修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　　情報・ネットワーク工学専攻　博士前期課程		
氏　　名	LI JIAQING	学籍番号	2031160
論 文 題 目	Unity によるマイクロマグネティクス シミュレーションのリアルタイム可視化		

要　旨

マイクロマグネティクスとは，磁石内部に現れる原子磁気モーメントによって作られる磁化構造やその動的な変化を扱う分野であり，ハードディスクのヘッドや MRAM のシミュレーションなどに用いられる．マイクロマグネティクスシミュレーションでは，磁気モーメントの相互作用があるため計算量が多くその動きは予測しずらい．シミュレーションを理解するには可視化する必要がある．また，解析サイクルを早くできるように，GPU の高い演算能力を用いてリアルタイムにシミュレーションすることは有用である．

本研究では，ゲームエンジンの Unity を用いてマルチプラットフォーム対応のマイクロマグネティクスシミュレーションのリアルタイム可視化システムを開発した．本システムでは，物理シミュレーションの部分はグラフィクス API(OpenGL，Direct3D など)に内蔵される GPGPU 機能(Compute shader)で高速に計算される．計算されたシミュレーションの結果は Unity のビルトインレンダリングパイプラインに送り，リアルタイムに描画される．プラットフォームにとらわれず，あらかじめドライバーなどをインストールする必要もなく，ほとんどのデバイスで実行可能のため，本システムは簡易的なシミュレーションを行う場面で有効である．

本文では C++(CUDA)と Unity(Compute shader)を用いた二種類のシミュレーションシステムを実装し，比較，評価した．同じ手法と条件でマイクロマグネティクスシミュレーションを実行すると，Compute shader を用いた場合は CUDA より 2 倍，CPU より約 40 倍速かった．1152 粒子の条件で，最新のスマートフォンだと 20FPS 以上のフレームレートが得られる．小規模のシミュレーションであれば，スマートフォンなどのデバイスでもリアルタイムにシミュレーションと可視化が可能となった．学生などのユーザーはスマートフォンなどのデバイスで手軽にシミュレーションができ，教育用途に役立つ可能性がある．

令和 3 年度　修士論文

Unity によるマイクロマグネティクス シミュレーションのリアルタイム可視化

電気通信大学　情報理工学研究科
情報・ネットワーク工学専攻
コンピュータサイエンスプログラム

学籍番号 2031160
氏名 LI JIAQING

指導教員　成見　哲　教授
副指導教員　仲谷　栄伸　教授
令和 4 年度 1 月 28 日

専攻長印	主指導教員印	指導教員印

概要

マイクロマグネティクスとは, 磁石内部に現れる原子磁気モーメントによって作られる磁化構造やその動的な変化を扱う分野であり，ハードディスクのヘッドや MRAM のシミュレーションなどに用いられる．マイクロマグネティクスシミュレーションでは，磁気モーメントの相互作用があるため計算量が多くその動きは予測しずらい. シミュレーションを理解するには可視化する必要がある．また，解析サイクルを早くできるように，GPU の高い演算能力を用いてリアルタイムにシミュレーションすることは有用である．

本研究では, ゲームエンジンの Unity を用いてマルチプラットフォーム対応のマイクロマグネティクスシミュレーションのリアルタイム可視化システムを開発した. 本システムでは，物理シミュレーションの部分はグラフィクス API(OpenGL，Direct3D など)に内蔵される GPGPU 機能(Compute shader)で高速に計算される．計算されたシミュレーションの結果は Unity のビルトインレンダリングパイプラインに送り，リアルタイムに描画される．プラットフォームにとらわれず，あらかじめドライバーなどをインストールする必要もなく，ほとんどのデバイスで実行可能のため，本システムは簡易的なシミュレーションを行う場面で有効である．

本文では C++(CUDA)と Unity(Compute shader)を用いた二種類のシミュレーションシステムを実装し，比較，評価した．同じ手法と条件でマイクロマグネティクスシミュレーションを実行すると，Compute shader を用いた場合は CUDA より 2 倍，CPU より約 40 倍速かった．1152 粒子の条件で，最新のスマートフォンだと 20FPS 以上のフレームレートが得られる．小規模のシミュレーションであれば，スマートフォンなどのデバイスでもリアルタイムにシミュレーションと可視化が可能となった. 学生などのユーザーはスマートフォンなどのデバイスで手軽にシミュレーションができ，教育用途に役立つ可能性がある．

目次

第１章　はじめに

本章では背景や研究目的，論文構成について記載する．

1.1　背景

自然現象の解析や分析は工業製品の開発に欠かせないものであり，日頃から盛んに行われている．これらの解析と分析にはコンピューターでシミュレーションがよく用いられる．そして，シミュレーションされたデータは科学的可視化を行ってはじめて目で確認できる．科学的可視化は膨大な科学データを分析するため，科学データをひと目で分かるように映像化することである．グラフィック技術を用いて科学データを分かり易く可視化できれば，研究者のデータ分析もスムーズに進む．

特に，リアルタイムに可視化しながらシミュレーションできれば，解析サイクルを早く出来る．画像処理を担当するGraphics Processing Unit(GPU)に汎用計算を行わせるGPGPU(General-purpose computing on GPU)が近年流行しており[1]，シミュレーションをした結果をそのまま描画にも使用することで可視化しながらのシミュレーションも高速化できる．

さらに，近年ではモバイル端末が普及しており，性能も高くなっている．これまで高性能なPCでしか不可能だったリアルタイムシミュレーションがスマートフォンでも可能になってきている

1.2　研究目的

本研究では，Unityよるマイクロマグネティクスシミュレーションのリアルタイム可視化システムを開発する．Unity の機能を使って，Compute shader(cs)を用いてシミュレーションを高速化し，Shaderプログラムを用いて描画する．NVIDIAのGPUだけではなく，他のブランドの GPU，あるいは CPU 内蔵GPU でシミュレーションを実行することができる．これにより，学生はスマートフォンなどのデバイスで手軽にシミュレーションができ，教育用途に役立つ．

1.3 構成

本文の構成は以下のようになっている．

第 1 章　はじめに
本研究の背景，目的を記載する．
第 2 章 マイクロマグネティクスシミュレーション
マイクロマグネティクスについて記載する．
第 3 章　GPU と GPGPU
GPU および GPGPU について記載する．
第 4 章　可視化とゲームエンジン
可視化とゲームエンジン Unity について記載する．
第 5 章　既存研究
関連する既存研究について記載する．
第 6 章　Unity によるリアルタイムシミュレーション可視化システムの開発
Unity によるシステムのシミュレーション計算プログラムと可視化プログラムの実装について記載する．
第 7 章　CUDA ベースのリアルタイムシミュレーションシステムの開発
C++と CUDA で実装したシミュレーション計算プログラムについて記載する．
第 8 章　システムの評価
実験内容および結果と考察を記載する．
第 9 章　おわりに
研究のまとめおよび今後の課題について記載する．

第 2 章　マイクロマグネティクスシミュレーション

本章ではマイクロマグネティクスシミュレーションの原理とシミュレーション手法について記述する．

2.1　原子磁気モーメントの運動

磁石の内部を詳しく観察すると，各電子は N 極と S 極を持つ原子の磁気モーメントが一様ではなく，さまざまな構造を形成していることがわかる．原子が磁場の中に放置されると，原子の磁気モーメントを回転させることができる（図 2.1）．この運動は，式（2.1）の Landau-Lifshitz-Gilbert 方程式（LLG 方程式）で記述される．以下では，原子の磁気モーメントを磁気モーメントに略称する．

$$\dot{\vec{M}} = -|\gamma|\vec{M} \times \vec{H} + \frac{\alpha}{M}(\vec{M} \times \dot{\vec{M}}) \qquad (2.1)$$

ここで，$\vec{M}$ は磁気モーメント，$\dot{\vec{M}}$は$\vec{M}$ の時間微分，$\vec{H}$は磁気モーメントに加わる磁界，γは磁気回転比と呼ばれる物理定数で，$\gamma = 1.76 \times 10^7$[rad/(sOE)]，α は Gilbert の損失定数と呼ばれる無次元定数である．

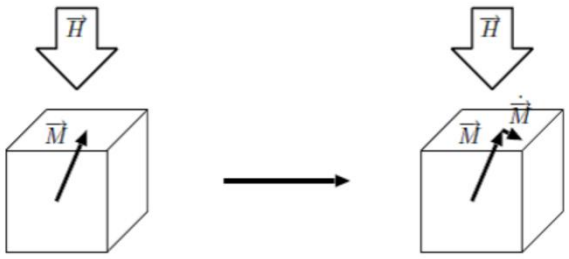


図 2.1:原子磁気モーメントの運動(参考文献[1]の図 2.1)

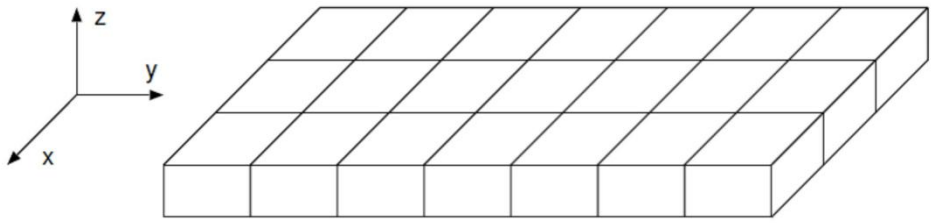


図 2.2:原子磁気モーメントの運動(参考文献[1]の図 2.2)

2.2 LLG 方程式の数値解法

LLG 方程式を数値計算方法により，磁気モーメントの運動をシミュレーションすることができる．LLG 方程式の数式をコンピュータで計算するために空間と時間上の離散化を行う必要がある．

2.2.1 空間の離散化

磁気モーメントの運動は位置によって異なるので，計算領域を直方体格子状（セル）に区切り，各セルについて個別に磁気モーメントの運動を求めることにする．各セルの中心に磁気モーメントを１つ位置して，そのセルでの磁気モーメントを代表させる．これを空間の離散化という（図 2.2）．

2.2.2 時間の離散化

LLG 方程式では，$\vec{M}$の時間微分項がある．そこで，時間の離散化を行い，微分を差分に近似して数値的に解けるようにする．式（2.1）は，磁気モーメントの時間微分 $\dot{\vec{M}}$ が両辺に現れているので，これを陽にあらわす形に変形する(式(2.2))．ここで，は$\vec{M} = |M|\vec{m}$を満たす単位ベクトルとし，磁気モーメントの絶対値は変化しないということを表す．

$$\dot{\vec{m}} = \frac{|\gamma|}{1+\alpha^2}\left(\vec{m} \times \vec{H} + \alpha\left((\vec{m} \cdot \vec{H})\vec{m} - \vec{H}\right)\right) \qquad (2.2)$$

式の簡単化のために $\mathrm{d} = |\gamma|/(1+\alpha^2)$，$\mathrm{C} = \vec{\mathrm{m}} \cdot \vec{\mathrm{H}}$ と置くと，

$$\dot{\vec{m}} = \mathrm{d}\left(\vec{m} \times \vec{H} + \alpha\left(\mathrm{c}\,\vec{m} - \vec{H}\right)\right) \qquad (2.3)$$

となる．式(2.3)をベクトル成分ごとに書き表すと，

$$\dot{\overrightarrow{m_x}} = d(m_y H_z - m_z H_y + \alpha(cm_x - H_x))$$

$$\dot{\overrightarrow{m_y}} = d(m_z H_x - m_x H_z + \alpha(cm_y - H_y)) \qquad (2.4)$$

$$\dot{\overrightarrow{m_z}} = d(m_x H_y - m_y H_x + \alpha(cm_z - H_z))$$

となる.

オイラー法

オイラー法は微分方程式を数値計算するための解法の一つである．式(2.3)を時間方向に離散化して, 数値的に解くのにオイラー法が使われる. 一般に, $f(x)$は，ある狭い範囲で微分可能とすると，$f(x)$の Taylor 展開は,

$$f(x + dx) = f(x) + \frac{df(x)}{dx}dx + \frac{1}{2}\frac{d^2f(x)}{dx^2}dx^2 + \frac{1}{6}\frac{d^3f(x)}{dx^3}dx^3 + O(dx^4) \quad (2.5)$$

となる．ただし，dx は x の微小な増分である．式(2.5)の第 2 項までを取り，第３項以降を無視すると,

$$f(x + dx) = f(x) + \frac{df(x)}{dx}dx$$

が得られる.

これを，式(2.3)に適用すると，次のようになる．時刻 t における磁気モーメントの値$\overrightarrow{m}$を$\overrightarrow{m}$(t)と書くことにすると，時刻 t + Δt における磁気モーメントの値 $\overrightarrow{m}$(t + Δt) は次の漸化式で計算することができる.

$$\overrightarrow{m}(t + \Delta t) = \overrightarrow{m}(t) + \Delta t \cdot d(\overrightarrow{m}(t) \times \vec{H} + \alpha(c\overrightarrow{m}(t) - \vec{H})) \qquad (2.6)$$

式(2.6)を使って計算を行うと，式(2.6)の第２項により，$\overrightarrow{m}$の大きさが１ステップの計算ごとに増大する. そこで，１ステップの計算のあとに, $\overrightarrow{m}$の各成分を$\overrightarrow{m}$の絶対値で割り，$\overrightarrow{m}$の絶対値を再び１に戻す操作を行う．この操作を再規格化という．計算を進めていくには，再規格化を考慮に入れた次のアルゴリズムを用いる.

