2. 2次元粒子法シミュレーション（＋少しだけOpenGL）

茨城大学工学部
教授　乾　正知

茨城大学
Ibaraki University

smartec

準備

計算結果を可視化するためにOpenGLを利用する.

OpenGL

- 3次元コンピュータグラフィックス用の標準的なライブラリ.
  - 特にCADやアート，アニメーション分野（ゲーム以外の分野）で広く利用されている.
- OpenGLは仕様がオープンに決められており，企業から独立した団体が仕様を管理している.
- OpenGLはWindowsはもちろん，UNIX，Linux，Macといったあらゆるプラットフォームで利用可能.
  - OpenGLを用いて作成されたプログラムは互換性が高い.
  - Windowsには標準的に搭載されている.

glutとfreeglut

- **OpenGL**には，ウィンドウ生成やGUI構築のための機能は用意されていない．
- これらについては，プログラマーがプラットフォームに合わせて用意する必要がある．
  - Windows環境：MFC．
  - UNIXやLinux：X window．
- glutは非常にシンプルな，プラットフォームから独立したOpenGLアプリ用のユーザインターフェイス構築ツール（toolkitと呼ばれる）．
  - 研究の世界ではOpenGL+glutで簡易アプリを開発することが一般的．
- glutの開発は最近停滞しており，代わりにfreeglutを使うことが多い．

freeglutの導入

- freeglutは頻繁にバージョンアップされている. 最新版は以下のサイトから無料で入手できる.

  http://sourceforge.net/projects/freeglut/files/latest/download

- ただしソースコードのみなので，自分でビルドする必要がある. “download”をチェック.
- 導入法については参考書を見て欲しい.
- 今回はfreeglutが導入済であることを仮定して説明を進める.

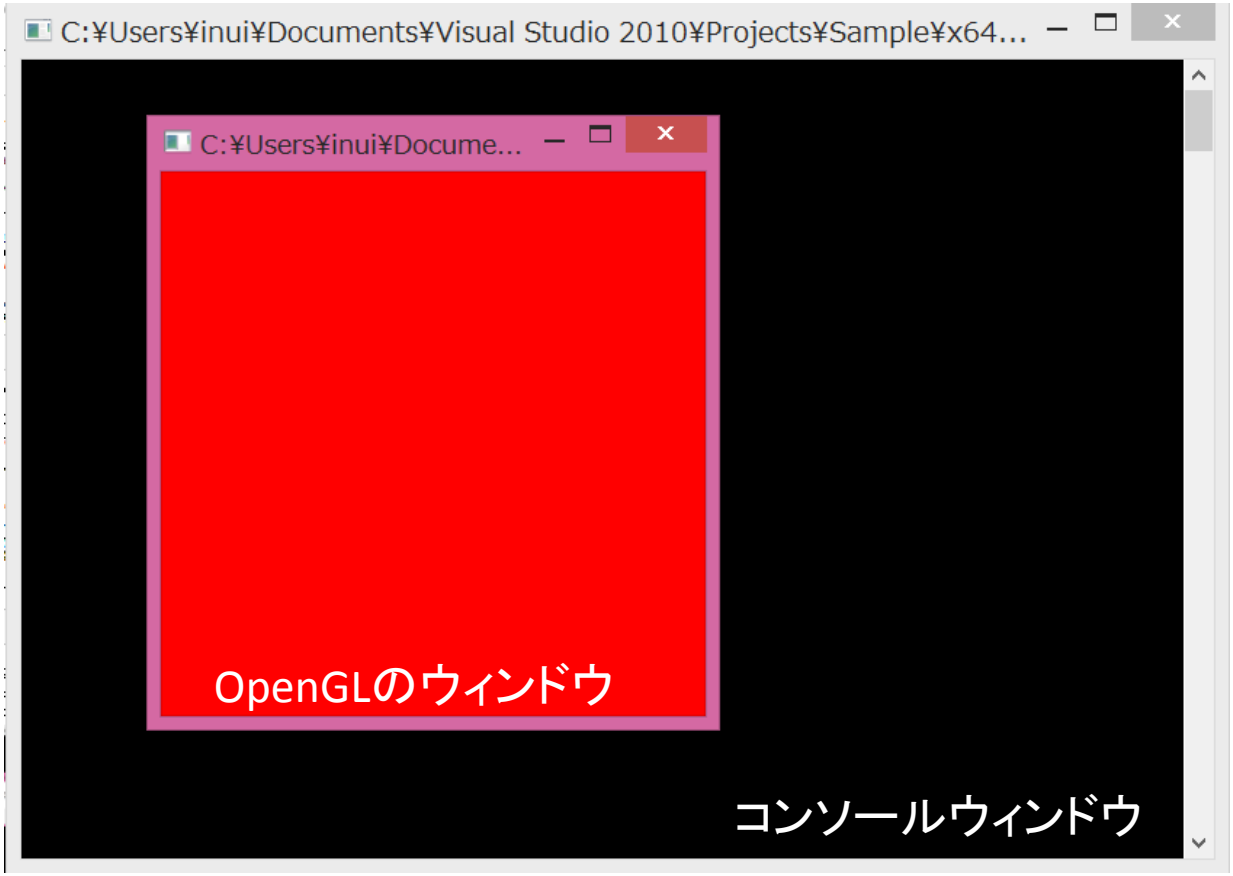
簡単なOpenGL+glutプログラム（1/2）

- 以下の内容のSample.cppを作成.

```
#include <GL/freeglut.h>   freeglutのヘッダーファイル

void display(void)
{
          glClear(GL_COLOR_BUFFER_BIT);
          glFlush();
}

void initGL(void)
{
          glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
}

int main(int argc, char *argv[])
{
          glutInit(&argc, argv);
          glutInitDisplayMode(GLUT_RGBA);
          glutCreateWindow(argv[0]);
          glutDisplayFunc(display);
          initGL();
          glutMainLoop();
          return 0;
}
```

簡単なOpenGL+glutプログラム（2/2）

- コンパイル，リンクし（ビルドし）実行すると以下の2個のウィンドウが現れる．



main関数(1/3)

- glutの初期設定を行う関数が起動.

```
int main(int argc, char *argv[])
{
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_RGBA);
 glutCreateWindow(argv[0]);
 glutDisplayFunc(display);
 initGL();
 glutMainLoop();
  return 0;
}
```

Glutの初期化.

フレームバッファ(画面)の初期化.

今回定義したOpenGL関係の初期化.

main関数(2/3)

- OpenGLによる画像表示用ウィンドウの生成.

```
int main(int argc, char *argv[])
{
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_RGBA);
 glutCreateWindow(argv[0]);
 glutDisplayFunc(display);
 initGL();
 glutMainLoop();
  return 0;
}
```

OpenGL用のウィンドウの生成. 引数にはウィンドウのタイトルを与える.

main関数(3/3)

- イベントに応じて駆動するコールバック関数の設定.

```
int main(int argc, char *argv[])
{
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_RGBA);
 glutCreateWindow(argv[0]);
 glutDisplayFunc(display);
 initGL();
 glutMainLoop();
  return 0;
}
```

ウィンドウに何らかの操作(サイズ変更など)を行うと,display関数が起動する.

コールバック関数.

イベント待ちループ.

initGL関数

- OpenGL関係の初期化処理を行う関数．プログラマが定義．
- 文法：

  void glClearColor(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

  red，**green**，**blue**には，0.0～1.0の浮動小数点値を与える．0.0を与えるとその色成分はゼロ，また1.0を与えるとその色成分はフル．

  red，green，blueが全て0.0だと背景色は黒，全て1.0だと背景色は白．

  alpha成分は色の透明度1.0（＝完全に不透明）を設定．

```
void initGL(void)
{
        glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
}
```

画面を初期化する色の設定．
背景色の設定と考えてよい．

display関数（1/3）

- ウィンドウに何らかの操作（イベント）が発生すると自動的に起動する，表示用の「コールバック関数」.
- 文法：

  void glClear(GLbitfield mask); maskにはビットパターンのマスクのORが与えられる．マスクにはGL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BITなどがある.

