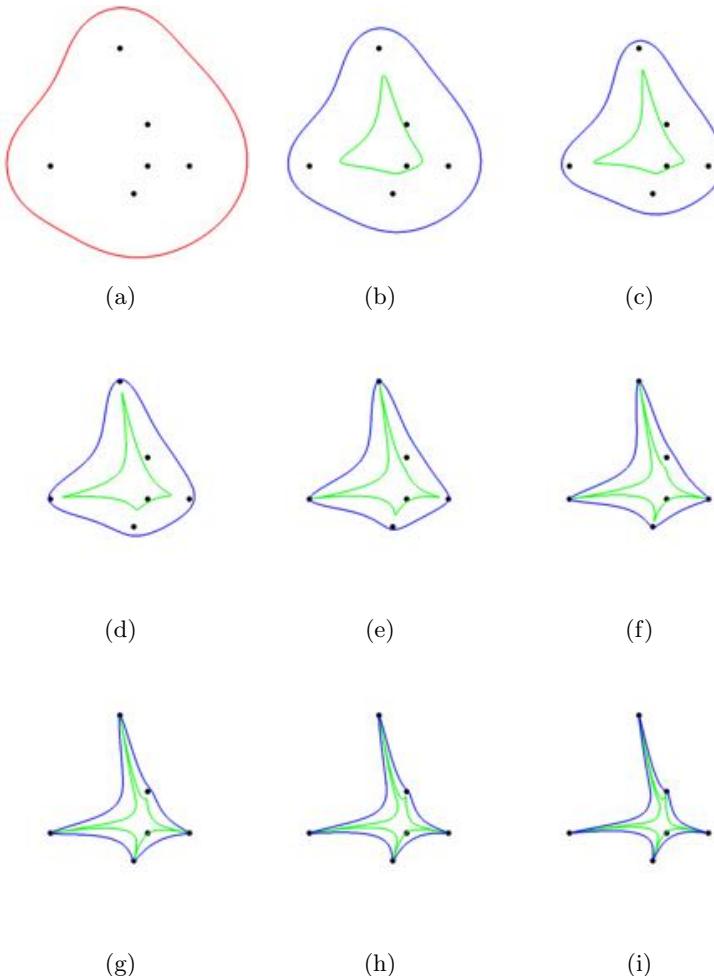


図 4.5: 2 次元での計算例. 図中の黒点は粒子位置を示したものである. (a) は濃度分布の表面を抽出したものであり, (b) から (i) の青線と緑線はそれぞれ $t = 1, 2, 3, 4, 5, 6, 7, 8$ での表面 V^t と T^t を示している. パラメータ c の値には 0.3 を用いた.

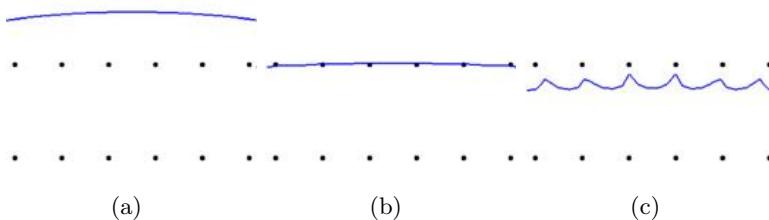


図 4.6: パラメータ t を変えることによる均等に並んでいる粒子から構築された表面への影響. $c = 0.3$ が用いられ, 図は左からそれぞれ $t = 3, 5, 7$ の結果を示している.

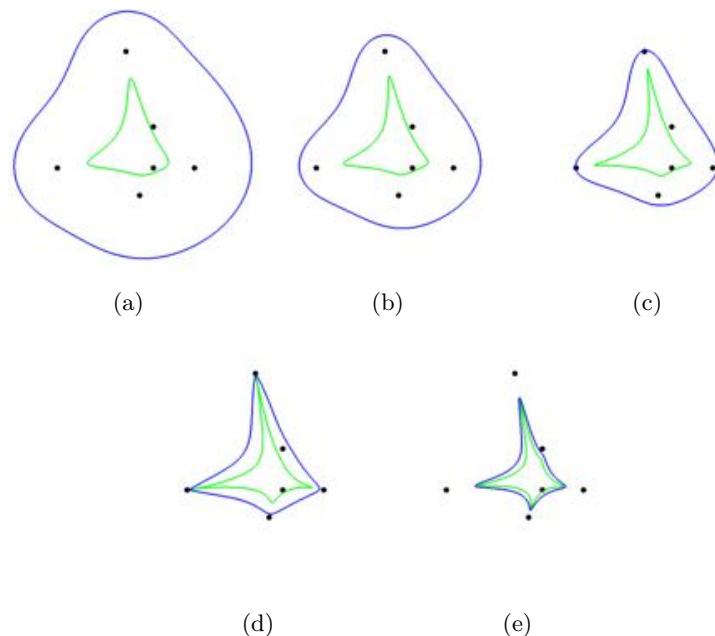


図 4.7: パラメータ c の変化による表面の変化. 青線は V^t であり緑線は T^t である. (a) から (e) に用いたパラメータ c はそれぞれ $c = 0, 0.2, 0.4, 0.6, 0.8$ であり, $t = 2$ での表面を示したものである.

になっている。それらに濃度分布を与え、表面抽出した図が上段2個目の図であり、右に行くに従い反復回数 t を増やしたものである。

SPH で粒子を用いて流体計算を行なった結果に本手法を適用したものを見図 4.9 に示す。円の形状をした流入口が存在し、球に流体が衝突する。図左は流体粒子に濃度分布を割り当て、表面を構築したものであり、図右はその表面から本手法を用いて構築した表面である。14,272 個の流体粒子を用い、パラメータには $c = 0.3, t = 2$ を用いた。これらの図からわかるように濃度球のみを用いた場合は薄い膜を表現することが困難である。上段の図の流体の端に着目すると図左の濃度球を用いた方法では1粒子分の厚みで表現されている。この厚みは1粒子の直径と考えることができる。それに比べ図右では同じ部分が鋭くなっている。この鋭さを濃度球で実現するためには、その最も薄い部分の厚さの直径の粒子を用いなければならない。つまり、薄い膜を表現するためにはそれだけ流体計算に用いる粒子の解像度を上げなければならない。しかし本手法を用いると薄い膜も表現することが可能になっている。上段の図からわかるように濃度球のみを用いた場合は流体のエッジが丸まっているのに比べ、本手法を用いた結果では鋭いエッジを作成している。この計算例では流体の表面は 39,564 頂点から構成されており、ポリゴン数は 79,100 であるが、図 4.9 の左に示す濃度球を用いて構築した表面から図右の表面を作成するのにかかった計算時間は Core2 X6800 の CPU を搭載した PC にて 15 秒であった。本手法は濃度球を用いて構築した表面を元にして新しい表面を構築するため、濃度球を用いた手法よりも計算時間がかかる。本手法の計算時間は t の値が増加するに従って計算時間は線形に増加する。図 4.9 の例では $t = 2$ であるが、 $t = 1$ の場合はその計算時間の約半分で計算することができる。

4.5 結論

本論文では粒子から薄い膜や鋭いエッジを持つ表面を構築する手法を提案した。濃度球を用いて構築した表面を処理することによって今まで表現が困難であった表面も構築することができるようになった。また本手法では濃度分布関数だけでなく表面を制御するパラメータを調節することによって様々な表面を作成することができる。本論文ではパラメータ c と t を変化させることによる表面の変化を 2 次元計算で示した。最後に粒子法による流体計算結果に本手法を適用することによって、薄い膜や鋭いエッジを持つ表面を構築することができることを示した。

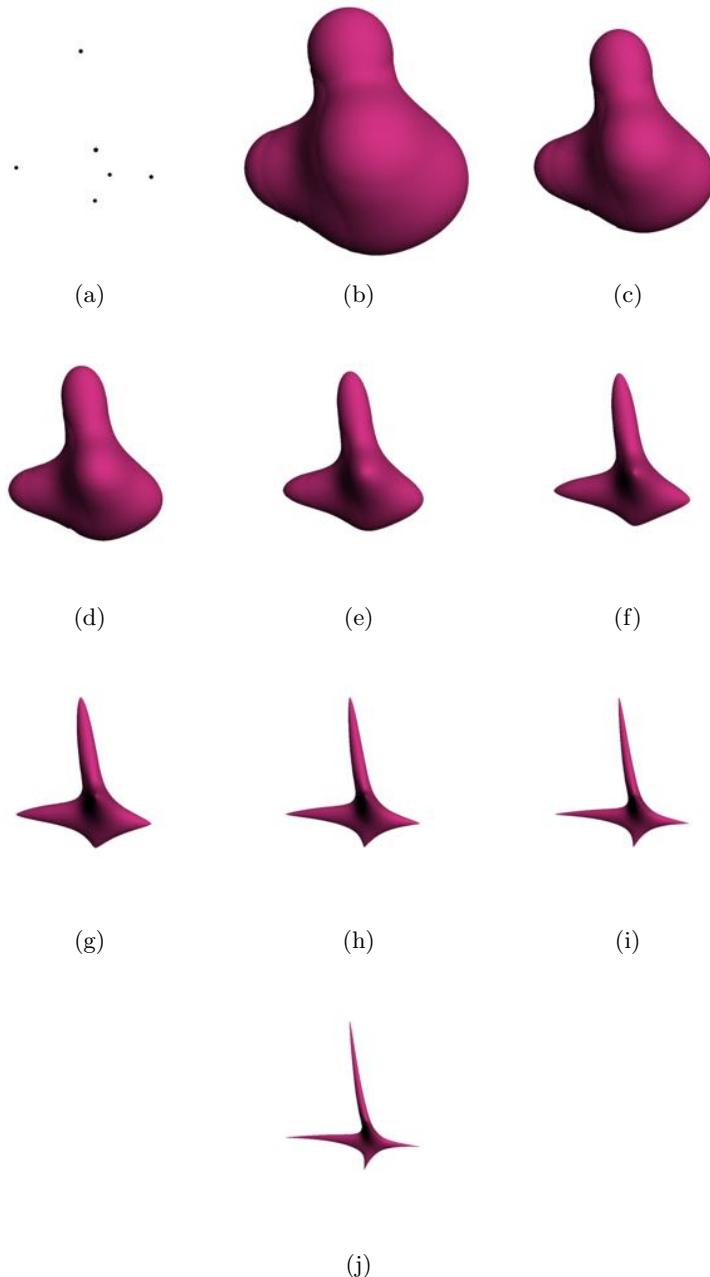


図 4.8: 3 次元での計算例。6 個の粒子を用いており、(a) はそれらの位置を示したものである。それ以外の図は表面ポリゴンを示している。(b) から (j) の図はそれぞれ $t = 0, 1, 2, 3, 4, 5, 6, 7, 8$ での表面を示している。パラメータ c の値には 0.3 を用いた。

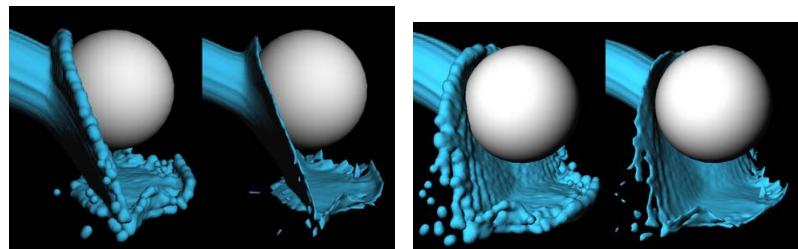


図 4.9: 3 次元流体シミュレーションへの応用例。左列の図は濃度球を用いた結果であり、右列は提案手法を用いた結果である。



図 4.10: 本手法を大規模粒子法シミュレーション結果に適用した例。

第5章 スライスグリッド

5.1 序論

粒子を用いる流体計算手法は、計算粒子自体が流体を表すため、格子を用いる流体計算手法のように流体の速度の計算の他に流体の界面を追跡する必要がない。さらに、計算格子を用いないため細かい飛沫なども容易に計算することができる。しかし格子を用いた解法ではそれぞれの格子の接続関係が決まっているが、粒子法では計算粒子同士が接続関係を持たず、動的に配置を変更するため、ある座標での物理量を計算するためには各タイムステップにおいて、その座標の近傍に存在する粒子を探索しなければならない。計算領域を内包するように計算格子を用意する。そして近傍粒子探索を効率的に行なうために、計算領域に格子を配置して、ある粒子の番号をその粒子が入っているボクセルに格納することで効率化することができる。しかし固定された等間隔格子を用いると、流体の計算領域内の分布が変化するときには粒子が存在しないボクセルが多くなってしまい、メモリ効率が悪い。プログラムが使用可能なメモリの大きさは限りがあるため、格子に用いるメモリを少なくすればするほど、流体に用いることのできるメモリが増え、より大規模なシミュレーションを行なうことができるようになる。またゲームやバーチャル外科手術などでは、ユーザーの視点が移動することがあり、広い領域を計算する必要がある。

本研究では粒子法シミュレーションの近傍探索に用いる格子のメモリ効率の良いデータ構造を提案する。本研究で提案するスライスグリッドと既存のデータ構造との比較は5.3章で考察するが、本手法はデータ構造構築が容易であり、データ構造の構築のコストも低いため、リアルタイムアプリケーションにも用いることができる[135, 52]。

5.2 関連研究

粒子法を用いた流体シミュレーション手法はMoving Particle Semi-implicit(MPS) Method[76]とSmoothed Particle Hydrodynamics(SPH)[83]がある。PremozeらはMPSをコンピュータグラフィックスの分野に導入した[102]。MPSは連続の式から導出するポワソン方程式を解いて非圧縮流れを計算することができるが、SPHに比べて計算コストは高い。そのためコンピュータグラフィックスの分野では全て陽的に解くことができるSPHが多く用いられている。例えばMüllerらはSPHをリアルタイムアプリケーションに用いた[94]。コンピュータグラフィックスでは広い領域のシミュレーションを効率よく行なうために粒子径を動的に変更する研究も行なわれた[1]。粒子法では粒子間には接続情報ではなく、自由に移動できる。そのため、毎タイムステップごとに相互作用を行なう近傍粒子を探索しなければならない。全粒子から近傍粒子を探索すると、計算コストは

$O(n^2)$ になり、粒子数が増加するに従って計算コストが高くなる。そこで SPH の計算では固定された等間隔格子を用いてボクセルに入っている粒子のリストを作り、効率化する手法が多く用いられている [88, 23]。粒子の影響半径が一定の計算ではこのような単一の大きさのボクセルを用いる手法が効率的であるが、それぞれの粒子の影響半径が異なる場合には、Octree などの階層格子が用いられる [66]。しかし階層格子でのリーフノードのアクセスの計算コストは $O(\log n)$ であるため、粒子の影響半径が同じ場合には、等間隔格子の方が計算効率は良い。

等間隔格子の応用手法として必要なボクセルのみのメモリを確保するために、まず仮想的に等間隔格子を考え、それぞれの粒子に対してその粒子が存在しているボクセルの番号を計算する。そしてそのボクセル番号をキーとしてソートして近傍探索を効率的にするデータ構造を構築する手法もある。この手法ではデータ構造構築のために全粒子のソートが必要であり、さらにボクセルに格納されている値にアクセスするために、二分探索を行なわなければならないため、頻繁にデータ構造の再構築を行ないボクセルのメモリを読み出す必要のある粒子法シミュレーションには不向きである [105]。

近傍探索にハッシュ格子を用いる方法も存在する [119]。1つ1つの格子の座標からハッシュを計算し、3次元の等間隔格子をある長さの1次元の格子に変換する。等間隔格子を用いた場合と比べると粒子が格納されていないボクセルのメモリを確保しなくてもよいという利点がある。しかしこの手法では粒子番号を格納するのに固定の大きさの1次元配列しか必要としないが、1次元配列の1要素に必ず3次元空間内の1個の格子が割り当てられるとは限らない。つまり1要素に複数個の格子が割り当てられることがある。これをハッシュの衝突という。従って、ハッシュを用いて格納されたデータへのアクセスは固定長の処理になるとは限らない。このような処理は GPU で計算するには非効率であるため、固定長の処理でデータアクセスできるハッシュの研究も行なわれている [116]。しかしこのデータ構造の構築は計算コストが高く、粒子法シミュレーションのように毎タイムステップにおいてデータ構造を構築しなければならないようなアプリケーションには向いていない。コンピュータグラフィックスの分野ではこのように様々な格子のデータ構造についての研究が存在するが、計算力学分野では格子のデータ構造についての研究は行なわれていない。本論文で提案するスライスグリッドは今まで研究されていない。

5.3 粒子法シミュレーションに求められる条件

近傍粒子探索を効率化するために導入することのできるデータ構造は、固定された等間隔格子、ハッシュ格子、階層格子に分類される。粒子法の近傍粒子探索を行なう上で、必要な条件は3つある。1つ目はデータ構造の構築の計算コストが低いことである。これは粒子法シミュレーションでは毎タイムステップで粒子が移動するため、このデータ構造を再構築しなければならないからである。2つ目はある座標を含んだボクセルに容易にアクセスできることである。これは1タイムステップにおいて多くの近傍粒子を探索する必要があるためこのコストが低いことが望ましい。特に流体シミュレーションでは多くの近傍粒子を探索しなくてはならないので、この計算コストは重要になってくる。そして3つ目はデータ構造に用いるメモリが少ないことである。それぞれのデータ構造に対して、これらの条件を満たしているかを順に考察していく。

まず固定された等間隔格子ではデータ構造の構築の計算コストは粒子数を n とすると $O(n)$ であり、十分に低い。そして2つ目の条件も座標から数回の演算でその点を内包するボクセルの番号を求めることが可能、その計算コストは粒子数には関係ない。これらの2つの条件は良く満たしているが、計算領域のバウンディングボックス内のメモリを確保するため、粒子を格納しないボクセルのメモリも確保する必要があるため、多量のメモリを必要とする。よってメモリ効率は悪い。

ハッシュ格子は全てのボクセルがメモリの1要素に割り当てられれば、ボクセルへのアクセスが容易であり、多量のメモリも必要としないため、望ましいデータ構造である。しかし一般的にいくつかのボクセルがメモリの1要素に割り当てられるハッシュの衝突が起こるため、ボクセルのアクセスはハッシュの衝突の回数によって変わってしまう。よってデータ構築とボクセルへのアクセスのコストはハッシュの衝突が多くなるほど非効率になる。

階層格子はデータ構造の構築には $O(n \log n)$ のコストがかかるため、等間隔格子と比べるとコストは高い。階層構造にすることで複数の階層のボクセルのメモリを確保する必要があるが、それらの数は一般的に最下位の不要なボクセルの数に比べると少ないため、階層格子のメモリ効率は良い。またボクセルへのアクセスは等間隔格子のように座標からその座標を含んだボクセルの番号を直接計算することはできず、階層構造をたどつて行かなければならなく、 $O(\log n)$ の計算を行なわなければならない。この $\log n$ の計算コストは少数の粒子の計算では十分に小さいが、本研究で取り扱うような多数の粒子が存在する広い領域での計算においては大きくなる。

またアルゴリズムを GPU のような複数の計算ユニットが並列で処理を

行なうプロセッサ上で実行するときには、処理を全て並列化することができる望まれる。これはCPUで処理を行なう必要があると、それらの間で速度の遅いデータ転送を行なわなければならないためである。またもう1つの条件としては全ての計算ユニットでの負荷がほぼ均等になっていることが望ましい。

本論文では流体や粉体などの粒子法シミュレーションに適したデータ構造を提案することであり、これらのシミュレーションでは全ての粒子の大きさは一定のものが一般的であり、このような場合には単位体積あたりの粒子数がほぼ一定になる。つまり、粒子の疎密が生じないということであり、粒子が存在している部分では空間を等分割したボクセルに割り当てる粒子数はほぼ一定となる。等間隔格子は、Haradaらが示したように全てを並列化することができ[54]、さらにボクセルへのアクセスの計算コストは全て同じであるため、計算負荷のバランスもよい。ハッシュ格子はハッシュの衝突によりこれらの条件を満たすのが困難である。まずハッシュの衝突は毎タイムステップ異なるので、データを格納するために、メモリ構造を再構築しなければならないだけでなく、この処理をGPUで行なった研究はない。またボクセルへのアクセスもハッシュの衝突により、計算の長さが変わってしまうため、計算ユニットでの負荷は均一にはなるとは限らない。

これらをまとめると等間隔格子はメモリ効率が悪く、ハッシュ格子はGPU上での実装に向かない。また階層格子はデータ構造の構築とボクセルへのアクセスの計算コストが高い。本論文で提案するスライスグリッドは、前述の条件を全て満たしている。データ構造の構築の計算コストは $O(n)$ であり、ボクセルへのアクセスの計算コストも n に依存せず、等間隔格子を用いた場合とほぼ同じである。また大部分の不要なボクセルのメモリを確保しないため、データ構造に必要なメモリも小さい。さらに全ての処理をGPU上で行なうことができ、計算負荷に偏りがない。

5.4 スライスグリッド

本手法ではまず計算に用いられる空間に等間隔格子を用意する。しかしここでは仮想的に考えるだけであり、実際にメモリの確保はしない。この格子を構成する1つ1つの立方体をボクセルと呼ぶ。この格子は無限の広がりを持つものであり、計算中は空間上の配置を変更しない。そして本手法ではこれらのボクセルのうち必要なものだけをメモリ上に保持する。等間隔格子を用いる場合だと計算領域のバウンディングボックスを定め、その内部のボクセルのためのメモリを確保していた。

本手法は計算空間に用意されたボクセルの1軸に直交するスライスに格

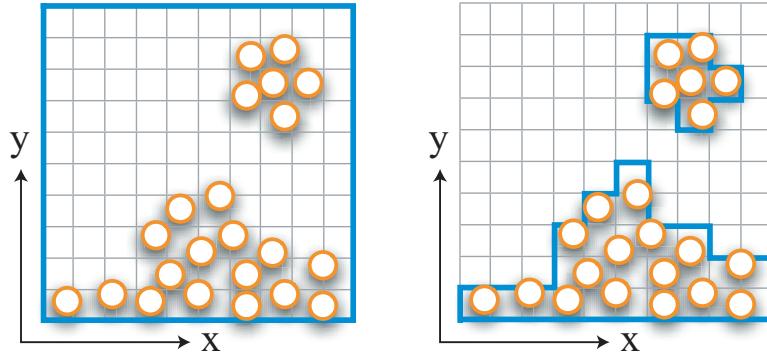


図 5.1: 空間固定されたユニフォームグリッドとスライスグリッド. それぞれメモリに確保される領域は青線で囲まれた領域内である.

子を分割する. スライスの次元は計算領域の次元よりも 1 次元低いものになる. この分割ではそれぞれのスライスが軸方向に 1 ボクセルだけ持つようにする. 計算領域が 3 次元であり, Y 軸方向を軸と定めるとスライスは XZ 軸方向に広がりを持つ 2 次元格子になる. 以下ではこのように 3 次元の計算領域を Y 軸方向に垂直なスライスに分割した場合を例として説明していく. 計算領域をスライスに分解した後, それぞれのスライスにおいて XZ 軸方向のバウンディングボックスを定義する. つまり XZ 軸方向における最大と最小のボクセルの開始座標 $b_{i,max}^x, b_{i,min}^x, b_{i,max}^z, b_{i,min}^z$ を計算する. ここで i はスライスの番号である. するとスライス i 上の XZ 方向のボクセルの数 n_i^x, n_i^z は以下のように計算できる.

$$n_i^x = \frac{b_{i,max}^x - b_{i,min}^x}{d} + 1 \quad (5.1)$$

$$n_i^z = \frac{b_{i,max}^z - b_{i,min}^z}{d} + 1 \quad (5.2)$$

d はボクセルの 1 辺の長さである. これらを用いてスライス i 上でのボクセルの数を $n_i = n_i^x n_i^z$ と求めることができる. 本手法ではスライス内のバウンディングボックス内の領域のみ, メモリを確保する.

ここで計算領域が n 個のスライス $\{S_0, S_1, S_2, \dots, S_{n-1}\}$ に分解されるとする. そしてスライス S_i での先頭のボクセルの番号 p_i はスライス S_0 から S_{i-1} にあるボクセルの数の和として

$$p_i = \sum_{j < i} n_j \quad (5.3)$$

と計算する. するとある点 x, y, z が存在するボクセルの番号を計算するために, まず Y 軸方向の最小のボクセルの開始座標 b_{min}^y を用いて, その点

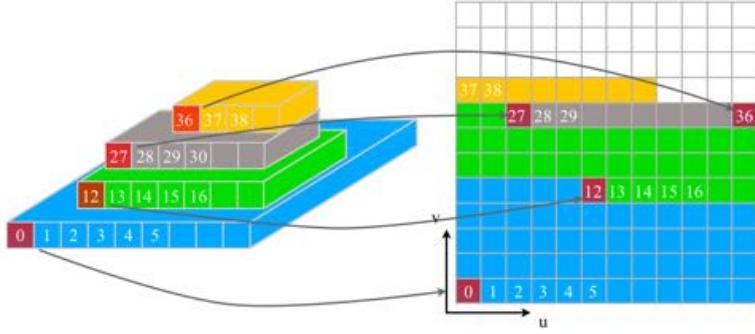


図 5.2: 空間に存在する格子とそのメモリ配置。ここでは 2 次元配列のメモリ上での配置を示す。メモリに確保されるボクセルはスライスを一方に向かに順に格納される。

が含まれるスライスの番号を以下のように計算する。

$$i = \left[\frac{y - b_{min}^y}{d} \right] \quad (5.4)$$

そしてボクセル番号 $v(x, y, z)$ は

$$v(x, y, z) = p_i + \left(\left[\frac{x - b_{i,min}^x}{d} \right] + \left[\frac{z - b_{i,min}^z}{d} \right] n_i^x \right) \quad (5.5)$$

と計算することができる。式 (5.5) より、ある点が存在するボクセル番号を計算するために必要な値は各スライスのバウンディングボックスを定義する値 $b_{i,min}^x, b_{i,min}^z, n_i^x$ 、スライス番号を決定する値 b_{min}^y 、各スライスの先頭番号 p_i 、そしてボクセルの一辺の長さ d であることがわかる。このデータ構造を用いると図 5.1 右に示すようにメモリ効率よく格子を作成することができる。このデータ構造をスライスグリッドと呼ぶ。

5.5 結果

本手法を Core2 X6800 CPU, GeForce 8800GTX GPU, 2GB のメモリを搭載した PC 上で、プログラムは C++, OpenGL, C for Graphics を用いて開発し、粒子法シミュレーションに組み込んだ [36, 15]。

DEM の 3 次元シミュレーションの計算例を図 5.3 に示す。65,536 粒子を用いたシミュレーションであり、立方体の容器内で柱状に配置した粒子を落下させた計算を Z 軸方向に正射影して描画した結果である。レンダリングにはポイントスプライトを用いて粒子位置に球を描画した。本手法を用いずに等間隔格子を用いた場合は赤線で囲まれた領域全体を格子として

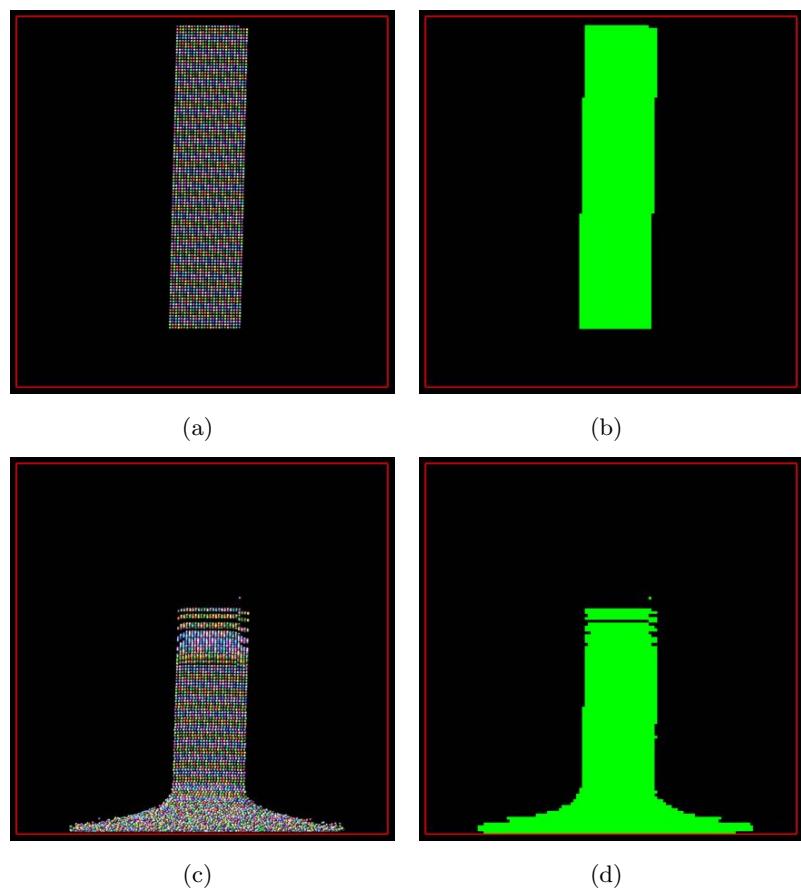


図 5.3: 3 次元の DEM によるシミュレーションと、そのときのスライスグリッド。図右列の緑色の領域がメモリに確保された領域

128^3 ボクセルのメモリを確保しなければならないが、本手法を用いた場合は緑色で示した領域のみメモリを確保すればよい。等間隔格子を用いた場合は粒子番号を格納するメモリとして 32MB 必要なのに対して、図の計算例では最大約 2MB しか必要なかった。本手法を用いると格子が粒子分布にタイトになり、メモリ効率が良くなっていることがわかる。表 5.1 は 128^3 のボクセルに必要なメモリ量に対する、本手法を用いた計算において粒子番号を格納するメモリとして確保したメモリの量の割合を示している。本手法を用いると等間隔格子を用いた場合の約数パーセントから数十パーセントのボクセルのみメモリを確保すれば良いことがわかる。しかし粒子数 589,824 の場合は計算領域の約半分の量のボクセルのメモリを確保しなければならなかった。表 5.2 には DEM のシミュレーションで総粒子数を変えて計算したときの計算速度の比較を示す。この表の格子の作成の時間とはスライスを決める値を計算し、粒子番号をプールに格納するまでの処理である。表より全ての計算例において格子作成の時間は本手法を用いた場合の方が速いことがわかる。しかし、この時間の差は総粒子数が多くなるほど小さくなっている。これはまた、本手法を用いた時に用意しなければならないボクセル数が、等間隔格子を用いた場合に近づくほど大きくなっているとも見ることができる。これは本手法ではスライスを決める値を計算する必要があるが、プールバッファの大きさが等間隔格子のバッファよりも小さいからであり、バッファが大きいほどバッファに書き込むときの用意に要するオーバーヘッドが長いことに起因すると考えられる。また総計算時間は本手法を用いた場合の方が速いが格子の作成の時間の差よりも計算時間の差は縮まっている。これは座標からその座標が入っているボクセルに格納されているデータにアクセスするときに等間隔格子よりも多く計算を行なわなければならないからであると考えられる。

次に本手法を SPH の 3 次元シミュレーションに用いた場合の計算時間と等間隔格子を用いた場合の計算時間の比較を表 5.3 に示す。計算領域の広さは粒子径を 1 とすると $256 \times 256 \times 256$ である。すなわち等間隔格子を用いた場合は 256^3 ボクセル分のメモリを確保しなければならない。この等間隔格子は 256MB のメモリを必要とするのに比べ、本手法を用いた 1,048,576 粒子のシミュレーションでは最大約 15MB のメモリで計算することができた。表から本手法を用いた場合の方が計算時間が短くなっていることがわかる。例えば表 5.2 の 65,536 粒子を用いた場合の格子作成の計算時間の差と、表 5.3 の 1 タイムステップの計算時間の差を比較すると 1 タイムステップの計算時間の差の方が大きい。つまり格子生成以外の計算においても本手法を用いた場合の方が計算時間が短いということである。格子生成以外の計算では粒子番号をバッファから読み出す部分しか差がない。つまり、粒子番号をバッファから読み出す部分において、本手法

の方が計算量が多いけれども速度は速いということである。これは本手法を用いた場合の方がキャッシュヒット率が高くなつたことが原因であると考えられる。本手法を用いた場合は格子バッファに粒子番号がより密に格納されているため、粒子が存在するボクセルの値がより多くキャッシュに残るようになったと考えられる。¹

図5.4は2個の計算例のスクリーンショットとそのタイムステップでのプールのバッファを示したものである。流体シミュレーション結果のレンダリングにはDEMと同様にポイントスプライトを用いたが、1次元バッファから密度の値を用いて色を計算した。密度が低いものは白くレンダリングされている。プールのバッファの黒いテクセルは粒子番号が格納されていないボクセルであり、赤、黄、白のテクセルはそれぞれ1、2、3個のインデックスが格納されているものである。計算領域はY軸に垂直な面でスライスされており、スライスは図5.2に示すように計算領域の下からプールバッファに格納されている。プールバッファの縦横はバッファ座標の u, v の軸である。よってバッファの上部は計算領域の上部のボクセルを格納している。図上の計算例は本手法が特に効果的であるシーンである。プールのバッファを見ても密にデータが格納されている。図下の例でも比較的密にデータが格納されているが、バッファ上部に空のボクセルが見られる。これらは液滴によって水面が乱されている部分によるものである。

5.3章で述べたように階層格子はボクセルへのアクセスの計算コストは高いがメモリ効率は良い。そこでスライスグリッドと階層格子の1つであるOctreeのメモリ消費量を比較した結果を図5.7に示す。この結果は図5.3のシミュレーションにおけるメモリ消費量である。計算開始直後は粒子は直方体内に整列しており、この時にはスライスグリッドはメモリを無駄にすることがなく、確保するメモリ量はOctreeの葉ノードのメモリ量と等しい。しかしOctreeでは階層構造を構築しなければならないため、スライスグリッドよりも多くのメモリを必要とする。計算が進むにつれ、スライスグリッドでは無駄なボクセルのメモリも確保してしまうため、Octreeよりも多くのメモリを使用するという結果になった。しかし図5.7の最後の状態での等間隔格子のメモリ消費量に対するスライスグリッドとOctreeのメモリ消費量はそれぞれ6.25%，3.24%であったため、スライスグリッドは階層格子ほどではないが、効果的にメモリ消費量の削減ができていることがわかる。

また本手法は等間隔格子を用いないため、計算領域の大きさの制限を受けない。そのため、広い領域であっても容易に計算することが可能である。図5.6に広い領域内で計算を行なつた例を示す。この計算は約4,000,000粒子を用いた計算結果であり、計算領域の広さは粒子径を1とすると700×

¹GPUのキャッシュサイズについては明らかにされていない。またGPUのキャッシュヒット率は低いが構造に関してはCPUとほぼ同じ構造である[90]。

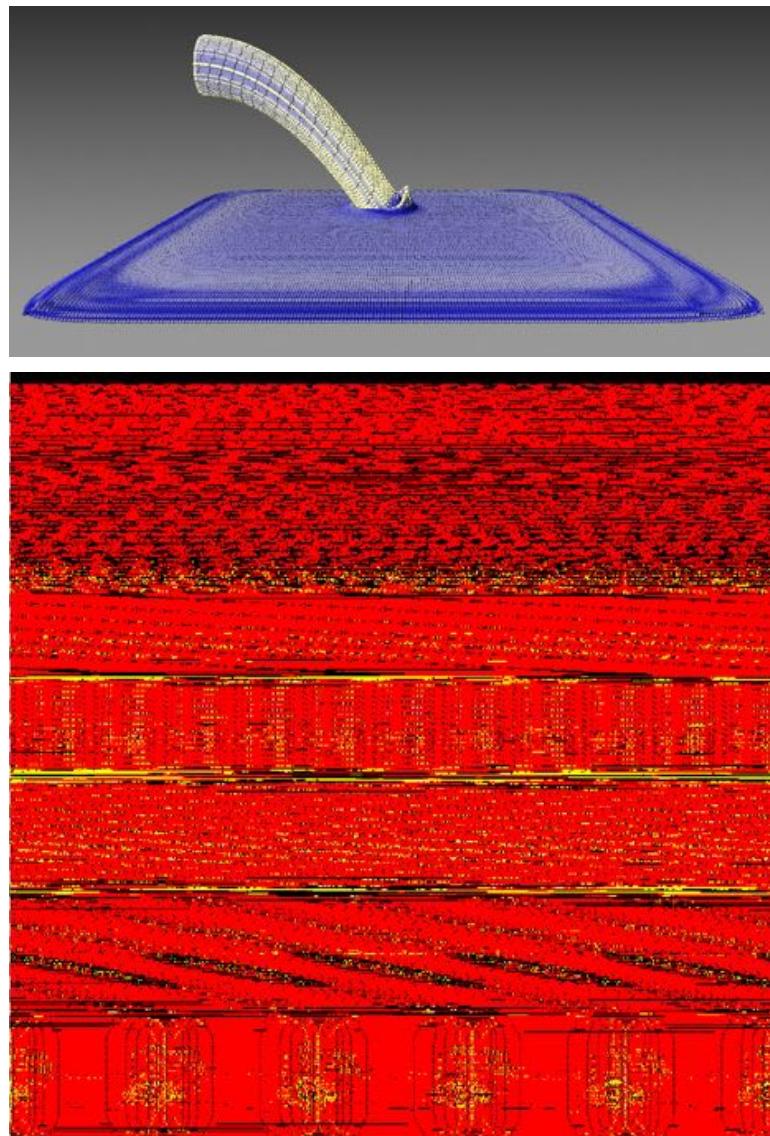


図 5.4: SPH による流体シミュレーションと、メモリ内を可視化したもの。
図下において黒いテクセルは空のボクセルを示し、赤、黄、白色のテクセルはそれぞれ 1、2、3 個のデータが格納されている。

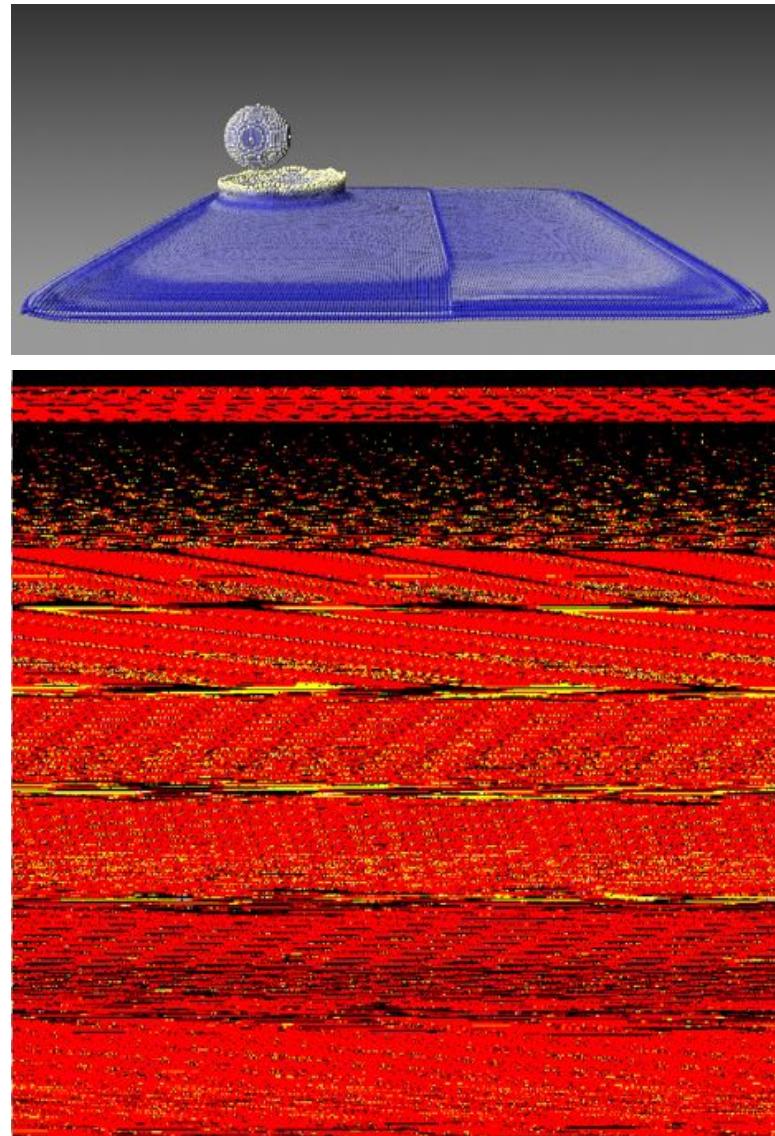


図 5.5: SPH による流体シミュレーションと、メモリ内を可視化したもの。
図下において黒いテクセルは空のボクセルを示し、赤、黄、白色のテクセルはそれぞれ 1、2、3 個のデータが格納されている。

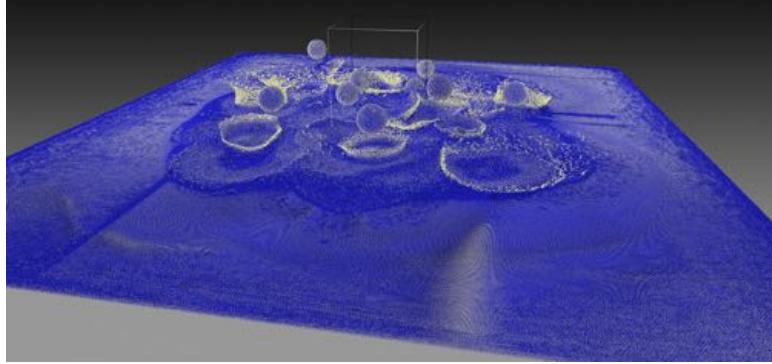


図 5.6: 約 4,000,000 粒子を用いた広域の流体シミュレーション.