1. t = 0 とし，磁気モーメントの初期値$\vec{m}(0)$を与える．
2.以下の(a)，(b)，(c)を行う．
   (a) $\vec{m}(t + \Delta t) \leftarrow \vec{m}(t) + \Delta t \cdot d(\vec{m}(t)) \times \vec{H} + \alpha(c\vec{m}(t) - \vec{H}))$
   (b) $\vec{m}(t + \Delta t) \leftarrow \vec{m}(t + \Delta t) / |\vec{m}(t + \Delta t)|$
   (c) $t \leftarrow t + \Delta t$

3.平衡状態に達していなければ，ステップ 2.に戻る．
このように，$\vec{m}(t)$ から$\vec{m}\,(t + \Delta t)$を求めていく方法をオイラー法と呼ぶ．オイラー法は精度が低く，Δtを小さくして計算を行わなければならない．より精度を上げるために，次の節で述べる 4 次のルンゲクッタ法を使用する．

ルンゲクッタ法

4 次のルンゲクッタ法では，

$$\begin{aligned} \vec{m}(t + \Delta t) &= \vec{m}(t) + \Delta t/6(k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= f(t, \vec{m}(t)) \\ k_2 &= f(t + \Delta t/2,\ \vec{m}(t) + \Delta t/2 \cdot k_1) \\ k_3 &= f(t + \Delta t/2,\ \vec{m}(t) + \Delta t/2 \cdot k_2) \\ k_4 &= f(t + \Delta t,\ \vec{m}(t) + \Delta t \cdot k_3) \end{aligned} \qquad (2.7)$$

とする．4 次のルンゲクッタ法を使うことでΔtを大きくすることができる．

2.3 実効磁界の計算

式(2.1)では$\vec{H}$を外部磁界としたが，外部磁界以外の影響も実効磁界として式(2.1)に組み込むことができる．実効磁界は外部磁界$\overrightarrow{H^{EXT}}$，一軸異方向エネルギーによる磁界$\overrightarrow{H^{A}}$，交換エネルギーによる磁界$\overrightarrow{H^{E}}$，静磁界$\overrightarrow{H^{D}}$の和からなるとする．

$$\vec{H} = \overrightarrow{H^{EXT}} + \overrightarrow{H^{A}} + \overrightarrow{H^{E}} + \overrightarrow{H^{D}} \qquad (2.8)$$

式(2.8)の各磁界の項は，エネルギーを磁界に換算することにより求める．磁気エネルギー密度をϵとすると，実効磁界は以下の式で表される．

$$\vec{H} = -\frac{\delta\epsilon}{\delta\vec{M}} \qquad (2.9)$$

以下では，一軸異方性による磁界，交換エネルギーによる磁界，静磁界を考える．

2.3.1 一軸異方性

一軸異方性とは，磁気モーメントがある一つの方向に向きやすい性質を指す．磁気モーメントが向きやすい方向のことを容易軸という．一軸異方性エネルギーϵ^kは，一軸異方性定数をK_u（容易軸を z 方向），磁気モーメントを$\overrightarrow{M}$とすると，

$$\epsilon^K = K_u(1 - M_z^2) \tag{2.10}$$

で表される．ここで，

$$\overrightarrow{H^A} = -\frac{\delta\epsilon^K}{\delta\overrightarrow{M}}$$

として，一軸異方性エネルギーを磁界に換算すると，

$$(\overrightarrow{H_x^A},\ \overrightarrow{H_y^A},\ \overrightarrow{H_z^A}) = (0\,,0\,,\ \frac{2\mathrm{K_u}}{\mathrm{M}^2}\ \mathrm{M_z}) \tag{2.11}$$

となる．

2.3.2 交換エネルギーによる磁界

磁気モーメントには，交換エネルギーを抑制し，小さくするような存在がある．これを磁界に変換したものを交換磁界という．隣接する磁気モーメントに大きな差があるとき交換エネルギーが大であり，差が少ないときは交換エネルギーが小である．そのため，隣接する磁気モーメントは，交換エネルギーが小さくなるようにできるだけ平行になろうとする．

交換エネルギーϵ^Aは，交換定数を A，磁気モーメントを$\overrightarrow{m}$とすると，

$$\begin{aligned}\epsilon^A &= A(\,\nabla\overrightarrow{m}\,)^2 \\ &= A(\,(\frac{\partial m}{\partial x}),(\frac{\partial m}{\partial y}),(\frac{\partial m}{\partial z}))^2 \\ &= A(\,(\frac{\partial m}{\partial x})^2 + (\frac{\partial m}{\partial y})^2 + (\frac{\partial m}{\partial z})^2)\end{aligned}$$

で表される．これを磁界に換算すると，交換エネルギーによる磁界$\overrightarrow{H^E}$は，

$$\overrightarrow{H^E} = -\frac{\partial \epsilon^A}{\partial \vec{M}}$$
$$= \frac{2A}{M}\left(\frac{\partial^2 \vec{m}}{\partial x^2} + \frac{\partial^2 \vec{m}}{\partial y^2} + \frac{\partial^2 \vec{m}}{\partial z^2}\right)$$
$$= \frac{2A}{M}\left(\frac{\partial^2 \vec{m}}{\partial x^2} + \frac{\partial^2 \vec{m}}{\partial y^2}\right) \quad (2\text{ 次元の場合})$$
$$= \left(\frac{2A}{M}\left(\frac{\partial^2 m_x}{\partial x^2} + \frac{\partial^2 m_x}{\partial y^2}\right), \frac{2A}{M}\left(\frac{\partial^2 m_y}{\partial x^2} + \frac{\partial^2 m_y}{\partial y^2}\right), \frac{2A}{M}\left(\frac{\partial^2 m_z}{\partial x^2} + \frac{\partial^2 m_z}{\partial y^2}\right)\right)$$

となる.

2.3.3 静磁界計算

磁気モーメントは一つ一つの小さいな磁石である．そのため，自身の周囲に磁界を作る．１つの磁気モーメントは周囲にある磁気モーメントからの磁界の影響を受けていることになる．磁石が作る磁界のことを静磁界という．

磁気モーメントを求める点は，各計算セルの中心部に配置する．１つの計算セル内では，磁気モーメントはすべて同じ方向を向くと仮定する．これによって，計算領域内に現れる磁荷はすべて，計算セルの表面に現れることになる．１つの計算点での静磁界は，計算領域内のすべての磁荷が，今考えている計算点に作り出す静磁界の和として求めることができる．次に，直方体セルの表面に現れる磁荷が作り出す静磁界を求める．

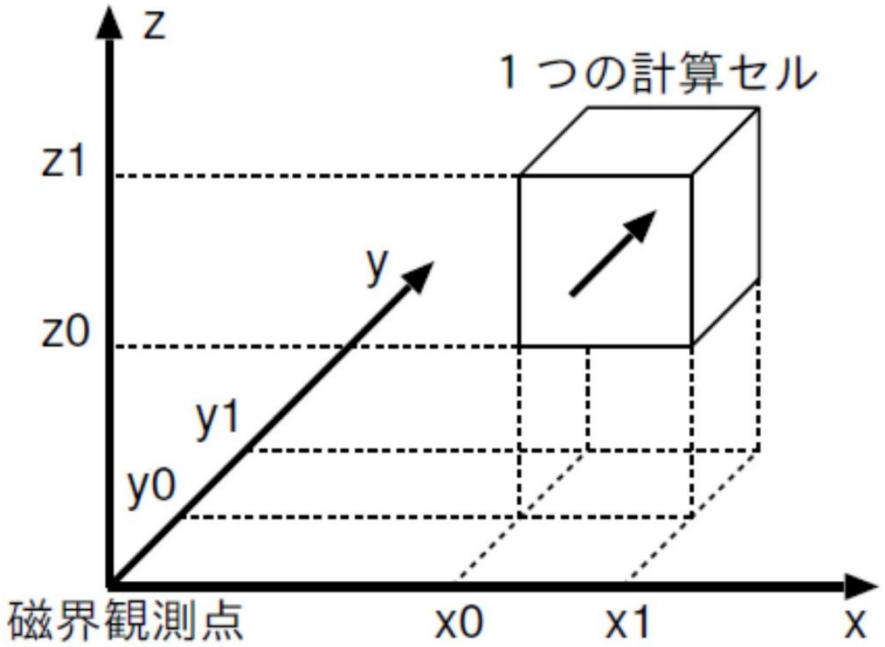


図 2.3 静磁界の計算(参考文献[1]の図 2.3)

計算領域内の 1 つの計算セルと磁界観測点との関係は図 2.3 で表される．計算セルの各面は x-y， y-z， x-z 面にそれぞれ平行であるとする．計算セルの各面は，それぞれ面密度 ± Mx， ± My および± Mz で磁荷が分布しているものとする．これらの 6 つの面を，y-z 面に平行な面， x-z 面に平行な面および y-z 面に平行な面に分け，それぞれの面に現れる磁荷が観測点に作る静磁界を求める．まず y-z 面に平行な右側の面を考える．観測点から(x_1, y_1, z_1)離れたこの面上の微小領域が観測点に作り出す磁界は，以下の式で表される．

$$\Delta H_x = -\frac{Mx}{r^2}\frac{x_1}{r}\Delta y \Delta z,$$

$$\Delta H_y = -\frac{Mx}{r^2}\frac{y_1}{r}\Delta y \Delta z,$$

$$\Delta H_x = -\frac{Mx}{r^2}\frac{z_1}{r}\Delta y \Delta z,$$

$$r = \sqrt{x^2 + y^2 + z^2}$$

対象とする面上の磁荷が観測点に作り出す磁界は，これらの微小領域が作り出す磁界を，面にわたって積分することで求められる．

$$H_x = \int_{z_0}^{z_1}\int_{y_0}^{y_1} \Delta H_x$$

$$H_y = \int_{z_0}^{z_1}\int_{y_0}^{y_1} \Delta H_y$$

$$H_z = \int_{z_0}^{z_1}\int_{y_0}^{y_1} \Delta H_z$$

同様に，y-z 面に平行な左側の面上の磁荷が観測点に作り出す磁界を求め，これらの式をまとめる．同様の操作を x-z 面に平行な 2 つの面，x-y 面に平行な 2 つの面で行い，それぞれをまとめると，計算セル上の磁荷が観測点に作り出す磁界は，以下の式で表される．

$$H_x = qxx \cdot mx + qxy \cdot my + qxz \cdot mz,$$

$$H_y = qxy \cdot mx + qyy \cdot my + qyz \cdot mz,$$

$$H_z = qxz \cdot mx + qyz \cdot my + qzz \cdot mz$$

今回の計算では，各計算セルは同じ大きさであり，同じ間隔で規則的に並んでいるために，計算セルが他の計算点に作り出す静磁界を求めるために使う静磁界係数は，これらの点の間隔だけで決まる．このことより，ある計算点の静磁界は以下の式で表される．

$$H_x(i,j) = \sum_{i'=1}^{N}\sum_{j'=1}^{N}[qxx(i'-i,j'-j)\cdot mx(i',j') + qxy(i'-i,j'-j)\cdot my(i'-j') + qxz(i'-i,j'-j)\cdot mz(i',j')]$$

$$H_y(i,j) = \sum_{i'=1}^{N}\sum_{j'=1}^{N}[qxy(i'-i,j'-j)\cdot mx(i',j') + qyy(i'-i,j'-j)\cdot my(i'-j') + qyz(i'-i,j'-j)\cdot mz(i',j')]$$

$$H_z(i,j) = \sum_{i'=1}^{N}\sum_{j'=1}^{N}[qxz(i'-i,j'-j)\cdot mx(i',j') + qyz(i'-i,j'-j)\cdot my(i'-j') + qzz(i'-i,j'-j)\cdot mz(i',j')]$$

ここで i と j は粒子のインデックスであり，N は計算点の数である．これにより，一つの粒子が受けた他の粒子からの静磁界の値を計算する．

すべての計算点で静磁界を求める計算を行うには，$qxx(i,j)$，$qyy(i,j)$，$qzz(i,j)$，$qxy(i,j)$，$qxz(i,j), qyz(i,j)$の静磁界係数が必要となる．

静磁界係数は[2]を参考して，以下の式を用いた．

$$\mathrm{qxx(I,J,K)} = \mathrm{M}\sum_{\mathrm{i}=0}^{1}\sum_{\mathrm{j}=0}^{1}\sum_{\mathrm{k}=0}^{1}(-1)^{\mathrm{i+j+k}}\tan^{-1}\left(\frac{(\mathrm{K}+\mathrm{k}-\frac{1}{2})(\mathrm{J}+\mathrm{j}-\frac{1}{2})\mathrm{dzdy}}{\mathrm{r_{ijkP}}(\mathrm{I}+\mathrm{i}-\frac{1}{2})\mathrm{dx}}\right)$$

$$\mathrm{qyy(I,J,K)} = \mathrm{M}\sum_{\mathrm{i}=0}^{1}\sum_{\mathrm{j}=0}^{1}\sum_{\mathrm{k}=0}^{1}(-1)^{\mathrm{i+j+k}}\tan^{-1}\left(\frac{(\mathrm{K}+\mathrm{k}-\frac{1}{2})(\mathrm{I}+\mathrm{i}-\frac{1}{2})\mathrm{dzdx}}{\mathrm{r_{ijkP}}(\mathrm{J}+\mathrm{j}-\frac{1}{2})\mathrm{dy}}\right)$$

$$qzz(I,J,K) = M\sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1}(-1)^{i+j+k}\tan^{-1}\left(\frac{(I+i-\frac{1}{2})(J+j-\frac{1}{2})dxdy}{r_{ijkP}(K+k-\frac{1}{2})dz}\right)$$

$$qxy(I,J,K) = -M\sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1}(-1)^{i+j+k}\log\left((K+k-\frac{1}{2})dz + r_{ijkP}\right)$$

$$qxz(I,J,K) = -M\sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1}(-1)^{i+j+k}\log\left((J+j-\frac{1}{2})dy + r_{ijkP}\right)$$

$$qyz(I,J,K) = -M\sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1}(-1)^{i+j+k}\log\left((I+i-\frac{1}{2})dx + r_{ijkP}\right)$$

$$r_{ijkP} = \sqrt{(I+i-\frac{1}{2})^2dx^2 + (J+j-\frac{1}{2})^2dy^2 + (K+k-\frac{1}{2})^2dz^2}$$

静磁界の計算に必要な計算量を見積もる．計算領域全体にある計算は N 個あり，各計算点における静磁界を求めるには N に比例する回数の乗算が必要であるので，計算量は計算点の個数の 2 乗に比例する．

2.4 シミュレーションプログラムの流れ

シミュレーションプログラムで[図 2.4]は，シミュレーションと関連するパラメータを初期化してからシミュレーション計算ループを始める．CPU あるいは GPU で LLG 方程式の数値計算を行い,それぞれの粒子の計算結果を算出する．そして，4 次ルンゲクッタ法を利用するため， 1 フレームの計算では，4 回の LLG 方程式を計算する必要があり，4 回の計算結果をそれぞれ K1，K2，K3，K4 とする．最後の計算結果 m はルンゲクッタ法により得られる．

計算点数を N とすると静磁界の計算量の影響で，メインループの計算量は $O(N^2)$となる．静磁界の計算は，等間隔のグリッド上にあることから FFT(fast Fourier transform)を用いて高速に計算できる．しかし本研究では使用していない．GPU を用いた場合は計算点数が少ない場合に FFT のメリットが出にくいこと．Compute shader では FFT ライブラリが使えず，プログラム開発に時間がかかることが理由である．