  GL_COLOR_BUFFER_BITがマスクに含まれていると，ウィンドウがglClearColor関数が設定する色で染められる.

  void glFlush(void); この関数が起動すると，バッファに溜め込まれていたOpenGLの全ての関数が強制的に実行される.

```
void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glFlush();
}
```

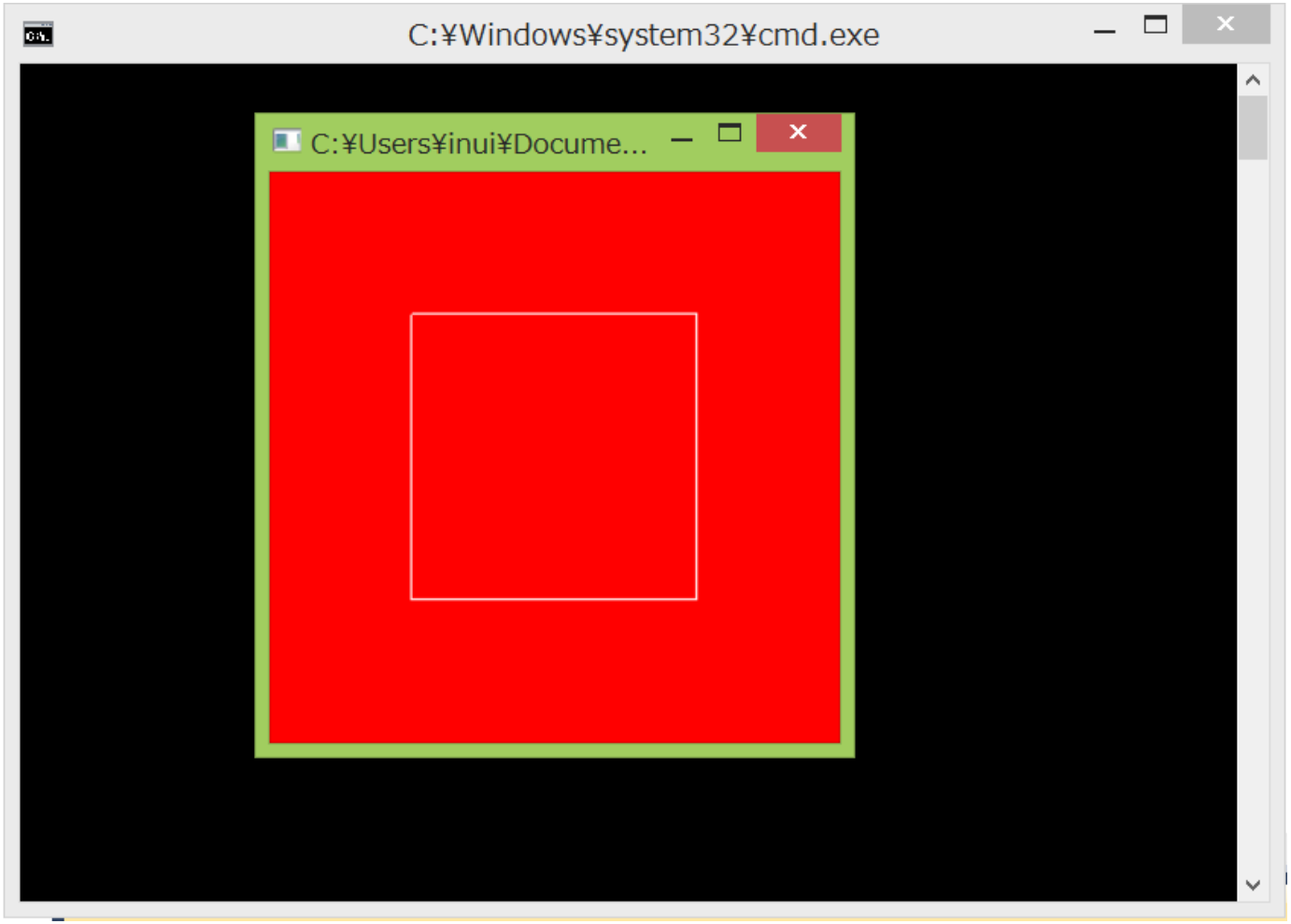
display関数（2/3）

- display関数を修正すると，OpenGLウィンドウに描かれる画像を変更できる.
- display関数を右に示すように変更.
- 描かれる画像と右に示された関数の関係を類推してみよう.

```
void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0f, 1.0f, 1.0f);
        glBegin(GL_LINES);

        glVertex3f(0.5f, 0.5f, 0.0f);
        glVertex3f(- 0.5f, 0.5f, 0.0f);

        glVertex3f(- 0.5f, 0.5f, 0.0f);
        glVertex3f(- 0.5f, - 0.5f, 0.0f);

        glVertex3f(- 0.5f, - 0.5f, 0.0f);
        glVertex3f(0.5f, - 0.5f, 0.0f);

        glVertex3f(0.5f, - 0.5f, 0.0f);
        glVertex3f(0.5f, 0.5f, 0.0f);

        glEnd();
        glFlush();
}
```

display関数（3/3）

- 今度は以下に示すような画像が描かれる.



GL_POINTSの利用（1/3）

- 以下の行を**Sample.cpp**に追加.

```
#define X 0
#define Y 1
#define Z 2

unsigned int num_points = 5;
double point[][3] = {{0.5, 0.5, 0.0},
 {-0.5, 0.5, 0.0}, {-0.5, -0.5, 0.0},
 {0.5, -0.5, 0.0}, {0.0, 0.0, 0.0}};
```

これらのマクロを使ってpoint[][0], point[][1], point[][2]をpoint[][X], point[][Y], point[][Z]と表記.

5個の点を定義.

GL_POINTSの利用(2/3)

- **initGL関数とdisplay関数を以下のように更新.**

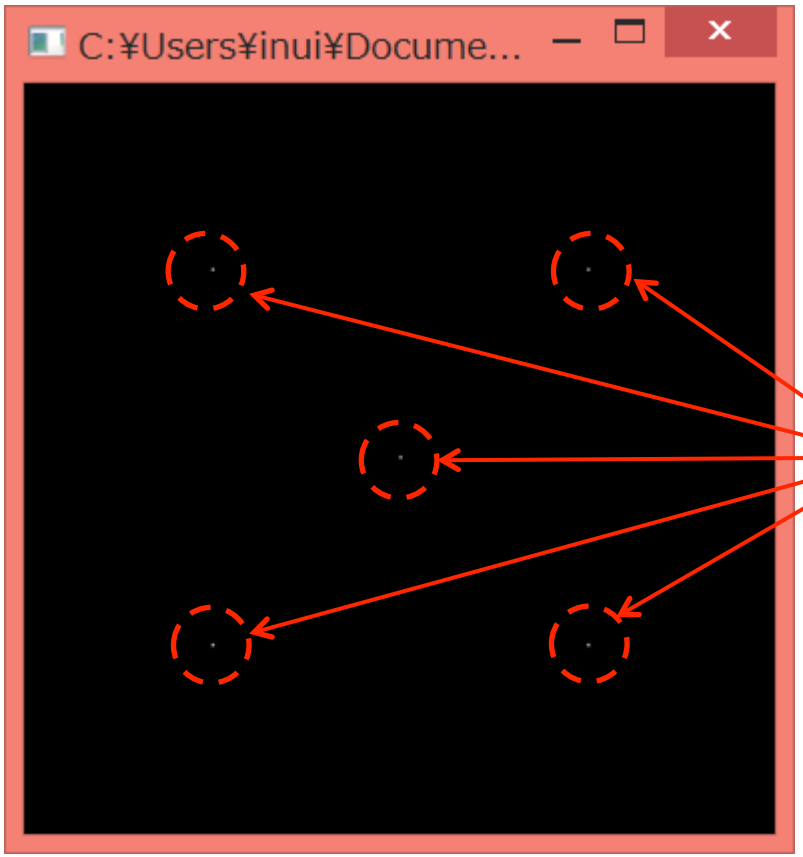
```
void display(void)
{
        unsigned int i;
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
        for (i = 0; i < num_points; i++)
                glVertex3d(point[i][X], point[i][Y],
                        point[i][Z]);
        glEnd();
        glFlush();
}

void initGL(void)
{
        glClearColor(0.0, 0.0, 0.0, 1.0);
}
```