700×256 である。つまり等間隔格子を用いると、粒子番号を格納する格子だけで約 2GB のメモリが必要になる。このメモリの大きさは現在利用できるグラフィックスカードの最大メモリ量を超えていたため、等間隔格子を用いたのでは不可能な領域の大きさの計算である。しかし本手法を用いた場合は最大約 150MB のメモリで計算することができた。

本手法では 1 次元テクスチャの最後の 1 テクセルの 1 チャネルの値を読み出すことでプールに必要な最大粒子数がわかる。少ないデータ量の通信で最大粒子数がわかるという点も本手法の利点の 1 つである。これを用いると容易にプールの大きさを変えることも可能である。毎タイムステップ最大粒子数を CPU に読み戻して動的に必要な大きさのプールテクスチャを用意するプログラムも開発したが、計算時間を測定すると固定の大きさのプールテクスチャを用いた場合の方が高速であった。これはメモリの解放と確保に時間がかかったためであると考えられる。そのため、議論してきた計算速度はプールの大きさは動的に変化させず固定の大きさにしたプログラムを用いたものである。本手法を用いると前のタイムステップで用いたプールのメモリの大きさと、次のタイムステップで用いたプールの大きさは前述のように容易に比較することができる。よって毎タイムステップでプールの大きさを変更しなくとも、確保しているプールの大きさよりも大きなメモリが必要になったときにメモリを余裕を持って大きくすることによって、メモリ解放と確保の回数を減らして動的にプールの大きさを変更することができる。

5.6 結論

本論文では粒子法シミュレーションの近傍探索に有用なスライスグリッドを提案した。このデータ構造を用いることで粒子配置にタイトに計算格

表 5.1: スライスグリッドを用いた計算におけるユニフォームグリッドを用いた場合とスライスグリッドを用いた場合におけるメモリに格納されるボクセル数の割合(スライスグリッド/ユニフォームグリッド).

Number of particles	Ratio of voxels
16,386	0.0394
65,536	0.0619
262,144	0.269
589,824	0.488

表 5.2: 128^3 の計算領域における DEM のシミュレーションの計算時間(ミリ秒). ユニフォームグリッドを用いた場合のグリッド構築時間を (a) に示し, スライスグリッドを用いた場合を (b) に示す. また 1 タイムステップにかかった計算時間を (c), (d) に示すが, これらはそれぞれユニフォームグリッドとスライスグリッドを用いた計算時間である. また (e) にメモリに確保された最大のボクセル数を示す.

Number of particles	(a)	(b)	(c)	(d)	(e)
16,386	1.48	0.688	2.53	2.19	82,675
65,536	3.91	2.56	9.68	8.58	129,781
262,144	13.9	10.3	42.0	36.9	564,354
589,824	31.7	23.0	96.4	90.9	1,022,551

表 5.3: 256^3 の計算領域における SPH のシミュレーションの計算時間(ミリ秒). (a), (b) はそれぞれユニフォームグリッド, スライスグリッドを用いた場合の 1 タイムステップの計算時間.

Number of particles	(a)	(b)
65,536	60.9	53.1
262,144	280.5	231.3
589,824	685.9	567.9
1,048,576	1160.9	1070.3

表 5.4: GPU(a) と CPU(b) でのスライスグリッドの構築の時間の比較.

Number of particles	(a)	(b)
16,386	0.688	31.2
65,536	2.56	37.4
262,144	10.3	68.7
589,824	23.0	117.2

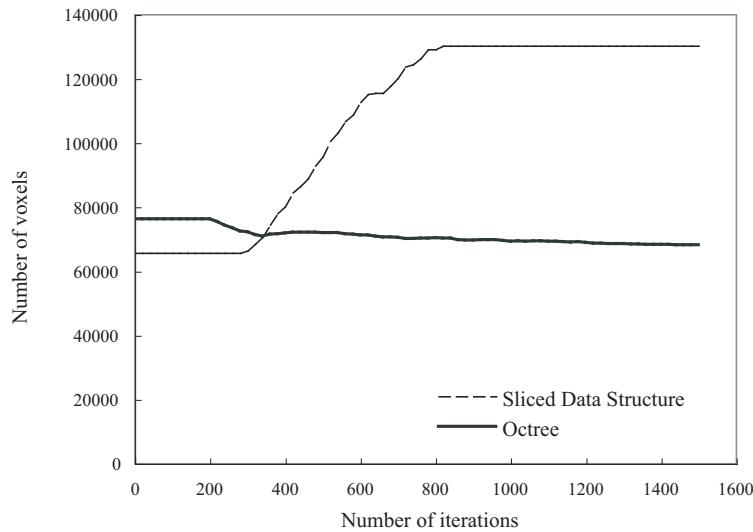


図 5.7: スライスグリッドと Octree のメモリを確保するボクセル数の比較.

子を作成することができるようになった。スライスグリッドではボクセルのアクセスは固定された等間隔格子とほぼ同じ計算コストであり、階層格子に近いメモリ削減を実現した。またこのデータ構造の構築等を GPU を用いて実装し、GPU を用いた DEM と SPH の粒子法シミュレーションに導入し、手法の定量的評価を行なった。本手法を用いることによって格子に用いるメモリの量が減り、メモリ効率が上がっただけでなく、SPH の流体計算においては計算時間の向上も見られた。本論文で提案手法の有用性を示した。

第II部

粒子法シミュレーションの GPUを用いた高速化

第6章 近傍粒子探索のGPUを用いた高速化

6.1 序論

現在のプロセッサテクノロジは周波数を向上させるのではなく、効率性を向上させる方向に向かっている。並列アーキテクチャを備えたプロセッサも多く存在する。Cell Broadband Engine (Cell BE) Architecture は汎用計算用に開発されたマルチコアプロセッサであり、Graphics Processing Units (GPUs) はグラフィックス処理に特化した並列プロセッサである。さらに CPU もマルチコア化されてきている。よって我々は今後これらの計算能力を最大限に活用することができる並列計算アルゴリズムを開発していかなければならない。

粒子ベースシミュレーションはメッシュを用いない計算手法である。粒子間で接続情報を持たないため、近傍粒子を毎タイムステップで探索しなければならない。視覚的に十分な結果を得るためにには、多くの粒子を用いなければならないが、多くの粒子の挙動をリアルタイムで計算するには計算コストが高かった。既存の GPU を並列計算機としてシミュレーションに用いた研究は全て計算ノードの接続関係が計算中に変化しないという共通点を持っていた。例えば格子を用いた流体計算では各計算点はその計算点を囲む点のみと相互作用し、布のシミュレーションでは質点同士の接続は計算中に変化しない [61][27]。しかし粒子ベースシミュレーションは、計算粒子は自由に動くことが可能であり、タイムステップごとに相互作用する計算粒子が変化する。そのため、計算粒子の近傍粒子を探索しなくてはならない。接続関係が動的に変化する粒子法シミュレーションを GPU を用いて高速化する際に問題になるのは近傍粒子探索であり、今までそれを実現した研究は存在しない。本研究ではこの近傍粒子探索を格子を用いて GPU 上で実現する手法を提案する。本手法により粒子法シミュレーションを含め、計算要素間の接続関係が動的に変化するシミュレーションを GPU を用いて高速化することが可能になる。

本章ではまずユニフォームグリッドを用いた近傍探索手法を GPU 上で行なう手法について述べたあと、5 章で提案したスライスグリッドを GPU 上で行なう手法について述べる。本手法は Distinct Element Method, Smoothed Particle Hydrodynamics, 粒子法を用いた剛体シミュレーション、粒子法を用いた弾性体シミュレーション、そしてそれらの連成計算を高速化することが可能である。また粒子法以外の計算要素間の接続関係が変化しないメッシュを用いたシミュレーションと粒子法シミュレーションを連成させることも可能になる。粒子法以外にも一般的な衝突判定アルゴリズムなどに応用することができる。

6.2 関連研究

GPU はグラフィックスの処理に特化するため、内部はプロセッサが並列に処理をするという構造になっている。GPU は頂点の座標変換などの処理ならば CPU をはるかに超える速度で行なうことができる。GPU はグラフィックス専用のプロセッサであったが、シェーダと呼ばれるプログラムを書くことによって処理を制御することができるようになり、グラフィックス以外の処理に用いることができる可能性が出てきた [36]。そこで GPU を汎用的に利用する研究が多く行なわれている [100, 101, 35]。セルオートマトンや [62]、格子を用いた流体の計算 [61, 63]、布のシミュレーション [27]、ボクセル化 [47] などである。また粒子法を GPU で高速化する計算も行なわれてきた。Kipfer らは完全な粒子間の衝突を計算しないシミュレーションを GPU 上で行なった [71]。また SPH のシミュレーションに関しては Amada らは近傍粒子探索以外の処理を GPU を用いて高速化した [2]。Kolb らは物理量を一度格子に分配する手法を研究したが、この手法では数値拡散が起こってしまう [74]。Havok は GPU で狭域衝突計算を高速化した Havok FX を開発し、剛体シミュレーションを高速化したが、広域衝突計算は CPU で行なっているため GPU の性能を完全に引き出せていない可能性がある [16]。

6.3 ユニフォームグリッドを用いた近傍粒子探索の GPU での実装

6.3.1 データ構造

近傍粒子探索を行なうためにグリッドを利用するが、このグリッドを GPU 上で構築し、そのデータを GPU 上でのシミュレーションに用いるためにはグリッド自体をビデオメモリ上に確保しなければならない。3 次元計算だと、計算領域を囲むグリッドは 3 次元格子である。しかし shader を用いる場合は 3 次元格子に直接出力することができないため、Harris らが行なったように 2 次元格子を複数枚並べた 1 枚のテクスチャとしてメモリ上に確保する [61]。そしてグリッドを構成するボクセル 1 個に 1 ピクセルを割り当てる。

6.3.2 グリッド構築

グリッド構築はボクセル内に存在する粒子番号を、対応するボクセルに格納する処理である。この処理を逐次的に処理する場合は、問題は起きないが、GPU 等を用いてデータを経列処理する場合には問題が生じる。

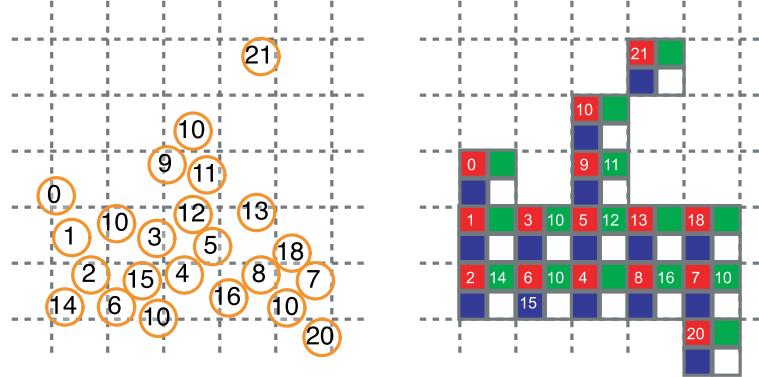


図 6.1: 粒子番号の格子への格納

GPU を用いる場合はこの処理は以下のように行なわれる。まずそれぞれの粒子を各スレッドが受け持ち、粒子座標からグリッド上の座標を計算する。そしてその粒子番号をそのグリッドに書き込む。1粒子のみがそのボクセルに存在する場合には問題は起きてないが、複数の粒子が同じボクセルに存在する場合には問題が生じる。それはそれぞれのスレッドが並列に実行されており、それぞれの処理内容を把握していない。そのため複数の粒子が同じボクセル上に存在する場合、複数のスレッドが同時に同じメモリに書き込み、全ての粒子番号を正しく格納することができない。そこでこの問題を解決するために以下の手法を提案する。

まずそれぞれのボクセルに格納される最大粒子数を定義する。そしてそれぞれのボクセルに対して最大粒子数のメモリを確保する。非圧縮流体などの計算を粒子法で行なう時には、粒子は初期の粒子数密度を保つように運動するため、理論的には粒子同士が重なることはない。そこでボクセルの一辺の長さを粒子径にすると 1 ボクセル内に存在する粒子の重心座標の数は最大でも 4 個である。以下の応用では粒子径と同じ長さの一辺を持つボクセルを用意し、それぞれのボクセルに 4 要素のメモリを確保する。

この手法は粒子番号をボクセルに格納する処理を、複数回に分けて行なう。複数個の粒子があるボクセルに存在する場合には 1 粒子のみの粒子番号を 1 回の処理でボクセルに格納する。1 回目の処理では書き込み先を 1 番目のメモリに設定し、2 回目の処理では書き込み先を 2 番目のメモリに設定する。そして 2 回目以降の処理では既に書き込まれた粒子を受け持っているスレッドの処理を無効にしてまだ書き込まれていないスレッドのみ書き込みを行なうようとする。

Shader

Shader を用いた実装では各ボクセルの要素を格納するメモリをテクスチャの 1 テクセルにする。1 テクセルには Red, Green, Blue, Alpha の 4 チャンネルが存在するため、これらを 4 個のメモリスロットとして用いて、最終的に図 6.1 のように RGBA チャンネルに粒子番号を格納する。粒子番号を粒子が存在するメモリに書き込むという処理は scattering operation であるため、フラグメントシェーダでは行なうことができない。そこで任意のメモリに書き込むことができるバーテックスシェーダを用いる。粒子数と同数の頂点を用意し、それぞれ対応付ける。バーテックスシェーダは粒子座標が格納されているテクスチャを読み、その粒子座標を用いてグリッド上の座標を計算する。そしてその座標に対応するメモリに 1 テクセルの大きさの点として頂点を出力する。このときに頂点の色を粒子番号とすることで、粒子番号をグリッドに格納することができる。また前述のように 2 回目以降の処理において既に書き込まれた粒子を受け持っているスレッドの処理を無効にするために、頂点の色だけでなく深度も粒子番号として出力する。それぞれの処理において深度テスト、ステンシルテスト、カラーマスクという GPU のグラフィックスの機能を用いることによって、書き込む粒子番号と書き込み先のメモリを選択する。

あるボクセル b に格納される小さい順に並んだ粒子番号 i_0, i_1, i_2, i_3 を考える。そしてこの粒子番号をボクセル b に割り当てられたピクセルの RGBA チャンネルに 4 回の処理で格納する。1 回目では i_0 を R チャンネルにレンダリングする。 i_0 に対応する頂点は最も小さい深度値を持つ頂点としてレンダリングされるため、デプスバッファを最大値で初期化し、小さい値がデプステストを合格するように設定することで i_0 を書き入れることができる。2 回目では i_1 を G チャンネルにレンダリングするが、このときにすでに値を書き入れた R チャンネルに上書きしないためにカラーマスクを行ない R チャンネルには値を書き入れないようにする。1 回目で用いた深度バッファを用いるがデプステストは大きな値が合格するように設定することで i_0 は書き込まれなくなる。しかしデプステストだけだと最も深度値が大きい i_3 が最終的に書かれてしまう。 i_2, i_3 の書き込みを防ぐため、ステンシルテストを用いる。ステンシルファンクションでは書き込まれるごとにステンシル値を増加させるように設定し、ステンシルテストは 1 より大きい値を不合格にするように設定する。こうすることで各ピクセルには 1 回値がレンダリングされたあとは書き込まれなくなる。つまり i_1 が書き込まれた後は値を書き込むことができなくなるの G チャンネルに選択的に i_1 を書き込むことができる。3 回目と 4 回目では基本的に 2 回目で行なった処理と同じ処理を行なう。しかし 3 回目では RG チャンネル、4 回目では RGB チャンネルへの書き込みを防ぐため、カラーマスク

を用いる。そして深度バッファは初期化せずに使い続け、ステンシルバッファはそれぞれの処理ごとに初期化する。

CUDA

CUDA を用いた場合にも同じように 4 回の処理でメモリに粒子番号を格納することができる [99]。ただし CUDA を用いる場合には shader を用いる場合よりも容易になる。それはメモリアクセスの制限が緩和されているため、書き込み先のメモリを自由に選択することが可能であるため、様々なテストを用意せずに書き込み先を選択可能である。しかし処理の流れは shader を用いた場合と全く同じである。それぞれの処理では 1 粒子の粒子番号のみ指定したメモリに書き込む。既に書き込まれた値のチェックはそのボクセルに割り当てられたメモリを読み、そのスレッドに割り当てられた粒子番号と比較することによって行なうことができる。

6.4 スライスグリッドを用いた近傍粒子探索の GPU での実装 (shader)

スライスベースデータ構造を構築するにはまずスライスのバウンディングボックスを決定し、スライスの先頭番号をそれから計算する(図 6.2)。このようにしてボクセルの番号を計算する値を求めた後、ボクセルに値を書き込んで行く。

6.4.1 データ構造

Shader を用いた GPU で計算を行なうためには、データをテクスチャとしてビデオメモリに格納しなければならない。まず各スライスに必要な値を格納するために、1 次元テクスチャを用意する。スライスの数が1次元テクスチャの大きさの最大値を超えるならば2次元テクスチャにすることもできる。各スライスに必要な値は $p_i, b_{i,min}^x, b_{i,min}^z, n_i^x$ の 4 個であるため、1 次元テクスチャの RGBA チャネルにこれらを格納すれば 1 枚のテクスチャで十分である。また格子に保持する値を格納するための 2 次元テクスチャを用意する。このテクスチャをプールテクスチャと呼ぶ。ここでは粒子番号をプールテクスチャに格納することを例として説明する。本研究では Harada らの研究で行なわれたように、1 個のボクセルに 1 テクセルを割り当て、最大 4 個の粒子番号を格納するようにした [54]。この場合では 2 次元テクスチャのテクセル数がプールテクスチャに格納できるボクセルの最大値である。

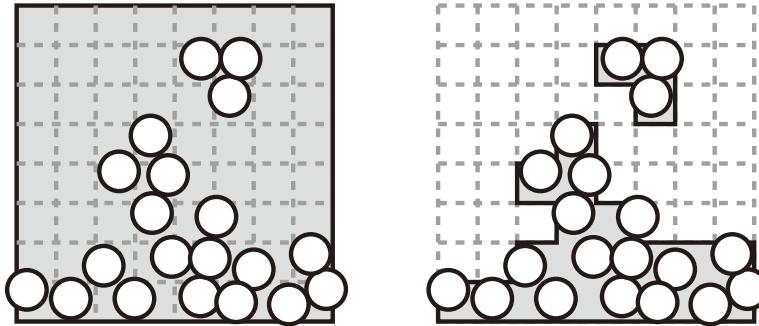


図 6.2: ユニフォームグリッドとスライスグリッドを用いた場合におけるメモリに格納される格子をそれぞれ図左と図右に示す。灰色の領域内がメモリに格納される。

6.4.2 スライスのバウンディングボックスの決定

スライスでのバウンディングボックスを決定するためにまず各スライスでの最大と最小のボクセルの開始座標を計算する。この処理は粒子の Y 座標からスライスの番号を計算し、各スライスに存在する粒子座標の最大値と最小値を計算することによって行なうことができる。GPU を用いる場合は粒子 1 個に頂点を 1 個割り当て用意した 1 次元テクスチャに書き込む。バーテックスシェーダでは粒子の Y 座標から式 (5.4) でスライス番号、つまり 1 次元テクスチャ上に書き込む位置を計算し、その位置に頂点を出力する、そしてフラグメントシェーダで XZ 座標を色として出力する。最大値最小値の選択はアルファブレンディング¹を用いて行なうことができる [79]。

6.4.3 先頭番号の計算

各スライスでの XZ 座標の最大値と最小値を用いることでそのスライス上のボクセルの数を計算することができる。先頭番号の計算は各スライスにおいて式 (5.3) を計算しなければならない。この処理をスライス i に対して n_0 から n_{i-1} までの値を読み出して計算することになる。スライスの総数が m だとすると最も多くメモリアクセスを行なわなければならない計算ユニットは $(m(m+1))/2$ 回メモリアクセスが必要になり、メモリ転送速度に処理の速度が制限される。GPU を用いるとこの処理はフ

¹アルファブレンディングとは指定されたアルファ値を用いて半透明な色を重ね合わせて描画することである。アルファ値を用いてバッファの色と描画する色をどのように混合させるかを指定できる。機能の中に色の最大値を描画するようにする機能があり、それをここでは使用した。

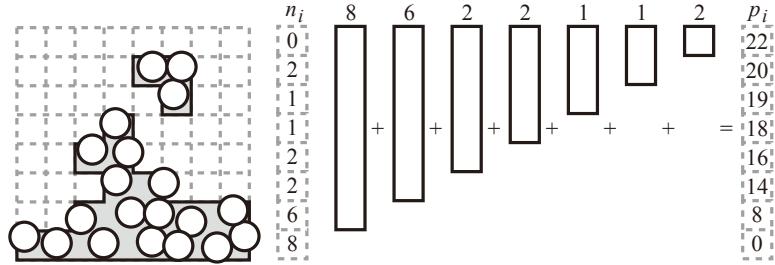


図 6.3: 先頭番号の計算は、それぞれのスライスに線を用意し、それらを加算ブレンディングしながら描画することで行なわれる。

グメントシェーダを用いた Gathering Operation になる。GPU はメモリ バンド幅は広いが、転送速度は遅いため、計算の並列度を高くしてこの点を隠蔽している。そのため、1 個の計算ユニットのメモリアクセス回数を減らし、並列度を高めた方が効率的に処理ができる。行列やベクトルの Reduction が、類似した計算であり 1 つの計算ユニットで多くのメモリアクセスを行なうより、何段階かに処理を分けて、1 個の計算ユニットのメモリアクセスを減らし並列度を高めて行なった方が効率が良いことが知られている。[77, 18]。

そこで本手法ではスライス i に存在するボクセル数 n_i を読み出して $i+1$ から m 番目のスライスの先頭番号配列に加算していく。こうすることによって m 回のメモリアクセスで先頭番号を計算することができる。この処理は Scattering Operation であり、GPU を用いる場合はバーテックス シェーダを用いて行なうことができる。この処理はスライスが m 個あるとすると要素数 m の 1 次元配列に書き込む処理である。つまり GPU を用いる場合は m テクセルの 1 次元テクスチャに書き込むことになり、スライス i に存在するボクセル数 n_i を $i+1$ から m 番目のスライスの先頭番号配列に書き込む処理は、 $i+1$ 番目のテクセルから m 番目のテクセルまでの線分を描画することで行なうことができる。このときにそのスライス内のボクセルの数を色として出し、アルファブレンディングで加算処理を行なうことで式 (5.3) を評価し、各スライスの先頭番号を計算することができる [79]。つまり各スライスごとに線分を用意する必要がある。図 6.3 はこの操作を示したものである。

6.4.4 値の格納と読み出し

このようにして計算したスライスの先頭番号とバウンディングボックスを定義する値を用いて、粒子番号をプールテクスチャに格納していく。

6.5. スライスグリッドを用いた近傍粒子探索の GPU での実装 (CUDA)85

粒子の Y 座標からスライス番号を計算して、そのスライスの先頭番号 p_i とスライスでの最小値 $b_{i,min}^x, b_{i,min}^z$ 、スライスでの X 軸方向のボクセル数 n_i^x を読み出す。これらを用いて式 (5.5) を計算し、その粒子に割り当てられているボクセルのインデックスを計算することができる。計算されたインデックスに対応するテクセルへの値の書き込みは、粒子 1 個に頂点を 1 個割り当てて、そのテクセルに粒子番号を色として書き込むことで行なうことができる。図 5.2 にスライスのボクセルがどのようにテクスチャに格納されているかを示す。下のスライスのボクセルからテクスチャに格納される。

ある座標が存在するボクセルに格納されている値の読み出しは、その座標から値の格納時と同様にボクセルのインデックス、つまりメモリのアドレスを計算することによって、行なうことができる。

6.5 スライスグリッドを用いた近傍粒子探索の GPU での実装 (CUDA)

前節ではスライスグリッドをシェーダとグラフィックス API を用いて GPU で実装したが、この実装はアルファブレンディングなどの GPU のグラフィックス特有の機能を必要とする [135]。しかし CUDA ではそれらの機能を用いることができないため、GPU をより一般化されたストリームプロセッサととらえて実装を行う必要がある [99]。そのため前節のスライスグリッドの実装をそのまま用いることはできないため、本節では CUDA でのスライスグリッドの実装方法を述べる [138]。

6.5.1 データ構造の構築

スライスグリッドを用いるためには均一格子とは異なり、まずデータにアクセスするためにそれぞれのスライスのバウンディングボックスと先頭ボクセルの番号を計算しなければならない。これらのデータの計算は数段階に分けて行なわれる。

バウンディングボックスの計算

まずは全てのスライスのバウンディングボックスを計算する。この計算は全ての粒子の格子空間での座標を計算し、それぞれのスライスに存在する粒子の格子の座標系の XZ 軸方向の最大値と最小値を求める処理である。1 コアの CPU など逐次処理を行なうプロセッサでは、この計算を行なうことは容易である。しかしこの処理はそれぞれの粒子の値を現在のス

ライスの値と比較する必要があるため、GPUなどの複数のコアがデータを並列に処理するプロセッサを用いた場合には容易に計算することができない。

そこで全粒子を複数の組に分割し、部分的に粒子の組の(ローカルな)バウンディングボックスを並列で計算し、最終的に全粒子の(グローバルな)バウンディングボックスを計算する方法を提案する。ここでは2段階でグローバルなバウンディングボックスを計算する場合を取り説明するが、何段階に分けて計算してもよい。まず n 個の全粒子を m 組に分け、さらにそれぞれの組を l 組に分ける。つまり1番目の分割ではそれぞれの組に n/m 個の粒子が存在し、2番目の分割ではそれぞれの組に $n/(ml)$ 個の粒子が存在する。粒子 $\{a|0 \leq a < n\}$ のスライス i のバウンディングボックスを定義する値の組を B_i とし、 m 組のうちの組 j の粒子 $\{a|jn/m \leq a \leq (j+1)n/m - 1\}$ のバウンディングボックスを定義する値の組を B_{ij} とし、組 j を分割してできた l 組のうちの組 k の粒子 $\{a|(jl+k)n/(ml) \leq a \leq (jl+k+1)n/(ml) - 1\}$ の値の組を B_{ijk} とする。

はじめに ml 個の組全てにおいて B_{ijk} を計算する。この計算は1スレッドが1組を受け持ち、組 l に割り当てられた粒子 a のボクセルの番号 (b_a^x, b_a^y, b_a^z) を用いて以下のように計算する。

$$\begin{aligned} B_{ijk,max}^x &= \max_{a \in P_i} \{b_a^x\} , \quad B_{ijk,min}^x = \min_{a \in P_i} \{b_a^x\} \\ B_{ijk,max}^z &= \max_{a \in P_i} \{b_a^z\} , \quad B_{ijk,min}^z = \min_{a \in P_i} \{b_a^z\} \end{aligned}$$

ここで $P_i = \{a|b_a^y = i, (jl+k)n/(ml) \leq a \leq (jl+k+1)n/(ml) - 1\}$ として、組 l に割り当てられた粒子の中でスライス i に含まれる粒子の番号とする。1スレッドの処理は直列化されており、ここで値の比較は各スレッドしかアクセスできない値の比較なので問題なく計算することができる。

次に計算された ml 個の B_{ijk} から m 個の B_{ij} を求める。各組 j を1スレッドが受け持ち、その組を分割して計算された l 個の B_{ijk} から1組の B_{ij} を以下のように求める。

$$\begin{aligned} B_{ij,max}^x &= \max_k \{B_{ijk,max}^x\} , \quad B_{ij,min}^x = \min_k \{B_{ijk,min}^x\} \\ B_{ij,max}^z &= \max_k \{B_{ijk,max}^z\} , \quad B_{ij,min}^z = \min_k \{B_{ijk,min}^z\} \end{aligned}$$

そして最後に m 組の B_{ij} から1組の B_i を以下のように計算する。

$$\begin{aligned} B_{i,max}^x &= \max_j \{B_{ij,max}^x\} , \quad B_{i,min}^x = \max_j \{B_{ij,min}^x\} \\ B_{i,max}^z &= \max_j \{B_{ij,max}^z\} , \quad B_{i,min}^z = \max_j \{B_{ij,min}^z\} \end{aligned}$$

6.5. スライスグリッドを用いた近傍粒子探索の GPU での実装 (CUDA)87

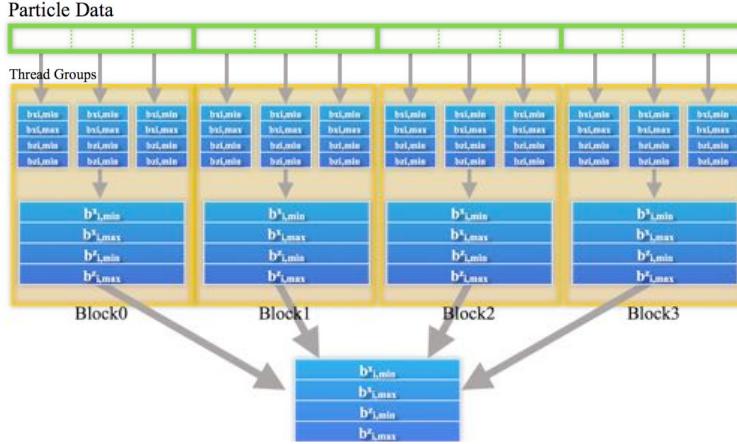


図 6.4: 3段階でのバウンディングボックスの計算

1段階目では ml 個のスレッドが並列で計算を行い、2段階目では m 個のスレッドが並列で計算を行なう。こうすることで計算に並列性を持たせ、複数コアのあるストリームプロセッサで計算するときの計算効率を高める。図 6.4 に 2段階でのバウンディングボックスの計算の様子を示す。

先頭番号の計算

各スライスのバウンディングボックスを求めたので、これらを用いて式 (5.3) を計算する。まず B を用いて各スライスのバウンディングボックス内のボクセル数 n_i を求める。この計算はスライスの数のスレッドを実行し、各スレッドがそれぞれのスライスの B をグローバルメモリから読み出し、それらを用いてボクセル数を計算してグローバルメモリに出力する。次にこれらの値を用いてプレフィックススキャンを計算する。原田らはアルファブレンディングを用いてこの計算を行なったが [135]、本研究では Harris *et al.* の木構造を用いたプレフィックススキャンを用いて、スライスの先頭番号を計算する [60]。プレフィックススキャンとは図 6.5 上に示すような入力とする配列に対して図 6.5 下に示す配列を求める操作のことである。

6.5.2 CUDA を用いたデータ構造の構築

CUDA を用いた場合は、6.5.1 で述べた 1段階目の分割単位にカーネルの 1 ブロックを割り当てる。すなわち m 個のブロックを実行し、それぞれのブロックが n/m 個の粒子を受け持つことになる。2段階目の分割単

Input array	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>0</td><td>3</td><td>1</td><td>2</td><td>0</td><td>0</td><td>1</td><td>2</td><td>1</td></tr></table>	1	0	3	1	2	0	0	1	2	1
1	0	3	1	2	0	0	1	2	1		
Output array	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td>4</td><td>5</td><td>7</td><td>7</td><td>7</td><td>8</td><td>10</td></tr></table>	0	1	1	4	5	7	7	7	8	10
0	1	1	4	5	7	7	7	8	10		

図 6.5: Prefix Sum

位にはブロック内のスレッドを割り当て, l 個のスレッドを実行する. まず 1 段階目の処理ではブロック j で実行されるスレッド k がそれぞれ B_{ijk} の値を計算する. それぞれのスライスでの最大と最小座標 B はスレッド j に割り当てられた粒子数回だけ読み出され, 比較されてメモリに書き込まれる. このように B は頻繁にアクセスされるのでデバイスマモリよりも高速なシェアードメモリに置いて計算する. シェアードメモリのアクセスはグローバルメモリへのアクセスと比較すると高速であるが, 1 ブロックあたり用いることのできる量は NVIDIA の GPU である G80 では 16KB である. ここで計算領域に最大 256 枚のスライスが存在するときの B に必要なメモリの量を計算すると, 256 枚のスライスの XZ 方向の最大最小座標を 32bit で確保すると, $256 \times 32bit \times 4 = 4KB$ である. 1 スレッドの B に 4KB 使用すると, 1 ブロックで数スレッドしか実行できなくなってしまう. GPU は計算の並列度が高いほど効率は良くなるが, 計算の並列度が低いと効率は低下するため, この実装では計算の効率が悪く, パフォーマンスを出すことはできない. そこで複数個のスレッドが 1 組の B を共有することによって, 一度に実行するスレッドの数を増やすことができる. しかし, これによって 1 スレッドが 1 組の B を持っていたときには起こらなかった問題が発生する. それは B を共有する複数のスレッドが読み出した粒子が, 同じスライスに存在する場合に, それらによるメモリアクセスの衝突が起こり, 正しく計算することができなくなってしまうという問題である. 全粒子が異なるスライスに存在する時には問題は起きないが必ずそのようになるとは限らない. そのため, 各スレッドが計算した値とスライスの現在の値との比較を直列化することによって, メモリアクセスの衝突を避けた. CUDA では 1 個のブロック内のスレッドの同期を取ることができるのでこのように部分的に処理を直列化することも可能である.

