粒子法による3次元地形の侵食解析プログラムの開発

大坪晃輔　草間晴幸

1　研究背景 / 目的

本研究ではCGで表現される地形形状に関し，3次元地形に対して侵食シミュレーションを行う手法について取り扱う．シミュレーションはボクセルデータによる3次元地形と，粒子法(SPH法)による流体の組み合わせによって構成される．流体を表す粒子の流速といった情報を参照し,ボクセルデータの濃度値を変更することで地形の侵食を再現できる．

粒子法は流体解析の手法であり，格子法に比べ液体の自由表面を取り扱うことに優れている[1)]．Krištof(2009)[2)]ではSPH法による流体を用い，高さマップに水流を流すことで，洗掘と堆積の現象を再現している．また粒子法と剛体等の衝突ついては多くの研究がなされている．本研究でも流体とボクセル地形との相互作用は主要な問題となるが，Harada(2007)[3)]の壁境界計算手法を参考とする．

侵食計算に関してはボクセルデータへ適用するための計算を行うが，実際に堤防の侵食に関する室内実験を行っているFujisawa(2011)[4)]を参考に結果の検証を行う．

最終的には侵食シミュレーションを用いて，3次元地形を侵食し地形形状を得ることを目的とする．

また侵食シミュレーションは開発環境としてUnity3D(Unity Technologies http://unity3d.com/)を用いており，同ゲームエンジンでコンパイルされるC#スクリプトによって処理の大部分が記述されている．これらのコードの一部と照会しつつシミュレーションの内容について紹介するが，Unity3Dのライブラリを利用している部分がある．

2　粒子法による流体解析

2.1　SPH法

粒子法では流体を多くの粒子の集合として表現する．流体の支配方程式は格子法を用いる場合には次のように表される[5)]．このとき$\boldsymbol{v}$は流速，ρは密度，Pは圧力，gは重力等の外力，μは粘性係数を表す．

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{v}) = 0$$
$$\rho\left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\nabla\boldsymbol{v}\right) = -\nabla P + \rho g + \mu\nabla^2\boldsymbol{v} \tag{1}$$

最初の式が質量保存則を表し，2つ目の式が運動量保存則を表している．粒子法における流体を記述する場合，粒子が受け持つ質量が一定であることから，質量保存則を省略することができる．また粒子が流速と共に移動することから，移流項$\boldsymbol{v}\cdot\nabla\boldsymbol{v}$もまた省略し，左辺を統合して$D\boldsymbol{v}/Dt$とすることができる．

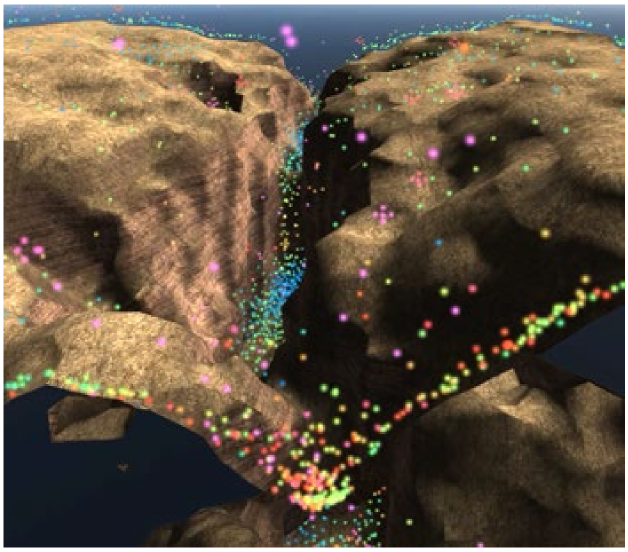


図-1　シミュレーションの様子

実際の粒子の速度変化の計算については，式(1)と，Becker(2007)[6)]で採用された右辺各項についての近似式に基づいて行う．

ある粒子における密度，圧力といった物理量は周囲の粒子に重み関数をかけた離散化式によって求めることができる[5)]．このときWが重み関数であり，hは重み関数の影響半径を表す．$\boldsymbol{r}$は座標を表し，mは質量，また添字のjは近傍粒子を表す．

$$A_S(\boldsymbol{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{2}$$

シミュレーションは0.002秒(本シミュレーションの場合)毎にステップを繰り返すことによって進行するが，1ステップの内容は次のようになる．

1. 近傍粒子探索格子の更新
2. 密度・圧力の計算
3. 圧力項・粘性項の計算
4. 粒子の移動

またこの計算ステップはC#のコードでは次のように実装されている．

ソースコード１　計算ステップ

```
public virtual void step_frame()
{
  grid.update_grid(); // process 1
  // process 2
  grid.delegate_valid(new ParticleGrid.NeighborProcess(
      calc_density_pressure));
  // process 3
  grid.delegate_valid(new ParticleGrid.NeighborProcess(
      calc_force));
  // process 4
  grid.delegate_valid(new ParticleGrid.Process(
      calc_position));
}
```

計算では各粒子について幾つかの計算結果を保持し，参照する必要があるが，粒子は次のような形で情報を保持している．

ソースコード２　粒子の持つ情報

```
public struct ParticleState
{
  public bool isFixed;

  public float density, pressure;
  public Vector3 velocity, force;

  public float boundary;
  public Vector3 boundary_gradient;
}
```

isFixed は粒子を固定するかの判定であり，流体を表す際には false に設定される．density，pressure は密度・圧力計算の結果で，圧力項と粘性項計算の際に利用する．force は圧力項，粘性項，外力の合計で位置移動の際に利用，velocity は位置移動と合わせて更新する．boundary，boundary_gradient は壁境界計算の結果を格納する値である．

2.2　近傍粒子探索

粒子法の計算では近傍粒子の探索を効率的に行うことが重要である．本研究では Fujisawa(2013)[7] を参考に，Green(2010)[8] と同様の手法を実装した．

まず計算範囲となる空間を格子状に分割する．この格子と粒子の関係を図-2に示す．

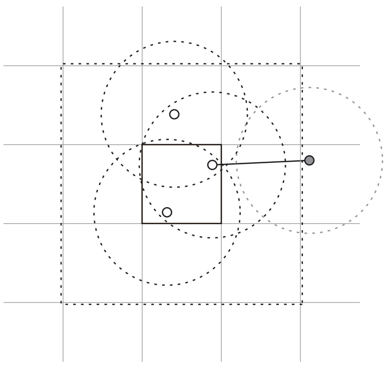


図-2　格子と粒子の関係

粒子は何れかのセル (格子で分割された領域) に内包されることになるが，この時セルの１辺は粒子の重み関数の影響範囲の半径よりも長い．したがって中央の太線で示されたセルについて見る場合，内包された粒子がセルの外縁に寄っている場合でも，点線で囲われた９つの近傍セルより外には影響半径が届かないことが保証されている．線分で結ばれた粒子を見ても，９つのセルに入らない限り近傍粒子 j に含む必要がないことが分かる．