GL_POINTSの利用

glVertex3dはOpenGLの関数で座標値(x, y, z)をGPUへ転送

←背景色は黒

GL_POINTSの利用(3/3)



座標(0.5, 0.5, 0.0),
(-0.5, 0.5, 0.0), (-0.5 -0.5, 0.0),
(0.5, -0.5, 0.0), (0.0, 0.0, 0.0)
の5個の点を白で表示.
色を特に指定しないと白が
使われる.

色の指示(1/2)

- **glColor3f**関数を用いて図形に色付けする.
- 文法:

  void glColor3f(GLfloat red, GLfloat green, GLfloat blue);

  red, **green**, **blue**には0.0～1.0の浮動小数点値を与える.

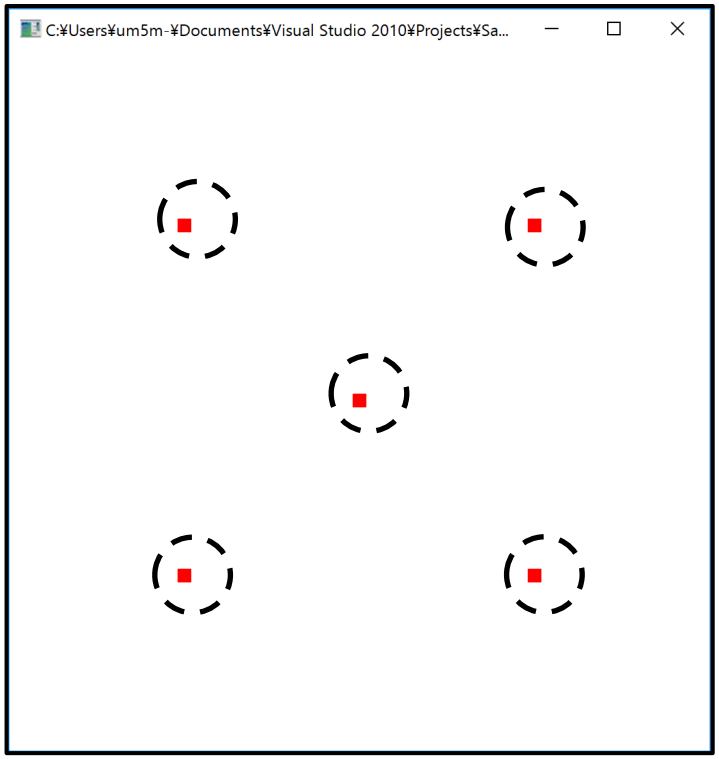
  0.0を与えるとその色成分はゼロ, また1.0を与えるとその色成分はフル.

  red, **green**, **blue**が全て0.0だと黒色, 全て1.0だと白色.

- 一度色を指示すると, **glColor3f**関数などを用いて別な色を指示するまで, 全ての図形は同じ色で染められる.

色の指示(2/2)

```
void display(void)
{
        unsigned int i;
        glClear(GL_COLOR_BUFFER_BIT);
        glPointSize(10.0f);
        glBegin(GL_POINTS);
        glColor3f(1.0f, 0.0f, 0.0f);
        for (i = 0; i < num_points; i++)
                glVertex3d(point[i][X],
                        point[i][Y],
                        point[i][Z]);
        glEnd();
        glFlush();
}
```



点を拡大し，点の色として赤を指定した例.

背景色は白に変更した.

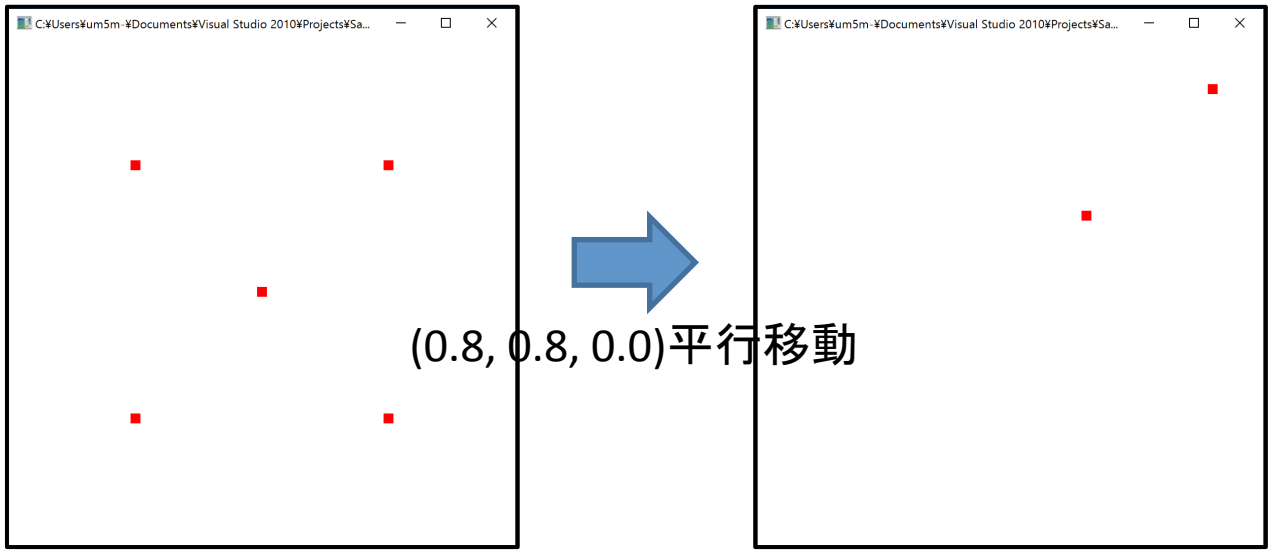
描画範囲の変更(1/4)

- これまで図形の描画範囲は議論してこなかった.
- 頂点の座標を以下のように少しずらすと…

```
double point[][3] = {{0.5, 0.5, 0.0}, {-0.5, 0.5,
0.0}, {-0.5, -0.5, 0.0}, {0.5, -0.5, 0.0}, {0.0, 0.0,
0.0}};
```