しかしここで問題は全て解決したわけではない. 1 ブロックが用いることのできる 16KB のシェアードメモリを全て用いることができるとしても, 1 個のブロックでは 4 組の B を計算するスレッドの組しか作ることができない. 1 つのスレッドの組の大きさを大きくすればするほど, 直列化しなければならない処理は長くなってしまい, 非効率的になる. そこで我々は用いる計算領域を 256^3 ボクセルと限定することによって, 一層の効率化

6.5. スライスグリッドを用いた近傍粒子探索の GPU での実装 (CUDA)89

を行なった。このように計算領域の大きさを制限することによってボクセルの最大と最小座標を $8bit$ で表すことができるようになる。それにより 1 スライスのバウンディングボックスの最大と最小座標の 4 個の値は合計で $32bit$ で表すことが可能になり、1 組の B に必要なメモリは 1 個の値に $32bit$ を割り当てていたときの $1/4$ の $1KB$ になる。このように値を $8bit$ に限定することによって 1 個のブロックで最大 16 個の B を確保することができ、直列化しなければならない処理を短くしたままで同時に実行することのできるスレッドの数を増やすことができる。さらにグローバルメモリへのアクセスは遅いため、 B のデータ量を削減した方が全体の計算速度は向上する。

このカーネルの最後に 2 段階目の処理を行い、 l 組の B_{ijk} から 1 組の B_{ij} を計算する。この処理では 1 スレッドを 1 個のスライスに割り当て、それぞれのスレッドが割り当てられたスライスの全ての B_{ijk} から最大と最小座標を計算する。このように B_{ijk} はシェアードメモリに置かれて計算されるが、シェアードメモリのデータはカーネル内部でしか用いることができないため、カーネルの最後で求めた B_{ij} はグローバルメモリに書き出される。

次に行なうのは B_{ij} から全粒子のバウンディングボックスを定義する値 B_i を計算することである。この処理は前のステップの最後に行なった操作とほぼ同じである。ただ前のステップで行なった操作では最大 16 組の B から 1 組の B を計算した。しかしここではより多くの B から 1 個の B を計算しなければならない。例えば、65,536 粒子を用いたシミュレーションで 1 ブロックが 256 粒子の B を計算したとすると、ここでは 256 個の B から 1 個の B を計算しなければならないことになる。そこで 1 スライスあたり複数のスレッドを割り当て、1 個のスレッドが 2 個の値から 1 個の値を求めていき、要素の数を半分にしていく手法を用いた。もちろんシェアードメモリの容量が大きくなり、1 ブロックで多くの B を計算することができるようになれば、そこでも同様に複数のスレッドを用いて求めることもできる。

6.5.3 データへのアクセス

データへのアクセスはシェーダを用いた実装と同様に行なうことができる。どのボクセルも対応するスライスのバウンディングバリューとそのスライスの先頭番号を読み出せば、メモリ上のアドレスを計算することができる。

第7章 應用 (change name)

7.1 関連研究

7.1.1 流体シミュレーション

流体シミュレーションの関連研究については 3.2 に譲る。

7.1.2 剛体シミュレーション

Baraff は剛体間の接触の際に働く力を解析的に求める手法を開発し [5], 剛体の接触に関して研究を行なった [6][7][8][9]. 剛体の衝突に力積ベースの手法を用いた研究も行なわれた [85][45][108]. Mirtich は衝突に関係している剛体のみを衝突時刻に戻す Timewarp アルゴリズムを開発した [85]. バネとダンパを用いるペナルティ法を衝突応答に用いた研究もある [91][111]. Hasegawa らはこのペナルティ法を用いてリアルタイムシミュレーションを行なっているが, 剛体数は非常に少なく, 凸形状のみしか扱えないという問題がある [64][65]. Guendelman らは形状表現にポリゴンと符号付き距離関数を用いて非凸形状の剛体に関して衝突計算を行なった [43]. Kaufman らは多数の剛体間の接触を高速に解く手法を開発した. しかし剛体シミュレーションではまず衝突検出を行なわなければ成らず, 剛体シミュレーション自体を高速化するためには, 高速な衝突検出手法が必要である [70]. Dingliana らは剛体を階層構造を持つ球で表現し衝突計算を行なう手法を開発し 2 次元計算を行ない [28], Suzuki らはこの手法を 3 次元に拡張した [115]. Bell らは剛体を複数の大きさの異なる球で表現して衝突計算を行なった [13]. Havok は GPU で狭域衝突計算を高速化した Havok FX を開発し, 剛体シミュレーションを高速化したが, 広域衝突計算は CPU で行なっているため GPU の性能を完全に引き出せていない可能性がある [16].

7.1.3 布シミュレーション

Provot は粒子とバネで布を表現してシミュレーションを行なった [103]. 硬い布を計算するためには強いバネが必要であり, 数値的に安定に解くためには小さなタイムステップを刻まなくてはならなくなる. そこでそれぞれのバネの伸びを一定の値までに制限することで硬い布を表現した. バネ定数が大きい計算を陽解法で解くためには, 小さなタイムステップが必要であったが, ここで Baraff and Witkin は陰解法を導入することで硬い布の時間積分を効率的にした [11]. 布の曲げ剛性や座屈のより正確なモデルの研究も行なわれてきた [122, 22]. また粒子とバネを用いた布の計算は簡単ではあるが, 布の正確な物性値を用いることができなかつたり, 計算解

像度に応じてバネ定数を調節しなければならないなどの欠点がある。そこで有限要素法を用いて布のモデル化を行った研究もある [33]。

弾性体の衝突計算に関する調査論文に、布の衝突計算の概要もまとめられている [120]。本章で衝突の効率化に用いた軸平行バウンディングボックスは、バウンディングボックス内に布が存在しない領域が多くあるため、より布に合ったバウンディングボリュームとして有向多面体を用いることもできる [84]。布の自己衝突を効率化するための手法に関する研究もある。Volino *et al.* はある連続な布の面があったとき、その布の面を射影して全て重ならないという方向があるならば自己衝突は起こっていないという経験則に基づき布の自己衝突を効率化した [123]。さらにこの手法をより単純化して布の各部分に対して法線ベクトルのが成す角度を求めて自己衝突を効率化する手法が開発された [104]。Bridson *et al.* はProvotの研究 [104] を拡張したが、この手法は1タイムステップ前の布の状態が完全に自己衝突を起こしていない状態である必要がある [17]。しかし実際の問題ではこの仮定が成り立たない状況が出てくることもある。Volino *et al.* はこの仮定を定めない頑強な手法を開発し [124]、Baraff *et al.* はこの手法を拡張し、布の表面を Flood Fill アルゴリズムを用いていくつかの領域に分割し、布の状態を分析することによって、自己衝突が起こっている領域を特定することによって自己衝突を解消するアルゴリズムを開発した [12]。しかしこの手法はまだ多くの問題を抱えている。1つはエッジ同士の衝突である。この問題は Wicke *et al.* や Volino *et al.* らが提案している交差線を最小化する手法で大部分の問題は解決することができるが、少数のポリゴン間で起こっている自己衝突は安定に解くことができない [127, 125]。またまだ研究されていない問題点も存在する。また布の衝突計算の時間の大部分を占めるのはトライアングル間の衝突計算である。問題は最終的には2個のトライアングル間の衝突計算になるが、この計算手法についても研究が行なわれてきた [87]。Ericson はトライアングル間の衝突を含めた様々な衝突判定手法についてまとめている [32]。

布はリアルタイムアプリケーションであるゲームなどでも用いられるが、そのときに重要なのは計算の速度だけでなく、計算の安定性である。布をバネや有限要素法を用いてモデル化し、陽解法で解く場合には計算を安定におこなうことができる時間刻み幅が存在する。よってどのような条件下でも安定に解くことができるような position-based の手法がリアルタイム物理エンジンで用いられている [95]。しかしこのモデル化では布の性質がメッシュの分割に大きく依存するなどの欠点もある。また近年の計算機の能力の向上に伴い、リアルタイムアプリケーションで用いることのできるモデルの解像度も高くなってきた。しかし単に計算能力が向上したからといって高解像度の布を用いることができるわけではない。高解像

度のメッシュを用いることによっていくつかの問題が引き起こされる。1つは計算の収束性の低下である。低解像度のメッシュを用いる場合に用いていた回数の反復計算で、高解像度の布を解こうとしても計算が十分に収束せずに布の伸びが大きくなってしまう。この問題を解決するためにマルチグリッドを用いる研究も行なわれている。陰解法を用いて布の速度を計算する場合には解くべき方程式は一般的な線形連立方程式になるためそれぞれの解像度での係数行列を物理的に組み立てればマルチグリッドをそのまま用いることができる[68]。しかし position-based な手法では解く方程式が非線形方程式になるだけでなく、restriction, prolongation operators を元の方程式から導出することができないという問題点があるため、マルチグリッドを応用するのは容易ではない。マルチグリッド的な計算手法を用いた研究も存在するが、論文では述べられていない問題点が数多く存在する[93]。それは異なるメッシュ間を解が移動する operators を物理的にモデル化できることや、解自体がメッシュの形状に非常に敏感なため、論文に提案されているような手法で粗いメッシュを構築すると非物理的な運動が始まったり、prolongation operator が運動量を保存しないなどの問題点がある。もう1つの問題点は布の自己衝突である。ゲームなどのアプリケーションでは必ずしも前のタイムステップで布が完全に自己衝突のない状況とは仮定することができないことが多い。よってオンラインのアプリケーション向けに研究されているような手法を用いることはできない。さらに1フレーム間の計算時間の中で実行できる十分に頑強な自己衝突解決のアルゴリズムというのはとても困難な問題であり、まだ解かれてはいない。

7.1.4 連成シミュレーション

剛体と流体のカップリングの研究も行なわれてきた。Yngve らは圧縮性流体を用いた爆発の計算を行ない、圧縮性流体と剛体とのカップリングを行なった[128]。Génevaux らは剛体を粒子をバネで繋いだ弾性体で表し、流体のマーカー粒子と剛体粒子間の力を計算することによって流体と剛体のカップリングを行なった[38]。Carlson らは剛体を流体として解き、剛体に変形の拘束条件を課すことで剛体と流体のカップリングを行なった[19]。Takahashi らは流体と飛沫と泡の計算を行ない、さらに剛体と流体のカップリングも行なった[117]。Takahashi らは Volume of Solid という各ボクセルでの剛体の占める割合を導入して剛体部分を特定した。そして剛体境界部分に圧力計算の境界条件を課し、計算された圧力を用いて剛体にかかる力を計算した。Müller らは SPH で流体を解き、Lagrangian メッシュで弾性体の計算を行ない、弾性体メッシュ表面に仮想粒子群を毎ステップ生成して流体粒子とそれらの間で相互作用を計算し、数千粒子で

リアルタイムシミュレーションを行なった [96]. Guendelman らは薄膜と流体の相互作用を弱連成で解いた [44]. Klingner らは境界適合格子を用いて、流体の圧力のポワソン方程式に剛体の圧力境界条件を組み込み、強連成で相互作用の計算を行なった [73]. Chentanez らは Klingner らの手法を拡張し弾性体と流体をカップリングさせた [21]. 自由表面流れ流体シミュレーションと剛体シミュレーションをカップリングする研究は行なわれているが、それらをリアルタイムで計算するという研究はほとんど行なわれていない。

7.1.5 GPU を用いた流体シミュレーション

GPU は処理によっては CPU よりも高速な計算が可能であり、様々な研究が行なわれてきた [100]. 格子を用いた流体シミュレーションの計算の大半は GPU のテクスチャマッピングの計算とほぼ同じ処理であるため、比較的容易に実装可能である。そのためセルオートマトンからはじまり [62]、2次元の流体シミュレーション [59]、それを応用した雲のシミュレーションなどの研究が行なわれてきた [61]. さらに近年の GPU の計算能力を用いて3次元の流体シミュレーションをリアルタイムに実現した研究も存在する [24]. 連立方程式の解法には大半の研究では、要素数が多くなると収束が遅いヤコビ法が用いられてきたが、Red-black ガウスザイデル法を研究した例もある [61]. 他にも共役勾配法やマルチグリッドを GPU で高速化する研究も行なわれてきた [40].

格子法の流体計算とは異なり粒子法シミュレーションは GPU で行なうことには困難であった。これは前章で述べたように粒子が計算領域を自由に動くことができるからである。もちろん N-body シミュレーションのように全粒子との相互作用を計算することも可能であるが [78]、大半の計算方法で用いられるカーネルは局所的にしか値を持たないため、これは非効率である。そこでどのように粒子の近傍粒子探索を行なうかということが問題となる。

Amada らは SPH を GPU で高速化したが、全ての計算を GPU で行なうことにはできず、近傍粒子探索は CPU で行なった [2]. Kolb らの手法では粒子位置における物理量を求める際にまず格子上の値を求め、それらを補間して求めるため補間による数値拡散を引き起こしてしまう [74]. このように粒子法シミュレーションを GPU 上で正確に実現した研究は存在しなかった。

7.2 Distinct Element Method

7.2.1 Distinct Element Method

Distinct Element Method(DEM) では粉体は粒子群として表され、接觸している粒子間で相互作用を計算する。様々な接觸力が用いられているが、本研究では線形バネとダッシュポットを用いて接觸力を計算する (Fig. 7.1).

粒子 i の運動は周囲の粒子からの接觸力と重力を用いて以下のように計算される。

$$\frac{d\mathbf{v}_i}{dt} = \frac{1}{m} \sum_{j \in contact} \mathbf{f}_{ij}^c + \mathbf{g} \quad (7.1)$$

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i \quad (7.2)$$

$$\mathbf{f}_{ij}^c = \mathbf{f}_{ij}^n + \mathbf{f}_{ij}^t \quad (7.3)$$

ここで $\mathbf{x}_i, \mathbf{v}_i, m$ はそれぞれ粒子 i の中心座標と並進速度、質量である。また \mathbf{f}_{ij}^c は粒子 j から粒子 i に働く接觸力であり、 \mathbf{f}_{ij}^c は法線方向の接觸力 \mathbf{f}_{ij}^n と接線方向の接觸力 \mathbf{f}_{ij}^t から構成される。法線方向は 2 粒子の中心を結ぶ方向であり接線方向はこの法線に垂直な平面の方向である。また \mathbf{g} は重力ベクトルである。粒子 i に接觸している全粒子において接觸力を求めて速度と座標の更新を行なう。

粒子 i, j 間に働く法線方向の接觸力は、

$$\mathbf{f}_{ij}^n = -k\Delta\mathbf{r}_{ij}^n - \eta\mathbf{v}_{ij}^n \quad (7.4)$$

と計算される。ここで k, η はそれぞれ線形バネのバネ定数とダッシュポットの減衰定数であり、 \mathbf{v}_{ij}^n は相対速度ベクトルの法線方向成分であり、 $\Delta\mathbf{r}_{ij}^n$ は 2 粒子が接觸していない状態における粒子間の距離からの法線方向の変位である。

粒子 i, j 間に働く接線方向の接觸力は摩擦力であり、粒子同士の接線方向のずれに応じた力としてバネとダッシュポットを用いて計算する。2 粒子が衝突した時刻の粒子間の相対座標を \mathbf{r}_{ij}^0 として保持しておく。この相対座標 \mathbf{r}_{ij}^0 の大きさをバネの自然長として用いる。そしてそれ以降の時刻における相対変位 $\Delta\mathbf{r}_{ij}$ を相対座標方向の単位ベクトル \mathbf{n} を用いて法線方向の成分 $\Delta\mathbf{r}_{ij}^n$ と接線方向の成分 $\Delta\mathbf{r}_{ij}^t$ に以下のように分解することができる。

$$\Delta\mathbf{r}_{ij}^n = (\Delta\mathbf{r}_{ij} \cdot \mathbf{n})\mathbf{n} \quad (7.5)$$

$$\Delta\mathbf{r}_{ij}^t = \Delta\mathbf{r}_{ij} - \Delta\mathbf{r}_{ij}^n \quad (7.6)$$

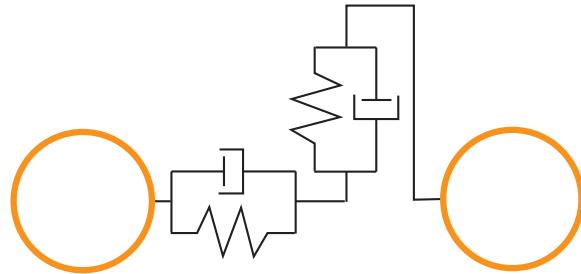


図 7.1: 粒子間に設定されるバネとダンパ。

従って粒子 i, j 間に働く接線方向の接触力は,

$$\mathbf{f}_{ij}^t = -k\Delta\mathbf{r}_{ij}^t - \eta\mathbf{v}_{ij}^t \quad (7.7)$$

と計算される。

ただしクーロンの法則により摩擦力の大きさは最大静止摩擦力を上回ることはない。よって接線方向の接触力が最大摩擦力を上回る場合には代わりに動摩擦力が働くため、背線方向の接触力は

$$\mathbf{f}_{ij}^t = \mu|\mathbf{f}_{ij}^n|\mathbf{n}. \quad (7.8)$$

ここで μ は動摩擦係数である。また動摩擦力が働く場合には相対変位の基準ベクトル \mathbf{r}_{ij}^0 を更新する。

壁境界

壁境界の形状が複雑な場合は、メッシュデータで形状を定義することができる。そうするとある粒子が壁に衝突しているかどうかは、メッシュを構成する全ての三角形と距離計算を行ない、最小距離を求めなければならない。この計算は計算コストが高いため、距離関数を導入した。距離関数とはある境界が与えられたときに、計算領域内での境界までの最小距離であり、これを用いることで粒子と壁境界との最小距離計算はその粒子の座標における距離関数を求めることで行なうことができる。距離関数で粒子と壁の距離が衝突距離以下であれば、その間にバネとダッシュポットを発生させ、粒子同士での衝突時と同様に処理を行なう。このときに粒子とその粒子と最小距離にある壁までの単位ベクトルを求めなければならない。ある点における距離関数の勾配はその点と最小距離にある壁までの単位ベクトルとなるので、距離関数の勾配を求めてこのベクトルを計算した。距離関数の計算には Baerentzen らによる手法を用いた [3]。

7.2.2 実装

GPU上で座標などの粒子データは全て2次元テクスチャに格納する。個別要素法をGPU上で実装するために必要なテクスチャは座標テクスチャ2枚、速度テクスチャ2枚、バケットテクスチャ1枚である。

まず近傍粒子探索を行なうために、粒子番号をバケットテクスチャに格納する必要があるが、この処理は前節で述べた通りに行なう。次に衝突検出を行なう。この処理は粒子 i と衝突している可能性のある近傍粒子の探索に粒子座標テクスチャと生成したバケットテクスチャを入力として、フラグメントシェーダで処理を行なう。バケットを用いることである粒子 i が格納されているバケットを囲む 3^3 個のバケット内に格納されている粒子番号を得ることができる。これらの粒子と粒子 i との距離計算を行ない衝突検出を行なう。

具体的には、まず粒子 i の粒子座標を粒子座標テクスチャから取得する。そしてこの座標からバケットのインデックスを計算する。そしてこのバケットを囲む 3^3 個のバケットテクスチャのRGBAチャネルに保持されている粒子番号を取り出し、それぞれの粒子番号からその粒子に対応する粒子座標テクスチャのテクスチャ座標を計算する。そのテクスチャ座標を用いてこれらの粒子の座標を取り出し、粒子 i の座標と距離を計算し衝突検出を行なう。

そして衝突が検出された粒子間で式(7.1)から粒子間に働く力を計算する。速度の更新は衝突検出を行なうシェーダ内で行なう。このシェーダでは粒子座標テクスチャ、粒子速度テクスチャ、バケットテクスチャを入力とし、更新された粒子速度テクスチャを書き出す。

最後に式(7.2)で座標の更新を行なうが、この処理もフラグメントシェーダで行なう。粒子速度と前タイムステップでの粒子座標が必要であるため、これらのテクスチャを入力としてフラグメントシェーダで座標の更新を行なう。

7.2.3 結果

本手法をCore 2 CPU X6800 2.93GHz、2.0GBのメモリを搭載したPCにて実装した。GPUはPCIExpressで接続されたGeForce7900GTXを用いた。このGPUのビデオメモリは512MBである。プログラムはC++,OpenGL,C for Graphics[36]を用いて開発した。Fig.7.2とFig.7.3に本手法を用いて解析を行なった結果を示す。ともに用いた総粒子数は262,144である。

本手法で粒子の座標は座標テクスチャ内に保持されている。これらを用いてレンダリングを行なうときにビデオメモリ内にあるテクスチャデータ

をメインメモリに読み戻してレンダリングすることも可能であるが、ビデオメモリとメインメモリ間のデータ転送は遅いため、ポイントスプライトとバーテックステクスチャフェッチを用いてレンダリングした[107]。VTFを用いると1つの頂点を1粒子に割り当てこの頂点を粒子座標に変位させることができる。バーテックスシェーダで直接ビデオメモリ内のデータを参照することができるため、CPU-GPU間のデータ転送を必要としないため高速にレンダリングすることができる。これでは計算粒子数の頂点が点で表示されるだけであるが、ポイントスプライトを用いることで各頂点座標にライティングされた球を描画することができる。

まず本手法の計算速度の測定を行なった。Table.7.1に本手法の各段階の計算時間を測定した結果を示す。総粒子数が約26万でも1タイムステップ133.59ミリ秒で計算することができている。全体の計算時間に占める衝突検出の割合が高いことがわかる(Table.7.1 Time(b))。これは衝突検出のシェーダプログラムが長く、多くの条件分岐を含むので、1ピクセルの計算コストが高いことに起因していると考えられる。また一般的にバーテックスシェーダからのテクスチャアクセスはフラグメントシェーダからのアクセスよりも速度が遅い。本手法ではバケット構築にバーテックスシェーダからテクスチャにアクセスしている。しかしTable.7.1 Time(a)の計算時間を見ると、数万以下の粒子数を用いている場合には計算のボトルネックにはなっていない。より一層の高速化を行なうためには衝突検出の計算速度を向上させなければならない。現在は衝突検出に1個のシェーダプログラムを用いているが、複数のプログラムに分割して1個のプログラムあたりの条件分岐とループを減らすことで高速化につながる可能性がある。

次にGPU上で実装した本手法の計算時間とCPUでの計算時間を比較した。CPUでの実装でも同様にバケットを用いて粒子間の衝突検出を行なった。CPUプログラムは最大限の最適化を行なうオプション(O3)を付けコンパイルした。SSEやデュアルコアに対応するようにはしなかった。5種類の粒子数で1タイムステップの計算時間を測定した結果をTable.7.2に示す。どの粒子数においてもCPUでの計算時間よりもGPUでの計算時間の方が短いことがわかる。GPUによる計算時間は最大でCPUの約16倍の計算速度が出ている。粒子数16,386でのCPUでの計算時間と粒子数262,144でのGPUでの計算時間はほぼ同じであることから、同じ計算時間で約16倍の粒子数で計算が可能になっている。Fig.7.4に計算時間の比較をグラフにしたものを見せる。CPUでの計算時間とGPUでの計算時間は、ともに粒子数に比例しているが、粒子数増加に伴う計算時間の増加はGPUを用いたときの方が大幅に小さい。よって粒子数が増加するほどGPUの方が計算の効率が良くなっている。

次に GPU は 64bit 浮動小数点を取り扱うことができず 32bit 浮動小数点でしか演算することができない。そこで CPU で 64bit 浮動小数点と 32bit 浮動小数点で計算した結果と比較して計算精度の検証を行なった。検証計算は Fig.7.5 に示す粒子直径 1.0m の 2 粒子を用いて行なった。2 粒子は床からそれぞれ 2.5, 4.5m の高さに配置され、自由落下する。 $k = 2.5 \times 10^1$, $\eta = 2.5 \times 10^{-2}$ をバネ定数とダッシュポットの減衰定数に用いた。これらの粒子の座標を出力し比較を行なった。Fig.7.6 に CPU, GPU ともに 32bit 浮動小数点を用いた結果、Fig.7.7 に CPU は 64bit, GPU は 32bit 浮動小数点を用いた結果を示す。それぞれ縦軸は 2 粒子の床からの高さであり、横軸は計算時間である。GPU での 32bit 浮動小数点を用いた結果と CPU での 32bit 浮動小数点を用いた結果では少しの差が存在した。GPU での 32bit 浮動小数点は IEEE754 と同じ仮数部 23 ビット指数部 8 ビットの形式で保持されるが、演算は全く同じ結果を返すとは限らない。これが原因となって現れた差であると考えられる。CPU で 64bit 浮動小数点を用いた結果との差も大きくはないということが確認できた。

バケットのデータは 2 次元テクスチャに格納したが、GPU で扱うことができる大きさには限界があるため、非常に大きな計算領域での計算などは行なうことはできない。本研究で用いた GPU が扱うことができる最大のテクスチャサイズは $4,096 \times 4,096$ であるため、バケット中の格子数がこの数を超える計算は行なうことができない。

本手法では各バケットに最大 4 個の粒子までしか格納することができない。粒子径の立方体の中に粒子を最密構造で充填させたとき、その立方体内部に存在する粒子は 4 個である。そのため理論的には最密構造で粒子を充填させても 4 個の粒子まで格納できれば十分である。しかし実際計算誤差によって粒子が最密構造よりも密に配置されることもあると考えられる。しかし本研究で用いた計算例は同様の計算を CPU でも行なっており、各バケットに格納される最大粒子数が 4 を超えないことを確認している。より密に充填される場合はバケットテクスチャをもう 1 枚作成し、8 パスのレンダリングを行なうことによって各バケットに蓄えることができる粒子数を 8 に増やすことが可能である。

本研究で用いた個別要素法の接触モデルでは粒子の回転と摩擦を考慮していない。粒子の回転を計算しなければならない計算では Multiple Render Target(MRT) を用いることで対応することができると考えられる [107]。MRT とは一度のレンダリングで複数枚数のバッファを書き出すことが可能な機能であり、MRT を用いることで速度と角速度の 2 つのバッファを同時に更新することができるようになる。

表 7.1: 計算時間の詳細(ミリ秒). Time(a), (b), (c) はそれぞれ格子の生成, 衝突検出, そしてその他の計算にかかる時間である.

Number of particles	Time(a)	Time(b)	Time(c)	Total
1,024	0.94	0.38	0.40	1.72
4,096	1.09	1.97	0.22	3.28
16,386	1.87	6.41	1.25	9.53
65,536	4.22	23.91	4.84	32.97
262,144	14.37	85.32	33.90	133.59

表 7.2: GPU と CPU による計算時間の比較. Time(a) が GPU, Time(b) が CPU での計算時間である.

Number of particles	Time(a)	Time(b)
1,024	1.72	6.75
4,096	3.28	29.0
16,386	9.53	116.35
65,536	32.97	539.6
262,144	133.59	1804.0

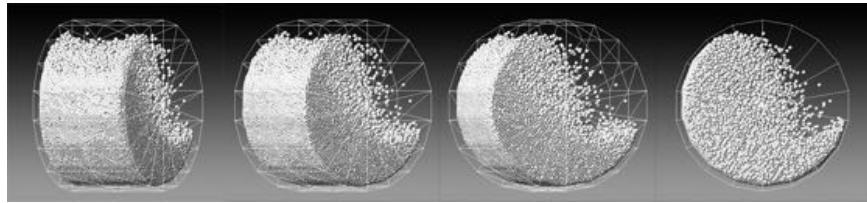


図 7.2: 262,144 粒子を用いた回転する容器内の粉体のリアルタイムシミュレーション.

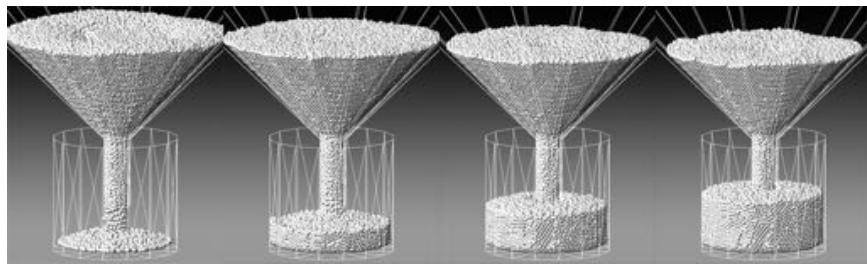


図 7.3: 262,144 粒子を用いた粉体の排出のリアルタイムシミュレーション.

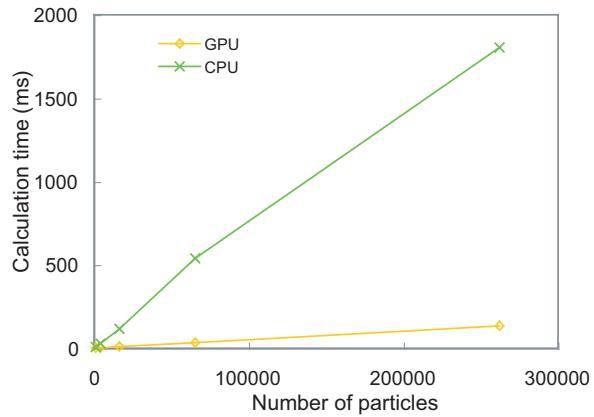


図 7.4: CPU と GPU による計算時間の比較.

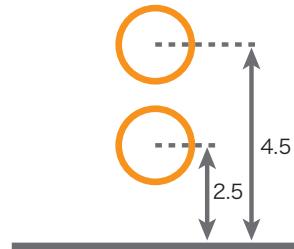


図 7.5: 検証計算の初期条件.

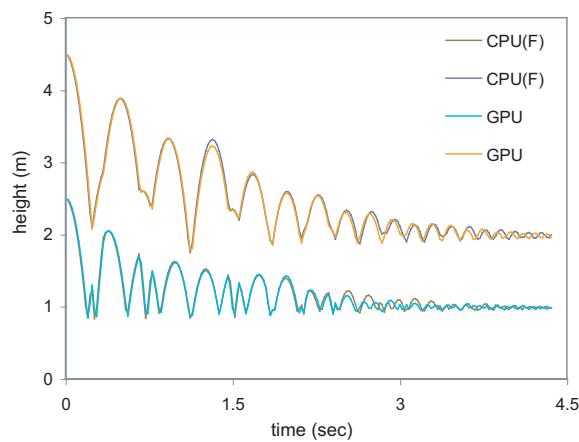


図 7.6: 単精度浮動小数での GPU と CPU での計算精度の比較.

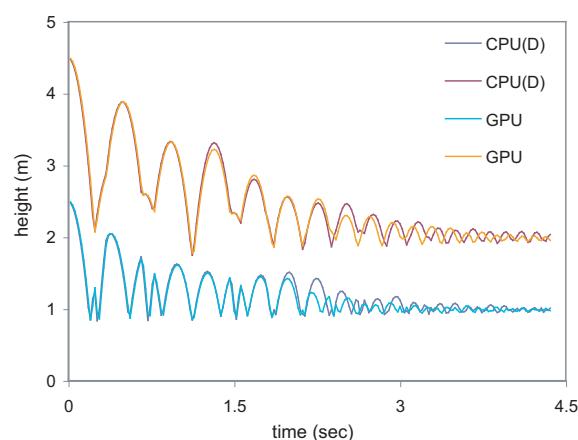


図 7.7: GPU での単精度浮動小数点を用いた場合と、CPU で倍精度浮動小数点を用いた場合の計算精度の比較。

7.3 Smoothed Particle Hydrodynamics

7.3.1 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics(SPH)による流体の運動方程式の定式化は3に示したものを用いた。また壁境界も3に示した手法を用いた。DEMとSPHの計算上の違いはDEMでは近傍粒子探索が1回で済むが、SPHでは密度の計算と加速度の計算において2回必要であるという点である。またDEMでは接触している粒子のみを探索するため、ある粒子の存在しているボクセルを含め周囲の 3^3 個のボクセルを探索するだけよいが、SPHは計算パラメータによってそれ以上の範囲の近傍粒子探索が必要となる。本研究では影響半径に粒子径の2倍の距離を用いているため、近傍粒子探索は 5^3 個のボクセルから探索する。このようにSPHはDEMと比べて計算コストが高い。

7.3.2 実装

基本的にSPHのGPUでの実装は前節のDEMと同じであるが、SPHではDEMと異なり、圧力の計算と、圧力を用いて粒子にかかる圧力勾配を求めるときに近傍粒子のデータが必要になる。本研究ではこれらの処理において2回近傍粒子探索を行なったが、これを近傍粒子探索を別のパスで行い、その結果をメモリに書き出し、このデータを読み込んで近傍粒子の値を参照することもできる。SPHでは近傍粒子を2回しか探索しないため、このように近傍粒子をメモリに書き出しても書き出さなくても結果に変化ないと判断し、2度探索する方法を選択した。しかしMPSのように反復して近傍粒子の値を読み出す必要がある場合は、近傍粒子番号を書き出した方が効率的だと考えられる。

7.3.3 結果

本手法をCore 2 X6800 2.93GHzのCPUと2.0GBのメモリ、GeForce 8800GTXを搭載したPCにて実装した。なおビデオメモリは768MBである。プログラムはC++,OpenGL,C for Graphics[36]を用いて開発した。

図7.8,7.9は約100万粒子を用いた計算である。GPU上で計算を行なった計算粒子座標をデータとして出力しCPUでレンダリングを行なった。まず計算に用いた粒子の1つ1つに濃度分布を与え、それらの足し合わせを行ない陰関数曲面を計算した。そしてMarching Cube法を用いて等値面を抽出し、その面をレイトレスでレンダリングした[80]。水槽に液滴が落下するシミュレーションを行なった結果を図7.8に示す。図7.9は

水槽に2カ所の流入部から流体を流入させた計算結果である。これらの約100万粒子を用いた計算では1タイムステップの計算を約1秒で行なうことができた。約400万粒子を用いて水柱崩壊の計算を行なった結果を図7.10に示す。この図のレンダリングにはポイントスプライト用いている。粒子の色は粒子数密度から計算し、粒子数密度が高い粒子は青く、粒子数密度が低い粒子は白くレンダリングされている。

粒子数を変えて本手法の計算時間を測定した結果を表7.3に示す。ここではポイントスプライトによる簡易的なレンダリングを行ない、計算とレンダリングを各1回ずつ行なった計算時間を測定した。なおこのレンダリングではバーテックスシェーダから、粒子座標テクスチャを参照しレンダリングしているため、CPU-GPU間のデータ転送は行なっていない。表から数万粒子を用いても数十ミリ秒、約100万粒子を用いても約1秒で計算できていることがわかる。また全体の計算時間に占めるバケット構築の割合が低く、計算時間の大部分が密度と粒子にかかる力を計算する部分であることがわかる。密度と力の計算では影響半径内に存在する近傍粒子を探索する必要がある。この計算では多くのバケットの中の粒子のインデックスを参照し、そのインデックスを用いて粒子の座標を参照しなければならないため、計算コストが高い。さらにこれらの処理の中では一度テクスチャから読み出した値を使って再度テクスチャアクセスを行なうため計算コストが他の処理に比べて大きくなってしまっていると考えられる。

またCPUプログラムとしてSPHを実装し計算を行ない、計算時間を測定した結果を表7.4に示す。結果から本手法はCPUでの計算速度の数倍から数十倍の速度で計算を行なうことができていることがわかる。粒子数が少ない場合は本手法はCPUの数倍の計算しかできていないが、粒子数が増加すればするほど本手法での計算の方が効率的に計算でき、数十倍の計算速度が出せることがわかる。

本研究では計算領域を直方体で囲み、その内部全体にバケットを作成した。しかし計算例では計算領域の上部はほとんど流体粒子が存在せず、バケットには情報は書き込まれない。この領域のメモリを確保する必要はない。今の実装だと広い領域を計算するためには、多量のメモリを必要とする。GPUでもメモリの上限があるため、計算領域の大きさに制限がでてしまう。メモリ効率を向上させさらに広い領域の計算を行なうために、流体粒子が存在する空間の周辺のみにバケットを作成する手法を開発する必要がある。

GPUは倍精度の計算は行なうことはできず、全ての変数を单精度で計算している。そのため、精度が必要とされる計算には用いることができない。CPUとの計算時間の比較からわかるように、本手法は計算速度を劇的に向上させた。そのため、本手法を工学的な計算にも用いることで計算

表 7.3: Time (a) は格子の生成にかかる計算時間であり, Time (b) は 1 タイムステップの計算とレンダリングにかかる計算時間(ミリ秒).

Number of particles	Time (a)	Time (b)
1,024	0.75	3.9
4,096	0.80	5.45
16,386	1.55	14.8
65,536	3.99	58.6
262,144	14.8	235.9
1,048,576	55.4	1096.8
4,194,304	192.9	3783.6

表 7.4: CPU での計算時間(ミリ秒)と CPU と GPU での計算時間の比.

Number of particles	CPU	CPU:GPU
1,024	15.6	4.0:1
4,096	43.6	8.0:1
16,386	206.2	13.9:1
65,536	1018.6	17.3:1
262,144	6725.6	28.5:1

時間の短縮を行なうことができるが, そのためには精度の検証を行なう必要がある. また GPU 上での計算ではメモリの制約もある. GPU はメインメモリを直接参照することはできず, 使用することができるメモリはビデオメモリのみである. ビデオメモリは増設することができないので, ビデオメモリ上に乗りきらない大きさの計算は行なうことができない. 本研究で計算した 400 万粒子の計算体系では, 256^3 の領域のバケットを確保した. この計算では約 600MB のメモリを使用している. 約 900 万粒子を用いた場合の使用メモリは 1.0GB を超える.

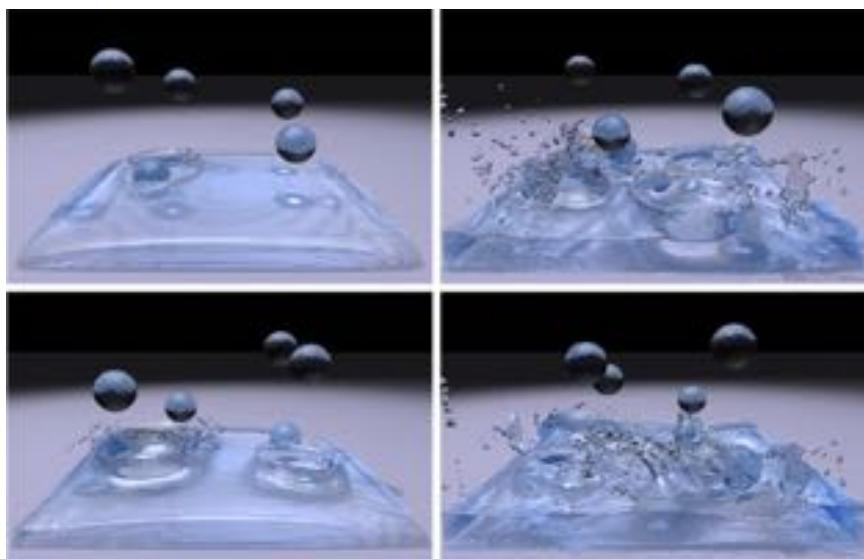


図 7.8: 水槽に液滴を落としたシミュレーション結果。

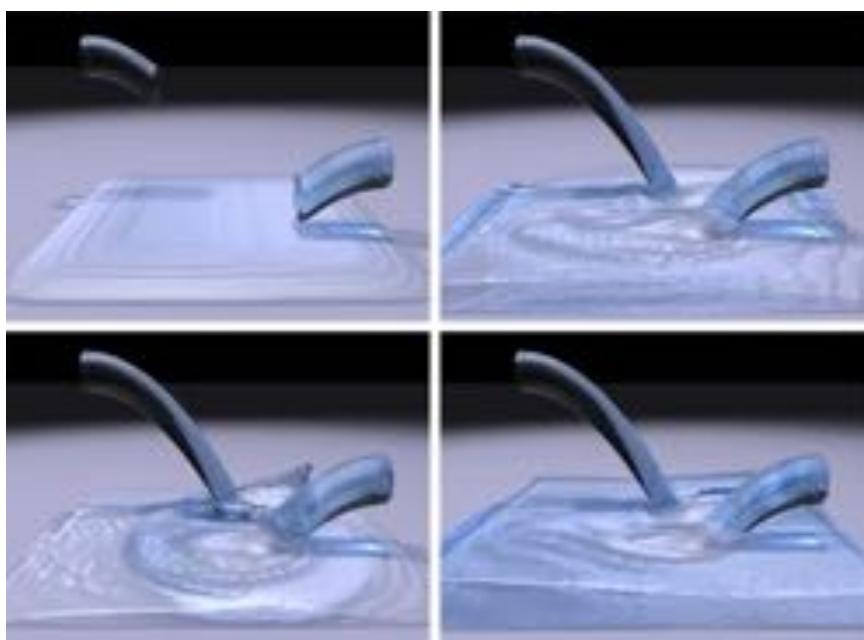


図 7.9: 流体を水槽に流入させたシミュレーション結果。

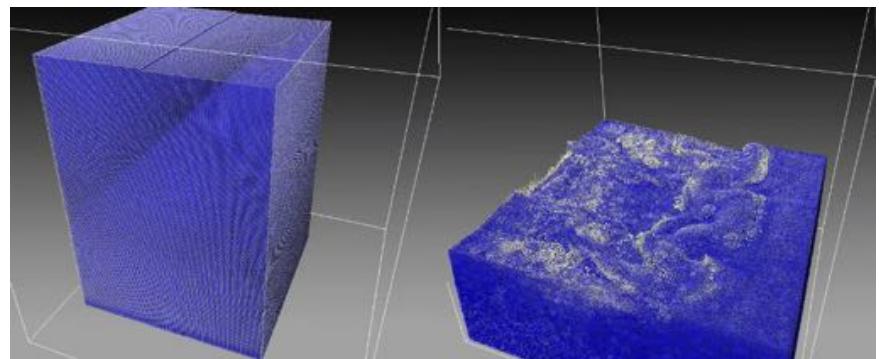


図 7.10: 高解像度でのダム崩壊のシミュレーション結果.