これを利用し，最終的に近傍粒子 j を NeighborCue に格納することで以後の粒子法の計算を可能としている．

2.3　密度・圧力の計算

最初に式 (2) から密度を求める [5]．

$$\rho_S(\boldsymbol{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W_{\mathrm{poly6}}(\boldsymbol{r} - \boldsymbol{r}_j, h) = \sum_j m_j W(\boldsymbol{r} - \boldsymbol{r}_j, h)$$

C#コードでは次のように表される．また以下のコードで示される関数は１つの粒子に対する処理を記述したものであり，全ての有効な粒子に対して引数 i を変えて呼ばれることになる．また第二引数 neighbor_cue には該当粒子 i からみた近傍粒子が格納されている．

ソースコード３　密度・圧力の計算

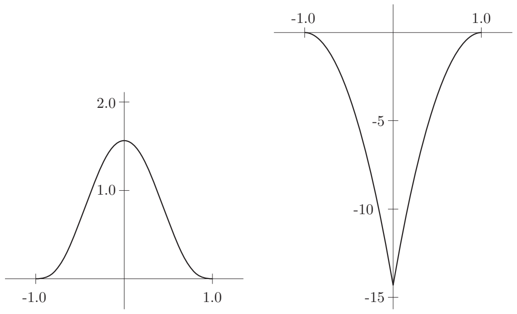
```
void calc_density_pressure(int i, int[] neighbor_cue)
{
  float weight = 0.0f;
  Particle center = grid.particles[i];

  // Sum Neighbor Particles
  for(int cue_i=0; cue_i+1<neighbor_cue.Length; cue_i+=2)
  {
    int start = neighbor_cue[cue_i];
    int end = neighbor_cue[cue_i + 1];
    if(start == -1)
      continue;

    for(int cue_j=start; cue_j<end; ++cue_j)
      weight += poly6_kernel(Vector3.Distance(grid.
          particles[grid.get_index(cue_j)].position,
          center.position), effectiveRadius);
  }
  particle_states[i].density = weight * mass;

  particle_states[i].pressure = pressureCoefficient * (
      Mathf.Pow(particle_states[i].density /
      initialDensity, 7.0f) - 1.0f);
}
```

関数内には総和 $\sum_j$ を行うための２重ループが存在している．ループ内では weight に対して関数 poly6_kernal の戻り値が加算されていくが，これが重み関数を表す．重み関数は用途によって異なるものを使用し，密度計算では Poly6 カーネルを用いるが，その形状を図-3の左に示す．

これを次の式によって求める．

$$W_{poly6}(r,h) = \alpha \begin{cases} (h^2 - r^2)^3 & (0 \le r \le h) \\ 0 & \text{otherwise} \end{cases}$$

$$\text{ただし } \alpha = \frac{4}{\pi h^8}, \frac{315}{64\pi h^9} \quad (\text{2D, 3D})$$



(a) poly6　(b) spiky(gradient)

図-3　重み関数 W_{poly6}，$\nabla W_{\mathrm{spiky}}$

また C#コードでは次のように表される．

ソースコード 4　Poly6 カーネル

```
public float poly6_kernel(float r, float h)
{
  if(r < h)
    return constant_poly6 * Mathf.Pow(h * h - r * r, 3.0f)
        ;
  else
    return 0.0f;
}

float constant_poly6 = 315.0f / (64.0f * Mathf.PI * Mathf.
    Pow(effectiveRadius, 9.0f));
```

尚，constant_poly6 は定数であり，その他の重み関数でも同様に定数を用いている．掲載コードでは定義時に値を計算しているが，effectiveRadius(重み関数の影響半径) がコンパイル時に不定である場合には，定義より後の適切なタイミングで計算を行う．

圧力は密度から求めることができるが，Müller(2003)[5] と同じく定常状態からの変位を圧力として用いる[9]．

$$P = k(\rho - \rho_0) \qquad (k \text{ はガス定数})$$

しかしながら本シミュレーションでは，次節で述べる圧力項・粘性項による Dv/Dt の計算を 6) に基いて行うため，次の式で圧力を計算する．

$$P = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \quad \text{ただし } \gamma = 7,\ B = \frac{\rho_0 c_S^2}{\gamma}$$

Becker(2007)[6] では計算領域の寸法などから $C_S \approx 88.5$m/s, $B \approx 1119$kPa としている．本研究では $C_S \approx 140.0$m/s, $B \approx 2795$kPa として良好な結果を得ている．

2.4　圧力項・粘性項の計算

圧力項は流体の圧力を均衡に保つよう作用する力で，次式で適用する[6]．

$$\frac{D\boldsymbol{v}}{Dt} = -\sum_{j \neq i} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\right) \nabla W_{\mathrm{spiky}}(\boldsymbol{r}_i - \boldsymbol{r}_j, h)$$

また粘性項を次式で適用する[6]．

$$\frac{D\boldsymbol{v}}{Dt} = \begin{cases} -\sum_{j \neq i} m_j \Pi_{ij} \nabla W_{\mathrm{spiky}}(\boldsymbol{r}_i - \boldsymbol{r}_j, h) & (\boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} < 0) \\ 0 & (\boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij} \geq 0) \end{cases}$$

$$\text{だたし } \Pi_{ij} = -\nu \left(\frac{\boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij}}{|\boldsymbol{r}_{ij}|^2 + \varepsilon h^2}\right),\ \nu = \frac{2\alpha h c_S}{\rho_i + \rho_j}$$

α は 0.08 から 0.5 の間で選択すると良いとされる．ここで εh^2 は $r = 0$ の場合に分母が 0 となる事を防ぐための値であり，$\varepsilon = 0.01$ である．尚，$\boldsymbol{v}_{ij}$ は $\boldsymbol{v}_i - \boldsymbol{v}_j$ を指し，粒子 j から見た粒子 i の流速を表す．したがって 2 つの粒子が近づく際の緩衝を記述している．