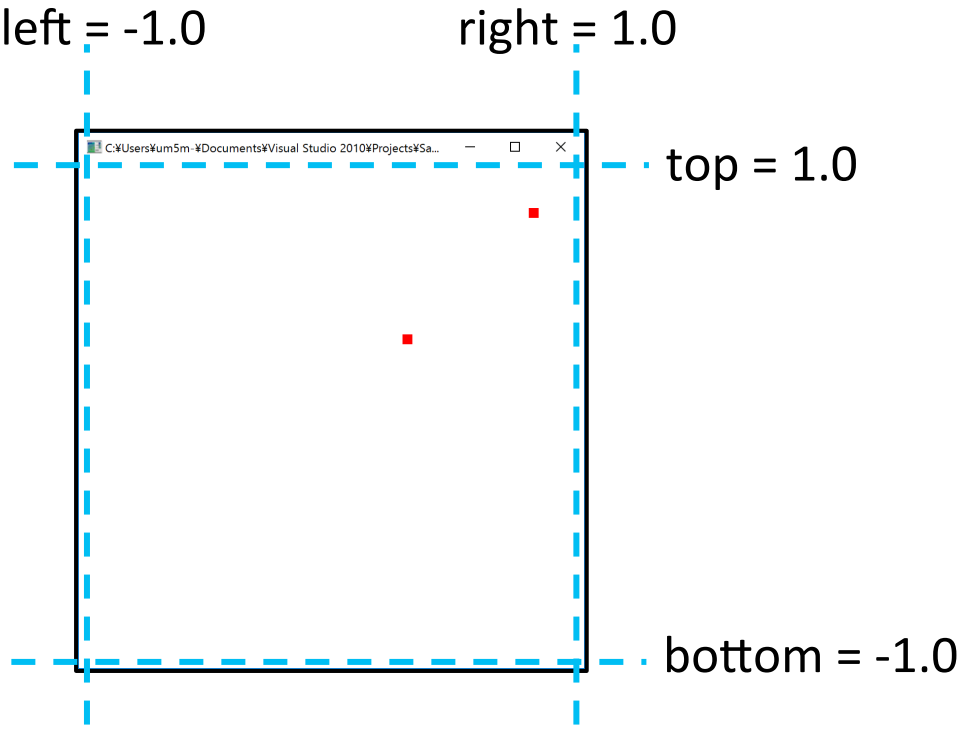
↓ 座標を(0.8, 0.8, 0.0)平行移動.

```
double point[][3] = {{1.3, 1.3, 0.0}, {0.3, 1.3, 0.0},
{0.3, 0.3, 0.0}, {1.3, 0.3, 0.0}, {0.8, 0.8, 0.0}};
```



描画範囲の変更(2/4)

- 初期設定では，OpenGLは(-1.0, -1.0)から(1.0, 1.0)までの，正方形領域を描くようになっている．
- 座標をずらすと図形の一部がはみ出してしまい，ウィンドウ内に描かれない．



描画範囲の変更(3/4)

- 描画範囲の変更には**glOrtho**関数を用いる．この関数は，平行投影を用いて3次元座標(x, y, z)を2次元座標(X, Y)へ変換する．平行投影については次回解説．
- 文法：

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,

GLdouble top, GLdouble nearVal, GLdouble farVal);

left は，描画範囲のx座標の最小値，**right**はx座標の最大値，**bottom**は描画範囲のy座標の最小値，**top**はy座標の最大値．**nearVal**と**farVal**にはとりあえず-100.0と100.0与えておく．

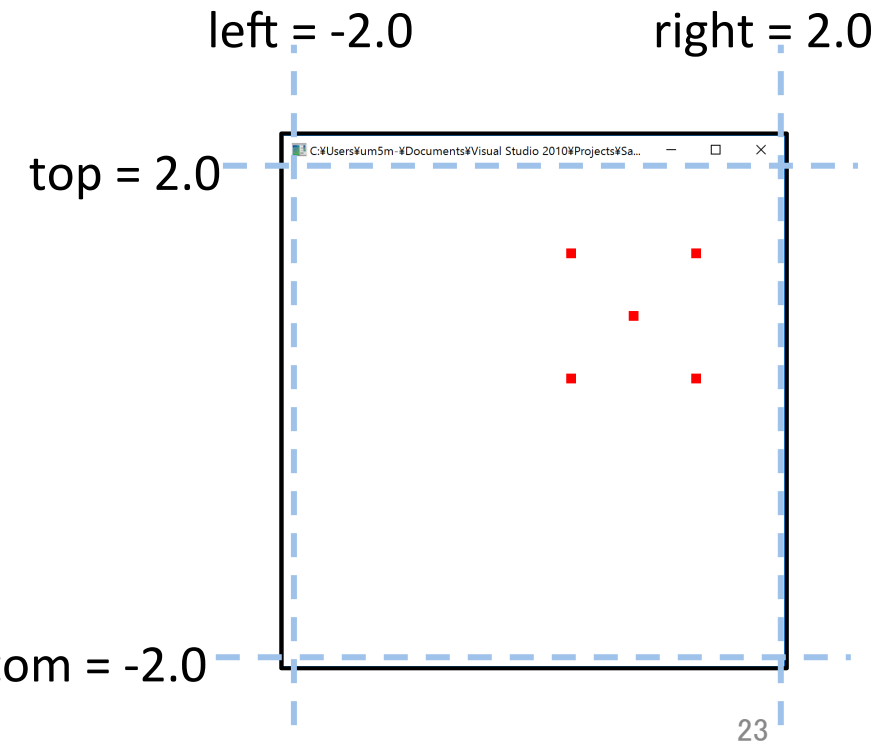
- **glOrtho**関数を起動する前に，以下の2つの関数を起動しておく．これも詳しくは次回．

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

描画範囲の変更(4/4)

```
void display(void)
{
        unsigned int i;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(-2.0, 2.0, -2.0, 2.0, -100.0, 100.0);
        glClear(GL_COLOR_BUFFER_BIT);
        glPointSize(10.0f);
        glBegin(GL_POINTS);
        glColor3f(1.0f, 0.0f, 0.0f);
        for (i = 0; i < num_points; i++)
                glVertex3d(point[i][X],
                        point[i][Y],
                        point[i][Z]);
        glEnd();
        glFlush();
}
```



Resizeコールバック関数

- 以下に示すresize関数を定義し，これをウィンドウの変形操作に応じて起動するコールバック関数として登録.

```
unsigned int window_width, window_height;
```

現在のウィンドウサイズを記録する大域変数を用意

```
void resize(int w, int h)
{
        printf("Size %d x %d\n", w, h);
        window_width = w;
        window_height = h;
}
```

この関数をコールバック関数として起動すると，ウィンドウサイズをwidthとheightに記録し，プリントアウトする

```
int main(int argc, char *argv[])
{
        ....
        glutDisplayFunc(display);
        glutReshapeFunc(resize);
        initGL();
        ....
}
```

Resize関数を，glutReshapeFunc関数を用いて，ウィンドウの変形に応じて起動するコールバック関数として登録

準備:ビューポート変換(1/2)

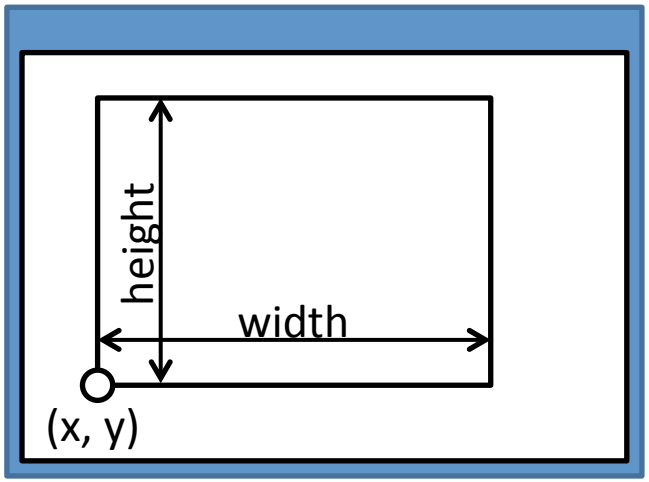
- 表示用の**display**関数に**glViewport**関数を追加.
- 文法:

  Void glViewport(Glint x, Glint y, Glsizei width, Glsizei height);

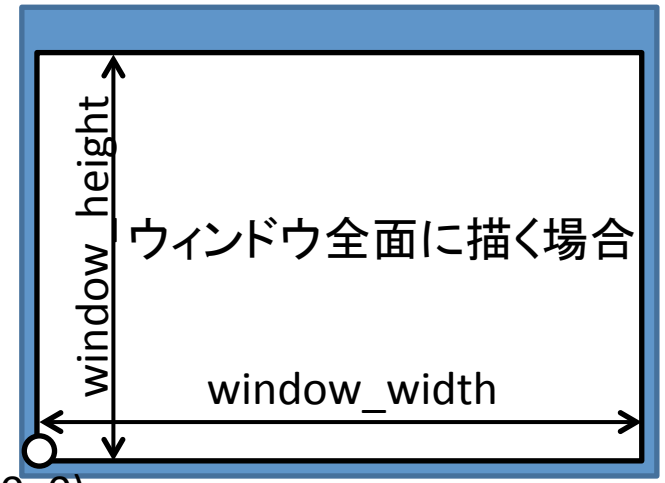
  OpenGLの生成画像をウィンドウの指定された範囲に描く.

  (x, y)は画像の左下隅のウィンドウ内の位置(単位はピクセル数).

  widthと**height**は画像の横と縦の範囲.



glViewport関数のパラメータ



glViewport(0, 0, window_width, window_height);

ビューポート変換(2/2)

```
void display(void)
{
        ....
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(-2.0, 2.0, -2.0, 2.0, -100.0, 100.0);
        glViewport(0, 0, window_width, window_height);
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
        ....
}
```

ウィンドウの初期化（1/2）

- 図形を表示するウィンドウのサイズや位置の変更を行うには，以下のglut関数を用いる．**main**関数で**glutCreateWindow**関数の前に起動する.
- 文法：

  void glutInitWindowSize(int width, int height);

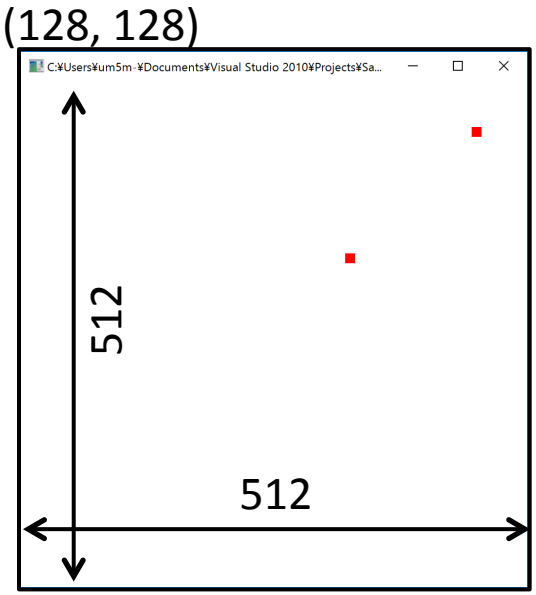
  生成するウィンドウの初期サイズを**width** x **height**に設定．単位はピクセル数.

  void glutInitWindowPosition(init origin_x, init origin_y);

  生成するウィンドウの左上隅の初期位置(**origin_x**, **origin_y**)を与える．位置は画面の左上隅から測る．単位はピクセル数.

ウィンドウの初期化(2/2)

```
int main(int argc, char *argv[])
{
    glutInitWindowPosition(128, 128);
    glutInitWindowSize(512, 512);
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);
    glutCreateWindow(argv[0]);
    glutDisplayFunc(display);
    initGL();
    glutMainLoop();
    return 0;
}
```



glutInitWindowPosition関数とglutInitWindowSize関数は，必ずglutCreateWindow関数の前に起動する．

2次元粒子法シミュレーション

概要

- 粒子法シミュレーション：物理現象，特に流体の物理的な挙動を，力学的な場に置かれた粒子の運動で解析する手法.
- CUDAは粒子法シミュレーションに向いている.
  - 各スレッドが1粒子の挙動解析を分担.
  - 全粒子の挙動の解析を並列処理.
- 簡単な粒子法シミュレーションのCUDAプログラムの実現. 粒子間の相互作用は扱わない（相互作用の扱いは「上級編」で議論）.

粒子法シミュレーションとは

- ある力学的な制約に基づく粒子の挙動を解析し，物理現象を可視化する手法.

  （例）水の流れに浮かぶ落ち葉の動きを解析することで，水の動きを知る.

- 位置が時間の関数($x(t)$, $y(t)$)である粒子を考える. 粒子の速度が，以下の微分方程式で与えられているものとする;

$$dx/dt = u(x, y, t)$$

$$dy/dt = v(x, y, t)$$

この時，時間tにおける$x(t)$と$y(t)$を求めたい.

微分方程式の数値解法

- 粒子の現在位置を(x_n, y_n)とする．微小時間Δt後の粒子の位置(x_{n+1}, y_{n+1})を求めることを繰り返す．

- **オイラー法（簡単だが精度が低い）：**

$$x_{n+1} = x_n + u(x_n, y_n, t_n)\Delta t$$

$$y_{n+1} = y_n + v(x_n, y_n, t_n)\Delta t$$

- **4段のルンゲクッタ（Runge-Kutta）法**：より高精度な計算が可能．

4段のルンゲクッタ法

$$x_{n+1} = x_n + (p_1 + 2 * p_2 + 2 * p_3 + p_4)/6\ dt$$

ただし

1段目 ⟶ $p_1 = u(x_n, y_n, t)$

2段目 ⟶ $p_2 = u(x_n+1/2p_1dt, y_n+1/2q_1dt, t+1/2dt)$

3段目 ⟶ $p_3 = u(x_n+1/2p_2dt, y_n+1/2q_2dt, t+1/2dt)$

4段目 ⟶ $p_4 = u(x_n+p_3dt, y_n+q_3dt, t+dt)$

$$y_{n+1} = y_n + (q_1 + 2 * q_2 + 2 * q_3 + q_4)/6\ dt$$

ただし

1段目 ⟶ $q_1 = v(x_n, y_n, t)$

2段目 ⟶ $q_2 = v(x_n+1/2p_1dt, y_n+1/2q_1dt, t+1/2dt)$

3段目 ⟶ $q_3 = v(x_n+1/2p_2dt, y_n+1/2q_2dt, t+1/2dt)$

4段目 ⟶ $q_4 = v(x_n+p_3dt, y_n+q_3dt, t+dt)$

今回の問題

- 中心が(0.5, 0.25)，一辺の長さが0.5の正方形領域内に与えられた，1024 x 1024 = 1,048,576個の粒子を考える.
- 各粒子の挙動が以下の微分方程式に従うときの，粒子群の動きを可視化する.

$$dx/dt = u(x, y, t)$$

$$dy/dt = v(x, y, t)$$

ただし

$$u(x,y,t) = -2\cos(\pi t/8)\sin^2(\pi x)\cos(\pi y)\sin(\pi y)$$
$$v(x,y,t) = 2\cos(\pi t/8)\cos(\pi x)\sin(\pi x)\sin^2(\pi y)$$

プログラミングの流れ

- 2次元の簡易な粒子法のプログラムを以下の手順で作成.
  1. 処理をCUDAを使わずCのみで実装.
     - 粒子位置の初期化
     - 微分方程式の扱いとルンゲクッタ法
     - OpenGLによる粒子群の描画
  2. 処理中の粒子ごとの繰り返し処理を，CUDAによる並列処理に置き換え.
     - ホストとデバイス間のデータ転送
     - グリッド，ブロックの定義，カーネル関数の実装