7.4 粒子を用いた剛体シミュレーション

7.4.1 剛体計算

物理量

剛体の計算に必要な物理量は、重心の座標 \mathbf{x} 、並進速度 \mathbf{v} 、回転量を表すクオータニオン \mathbf{q} 、角速度 \mathbf{w} 、慣性テンソル \mathbf{I} である。剛体の運動は並進運動と回転運動に分けて計算する。

並進運動

剛体の並進運動ではまず剛体に働く力 \mathbf{F} を計算する。 \mathbf{F} は以下のように衝突によって生じる力 \mathbf{F}_c とそれ以外の外力 \mathbf{F}_e に分解することができる。

$$\mathbf{F} = \mathbf{F}_c + \mathbf{F}_e \quad (7.9)$$

剛体の並進運動量 \mathbf{P} の時間微分は剛体に働く力 \mathbf{F} を用いて以下のように計算される。

$$\frac{d\mathbf{P}}{dt} = \mathbf{F} \quad (7.10)$$

計算された並進運動量 \mathbf{P} を用いて $\mathbf{P} = M\mathbf{v}$ より重心の速度 \mathbf{v} を計算する。そして重心の速度 \mathbf{v} を用いると剛体の重心位置 \mathbf{x} の時間微分は

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \quad (7.11)$$

と表される。

回転運動

剛体に働く力 \mathbf{F} は剛体の回転運動を生み、角運動量 \mathbf{L} を変化させる。角運動量 \mathbf{L} の時間微分は

$$\frac{d\mathbf{L}}{dt} = \mathbf{r} \times \mathbf{F} \quad (7.12)$$

と表される。ここで \mathbf{r} は剛体の重心から力の作用点までのベクトルである。角運動量を用いて角速度 \mathbf{w} は以下のように計算される。

$$\mathbf{w} = \mathbf{I}(t)^{-1}\mathbf{L} \quad (7.13)$$

ここで $\mathbf{I}(t)^{-1}$ は時刻 t での慣性テンソルの逆行列である。この時刻 t での慣性テンソルの逆行列 $\mathbf{I}(t)^{-1}$ は初期状態での剛体の慣性テンソル $\mathbf{I}(0)$ の逆行列 $\mathbf{I}(0)^{-1}$ を用いて

$$\mathbf{I}(t)^{-1} = \mathbf{R}(t)\mathbf{I}(0)^{-1}\mathbf{R}(t)^T \quad (7.14)$$

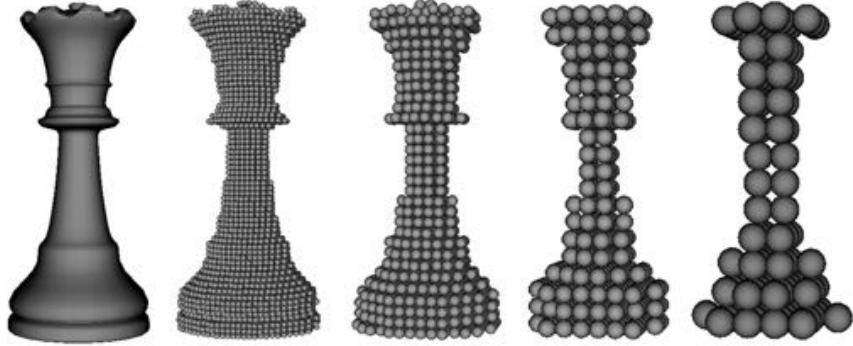


図 7.11: 粒子を用いた形状表現.

と計算される。ここで $\mathbf{R}(t)$ と $\mathbf{R}(t)^T$ は時刻 t での回転行列とその転置行列である。

次に式(7.13)で計算された角速度 \mathbf{w} を用いてクオータニオン \mathbf{q} を更新する。角速度 \mathbf{w} からクオータニオンの変化量 $d\mathbf{q}$ は以下のように計算される。

$$d\mathbf{q} = [\cos\left(\frac{\theta}{2}\right), \mathbf{a}\sin\left(\frac{\theta}{2}\right)] \quad (7.15)$$

ここで回転軸 \mathbf{a} と回転角 θ は

$$\mathbf{a} = \frac{\mathbf{w}}{|\mathbf{w}|} \quad (7.16)$$

$$\theta = |\mathbf{w}dt| \quad (7.17)$$

と計算される。

式(7.15)で計算されたクオータニオンの変化量と時刻 t でのクオータニオン $\mathbf{q}(t)$ を用いて dt 後のクオータニオン $\mathbf{q}(t+dt)$ は

$$\mathbf{q}(t+dt) = d\mathbf{q} \times \mathbf{q}(t) \quad (7.18)$$

と計算される。

7.4.2 衝突計算

概要

本手法では剛体を球の集合と近似し、それらを用いることで衝突計算を行なう。剛体のポリゴンモデルからの球の近似形状は原田、越塚の GPU を用いたボクセル化手法を用いて生成した[48]。用いる球の空間解像度を変化させることで計算精度と計算速度をコントロールすることができる(図 7.11)。

衝突検出

剛体の衝突を検出するために、剛体を構成する球の間の距離を計算する [118]。球の間の距離が球の直径よりも小さければ衝突しているとみなす。衝突検出は全ての剛体を構成する球と距離計算を行なわなければならぬ。この計算コストは球の数の2乗に比例し、剛体の数が多くなると球の数も多くなり、計算コストは膨大なものとなる。そこで計算量を減らすためにまず空間をバケットと呼ばれる格子に分割して、全ての球をバケットに格納する。バケットの一辺の長さを球の直径と同じ長さにすると、このデータ構造を用いることで球 i と衝突している可能性のある球は球 i が格納されているバケットと隣接している 3^3 個のバケット内に格納されている球に限定される。このようにバケットを構築して衝突検出を行なうことで計算コストを大幅に下げることができる。

衝突応答

衝突応答では剛体を構成する球にかかる力をまず計算する。衝突力の計算では個別要素法で用いられる線形バネとダッシュポットを用いた [26]。バネ定数と減衰定数を k, η として、衝突している球にめり込み量に比例した反発力と相対速度に比例した減衰力を働くとする。2つの球 i と球 j があり、これらの球の直径を d とすると、2つの球の距離 $|\mathbf{r}_{ij}|$ が d より小さいときに衝突しており、以下のバネによる力 \mathbf{f}_s とダンパによる減衰力 \mathbf{f}_d を働くとする。

$$\mathbf{f}_s = -k(d - |\mathbf{r}_{ij}|) \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|} \quad (7.19)$$

$$\mathbf{f}_d = \eta(\mathbf{v}_j - \mathbf{v}_i) \quad (7.20)$$

ここで $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ であり $\mathbf{r}_i, \mathbf{r}_j$ はそれぞれ球 i, j の位置ベクトルである。また剪断方向の摩擦力 \mathbf{f}_t は剪断方向の速度 \mathbf{v}_t に比例した力として以下のように計算される。

$$\mathbf{f}_t = k_t \mathbf{v}_t \quad (7.21)$$

これらの力から球 i に働く力 \mathbf{f}_i は

$$\mathbf{f}_i = \mathbf{f}_s + \mathbf{f}_d + \mathbf{f}_t \quad (7.22)$$

と計算される。

計算された球 i に働く力 \mathbf{f}_i を用いて衝突によって剛体に働く力 \mathbf{F}_c とトルク \mathbf{T}_c は以下のように計算される。

$$\mathbf{F}_c = \sum_{i \in RigidBody} \mathbf{f}_i \quad (7.23)$$

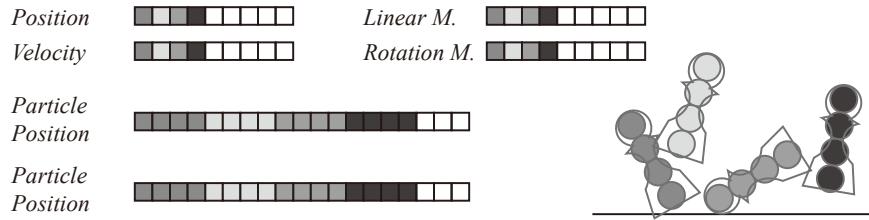


図 7.12: 粒子を用いた剛体シミュレーションにおいて用いられるデータ。

$$\mathbf{T}_c = \sum_{i \in \text{RigidBody}} \mathbf{r}'_i \times \mathbf{f}_i \quad (7.24)$$

と計算される。ここで \mathbf{r}'_i は剛体の重心に対する球の相対位置ベクトルである。

7.4.3 実装

GPUで計算するときに物理量はビデオメモリにテクスチャとして保持される。剛体計算を行なうために剛体重心座標、剛体クオータニオン、並進運動量、角運動量テクスチャをそれぞれ2枚ずつ必要とする。衝突の計算を行なうためには球の中心座標、球の速度、バケット、球に働く力、球に働くトルクにテクスチャをそれぞれ1枚ずつ用意する。

アルゴリズム

1 タイムステップでの計算は以下の 8 段階で計算される。

1. 球の物理量の計算
2. バケット構築
3. 衝突検出
4. 衝突応答(球に働く力とトルクの計算)
5. 剛体並進運動量の計算
6. 剛体角運動量の計算
7. 剛体重心座標の計算
8. 剛体クオータニオンの計算

そして計算された剛体の重心座標とクオータニオンを用いてレンダリングを行なう。

球の物理量の計算

まず衝突計算のために各剛体を構成する球の中心座標と速度を計算する。剛体 j の重心座標とクオータニオン、速度、角速度をそれぞれ $\mathbf{X}_j, \mathbf{Q}_j, \mathbf{V}_j, \mathbf{W}_j$ とする。回転していない状態での剛体の重心に対する球 i の中心の相対位置ベクトルを \mathbf{r}_i とするとクオータニオン \mathbf{Q}_j で回転している状態での相対位置ベクトル \mathbf{r}'_i は

$$\mathbf{r}'_i = \mathbf{Q}_j \mathbf{r}_i \mathbf{Q}_j^T \quad (7.25)$$

と計算される。

剛体 j を構成する球 i の中心座標と速度をそれぞれ $\mathbf{x}_i, \mathbf{v}_i$ とすると、これらは

$$\mathbf{x}_i = \mathbf{X}_j + \mathbf{r}'_i \quad (7.26)$$

$$\mathbf{v}_i = \mathbf{V}_j + \mathbf{W}_j \times \mathbf{r}'_i \quad (7.27)$$

と計算することができる。

レンダリング

このように計算された剛体の重心座標とクオータニオンを用いてレンダリングを行なう。頂点 i の回転していない時の剛体の中心からの相対座標ベクトル \mathbf{r}_i をまずクオータニオン \mathbf{Q} を用いて現在の剛体の回転量だけ回転させ、剛体の重心座標 \mathbf{X} 移動させると頂点 i の現在の座標 \mathbf{r}'_i を計算することができる。

$$\mathbf{r}'_i = \mathbf{X} + \mathbf{Q} \mathbf{r}_i \mathbf{Q}^T \quad (7.28)$$

そしてモデルビュープロジェクション行列 \mathbf{M} を用いて、以下のように現在レンダリングされている座標系での座標 \mathbf{r}''_i に変換してレンダリングする。

$$\mathbf{r}''_i = \mathbf{M} \mathbf{r}'_i \quad (7.29)$$

7.4.4 結果

本手法を Core 2 CPU X6800 2.93GHz, 2.0GB のメモリ、GeForce7900GTX を搭載した PC 上で実装した。プログラムは C++, OpenGL, C for Graphics を用いて開発した。

図 7.13 にチェスの駒 16,385 個落とした結果を示す。この計算時間は表 7.5 に示すように 1 タイムステップにかかる計算時間は 12.8 ミリ秒であった。剛体数を変え計算時間とフレームレートを測定した。1 タイムステップ計算した後にシーンをレンダリングしたフレームレートも、表 7.5 に

表 7.5: チェスの駒を用いた計算時間とフレームレート (ミリ秒).

Number of chess pieces	Simulation time	FPS
1,024	4.23	168
4,096	8.62	53
16,384	12.8	21.2

表 7.6: トーラスを用いた計算時間とフレームレート (ミリ秒).

Number of toruses	Simulation time	FPS
682	4.08	200
2,730	8.00	79
10,922	16.6	23

示すように、チェスの駒 4,096 個落とした場合でも 53FPS であった。また図 7.14 に 10,923 個のトーラスを落下させた計算結果を示す。トーラスも落下させる個数を変化させて計算時間を測定した。表 7.6 に示すように 10,922 個のトーラスを用いた場合でも 1 タイムステップにかかる計算時間は 16.6 ミリ秒であった。そしてレンダリングも行なった場合ではフレームレートは 23FPS であった。GJK などの衝突判定アルゴリズムでは物体の形状が凸でなければならないが、本研究では粒子を用いた衝突計算を行なっているため、物体の形状に制限はない。よって図 7.15 に示すような凸形状でない物体のシミュレーションも容易に行なうことが可能である。

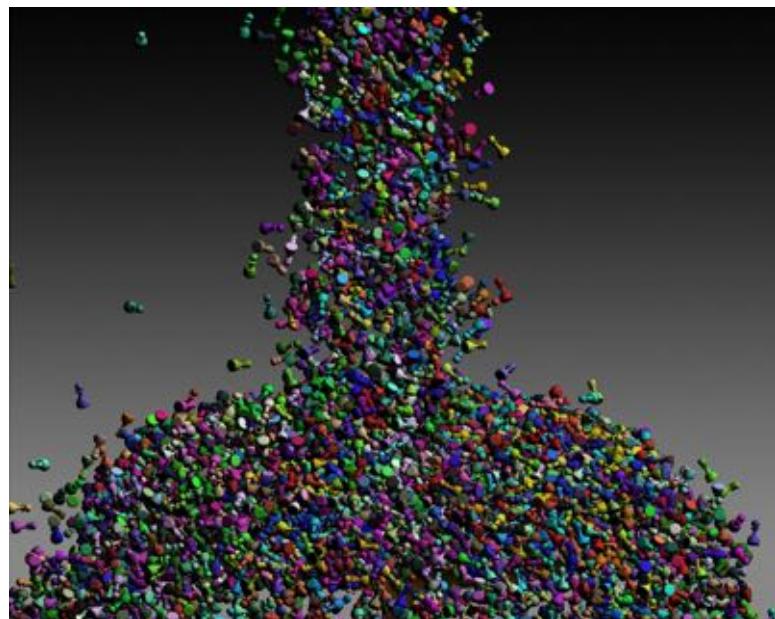


図 7.13: 16,383 個のチェスのコマのシミュレーション

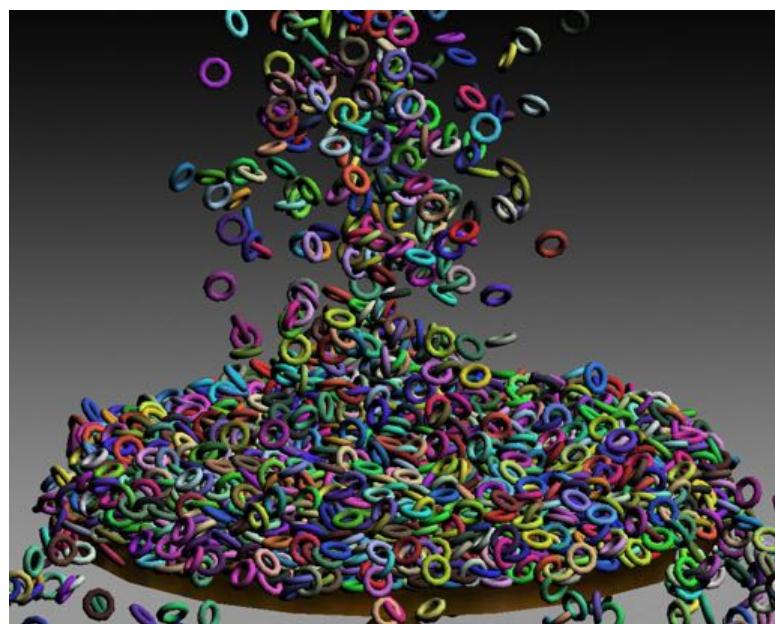


図 7.14: 10,922 個のトーラスのシミュレーション

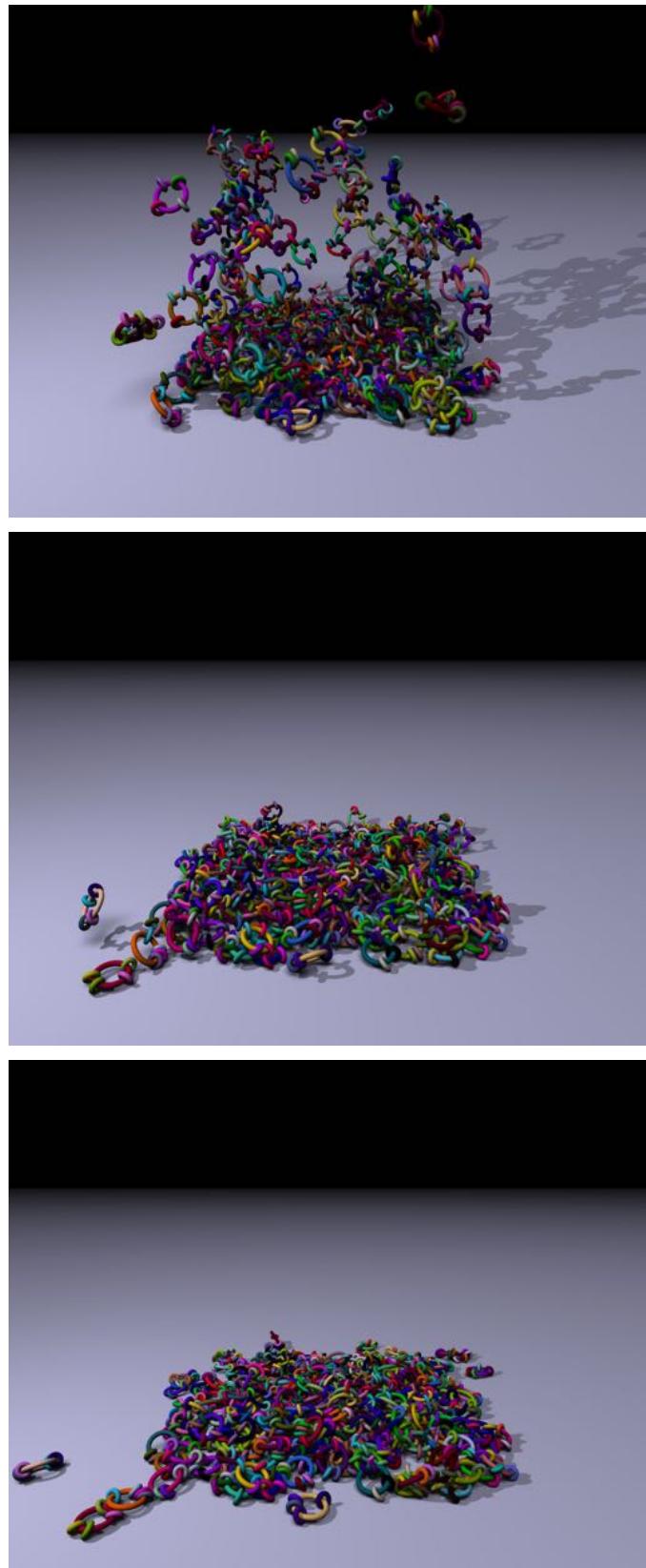


図 7.15: 組合わさったトーラスのシミュレーション

7.5 流体と剛体の連成計算

7.5.1 流体と剛体の連成計算

流体と剛体の相互作用は Carlson らの手法のようにまずは剛体を構成している粒子を流体として粒子にかかる力を計算する [19]. 剛体の粒子 i が流体から受ける力 $\mathbf{f}_{i,fluid}$ は圧力勾配による力と粘性力なので式 (3.11) と式 (3.17) を用いて

$$\mathbf{f}_{i,fluid} = \mathbf{f}_i^{press} + \mathbf{f}_i^{vis} \quad (7.30)$$

と計算される.

剛体を構成する粒子に働く力は剛体間の衝突による力 $\mathbf{f}_{i,rigid}$ と流体から受ける力 $\mathbf{f}_{i,fluid}$ であるので、剛体を構成する粒子にかかるこれらの力の和を計算することで、重心に働く力を計算することができる.

$$\mathbf{F}_c = \sum_{i \in RigidBody} (\mathbf{f}_{i,rigid} + \mathbf{f}_{i,fluid}) \quad (7.31)$$

また粒子に働く力と剛体の重心からの座標 \mathbf{r}'_i を用いて、その粒子に働く力が生む剛体のトルクを計算することができる. 剛体全体に働く力は粒子にかかるトルクの和を取り

$$\mathbf{T}_c = \sum_{i \in RigidBody} \mathbf{r}'_i \times (\mathbf{f}_{i,rigid} + \mathbf{f}_{i,fluid}) \quad (7.32)$$

と計算できる. 計算された力によって剛体の重心座標とクオータニオンを更新する. そして剛体を構成している粒子の座標をその粒子の初期配置、剛体の重心座標とクオータニオンから計算し、粒子の座標も更新する.

7.5.2 実装

GPU 上で物理計算を行なうため、物理量は全てテクスチャとしてビデオメモリ上に保持される. 流体シミュレーションで各粒子が持つ物理量は座標と速度と密度であるため、それぞれテクスチャを用意する. 計算を行なうにあたり、流体の密度を計算しなければならないが、密度はそれぞれのタイムステップで粒子座標から計算されるため、1枚のテクスチャを用意する. また剛体シミュレーションでは各剛体は座標、クオータニオン、並進運動量、角運動量であるため、それぞれにテクスチャを用意する. また慣性テンソルと初期粒子配置は同じ形状の剛体ならば共通のデータなので剛体ごとに用意する. 流体と剛体シミュレーションで必要なこれらの物理量は計算要素 1 つに対して 1 テクセル割り当てる. つまり流体粒子 1 個と剛体 1 個に 1 テクセルを割り当て、その物理量をテクセルに保持する.

表 7.7: 流体粒子数とフレームレート.

	Number of particles	FPS
Figure 7.16	49,153	17.1
Figure 7.17	64,512	16.2
Figure 7.18	64,512	15.2

7.5.3 結果

本手法を Core 2 X6800 CPU, 2.0GB のメモリ, GeForce 8800GTX を搭載した PC 上で実装した。プログラムは C++, OpenGL, C for Graphics を用いて開発した。本章で取り上げる計算結果は全てリアルタイムで計算した結果であり、それらの計算結果をオフラインでレイトレーシングしたものである。流体は計算粒子それぞれに濃度分布を与え、陰関数曲面を Marching Cubes を用いて抽出した [80]。

流体の計算に比べ剛体の計算は 2 つの理由から計算コストが低い。1 つの理由として、流体計算では近傍粒子探索を密度の計算と圧力項の計算において 2 回行わなければならないが、剛体計算では衝突計算で 1 回行うだけで済む。2 つ目の理由は衝突計算を行なうとき、粒子 i が入っているボクセルと、そのボクセルに隣接しているボクセルに格納されている粒子番号を取り出さなければならない。しかし流体計算を行なうときには影響半径が粒子径よりも大きいときには、それ以上のボクセル内に近傍粒子が存在している可能性があるため、より多くのボクセルから粒子番号を取り出さなければならない。

流体と剛体の連成計算結果を示す。図 7.16 では 10 個のグラスを積み重ねたシーンに 2 カ所から流体を流入させたシミュレーション結果である。流体がグラスを上から満たしていった。そして流体流入後、1 個のグラスを投げ入れ、積み重なったグラスを崩れ落とした。剛体の相互作用では静止摩擦力の計算をせず、動摩擦力の計算しか行なっていないが、このような複数の剛体が積み重なったシーンでも剛体をほぼ静止させることができた。図 7.18 では水槽を少量の水で満たし、256 個のチェスの駒を落とした。その後液滴を水槽に落とした。チェスの駒の密度として 2 種類の値を用いたため、重い駒は沈み、軽い駒は浮いている。図 7.17 では 1 軸で空中に固定された 5 個のギアが配置されたシーンにおいて 2 カ所から流体を流入させた。流体によってギアが回転し、その抵抗により流れが乱れている様子が計算されている。これらの計算に用いた粒子数と 1 タイムステップの計算時間を表 7.7 に示す。約 6 万粒子を用いた流体と剛体の計算は約 17fps で計算できている。

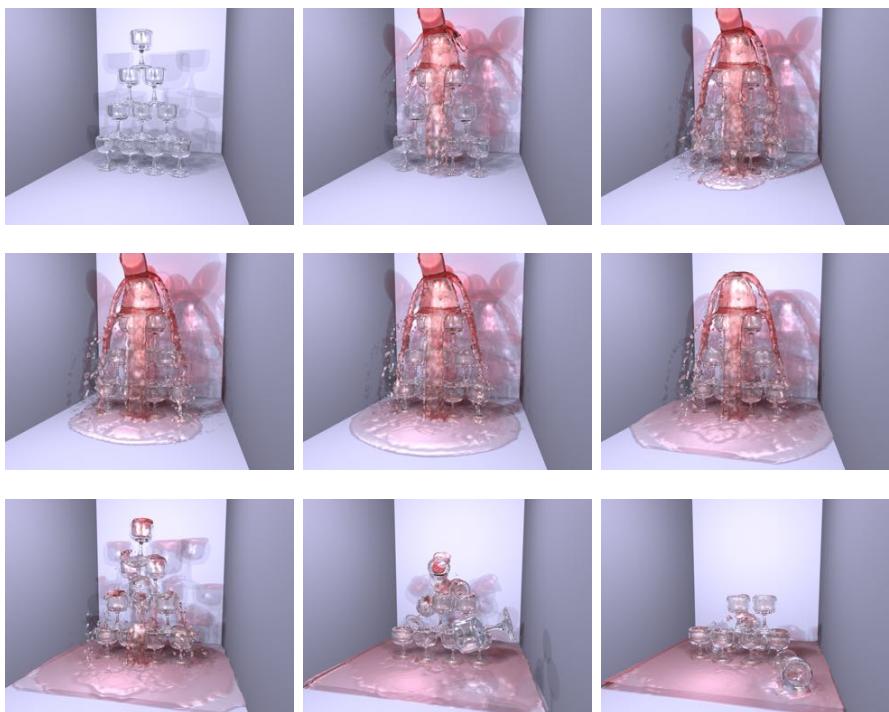


図 7.16: 11 個のグラスと流体のシミュレーション

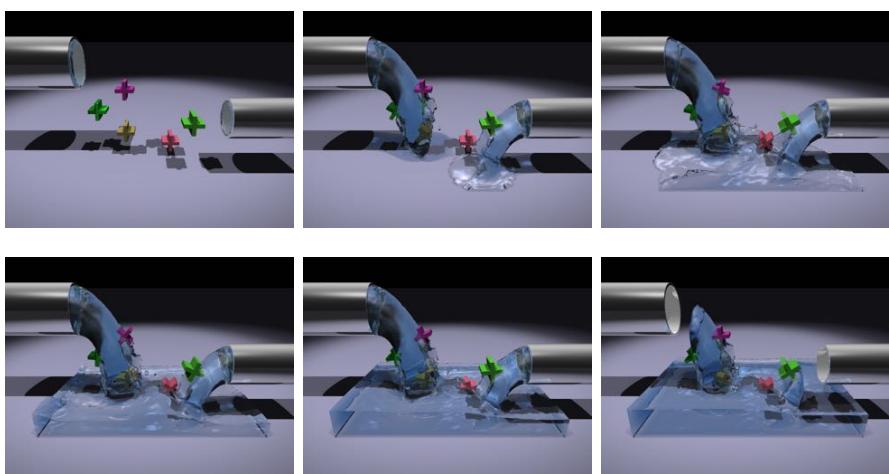


図 7.17: 剛体と流体の連成シミュレーション

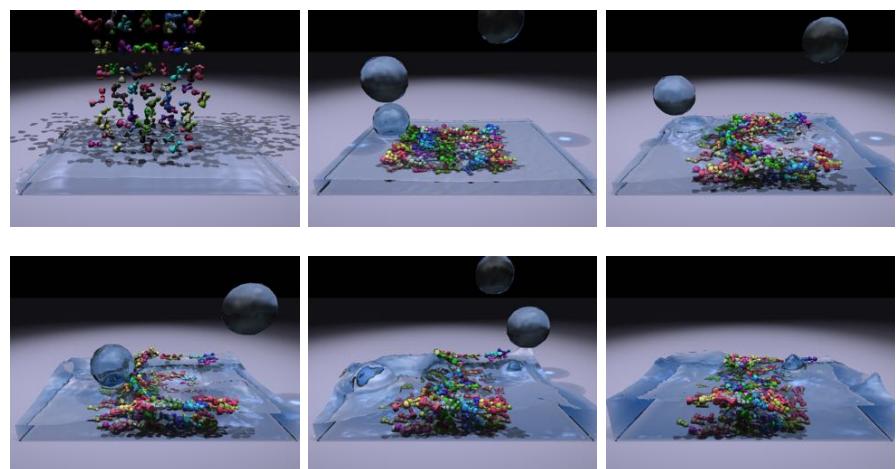


図 7.18: 2種類のチェスの駒と流体の連成シミュレーション

7.6 流体と布の連成計算

7.6.1 流体シミュレーション

SPHによる流体シミュレーションは3に示したもの用いた。またGPUでの実装に関しては7.3に示したもの用いた。

7.6.2 布シミュレーション

布と流体の相互作用から布に働く力は外力として求まる。そのため、どのような計算モデルを布に用いることもできる。本研究では布のモデルに粒子同士が2種類のバネで繋がれているモデルを用いた(図7.19)。そして時間積分にはオイラー陽解法よりも安定性の高いベルレ法を用いた。この手法は計算コストが低く安定性が高いため、リアルタイムシミュレーションに向いている。時刻 $t+dt$ での粒子*i*の座標 \mathbf{x}_i^{t+dt} は以下のように計算される。

$$\mathbf{x}_i^{t+dt} = \mathbf{x}_i^t + d(\mathbf{x}_i^t - \mathbf{x}_i^{t-dt}) + \frac{\mathbf{F}_i dt^2}{m_i} \quad (7.33)$$

m_i, d, dt はそれぞれ粒子の質量と減衰係数、時間刻み幅である。 \mathbf{F}_i は外力であり、重力 $m_i\mathbf{g}$ とバネによる力 $\mathbf{F}_{i,spring}$ と流体から受ける力 $\mathbf{F}_{i,fluid}$ の和である。バネによる力 $\mathbf{F}_{i,spring}$ は以下のように計算される。

$$\begin{aligned} \mathbf{F}_{i,spring} = & \sum_{j \in N_{adj}} k_{adj}(|\mathbf{x}_{ij}| - l_{adj}) \frac{\mathbf{x}_{ij}}{|\mathbf{x}_{ij}|} \\ & + \sum_{j \in N_{diag}} k_{diag}(|\mathbf{x}_{ij}| - l_{diag}) \frac{\mathbf{x}_{ij}}{|\mathbf{x}_{ij}|} \end{aligned} \quad (7.34)$$

k_{adj}, l_{adj} は上下左右の粒子を接続しているバネのバネ定数と自然長であり、 k_{diag}, l_{diag} は斜めの粒子を接続しているバネのバネ定数と自然長である。 \mathbf{x}_{ij} は粒子*i*から見た粒子*j*の相対座標である。

7.6.3 流体と布の相互作用

空間分割

SPH法も含めて粒子法では粒子にかかる力を計算するために近傍粒子を探索しなければならない。近傍粒子を*n*個の全粒子から探索するとその計算コストは $O(n^2)$ になるため効率が悪い。そこで計算領域を3次元格子で覆い、格子のそれぞれのボクセル内に存在する粒子番号をそのボクセルに対応づけることで、近傍粒子探索を効率化することができる[86]。本

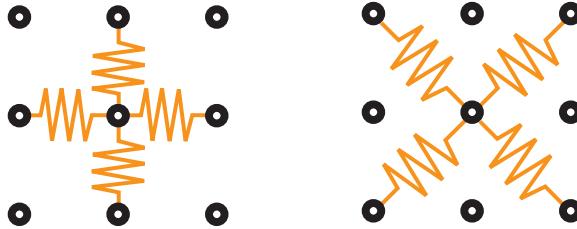


図 7.19: 布のモデル化に用いられた 2 種類のバネ.