C#コードでは次のように表される．

ソースコード 5　圧力項・粘性項の計算

```
void calc_force(int i, int[] neighbor_cue)
{
  Particle center = grid.particles[i];
  particle_states[i].force = Vector3.zero;
  Vector3 pressure_force = Vector3.zero, viscosity_force =
       Vector3.zero;
  Vector3 relativePosition, relativeVelocity, temp_kernel;

  float center_pressure = particle_states[i].pressure /
      Mathf.Pow(particle_states[i].density, 2.0f);

  for(int cue_i=0; cue_i+1<neighbor_cue.Length; cue_i+=2)
  {
    int start = neighbor_cue[cue_i];
    int end = neighbor_cue[cue_i + 1];
    if(start == -1)
      continue;

    for(int cue_j=start; cue_j<end; ++cue_j)
    {
      int j = grid.get_index(cue_j);

      relativePosition = center.position - grid.particles[
          j].position;
      float r = relativePosition.magnitude;
      if(r >= effectiveRadius)
        continue;

      // pressure
      float neighbor_pressure = particle_states[j].
          pressure / Mathf.Pow(particle_states[j].density
          , 2.0f);
      temp_kernel = spiky_gradient(r, relativePosition,
          effectiveRadius);

      pressure_force += (center_pressure +
          neighbor_pressure) * temp_kernel;

      // viscosity
      relativeVelocity = particle_states[j].velocity -
          particle_states[i].velocity;

      float dot = Vector3.Dot(relativeVelocity,
          relativePosition);

      if(dot_position_velocity < 0.0f)
        viscosity_force -= temp_kernel * dot / (
            relativePosition.sqrMagnitude + stability) *
            viscosity / (particle_states[i].density +
            particle_states[j].density);
    }
  }
  particle_states[i].force = mass * (viscosity_force -
      pressure_force);
}
```

また重み関数であるSpikyカーネルの勾配は次のように求められる．

$$\nabla W_{spiky}(\boldsymbol{r},h) = \alpha \begin{cases} (h-r)^2 \dfrac{\boldsymbol{r}}{r} & (0 \le r \le h) \\ 0 & \text{otherwise} \end{cases}$$

$$\text{ただし}\ \alpha = -\frac{30}{\pi h^5}, -\frac{45}{\pi h^6} \quad (\text{2D, 3D})$$

ソースコード 6　Spiky カーネル (勾配)

```
public Vector3 spiky_gradient(float r, Vector3
    relative_position, float h)
{
  if(r > 0.0f)
  {
    float q = h - r;
    return constant_spiky_gradient * relative_position * q
        * q / r;
  }
  else
    return Random.onUnitSphere * 0.0001f *
        constant_spiky_gradient;
}

float constant_spiky_gradient = -45.0f / (Mathf.PI * Mathf
    .Pow(effectiveRadius, 6.0f));
```

2.5　粒子の移動

圧力項と粘性項に外力を加え，ステップ毎の粒子の移動を行う．

ソースコード 7　粒子の移動

```
void calc_position(int index)
{
  if(!particle_states[index].isFixed)
  {
    particle_states[index].velocity += particle_states[
        index].force * timeStep + (enable_gravity == true
         ? gravity * timeStep : Vector3.zero);

    grid.particles[index].position += particle_states[
        index].velocity * timeStep;
  }
}
```

3　ボクセル地形と壁境界計算

ここでボクセルデータとは3次元の濃度分布を格子点で読み取り，濃度値の3次元配列としたものを指す．

地形の描画にはマーチング・キューブ法によって生成されるメッシュを用いる[10]．MC法(マーチング・キューブ法)はボクセルデータを可視化する手法であり，格子点8つに囲まれた立方体領域(キューブ)について，格子点における濃度値がそれぞれ閾値を超えているか否かを判定する[11]．これには$2^8 = 256$通りのパターンがあり，対応するメッシュを割り当てることで閾値の等値面をメッシュとして生成することができる．

3.1　粒子・壁境界の距離

本シミュレーションでは流体解析を行う空間に地形表現のためのボクセルデータが重なって存在し，これが壁境界を規定している．したがってボクセルデータを元に壁境界を直接求めることができれば，効率的にシミュレーションを行うことができる．

粒子がボクセル表面と隣接するとき，粒子は1つのキューブに内包されることになる．キューブ内の任意の点iにおける濃度値をd_iとすると，点iを囲む8つの格子点の濃度値を線形補間することでd_i，及び$\boldsymbol{n}_i$を求めることができる．そして壁境界までの距離l_Bをd_iが勾配に従って閾値に近づくとして次のように求める．

$$(l_B)_i = \frac{d_i - \mathrm{ISO}}{|\boldsymbol{n}_i|}$$

勾配を求める様子を2次元で表したものを図-4に示す．

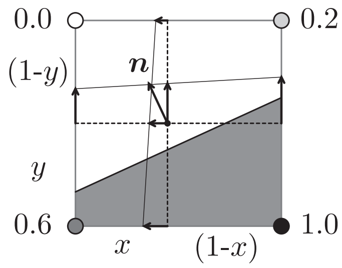


図-4　勾配の導出

図-5では1つのキューブについて，格子点の濃度値に従いマーチング・キューブ法によって傾斜をもつ上向きの面が張られている．ここで高さ約0.5の平面上に粒子を並べ，それぞれについて線形補間による壁境界位置を求めた．平面上の粒子と線分で繋がれた粒子が，その粒子における最近傍の境界点を示している．メッシュの面とは異なり湾曲した曲面を描くものの，このような例では適切に境界を求めることが可能である．

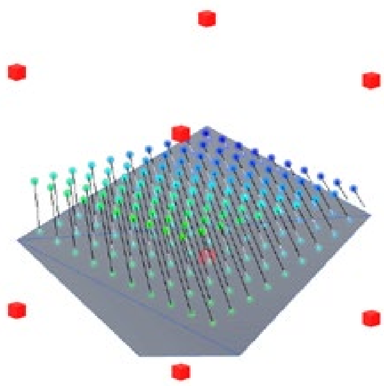


図-5　線形補間によって得られる境界位置

3.2　壁境界の作用

Harada(2007)[3]では壁重み関数を用いた効率的な壁境界計算を行っている．壁重み関数とは，均一に平坦に並べられた壁粒子群に粒子が近づいた時，粒子が受ける反発力，摩擦力と

いった力が粒子と壁との距離のみをパラメータとして表現できるとするものである．

壁粒子が粒子に及ぼす影響を流体粒子間の作用と分離すると，粒子の密度は重み関数を利用して次のように表される[3]．

$$\rho_S(\boldsymbol{r}) = \sum_{j\in\mathrm{fluid}} m_j W_{ij} + F_i^{\mathrm{density}}(l)$$