Cでの実装

準備

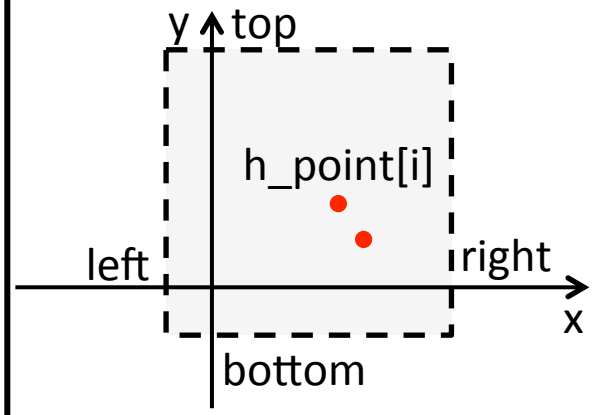
- 必要なヘッダーファイルなどを，以下のように指示．

```
#include <stdio.h>
#include <math.h>

#include <gl/freeglut.h>
#include <cuda_runtime.h>

#define INIT_X_POS 128
#define INIT_Y_POS 128
#define INIT_WIDTH 512
#define INIT_HEIGHT 512

unsigned int window_width;
unsigned int window_height;

double left = -0.25;
double right = 1.25;
double bottom = -0.25;
double top = 1.25;
```

← OpenGL用

← 後で使うCUDA用

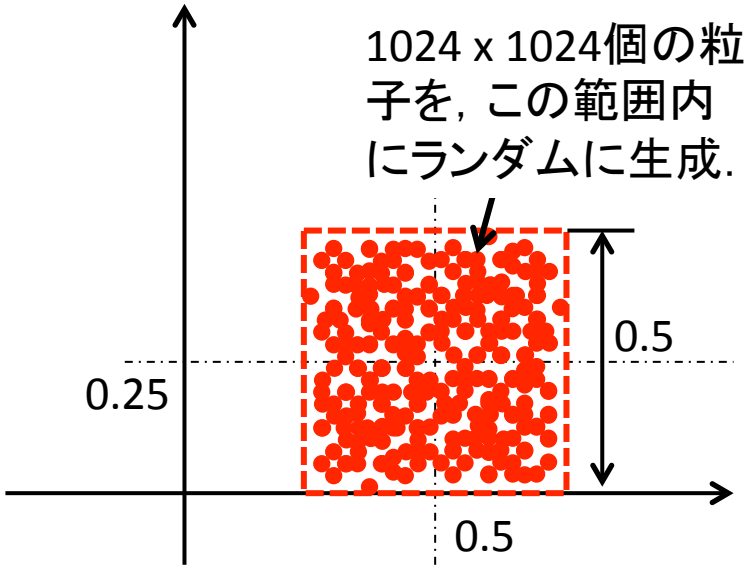
図形の描画範囲



left = -0.25
right = 1.25
bottom = -0.25
top = 1.25

初期化

- 初期位置への粒子の配置.




```
// 粒子数とその位置情報.
#define NUM_POINTS (1024 * 1024)
float h_point[NUM_POINTS][2];

// 処理時間と時間刻み.
float anim_time = 0.0f;
float anim_dt = 0.01f;

// 粒子を初期位置に配置.
void setInitialPosition(void)
{
        unsigned int i;
        srand(12131);
        for (i = 0; i < NUM_POINTS; i++) {
                h_point[i][0] = (float)rand() / RAND_MAX * 0.5f + 0.25f;
                h_point[i][1] = (float)rand() / RAND_MAX * 0.5f;
        }
}
```

0.0～0.5の範囲の乱数.

微分方程式の扱い

$$u(x,y,t) = -2\cos(\pi t/8)\sin^2(\pi x)\cos(\pi y)\sin(\pi y)$$
$$v(x,y,t) = 2\cos(\pi t/8)\cos(\pi x)\sin(\pi x)\sin^2(\pi y)$$

微分方程式をマクロで扱う

```
#define PI 3.141592

// CPU処理.
#define h_U(x, y, t)
(- 2.0f * (float)cos(PI * (t) / 8.0f)
 * (float)sin(PI * (x)) * (float)sin(PI * (x))
 * (float)cos(PI * (y)) * (float)sin(PI * (y)))

#define h_V(x, y, t)
(2.0f * (float)cos(PI * (t) / 8.0f)
 * (float)cos(PI * (x)) * (float)sin(PI * (x))
 * (float)sin(PI * (y)) * (float)sin(PI * (y)))
```

ルンゲクッタ法：Cによる実装（1/2）

```
// CPU用ルンゲ･クッタ法
void h_RungeKutta(int index,
  float (*pos)[2], float time, float dt)
// unsigned int index;
// float (*pos)[2];
// float time;
// float dt;
{
        float xn, yn, p1, q1, p2, q2,
p3,
              q3, p4, q4;
        float x, y, t;
        xn = pos[index][0];
        yn = pos[index][1];

        // 1段目.
        p1 = h_U(xn, yn, time);
        q1 = h_V(xn, yn, time);

        // 2段目.
        x = xn + 0.5f * p1 * dt;
        y = yn + 0.5f * q1 * dt;
        t = time + 0.5f * dt;
        p2 = h_U(x, y, t);
        q2 = h_V(x, y, t);

        // 3段目.
        x = xn + 0.5f * p2 * dt;
        y = yn + 0.5f * q2 * dt;
        t = time + 0.5f * dt;
        p3 = h_U(x, y, t);
        q3 = h_V(x, y, t);

        // 4段目.
        x = xn + p3 * dt;
        y = yn + q3 * dt;
        t = time + dt;
        p4 = h_U(x, y, t);
        q4 = h_V(x, y, t);

        // 粒子位置の更新.
        pos[index][0] = xn +
        (p1 + 2 * p2 + 2 * p3 + p4)
                / 6.0f * dt;
        pos[index][1] = yn +
        (q1 + 2 * q2 + 2 * q3 + q4)
                / 6.0f * dt;
}
```

現在の粒子位置

次の粒子位置への更新

ルンゲクッタ法：Cによる実装（2/2）

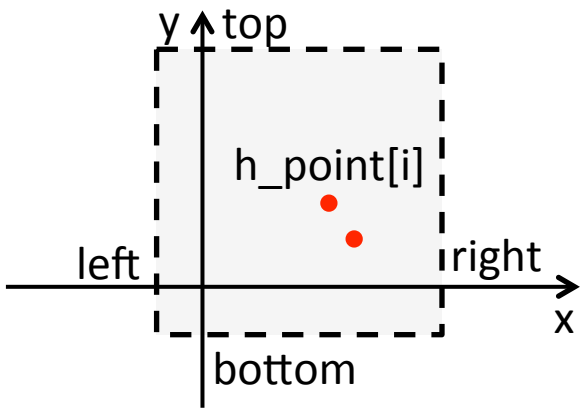
```
void runCPUKernel(void)
{
        launchCPUKernel(NUM_POINTS, h_point, anim_time, anim_dt);
        anim_time += anim_dt;
}


void launchCPUKernel(unsigned int num_particles, float (*pos)[2],
        float time, float dt)
// unsigned int num_particles;
// float (*pos)[2];
// float time;
// float dt;
{
        unsigned int i;

      for (i = 0; i < num_particles; i++)
            h_RungeKutta(i, pos, time, dt);
}
```