研究でも流体の近傍粒子探索においてこの手法を用いる。また布を構成するポリゴンと流体粒子の相互作用を計算するときには流体粒子とポリゴンとの最小距離を求める必要があるため、 m 個の全ポリゴンと距離を計算するとその計算コストは $O(mn)$ になってしまう。そこでこの計算も格子を導入し計算領域を分割することで効率化することができる。流体粒子はそれを質点として 3 次元格子内の座標を計算し、そのボクセルに粒子番号を格納する。布の場合はポリゴンが大きさを持った平面であるため、複数のボクセルに面の番号を記録しなければならない。しかし本手法ではポリゴンの重心座標が存在するボクセルにポリゴンの番号を記録する。すなわち、複数のボクセルにまたがって存在するポリゴンでも、その番号は 1 つのボクセルにしか記録されない。このように流体と布をカップリングするために、流体粒子と布のポリゴン用の 2 個の格子を導入する。

本研究で用いる布を構成するポリゴンは全て同じ大きさであり、布の物理的性質により計算中に大きく面積を変えることはない。そこで布の平均の大きさに応じてボクセルの 1 辺の長さを調節する。流体粒子の相互作用を計算するときには 1 辺が粒子径の大きさのボクセルを用いて空間分割を行なうが、本研究で用いた布ポリゴンの大きさは流体粒子の数倍の大きさであるため、布ポリゴンを格納するボクセルは 1 辺が粒子径の数倍のボクセルを用いた。

衝突検出

流体粒子と布との相互作用を計算するには、まず流体粒子と布との距離を計算する必要がある。流体粒子と布までの距離は粒子に最も近い位置に存在するポリゴンもしくはその辺、頂点までの距離であるため、これを探して布との距離を計算する。粒子と距離が最も小さいポリゴンを全てのポリゴンから探索する必要があるが、ポリゴンの番号が格納された格子を用いることで限られた数のポリゴンとの距離を計算するだけでよい。この処理ではまずそれぞれの粒子座標から布の 3 次元格子内での座標を計算す

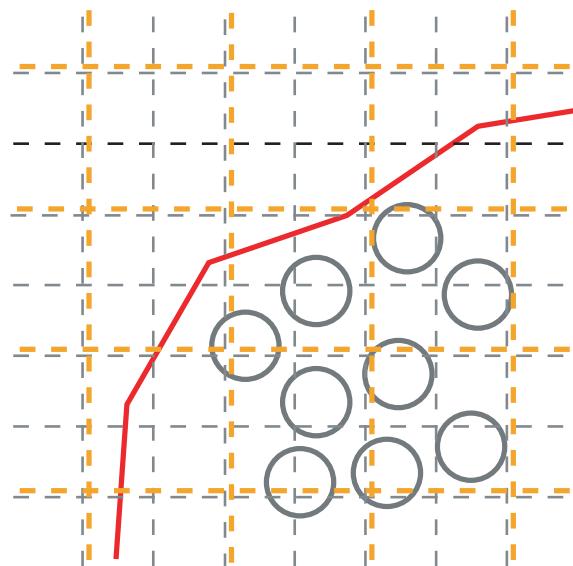


図 7.20: 二種類のグリッド。灰色の点線は流体粒子が格納される格子であり、橙色の点線は布のポリゴンが格納される格子である。

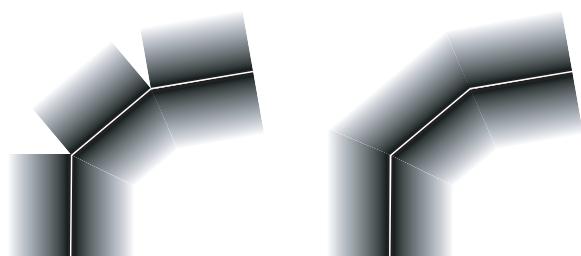


図 7.21: 不連続な距離場(図左)と連続な距離場(図右)。

る。その粒子と近距離に存在するポリゴンは、粒子が存在するボクセルとその周辺のボクセルの計 27 個のボクセル内にポリゴンの番号が格納されているので、それらのポリゴンとの距離を計算する。

布と流体粒子との相互作用は、布がある厚みを持っていると考え、粒子のめりこみに応じた力を加えることによって計算することができる。しかし布との距離の計算を粒子と布を構成するポリゴンとの距離を計算して最小値を取り粒子に働く力を計算すると、その力が空間で不連続になるため、粒子が突然飛び跳ねてしまうことがある。例えばポリゴンの接続部分近辺に存在する粒子からそれぞれのポリゴンに下ろした垂線の足はどのポリゴン上にもないため、この粒子には力は働くかない。この粒子に最も近い位置に存在するものはポリゴンの辺か頂点であるため、これらからの距離を計算すれば連続になる。しかしこの計算によって計算コストは大幅に増加するため、近似的にこのような接続部分においても粒子に働く力を連続にする手法を用いる。粒子の近傍に存在する n 個のポリゴンを $T = \{t_0, t_1, \dots, t_n\}$ として、まずポリゴン j の重心座標との距離を計算し、この値が最も小さいポリゴンを最も近くに存在するポリゴン $t_{closest}$ とする。ポリゴン j を構成する 3 頂点の座標 $\mathbf{v}_{j,0}, \mathbf{v}_{j,1}, \mathbf{v}_{j,2}$ を用いて、この条件は以下のように表すことができる。

$$t_{closest} = \arg \min_{t_j \in T} \left(\mathbf{x} - \frac{\mathbf{v}_{j,0} + \mathbf{v}_{j,1} + \mathbf{v}_{j,2}}{3} \right)^2 \quad (7.35)$$

ここで 3 頂点は同じ質量を持つものとし、重心座標は 3 頂点座標それぞれに $1/3$ の重みを付けて足し合わせたものとなる。

そしてポリゴン $t_{closest}$ との距離 d_t をこのポリゴンを構成する 3 頂点の座標を用いて以下のように求め、布と流体粒子までの距離とする。

$$d_t = |(\mathbf{v}_{j,1} - \mathbf{v}_{j,0}) \times (\mathbf{v}_{j,2} - \mathbf{v}_{j,0}) \cdot (\mathbf{x} - \mathbf{v}_{j,0})| \quad (7.36)$$

このように計算することでポリゴンの接続部分でも力を連続にすることができる、求まった粒子と布との距離が布の厚さ ϵ よりも小さければ布と粒子は衝突していると判定する。本研究では布の厚さ ϵ は粒子間平均距離とした。ポリゴンの接合部以外でも布の境界部分で力が不連続になる。しかし、この不連続性は計算結果に視覚的に大きな影響を与えたかったので、特別な処理は行なわなかった。

流体の衝突応答

流体の圧力項は粒子数密度を一定にするように働く力であり、粒子間距離が初期粒子間距離よりも近い場合には初期粒子間距離に戻す力となる。布と流体粒子間の力もこのようにモデル化する。つまり流体粒子 i と布と

の距離 d が ϵ より小さいときには、距離を ϵ に戻す力を加える。流体粒子が衝突しているポリゴンの法線ベクトル \mathbf{n} を用いて反発力 \mathbf{f}_i^{press} は以下のように計算する。

$$\mathbf{f}_i^{press} = m_i \frac{(\epsilon - d)\mathbf{n}}{dt^2} \quad (7.37)$$

また流体計算の影響半径内に布が存在する場合は粘性力を計算する。粒子座標から布に垂線を下ろし、その垂線に垂直な平面上に均一に粒子が並んでいると仮定する。そしてそれらの粒子が全て同じ速度 \mathbf{v} と密度 ρ を持つとして以下のように布からの粘性力を計算する。

$$\mathbf{f}_{i,wall}^{vis} = -\mu \frac{m}{\rho} (\mathbf{v} - \mathbf{v}_i) \sum_{j \in Wall} \nabla W_{vis}(\mathbf{r}_{ij}) \quad (7.38)$$

布上で粒子は均一に並んでいると仮定しているので、それらの粒子の配置は流体粒子 i と布までの距離によって一意に決まる。そのため、重み関数の和はあらかじめ計算しておき、そのテーブルから値を参照することで計算することができる。

流体粒子が布と近づくときに流体粒子の密度が上昇するが、布上に粒子が存在しないので布に近づいた流体粒子の密度が下がってしまう。これを防ぐために、布による密度の寄与を流体とは別に計算する。ここでも粘性項と同じように粒子座標から布に垂線を下ろし、その垂線に垂直な平面上に均一に粒子が並んでいると仮定して、以下のように布による密度の寄与 $\rho_{i,wall}$ を計算する。

$$\rho_{i,wall} = m \sum_{j \in Wall} W(\mathbf{r}_{ij}) \quad (7.39)$$

布上に均一に粒子が存在していると仮定するとそれらによる密度への寄与の重み関数の和もあらかじめ計算することができる。

布の衝突応答

布に働く力は流体に働く力の反力である。よってあるポリゴン i に働く力 \mathbf{F}_i はそのポリゴンに衝突した流体粒子に働く力 \mathbf{f}_j を用いて求めることができる。ポリゴン i に働く力 \mathbf{F}_i は以下のように求めることができる。

$$\mathbf{F}_i = - \sum_{j \in Colliding} \mathbf{f}_j \quad (7.40)$$

この力は厳密には流体粒子に働く力の作用点と力の方向を考慮しなければならないが、ここではそれらの力は面の法線方向と平行であり、全てポリ

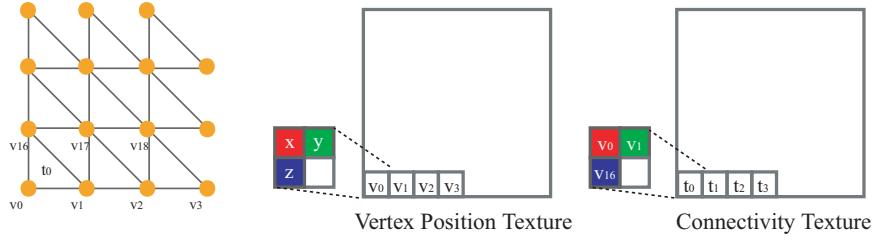


図 7.22: 頂点座標とポリゴンの接続情報のデータ構造。頂点座標は頂点座標テクスチャに、接続情報テクスチャには頂点番号を格納する。

ゴンの重心座標に働くと近似した。するとポリゴンを構成する3頂点に働く力 $\mathbf{f}_{i,0}, \mathbf{f}_{i,1}, \mathbf{f}_{i,2}$ は、3頂点に同じように分配されるので

$$\mathbf{f}_{i,0} = \mathbf{f}_{i,1} = \mathbf{f}_{i,2} = \frac{\mathbf{F}_i}{3} \quad (7.41)$$

と計算することができる。

7.6.4 実装

布の格子の構築

布のポリゴンの番号を格納する格子を構築するために、まずはポリゴンの重心座標を計算する。そしてそれぞれの重心に頂点を1個割り当て、その頂点を格子内での重心の座標に動かし、流体粒子の格子を構築するのに用いた方法と同様な方法を用いて行なう。粒子番号のかわりにポリゴン番号を色と深度として出力する。

相互作用の計算

圧力項は複数段階に分けて計算する。まずは流体粒子と布との衝突判定を行なうため、作成した布の格子を用いてそれぞれの粒子から最小距離に存在する布ポリゴンの番号と距離を求める。この処理はそれぞれの粒子の格子内での座標を計算し、そのボクセルと近傍のボクセルに格納されているポリゴンの番号を求ることで行なうことができる。そしてこの番号を用いてポリゴンを構成する3頂点の座標を読み出し、7.6.3で述べた手法を用いて布と流体粒子までの距離を求める。このときに最近距離に存在するポリゴンの番号も出力する。布と流体粒子との距離を計算することができたので、その距離が ϵ より小さい場合には式(7.37)を用いて流体粒子に働く圧力項を計算する。

次に布のポリゴンに働く力を計算する。この計算では粒子に働く力の計算と同様にポリゴンの周辺に存在する粒子座標を探して力を計算することも可能であるが、本研究で用いたポリゴンは流体粒子よりも大きいため、多数の粒子と距離計算を行なわなければならない。しかし流体粒子に働く力とその粒子と最小距離に存在するポリゴンの番号が分かっているため、ポリゴンに働く力は粒子に働く力の和を取ることで近傍粒子を探索せずに計算することができる。布を構成する頂点に働く力は2次元テクスチャ上のテクセル1個ずつに格納される。また流体粒子に働く力も2次元テクスチャ上のテクセル1個ずつに格納されている。この処理は1対1の写像ではなく、粒子に働く力を分散させる処理であるため、フラグメントシェーダでは処理することができない。そこでバーテックスシェーダで流体粒子に働く力のテクセルを読み出し、その力を対応する布の粒子の力を保持するテクスチャのピクセルに1ピクセルの点として出力することによって計算することができる。流体粒子に働く力の反力は布のポリゴンに働くため、そのポリゴンを構成する3粒子のピクセルに力を出力しなければならない。この処理は流体粒子1個に3頂点を割り当て、それぞれ3個の粒子のピクセルに力を出力するようにして処理する。またポリゴンを構成する頂点は各ポリゴン間に共有されているため、ある頂点に働く力は共有されているポリゴン全てに働く力の和となる。そこでアルファブレンディングを用いてそれらの和を計算する処理を行なった。

7.6.3で述べたように粘性項と密度への布の寄与の計算において重み関数の和はあらかじめ計算しておくことができる。影響半径内の数点において値を計算し、それ以外の値は計算した点の値を線形補間することで求めた。そして計算した点での値を1次元テクスチャに格納してGPUに送った。流体の密度と粘性項の計算では求めた布までの距離を用いて布の寄与を計算した。

7.6.5 結果

本手法をCore 2 X6800 CPU, 2.0GBのメモリ, GeForce 8800GTXを搭載したPC上で実装した。プログラムはC++, OpenGL, C for Graphicsを用いて開発した[36]。本章で示す結果はそれぞれの流体粒子に濃度分布を与え、それらの和を取り陰関数曲面を構築した。そしてそれらから流体の表面をMarching Cubesを用いて抽出し、レイトレスシングを行なった[80]。

図7.25に示すシーンは1枚の布の上に流体を流入させた結果である。流体を流入させているときには圧力によって布が変形し大きくなんでいるが、流入が終わるとたわみが解消されている。図7.24は1枚の布に複数の液滴を落下させた計算結果である。液滴が布に接触すると布が形を変えて

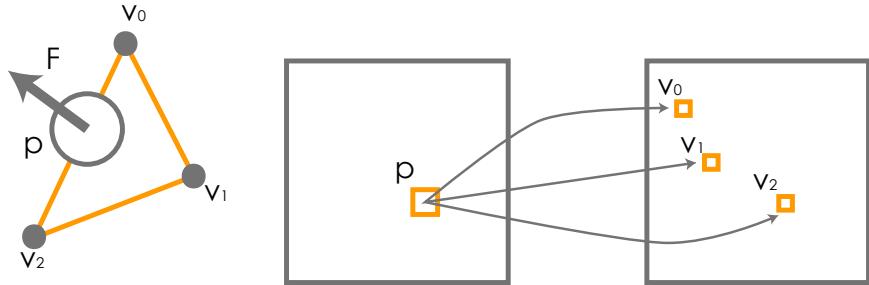


図 7.23: ポリゴンにかかる力の計算. 粒子 p が頂点 v_0, v_1, v_2 から構成されるポリゴン t と衝突したとき、流体粒子に働く力 \mathbf{F} が計算され、粒子に働く力を格納するテクスチャに書き込まれる。またそれと同時に衝突しているポリゴンの番号 t も書き込まれる。メッシュにかかる力は流体粒子に働く力の反力であるので、流体粒子にかかる力を 3 頂点に分配することによって計算する。

表 7.8: 流体粒子数とフレームレート

	16,384	65,536
Figure 7.24	53.3	14.9
Figure 7.25	49.2	13.1
Figure 7.26	45.7	12.3
Fluid only	64.1	16.6

いるのがわかる。また 2 枚の布を用いた結果を図 7.26 に示す。これらの計算では全て 65,536 個の流体粒子を用いており、1 枚の布は 8,192 枚のポリゴンから構成されている。これらの計算のフレームレートと流体粒子数を 16,384 個に減らしたシミュレーションのフレームレートを表 7.8 に示す。また布との相互作用を計算せずに流体だけの計算を行なった場合の結果も表 7.8 に示しており、16,384 粒子を用いた場合の計算が約 16fps、65,536 粒子を用いた場合が約 64fps であった。この結果よりカップリングの計算コストは流体計算に比べ低いことがわかる。本手法は布同士の衝突や布の自己衝突を計算していないため、それらを計算することはできない。布同士の衝突等は今後の研究課題の 1 つである。

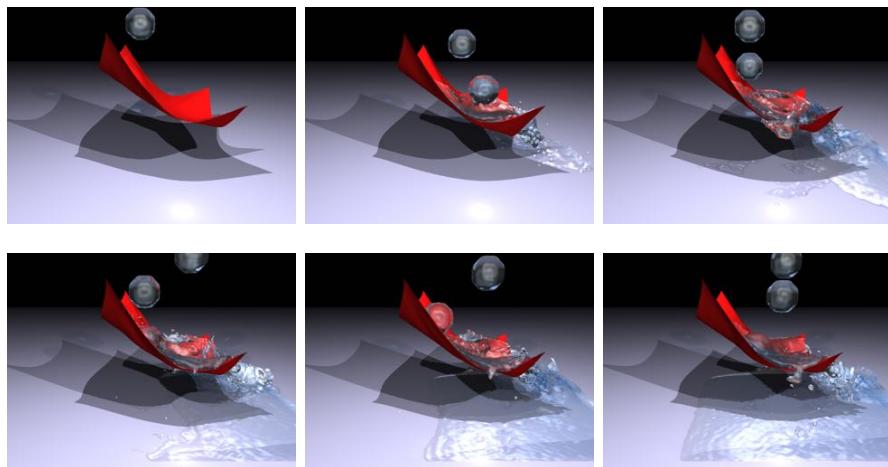


図 7.24: 1枚の布と流体の連成シミュレーション

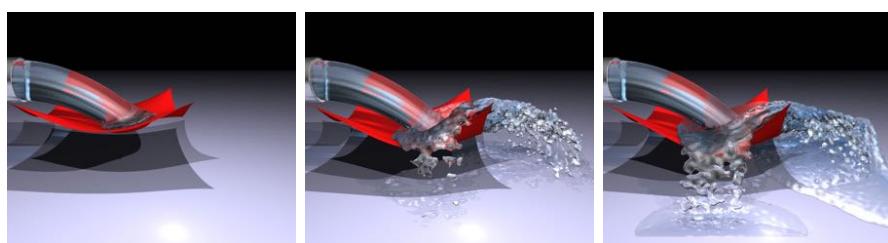


図 7.25: 2枚の布と流体の連成シミュレーション

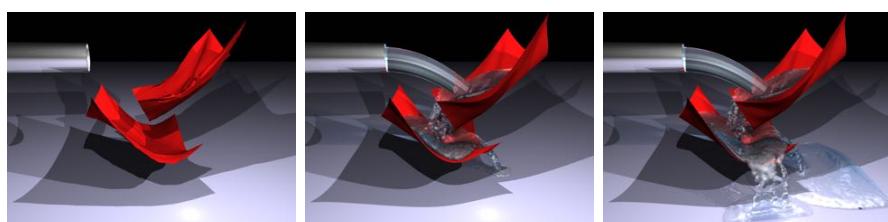


図 7.26: 2枚の布と流体の連成シミュレーション

7.7 弹性体シミュレーション

本研究で用いた弹性体の計算手法は変形勾配テンソルを用いて計算する手法を用いた [140, 134, 92]。弹性体の計算手法にはひずみ速度を求め、ひずみも更新していく手法も存在するが [42]、この手法の欠点は数値積分によって数値誤差も蓄積していくことである。特に单精度で計算をする GPU などでは、さらにこの問題が悪化する。しかし変形勾配テンソルを用いた計算手法では、現在の粒子配置からひずみ、応力を求めるので、数値積分によって数値誤差が蓄積することがないという点であり、GPU などにも適している。

7.7.1 変形勾配テンソルを用いた弹性体計算

弹性体の運動は、弹性体の変形勾配テンソルから応力を求めて計算される。材料座標系を $\mathbf{x} = (x, y, z)$ とすると $\mathbf{u} = (u, v, z)$ の変形勾配テンソル \mathbf{F} は、

$$\mathbf{F} = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{pmatrix} \quad (7.42)$$

となる。そして Green-Lagrange ひずみテンソルは、 \mathbf{F} を用いて、

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T \mathbf{F} - \mathbf{I}) \quad (7.43)$$

と求まる。ひずみテンソルを用いて応力テンソルは、4階のテンソル \mathbf{C} を用いて、

$$\mathbf{S} = \mathbf{C}\mathbf{E} \quad (7.44)$$

と計算できるが、等方的な弹性体の場合には、 \mathbf{C} はヤング率をポアソン比により計算でき、ラメの定数を用いて、第二 Piola-Kirchhoff 応力テンソルは、

$$\mathbf{S} = \lambda Tr(\mathbf{E})\mathbf{I} + 2\mu\mathbf{E} \quad (7.45)$$

と求まる。

弹性体を粒子で離散化し、ここまで定式化を用いて各物理量を粒子ごとに計算する。そしてこれらの物理量を用いて、粒子 i の運動方程式は、

$$m \frac{\partial \mathbf{v}_i}{\partial t} = \frac{m}{\rho} \sum_j (\mathbf{F}_i \mathbf{S}_i \mathbf{r}_{ij} \mathbf{A}_i^{-1} + \mathbf{F}_j \mathbf{S}_j \mathbf{r}_{ij} \mathbf{A}_j^{-1}) w_{ij} \quad (7.46)$$

と表される。なお次に説明するように $\mathbf{A}_i = \sum_j \mathbf{u}_{ij} \mathbf{u}_{ij}^T w_{ij}$ である。

変形勾配テンソルの計算

粒子 j のある物理量 \mathbf{p}_j とし、その近傍に存在する粒子 i の物理量 \mathbf{p}_i を、 \mathbf{p}_j を用いて一次近似すると、 $\tilde{\mathbf{p}}_i = \mathbf{p}_j + \mathbf{F}_i \mathbf{u}_{ji}$ となる。粒子 i の近傍粒子による \mathbf{p}_i の近似による誤差 \mathbf{e}_i は、ある重み関数 w を用いて、

$$\begin{aligned}\mathbf{e}_i &= \sum_j (\mathbf{p}_j - \tilde{\mathbf{p}}_i)^2 w_{ji} \\ &= \sum_j (\mathbf{p}_{ij} - \mathbf{F}_i \mathbf{u}_{ij})^2 w_{ij}\end{aligned}\quad (7.47)$$

と定義でき、この誤差を最小にする変形勾配テンソルは、この式を \mathbf{F}_i で偏微分し、その値が 0 になるものである。すると

$$\mathbf{F}_i \sum_j \mathbf{u}_{ij} \mathbf{u}_{ij}^T w_{ij} - \sum_i \mathbf{p}_{ij} \mathbf{u}_{ij}^T w_{ij} = \mathbf{O} \quad (7.48)$$

となる。これより変形勾配テンソル \mathbf{F}_i は、

$$\mathbf{F}_i = \left[\sum_j \mathbf{u}_{ij} \mathbf{x}_{ij}^T w_{ij} \right] \left[\sum_j \mathbf{u}_{ij} \mathbf{u}_{ij}^T w_{ij} \right]^{-1} \quad (7.49)$$

と求まる。

7.7.2 実装

ここでは近傍粒子は動的に変化させずに計算を行なった。そのため、近傍粒子探索は前計算となり、その結果をテーブルに書き出し、これを参照することで各タイムステップにおいて計算を行なった。プログラムとしては変形勾配テンソルを求めるカーネル、歪みを求めるカーネル、これらを用いて座標と速度を更新するカーネルの 3 カーネルから構成される。

またシミュレーションに用いる各粒子のデータである座標、変形前の座標、速度、応力等は全粒子において同じ値であるので、それぞれの型の粒子数だけ要素がある配列として格納したが、近傍粒子については各粒子によって粒子数が異なるため、近傍粒子数の最大を求め書く粒子においてその数だけのメモリを確保した。この方法では近傍粒子数が少ない粒子ではメモリに無駄が生じる。別の方針としては近傍粒子の番号を密な配列に書き出し、さらに各粒子のデータへのオフセットの配列も確保する方法であるが、この方法はメモリに無駄が生じないが、各粒子のデータにアクセスする際に余分なメモリアクセスが生じる。計算速度を優先させるアプリケーションの場合は多少のメモリの無駄は許容する前者方が効率的であるため、本研究ではそのフォーマットでデータを格納した。

7.7.3 結果

本手法を実装し, Intel Core 2 Duo E8500, 3.16GHz の CPU と 4.0GB のメインメモリ, GeForce 9800GX2 を搭載した PC にて計算速度を検証した. 用いたグラフィックスボードには 2 個の GPU が搭載されているが, 本研究では 1 個のみ用いた. これは 2 個の GPU を用いる場合には 1 個の GPU 上の並列性だけでなく, 2 個の GPU の間での計算の分散も考慮しなければならないため, 1GPU に対して開発した計算モデルをそのまま用いることができないからである.

計算体系は図 img:1a:elasticConfig に示すものであり, 弾性体的一面を固定した. 用いた物理量はそれぞれ, ヤング率 $1.5 \times 10^6 \text{ Pa}$, ポアソン比 0.4, 密度 $15,22 \text{ kg/m}^3$ である. 本研究では 1CPU を用いたプログラム, 1CPU 上で Streaming SIMD Extensions(SSE) を用いたプログラム, そして GPU を用いたプログラムの 3 プログラムを開発し, 計算解像度を変化させ計算速度を測定した. 全てのプログラムでは 32bit 浮動小数点を用いており, 64bit 浮動小数点は用いていない. 表 table:computationTimeElastic と図 img:1a:elasticGraph に粒子数を 2,187, 5,184, 17,172 粒子と変化させた結果を示す. まず CPU 上で SSE を用いた場合と用いない場合の計算は粒子数が少ない場合にはあまり差がないが, 粒子数 17,172 の計算では SSE を用いた計算が約 4 倍ほどの速度で計算できている. また GPU と CPU(SSE を用いない) での計算速度を比較すると, GPU の方が高速だが計算速度は最大約 10 倍に止まっている. 弹性体の計算は DEM や SPH を用いた計算等と比較すると GPU を用いることでの計算速度の向上は少ない. また一般的に GPU は計算要素数が多いほど計算の並列度が増し, その計算能力を引き出せるようになるが, ここで行なったテストでの最大粒子数である 17,172 での計算時間を GPU と SSE を用いた CPU での計算と比較すると約 2 倍にしかなっていない. また本研究で用いた CPU はデュアルコアプロセッサであるが, プログラムは 1 コアしか用いておらず, CPU の性能の半分しか用いていない. 弹性体の計算は計算の各段階は粒子ごとのデータの依存性はないため, マルチスレッドで並列に計算を行なうことも比較的容易である. 仮に 2 コアを用いて 2 スレッドで計算を行い, マルチスレッドのオーバーヘッドが無いと仮定すると計算速度は倍になる. すると粒子数 17,172 での計算は CPU を用いた場合と GPU を用いた場合では計算速度がほぼ同じになってしまふ. さらに本研究で用いた CPU は現在のフラッグシップモデルではなく, 最新の CPU である Intel Core i7 はクアッドコアであり, さらに 2 コアあたり 2 スレッドを実行できるため, 8 スレッドのプロセッサとして用いることができる. 仮にこの CPU でも弾性体の計算速度が線形にスケールするとすると粒子数 17,172 の計算が約 7ms で行なえることになり, GPU の速度を大幅に上回ってし

まう。

これは GPU 自体の性能の問題ではなく、GPU のアーキテクチャが粒子法を用いた弹性体を計算するのに向きなためであり、GPU の本来の性能を十分に引き出せていないからである。確かに粒子法を用いた弹性体の計算は計算解像度が高ければ多くの粒子上での物理量を計算しなければならないため、並列性が高く、さらに粒子のある物理量の計算は他の粒子の計算結果に依存しないため、完全に並列化可能であり、DEM や SPH の計算と同じく GPU に向いた計算のように思える。しかし DEM や SPH を用いた場合の計算と弹性体の計算の違いは 1 カーネルが計算に必要とするレジスタの数である。GPU は CPU と同じくメモリ転送速度は浮動小数点演算速度に比べ大幅に遅い。GPU は複数の計算を 1 プロセッサが行い、メモリアクセスのレイテンシを隠蔽するために、メモリアクセスが生じた場合に計算のセットを切り替えている。すなわち 1 プロセッサが十分な数の計算のセットを持っていないとメモリアクセスのレイテンシを隠蔽できず、ピーク演算性能に近づくことはできない。1 プロセッサが持つことのできる計算セットの数は 1 スレッドが必要とするリソースの量によって決まる。このリソースの 1 つがレジスタであり、1 プロセッサが使用することのできるレジスタ数は 8,192 個であり、これを複数のスレッドから構成される計算セットが分けて用いる。またプロセッサあたり同時に実行できるスレッドの最大数は 768 個であるため、最大数のスレッドを実行する場合に 1 スレッドが用いることができるレジスタ数は約 10 個になってしまう。すなわちカーネルの 1 スレッドあたり必要なレジスタ数が 10 個ならば、メモリアクセスのレイテンシを最大限に隠蔽できる可能性が出てくる。

ここで本研究の弹性体に戻る。DEM などでは大半の物理量が 3×1 ベクトルであるが、弹性体の場合はこのサイズのベクトル以外に 3×3 行列も必要とする。つまり、それぞれのカーネルが必要とするベクトルと行列の数が同じならば、必要なレジスタ数は弹性体の場合は 3 倍になってしまう。すなわち、1 スレッドあたりのリソースの必要量が多いため、実行できるスレッドの数が減少し、GPU のサイクルの多くをデータ待ちで無駄にしてしまうようになる。また 3×3 行列を 1 個読み込むだけで 9 個のレジスタを必要とする。さらに行列はベクトルに比べ 3 倍の大きさのデータであるので、メモリアクセスにかかる時間も増加する。本研究で開発したコードを nvcc コンパイラでコンパイルすることによって各スレッドが必要なレジスタ数を調べた。変形勾配テンソルから力を計算し座標を更新するカーネルは 47 個レジスタを使用していた。すなわち 1 プロセッサあたり同時に実行できるスレッド数は 173 であり、最大同時実行可能なスレッド数より大分少ない。よってメモリアクセスのレイテンシを隠蔽す

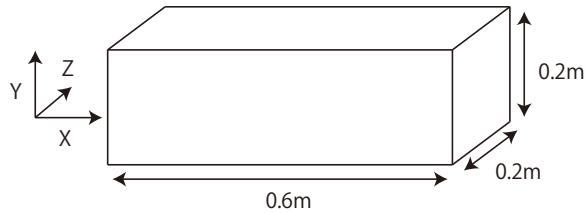


図 7.27: 計算体系

表 7.9: 計算時間(ミリ秒).

Number of particles	CPU	CPU(SSE)	GPU
2,187	28.14	21.56	3.70
5,184	69.77	41.33	8.74
17,172	239.31	58.20	28.31

ることができないことが、計算速度の低下の原因となっていると考えられる。この問題点を改善する方法の1つとして計算に局所性を持たせ shared memory を利用することが考えられるが、粒子法を用いた弾性体の計算では困難である。

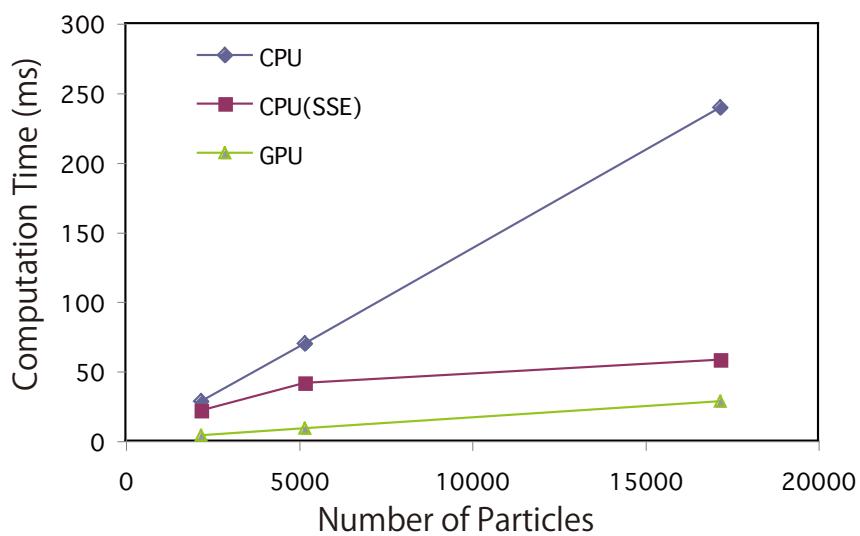


図 7.28: 計算時間の比較

7.8 撃力を用いる剛体シミュレーション

7.8.1 広域衝突検出

広域の衝突検出にはバウンディングスフィア(球)と均一格子による空間分割を用いる。

番号の均一格子への格納

まず球の番号をその球の存在する均一格子のボクセルに格納する。GPUでこの処理を実現する方法については原田らが研究しており、この手法を応用した。既存研究ではシェーダを用いた実装であり、グラフィックス特有の機能を用いたがCUDAではそれらを使用することができない。既存研究でのステンシルテストによる処理は格納された番号の数を数える処理であるため、そのかわりカウンタを用意することで同等な処理を行なうことができる。

近傍探索

番号を格納した均一格子を用いることによって近傍探索を効率的に行なうことができるようになる。この近傍探索も原田らの研究と同じように行なうことができるが、この手法では球 a と球 b は両方を近傍として探索してしまう。すると検出された衝突のペアに重複が生じるため、この重複を取り除く処理が近傍探索の後に必要になる。そこで本論文では重複したペアを検出しない近傍探索方法を提案する。図7.29に2次元の例を示して説明するが、球 i と衝突している球が存在するのは、球 a が格納されているボクセル $V_{x,y}$ とその周囲のボクセルの計9ボクセルである。この9ボクセルを全て探索すると球 a はボクセル $V_{x-1,y-1}$ に存在する球 b を近傍として見つける。しかし球 b の近傍探索でもこの球 b が格納されている周囲の9ボクセルを探索するため、ボクセル $V_{x-1,y-1}$ に存在する球 b からもボクセル $V_{x,y}$ を探索し、球 a を近傍として見つける。このように探索するボクセルのペアに重複が存在するため、近傍のペアに重複が生じる。

これを避けるためにここで2次元のボクセル $V_{i,j}$ の番号 (i,j) を1次元の番号に変換する関数 f を導入し、これらの9ボクセルのうち $f(x,y) \leq f(i,j)$ の条件を満たすボクセルのみを探索するようにする。例えば D をX軸方向のボクセルの数だとして、 $f(i,j) = i + jD$ と定義すると図7.29に示すようなボクセル $V_{x,y}$ の周囲の5ボクセルのみ探索すれば良いことになる。この f を用いるとボクセル $V_{x,y}$ に存在する球 a からはボクセル $V_{x-1,y-1}$ は探索されないため、球 b との近傍のペアは作成されない。逆に

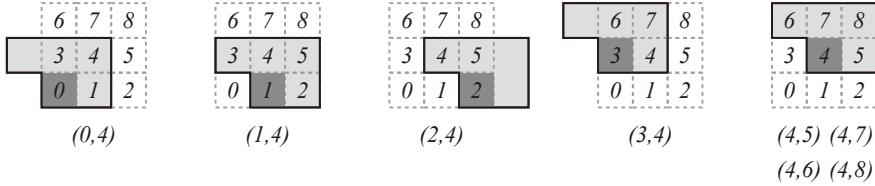


図 7.29: 非対称な近傍粒子探索. 黒色のセルが灰色のセルを探索する. 例えばセル 4 は, セル 5, 6, 7, 8 を検索する.

球 b からはボクセル $V_{x,y}$ は探索されるためここで初めて球 a との近傍のペアが作成される. ただし, ここで注意をしなければならないのは同じボクセルに存在する球同士は互いに近傍としてしまうため, その球の存在するボクセル内に存在する球は探索した球の番号が自分の番号よりも大きい場合のみ近傍としてペアを作成する.

この手法は3次元にも応用することが可能であり, その場合にはZ軸方向にも同じ条件を加え $f(x, y, z) \leq f(i, j, k)$ の条件を満たすボクセルのみ探索するようにすれば良い.

この処理をGPUで行なうときにはあらかじめ各球に対して近傍の球の番号を格納する大きさのバッファを用意しておく. 球の直径とボクセルの1辺の長さと同じにすると原田らが議論しているように1ボクセルに格納される球の最大数は4であり, 探索するボクセルは3次元では13個であるため, それぞれの球に対して 4×13 個の番号を格納するバッファを用意しておけばよい.

衝突ペアの圧縮

近傍探索から得られるのは球の総数の配列であり, 要素にそれぞれの球の近傍に存在する球の番号が格納されている. しかし, 全ての球の近傍にバッファの最大数である 4×13 個の球が存在するとは限らず, 今の状態では有効な要素がまばらな配列になっている. また番号が格納されていない要素はこれから計算の効率を下げる所以になるので, ここでこの無駄なメモリの領域を取り除く. 近傍の球の番号の配列を入力としてストリームリダクションを行い, 衝突している球のペア(衝突ペア)の配列を作成する. 図7.30にここで入力とする配列と出力とする配列を示す. またストリームリダクションではまず有効なメモリにフラグを立て, 先頭からのフラグの番号をプレフィックススキャンを用いて求める. そしてその番号を用いて配列を圧縮することで行なうことができる.

<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td><td>2</td></tr><tr><td>1</td><td>5</td><td></td><td></td><td>2</td><td>6</td><td>3</td><td>4</td><td></td><td></td><td></td><td></td></tr></table>	0	0	0	0	1	1	1	1	2	2	2	2	1	5			2	6	3	4					Input pairs
0	0	0	0	1	1	1	1	2	2	2	2														
1	5			2	6	3	4																		
<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	0	0	1	1	1	0	1	0	0	0	Set flag 1 to valid element												
1	1	0	0	1	1	1	0	1	0	0	0														
<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>2</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>6</td><td>6</td><td>6</td></tr></table>	0	1	2	2	2	3	4	5	5	6	6	6	Calculate prefix sum of the flag array												
0	1	2	2	2	3	4	5	5	6	6	6														
<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td><td>2</td></tr><tr><td>1</td><td>5</td><td></td><td></td><td>2</td><td>6</td><td>3</td><td>4</td><td></td><td></td><td></td><td></td></tr></table>	0	0	0	0	1	1	1	1	2	2	2	2	1	5			2	6	3	4					Compact the pairs
0	0	0	0	1	1	1	1	2	2	2	2														
1	5			2	6	3	4																		
<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td>5</td><td>2</td><td>6</td><td>3</td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	0	1	1	1	2							1	5	2	6	3	4							
0	0	1	1	1	2																				
1	5	2	6	3	4																				

図 7.30: 衝突ペアの圧縮.

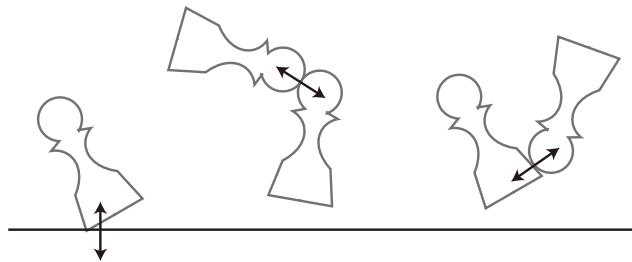


図 7.31: 並列での速度の計算が可能な例.

7.8.2 衝突ペアの組への分解

ここまでで粒子の座標から衝突ペアの配列を作成した。衝突を撃力を用いて解消するときには、撃力を逐次加えていかなければならない。逐次に計算を行なわなければならないのは撃力の計算に剛体の速度が必要であり、撃力を計算したあとに速度を更新するからである。従って GPU で全ての衝突ペア間の撃力を同時に求めて一斉に速度を更新するということはできない。しかし図 7.31 のように衝突ペア間で物体を共有していない場合にはそれらの速度を同時に更新することができる。そこで全ての衝突ペアを同じ剛体を共有していない衝突ペアの組に分解する。すなわち 1 組の衝突ペアの中にはそれぞれの剛体は一度しか出現しなく、1 組の衝突ペアを構成する剛体の速度を同時に更新しても、どの剛体も 2 つの衝突ペアから速度の更新はされない。よって 1 組の衝突ペアの衝突解決は並列で行なうことができる。そして全ての組の計算を順番に行なうことで方程式を 1 回反復することができ、これを解が収束するまでくり返していく。

この組み分けは逐次処理で行なうには問題はないが、GPU などのマルチコアプロセッサではその手法をそのまま用いることはできない (CITE Intel)。CUDA を用いて組み分けを行なうときにはまず剛体の数 n のカウンタ $C[n]$ と、衝突ペアの数 m の配列 $P[m]$ を用意して、複数回の処理とすることで行なうことができる。全ての処理では各スレッドが 1 つずつ衝

突ペアを受け持つ。まず1回目の処理では1組目の衝突ペアを抽出する。ある衝突ペア a を構成している2個の剛体を i, j とするとそれらのカウンタの値 $C[i], C[j]$ を読み出し、それらが初期化されたままであつたら、2個のカウンタをインクリメントし、衝突ペアの配列 $P[a]$ に組の番号である1を書き入れる。しかしこれはGPUの並列性により正しく行なうことができない。それは2個のカウンタに複数のスレッドが同時にアクセスする可能性があり、そうすると1つの剛体を共有している複数の衝突ペアに組の番号である1を書き込んでしまい、1個の剛体が1組の中で複数回出現してしまう。そこでCUDAの機能であるブロック内での同期を取る機能を用いる。1つの衝突ペアが剛体のカウンタの読み出し、書き込みを行なう間は他のスレッドがそのカウンタにアクセスできないようとする。この処理はあるスレッドがこの処理を始める前と終えた後に同期を取ることで行なうことができるようになる。このようにスレッドの同期を取る機能を用いることでブロック内のスレッドの処理を直列化することでブロック内のスレッドは正しくペアの組み分けを行なうことができるようになる。実際はブロック内のスレッドの数だけの同期を取る必要はなく、GPU内において物理的に同時に処理を行なう実行単位内において処理を直列化すればよい。しかしこの処理で1つのカーネルで実行している全てのスレッドで同期を取れるわけではなく、1ブロック内だけで同期を取ることができる。すなわち、1ブロック内のスレッドのみで計算を行なうことができる衝突ペアの数の計算ならば問題はないが、それ以上の大きな問題の場合はこの問題は解決していないことになる。ブロック内では処理は直列化されたが、 N ブロックのスレッドが実行された場合は N 個のスレッドが同時に処理を行なうことになり、1個の剛体が1組の中で複数回出現してしまう可能性が出てくる。そこで本研究では複数のスレッドが同時に処理を行なっている場合は、どれか1つのスレッドの処理を有効にするという手段を取り、この問題を解決した。まず衝突ペアの数の配列 $P[m]$ の代わりに、剛体の数の配列を $K[n]$ を用意する。そしてある衝突ペア a を構成している2個の剛体 i, j のカウンタ $C[i], C[j]$ が初期化されたままならば、用意した新しい配列の中のこれらの2個の剛体の番号の要素 $K[i], K[j]$ に衝突ペアの番号 a を書き入れる。例えば2つのスレッドが同時に実行され、それぞれの衝突ペアが i, j と j, k の場合、剛体 j が2つの衝突ペアの中で重複している。両方のスレッドが同時に初期化されたままのカウンタを読み出すと両方のスレッドが衝突ペアの番号 a, b を剛体の番号の要素 $K[i], K[j]$ と $K[j], K[k]$ に書き込もうとする。 $K[j]$ への書き込みの競合が起こるが、どちらかのスレッドが書き込みを成功させ、 $K[i] = a, K[j] = a, K[k] = b$ 、もしくは $K[i] = a, K[j] = b, K[k] = b$ という状態になる。そして次のカーネルでは全ての衝突ペアについてこの配

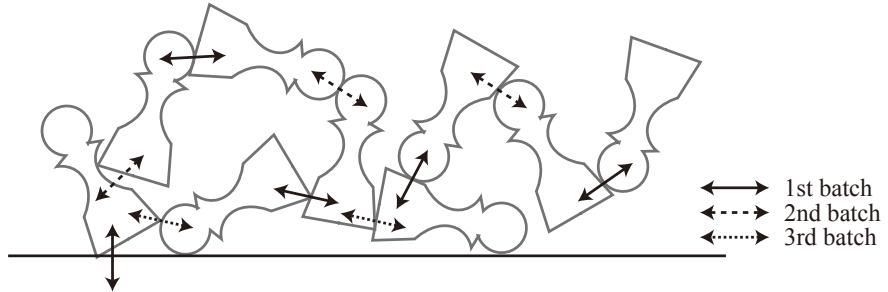


図 7.32: 衝突ペアのバッチへの分解.