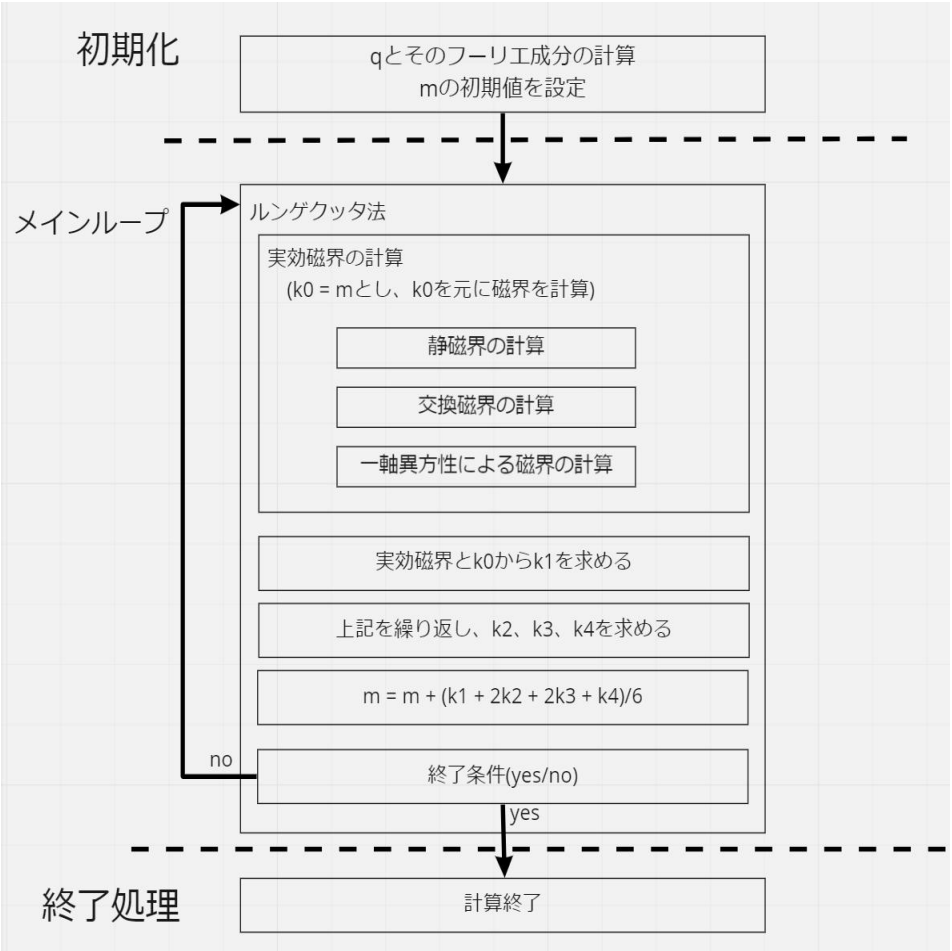


図 2.4:シミュレーションプログラムの流れ

第 3 章 GPU と GPGPU

本章では GPU と GPGPU について記載する.

3.1 GPU

Graphics Processing Unit（グラフィックス プロセッシング ユニット，略して GPU）[2]はリアルタイムレンダリングに特化した並列プロセッサである. GPU はリアルタイムレンダリングの大量かつ同じような計算に特化し，パイプラインの並列構造になっている. 最近の高性能な GPU は頂点処理とピクセル処理ためのプログラマブルシェーダーユニットだけではなく，人工知能ための TENSOR コアやレイトレーシングための RT コアなども搭載されている.

GPU は主に NVIDIA, AMD, ARM などのブランドがある. GPU で描画をするのに, OpenGL, Dirtect3D, VLUKAN などグラフィック API を利用することができる. これらの API により, GPU に命令を送ったり shader プログラムを送ったりすることで GPU をコントロールできる.

近年, GPU はリアルタイム 3DCG の処理だけではなく，数値的な計算にも利用されており, GPGPU(General-purpose computing on graphics processing units)と呼ばれている.

3.2 GPGPU

3.2.1 概要

GPU の発展に伴い，固定機能パイプラインは捨てられ，プログラマブルステップが増えている. これらの機能を利用することで汎用的な計算も行うことができる.

GPGPU[3]を利用するには，CUDA，Compute shader，OpenCL などいくつかの実現方法がある. 本研究では，Compute shader と CUDA を用いてマイクロマグネティクスシミュレーションを高速化している.

3.2.2 CUDA

CUDA(Compute Unified Device Architecture)[4]は NVIDIA 社が開発した GPGPU を行うための並列計算プラットフォームである. CUDA は最も古い. そのため，関連する資料が多く，数値計算などのライブラリも充実している.

使用するには，レンダリングパイプラインについての知識は不要でレンダリングと独立した計算リソースとして使える．

しかし，CUDA は NVIDIA の GPU だけをサポートしている．さらに，CUDA はいろいろなコマンドがあり，自由度が高いが，コーディングの仕方によって速度差が出やすい．また，描画プログラムと連携するためには，cuda_gl_interop で連携プログラムを構築する必要がある．

3.2.3 Compute shader

計算シェーダー(Compute shader)[5]は，グラフィック API で GPGPU を行うテクノロジーである．Compute shader はレンダリングパイプラインの前段階に存在している．Direct3D，OpenGL，Metal，Vulkan などのグラフィック API にそれぞれ異なるものが導入されている．

Compute shader はグラフィック API(OpenGL，Direct3D など)に含まれているため，使用するときは他の GPGPU ツールのようにあらかじめ実行環境を構築する必要がなく，グラフィック API を呼び出すだけで利用できる．

さらに，Compute shader はすでにゲームエンジンの一部となっているため，Unity からは C#スクリプトを駆使すれば Compute shader と連携することができる．

3.2.4 OpenCL

本研究では用いていないが，OpenCL[6]も GPGPU を実現する１つの方法である．OpenCL は CPU，GPU，FPGA などのデバイスで利用可能かつマルチプラットフォーム対応の並列計算プログラミングモデルである．主に科学的計算や画像処理などの場面でハイパフォーマンスコンピューティングのために利用される．シミュレーション可視化で使う場合にもグラフィック API と連携し描画できる．

表 3.1：GPGPU API の比較

	CUDA	Compute shader	OpenCL
ドライバーをインストール必要	ない	ある	ある
GPU ブランド依存	なし	NVIDIA	なし

第 4 章　可視化と Unity

本章では，可視化一般と本文で可視化するために利用されるゲームエンジン Unity について説明する．

4.1 可視化と科学的可視化

可視化(Visualization)[7]は，人が見えないデータの関連性を見えるように，データをグラフィック技術で処理し，見えるものにする（画像・グラフ・図・表など）ものである[図 4.1]．本論文のマイクロマグネティクスシミュレーションの可視化は可視化カテゴリーの科学的可視化[8]に属する．

科学的可視化は科学のデータの関連性を見えるように，可視化を行うものである．目標は研究者が科学のデータから関連性を分析できるように，グラフィック技術で科学データをグラフィカルに解釈することである．また，科学現象をより観察しやすいように，リアルタイム可視化は有効である．リアルタイム可視化はリアルタイムレンダリング API(OpenGL, Direct3D など)あるいは関連するツールを通じて実現できる．

本論文では，ゲームエンジン Unity のリアルタイムレンダリングパイプラインを通じて，リアルタイム科学的可視化の描画プログラムを実装する．ゲームエンジン Unity を使えば，OpenGL，Direct3D などのグラフィック API で可視化プログラムを作成するのと比べ，少ない手間で可視化システムを構築することができる．

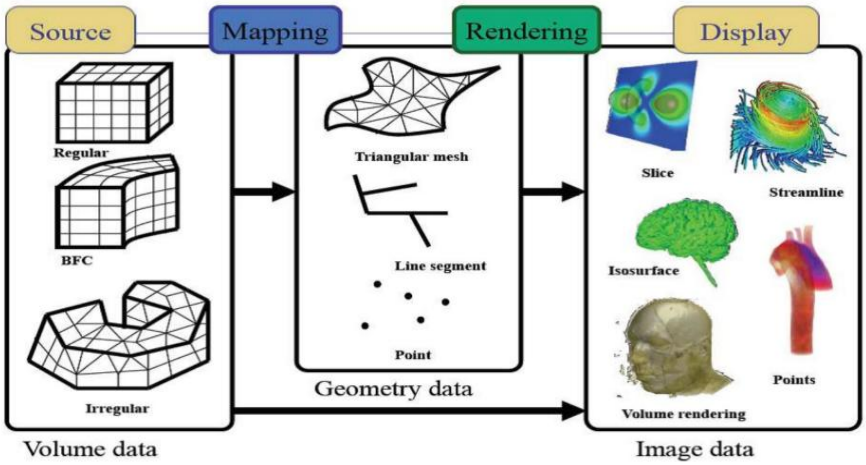


図 4.1：ボリューム可視化(参考文献[1]，図 3.1)

4.2 ゲームエンジンと Unity

ゲームエンジン[9]は，コンピュータゲームを作るための統合開発環境ソフトウェアである．最も重要な機能はリアルタイムレンダーだが，ゲームの進化に伴い，ゲームエンジンも進化している．現存ではネットワーク，スクリプト，オーディオ，アニメーション，UI，AI，MR など様々な機能が付随されている．また，ゲームだけではなく，映像やインタラクティブアートなどの場面にも広く使用される．

現在の主な市販および外部提供エンジンは Unreal エンジン，Unity エンジン[10]，CRYENGINE エンジンなどがある．これらのエンジンのうち，Unity は比較的にシンプルで，関連する情報が多く，モバイル端末での開発によく使われている，このため，本研究では，Unity を採用する．

Unity にはゲームエンジンとしてのリアルタイムレンダー，スクリプト，オーディオ，アニメーション，UI が搭載されている．また，マルチプラットフォーム対応しており，Unity で開発されたアプリケーションはデスクトップパソコンだけではなく，モバイル端末，ブラウザ上でも実行できる．Unity 自体は C/C++で作られているが，開発するにあたっては主に C#を用いてスクリプトを書くことが多い．C#スクリプトは C/C++での開発と比べて，より少ない手間で開発することができる．

4.3 レンダリングパイプラインと Shader

レンダリングパイプライン(rendering pipeline)[11]は，ある三次元のシーンのデータ(オブジェクト，シェーダーなど)を二次元の画像に変換する過程である．この過程はパイプライン化により高速，並列に行われている．

レンダリングパイプラインは図 4.2 のように，いくつかのステージがある．青色のステージはプログラマブルステージであり，これらのステージ（主に頂点シェーダーとフラグメントシェーダー）で実行されるプログラムをシェーダー(shader)[12]と呼ぶ．開発者はシェーダープログラムを書くことで，自由にこれらのステップをコントロールすることができる．

アプリケーションステージでは三次元シーンのデータを CPU メモリー(Random Access Memory)から GPU メモリー(Video Random Access Memory)に転送する．一フレームに対し，カメラのパラメータ，シーンのデータなどをレンダリングパイプラインに送る必要がある．

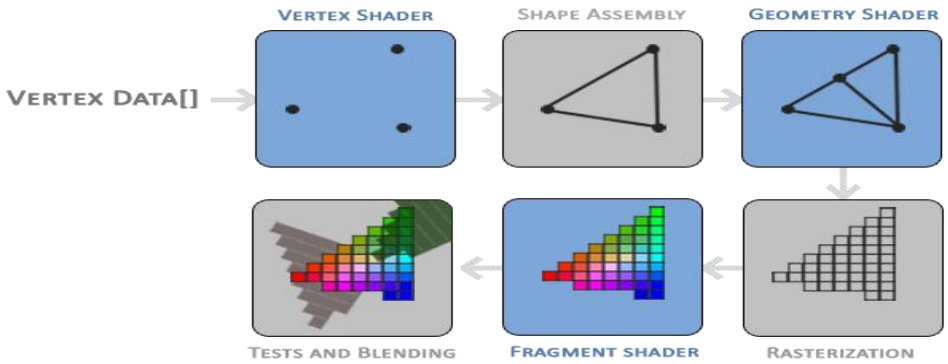


図 4.2:リアルタイムレンダリングパイプライン(参考文献[13])

頂点シェーダーステージではすべての頂点に対して，頂点シェーダーが実行される．頂点シェーダーでは，頂点に対し，座標変換処理と他の処理が行われる．
ラスタライズステージでは頂点データを入力し，頂点の間のピクセルを出力する．
フラグメントシェーダーステージではラスタライズステージからもらったピクセルを処理する．

第 5 章　既存研究

5.1　リアルタイムシミュレーションと可視化

Unity と Compute shader でクロスシミュレーションを行う研究が存在している[14]．この研究では，Compute shader を用いてばね質量ダンパー系(Mass-Spring System)ベースのリアルタイム布シミュレーションアプリケーションを開発した．Unity エンジンでリアルタイムでシミュレーションとレンダリングされ，AR/VR アプリケーションで利用できる．本研究も Unity と Compute shader でリアルタイムシミュレーションとレンダリングを行うが，シミュレーション対象はマイクロマグネティクスである．

5.2　CUDA と Compute shader の比較

Compute shader は CUDA に負けない速度をもち，ボリュームレイキャスティングで CUDA より速いと報告されている[15]．ボリュームレイキャスティングはボリュームレンダリングの主な手法であり，医療，物理シミュレーションなどの分野で利用される．この研究では，ボリュームレイキャスティングに対し，OpenGL のフラグメントシェーダー，OpenGL の Compute shader，OpenCL，CUDA で実装した．パフォーマンスにおいて Compute shader は最も良い結果を示している．フラグメントシェーダーより 1.03~2.9 倍，OpenCL より 1.5~1.7 倍，CUDA より 1.01~1.3 倍速い．しかし，この研究では，計算誤差を示しておらず，懸念するところがある．本研究では，それぞれ Compute shader と CUDA を用いて計算を加速し，相対誤差を測定する．

5.3　CUDA によるマイクロマグネティクスシミュレーション

CUDA でマイクロマグネティクスシミュレーションを行い，グラフィック API(OpenGL)を用いて可視化を行っている研究は存在している[1]が OpenGL で可視化プログラムを構築するのは手数がかかる．また，CUDA は NAVIDA の GPU だけをサポートし，別のブランドの GPU を使ったり，スマートフォンを使うと動作しないという問題がある．本研究では，プラットフォーム対応を求め，CUDA ではなく Compute shader でマイクロマグネティクスシミュレーションアルゴリズムを加速するリアルタイムシミュレーション可視化システムを提案する．

第 6 章　Unity による可視化システムの開発

本章ではマイクロマグネティクスシミュレーションのリアルタイム可視化システムの開発について説明する．このシステムでは GPU あるいは CPU でリアルタイムにシミュレーションを計算し，計算結果を 3 次元の矢印モデルとして可視化する[図 6.1].