境界計算を含めた圧力項の計算は次のようになる．

$$\left(\frac{D\boldsymbol{v}}{Dt}\right)_{\mathrm{press}} = -\sum_{(j\neq i)\in\mathrm{fluid}} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\right)\nabla W_{ij} + F_i^{\mathrm{press}}(l)$$

粘性項の計算には，境界に対する垂直方向と，水平方向の2つの壁重み関数を用いて次のように計算する．

$$\left(\frac{D\boldsymbol{v}}{Dt}\right)_{\mathrm{vis}} = -\sum_{(j\neq i)\in\mathrm{fluid}} m_j \Pi_{ij} \nabla W_{ij} + \left(\boldsymbol{\nu}_n F_i^{\mathrm{vis(n)}}(l)\right)^* + \boldsymbol{\nu}_p F_i^{\mathrm{vis(p)}}(l)$$

壁重み関数は計測用のモデルによって事前に計算を行う．今回使用したモデルを図-6に示す．壁境界を表す固定粒子の間隔は $a = 0.55$ として規定している．

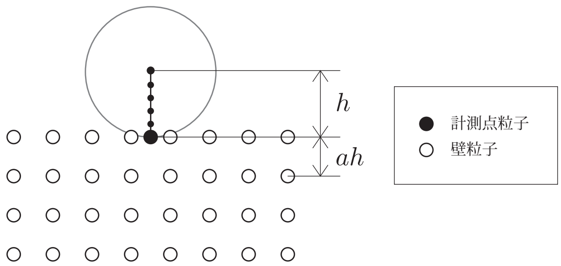


図-6　壁重み関数の計測モデル

計測モデルを用い，1つの計測粒子と壁境界との距離を徐々に変えつつ，計測粒子が受ける影響をプロットする．これは次のようにSPH法の処理を改変したコードによって計測することができる．

ソースコード 8　壁重み関数の計測

```
public void bake()
{
  for(int i=0; i<keys.Length; ++i)
  {
    grid.particles[index].position = transform.position +
        Vector3.up * keys[i] * effectiveRadius;
    grid.update_grid();
    sph.step_frame();

    // density
    grid.delegate_selected(new ParticleGrid.
        NeighborProcess(bake_density), index);

    data.density.AddKey(new Keyframe(keys[i], sph.
        ParticleStates[index].density / mass));

    // pressure viscosity
    grid.delegate_selected(new ParticleGrid.
        NeighborProcess(bake_force), index);

    data.pressure_force.AddKey(new Keyframe(keys[i],
        repulsion.y));

    data.friction.AddKey(new Keyframe(keys[i], friction));
    data.adhesive.AddKey(new Keyframe(keys[i], adhesive));
  }
}

void bake_force(int i, int[] neighbor_cue)
{
  ParticleSystem.Particle center = grid.particles[i];
  pressure_force = Vector3.zero;
  friction = adhesive = 0.0f;
  Vector3 relativePosition, relativeVelocity, temp_kernel;
  float coef_center_pressure = 1.0f / Mathf.Pow(sph.
      ParticleStates[i].density, 2.0f);

  for(int cue_i=0; cue_i+1<neighbor_cue.Length; cue_i+=2)
  {
    int start = neighbor_cue[cue_i];
    int end = neighbor_cue[cue_i + 1];
    if(start == -1)
      continue;

    for(int cue_j=start; cue_j<end; ++cue_j)
    {
      int j = grid.get_index(cue_j);
      if(i == j)
        continue;

      // pressure
      relativePosition = center.position - grid.particles[
          j].position;
      float r = relativePosition.magnitude;
      if(r > effectiveRadius)
        continue;

      float neighbor_pressure = sph.ParticleStates[j].
          pressure / Mathf.Pow(sph.ParticleStates[j].
          density, 2.0f);
      temp_kernel = spiky_gradient(r, relativePosition,
          effectiveRadius);

      // pressure
      pressure_force += (coef_center_pressure * Mathf.Max(
          sph.ParticleStates[i].pressure, 0.0f) +
          neighbor_pressure) * temp_kernel;

      // friction
      relativeVelocity = sph.ParticleStates[j].velocity -
          Vector3.right;
      float dot_position_velocity = Vector3.Dot(
          relativeVelocity, relativePosition);

      if(dot_position_velocity < 0.0f)
        friction -= (temp_kernel * dot_position_velocity /
             (relativePosition.sqrMagnitude + sph.
            desbrum_stability) * viscosity * sph.
            desbrum_viscosity / (sph.ParticleStates[i].
            density + sph.ParticleStates[j].density)).x;

      // adhesive
      relativeVelocity = sph.ParticleStates[j].velocity -
          Vector3.up;
      dot_position_velocity = Vector3.Dot(relativeVelocity
          , relativePosition);

      if(dot_position_velocity < 0.0f)
        adhesive -= (temp_kernel * dot_position_velocity /
             (relativePosition.sqrMagnitude + sph.
            desbrum_stability) * viscosity * sph.
            desbrum_viscosity / (sph.ParticleStates[i].
            density + sph.ParticleStates[j].density)).y;
    }
  }
}
```

計測によって得られた壁重み関数の曲線を図-7に示す．

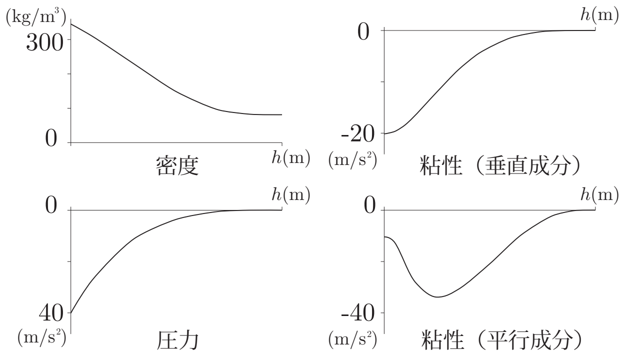


図-7　壁重み関数の計測結果

4　侵食メカニズム

単位質量あたりの水流に削り取られる土砂の量は次の数式で表される[2, 12]．

$$\frac{dM_b}{dt} = -\sum_j L_b^2 \varepsilon(j) \tag{3}$$

ここで L_b は地形の単位長さを，ε は流体粒子ごとの侵食の作用を示す．ε は流体のせん断応力によるものとされ，せん断応力 τ によって次のように表される．

$$\varepsilon = K_\varepsilon(\tau - \tau_c) \quad 但し\ \tau = K\theta^n$$

ここで K_ε は侵食の強度，τ_c は限界せん断力，K はせん断応力の定数，θ はせん断速度を示す．$K = 1$ とし，また，べき乗数 n は擬塑性流体に適用される値である 0.5 を用いる[2]．