列 K を調べ、ある衝突ペア a をの 2 個の剛体 i, j の配列要素 $K[i], K[j]$ の値が a と一致した場合のみ、その衝突ペアを 1 組目にする。このようにすることで 2 つのスレッドがある剛体を共有する 2 つの衝突ペアを同じ組にしようとしても共有している剛体の配列要素にはどちらかの衝突ペアしか正しくあたいを書き入れることができず、複数個のスレッドのブロックを実行する場合でも必ず剛体を共有している複数の衝突のペアが同じ組に分けられることはなくなる。

7.8.3 狹域衝突検出

広域衝突検出では正確な衝突点の情報を求めていない。狭域衝突検出ではそれぞれの衝突ペア間での正確な衝突点の情報を計算する。この段階の衝突計算手法は様々なものが開発されているが、計算コストの低さや、本研究では実装の容易さから距離関数による衝突検出を用いた。距離関数を用いた手法ではそれぞれの物体は符号付き距離関数として表され、剛体表面にサンプリングポイントが置かれる。そして 1 つの剛体のサンプリングポイントでのもう 1 つの剛体の距離関数を読み出す。するとそのサンプリングポイントでの値の符号でその点が物体内部にあり衝突しているのかを判定することができる。さらにその値自体が剛体の表面からサンプリングポイントまでの距離であり、また衝突点での法線の情報もその点での距離関数の勾配を取ることで求めることができる。

しかし距離関数を用いた衝突検出では図 7.33 のような場合に正しく衝突の法線を取ることができず、衝突していても衝突を解消することができない。これを解消するために図 7.33 のように頂点上ではなく返上にもサンプリングポイントを配置することで改善することができる。しかしこの場合でも完璧ではなく、図 7.33 のような場合に衝突を解消することができなくなる。1 辺上でのサンプリングポイントを 2 個にするとこのような問題は解決する。

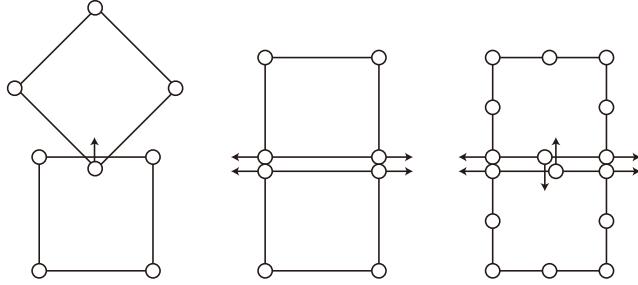


図 7.33: 距離関数を用いた衝突検出.

ここで距離関数を用いた衝突検出を用いたが、本論文の寄与である衝突ペアへの分解はどのような狭域衝突検出手法とも組み合わせができる。

7.8.4 衝突解決

衝突点での撃力の計算は、撃力を加えたときの速度の変化の式、速度の拘束条件、そして仮想仕事の原理から導出される以下の 3 式を用いて求める。

$$\mathbf{v}'_{rer} - \mathbf{v}_{rer} = \mathbf{K}\mathbf{P} \quad (7.50)$$

$$\mathbf{J}\mathbf{v}'_{rer} + b \geq 0 \quad (7.51)$$

$$\mathbf{P} = \mathbf{J}^T \lambda \quad (7.52)$$

なおここで $\mathbf{v}'_{rer}, \mathbf{v}_{rer}, \mathbf{P}, \mathbf{J}, b, \lambda$ はそれぞれ衝突後の衝突点での相対速度、衝突前の相対速度、撃力、速度の拘束条件におけるヤコビアン、速度の拘束条件における定数項、ラグランジュ乗数である。また $\mathbf{K} = \mathbf{M}_a^{-1} + \mathbf{R}_a^{*T} \mathbf{I}_a^{-1} \mathbf{R}_a^* + \mathbf{M}_b^{-1} + \mathbf{R}_b^{*T} \mathbf{I}_b^{-1} \mathbf{R}_b^*$ であり、 $\mathbf{M}_i, \mathbf{I}_i, \mathbf{R}_i^*$ はそれぞれ剛体 i の質量、慣性モーメント、衝突点の剛体 i の重心からの相対座標の外積行列である。そしてこの方程式を projected gauss seidel 法を用いて解く。

またこの方程式では速度の拘束条件のみ考慮されており、座標の拘束条件が考慮されていない。そこで Baumgarte Stabilization を用いて位置の拘束条件を加えると同時に安定化を行なった [20]。

7.8.5 結果

図 7.34 に本手法を用いて行なった 2 次元剛体シミュレーションの結果を示す。図では剛体の形状だけでなく、バッチのペアを示している。1 つの図では計算されたバッチを色分けしてしめしてある。同じバッチに含

まれている制約条件は同じ色の線で示されている。また他の図ではそれぞれのバッチの組に含まれている制約条件を1組づつ示してある。よって1つの図に描かれている剛体を結ぶ線は決して剛体を共有していない。

本研究では広域衝突検出手法の制限のため、単純な形状でしか実験できていないが、本手法つまり制約条件のバッチへの分解はどのような広域衝突検出手法とも組み合わせることができる。すなわちより複雑な広域衝突検出をCPUで行い、その後の処理を本手法を用いて全てGPUで行なうということも可能である。また本手法では必ずしも最良のバッチの組を計算できるとは限らないという欠点がある。つまり最良のバッチの数よりも多いバッチを生成してしまうことがある、これによって各バッチでの計算要素数を減らしてしまうということである。GPUは計算要素数が減るほどその能力を発揮することができないため、計算効率が下がってしまうことがある。

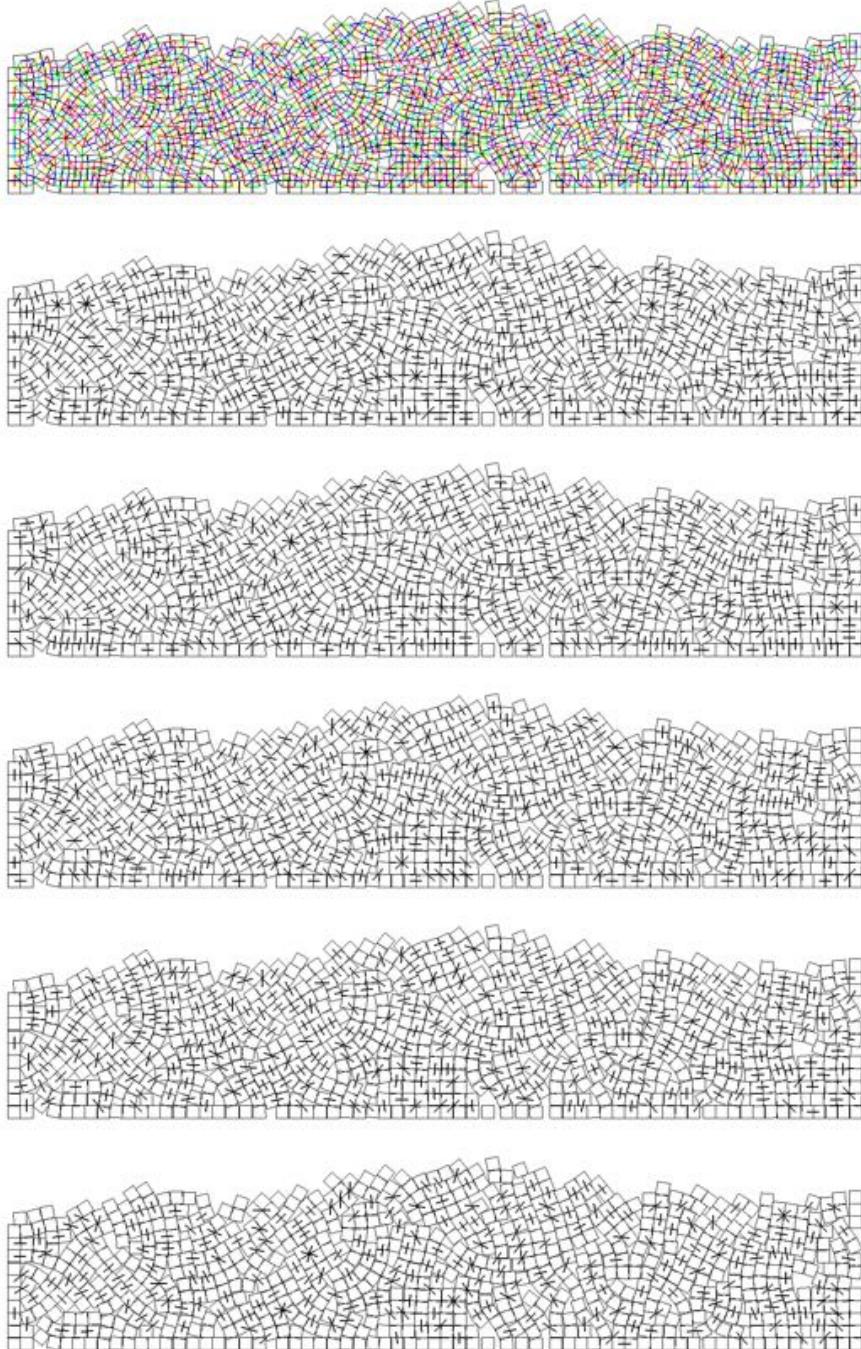


図 7.34: 計算結果

7.9 木構造を用いた広域衝突検出

粒子法のシミュレーションでは一般的に均一の大きさの計算粒子が用いられ、近傍粒子探索の高速化のデータ構造としてはユニフォームグリッドが効率的であり、良く用いられる。しかし計算粒子が均一でない粒子法も開発されており、そのような場合にはユニフォームグリッドは非効率的である。同様の問題として一般的な剛体シミュレーションが挙げられる。剛体シミュレーションでは均一な剛体のみが用いられることが多いが、一般的に様々な形状の剛体が用いられる。よって粒子径が均一でない粒子法シミュレーションと同じくユニフォームグリッドは効率的でないため、あまり用いられない。剛体シミュレーションに関して言えば、一般的に広域衝突検出は sweep and prune が用いられる。その他には階層格子、すなわち木構造を用いた探索を用いることができる。

本研究では衝突検出に用いることができる GPU 上での効率的な木構造の巡回手法を提案する。再帰やスタックを用いた木構造の巡回が一般的であるが、これらは GPU 上で実現することが困難であったり、非効率的であるなどの問題がある。本手法はこれらを用いず、さらに 1 つの要素を用いた木構造への問い合わせにおいて必要なリソースも少なく、GPU 上での実装に適しているという特徴がある。最後に CPU での木構造の巡回、GPU でのスタックを用いた木構造の巡回と比較することによって本手法の有用性を示す。

7.9.1 巡回履歴を用いた木構造の巡回

本手法では親ノードから子ノードへ、さらに子ノードから親ノードへ双方的にアクセスするため、これらのポインタ等をそれぞれのノードに格納しておく必要がある。完全な木構造ならば親ノードの番号から、子ノードの番号を計算することができるので、このような追加の情報は不要ない。本手法では、木構造を巡回する 1 つのオブジェクトにつき、それぞれ $n_{depth} \times n_{division}$ bit のメモリを巡回履歴用に確保する。なお n_{depth} は木構造の深さであり、 $n_{division}$ は 1 レベルにおける分割数である。四分木なら $n_{division} = 4$ 、八分木なら $n_{division} = 8$ である。以下では木構造が四分木であると仮定して説明をする。この巡回履歴のメモリは図に示すように 4 ビットずつに区切られ、木構造の各階層にそれぞれ 4 ビット割り当てられる。そしてそれぞれの 4 ビットはその階層のノードを示す。まずこの巡回履歴は巡回を始める前に全てのビットを 0 で初期化する。そして k 階層の探索を行なうにはこの階層に割り当てられた 4k から 4k+3 までのビットを読み出し、その中から最も小さいビットに存在する 0 の位置を得る。この階層の探索を始めた直後には 4 つの全てのビットが 0 であるので 0 番

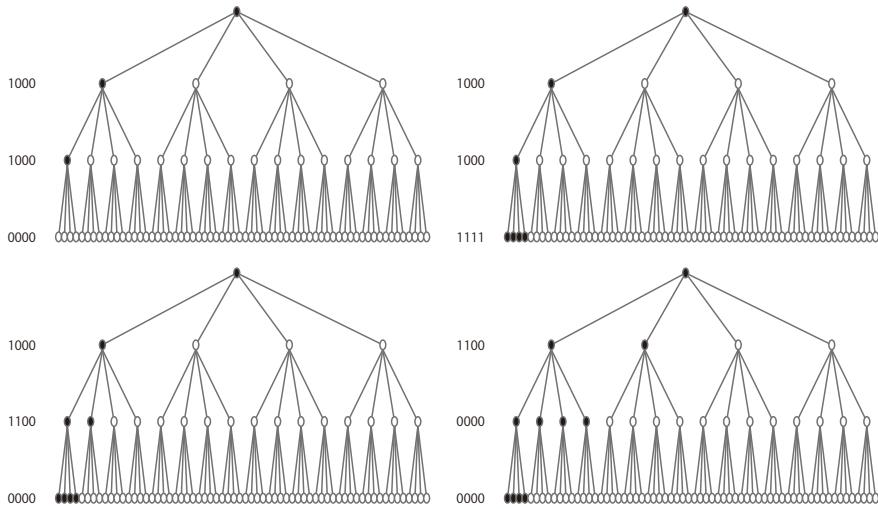


図 7.35: 四分木の巡回. (説明追加)

目を得る. そしてこのノードと交差判定を行なうと同時に巡回履歴の階層 k の 0 番目のビットを 1 に反転させる. このノードと交差していない場合は再度このレベルの巡回履歴のビットを読み最下位の 0 の位置を得る. 今度は 1 番目であるので 1 番目のノードと交差判定を行なう. 前回と同様に巡回履歴の 1 番目のビットを反転させる. ここでこの 1 番目のノードと交差している場合は, 階層 k の巡回履歴はそのままにしておき, 次の階層 $k+1$ に進む. 階層 $k+1$ の巡回履歴のビット $4(k+1)$ から $4(k+1)+3$ までを読み出し, 階層 k と同様に最下位の 0 の場所を得る. 階層 $k+1$ の巡回も階層 k と同様に行なう. 階層 $k+1$ の巡回履歴のビットが全て反転すると, 最下位の 0 の位置を得られなくなる. これはこの階層の巡回を終えたということであるので, 階層 k に戻る. このときに階層 $k+1$ の巡回履歴のビットを全て反転させ 0 に戻しておく. 階層 k の巡回履歴は以前更新したものから最下位の 0 の位置を求め 2 を得る. つまり 2 番目のノードとの交差判定, そしてそのビットの反転を行なう. この 2 番目のノードと交差している場合は, 再度木を下り階層 $k+1$ に進む. 階層 $k+1$ の巡回履歴は以前巡回を終えたときに初期化してあるので最初のノードから巡回を開始することができる. このように各階層に 4 ビットのフラグを確保するだけで, スタックを用いずに四分木を巡回することができる.

7.9.2 実装

本手法ではそれぞれのオブジェクトごとに必要なリソースは巡回履歴用のビット配列のみであり, 8 階層の八分木ならば各オブジェクトごとに 64

ビットだけ確保すればよい。このように巡回に必要なデータ量が非常に小さく、さらに巡回中に大きさが変化しないため、比較的高速なシェアードメモリにこのデータを置くことができる。また本手法は一般的なスタックを用いた巡回のプログラムに、ある階層の最下位の0の位置を求める関数、ある階層のある位置のビットを反転させる関数、そしてある階層のビットを初期化する関数の3関数を追加するだけで実装することができる。

葉ノードとの交差判定

一般的に木構造を用いた交差判定では、葉ノードに到達するとその葉ノードに格納されている要素全てと交差判定を行なう。しかしGPU上で の実装ではこのように葉ノードに到達したスレッドでその中の要素全てと 交差判定を行なってしまうと、並列計算単位の他のスレッドも葉ノードに 到達しているとは限らないため、負荷バランスが悪くなり、パフォーマンス の低下につながる。そこで本研究では葉ノードに到達した後でもそこで その中の要素と交差判定を行なわずに、交差している葉ノードを出力する だけにとどめた。こうすることでスレッドの負荷バランスの悪化を防いだ。 そして全てのノードを巡回した後に、全ての要素において交差が報告された 全てのノード内に含まれている要素と交差判定を行なう。またGPUは 出力先のメモリの大きさを動的に変化させることができないので、あらかじめ 報告の上限を定めてメモリを確保し、その中に報告されたものを書き出すとした。

木構造の構築

本研究ではGPUを用いた木構造の巡回のみに焦点を絞った。動的な木構造の構築に関しても研究が行なわれており、完全な木ならば木の構築の処理をモートンコードを用いたソートの処理に置き換えることができ、 比較的GPUに向いた処理とすることができるなどの研究が行なわれている。 本手法とこの手法を組み合わせることによって完全にGPU上で動的に木構造を構築し、木構造を巡回することができるようになる。これらをあわせた評価は今後の課題である。

7.9.3 結果

本手法を実装し、Intel Core 2 Duo E8500, 3.16GHzのCPUと4.0GBのメインメモリ、GeForce 9800GX2を搭載したPCにて計算速度を検証した。

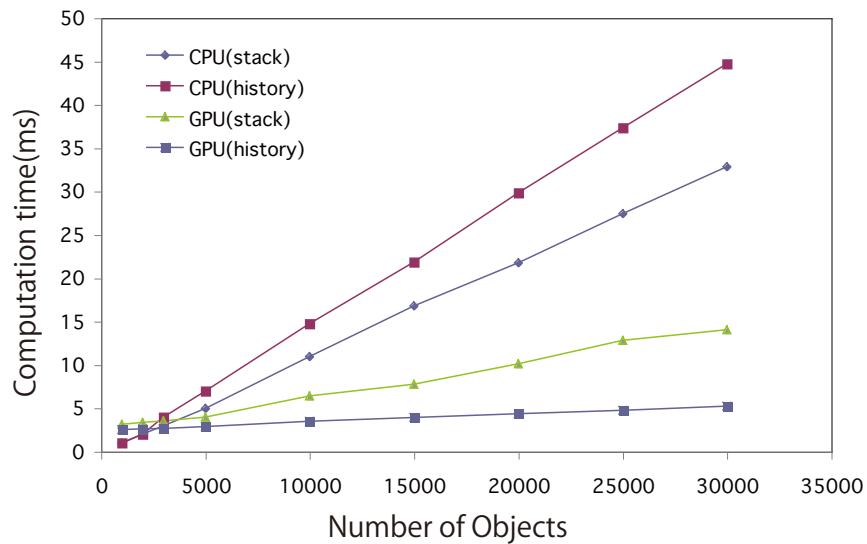


図 7.36: 要素数と木構造の巡回の計算時間

表 7.10: 計算時間 (ミリ秒). なお Time(a),(b),(c),(d) はそれぞれスタックあり (CPU), 巡回履歴 (CPU), スタックあり (GPU), 巡回履歴 (GPU) を用いた結果.

Number of objects	Time(a)	Time(b)	Time(c)	Time(d)
1,000	0.97	1	3.13	2.53
2,000	2.00	2.00	3.35	2.63
3,000	2.97	3.97	3.55	2.67
5,000	4.97	6.97	3.97	2.86
10,000	10.93	14.76	6.41	3.50
15,000	16.80	21.86	7.75	3.93
20,000	21.79	29.83	10.12	4.38
25,000	27.45	37.34	12.84	4.78
30,000	32.86	44.72	14.03	5.25

固定領域内に乱数を用いて発生させた直方体を、さらに乱数を用いて動かして、ベンチマークソフトウェアを開発した。ソフトウェアは C++ と GPU のプログラムは CUDA を用いて開発した。図にアプリケーションのスクリーンショットを示す。このベンチマークにて 1,000 から 30,000 個のまで物体の数を変化させて計算時間を測定した。それぞれの物体数において一定時間計算を続け、計算時間の平均を取った結果を図 7.36 と表 7.10 に示す。まずスタックを用いた CPU の実装とスタックを用いた GPU の実装を比較すると、物体数が 3,000 以下の場合には GPU での計算速度は CPU での計算速度よりも遅い。しかし計算要素数が増加するに従い GPU のほうが高速に計算できていることがわかる。次に GPU 上においてスタックを用いた計算と、巡回履歴を用いた計算結果を比較すると、全ての場合において巡回履歴を用いた計算のほうが高速であることがわかる。特に要素数が 10,000 以上では 2 倍以上の速度で計算できている。また GPU 上で巡回履歴を用いた計算は CPU 上でスタックを用いた計算と比較すると、要素数 30,000 において約 6 倍の速度で計算できていることがわかる。また CPU 上で巡回履歴を用いた手法を実装し、スタックを用いた手法と計算速度を比較したが、スタックを用いた手法の方が高速であった。これは巡回履歴を用いるためには、スタックを用いる場合よりも多くの演算が必要になるためであり、用いた PC の構成ではこのような結果が得られた。しかし、CPU の計算でも明示的に巡回履歴を GPU でのシェアードメモリのような高速なメモリに配置することが可能であれば、巡回履歴を用いた手法の方が高速になる可能性がある。

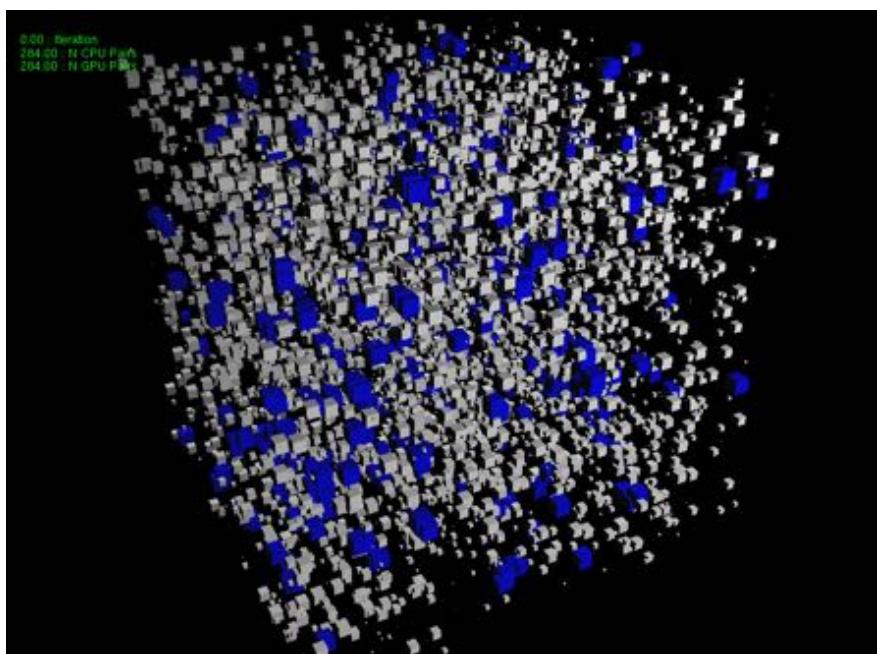


図 7.37: 5,000 要素を用いた衝突検出

7.10 結論

本研究ではユニフォームグリッドとスライスグリッドを並列に構築可能なアルゴリズムを開発し、それらをGPU上で実装と評価を行なった。これらの手法は粒子法シミュレーションにおける近傍粒子探索を効率化するために必要であるため、この技術は粒子法シミュレーション一般に有用である。そこで本手法を用いてDEMによる粉体シミュレーション、SPHによる流体シミュレーション、粒子を用いた剛体シミュレーション、さらにこれらの連成計算をGPUを用いて高速化し、CPUでの計算速度と比較することによって本手法の有用性を示した。さらに布と流体の連成計算においては2種類の格子を用いることで粒子だけでなくポリゴンも存在するシーンで高速に衝突検出ができる事を示した。また粒子法を用いた弾性体シミュレーションは、近傍粒子探索を行なわないが、これについてもGPU上で実装し、評価を行なった。その結果として現在のGPUの欠点であるレジスタの大きさが明確になった。

本手法は一般化すると計算領域に存在している要素の大きさがほぼ同じ場合の衝突判定問題の解決手法であるため、粒子法シミュレーションだけでなく、他の問題にも有効である。

本手法は、均一の大きさの物体のみが存在するときには有効であるが、それ以外の場合には効率的に衝突検出を行なうことができない。そのようなシミュレーションへの1つの解法として本研究では、GPU上で効率的に実行することができる巡回履歴を用いた木構造の巡回手法を開発し、GPU上においては一般的なスタックを用いる巡回法よりも高速に巡回することができることを示した。この手法は粒子径の異なる粒子法シミュレーション等に応用することができ、それらへの応用は今後の課題である。またもう1つの問題として、物体間の相互作用は全てペナルティ法のように解くことができるわけではない。ここでペナルティ法とは相互作用を計算する入力に出力データが含まれないということである。その例としては撃力を計算して相互作用を求める剛体シミュレーション等であり、ここでは相互作用の計算に各要素の速度を入力とし、出力も速度である。このような問題の解法として制約条件を複数のバッチに分解する手法を開発した。

第8章 複数のGPUを用いた粒子法 の高速化

8.1 序論

粒子法シミュレーションはコンピュータグラフィックスの分野で重要な技術の1つであり、1枚の画像を作るのに長い時間をかける映像制作からリアルタイムアプリケーションまで幅広く用いられている[132]。これらの分野ではシミュレーションが高速なほどよい。特にリアルタイムアプリケーションでは計算の速度が実時間程度である必要があり、シミュレーションの速度への要求が厳しい。そこでシミュレーションを高速化するために、マルチコアのストリームプロセッサの1つであるGraphics Processing Unit (GPU)を用いた研究も行なわれてきた[100]。代表的なストリームプロセッサとしてはGPU, Cell Broadband Engineが存在するが、シーケンシャルに処理を行なうCPUも近年ではマルチコアになってきており、今後はこのようなマルチコアプロセッサの性能を引き出すことができるアルゴリズムの重要性が高まっていく。しかし効率を上げるためにコア数を増加させるのも限界があると考えられる。そうなると複数のストリームプロセッサをさらに並列に配置して、さらに計算効率を上げる必要性が出てくると考えられる。このような計算環境ではストリームプロセッサで実行できるアルゴリズムを開発するだけでなく、それらが並列で計算するため、もう一段階の計算の並列化を考えなければならない。

本研究では粒子法シミュレーションに対して複数のGPUを用いて2段階の並列化を行なうことのできる手法を開発する。このような2段階の並列化を考えるときには2種類の計算モデルが考えられる。1つはサーバクライアント型の計算モデルであり、サーバとなる全体の計算を管理するプロセッサを置き、そのプロセッサが他のプロセッサに計算を割り当て、その結果を回収する。しかしこの計算モデルではサーバとなるプロセッサの処理は並列化されていないため、クライアントのプロセッサを増やしても効率化はできない。そのため、サーバとなるプロセッサの処理がボトルネックとなりうる。またサーバはクライアントに計算データを渡し、計算結果を受け取るため、多量のデータ転送が必要になり、並列化のオーバーヘッドが大きくなる。そこで本研究ではサーバとなるプロセッサを置かず、全ての計算プロセッサがそれぞれのデータを管理するピアツーピア型の計算モデルを提案する。この計算モデルでは全体のデータを全て管理するプロセッサを置かず、それぞれのプロセッサが計算データを分散して管理する。計算はそれぞれのプロセッサが持つデータのみでは行なえないため、他のプロセッサが持つデータを受け取る必要がある。この転送するデータの管理もそれぞれのプロセッサが並列で処理を行なうため、計算全体ではサーバクライアント型の計算モデルのように並列化できない部分がなく、並列化のオーバーヘッドを減らすことができる。

粒子法シミュレーションを複数のプロセッサ上で並列化するときには、

もう1つ考慮しなくてはならないものがある。計算要素間の接続が固定された格子のシミュレーションでは、計算領域を分割し1つのプロセッサが1つの計算領域の中にある要素の計算を行なう。そして隣接する領域を計算しているプロセッサから自分の計算要素に隣接している要素の物理量を受け取り、自分の計算要素の値を求める。計算要素間の接続が固定されているため、計算領域の分割は計算の前に1度行なっておけばよく、データを転送する必要がある要素も固定されているため、このリストも計算の前に作成することができる。しかし粒子法シミュレーションではそのようにすることができない。粒子法では粒子が自由に動くため、計算領域を分割してもその中に存在する粒子は固定されていない。すなわち、計算領域を分割してプロセッサに割り当てるときには、毎タイムステップにおいて計算領域内に存在する粒子を求める必要がある。さらに計算要素間、つまり粒子間の接続が固定されておらず、毎タイムステップごとに接続情報を求めなければならないため、計算を領域で分割したとしても、隣接する領域を計算しているプロセッサに転送するデータも動的に変化する。このように粒子法シミュレーションを複数のプロセッサ上で並列化するときには、並列化のオーバーヘッドが大きくなってしまう可能性があり、これらをどのように処理するかが重要になってくる。本研究では粒子法シミュレーションの計算を領域で分割し、隣接した領域を計算しているプロセッサに転送するデータの管理には、近傍粒子探索を効率化するために構築した格子データを再利用することによって並列化のオーバーヘッドを押さえる手法を提案し、GPU上でCompute Unified Device Architecture (CUDA)を用いた実装を行なう[99]。

また複数のプロセッサでリアルタイムシミュレーションを行い、レンダリングまで行なう場合は、シミュレーションの負荷は複数のプロセッサで分散するのに対し、レンダリングを行なうGPUは全てのプロセッサからのデータを受け取り、処理を行なわなければならない。そこでレンダリングを行なうGPUの負荷を軽減するため、シミュレーションを行なっている複数のGPUでデータの縮小を行い、レンダリングの負荷を分散させると同時に、システム全体のデータ転送量を減少させる手法も提案する。

8.2 関連研究

粒子法では計算要素が固定の接続情報を持たない粒子であり、粉体の計算手法である Distinct Element Method (DEM) や[26]、流体の計算手法である Smoothed Particle Hydrodynamics (SPH)[83] や Moving Particle Semi-implicit (MPS) Method が研究されてきた[76, 130, 131]。しかしこの固定の接続情報を持たないという点は粒子法の利点でもあり、欠点でも

ある。つまり毎タイムステップにおいて近傍粒子粒子を探索し、接続情報を計算しなければならないため、計算コストが高くなってしまう。そこでGPUをストリームプロセッサとして用い、計算を高速化する研究も行なわれてきた。DEM, SPH, 粒子法を用いた剛体のシミュレーション、流体剛体連成まで様々な粒子法を並列化し、GPUで計算する手法が開発され、シミュレーションの高速化が行なわれてきた[58, 55, 46]。

複数のプロセッサを用いて計算を並列化する研究も行なわれてきた。序論で議論したように計算要素間の接続関係が固定されている問題は比較的容易に複数のプロセッサを用いて並列化することが可能である。例えばThomaszewskiらによる布の計算やRibeiroらによる有限要素法の計算があげられる[121, 106]。また全ての計算を陽解法で行なうLattice Boltzmann Method (LBM)は特に複数のプロセッサを用いた並列化に向いており、複数のGPUを用いた研究も行なわれている[34]。

しかし粒子法のシミュレーションでは計算要素間の接続関係が固定されていないため、格子を用いるシミュレーションの並列化手法をそのまま適用することはできない。粒子法を複数のプロセッサを用いて並列化した研究としてはGRAvity PipE (GRAPE)という専用ハードウェアを用いた研究があるが、この研究ではSPHの全ての計算を行なうことができないため、CPUがサーバとなりGRAPE上の計算を管理しているため、これらの間での多量のデータ転送が必要である[98]。ここで用いられている計算モデルがサーバクライアントモデルである。この研究とは異なり、より効率の良い計算モデルを採用している研究もある。例えば粒子法の1つであるMacro-scale Pseudo-particle Modeling(MaPPM)を並列化した研究がある[126]。また近年では、StratfordらはLBMによる流体のシミュレーションを並列化するだけではなく、さらに粒子の計算も並列化した[113]。しかし、これらの研究では複数のプロセッサがいくつかの粒子のデータを重複して保持するため、数回のデータ転送が1タイムステップの計算において必要であるという欠点があった。またこれらの研究では粒子の物理量を更新した後にデータをプロセッサ間で交換するが、この処理をどのように行なうかということについては考察がない。これは1タイムステップの計算時間が数秒以上と長かったため、データの選別にかかる時間が全体の計算時間に占める割合が小さいため、この処理はほとんど問題にならなかつたからだと思われる。MPS法をPCクラスタを用いて並列化し、高速化した研究も存在する。入部らはMPSの連立方程式の計算を並列化して高速化を行なった[139]。しかし連立方程式の計算以外の部分が並列化されていないため、それ以外の部分の計算負荷の割合が高い手法に適用した場合には並列化効率が上がらない。よってこの並列化はMPSのように連立方程式を解く必要のある手法には有効であるが、DEMやSPHなど

連立方程式を解く必要のない手法には適用することができない。

このように様々な研究者が複数のプロセッサを用いた並列化を研究してきたが、リアルタイムアプリケーションに複数のプロセッサを用いた並列化を適用した例は我々の知る限り存在しない。しかし我々の研究目的はリアルタイムアプリケーションに複数のプロセッサを用いて並列化することであるため、既存研究では大きな問題にならないようなことも問題になってくる。例えば粒子法のシミュレーションでは転送する必要のあるデータの選別などである。

本研究の新規性は具体的には以下のようにまとめられる。粒子法のシミュレーションを並列化する際に、計算粒子を重複せずに管理する手法を開発した。これは1粒子が複数のプロセッサで計算されることがないということである。これによって計算粒子を重複して管理していた場合に比べてデータ転送回数が減り、並列化の効率を良くすることが可能になる。そしてこの計算に必要なゴースト領域の導入方法も既存の粒子法の並列化を行なった論文とは異なる。また本論文では、近傍粒子探索を効率化するために計算した格子のデータを再利用するプロセッサ間のデータ転送方法を提案する。データ転送方法については前述のように既存研究では焦点が当てられてこなかった。本研究では複数のGPUを用いてリアルタイムアプリケーションを並列化する手法を開発したが、リアルタイムアプリケーションで複数のプロセッサを用いて並列化を行なった研究はなかった。上記の本研究の新規性は1タイムステップの計算時間が極めて短いリアルタイムアプリケーションを並列化する際に直面する問題点の解決手法であるとまとめられる。

8.3 格子を用いた近傍粒子探索の効率化

粒子法では近傍粒子探索を毎タイムステップ行なう必要があり、これを効率化するために、格子を導入することができる。本手法は粒子法シミュレーションにおいて格子が導入されていることを前提とするため、本章で格子を用いた近傍粒子探索の効率化について簡潔に述べる。なお詳細は参考文献に譲る[132]。

ある粒子の物理量の計算にはその粒子の近傍に存在する粒子の物理量が必要になる。近傍粒子を全粒子から探索したのでは粒子数を n とすると $O(n^2)$ の計算になり、粒子数の増加に伴い計算コストが大きくなってしまう。そこで計算領域を覆う格子を用意し、その格子のそれぞれのボクセル内の領域に存在している粒子番号を格納する。この粒子番号が格納された格子を用いることによってある粒子の近傍粒子はその粒子が存在している格子の周囲の格子に粒子番号が格納されている粒子に限定され、計算コス

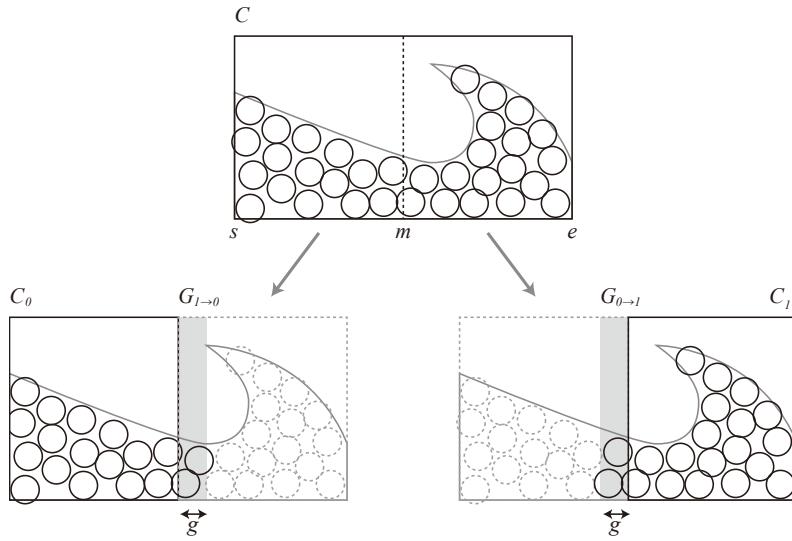


図 8.1: 2 個のプロセッサを用いたときの計算の分割.