6.1　本システムの概要

本システムのシミュレーション対象は第 2 章で述べたマイクロマグネティクスシミュレーションである．本システムは主にシミュレーション計算プログラムと可視化プログラムがある[図 6.2].

シミュレーションプログラムでは，C#スクリプトで CPU ベースのシミュレーションプログラムあるいは Compute shader プログラムで GPU ベースのシミュレーションプログラムを実行し，計算する．Compute shader を選ぶと GPGPU により，シミュレーションを高速に行う．

可視化プログラムでは，前述のシミュレーショプログラムで計算された結果を Unity のリアルタイムレンダリングパイプラインに送り，shader プログラムを実行し，それぞれの色と方向が付けられた矢印ポリゴンを描画する．最後に，ポストプロセスプログラムにより，描画された画像を処理し，矢印ポリゴンをより観察しやすくすることができる．

Unity の UGUI でヒューマンインタフェース(略称 UI)を構築した．シミュレーションに関するパラメータの入力，カメラのコントロールなどができる．表ＸＸは開発環境である．

表 6.1：システムの開発環境

OS	Windows 10 x64
Unity	Unity 2020.3.24f1
CPU	Intel(R)Core(TM)i7-4720HQ　2.60Ghz
GPU	NVIDIA GeForce GTX 970M
CUDA	CUDA 11.6
RAM	16.0GB

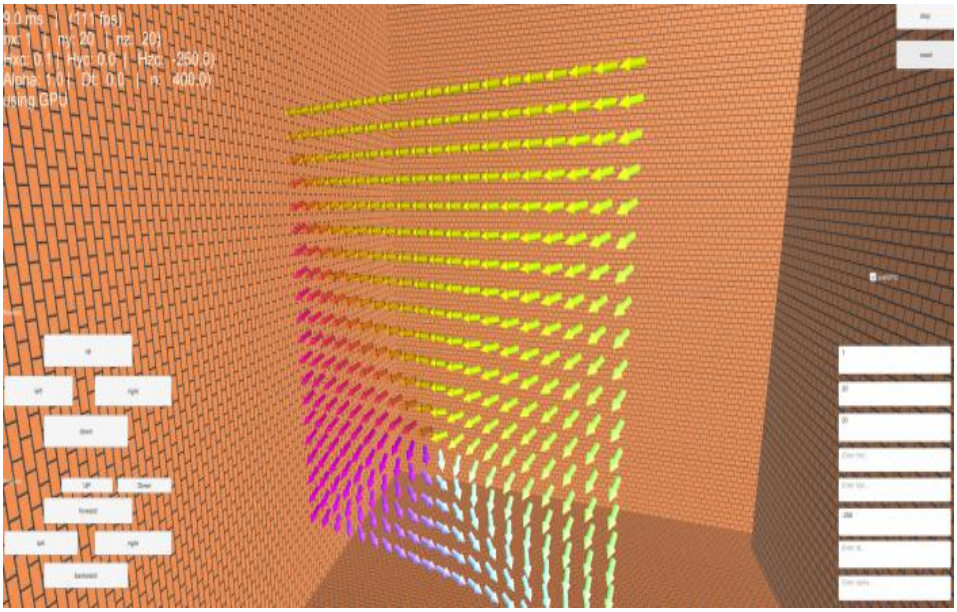


図 6.1　可視化システムの外見

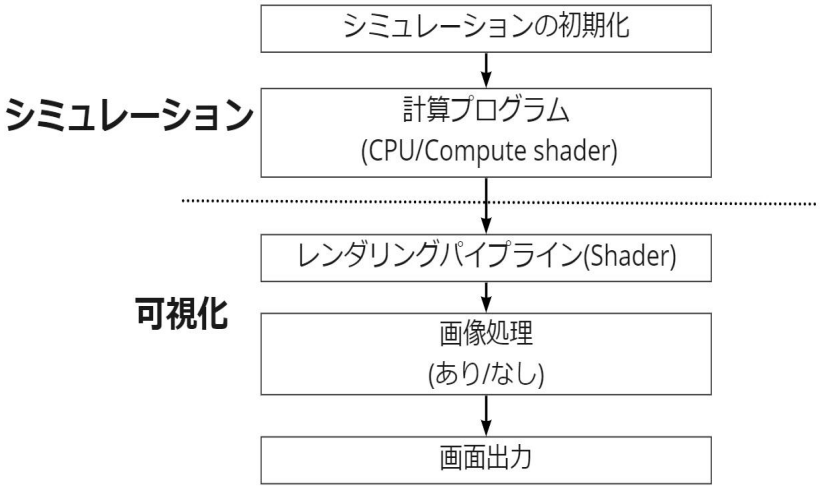


図 6.2　システムの構成

6.2 シミュレーション計算プログラム

シミュレーション計算プログラムは C#スクリプトあるいは Compute shader で第２章のマイクロマグネティクスを計算する.

図 6.3 は Compute shader を使う場合の計算の流れを示している.

Compute shader で計算する場合は GPU メモリーとして ComputeBuffer を使う必要がある. シミュレーションの初期化されたデータはメモリーから ComputeBuffer に転送される. shader と Compute shader プログラムでは, ComputeBuffer は StructuredBuffer<T>の形で渡されている.

フレームごとに, Compute shader の結果は ComputeBuffer に格納され, レンダリングパイプラインに渡り, 描画プログラムの入力として使われる.

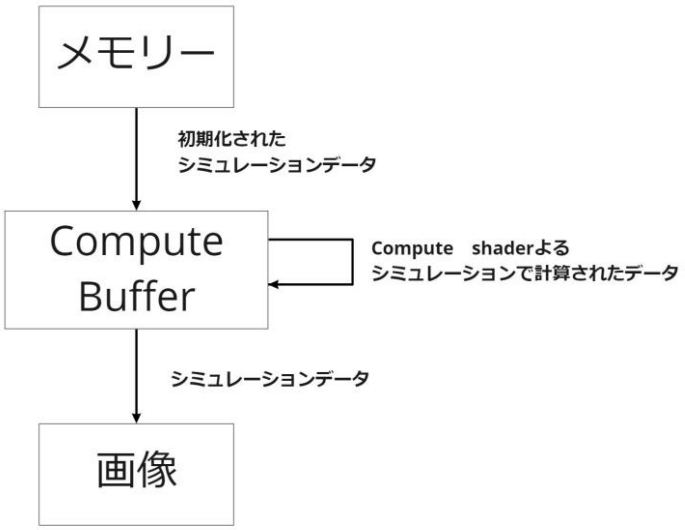


図 6.3：シミュレーション計算プログラムのデータの流れ

6.3　描画プログラム

描画プログラムはリアルタイムレンダリングパイプラインと関連するプログラムである．シミュレーション計算プログラムで計算された結果の数値(以下は VEC と称する)をレンダリングパイプラインに渡し，3 次元のベクトルのデータを矢印ポリゴンに変換し，最後に 2 次元の画像の形式でディスプレイに表示する．描画プログラムの主な処理を[図 6.4]に示す．VEC により変換行列と色を計算し，Phong シェーディングモデルにより最終的な色を計算し 2 次元の画像を表示する．また，2 次元の画像を表示する前に画像処理を行うことができる．

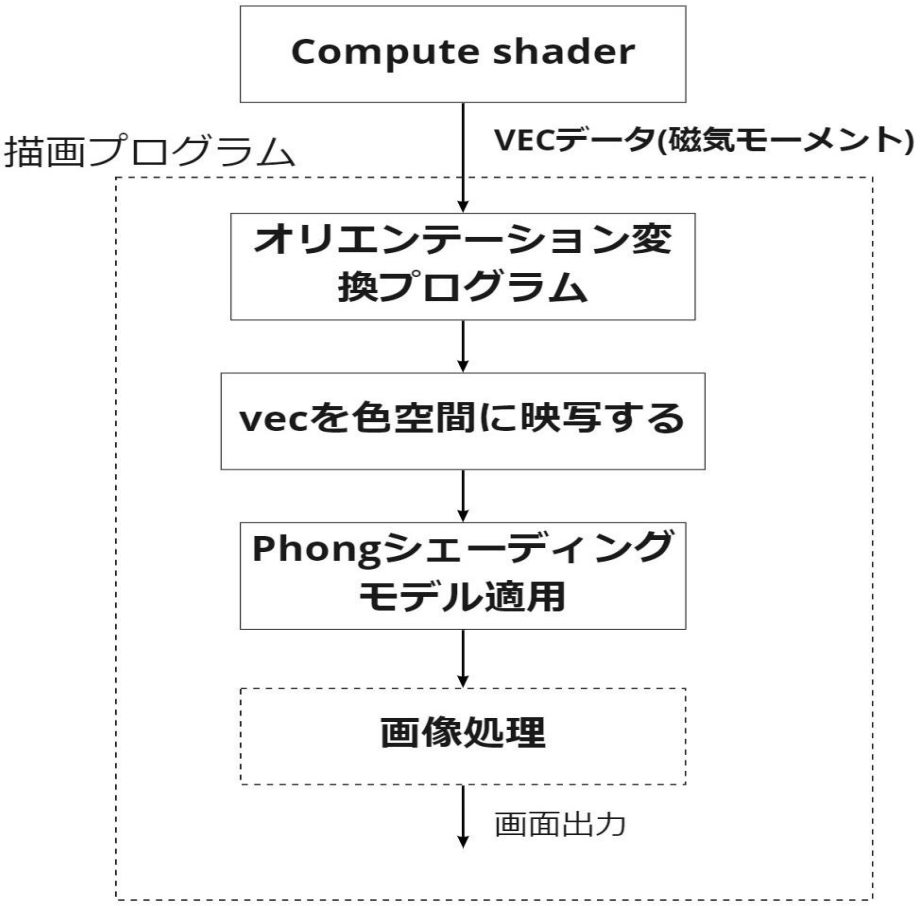


図 6.4：可視化プログラムの処理の流れ

6.3.1 オリエンテーション変換

回転行列とはユークリッド空間内における原点中心の回転変換の表現行列のことである[図 6.5]．回転行列についての計算は頂点 shader で行う．粒子ごとのベクトルVEC_{xyz}値により，粒子の方向を計算する．

まず，一つの粒子の方向は，三つの軸に関する回転によって決める．
以下の式の Pitch，Yaw，Roll は[図 5.4.1]に示す回転方向である．

Pitch: $R_z = \begin{bmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Yaw: $R_y = \begin{bmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{bmatrix}$

Roll: $R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{bmatrix}$

三つの方向の回転行列R_x，R_y，R_z を一つの行列 R にまとめて

$R = R_z\ R_y\ R_x$

$$= \begin{bmatrix} cos\gamma cos\beta & -sin\gamma cos\alpha + cos\gamma sin\beta sin\alpha & sin\gamma sin\alpha + cos\gamma sin\beta cos\alpha \\ sin\gamma cos\beta & cos\gamma cos\alpha + sin\gamma sin\beta sin\alpha & -cos\gamma sin\alpha + sin\gamma sin\beta cos\alpha \\ -sin\beta & cos\beta sin\alpha & cos\beta cos\alpha \end{bmatrix}$$

が得られる．

そして，粒子の位置($position_{xyz}$)と粒子の大きさ(size)に関する行列を回転行列 R に加えると

粒子の空間行列 E =

$$\begin{bmatrix} size * cos\gamma cos\beta & -sin\gamma cos\alpha + cos\gamma sin\beta sin\alpha & sin\gamma sin\alpha + cos\gamma sin\beta cos\alpha & position_x \\ sin\gamma cos\beta & size * (cos\gamma cos\alpha + sin\gamma sin\beta sin\alpha) & -cos\gamma sin\alpha + sin\gamma sin\beta cos\alpha & position_y \\ -sin\beta & cos\beta sin\alpha & size * cos\beta cos\alpha & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

が得られる．

粒子の空間行列 E のパラメータについてはシミュレーションの計算値VEC_{xyz}と粒子の大きさ，位置により決める．

表 6.2　回転行列 E のパラメータ

γ	Pitch 回転と関連し，VEC_zで算出
β	head 回転と関連し，VEC_x，VEC_yで算出
α	roll 回転は意味ないため，定数に設定
size	矢印モデルの大きさ
$position_x$，$position_y$，$position_z$	矢印モデルの位置

粒子の空間行列 E により，一つ一つの粒子は自分のパラメータにより行列が計算され，3D 空間上の位置と大きさ，方向が決められる．以上の処理で，シミュレーションした結果VEC_{xyz}が方向などの空間上の情報に転換される．

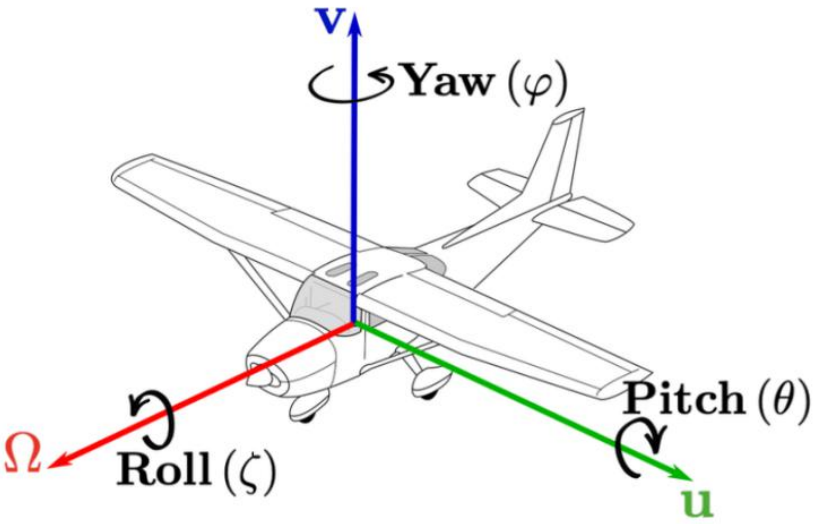


図 6.5：Pitch，Yaw，Roll 回転概要図(参考文献 16)