せん断速度は粒子速度のうち地形表面と平行な成分 $\boldsymbol{v}_p$ から求めることができ，次の近似式で求める．

$$\theta = \frac{\boldsymbol{v}_p}{l}$$

ここで l は粒子と地形表面との距離を示す．

4.1　格子点の侵食計算

ボクセルデータに対する侵食を表現するため，各格子点について次の式に基づく濃度値の増減を計算する．

$$\frac{dM_b}{dt} = -K_\varepsilon \sum_j \begin{cases} \left\{\left(\dfrac{|\boldsymbol{v}_{ij}| - \boldsymbol{v}_{ij}\cdot\hat{\boldsymbol{r}}_{ij}}{l}\right)^n - \tau_c\right\} & (|\boldsymbol{r}_{ij}| < h) \\ 0 & (|\boldsymbol{r}_{ij}| \geq h) \end{cases}$$

ここで各格子点の座標は $\boldsymbol{r}_i$ であり，j は $\boldsymbol{r}_i$ を中心とした近傍粒子を示す．また式 (3) の L_b^2 は表面積を表すが，ボクセルに適用することから，これを一定として K_ε に組み込む．

C#コードでは次のように実装されている．

ソースコード 9　格子点の侵食計算

```
void cross_point_erosion(int x, int y, int z, int[]
    neighbor_cue)
{
  float erosion_rate = 0.0f;
  Vector3 relativePosition, relativeVelocity;
  float r_min = 0.4f * sph.effectiveRadius;
  float r_base = 0.4f * sph.effectiveRadius;

  for(int cue_i=0; cue_i+1<neighbor_cue.Length; cue_i+=2)
  {
    int start = neighbor_cue[cue_i];
    int end = neighbor_cue[cue_i + 1];
    if(start == -1)
      continue;

    for(int cue_j=start; cue_j<end; ++cue_j)
    {
      int j = sph_grid.get_index(cue_j);

      relativePosition = voxel_grid.cell_position(x, y, z)
           - sph_grid.particles[j].position;
      float r = relativePosition.magnitude;
      r = r - Mathf.Min(sph.ParticleStates[j].boundary,
          r_base); // Rectified
      if(r > erosionRadius)
        continue;

      relativeVelocity = sph.ParticleStates[j].velocity;
      float shear_rate = relativeVelocity.magnitude -
          Mathf.Abs(Vector3.Dot(relativeVelocity,
          relativePosition.normalized));
      shear_rate = shear_rate / Mathf.Max(r, r_min);
      erosion_rate += Mathf.Max((Mathf.Pow(shear_rate, 0.5
          f) - critical_stress), 0.0f);
    }
  }

  // Apply to voxel data (Rectified)
  erode[x,y,z] = (byte)Mathf.Min(erode[x,y,z] + Mathf.
      CeilToInt(erosion_rate * byteFit), 255);
  int transfer = Mathf.FloorToInt(erode[x,y,z] *
      erode_strength / byteFit);
  voxel_grid.voxel[x,y,z] = (byte)Mathf.Min(voxel_grid.
      voxel[x,y,z] + transfer, 255);
  erode[x,y,z] = (byte)(erode[x,y,z] % (byteFit /
      erode_strength));
}
```

尚，上記のコードは幾つかの修正処理が追加されたものである．

4.2　越流破堤モデル

侵食の検証の為に堤防の越流侵食を再現するモデルを用いる．これは河川，海岸等の堤防について，増水，高潮によって水位が堤高を超えた状況を示す．

水流は堤体を越えて堤内側に流れ落ちるが，この射流によって堤体は侵食を受ける．越流破堤現象については Fujisawa(2011)[4] を参考とする．射流による侵食は堤体の材質や越流の量によって侵食の特徴を変える[4]．図-8にその概略図を示す．

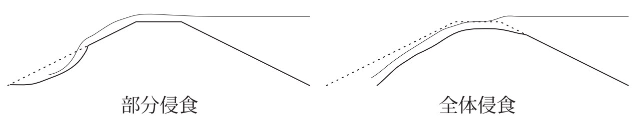


図-8　侵食の種類

堤体が比較的高い抵抗力を持つ場合，天端付近 (堤体の上面部)

では侵食を受けにくい．射流が流れ落ちるにしたがって流速が増し，限界点から下で侵食を受けることとなる[4)]．このような侵食をここでは部分侵食と呼び，再現を試みることとした．

シミュレーションで用いる堤体モデルを図-9のようにボクセルデータによって作成した．尚奥行き方向の寸法は1.2[m]となっている．また図中に重ねて，試験的に行ったシミュレーションの結果を示す．

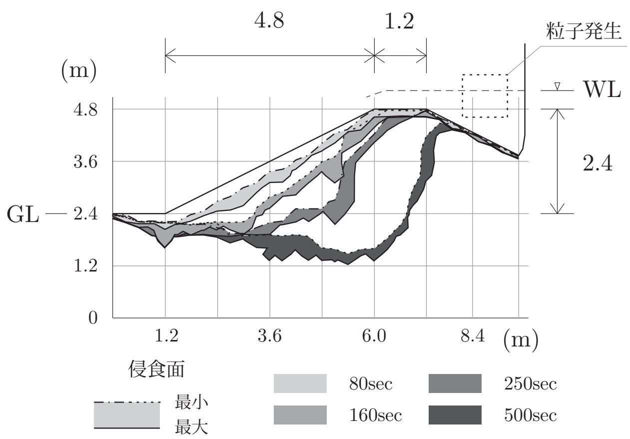


図-9　堤防モデル

試験の結果から以下の修正点が見つかった．

- 壁境界計算における摩擦係数の決定
- 不安定粒子による過剰侵食への対処
- 射流を減速する階段パターンへの対処
- 壁境界による引力の発揮
- タイムステップあたりの侵食量に関する分解能の拡張

4.3　壁境界による引力

修正の中で，壁境界計算の圧力項に引力を発揮させるよう計算の変更を行った．下流で部分侵食が始まった場合，それを広げる為には水流が凹部に入り込む必要がある．しかし引力が発揮されなければ凹部への侵入は限られたものとなり，部分侵食パターンを再現することは難しいことが分かった．

そこで壁粒子が発揮する斥力のみを取り出してプロットしていた圧力項の壁重み関数に加え，引力のみを取り出しプロットした壁重み関数を用意した．

図-10の(a)が斥力のみを計測した壁重み関数であり，(b)は引力のみを計測したもの，(c)は計測粒子の密度値を操作せず，粒子にかかる力をそのまま計測した曲線である．ここで(c)において力が0となる距離をl_Rとする．