トは $O(n)$ に改善される。ボクセルの大きさの選択は任意であるが、本研究で用いる DEM の場合は衝突粒子を探索するため、ボクセルの 1 辺の大きさを粒子の直径 l_0 とすることで、衝突粒子はその粒子が格納されているボクセルの周囲の 3^3 個のボクセル内に存在する粒子に限定されるようになり、最も効率的である。

8.4 計算の分割

複数のプロセッサで 1 個のシミュレーションを行なう時には、計算を分割しなければならない。計算の分割の方法としてはいくつかの選択肢があるが、本研究では計算領域を分割し、それぞれのプロセッサが割り当てられた領域内に存在する粒子の物理量を計算するようにし、割り当てられた領域内に存在する粒子の物理量のみを更新する。

粒子法ではある粒子の物理量を計算するときに近傍粒子の物理量が必要になる。DEM では近傍粒子と衝突計算を行い、SPH では近傍粒子の持つ物理量を積分する。しかしあるプロセッサが計算している領域の境界付近に位置する粒子の近傍粒子は、そのプロセッサの計算領域の外に存在し、隣のプロセッサの計算領域に存在していることがある。このような場合には隣接するプロセッサのメモリに存在するその近傍粒子のデータにアクセスしなければならない。この処理を境界に位置する全ての粒子において必要に応じて行なうのは小さいデータを複数回転送しなければならず非効率

的であるため、本研究ではゴースト領域を導入する。ここで計算領域が

$$C = \{x | s < x \leq e\} \quad (8.1)$$

であり、2つのプロセッサ p0, p1 が1つの計算を行なっているとする。計算領域を X 軸に垂直な平面で分割して、それぞれの計算領域を

$$C_0 = \{x | s < x \leq m\} \quad (8.2)$$

$$C_1 = \{x | m < x \leq e\} \quad (8.3)$$

とする。ここで $m = (s + e)/2$ であり計算領域の中点の X 座標である。すると p0 のゴースト領域 $G_{1 \rightarrow 0}$ は C_0 に隣接した幅 g の C_1 内の領域であり、

$$G_{1 \rightarrow 0} = \{x | m < x \leq m + g\} \quad (8.4)$$

とし、また p1 のゴースト領域 $G_{0 \rightarrow 1}$ は同様に C_1 に隣接した C_0 の領域であり

$$G_{0 \rightarrow 1} = \{x | m - g < x \leq m\} \quad (8.5)$$

と定義する。n 個のプロセッサを用いる場合は、計算領域を n 分割し、0 番目と n-1 番目のプロセッサ以外は 2 個のプロセッサの計算領域に隣接しているため、2 個のゴースト領域を持つようになる。粒子の影響半径を $r_e = g$ とすると、プロセッサ p0 が計算している粒子の近傍粒子は $C_0 \cup G_{1 \rightarrow 0}$ の領域内に存在していることになり、計算の前にゴースト領域の粒子データを受け取れば近傍粒子探索の時に隣接するプロセッサのデータを参照しなくても良くなる。このゴースト領域内の粒子をゴースト粒子と呼ぶ。p0 はこのゴースト粒子の物理量は更新せず、 C_0 の領域内に存在する粒子の物理量のみ更新する。いずれの粒子も $C_0 \cup C_1$ に存在しているため、全てのプロセッサが、割り当てられた領域内の全粒子の物理量を更新すれば、全ての粒子の物理量が更新される。すなわち $G_{1 \rightarrow 0} \subset C_1$ であり、 $G_{0 \rightarrow 1} \subset C_0$ であるため、各プロセッサはゴースト粒子の物理量を更新しなくとも問題はない。図 8.1 に 2 個のプロセッサを用いた場合のこれらの領域の関係を示す。

8.5 データの管理

粒子法では格子法とは異なり計算粒子は自由に動くことができる所以、計算中に粒子が他のプロセッサの計算領域に出て行くこともあり、また逆に他のプロセッサの領域から入ってくることもある。粒子が動くということはゴースト粒子も各タイムステップにおいて変化することである。よって計算のデータを管理する必要性が生じる。最も容易なデータの

管理方法は、サーバ、クライアントモデル的な管理方法であり、サーバとなるプロセッサを用意して、そのプロセッサが全粒子の物理量を保持する。そして各タイムステップごとにクライアントに必要な情報を送る方法である。しかしこの方法では毎タイムステップにおいてサーバは全てのクライアントから全粒子のデータを受け取り、さらに全クライアントに送るデータを計算しなければならなく、毎タイムステップにおいて多量のデータ転送を行なう必要がある。さらに処理をサーバが行なっている時にはクライアントは処理を行なうことができないため、計算のボトルネックとなりうる。そこで本研究ではサーバを置かずに、各プロセッサがデータを管理するようにした。本節ではデータ管理方法を述べる。

8.5.1 重複のない計算粒子の管理法

簡単なデータ管理方法は全てのプロセッサが全ての粒子データを持つという方法である。そしてそれぞれのプロセッサに割り当てられた領域内に存在する粒子の物理量の更新を行なう。しかしこの手法では全てのGPUが全ての粒子データのメモリを確保しなければならなく、メモリ効率が悪い。そこで本研究ではそれぞれのGPUがそれぞれの計算領域内に存在する粒子のデータのみをメモリに持ち、計算を行なう。

データの送信

前述のようにあるプロセッサの計算領域の境界付近の計算を行なうためには隣接するプロセッサの持つデータが必要である。また粒子が計算領域にとらわれず自由に動くため、あるプロセッサの計算領域から出た粒子のデータを隣接するプロセッサに渡す必要がある。よって1タイムステップごとにそのプロセッサの計算領域から出て行く粒子と隣接したプロセッサの計算に必要なゴースト粒子を送る必要がある。ある時刻 t で p_0 が計算している粒子 i の座標を x_i^t として、そのタイムステップで計算した後の時刻 $t + \Delta t$ での粒子の座標を $x_i^{t+\Delta t}$ とする。粒子 i は p_0 の計算領域内に存在するため、 $x_i^t \in C_0$ である。まずこの計算領域 C_0 から C_1 に出て行く粒子は

$$EP_{0 \rightarrow 1}^{t+\Delta t} = \{i | m < x_i^{t+\Delta t}, x_i^t \leq m\} \quad (8.6)$$

の粒子である。また p_0 の計算領域内に存在する p_1 に必要なゴースト粒子は

$$GP_{0 \rightarrow 1}^{t+\Delta t} = \{i | m - g < x_i^{t+\Delta t} \leq m\} \quad (8.7)$$

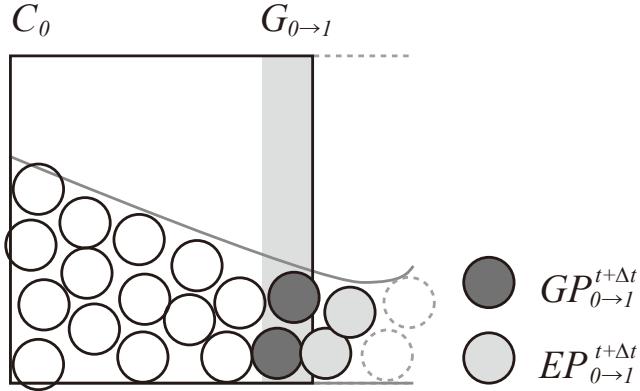


図 8.2: プロセッサ 1 からプロセッサ 2 へ送る粒子.

に存在する粒子である。よって隣の p_1 に送らなければならない粒子は、式 (8.6) と式 (8.7) より、

$$\begin{aligned} SP_{0 \rightarrow 1}^{t+\Delta t} &= EP_{0 \rightarrow 1}^{t+\Delta t} + GP_{0 \rightarrow 1}^{t+\Delta t} \\ &= \{i | x_i^{t+\Delta t} > m - g, x_i^t \leq m\} \end{aligned} \quad (8.8)$$

である(図 8.2)。

データの送信方法にもいくつかの選択肢がある。1つ目の方法は全てのデータを送り、送り元のプロセッサの領域に存在した粒子のデータを、受け取った先のデータに上書きすることである。しかし実際は隣接したプロセッサの持つ全粒子のデータは必要ではなく、計算領域に入ってくる粒子とゴースト粒子の物理量のみ受け取り更新すればよい。また一般的に送る必要のあるデータ量が全粒子の物理量の中に占める割合は少ないため、この送信方法は非効率的である。最も効率が良いのは $SP_{0 \rightarrow 1}^{t+\Delta t}$ の粒子の集合のみ送ることであるが、それには全粒子からこの条件に合致した粒子のみを選択しなければならない。そのためには全粒子の中でその条件にあった粒子にフラグをたて、そのフラグの全粒子の中での番号を計算して、データを送るバッファに書き込まなければならない。このように送るデータを選択するには Prefix sum を用いることができるが、CUDA を用いた実装ではまず有効な粒子にフラグを立てるカーネル、Prefix sum を求めるカーネル、有効なデータを Prefix sum で計算したアドレスに移動させるカーネルが必要であり、計算量が増加する [60]。Prefix sum とは図 6.5 に示すように入力とする配列 $a_0, a_1, a_2, \dots, a_{n-1}$ に対して、 $b_i = \sum_{j < i} a_j$ の配列 b を求めることである。

そこで近傍粒子探索を効率化するために構築した格子を用いて他のプロセッサに送るデータを選択する。格子のボクセルに格納された粒子番号を

参照することで送信する必要のある粒子を探すことができる。つまりボクセルのX座標 x_v が $x_v > m - g$ のボクセルに格納されている粒子が隣のプロセッサに送る粒子である。しかし格子に粒子番号を格納するために用いた粒子座標は、タイムステップで更新された $x_i^{t+\Delta t}$ の座標ではなく、更新される前の x_i^t であるため、 $x_v > m$ のボクセルを参照したのでは $SP_{0 \rightarrow 1}^{t+\Delta t}$ の粒子の集合を見つけることはできない。ここで再度格子に粒子番号を格納することもできるが、やはり計算量が増加する。そこで本研究では時刻 t で構築した格子を再利用する。粒子法のシミュレーションでは一般的に任意の大きさのタイムステップを刻めるわけではない。計算安定条件として1タイムステップで進む粒子の距離が、粒子の直径以下であるというクーラン条件を課す。つまり粒子の速度、時間刻み幅、粒子の直径をそれぞれ $v, \Delta t, l_0$ とすると $v\Delta t/l_0 < c$ と表せる。ここで c はクーラン数と呼ばれる。計算においてクーラン数に $c < 1$ の条件を課すということは、前述のように粒子の1タイムステップでの移動は初期粒子間距離以下に制限されるということである。また近傍粒子探索を効率化する格子のボクセルの大きさを粒子の直径と同じにしているため、粒子は1タイムステップにおいて1ボクセル以上移動することはない。よって $x_i^{t+\Delta t} > m - g$ の粒子は $x_i^t > m - g - l_0$ の条件を満たす。すなわち隣接したプロセッサに送る粒子は $x_v > m - g - l_0$ の条件を満たすボクセルに格納された粒子であると言える。また粒子の集合 $EP_{0 \rightarrow 1}^{t+\Delta t}$ は、時刻 t では p_0 によって計算されるため、 $x \leq m$ である。同様にクーラン数の条件から粒子の集合 $EP_{0 \rightarrow 1}^{t+\Delta t}$ は $m - l_0 < x^t \leq m$ の条件を満たす。すなわち p_0 から p_1 に送る粒子 $SP_{0 \rightarrow 1}^{t+\Delta t}$ は時刻 t において図8.3に示すように、

$$S'_{0 \rightarrow 1} = \{m - d - l_0 < x \leq m\} \quad (8.9)$$

に存在する粒子であり、 $d = r_e$ のとき領域 S' は、

$$S'_{0 \rightarrow 1} = \{m - 2d < x \leq m\} \quad (8.10)$$

となる。

この選択された粒子の物理量を送るためにデータを格納するバッファを用意しなければならない。均一格子を用いた場合で格子のX軸での断面が $v_y \times v_z$ であるとき、 $g = r_e$ の時にはX軸方向に2ボクセル分のデータを送信する必要があり、1ボクセルに格納される最大粒子数が n であるときには $v_y \times v_z \times 2 \times n$ の粒子のデータを格納できるバッファを用意して、格子を参照して送るデータを作成する。

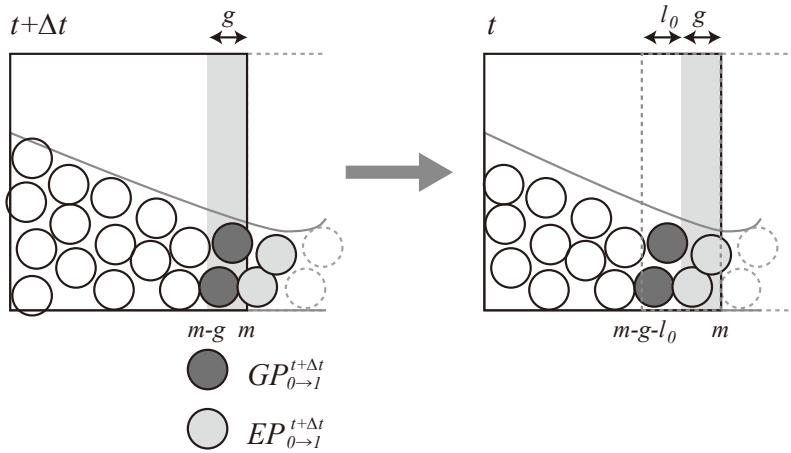


図 8.3: 送信する粒子データ. 灰色の領域が p_0 の計算領域内のゴースト領域であり, 図左において p_0 から p_1 に送信する粒子を 2 色で示している. これらの粒子の時刻 t での配置を図右に示す.

データの受信

データの受け取りは、全てのプロセッサが同じ粒子番号を用いている場合には、隣接したプロセッサが送ったデータを受け取り、そのデータ内の粒子番号の粒子の物理量を上書きすることによって、受け取った粒子の物理量を最新のものに更新すればよく容易である。しかし本手法では、そのようにすることはできない。これはそれぞれのプロセッサが独自の粒子番号を用いて粒子を管理しており、境界に存在するある粒子の番号は、その境界を挟む両方のプロセッサで、それぞれ異なった粒子番号として認識されているからである。そこで受け取ったデータをそのプロセッサの持つデータに追加する。しかしこのときに注意しなければならないのは粒子データの重複である。粒子データの重複を起こさないために、隣接したプロセッサに送るデータを作成する前に、前のタイムステップでゴースト領域に存在した粒子のデータを削除する。粒子を移動させたあとの座標がゴースト領域に存在する粒子を削除するのではないことに注意する必要がある。これは以下の 2 つの命題が真であることから証明することができる。

- ある時刻 t で $G_{1 \rightarrow 0}$ に存在する粒子の集合 $GP_{1 \rightarrow 0}^t$ は、次の時刻 $t + \Delta t$ で隣接した計算領域から送られてくる粒子の集合 $SP_{1 \rightarrow 0}^{t+\Delta t}$ に含まれる。
 - ある時刻 t で C_0 に存在し、次の時刻で $G_{1 \rightarrow 0}$ に存在する粒子の集合 $EP_{0 \rightarrow 1}^{t+\Delta t}$ は、次の時刻 $t + \Delta t$ で隣接した計算領域から送られてくる

粒子の集合 $SP_{1 \rightarrow 0}^{t+\Delta t}$ に含まれない。

まず1つ目の命題において、

$$SP_{1 \rightarrow 0}^{t+\Delta t} = \{i | m < x_i^t \leq m + d + l_0\} \quad (8.11)$$

の領域に存在する粒子であり、

$$GP_{1 \rightarrow 0}^t = \{i | m < x_i^t \leq m + d\} \quad (8.12)$$

である。よって式(8.11)と式(8.12)より、 $GP_{1 \rightarrow 0}^t \subset SP_{1 \rightarrow 0}^{t+\Delta t}$ である。これにより、時刻 t においてゴースト粒子であったものは、データを受け取る前に削除しなければならないことがわかる。

また2つ目の命題において、

$$EP_{0 \rightarrow 1}^{t+\Delta t} = \{i | m - d < x_i^t \leq m, m < x_i^{t+\Delta t}\} \quad (8.13)$$

であり、

$$SP_{1 \rightarrow 0}^{t+\Delta t} = \{i | m < x_i^t \leq m + d + l_0\} \quad (8.14)$$

なので、 $EP_{0 \rightarrow 1}^{t+\Delta t} \notin SP_{1 \rightarrow 0}^{t+\Delta t}$ である。よって時刻 $t + \Delta t$ においてゴースト領域に存在する粒子は削除してはいけないことがわかる。受け取る粒子とそのプロセッサの計算領域内の粒子は重複がないため、受け取ったデータをそのプロセッサの持つデータの後ろに付け加えるだけで良い。

格子を用いて送るデータを構築する場合は送るデータ配列に粒子が存在しないボクセルが存在する。また全てのボクセルがボクセルに格納できる最大粒子数の粒子番号を保持しているわけではないため、無効なデータが存在する。そのため、データの送受信がくり返されると無効なデータが蓄積してしまい、粒子データの配列が拡大し続けてしまう。これを防ぐために粒子のデータをソートする前に(このソートについては後述する)配列の圧縮を行い無効なデータを取り除く。具体的にはPrefix sumを用いて配列の圧縮を行なう。

アルゴリズム

このようにゴースト粒子を用いて計算領域の境界の計算を行い、格子を利用してデータの送受信をする。単一プロセッサでのアルゴリズムにゴースト粒子の削除とデータの送受信を追加することで複数のプロセッサで計算を行なうことができるようになる。GPU間では直接データの転送を行なうことができないため、各GPUは用意した境界のボクセルの粒子データをビデオメモリからメインメモリに送る。データの受信は隣接したプロ

セッサがデータの送信を完了した後に行なう必要があるので、全てのプロセッサで同期を取る。後述する計算結果は並列化にOpenMPを用いたため、全てのスレッドの同期はOpenMPの命令文を挿入することで行なつた。それからそれぞれが送られたデータをメインメモリからビデオメモリに読み込み、受け取った粒子の物理量を更新する。これが1タイムステップの計算の流れである。

8.6 格子の選択

8.5章で述べた方法でプロセッサ間のデータの転送を行なうには、近傍粒子探索を効率化するために格子を導入していることを前提にした。最も単純な格子は均一格子であるが、均一格子は計算領域全体のメモリを確保する必要があるため、メモリ効率が悪い。そこで本研究ではこの均一格子の問題点を改善するために5で提案したスライスグリッドを用いる。

8.6.1 スライスグリッド

スライスグリッドでは計算領域内の1軸を取り計算領域をこの1軸に直交するスライスに分割して、それぞれのスライスで軸平行バウンディングボックスを求め、その内部のメモリのみを確保する。これによってメモリ効率が向上し、さらに粒子番号が格納されているボクセルが密に格納されるようになるため、キャッシュ効率も上がり計算速度も向上する。さらに粒子番号と同様に粒子データも粒子の空間配置でソートすることでこれらのデータのキャッシュ効率も向上し計算の高速化が可能である。またプロセッサ間のデータの転送においても均一格子を用いるより、スライスグリッドを用いた方が転送データ量も少なくなる。本研究ではまずスライスグリッドを用いた実装をさらに高速化するために、ソートを導入した。本章ではまずこのソートの導入について述べ、最後にスライスグリッドを用いたときの複数プロセッサ間でのデータの転送について述べる。

8.6.2 スライスグリッドへのソートの導入

スライスグリッドを用いるとメモリ効率を上げることができるだけでなく、ボクセルのデータが密に格納されているため、キャッシュ効率も向上し、計算速度があがることがわかっている。粒子の物理量のメモリ配置が粒子の空間配置になっている場合は、近傍粒子のデータがメモリの近くの領域に連続して存在するようになるため、粒子の物理量へのアクセスにおいてもキャッシュ効率が向上する。しかし粒子法のシミュレーションでは

計算要素である粒子間に固定された接続がなく、自由に動き回ることができるため、計算が進行するにつれてメモリ内の粒子の物理量の配置がランダムになり、キャッシュ効率が下がり計算速度が下がる。よってスライスグリッドのみを用いた場合よりもシミュレーションを高速化する1つの選択肢は粒子の物理量を粒子の空間配置通りにソートすることである。しかしソートによって近傍粒子探索は効率化されて高速化しても、ソートのペナルティがそれ以上であつたらシミュレーション全体は高速化されない。

関連研究で述べたようにGPUでのソートとして以前から研究されてきたのはソーティングネットワークであるバイトニックマージソートや、奇偶マージソートであるが、これらでは多くのパスが必要であるという欠点があった。しかしCUDAを用いることによってパス数が少なく効率的であるラディックスソートも実装することができるようになったが、これを使って約30,000要素のソートでも約6msかかってしまい[41]、粒子法の中でも計算コストが低いDEMのシミュレーションの1タイムステップでの近傍粒子探索よりも長い時間が必要になってしまう。よって粒子法のリアルタイムシミュレーションの高速化に用いることはできない。

粒子法のシミュレーションによる近傍粒子探索と類似した問題として剛体シミュレーションのブロードフェイズの衝突判定計算が挙げられる。剛体シミュレーションにおいても剛体同士は固定された接続情報を持たないため、自由に動き回ることができる。ブロードフェイズの衝突計算を行なう手法の1つとしてsweep and pruneがある[10]。この手法でも剛体をソートするが、ここで用いられるのは挿入ソートである。挿入ソートはほぼ整列したリストのソートを効率的に行なうことができるため、フレーム間にコヒレンシのある剛体シミュレーションのブロードフェイズの衝突計算のソートに用いるのに適している。ここで我々の取り上げている問題に戻るが、粒子法シミュレーションもフレーム間にコヒレンシがあるため、挿入ソートのようなほぼ整列したリストに適したソートが有効である。しかし挿入ソートは完全にシーケンシャルなソートであるため、GPUで実装できない。そこでGPUなどのストリームプロセッサに適したほぼ整列したリストに有効なブロックトランジションソートを提案する。

8.6.3 ブロックトランジションソート

ブロックトランジションソートは奇偶トランジションソートの考え方を一般化したソートである。奇偶トランジションソートはではまず偶数番目の要素が上の要素と比較され、次に奇数番目の要素が上の要素と比較される。この2種類の操作をくり返すことによってソートが行なわれる。このソートはn個の要素が逆の順番に並んでいた時に最も効率が悪く、 $n/2$ 個の要素の比較を $O(n)$ のパス数だけ操作を行なわなければならないが、ほ

ば整列したリストに対しては数回の操作でソートが完了する。例えば2つの要素が逆転している場合は1回もしくは2回の操作でソートが完了する。ブロックトランジションソートはこの性質に着目したソートである。

ブロックトランジションソートでは1回の操作で2つの要素の比較を行なうのではなく、全体のリストを $m/2$ 個の要素からなるブロックに分解して、隣接した2個のブロックの m 個の要素をソートする。1回目の操作では偶数番目のブロックと上のブロックをあわせた m 個の要素がソートされ、2回目の操作では奇数番目のブロックとその上のブロックをあわせた m 個の要素がソートされる。よってブロックトランジションソートでは $m/2$ 個の範囲での要素の逆転は1回もしくは2回の操作でソートすることができる。このようにほぼ整列したリストにはブロックトランジションソートは有効なソートである。2個のブロックのソートにはどのソートを用いても良い。図8.4はブロックトランジションソートによるソートの例である。この図では7ステップのソートが行なわれ、図の左から右へ進むにつれてソートが進行している。

またこのように計算要素をブロックに分割してソートするのは CUDA などのシェアードメモリが存在するプロセッサのプログラミングモデルにも適している。CUDA ではブロック内のスレッドがデータを共有し、処理を行なうことができるからである。2個のソートのブロックのソートにカーネルの1ブロックを割り当て、その中でキーをシェアードメモリに置き、グローバルメモリに書き出さずにソートすることも可能である。ここでは各ブロックのソートにバイトニックマージソートを用いた。シェアードメモリを用いることのできない、シェーダでの実装では1回の要素の交換の後にデバイスマメモリに書き出さなければならぬが、シェアードメモリを用いることによってソートが終了したときだけデバイスマメモリに書き出せば良くなる。また奇偶マージソートも安定なソートであり、Kolb らが行なったように1回のソートをシミュレーションの数タイムステップの間に分けることが可能である[75]。しかし奇偶マージソートでは各ステップにおいてソートを行なうブロックの大きさが異なる。ソートを行なうブロックの大きさがカーネルの1ブロックで計算できる大きさよりも大きくなってしまうと2つのリストを結合する1ステップで何度もグローバルメモリにアクセスする必要がでてしまい、非効率的になる。

ブロックトランジションソートでも、カーネルの1ブロックで処理しきれない数の要素をソートする時にはグローバルメモリにソート結果を書き出す必要があるため、ここではグローバルメモリに書き出す必要のない要素数を用いた。具体的にはソートの1ブロックの最大要素数を256個にして、1スレッドが2個のキーをグローバルメモリから読み出すようにした。この場合512個の要素をソートするためにカーネルは1ブロック 256



図 8.4: ブロックトランジションソートの例.

個のスレッドを実行した。

スライスグリッドへのソートの導入

あるプロセッサの担当している計算領域から出て行った粒子は、それ以降はそのプロセッサで計算する必要がないため、フラグを立てて、物理量の配列の後方に移動される。こうすることによって計算に必要な粒子が小さい粒子番号に固まるようになり、物理的に同時に実行されるスレッドの処理の発散も防ぐことができるようになる。ただしソートすることで粒子番号は変わり、リナンバリングを行なっていることになる。

データの送信

既存研究ではY軸で空間をスライスに分割し、XZ平面のスライスを作成していたが、この軸は任意に取ることができる。計算領域をX軸方向に複数のプロセッサで分割するときは、X軸方向に2ボクセル分のデータを隣接したプロセッサに送る必要がある。そのため、スライスグリッドの分割軸をX軸に定め、YZ平面のスライスを作成すれば、送信する必要があるデータは連続した2枚のスライスのデータとなり、メモリ上で連続した領域のデータを送ればよく、容易に転送するデータを取り出すことができる。具体的にはそれらのスライスの先頭のボクセルの番号を読み出し、そのアドレスからそれらのスライスに存在するボクセルの数のデータを送る。X軸でスライスに分割するということはそれぞれのボクセル座標で同じX座標のもののバウンディングボックスを求め、その内部のみのボクセルをメモリに確保するため、均一格子を用いた場合より、転送データ量を比較すると効率的である。

8.7 レンダリング

複数の GPU で計算されたデータを 1 つの GPU がレンダリングを行なう。レンダリングにはそれぞれの GPU が計算している全粒子の座標をレンダリングを行なう GPU に送らなければならない。例えば 100 万粒子の計算を 4GPU で行なうと、それぞれの GPU は 25 万粒子のデータを送るだけで良いが、レンダリングを行なう GPU は 100 万粒子を受け取り、レンダリングしなければならぬ。レンダリングを行なう GPU の負荷が高くなる。本研究で計算しているような数十万から百万までの粒子数だと単にそれぞれの粒子を球として可視化するだけでも負荷が高い。もちろんこれらの計算粒子から流体の表面抽出を行い、流体を半透明にレンダリングすることも考えられるが、この計算負荷はさらに高く、現在研究されている手法を用いてもリアルタイムでは行なうことができない [69]。本研究では粒子を球としてレンダリングする場合のみを考える。このような場合には内部の粒子は表面の粒子に隠れているためレンダリングする必要はない。レンダリングの負荷を下げるためにレンダリングを行なう GPU が表面を抽出する方法も考えられるが、この GPU は 100 万粒子から表面抽出を行なわなければならぬ、1GPU へ高い負荷がかかる。そこでレンダリングを行なう GPU の負荷を減らすために、本研究ではシミュレーションを行なっている GPU がそれ自身不必要な粒子データを削除し、レンダリングに必要なデータのみを転送した。具体的にはシミュレーションを行なっている GPU がそれぞれ粒子の密度を計算し、その密度を用いて表面粒子を抽出する。そしてこの表面粒子のみを転送し、レンダリングを行なう GPU が表面だけを描画する。粒子の密度の計算は粒子に働く力を計算するときに用いた粒子番号が格納された格子を用いて計算する。これによってシミュレーションを行なう GPU の計算コストは増えるが、転送されるデータ量が減りレンダリングを行なう GPU の計算負荷が下がる。レンダリングを行なう GPU が表面抽出を行なう場合と比較すると、3 つの利点がある。1 つ目は表面抽出の計算の負荷が 1GPU に集中せず、複数の GPU に分散されることである。またレンダリングを行なう GPU が表面抽出を行なうには近傍粒子探索を効率化するために、格子に粒子番号を格納しなければならないが、シミュレーションを行なう GPU が表面抽出を行なうとシミュレーションに使ったデータ、つまり粒子番号が格納された格子を用いることができるので効率がよい。これが 2 つ目の利点である。そして最後にシステム全体を見たときに転送されるデータ量を大幅に削減できるため、データ転送のコストという観点で見たときにも効率的になっている。

表 8.1: 計算時間の比較(ミリ秒).

N of P	1GPU		2GPUs		4GPUs	
	Sim	Total	Sim	Total	Sim	Total
200K	17.57	17.57	10.56	11.31	6.22	7.06
400K	33.53	33.53	17.23	18.44	9.68	13.97
600K	53.82	53.82	24.98	27.09	13.89	18.31
800K	71.96	71.96	31.17	37.75	19.99	24.49
1M	93.76	93.76	44.45	46.59	23.19	30.69

8.8 ベンチマーク

以下の計算は全て C++, CUDA 1.1 を用いて実装し, Intel Core2 Q6600 の CPU, GeForce 8800GT, Tesla S870 の GPU を搭載した PC 上で実行した結果である。なお Tesla S870 には 4 つの C870 の GPU が搭載されているため, 本研究で用いた PC には 5GPU が搭載されていることになる。

GPU 間で直接データ転送を行なうことはできない。そのため本研究ではビデオメモリに格納されているデータを一度メインメモリに読み戻してから、別の GPU のビデオメモリに転送を行なう。そこでビデオメモリ、メインメモリ間のデータ転送速度のベンチマークを行なった。図 8.5 にそれぞれの GPU が逐次にデータ転送を行なった場合の結果と、全ての GPU に一斉に並列でデータ転送を行なった結果を示す。ここではメインメモリからビデオメモリへのデータ転送を測定した。この図から GPU が逐次にデータ転送を行なった場合は送るデータサイズが大きくなるほど転送速度が速くなっていることがわかる。さらに、どの GPU でも転送速度はほぼ同じであることがわかる。しかし、並列でデータ転送を行なった場合は挙動が異なる。1GPU が 1 つの PCIe のスロットに接続されている GeForce 8800GT は逐次にデータ転送を行なった場合に近い結果が得られた、それぞれ 2 つの GPU が 1 つのスロットに接続されている S870 ではそれぞれの GPU の転送速度が逐次にデータ転送を行なった場合に比べて大きく下回っている。しかし、この場合 1 個の PCIe のスロットでの総データ転送量は 2 つの GPU の転送量の和になるため、単位時間あたりの転送量は 1GPU の転送量よりも大きなものになっている。また GPU によって、データ転送速度にぶれがあることもわかる。

8.9 結果

粒子径を 1 として $256 \times 128 \times 128$ の計算領域内のシミュレーションを 1, 2, 4GPU で行った。4GPU での計算では計算領域は図 8.6 に示すように

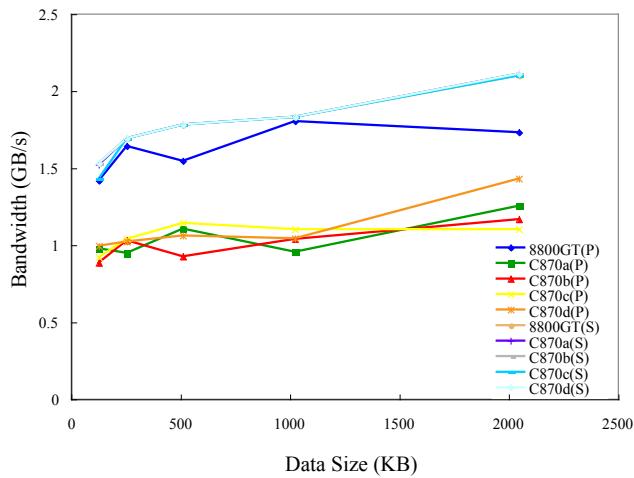


図 8.5: 5GPU を用いた並列でのデータ転送と逐次でのデータ転送の bandwidth の計測。

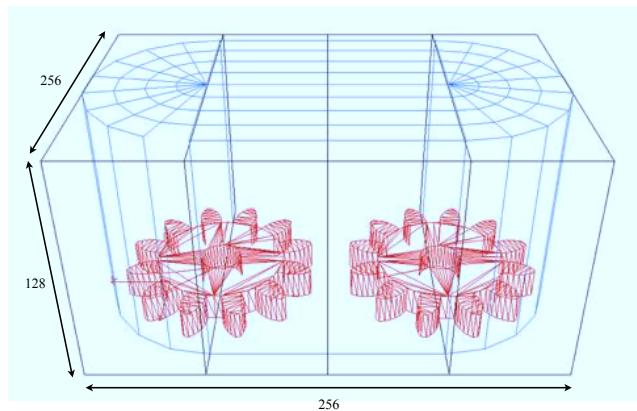


図 8.6: 計算体系。計算領域は 4 つの領域に分割され、1GPU は 1 つの領域の計算を行なう。

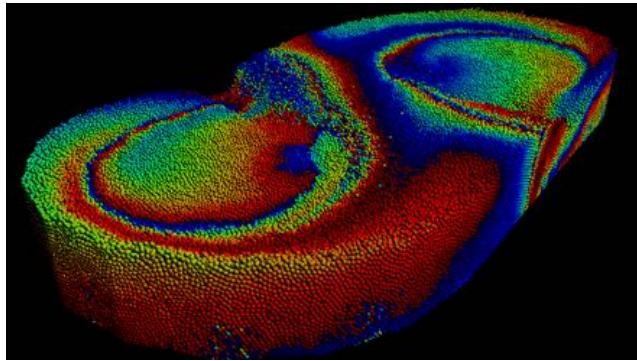


図 8.7: 500,000 粒子のシミュレーション

分割され、それぞれの GPU に計算を割り当てた。これらの計算体系において粒子数を変化させて計算時間を計測した結果を表 8.1 に示す。2GPU での計算ではそれぞれの GPU がその接続された領域のデータの転送を行なえばよいが、4GPU では両端の GPU は隣接した GPU に、それ以外の 2 つの GPU は両隣の 2 つの GPU にデータ転送を行なわなければならいため、同じ計算では転送量は 2 倍になる。この計算でも同じ粒子数だと 4GPU では 2GPU の転送時間のほぼ 2 倍になっている。またこのシミュレーションでは 2GPU を用いると 1 つの GPU が計算する計算粒子数は約半分になり、4GPU を用いると 1GPU が計算する計算粒子数は約 1/4 になる。この表からシミュレーションのみにかかる計算時間は 2GPU では 1GPU の時の計算時間の約半分、4GPU を用いると約 1/4 になっており、計算速度が GPU の数にスケールしていることがわかる。なおこの計算時間にはデータ転送の時間は含まれていないが、複数 GPU で計算するために行なわなければならない操作も含まれており、その計算時間が十分小さいことがわかる。しかし、複数 GPU で計算を行なう時には 1GPU での計算と異なり GPU でのデータの管理と転送を行なう必要がある。この転送時間も含めた 1 タイムステップにかかる計算時間では、4GPU を用いた場合は、粒子数が少なくシミュレーション時間が短い場合は転送がしめる割合が高くなり 1GPU を用いた場合の約 2.5 倍の計算速度に、粒子数が多くなると転送が占める割合が低くなり 1GPU を用いた場合の約 3 倍の計算速度になっている。

仮にサーバクライアント型の計算モデルを用いたとすると、毎タイムステップにおいてまずサーバとなる CPU がそれぞれの粒子を計算するプロセッサを計算し、それぞれのプロセッサに送るデータを用意する必要がある。そしてそのデータを送り、各 GPU に計算させた後にデータを全て回収し、CPU がメインメモリ上の粒子データを更新する。60 万粒子を

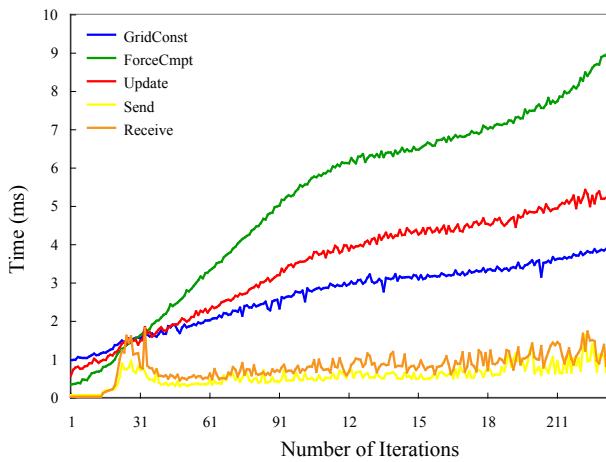


図 8.8: 2GPU を用いたときの計算時間.