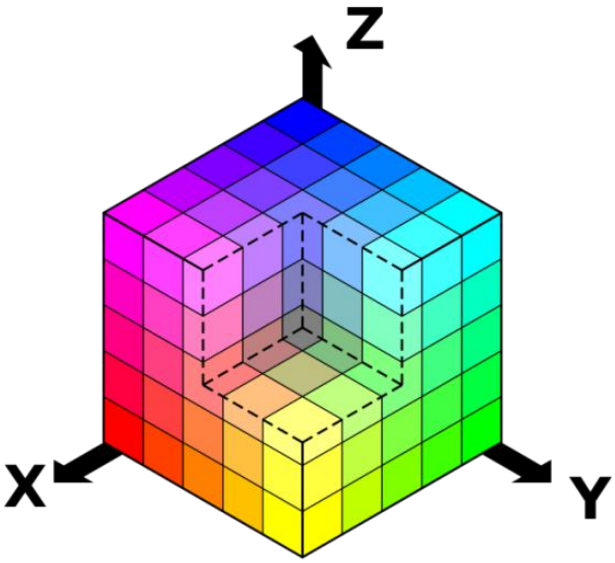


図 6.6　数値による色付け(参考文献 17)

6.3.2　色空間

粒子ごとのVEC_{xyz}値により，色を計算する．頂点 shader で計算する．
シミュレーション計算プログラムで計算された結果の値VEC_{xyz}は[-1，　1]の範囲にあり，色空間[0，　1]に変換する必要がある．以下の式はその変換式である．

$$\mathrm{Color_x} = \frac{1}{2} \times \mathrm{VEC_x} + \frac{1}{2}$$

$$\mathrm{Color_y} = \frac{1}{2} \times \mathrm{VEC_y} + \frac{1}{2}$$

$$\mathrm{Color_z} = \frac{1}{2} \times \mathrm{VEC_z} + \frac{1}{2}$$

これにより，VEC_{xyz}は図 6.6 の色空間($Color_{xyz}$)に映写することができる．

6.3.3　Phong シェーディングモデル[図 6.7]

立体感をもたらすように，描画プログラムでは Phong シェーディングモデルを実装する．Phong シェーディングモデルは ambient と diffuse，specular の三つの部分があり，計算は頂点 shader あるいはピクセル shader で行う．本システムでは ambient は 5.4.2 の色を使い. diffuse と specular は頂点シェーダーで計算する．

まず，Phong シェーディングモデルの diffuse 反射の式で計算する．

$$c_{diffuse} = (c_{light} \cdot m_{diffuse})\, max(0, \hat{n} \cdot I)$$

ここで，$c_{diffuse}$ は計算結果，c_{light} はライトの色，$m_{diffuse}$ はマテリアル(モデル)の色．$\hat{n}$ は表面法線ベクトル，I はライト方向ベクトルである．$\hat{n}$ と I の内積により diffuse の効果をシミュレートする．

$max(0, \hat{n} \cdot I)$は内積の計算結果がマイナスになるのを防ぐ．

Unity ではmax関数の代わりに，同じ機能の saturare 関数を利用し下記のように記述する．

$$c_{diffuse} = c_{light} * m_{diffuse} * \text{saturare}(\text{dot}(\hat{n}, \quad I))$$

鏡面反射部分(Specular)は以下のように計算する．

$$c_{specular} = (c_{light} \cdot m_{specular}) \max(0, \hat{v} \cdot r)^{m_{gloss}}$$

Unity での擬似コードに書き換えると，以下の式になる:

$$c_{specular} = c_{light} * m_{specular} * \text{pow}(\text{saturare}(\text{dot}(\hat{v}, r), m_{gloss})$$

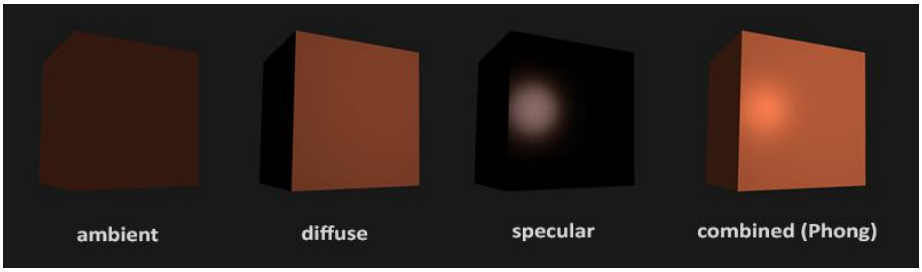


図 6.7 Phong シェーディングモデル[参考文献 13]

最後に，三つの部分を足し合わせて最終的な色を求める．

$$\text{Color} = C_{ambient} + C_{diffuse} + C_{specular}$$

6.4 ジオメトリインスタンシング

本システムでは同じ形の矢印物体を向きや位置，色を変えながら多数描画することになる．多くの Draw call を行うよりは，一回の Draw call で複数の同じ形のオブジェクトを描画すれば効率が良い．これをジオメトリインスタンシングと呼ぶ手法で実現する．

Unity では, 描画機能クラス Graphics に DrawMeshInstanced 関数としてジオメトリインスタンシング機能がサポートされている．本システムでは，回転行列を使って大量の矢印モデルを描画する．

6.5 画像処理

画像処理は２次元の画像データの処理に関する技術のことである．Unity では，このような処理をポストプロセスとも呼ばれる．ポストプロセスはまず頂点 shader で画像の 4 つの頂点に対し処理する．そして，ピクセル shader で画像のピクセルごとに処理する．

本節では，Unity でポストプロセスを行う仕組みとエッジ検出と画像の調整の画像処理について記述する．

6.5.1 概要

Unity でポストプロセスを行う[図 6.8]には，レンダリングパイプラインに出力された画像を保存し，再びレンダリングパイプラインに送り，画像を 1 個の四角形として(4 頂点)，shader プログラムにより並列処理される．処理される前の画像は src，処理された画像は dest に格納される．画像処理のコアの部分は shader で実装されており，OnRenderImage 関数が呼ばれると，blit 関数や関連する shader が実行される．

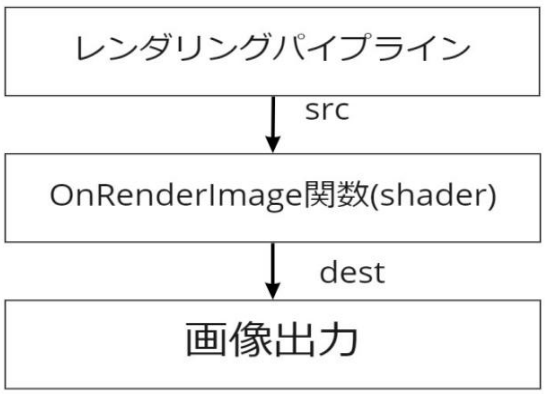


図 6.8：Unity ポストプロセスの流れ

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1

図 6.9　Soble 演算子(参考文献[18])

6.5.2 エッジ検出

通常，辺は色の変化が激しいところである．つまり，辺となるピクセルは色の値の勾配が大きところだと考えられる．Soble 演算子[図 6.9]により，畳み込み(convolution)操作を通じて，勾配を計算し，エッジ検出する．

計算により，縦方向と横方向の勾配が求められて，以下の式により全体の勾配を求める．

$$G = |G_x| + |G_y|$$

ここで，G は勾配の和，G_x は横方向の勾配，G_y は縦方向の勾配である．

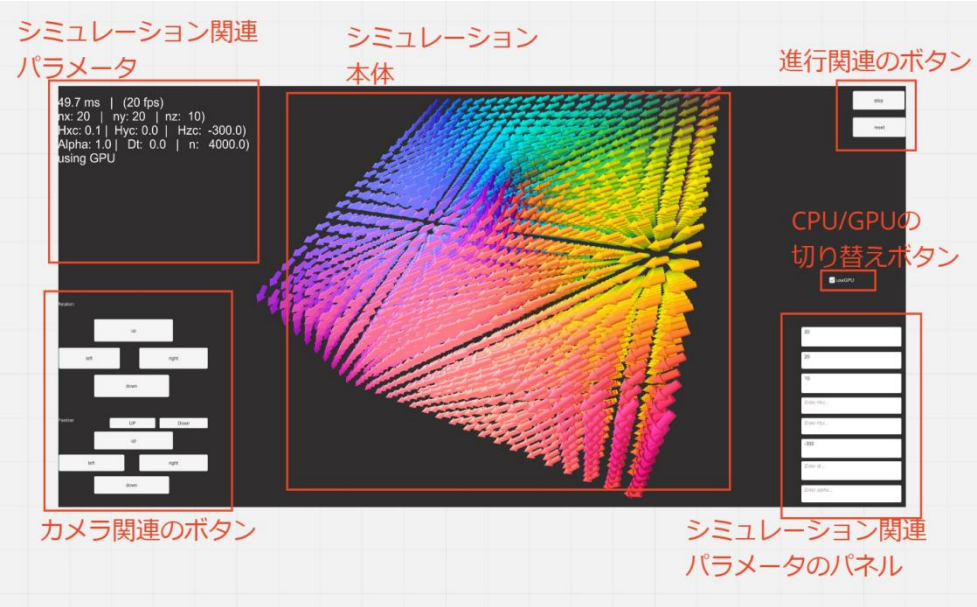


図 6.10　ヒューマンインタフェース概要図

6.6 本システムの機能概要と評価

本節では，本システムのヒューマンインタフェースでシミュレーションをコントロールする結果について述べる．

ヒューマンインタフェースでは，図 6.10 のように，いくつかの部分がある．

・シミュレーションと関連するパラメータを表示するパネル
・カメラの位置と向きをコントロールするボタン
・シミュレーションの進行と関連するボタン
・CPU あるいは GPU でシミュレーションを計算するかの切り替えボタン
・シミュレーションと関連するパラメータを書き換えるパネル

6.6.1 操作パネル

本システム[図]では，ヒューマンインタフェースより，ボタンをタッチすることでコントロールできるようにしている．パソコンで本システムを実行する場合はキーボードでカメラをコントロールできる．表 6.3 はカメラコントロールのためのキーをまとめている．

6.6.2 右上パネル

右上のパネルでは，stop ボタンでシミュレーションをストップし，reset ボタンでシステムをリセットできる．

表 6.3 カメラコントロールキー

カメラの位置	
W	前進
S	後退
A	左に移動
D	右に移動
Q	上に移動
R	下に移動
カメラの向き	
カーソル左	左に回転
カーソル右	右に回転
カーソル上	上に回転
カーソル下	下に回転

6.6.3 左下パネル

左下のパネル[図 6.11]のボタンでカメラをコントロールできる．

Rotation 部分では up，down，left，right ボタンを押すことで，カメラの向きを上，下，左，右に回転できる．

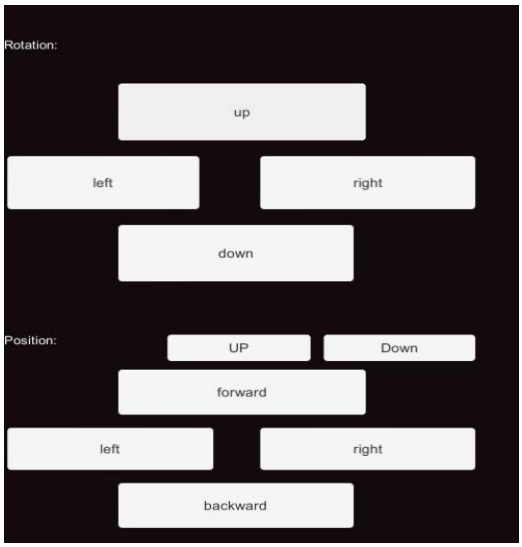
position 部分では forward，backward，left，right ボタンを押すことで，カメラの位置を前，後，左，右に移動できる．さらに，UP と Down ボタンを押すことで，カメラを上と下に移動できる．

6.6.4 右下パネル

右下の操作パネル[図 6.12]により，シミュレーションに関連するパラメータを入力することができる[表 6.4].

表 6.4 シミュレーションパラメータ

Enter Nx	x 方向の粒子数
Enter Ny	y 方向の粒子数
Enter Nz	z 方向の粒子数
Enter Hxc	x 方向の外磁場
Enter Hyc	y 方向の外磁場
Enter Hzc	z 方向の外磁場
Enter dt	時間ステップ
Enter alpha	損失定数



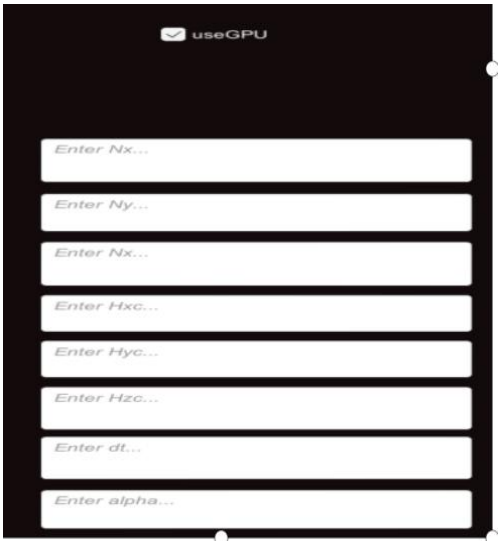


図 6.11　左下操作パネル　　図 6.12　右下操作パネル

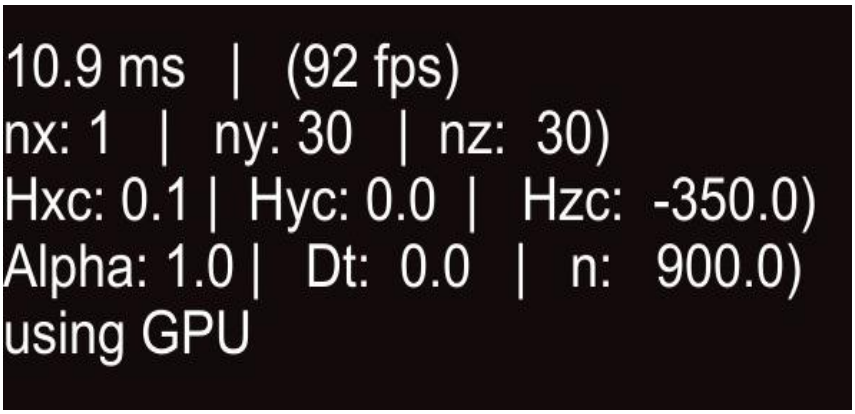


図 6.13 文字情報の画面出力

6.6.5 シミュレーションパラメータを表示するパネル

右下パネルで入力したパラメータは文字情報の画面出力[図 6.13]に反映される.