曲線(a)をF^{repul}，曲線(b)をF^{attract}とし，圧力項の計算を

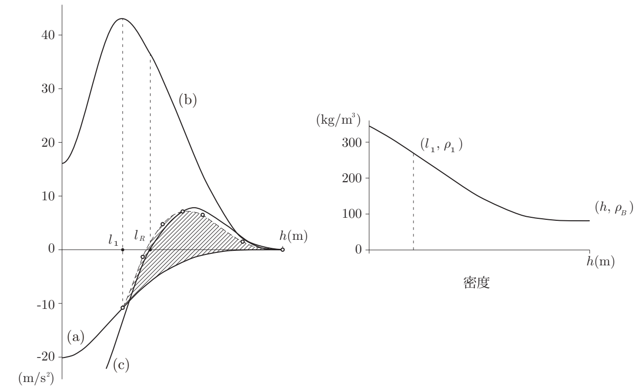


図-10　引力を含む圧力項の曲線

次のように行う．

$$F^{\mathrm{press}}(l) = \begin{cases} \dfrac{\rho}{\rho_1} F^{\mathrm{repul}}(l) + \left(1 - \dfrac{\rho}{\rho_1}\right) F^{\mathrm{attract}}(l) & (\rho < \rho_1) \\ F^{\mathrm{repul}}(l) & \text{otherwise} \end{cases}$$

これは粒子の密度が一定以下の場合に，その密度に応じて曲線(a)と(b)の間で線形補間を行うことを意味している．この線形補間によってF^{press}が取りうる値を図の斜線部に示した．斜線部の上辺をなす破線は単独の粒子が壁境界に近づいた際の計算を意味するが，実際の計測結果(c)とほぼ一致する事がわかる．ここでl_1は補間曲線がl_Rにおいて0となることを目標に設定した．

次に実際のモデルで実験を行ったところ，図-11に示す結果が得られた．左から右に向かうにしたがって，より引力が働くよう線形補間を行っている．

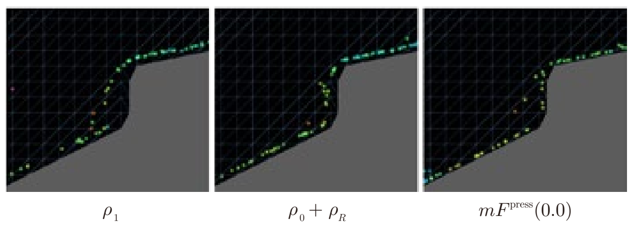


図-11　引力計算による流れの違い

図-10の斜線部で示した補間は左のρ_1のものにあたる．一方最も引力を働かせる場合，即ち線形補間を$l = 0$から開始する場合が右の$mF^{\mathrm{press}}(0.0)$にあたり，強く粒子を引きつけていることが確認できる．

圧力項の補間計算を始める密度値は，目的とするシミュレーションに合わせて変更することが相応しく，侵食の形態を左右する強力なパラメータであることが分かった．

中央の画像では粒子の密度が$\rho_0 + \rho_B$を下回った場合に補間を始めており，適正値の1つと見ている．粒子が壁境界に付着している場合の安定密度を示すわけではなく，文字通り粒子1

つ分定常密度より高い密度値ということになる(ρ_0 の中に粒子自身による密度値も含まれるため)．この値に物理的意味は無いが，この付近の値を採用することで，水圧によって押し付けられた粒子と分離粒子とを比べると，壁境界との距離がほぼ同一になるという特徴が見られる．

4.4　越流破堤シミュレーションの結果

修正後のシミュレーションによって得られた部分侵食パターンの再現を図-12に示す．限界せん断応力を $\tau_c = 6$，侵食強度を $K_\varepsilon = 2$ としている．尚，20秒間隔で200秒までの侵食経過を示す．

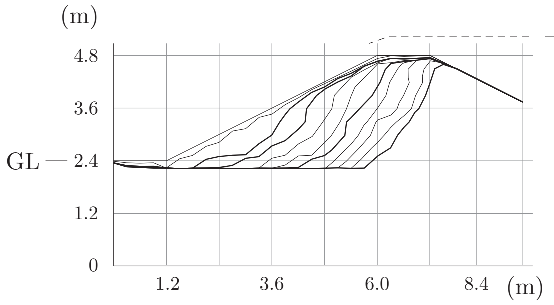


図-12　部分侵食パターン

また図-13は同シミュレーションの40[s]時点の様子を取り出したものである．

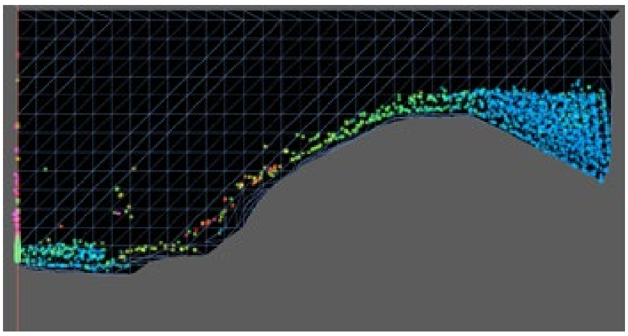


図-13　部分侵食パターン 40sec

修正前の結果と比べ滑らかな形状を保ちつつも，法面下部から侵食が始まる様子が確認できる．

次に全体侵食の再現を行ったが，全体侵食は限界せん断応力を低く設定することで，比較的簡単に再現できる．

図-14におけるシミュレーションは，限界せん断応力を $\tau_c = 1.2$，侵食強度を $K_\varepsilon = 0.05$ としたものである．天端部分でも最初期から侵食が始まり，全体に侵食が進んでいる．全体侵食パターンは比較的簡単な条件で再現することができ，滑らかな侵食結果を得ることができる．水平の地盤に対しても深い位置まで侵食が進むため，図-14では一定以下の標高の格子点について，侵食を無効としている．

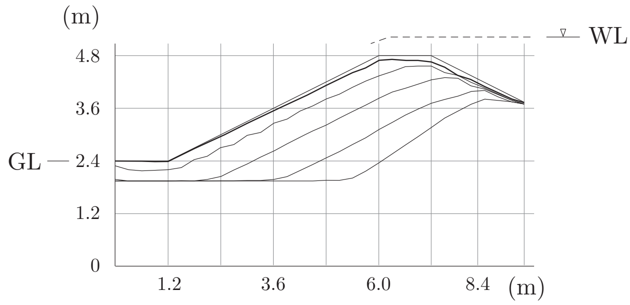


図-14　全体侵食パターン

5　地形への適用

ボクセルデータに対する侵食シミュレーションの作成と，越流破堤モデルによる検証を行ったが，このシミュレーションがリアリスティックな地形に対してどのような効果をもたらすのかについて観察を行う．