粒子ごとのルンゲクッタ処理の繰り返し
この繰り返しを後でスレッドに置き換える

描画処理，display関数

```
// 表示.
void display(void)
{
    unsigned int i;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(left, right, bottom, top, -100.0, 100.0);
    glViewport(0, 0, window_width, window_height);

  // 粒子位置の更新.
   runCPUKernel(); // CPU処理.

  // 点群の描画.
  glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_POINTS);
    for (i = 0; i < NUM_POINTS; i++)
        glVertex2fv(h_point[i]);
    glEnd();

    // 画像の更新.
  glutPostRedisplay();
}
```

left, right, bottom, topには画像の描画範囲を与える



NUM_POINTS個の点を赤色で描画

← 画面の強制書き換え

resize関数

- 画面サイズを取得するコールバック関数.
  プログラムの起動時に必ず呼び出される.

```
// リサイズ.
void resize(int width, int height)
{

        // ウィンドウサイズの取得.
        window_width = width;
        window_height = height;
}
```

keyboard関数とinitGL関数

- **keyboard**:キー入力に対応するコールバック関数.

```
// キー処理.
void keyboard(unsigned char key, int x, int y)
{
        switch (key) {
                case 'q':
                case 'Q':
                case '\033':
                        exit(0);
    }
}
```

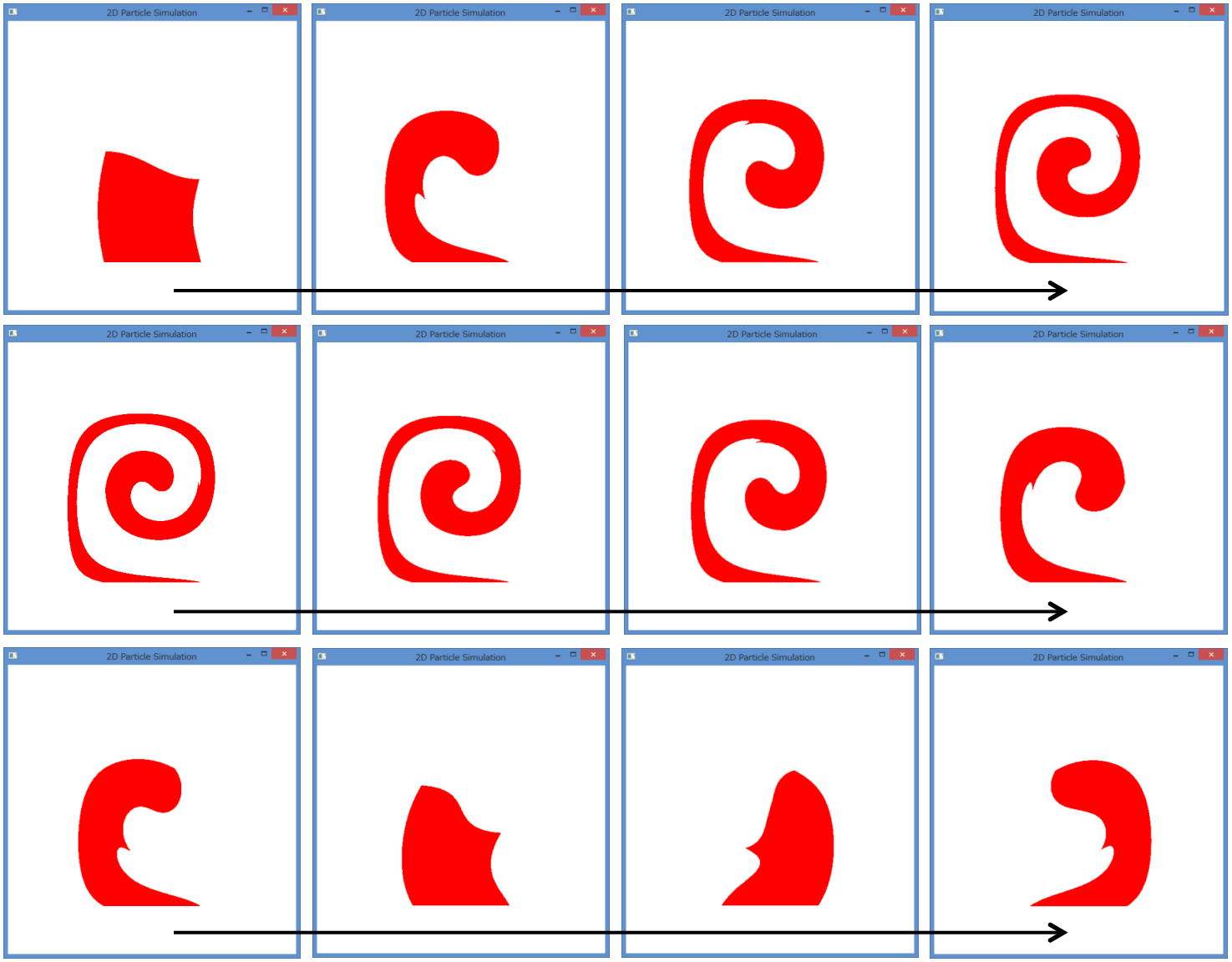
- **initGL**:OpenGL関係の初期化.

```
// OpenGL関係の初期設定.
bool initGL(void)
{
        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        return true;
}
```

main関数

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);
   glutInitWindowPosition(INIT_X_POS, INIT_Y_POS);
    glutInitWindowSize(INIT_WIDTH, INIT_HEIGHT);
    glutCreateWindow("2D Particle Simulation");
    glutDisplayFunc(display);
    glutReshapeFunc(resize);
    glutKeyboardFunc(keyboard);

  // 粒子を初期位置に配置.
  setInitialPosition();

    // OpenGLの設定.
    if (!initGL())
        return 1;