表 6.5：文字情報の意味

第１行目は１フレームの計算時間(ms)とフレームレート
第２行目は x 軸方向の粒子数 nx，y 軸方向の粒子数 ny，z 軸方向の粒子数 nz
第３行目は xyz 軸の外部磁界 Hxc，Hyc，Hzc
第４行目は損失定数 Alpha，時間ステップ dt，粒子数 n
第５行目はシミュレーションプログラムが使っている計算デバイス(CPU/GPU)

図 6.14 コンソールのシミュレーション情報表示

シミュレーション中のシミュレーション計算結果などは Unity のコンソール[図 6.14]で表示される.

第 7 章　CUDA ベースシミュレーションシステム開発

本章では CUDA ベースシミュレーションについて記述する．

7.1　概要

本システムは前章で開発した Unity によるリアルタイムシミュレーションの可視化システムと比較評価するために使用する．このシステムでは可視化プログラムは実装されず，単にシミュレーションを GPGPU(CUDA)で加速するシステムである．開発環境は表 6.1 と同じである．

7.2　可視化対象

CUDA ベースシミュレーションシステムの計算対象は第 6 章のシステムと同じく，マイクロマグネティクスである．マイクロマグネティクスシミュレーションについて簡単な問題である．図 7.1 の磁化構造データをシミュレーションする．

磁気モーメントは最初は同じ方向に向かっているが外部磁場の影響により，磁化構造が変化する．

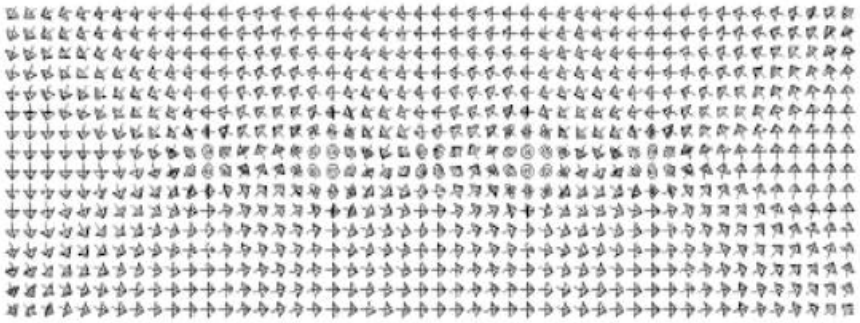


図 7.1　枕木磁壁の磁荷構造(参考文献[1]の図 4.2)

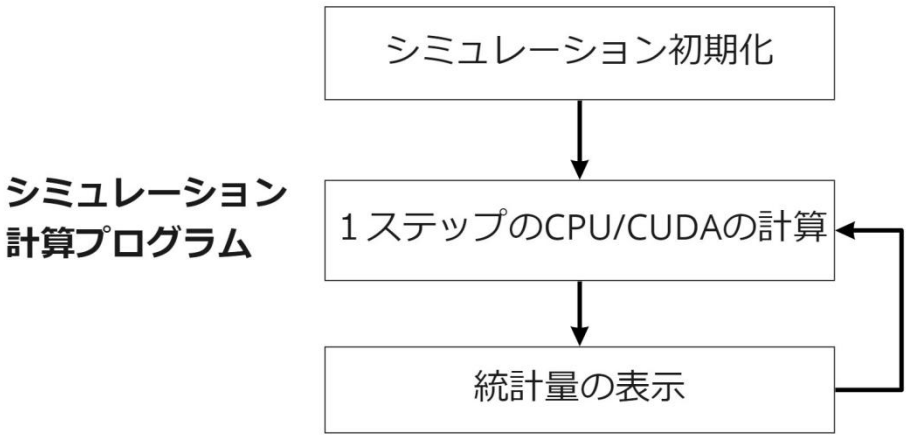


図 7.2　CUDA ベースの計算プログラムの流れ

7.3　計算の流れ[図 7.2]

まずはシミュレーションについてのパラメータを初期化する．そして，シミュレーション計算プログラムにより 1 ステップ計算される.CUDA だけではなく，CPU のみで計算に切り替えることもできる．計算された結果はコンソールに出力され，収束するまでくり返す.

第 8 章 システムの評価

本章では，開発したリアルタイムシミュレーション可視化システムについて評価する．操作パネルで操作できること，シミュレーションが動くこと，マルチプラットフォームで動くこと，描画方法の違い，処理の速度，計算精度について評価する．

8.1 カメラ関連の確認

操作パネルでカメラをコントロールすることで，シミュレーションをい任意の角度で見ることができる．
図 8.1 は水平方向でシミュレーションを観察した画像である．
図 8.2 はカメラを動かし，シミュレーションの内部に入り込んだ様子である．
図 8.3 は上からシミュレーションを観察した様子である．
図 8.4 は下からシミュレーションを観察した様子である．

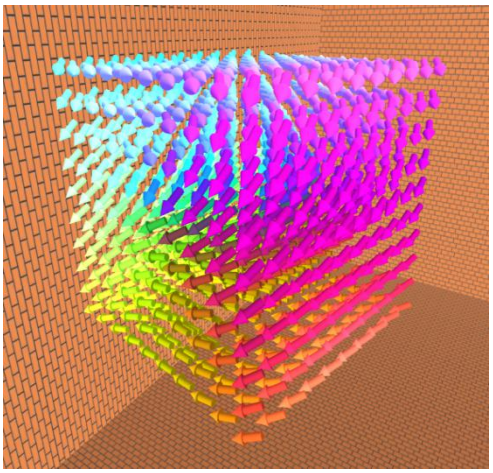


図 8.1 正面

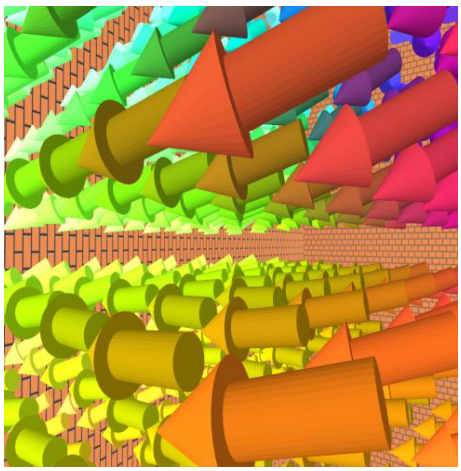


図 8.2 入り込み

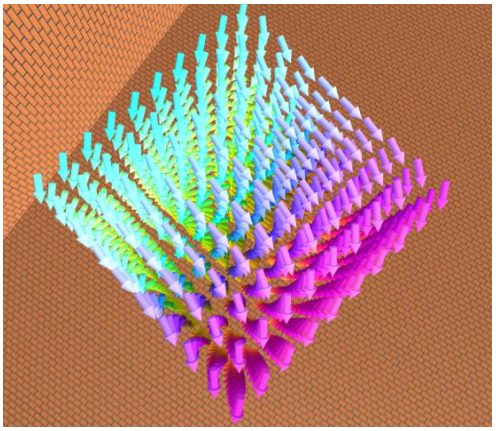


図 8.3 上から観察

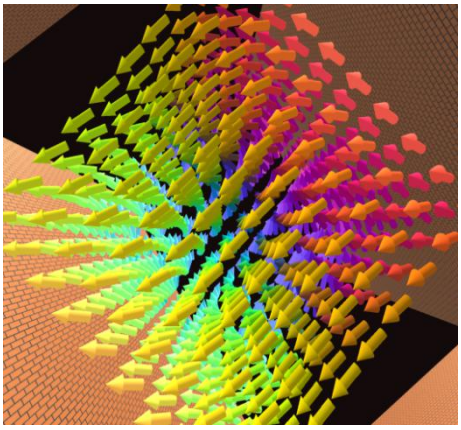


図 8.4 下から観察

8.2　シミュレーション関連の確認

入力パネルで粒子数を変えることができることを確認した.
図 8.5 は(nx:1，ny:4，nz:4)に設定した初期状態である.
図 8.6 は y 軸方向の粒子数を二倍にする画像である
図 8.7 は y 軸方向の粒子数を二倍にする画像である
図 8.8 は y 軸方向の粒子数を二倍にする画像である

ny を倍増すると，粒子ブロックは立て方向に伸ばす.
nz を倍増すると，粒子ブロックは横方向に伸ばす.

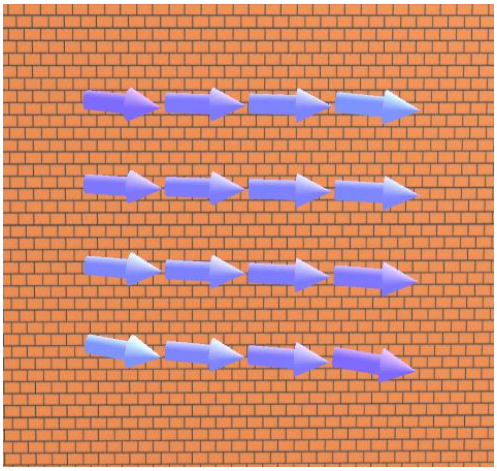


図 8.5　初期状態

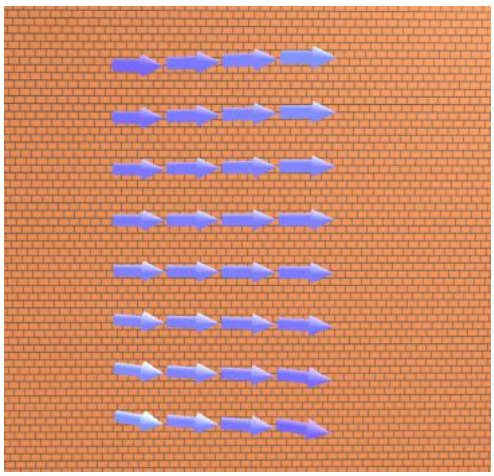


図 8.6　ny を二倍（1x8x4）

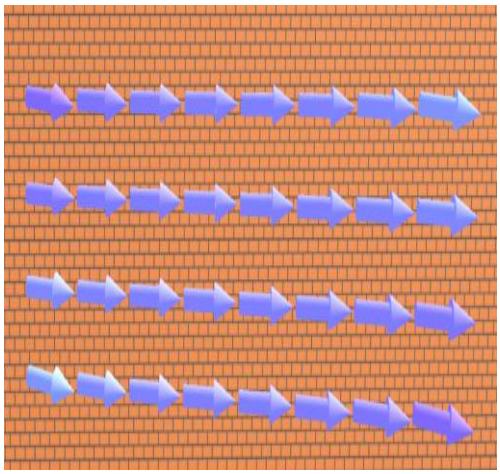


図 8.7　nz を二倍（1x4x8）

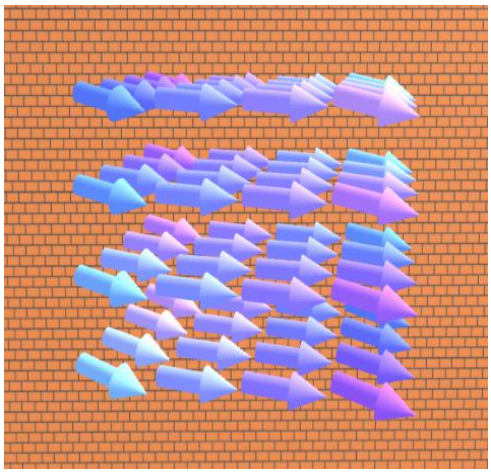


図 8.8　nx を四倍(4x4x4)

8.3　シミュレーション動作の確認

シミュレーションは表 8.1 のパラメータで行った．Z 軸方向に外部磁場がかかっている状況のシミュレーションである．

シミュレーションの初期段階[図 8.9]では，ベクトルははＺ軸方向の正方向に向かっている．時間の経過とともにベクトルは外部磁場の影響により，徐々に逆転する[図 8.10 と図 8.11]．最終的に，すべてのベクトルは逆転した[図 8.12]．

表 8.1：シミュレーションプロパティ

x 軸方向の粒子数　Nx	1
y 軸方向の粒子数　Ny	30
z 軸方向の粒子数　Nz	30
x 軸方向の外部磁場　Hxc	0.1
y 軸方向の外部磁場　Hyc	0
z 軸方向の外部磁場　Hzc	-350
損失定数 α	1
粒子数　N	900

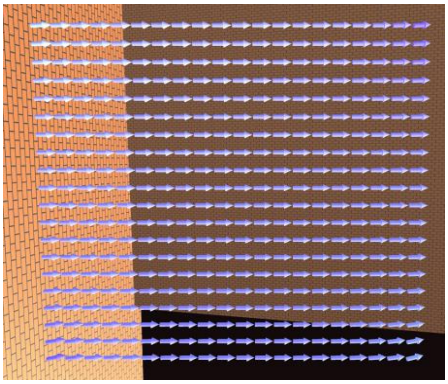


図 8.9 シミュレーション初期状態

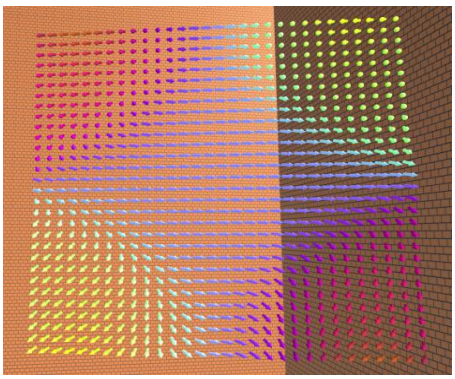


図 8.10　シミュレーション中盤 1

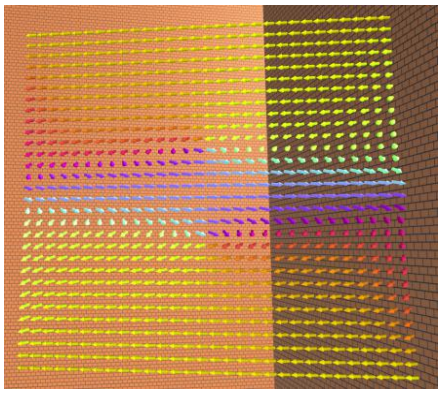


図 8.11　シミュレーション中盤 2

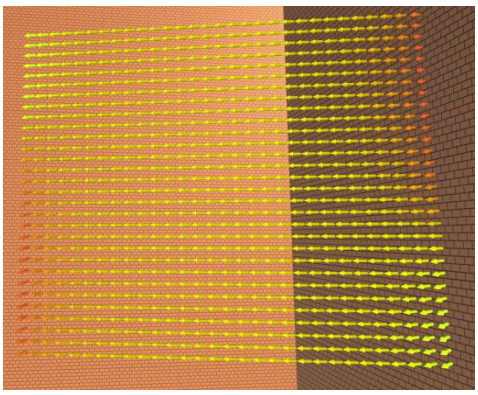


図 8.12　シミュレーション終盤

8.4　マルチプラットフォーム対応の確認

本システムはマルチプラットフォームに対応している.