5.1　モデルの準備

ボクセルデータの初期地形を生成するが，これは主に地形表面の標高データと地下の3次元データの組合せによって形作られる．

地形表面を形成するため，World Machine 2 (World Machine Software, LLC http://www.world-machine.com/) を用いている．これは主にパーリンノイズ[13]を元として，加減算等の各種処理やセル・オートマトンによる侵食再現等を行うソフトウェアである．地形内部には3次元のパーリンノイズ[13]を用いることとした．主に地下の空洞として形成され，地表へ達し穴を空けることもある．

準備された地形に対し，流体粒子は降雨を再現する形で供給される．この時5つの粒子がセットとなり水滴を形成する．

5.2　山間湖モデル

このモデルは山岳状の地形をパーリンノイズを元に作成したものであり，高さマップでも表現可能な地形をボクセルデータへ変換している．図-15には初期形状にごく近い状態を示しているが，降雨した粒子が溜まり幾つかの湖を形成している．降雨が続けば溢水によって湖は接続され，またその過程で地形が侵食を受ける．これによって地形上も湖は一体化していき，図-16に示すように河川のような流れが生まれていく．

5.3　渓谷モデル

中央に谷を形成する高さマップと，地下の空洞によって構成された渓谷モデル(図-17)に対するシミュレーション結果を見る．渓谷モデルは今回の侵食シミュレーションの垂直面に対する効果を見ることを目的としている．

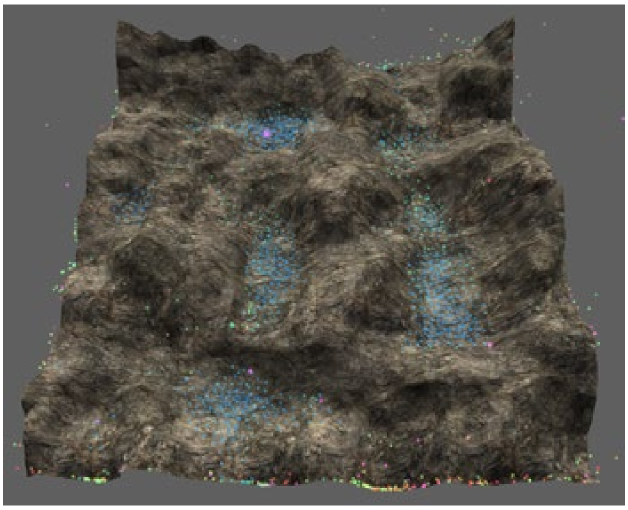


図-15　山間湖モデル 初期地形

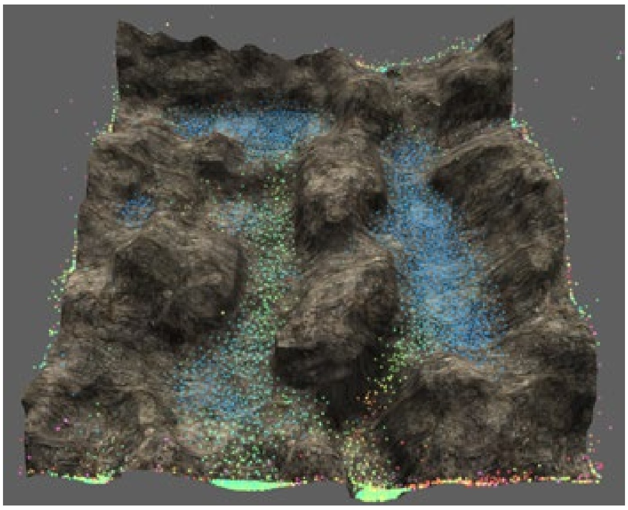


図-16　山間湖モデル 侵食後

着色された円形で示された部分は地下の空洞の大まかな位置を示している．粒子は降雨によって地形に降り注ぐが，谷底に集まって東へと向い，さらに地下空洞へ流れ込んだ後に計算領域外へ流れる．また図-18に南北方向に複数のラインで切った断面図を示す．薄い色で示された部分は1次計算での侵食部分であり，より暗い色で示された部分は2次計算での侵食部分である．

まず東西方向の断面(図-17右下)では谷底を流れる水流が，幾つかの滝を形成しつつ流れた結果による侵食を見ることができる．崖と水平面の組み合わさった断面形状を持ち，越流破堤実験で形成された地形と同等の侵食を示している．

次に南北方向の断面図v6を見ると，初期形状で排水経路である地下空洞へ向かって斜面が存在する事を確認できる．またv4断面は他の断面と比べても垂直部分が目立つが，1次侵食時と2次侵食時で壁面が後退していない．このように垂直な壁面が維持されている部分では，その下を谷底の水流が抉るように