用いた計算を考えると毎タイムステップにおいて CPU は全粒子の座標と速度などの粒子の物理量を GPU に送らなければならない。このデータ量は 16.8MB であり、図 8.5 のベンチマークの最大値である 1GB/s でデータ転送ができるとしてもこの転送には 16.8ms かかる計算になる。さらに計算後に同じデータ量を再度転送しなければならないため、サーバクライアント型の計算モデルを採用した場合にデータ転送のみにかかる時間は 33.6ms である（実際にはゴースト領域の粒子データも転送する必要があるのでより転送時間は増加する）。4GPU を用いたときに 1 タイムステップの計算にかかる時間が 13.89ms であるため、計算時間の 2 倍以上の時間をデータ転送のみに費やすことになってしまうため、非常に計算の効率が悪い。さらにこの計算モデルを採用した場合には CPU がデータを管理する時間もかかってしまう。この計算時間の推定から、本研究で行なったような 1 タイムステップの計算時間が非常に短い計算には本手法が適していると考えることができる。

また図 8.8 と図 8.9 にこの計算体系に粒子を流入させたシミュレーションの 2GPU と 4GPU での計算時間の変化を示す。この計算では総粒子数が 500,000 になるまで流入を続けた。2GPU と 4GPU での計算時間を比較すると、全ての計算において 4GPU での計算時間は約半分になっていることがわかる。さらに図 8.8 の 2GPU の結果を見るとデータの送信と受信にかかる時間に激しいぶれが生じていることがわかる。これはベンチマークでも確認されたように GPU と CPU でのデータ転送にかかる時間が安定していないことによるものであり、さらにこれらは 200 タイムステップ付近ではそれぞれ平均すると 1 ミリ秒程度であり、全体の計算時間に占め

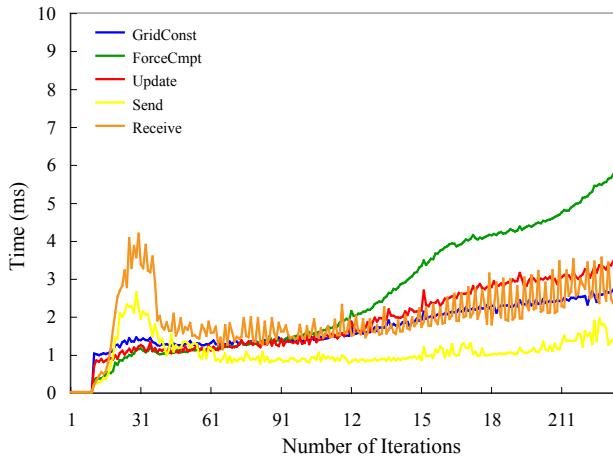


図 8.9: 4GPU を用いたときの計算時間.

る割合が十分に小さいことがわかる。それに比べ図 8.9 の 4GPU での結果を見るとデータ転送のぶれ幅がさらに大きくなっていることがわかる。これは前述のように 4GPU での計算の場合、計算領域の両端で接続されている GPU は両隣の GPU にデータ転送する必要があるため、2GPU のときの処理を 2 回行なう必要あるため、ぶれ幅も約 2 倍になっている。転送時間を見ても 2GPU のときの約 2 倍になっている。また図 8.10 にこの 4GPU を用いたときの計算において転送されるボクセルの数を示す。この図は 0 番目と 3 番目の GPU が 1 番目の GPU に送る量を示している。転送量がタイムステップによって変化しているのは、ライスグリッドを用いて動的に転送するボクセルを計算しているためである。この計算において均一格子を用いて、固定のボクセルを転送すると、32,768 ボクセルのデータを毎タイムステップで転送しなくてはならない。これと比べるとライスグリッドを用いることにより、転送量をその約 10 %程度に削減することができていることがわかる。

図 8.8 と図 8.9 のデータの送受信の時間はビデオメモリとメインメモリ間の物理的なデータの転送とそのデータの処理を行なう時間が含まれている。データを送信するためには粒子番号が格納されている格子を参照し、送信するバッファを用意する必要があり、データを受信したあとはそのデータを粒子データに追加しなければならない。そこで 2GPU の計算においてデータ処理の時間の転送処理全体に占める割合を示したもの図 8.11 に示す。この図からデータ送信の準備の割合は 20 %以下であり、データ受信後の処理の割合は約 10 %であることがわかる。すなわち図 8.8 や図 8.9 におけるデータ転送の時間の大半は物理的なデータの転送時間で

あり、PCIExpress の転送効率が向上すれば本手法を用いて複数 GPU で粒子法シミュレーションを行なった時には転送時間が十分小さくなり、計算時間は GPU の数にほぼ比例するようになる。

最後に表 8.2 に全粒子をレンダリングした場合と、表面の粒子のみをレンダリングした場合のフレームレートを示す。なおここでは粒子はポイントスプライトとして描画され、シェーダを用いてライティングの計算を行い球として描画した。表面だけをレンダリングすることで負荷を軽減することができていることがわかる。また図 8.12 に表面粒子のみレンダリングされた結果と、その断面を示す。レンダリングの質を落とすことなく、内部の粒子のみ削除されていることがわかる。

8.10 結論

本研究では複数の GPU を用いて粒子法シミュレーションの 2 段階の並列化を行なった。複数の GPU 上で並列化する際に、ピアツーピア型の計算モデルを用い、並列化のオーバーヘッドを減らした。また粒子法シミュレーションの計算領域を分割し、1 つの計算領域を 1 つのプロセッサに割り当て、各プロセッサがそれぞれの計算データを管理するモデルを開発した。またプロセッサ間のデータ転送に近傍粒子探索を効率化するために構築した格子を再利用することによって動的なデータ管理のオーバーヘッドを押さえ、計算を行なう GPU の数にほぼスケールした計算性能を出した。最後に複数の GPU でシミュレーションを行い、1 つの GPU でレンダリングを行なうシステムにおいてレンダリングの負荷をシミュレーションを行なっている GPU に分散させる手法も提案した。

しかし本手法の欠点としては計算領域を固定し、複数プロセッサ上の計算領域の分割があらかじめ決まっているということである。本手法で最も良い性能を出せるのは粒子が計算領域内に均等に分布しているときであり、粒子が 1 つのプロセッサの計算領域内に固まっている場合は並列化しても性能が向上しない。よって今後の課題としてはどのような粒子配置であっても、計算負荷を全てのプロセッサ上で均等にすることがあげられる。

本研究は 1PC に複数個の GPU が搭載されている環境で実装と評価を行なったが、本手法はこれ以外の計算機環境にも用いることができる。例えば、複数台の PC を接続した GPU クラスタにも適用可能である。また GPU を用いない計算環境でも有用である。例えばマルチコア CPU のみを用いた場合や、一般的な PC クラスタでの計算にも用いることができる。

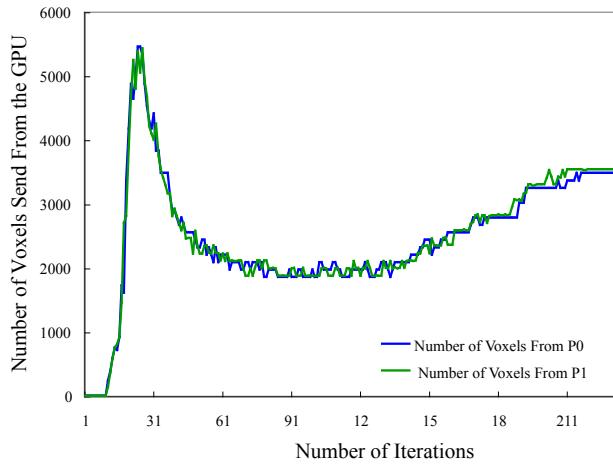


図 8.10: 転送されるボクセル数.

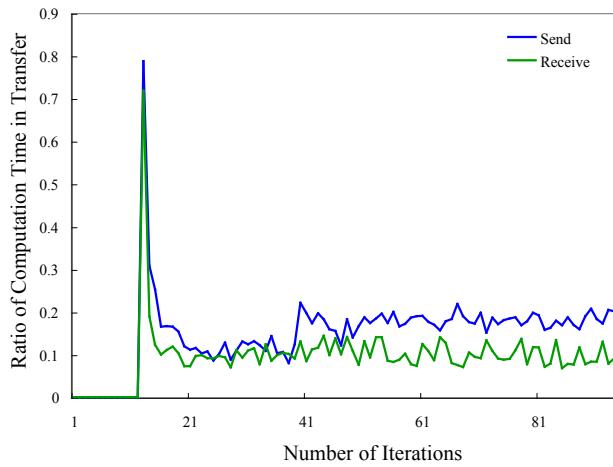


図 8.11: 転送時間におけるデータの処理計算の占める割合.

表 8.2: レンダリングのフレームレートの比較(フレーム/秒). Solidでは全ての粒子がレンダリングされており, surfaceでは表面の粒子のみレンダリングされている.

N of P	Solid	Surface
10K	151	181
20K	123	177
30K	99	164
40K	80	160

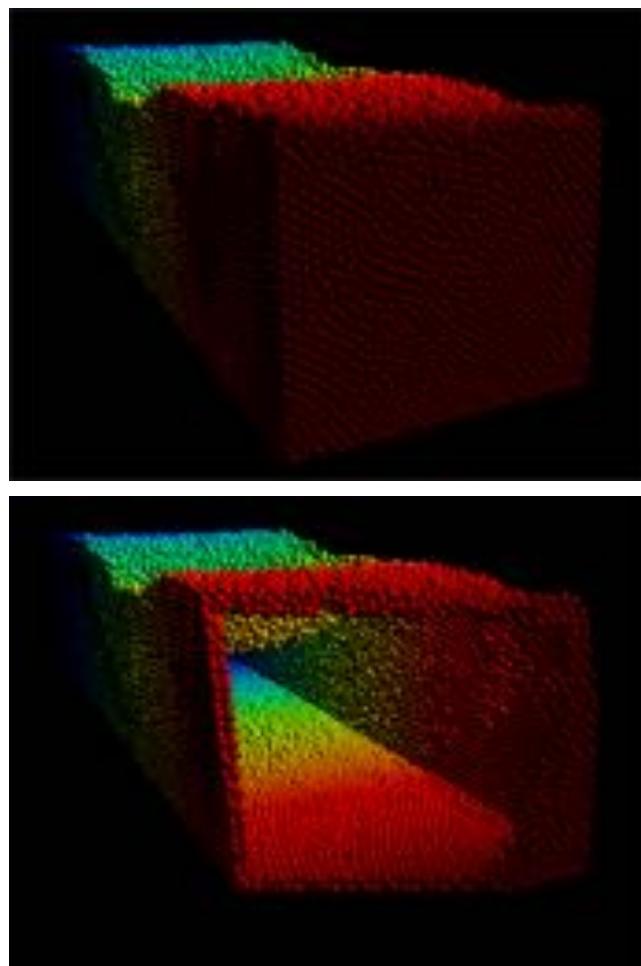


図 8.12: 表面粒子だけのレンダリング。図上は表面粒子のみレンダリングされた結果であり、図下はある面以前の粒子を描画しなかったもの。

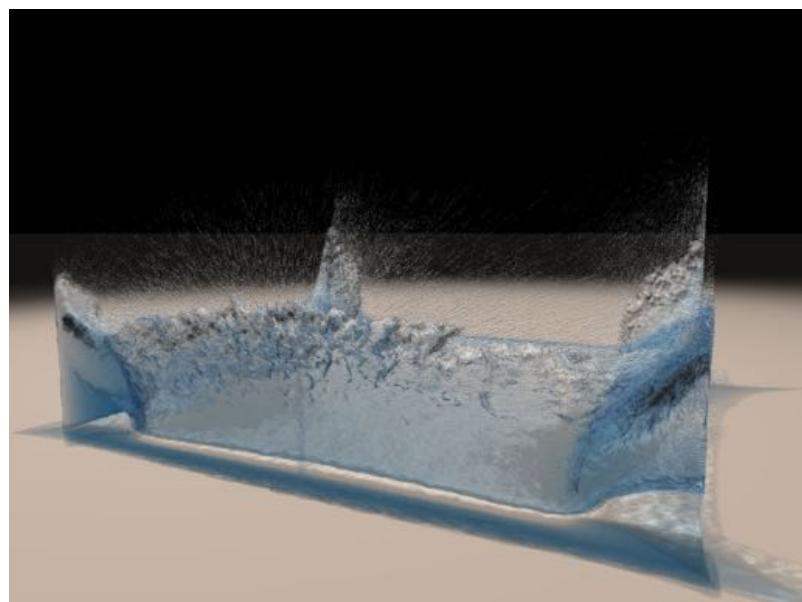


図 8.13: リアルタイムシミュレーションの結果をオフラインでレンダリングした結果。

第9章 結論

本研究ではまず映像制作のための粒子法の開発を行なった。壁粒子を用いた境界条件の表現は計算効率、計算精度、そして視覚的に良好でない計算結果を出すため、MPS法とSPHにおいて壁粒子を用いない壁境界計算手法を開発した。この手法によって傾きを持った境界なども、精度よく計算することが可能になり、複雑な境界条件下での計算も可能になった。また粒子法の計算結果の可視化において薄い膜や鋭いエッジを持つ表面を計算粒子から構築する手法を開発した。この手法を用いることによって様々な流体の表現が可能になった。粒子法シミュレーションでは近傍粒子探索においてユニフォームグリッドを用いるとメモリ効率が悪いため、広い計算領域に適用することが困難になるなどの問題があった。そこで本研究ではメモリ効率の良いスライスグリッドを開発した。スライスグリッドを用いることによってメモリ効率よく計算することが可能になったが、これは同時に同じ大きさのメモリを用いることが可能であれば、より広い計算領域を計算することができるようになったということである。これらの手法はオフラインの映像制作だけでなく、ゲームなどのリアルタイムアプリケーションなどにも用いることができる。

次に粒子法シミュレーションの近傍粒子探索をGPUを用いて高速化した。まずユニフォームグリッドを用いる近傍粒子探索をGPU上で並列で計算するアルゴリズムを開発し、さらにスライスグリッドを用いる近傍粒子探索をGPU上で実装するアルゴリズムを開発した。そして手法の応用としてまずDEMを用いた粉体、SPHを用いた流体のシミュレーションを高速化し、両方のシミュレーションで約十倍以上の計算の高速化を実現した。これによってリアルタイムで計算できる粒子数が数千から数万になり、高解像度のシミュレーションをリアルタイムアプリケーションで用いることができるようになった。第二部で提案した手法は粒子法を用いた粉体や流体シミュレーション以外の計算にも有効である。そこで本研究ではこれらのシミュレーション以外に、粒子を用いた剛体、布、そしてそれらの連成計算に本手法を応用した。さらに粒子法を用いた弾性体の計算もGPUを用いて実装したが、芳しい結果は得られなかった。この弾性体の計算と流体の計算を比較することで、この計算速度の向上の差の考察を行い、現在のGPUの欠点を指摘した。さらにGPU等のデータ並列プロセッサを用いてより広いアプリケーションを実装するための手法である、撃力ベースのシミュレーションの並列化や、木構造を用いた広域衝突検出を開発した。

さらに第三部ではこれらの応用として複数のGPUを用いた粒子法の高速化手法を開発した。この手法ではサーバとなる全てのデータを管理するプロセッサを置かず、計算領域を分割し、それぞれのGPUが一領域を受け持ちそのデータ管理を行なった。特にこの手法は計算の周期が速いリア

ルタイムアプリケーションをターゲットとしており少しの無駄な計算さえも如実に結果に現れる。よってデータ管理において、近傍粒子探索を効率化するために構築した格子を再利用することによって動的なデータ管理のオーバーヘッドを押さえ、計算を行なう GPU の数にほぼスケールした計算性能を出した。

GPU は行うことのできる計算に多くの制約があったが、その制約が緩和されつつある。また Intel も多くのコアを備えた Larrabee を開発しており [109]、今後これらのようなマルチコアプロセッサでの計算が重要になってくることは間違いない。本研究の成果は、それらのプロセッサ上での物理シミュレーションの基盤となるであろう。本研究では GPU によって多くの計算を高速化したが、同時に現在の GPU の限界も浮き彫りにした。その 1 つはレジスタ数の不足である。例えば粒子法でも弾性体の計算を GPU 上で実装する場合には、流体計算に比べて多くのレジスタを必要とする。1GPU 上のレジスタ数には限りがあり、1 スレッドがレジスタを多く使用すればするほど同時に実行できるスレッド数は減ってしまう。すなわちメモリアクセスのレイテンシを隠蔽することができなくなり、GPU の本来の性能を引き出すことができなく、問題によっては、マルチコア CPU において Streaming SIMD Extensions(SSE) を用いた場合のほうが高速に計算できる場合がある。また動的な木構造は剛体シミュレーションの広域衝突検出をはじめとする様々な分野で用いられる重要なデータ構造の 1 つであるが、この構築を効率的に行なうことができない。少なくとも CPU において最も効率的なアルゴリズムは GPU 上では非効率的であり、この問題は未解決な問題である。このように GPU はいくつかの限界があり、これらを解決しなければ全ての問題は GPU を用いることで高速化することができるとは断言することはできない。

本研究では粒子法の計算結果の可視化を改良する手法も開発し、この手法を用いることにより計算結果をオフラインで処理する場合の品質を上げることができる。しかしリアルタイムの粒子法の計算結果の可視化についてはまだ解決されておらず、様々な研究が行なわれており、我々も試行錯誤を重ねている。リアルタイムでの粒子法の計算の可視化は今後の課題の 1 つである。

もちろん本研究結果はゲームなどのリアルタイムアプリケーションに用いることが可能であるが、同時に工学利用も可能であると思われる。そのためには現在の GPU は単精度浮動小数点演算のみ高速に計算することしかできない。よって計算精度が求められる工学応用のためには、計算精度の定量的評価を行なう必要がある。

謝辞

謝辞

関連図書

- [1] B. Adams, M. Pauly, R. Keiser, and L.J. Guibas. Adaptively sampled particle fluids. *ACM Transactions on Graphics*, Vol. 26, No. 3, 2007.
- [2] T. Amada, M. Imura, Y. Yasumoto, Y. Yamabe, and K. Chihara. Particle-based fluid simulation on gpu. In *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004.
- [3] J.A. Baerentzen and H. Aanaes. Generating signed distance fields from triangle meshes. *IIM Technical Report*, p. 21, 2002.
- [4] J.A. Baerentzen and Aanaes.H. Generating signed distance fields from triangle meshes. *IIM Technical Report*, Vol. 21, , 2002.
- [5] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics*, Vol. 23, No. 3, pp. 223–232, 1989.
- [6] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *Computer Graphics*, Vol. 24, No. 4, pp. 19–28, 1990.
- [7] D. Baraff. Coping with friction for non-penetrating rigid body simulation. *Computer Graphics*, Vol. 25, No. 4, pp. 31–40, 1991.
- [8] D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Computer Graphics Proc., Annual Conference Series*, pp. 23–34, 1994.
- [9] D. Baraff. Interactive simulation of solid rigid bodies. *IEEE Computer Graphics and Applications*, Vol. 15, pp. 63–75, 1995.
- [10] D. Baraff. An introduction to physically based modeling: rigid body simulation ii. *SIGGRAPH Course Note*, 1997.

- [11] D. Baraff and A. Witkin. Large steps in cloth simulation. In *Proc. of SIGGRAPH 98*, pp. 43–54, 1998.
- [12] D. Baraff, A. Witkin, and M. Kass. Untangling cloth. *ACM Transactions on Graphics*, Vol. 22, No. 3, pp. 862–870, 2003.
- [13] N. Bell, Y. Yu, and P.J. Mucha. Particle-based simulation of granular materials. In *Proc. of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 77–86, 2005.
- [14] J. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, Vol. 1, No. 3, pp. 235–256, 1982.
- [15] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide 5 edition*. Addison-Wesley Professional, 2005.
- [16] A. Bond. Havok fx: Gpu-accelerated physics for pc games. In *Game Developers Conference*, 2006.
- [17] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics*, Vol. 21, No. 3, pp. 594–603, 2002.
- [18] I. Buck. *GPU Gems2*, chapter Taking the Plunge into GPU Computation. Addison-Wesley Pearson Education, 2005.
- [19] M. Carlson, P.J. Mucha, and G. Turk. Rigid fluid: Animating the interplay between rigid bodies and fluid. *ACM Transactions on Graphics*, Vol. 23, No. 3, pp. 377–384, 2004.
- [20] E. Catto. Modeling and solving constraints. In *Game Developers Conference, physics for game programmers*, 2008.
- [21] N. Chentanez, T.G. Goktekin, B.E. Feldman, and J.F. O’Brien. Simultaneous coupling of fluids and deformable bodies. In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 83–89, 2006.
- [22] K.J. Choi and H.S. Ko. Stable but responsive cloth. *ACM Transactions on Graphics*, Vol. 21, No. 3, pp. 604–611, 2002.
- [23] E. R. Clifford. A fast algorithm for calculating particle interactions in smooth particle hydrodynamics simulation. *Computer Physics Communications*, Vol. 70, pp. 478–482, 1992.

- [24] K. Crane, I. Llamas, and S. Tariq. *GPU Gems3*, chapter Real-Time Simulation and Rendering of 3D Fluids. Addison-Wesley Pearson Education, 2007.
- [25] S.J. Cummins and M. Rudman. An sph projection method. *Journal of Computational Physics*, Vol. 152, No. 2, pp. 584–607, 1999.
- [26] P.A. Cundall and O.D.L. Strack. A discrete numerical model for granular assemblies. *Geotechnique*, Vol. 29, pp. 47–65, 1979.
- [27] Cyril.Z. Cloth simulation on the gpu. In *ACM SIGGRAPH Sketches, No.39*, 2005.
- [28] J. Dingliana and C. O’Sullivan. Graceful degradation of collision handling in physically based animation. *Computer Graphics Forum*, Vol. 19, No. 3, pp. 239–247, 2000.
- [29] Z. Dong, W. Chen, H. Bao, H. Zhang, and Q. Peng. Real-time voxelization for complex polygonal models. In *Proc. of Pacific Graphics*, pp. 73–78, 2004.
- [30] Y.M.L. Edmond and S. Shao. Simulation of near-shore solitary wave mechanics by an incompressible sph method. *Applied Ocean Research*, Vol. 24, pp. 275–286, 2002.
- [31] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. *ACM Transactions on Graphics*, Vol. 21, pp. 721–728, 2002.
- [32] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Pub, 2005.
- [33] O. Etzmuss, M. Keckeisen, and W. Strasser. A fast finite element solution for cloth modeling. In *Proc. of Pacific Graphics*, pp. 244–251, 2003.
- [34] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *Proc. of the ACMIEEE conference on Supercomputing*, pp. 47–47, 2004.
- [35] R. Fernando. *GPU gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004.

- [36] R. Fernando and M. Kilgard. *The Cg Tutorial*. Addison-Wesley Pearson Education, 2003.
- [37] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proc. of Siggraph*, pp. 15–22, 2001.
- [38] O. Génevaux, A. Habibi, and J.M. Dischler. Simulating fluid-solid interaction. In *Graphics Interface*, pp. 31–38, 2003.
- [39] T. Goktekin, A.W. Bargteil, and J.F. O’Brien. A method for animating viscoelastic fluids. *ACM Transactions on Graphics*, Vol. 23, pp. 464–467, 2004.
- [40] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problem using programmable graphics hardware. In *Proc. of Graphics Hardware*, pp. 1–11, 2003.
- [41] S.L. Grand. *GPU Gems3*, chapter Broad-Phase Collision Detection with CUDA. Addison-Wesley Pearson Education, 2007.
- [42] J. P. Gray, J. J. Monaghan, and R. P. Swift. Sph elastic dynamics. *Computer methods in applied mechanics and engineering*, Vol. 190, No. 49, pp. 6641–6662, 2001.
- [43] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics*, Vol. 22, pp. 871–878, 2003.
- [44] E. Guendelman, A. Selle, F. Losasso, and R. Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Transactions on Graphics*, Vol. 24, pp. 910–914, 2005.
- [45] J.K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, Vol. 22, No. 4, pp. 299–308, 1988.
- [46] T. Harada. *GPU Gems3*, chapter Real-time Rigid Body Simulation on GPUs. Addison-Wesley Pearson Education, 2007.
- [47] T. Harada and S. Koshizuka. Fast solid voxelization using graphics hardware. *Transactions of Japan Society for Computational Engineering and Science*, No. 20060023, 2006.

- [48] T. Harada and S. Koshizuka. Fast solid voxelization using graphics hardware. *Transactions of JSSES*, p. No.20060023, 2006.
- [49] T. Harada and S. Koshizuka. Sphにおける壁境界計算手法の改良. 情報処理学会論文誌, Vol. 48, No. 4, pp. 1838–1846, 2007.
- [50] T. Harada, S. Koshizuka, and Y. Kawaguchi. Construction of non-blobby surface from particles. In *Proc. of Eurographics short paper*, pp. 41–44, 2007.
- [51] T. Harada, S. Koshizuka, and Y. Kawaguchi. Improvement of the boundary conditions in smoothed particle hydrodynamics. Computer Graphics & Geometry, Vol. 9, No. 3, pp. 2–15, 2007.
- [52] T. Harada, S. Koshizuka, and Y. Kawaguchi. Slided data structure for particle-based simulations on gpus. In *Proc. of GRAPHITE*, pp. 55–62, 2007.
- [53] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics in complex shapes. In *Proc. of Spring Conference on Computer Graphics*, pp. 235–241, 2007.
- [54] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Proc. of Computer Graphics International*, pp. 63–70, 2007.
- [55] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on gpus. In *Proc. of Computer Graphics International*, pp. 63–70, 2007.
- [56] T. Harada, S. Koshizuka, and K. Shimazaki. A wall boundary computation model by polygons for moving particle semi-implicit method. In *Proc. of WCCM8/ECMAS*, 2008.
- [57] T. Harada, Y. Suzuki, S. Koshizuka, T. Arakawa, and S. Shoji. Simulation of droplet generation in micro flow using mps method. *Japan Society of Mechanical Engineering International Journal Series B*, Vol. 49, No. 3, pp. 731–736, 2006.
- [58] T. Harada, M. Tanaka, S. Koshizuka, and Y. Kawaguchi. Acceleration of distinct element method using graphics hardware. *Transactions of Japan Society for Computational Engineering And Science*, No. 20070011, Vol. 2007, , 2007.

- [59] M. Harris. *GPU Gems3*, chapter Fast Fluid Dynamics Simulation on the GPU. Addison-Wesley Pearson Education, 2005.
- [60] M. Harris, S. Segupta, and J.D. Owens. *GPU Gems3*, chapter Parallel Prefix Sum (Scan) with CUDA. Addison-Wesley Pearson Education, 2007.
- [61] M.J. Harris, W.V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proc. of the SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 92–101, 2003.
- [62] M.J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. *Proc. of the SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 109–118, 2002.
- [63] M.J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Real-time 3d fluid simulation on GPU with complex obstacles. In *Proc. of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pp. 247–256, 2004.
- [64] S. Hasegawa, N. Fujii, Y. Koike, and M. Sato. Real-time rigid body simulation based on volumetric penalty method. In *Proc. of the 11th Symposium on Haptic Interfaces for VirtualEnvironment and Teleoperator Systems*, pp. 326–332, 2003.
- [65] S. Hasegawa and M. Sato. Real-time rigid body simulation for haptic interactions based on contact volume of polygonal objects. *Computer Graphics Forum*, Vol. 24, No. 3, pp. 946–956, 2004.
- [66] L. Hernquist and N. Katz. Treesph: A unification of sph with the hierarchical tree method. *The Astrophysical Journal Supplement Series*, Vol. 70, pp. 419–446, 1989.
- [67] G. Irving, E. Guendelman, F. Losasso, and R. Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Transactions on Graphics*, Vol. 25, pp. 812–819, 2006.
- [68] N. Junyong, W. Kwangyun, and O. SeungWoo. A physically faithful multigrid method for fast cloth simulation. *Computer Animation and Virtual Worlds*, Vol. 19, No. 3–4, pp. 479–492, 2008.

- [69] Y. Kanamori, Z. Szego, and T. Nishita. Gpu-based fast ray casting for a large number of metaballs. *Computer Graphics Forum*, Vol. 27, No. 2, pp. 351–360, 2008.
- [70] D.M. Kaufman, T. Edmunds, and D.K. Pai. Fast frictional dynamics for rigid bodies. *ACM Transactions on Graphics*, Vol. 24, No. 3, pp. 946–956, 2005.
- [71] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A GPU-based particle engine. *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 115–122, 2004.
- [72] B.M. Klingner, B.E. Feldman, N. Chentanez, and J.F. O’Brien. Fluid animation with dynamic meshes. *ACM Transactions on Graphics*, Vol. 25, pp. 820–825, 2006.
- [73] B.M. Klingner, B.E. Feldman, N. Chentanez, and J. O’Brien. Fluid amination with dynamic meshes. *ACM Transactions on Graphics*, Vol. 25, No. 3, pp. 820–825, 2006.
- [74] A. Kolb and N. Cuntz. Dynamic particle coupling for GPU-based fluid simulation. In *Proc. of 18th Symposium on Simulation Technique*, pp. 722–727, 2005.
- [75] A. Kolb, L. Latta, and C. Rezk-Salama. hardware based simulation and collision detection for large particle systems. *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 123–131, 2004.
- [76] S. Koshizuka and Y. Oka. Moving-particle semi-implicit method for fragmentation of incompressible flow. *Nucl. Sci. Eng.*, Vol. 123, pp. 421–434, 1996.
- [77] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, Vol. 22, No. 3, pp. 908–916, 2003.
- [78] N. Lars and M. Harris. *GPU Gems3*, chapter Fast N Body Simulation with CUDA. Addison-Wesley Pearson Education, 2007.
- [79] E. Lengyel. *The OpenGL Extensions Guide*. Charles River Media, 2003.

- [80] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proc. of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 163–169, 1987.
- [81] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics*, Vol. 23, pp. 457–462, 2004.
- [82] F. Losasso, T. Shinar, A. Selle, and R. Fedkiw. Multiple interacting liquids. *ACM Transactions on Graphics*, Vol. 25, pp. 812–819, 2006.
- [83] L.B. Lucy. Numerical approach to testing the fission hypothesis. *Astronomical Journal*, Vol. 82, pp. 1013–1024, 1977.
- [84] J. Mezger, S. Kimmerle, and O. Etzmuß. Hierarchical techniques in collision detection for cloth animation. *Journal of WSCG*, Vol. 11, No. 1, pp. 322–329, 1995.
- [85] B. Mirtich. Timewarp rigid body simulation. In *Proc. of SIGGRAPH 2000*, pp. 193–200, 2000.
- [86] B.K. Mishra. A review of computer simulation of tumbling mills by the discrete element method: Parti-contact mechanics. *International Journal of Mineral Processing*, Vol. 71, No. 1, pp. 73–93, 2003.
- [87] T. Moller. A fast triangle triangle intersection test. *Journal of Graphics Tools*, Vol. 2, No. 2, pp. 25–30, 1997.
- [88] J.J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, Vol. 30, pp. 543–574, 1992.
- [89] J.J. Monaghan. Simulating free surface flows with sph. *Journal of Computational Physics*, Vol. 110, No. 2, pp. 399–406, 1994.
- [90] J. Montrym and H. Moreton. The geforce 6800. *IEEE Micro*, Vol. 25, No. 2, pp. 41–51, 2005.
- [91] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, Vol. 22, No. 4, pp. 289–298, 1988.

- [92] Matthias Mueller, Richard Keiser, Andy Nealen, Mark Pauly, Markus Gross, and Marc Alexa. Point-based animation of elastic, plastic, and melting objects. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Computer Animation*, 2004.
- [93] M. Muller. Hierachical position based dynaics. In *Proc. of Virtual Reality Interactions and Physical Simulations*, pp. 1–10, 2008.
- [94] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proc. of Siggraph Symposium on Computer Animation*, pp. 154–159, 2003.
- [95] M. Muller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, Vol. 18, No. 2, pp. 109–118, 2007.
- [96] M. Muller, S. Schirm, M. Teschner, B. Heidelberger, and M. Gross. Interaction of fluids with deformable solids. *Journal of Computer Animation and Virtual Worlds*, Vol. 15, No. 3, pp. 159–171, 2004.
- [97] M. Müller, B. Solenthaler, R. Keiser, and M. Gross. Particle-based fluid-fluid interaction. In *Proc. of Siggraph Symposium on Computer Animation*, pp. 237–244, 2005.
- [98] N. Nakasato, M. Mori, and K. Nomoto. Smoothed particle hydrodynamics with grape and parallel virtual machine. *Astrophysical Journal*, Vol. 484, pp. 608–617, 1997.
- [99] NVIDIA. Compute unified device architecture. http://www.nvidia.com/object/cuda_home.html.
- [100] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, pp. 21–51, 2005.
- [101] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.
- [102] S. Premoze, T. Tasdizen, J. Bigler, A. Lefohn, and R.T. Whitaker. Particle-based simulation of fluids. *Computer Graphics Forum*, Vol. 22, No. 3, pp. 401–410, 2003.

- [103] X. Provot. Deformation constraints in a mass spring model to describe rigid cloth behavior. In *Proc. of Graphic Interface*, pp. 147–154, 1995.
- [104] X. Provot. Collision and self-collision handling in cloth model dedicated to design garments. In *Proc. of Computer Animation and Simulation*, pp. 177–189, 1997.
- [105] T.J. Purcell, M. Cammarano, H.W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 41–50, 2003.
- [106] F.L.B. Ribeiro and I.A. Ferreira. Parallel implementation of the finite element method using compressed data structures. *Computational Mechanics*, Vol. 41, No. 1, pp. 31–48, 2007.
- [107] R.J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2004.
- [108] H. Schmidl and V.J. Milenkovic. A fast impulsive contact suite for rigid body simulation. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 10, No. 2, pp. 189–197, 2004.
- [109] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Adam Lake Pradeep Dubey1, Stephen Junkins1, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, , and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, Vol. 27, No. 3, 2008.
- [110] K. Shibata and S. Koshizuka. Numerical analysis of shipping water impact on a deck using a particle method. *Ocean Engineering*, pp. 589–593, 2005.
- [111] J.M. Snyder, A.R. Woodbury, K. Fleischer, B. Currin, and A.H. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. In *Proc. of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 321–334, 1993.
- [112] J. Stam. Stable fluids. In *Proc. of ACM SIGGRAPH 99*, pp. 121–128, 1999.

- [113] K. Stratford and I. Pagonabarraga. Parallel simulation of particle suspensions with the lattice boltzmann method. *Computers & Mathematics with Applications*, Vol. 55, No. 7, pp. 1585–1593, 2008.
- [114] M. Sueyoshi. Numerical simulation of free surface problems on naval and ocean engineering by using moving particle semi-implicit method. In *Proc. of International Conference on Computational Methods*, p. 304, 2007.
- [115] K. Suzuki, J. Kubota, and H. Ohtsubo. Rigid body simulation using volume based collision detection. *Structural Engineering / Earthquake Engineering*, Vol. 22, pp. 185–192, 2003.
- [116] L. Sylvain and H. Huges. Perfect spatial hashing. *ACM Transactions on Graphics*, Vol. 25, No. 3, 2006.
- [117] T. Takahashi, H. Fujii, A. Kunimatsu, K. Hiwada, T. Saito, K. Tanaka, and H. Ueki. Realistic animation of fluid with splash and foam. *Computer Graphics Forum*, Vol. 22, No. 3, pp. 391–400, 2003.
- [118] M. Tanaka, M. Sakai, and S. Koshizuka. Particle-based rigid body simulation and coupling with fluid simulation. *Transactions of Japan Society for Computational Engineering And Science*, No.20070007, Vol. 2007, , 2007.
- [119] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proc. of Vision, Modeling, Visualization*, pp. 47–54, 2003.
- [120] M. Teschner, S. Kimmerle, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.P. Cani, F. Faure, N. Magnetat-Thalmann, and W. Strasser. Collision detection for deformable objects. In *Eurographics State of the Art Report*, pp. 119–139, 2004.
- [121] B. Thomaszewski and W. Blochinger. Parallel simulation of cloth on distributed memory architectures. In *Proc. of EG Symposium on Parallel Graphics and Visualization*, pp. 35–42, 2006.
- [122] B. Thomaszewski, M. Wacker, and W. Straßer. A consistent bending model for cloth simulation with corotational subdivision finite

- elements. In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 0–10, 2006.
- [123] P. Volino and N. Magnenat-Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum*, Vol. 13, pp. 155–166, 1994.
- [124] P. Volino and N. Magnenat-Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. In *Proc. of SIGGRAPH*, pp. 137–144, 1995.
- [125] P. Volino and N. Magnenat-Thalmann. Resolving surface collisions through intersection contour minimization. *ACM Transactions on Graphics*, Vol. 25, No. 3, pp. 1154–1159, 2006.
- [126] X. Wang, L. Guo, D. Tang, J. Ma, Z. Yang, and J. Li. Parallel implementation of micro scale pseudo particle simulation for particle fluid systems. *Computers & Chemical Engineering*, Vol. 29, No. 7, pp. 1543–1553, 2005.
- [127] M. Wicke, H. Lanker, and M. Gross. Untangling cloth with boundaries. In *Proc. of Vision, Modeling and Visualization*, pp. 349–356, 2006.
- [128] G.D. Yngve, O'Brien J.F., and J.K. Hodgins. Animating explosions. In *Proc. of ACM SIGGRAPH 2000*, pp. 29–36, 2000.
- [129] Y. Zhu and R. Bridson. Animating sand as a fluid. *ACM Transactions on Graphics*, Vol. 24, No. 3, pp. 965–972, 2005.
- [130] 越塚誠一. 数値流体力学. 培風館, 1997.
- [131] 越塚誠一. 粒子法. 丸善株式会社, 2005.
- [132] 越塚誠一, 原田隆宏, 田中正幸, 近藤雅裕. 粒子法シミュレーション. 培風館, 2008.
- [133] 近藤雅裕, 越塚誠一, 鈴木幸人. 3次元mps法弹性体解析へのシンプレクティックスキームの適用. 日本機械学会論文集(A), Vol. 72, No. 4, pp. 425–431, 2006.
- [134] 近藤雅裕, 鈴木幸人, 越塚誠一. 最小自乗近似による粒子法弹性解析手法の振動抑制. 日本計算工学会論文集, No. No. 20070031, 2007.

- [135] 原田隆宏, 越塙誠一, 河口洋一郎. Gpu を用いた粒子法シミュレーションのためのスライスデータ構造. 日本計算工学会論文集, No. No. 20070028, 2007.
- [136] 原田隆宏, 越塙誠一, 河口洋一郎. 粒子からの表面構築の改良. 情報処理学会論文誌, Vol. 48, No. 12, pp. 4059–4063, 2007.
- [137] 原田隆宏, 越塙誠一, 島崎克教. Mps 法における壁境界計算モデルの改良. 日本計算工学会論文集, No. No. 20080006, 2008.
- [138] 原田隆宏, 政家一誠, 越塙誠一, 河口洋一郎. Gpu 上での粒子法シミュレーションの空間局所性を用いた高速化. 日本計算工学会論文集, No. No. 20080016, 2008.
- [139] 入部綱清, 藤澤智光, 柴田和也, 越塙誠一. Mps 法を用いた流体並列解析に関する基礎的研究. *Transactions of JSCES*, Vol. 20060015, , 2006.
- [140] 鈴木幸人, 越塙誠一. 非線形弾性体に対する粒子法の開発. 日本計算工学会論文集, No. No. 20070001, 2007.