図 8.13 はパソコンで実行している画像である.

図 8.14 はスマートフォン端末で実行している画像である.

図 8.15 はタブレット端末で実行している画像である.

本システムはあらかじめ GPGPU 実行環境をインストール必要はなく, GPGPU を行うための Compute shader は端末のグラフィックスドライバーに最初から存在している.

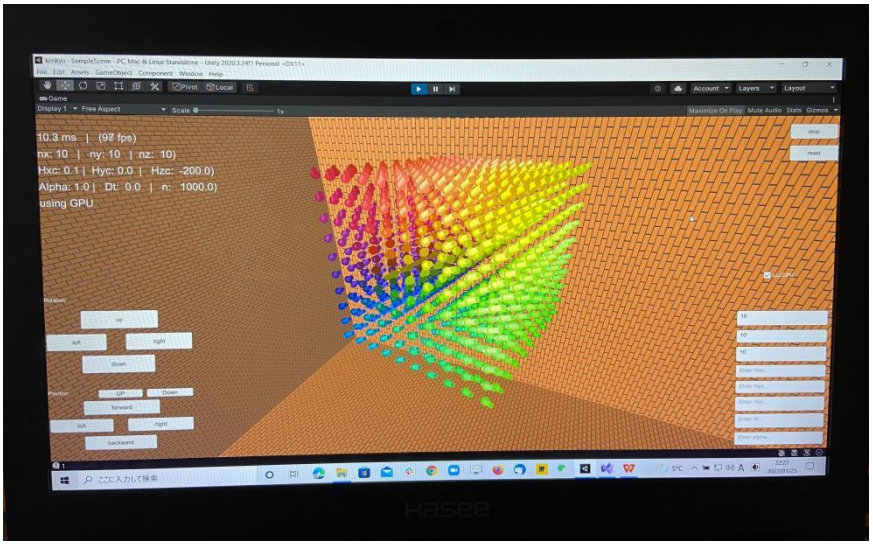


図 8.13 パソコンで実行

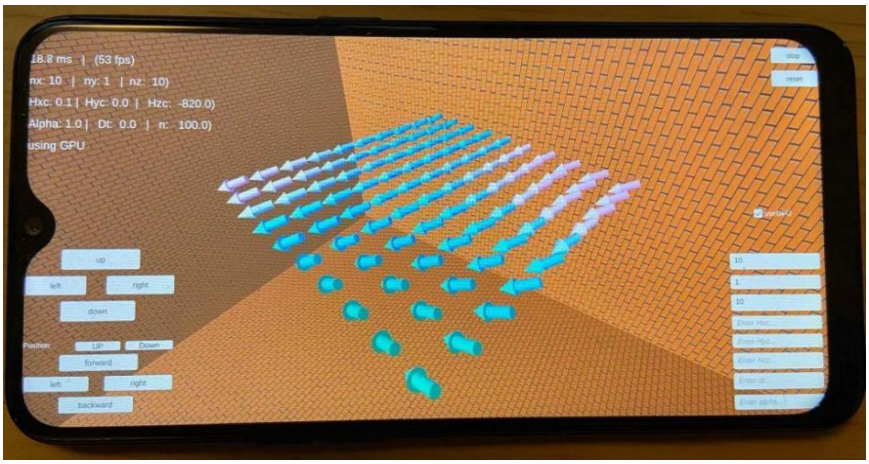


図 8.14 スマートフォンで実行

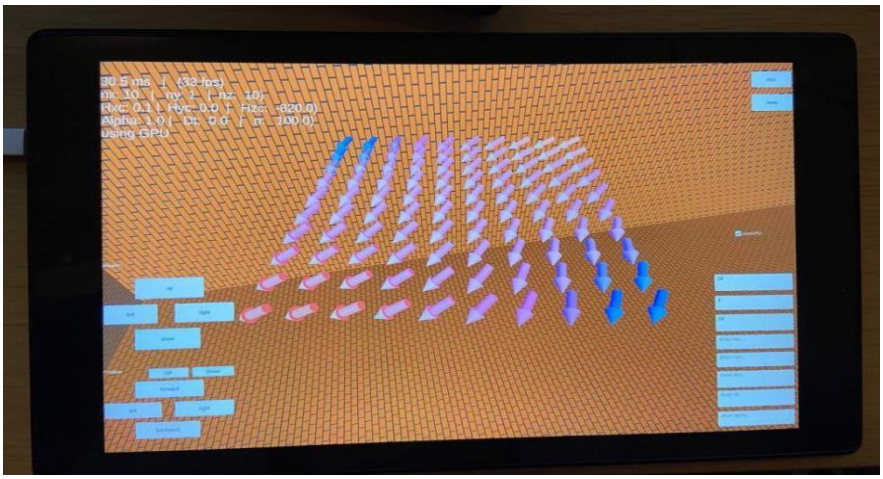


図 8.15 タブレットで実行

8.5 描画プログラムの評価

本システムの描画プログラムについて評価する.

8.5.1 Phong シェーディングモデル

矢印モデルにただ色を塗るだけだと立体感がない問題がある．Phong シェーディングモデルのライティング効果により，矢印の位置がよく分かるようになった[図 8.16].

8.5.2 Unity ポストプロセス

矢印モデルの周りに黒い線が引かれ，矢印モデル間の境界がはっきりとした(図 8.17).

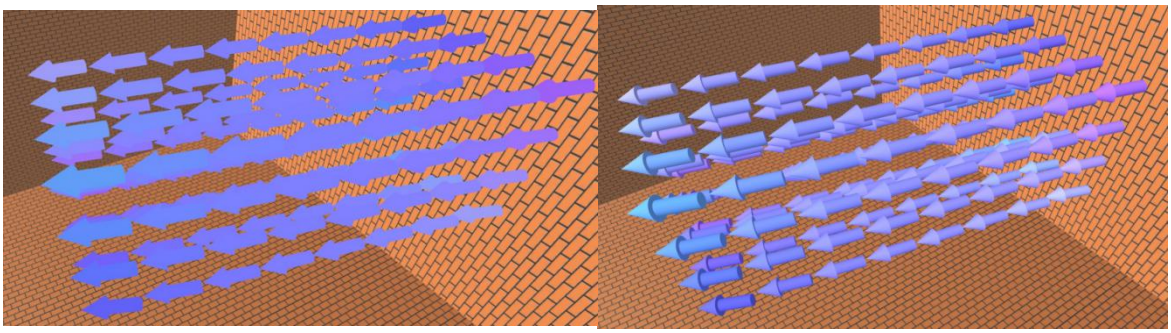


図 8.16 シェーディングモデルを適用しない(左)/した(右)場合図

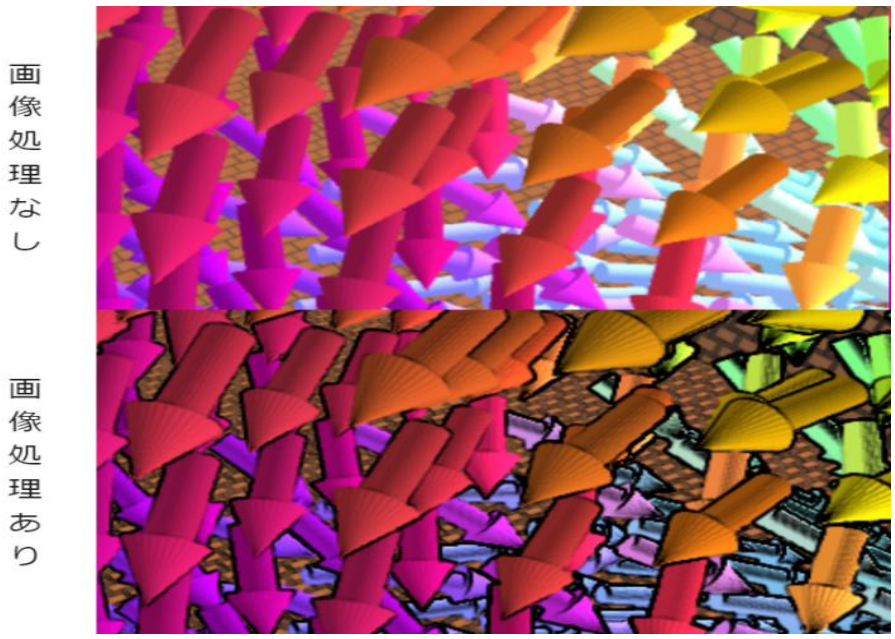


図 8.17 画像処理比較図

8.6 性能評価

性能評価で使われているデスクトップパソコンは表 6.1 と同じである.

8.6.1 性能評価 1

図 8.18 は同じ PC で計算方法を変えた時の 1 フレームの計算時間を示している．すべての計算は float 精度に行う．Compute shader は CUDA より約 2 倍，CPU(c#)より約 40 倍速い.

CPU の C++は C#より約 2 倍くらい速いが, 粒子数が 1000 粒子を超えるとリアルタイムでシミュレーションすることができなくなる.

Compute shader と CUDA を使っている計算プログラムは 8000 粒子まではリアルタイムでシミュレーションすることができている.

8.6.2 性能評価 2

図 8.19 は CUDA と Compute shader の double, float 精度の性能を示している. 粒子数が増えると，CUDA では，double，float 精度の性能はほぼ同じである. Compute shader では，float 精度の性能は double より約 2 倍速い.

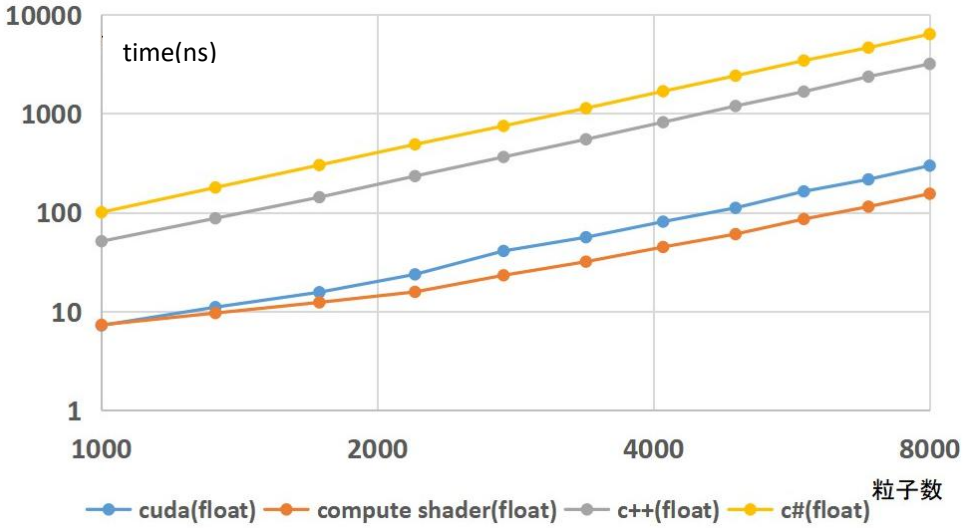


図 8.18　性能評価 1

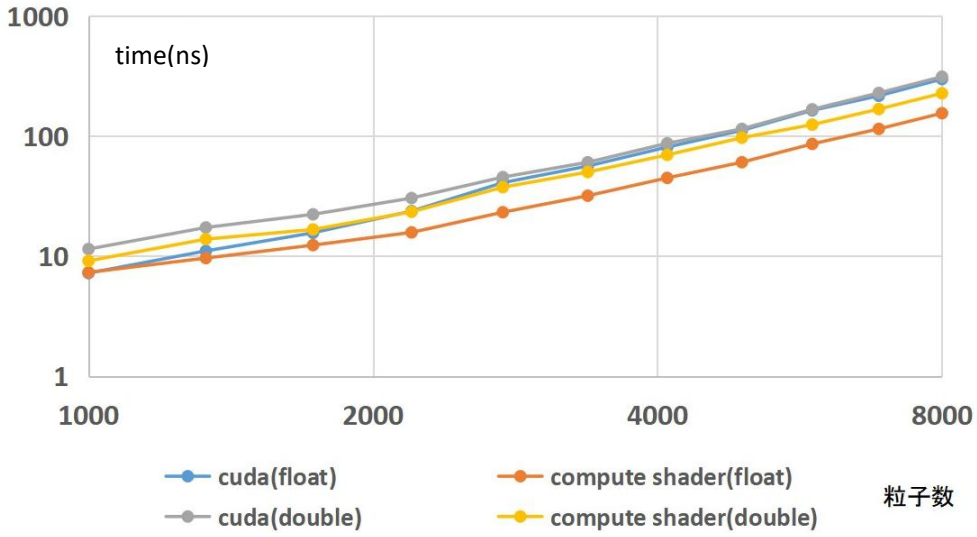


図 8.19　性能評価 2

8.6.3　性能評価 3

図 8.20 は比較的性能の低いスマートフォン(OPPO AX7[表 8.2])とタブレット(Fire HD 10[表 8.3])の計算時間を PC の CPU と比較したものである．モバイル端末でも GPU を使えばパソコンの CPU 並の性能が出る．