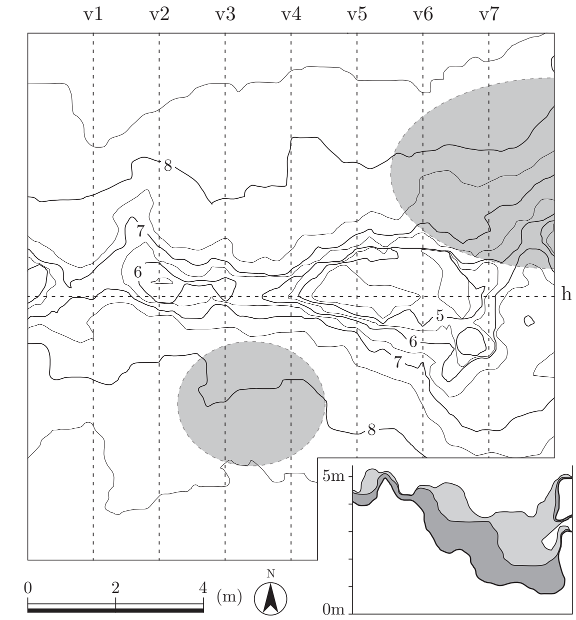


図-17　渓谷モデル

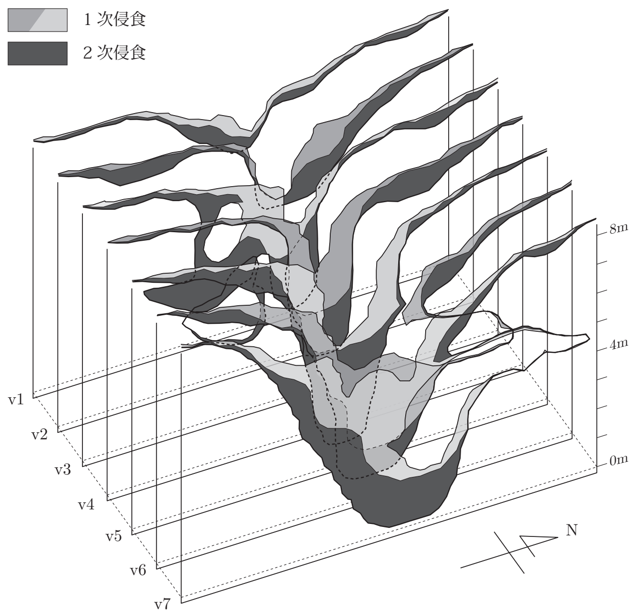


図-18　渓谷モデル 断面図

侵食する様子が僅かではあるものの確認できた．しかし谷底を下へ削る侵食の進行が早く，これが横方向への侵食が広がらない原因の1つと考えられる．

また図-19では渓谷と地下空洞の接続孔を粒子が流れ落ちている．流れは画面右奥から左手前に向かって渓谷を流れているため，左下へ向かって垂れ下がるように接続孔が拡大した経緯がある．

6　結論

本研究ではまずSPH法の実装を行ったが，粒子と地形の境界を計算する手法の確立が必要となった．ここでHarada(2007)[3]を参考とした計算を行ったが，圧力項と粘性項に関しては計測

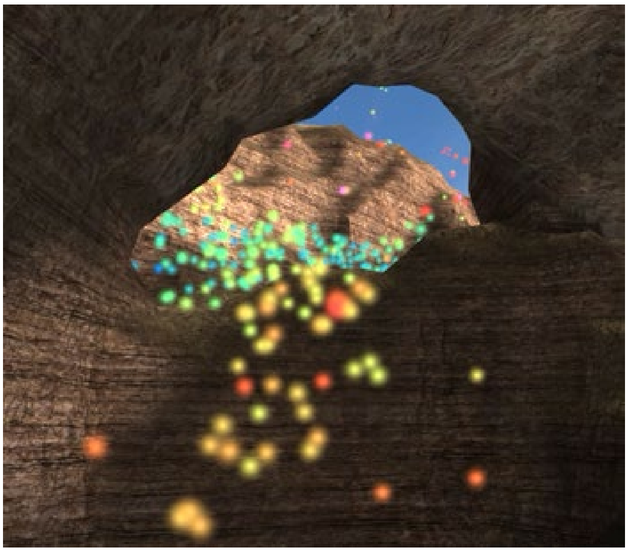


図-19　地下空洞へ流れる粒子

粒子による重み関数の計測結果が変動することに影響を受ける．次に壁と粒子の距離を得るため，ボクセルデータの等値面と粒子の距離を割り出す手法が必要であったが，本研究ではこれをボクセルデータの濃度値から直接求めることとした．結果的にこの壁境界計算手法は，粒子が地形の内側に入り込んだり，跳ね返された粒子が不安定なることもなく，以降の侵食計算を支えた．これらは侵食シミュレーションを3次元の地形に対して適用する際の計算負荷の増大を抑える点で大きな効果をもたらしたと考えられる．

侵食のメカニズムに関してはせん断応力による侵食モデルを採用し，これを格子点を中心とした近傍粒子探索によって，侵食量を割り出すこととした．また，この侵食計算の検証のため，堤防が越流によって受ける侵食についてのシミュレーションを行った．この越流破堤モデルによる検証を通じ，侵食計算について当初の予測以上の改善を行うことができ，特に壁境界計算において壁・粒子間の引力が調節可能となるなどした．

以上の組合せによって構成される侵食シミュレーションを用い，地形形状に対する侵食の効果について観察を行った．いくつかのモデルについてシミュレーションを行ったが，一部では越流破堤と同様の特徴的侵食を見せるなど，良好な結果が得られた．さらに垂直面に対する侵食について再現の可能性を見ることが出来たが，その発達に関しては課題が確認された．

最後に実行効率について，壁境界計算はSPH法計算からループ等を増やしておらず，侵食計算はSPH法の50倍のタイムステップで計算している．したがって実行時間の増加は線形で，ほぼ粒子探索の効率に准している．

またGPUプログラミングなどの高速化手法を用いなかったため，よりデータ量の大きい地形に適用するには課題を残す事となった．しかし基本的な階層までをC# のスクリプトによって取り扱ったことが，研究の進行を大きく助けたと考えている．

参考文献

1) 酒井譲, 楊宗億, and 丁泳鏞. Sph 法による非圧縮粘性流体解析手法の研究 (流体工学, 流体機械). **日本機械学會論文集**. *B* **編**, 70(696):1949–1956, 2004.

2) Krištof Peter, Bedrich Beneš, J Křivánek, and Ondrej Št'ava. Hydraulic erosion using smoothed particle hydrodynamics. In *Computer Graphics Forum*, volume 28, pages 219–228. Wiley Online Library, 2009.

3) 原田隆宏 and 越塚誠一. Sph における壁境界計算手法の改良 (コンピュータグラフィックス). **情報処理学会論文誌**, 48(4):1838–1846, 2007.

4) 藤澤和謙, 村上章, and 西村伸一. 砂・粘土混合材料の侵食速度測定と室内越流破堤実験. **農業農村工学会論文集**, 79(3):45–55, 2011.

5) Müller Matthias, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association, 2003.

6) Becker Markus and Matthias Teschner. Weakly compressible sph for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 209–217. Eurographics Association, 2007.

7) Fujisawa Makoto. Sph法の実装(gpu実装含む) 2014-11-18 アクセス, 2014.

8) Green Simon. Particle simulation using cuda. *NVIDIA Whitepaper, December 2010*, 2010.

9) Desbrun Mathieu and Marie-Paule Gascuel. *Smoothed particles: A new paradigm for animating highly deformable bodies.* Springer, 1996.

10) Adam Benoit. voxel-terrain-unity voxel terrain asset for unity 2014-6-24 アクセス, 2013.

11) Matthew Fisher. Matt's webcorner marching cubes 2015-1-6 アクセス, 2014.

12) Wojtan Chris, Mark Carlson, Peter J Mucha, and Greg Turk. Animating corrosion and erosion. In *Proceedings of the Third Eurographics conference on Natural Phenomena*, pages 15–22. Eurographics Association, 2007.

13) Perlin Ken. Improving noise. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 681–682. ACM, 2002.