表 8.2：スマートフォン OPPO AX7 のパラメータ

CPU	GPU	RAM
Qualcomm® Snapdragon™450	adreno 506	4GB

表 8.3：タブレット端末 Fire HD 10 のパラメータ

CPU	GPU	RAM
Mediatek MT8183 (Helio P60T)	Mali-G72 MP3	3GB

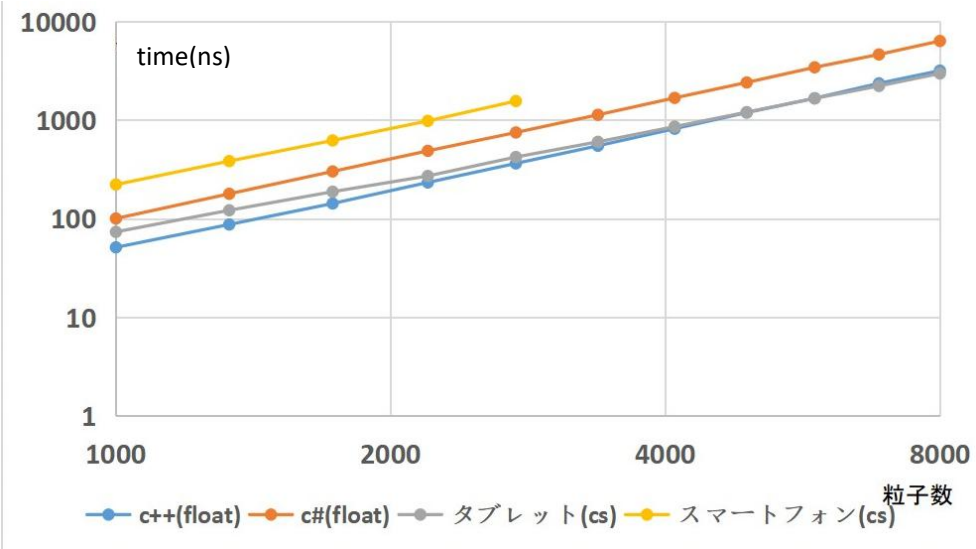


図 8.20　性能評価 3

8.6.4　性能評価 4

表 8.4 は粒子数が 1152 の場合にスマートフォンでの性能を比較したものである．比較的新しいスマートフォンを用いればリアルタイムのシミュレーションが可能であることが分かる．

表 8.4：各スマートフォンでの性能(フレームレート)

機種	Pixel3a	GalaxyZ Flip3	xiaomi mi9t pro	Xperia XZ1
発売年	2019	2021	2019	2017
性能	8fps	34fps	27fps	20fps

8.7 精度評価

本節では，実装した二つのシミュレーション計算プログラムについて，精度を評価する．

精度の計算において，CPU で計算された結果を基準として，相対誤差を算出する．

計算精度E_{avr}は表 x のようにして計算する．

CPU(double 精度)を用いて計算した i 番目の粒子の Vec 値を$\overrightarrow{V_i}$とし，GPU による i 番目の粒子の Vec 値を $\overrightarrow{G_i}$とする．

表 8.5 相対誤差の計算方法

1. CPU で全粒子の平均$\overrightarrow{V_i}$値V_{avr}を求める．
2. GPU で i 番目の $\overrightarrow{U_i}$を計算する．
3. 粒子の Vec の相対誤差 $E_i = \left\|(\overrightarrow{V_i} - \overrightarrow{G_i})/V_{avr}\right\|$ を求める
4. 全粒子の平均相対誤差E_{avr}を求める

CUDA と Compute shader で実装した計算プログラムのデータを double 型と float 型で切り替えて計算している．

シミュレーションは表 8.6 のパラメータで行う．

表 8.6：シミュレーションプロパティ

x 軸方向の粒子数　Nx	4
y 軸方向の粒子数　Ny	4
z 軸方向の粒子数　Nz	4
x 軸方向の粒子数　Hxc	0.1
y 軸方向の粒子数　Hyc	0
z 軸方向の粒子数　Hzc	-200
損失定数 α	1
粒子数　N	64

8.7.1　CUDA ベースシミュレーション計算プログラムの誤差解析

図 8.21 は CUDA と C++で実装されたシミュレーション計算プログラムの計算誤差を示している．粒子は外部磁場の影響により，向きが激変している時間帯(28~31ns)の計算は誤差が大きいが，収束結果は合っていることがわかる．

8.7.2　Compute shader シミュレーションプログラムの誤差解析

図 8.22 は, Compute shader でシミュレーションを行った際の相対誤差を示している.

レンダリングするためのシェーダーでは，double でデータを格納することはできないため，double 型のデータで Compute shader を実行しても，計算された結果はレンダリングできない．そのため，Compute shader を使う際には float 型を使う方が良いと考えられる.

本来は Compute shader の誤差は CUDA と同じだと期待しているが，少し Compute shader の方が悪い.

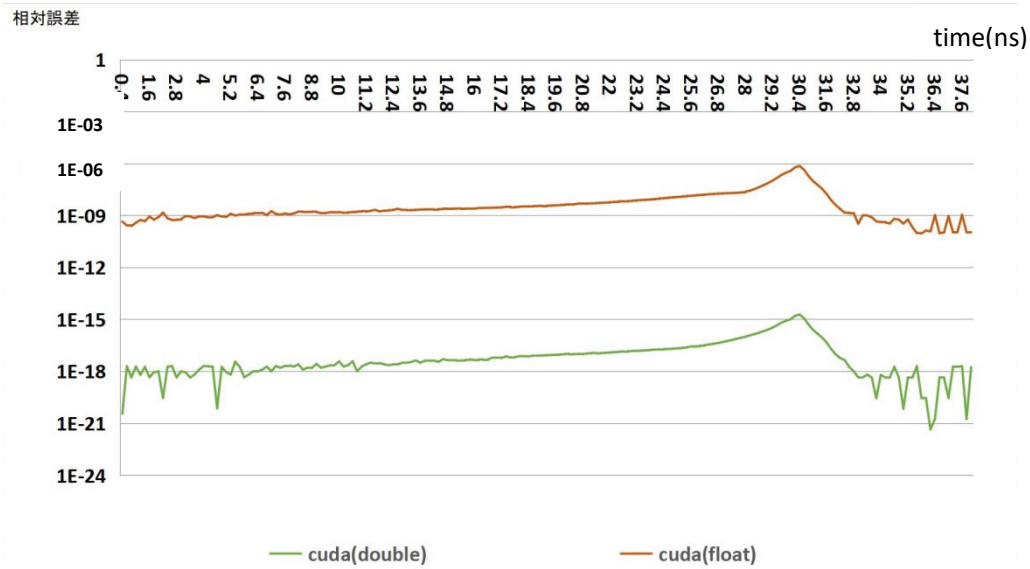


図 8.21　CUDA 計算誤差

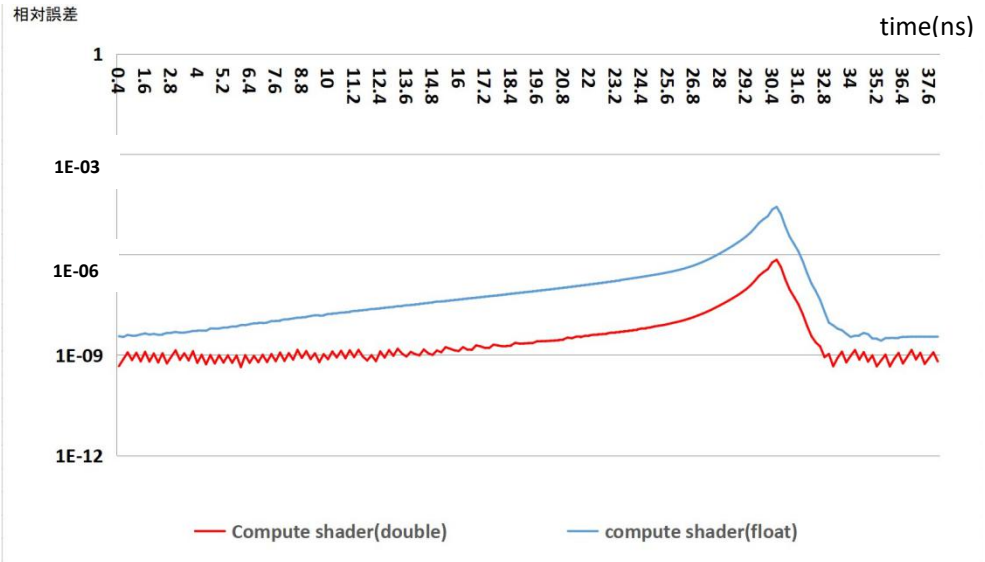


図 8.22　Compute shader 計算誤差

8.8 考察：

Compute shader はあくまでもグラフィック向けのツールであり，計算精度が足りない問題があるため，精密な数値計算に向いていないと考えられる．関連研究では，Compute shader でクロスシミュレーション，PBD ベースの流体シミュレーションなどのグラフィック関連の研究は存在しているが，科学的物理シミュレーションで利用したものは少ない．また，数値計算するためのライブラリなどはサポートされておらず，難しい物理シミュレーションアルゴリズムの実装は未だに困難である．マイクロマグネティクスシミュレーションの計算プログラムでは，計算量が最も重い静磁界の計算部分は FFT により，計算量を O(nlogn)まで削減できるが，Compute shader は FFT ライブラリがないため，本システムでは，計算量は $O(n^2)$のアルゴリズムを実装した．小規模なシミュレーションでは FFT を使うメリットが少ないが，大規模な場合は FFT の実装が不可欠であろう．総合的に考えると，Compute shader で精密な科学的物理シミュレーションをするには注意が必要である．

描画プログラムは計算プログラムで計算した結果を矢印モデルとして可視化した．Compute shader はレンダリングパイプラインとの連携は強いため，計算したデータをホスト側に転送する必要がなく，そのまま shader プログラムと連携し，手軽に描画できる．描画された画像はポストプロセスにより観察しやすくなった．

本システムでは，マルチプラットフォーム対応しており，スマートフォン，タブレットなどの端末でも実行できる．計算プログラムで使用される Compute shader はグラフィックスドライバーの一部であるため，CUDA のように数ギガバイトの実行環境をインストールする必要がない．そのため，利用者に対する負担が低く，手軽に使用できる．精度にこだわらないエンターテインメント向けのシミュレーション(位置ベースダイナミックなど)では，Compute shader が向いていると考えられる．さらに，Compute shader はピクセルシェーダーでシミュレーションを実装するより簡単である．そのため，本来ならピクセルシェーダーで実装されるポストプロセスは Compute shader で実装することもできる．

第 9 章　　おわりに：

本章では本文のまとめとシステムの問題点，改善すべき点について述べる．

9.1 まとめ

マイクロマグネティクスによる新たなメモリー材料の開発が期待されている．本研究では，ゲームエンジンにより，マルチプラットフォーム対応のマイクロマグネティクスリアルタイムシミュレーションの可視化システムの開発と評価を行った．

リアルタイムシミュレーション計算プログラムのコアとなる部分は GPGPU 技術を用いている．これにより，CPU ではなく GPU で数値計算を高速に行うことで，リアルタイムでの計算が可能になる．本システムでは，グラフィックスドライバーに内蔵されている Compute shader を利用する．

可視化プログラムでは，前段階のシミュレーション計算プログラムで計算されたシミュレーションデータを位置，方向，色などの 3D 空間情報に転換し，リアルタイムに描画した．また，描画された画像はポストプロセスにより，より観察しやすくなった．

数値計算の分野では，Compute shader より，CUDA の方が向いていると考えられる．難しいアルゴリズムの実装では，Compute shader は数多くの制限がある．ただし注意深く制限の中で使用すれば有用な部分も多い．

様々なデバイス上で実行できる手軽さにより，教育，エンターテインメント向けなどで物理シミュレーションを体験することができると期待している．

今回のようにゲームエンジンで可視化システムを開発したのは適切だったと考えられる. 描画の際に, 手軽にレンダリングパイプラインを制御できる. マルチプラットフォームに対応していることも有用である．

9.2 今後の課題

今後の課題としては，Compute shader で FFT を用いて計算を高速化することである．ライブラリが使えないが独自に FFT アルゴリズムを実装すれば，計算プログラムの計算量は $O(n^2)$から $O(nlogn)$にまで削減できる．

本研究の評価では Compute shader の精度が CUDA の float より悪い結果となった．その原因をつきとめ改善できれば，今よりも精密なシミュレーションにも応用できる可能性がある．

謝辞

マイクロマグネティクスシミュレーションのサンプルコードを提供頂いた仲谷教授に感謝致します.

参考文献：

[1] 清水陽介，マイクロマグネティクスシミュレーションリアルタイム可視化，電気通信大学情報理工学研究科成見研 修士論文，2012

[2]GPU
https://ja.wikipedia.org/wiki/Graphics_Processing_Unit

[3]GPGPU
https://ja.wikipedia.org/wiki/GPGPU

[4]CUDA
https://ja.wikipedia.org/wiki/CUDA

[5]Comopute shader
https://docs.unity3d.com/ja/2018.4/Manual/class-ComputeShader.html

[6]OpenCL
https://ja.wikipedia.org/wiki/OpenCL

[7]可視化
https://ja.wikipedia.org/wiki/可視化

[8]科学的可視化
https://en.wikipedia.org/wiki/Scientific_visualization

[9] ゲームエンジン
https://ja.wikipedia.org/wiki/ゲームエンジン

[10]Unity
https://ja.wikipedia.org/wiki/Unity_ゲームエンジン

[11]レンダリングパイプライン
https://ja.wikipedia.org/wiki/レンダリングパイプライン

[12]shader
https://ja.wikipedia.org/wiki/シェーダー

[13]Learn OpenGL-Graphics Programming，Joey de Vries
https://learnopengl.com/

[14]Hongly Va,Min-Hyung Choi,Min Hong,”Real-Time Cloth Simulation Using Compute Shader in Unity3D for AR/VR Contents”,Multidisciplinary Digital Publishing Institute,vol.11,issue.17,no.8255,2021

[15]Francisco Sans,Rhadams Carmona,”A Comparison between GPU-based Volume Ray Casting Implementations: Fragment Shader,Compute Shader, OpenCL,and CUDA”,CLEI ELECTRONIC JOURNAL,vol.20,no.2,2017

[16]a-Pitch-yaw-and-roll-angles-of-an-aircraft-with-body-orientation-O-u-v-original
https://www.researchgate.net/figure/a-Pitch-yaw-and-roll-angles-of-an-aircraft-with-body-orientation-O-u-v-original_fig7_348803228

[17]RGB Color Model RGB Color Space
https://favpng.com/png_view/primary-color-rgb-color-model-rgb-color-space-cie-1931-color-space-png/GZ0xXWqA

[18]An Implementation of Sobel Edge Detection
https://www.projectrhea.org/rhea/index.php/An_Implementation_of_Sobel_Edge_Detection

付録 A　プログラムリスト

A.1　Compute shader プログラム

A.2　shader プログラム