  // シミュレーションとアニメーション描画のループ.
  glutMainLoop();
    return 0;
}
```

プログラムの動き：非常に緩慢

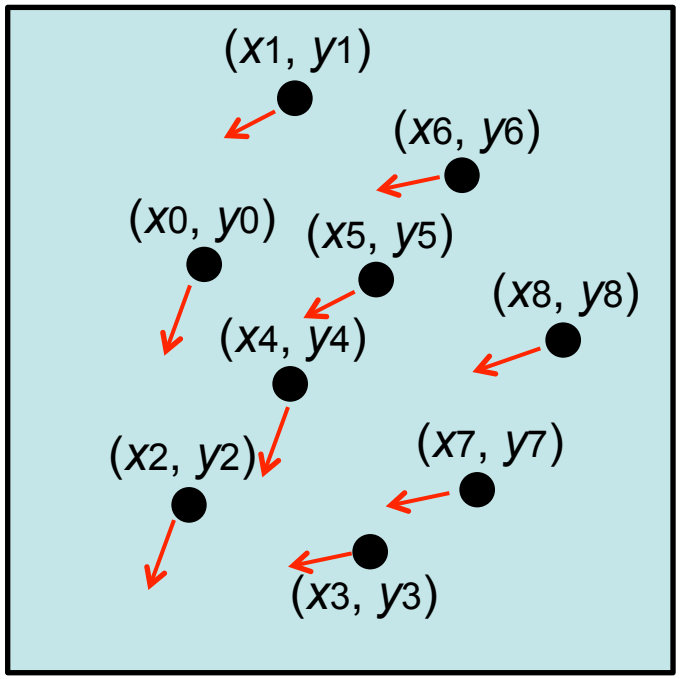


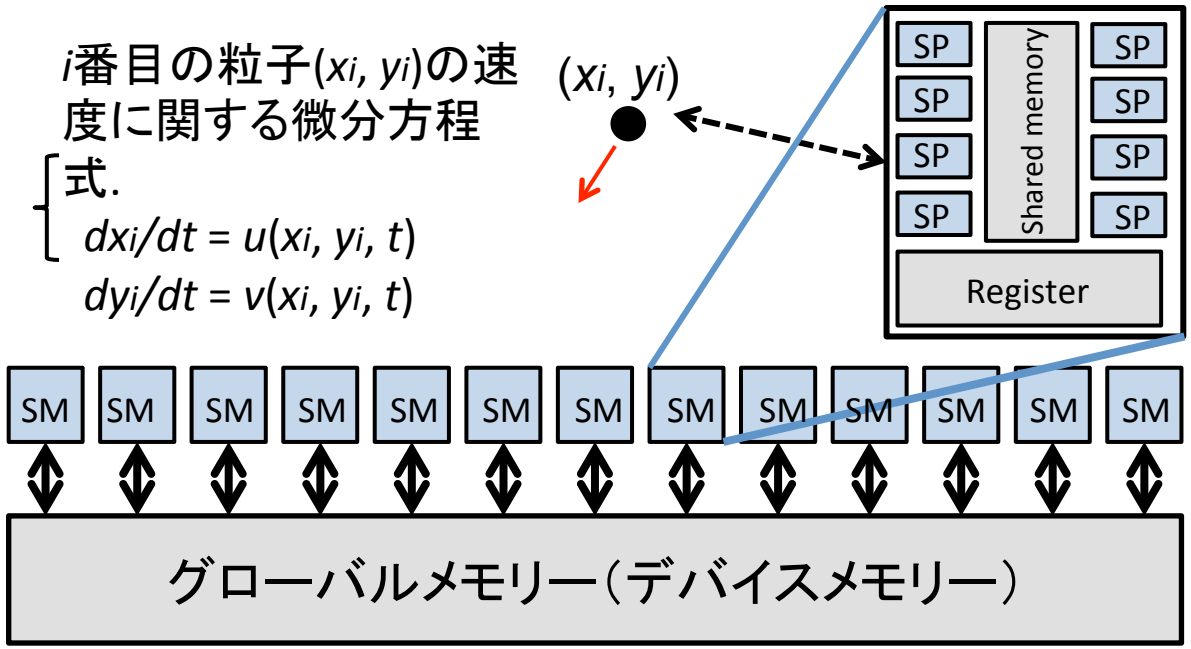
粒子の挙動の変化

CUDAによる並列処理への置き換え

CUDAを用いた実装

- CUDAは粒子法シミュレーションと相性がよい.
- 実装の方針：各粒子の挙動解析を，1つのスレッドに割り当てる．スレッドに対応するSPが計算.
- 全粒子の挙動を並列に計算できる.





GPU処理のための初期化

- 粒子の初期位置をデバイスメモリーへ転送.

```
#define NUM_POINTS (1024 * 1024)
float h_point[NUM_POINTS][2];
float (*d_point)[2];

void setInitialPosition(void)
{
        unsigned int i;
        srand(12131);
        for (i = 0; i < NUM_POINTS; i++) {
                h_point[i][0] = (float)rand() / RAND_MAX * 0.5f + 0.25f;
                h_point[i][1] = (float)rand() / RAND_MAX * 0.5f;
        }
        // GPU側にデータの初期位置を転送.
        cudaMalloc((void**)&d_point, NUM_POINTS * 2 * sizeof(float));
        cudaMemcpy(d_point, h_point,  NUM_POINTS * 2 * sizeof(float),
                        cudaMemcpyHostToDevice);
}
```

h_point（ホスト側データ）→ d_point（デバイス（GPU）側データ）

グリッドとブロックの定義(1/2)

```
void runGPUKernel(void)
{

        launchGPUKernel(NUM_POINTS, d_point, anim_time, anim_dt);

        cudaMemcpy(h_point, d_point, NUM_POINTS * 2 * sizeof(float),
                cudaMemcpyDeviceToHost);
```

表示のため，d_point(デバイス側データ)→ h_point(ホスト側データ)

```
        anim_time += anim_dt;
}

void launchGPUKernel(unsigned int num_particles, float (*pos)[2],
        float time, float dt)
// unsigned int num_particles;
// float (*pos)[2];
// float time;
// float dt;
{
    dim3 grid(num_particles / 512 + 1, 1);
    dim3 block(512, 1, 1);
    d_RungeKutta<<< grid, block >>>(num_particles, pos, time, dt);
}
```

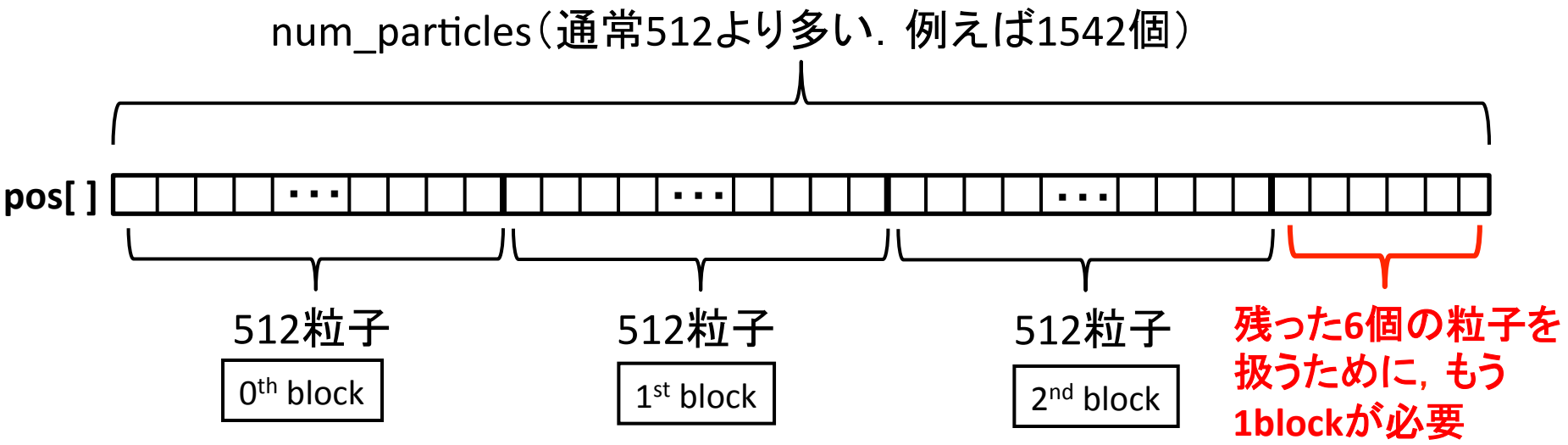
← num_particles / 512 + 1個のblock

← 各blockは512個のスレッド

グリッドとブロックの定義(2/2)

- 全ての粒子を処理できるように，十分な数のスレッドを生成する.

```
dim3 grid(num_particles / 512 + 1, 1);
dim3 block(512, 1, 1);
d_RungeKutta<<< grid, block >>>(num_particles, pos, time, dt);
```



1542 / 512 = 3(ブロック)・・・ 6(余りのスレッド)

微分方程式の扱い

$$u(x,y,t) = -2\cos(\pi t/8)\sin^2(\pi x)\cos(\pi y)\sin(\pi y)$$
$$v(x,y,t) = 2\cos(\pi t/8)\cos(\pi x)\sin(\pi x)\sin^2(\pi y)$$

```
#define PI 3.141592

// GPU処理.
#define d_U(x, y, t)
(- 2.0f * __cosf(PI * (t) / 8.0f)
 * __sinf(PI * (x)) * __sinf(PI * (x))
 * __cosf(PI * (y)) * __sinf(PI * (y)))

#define d_V(x, y, t)
(2.0f * __cosf(PI * (t) / 8.0f)
 * __cosf(PI * (x)) * __sinf(PI * (x))
 * __sinf(PI * (y)) * __sinf(PI * (y)))
```

高速計算のために __sinf()と__cosf()を利用.

ルンゲクッタ法：カーネル関数（1/2）

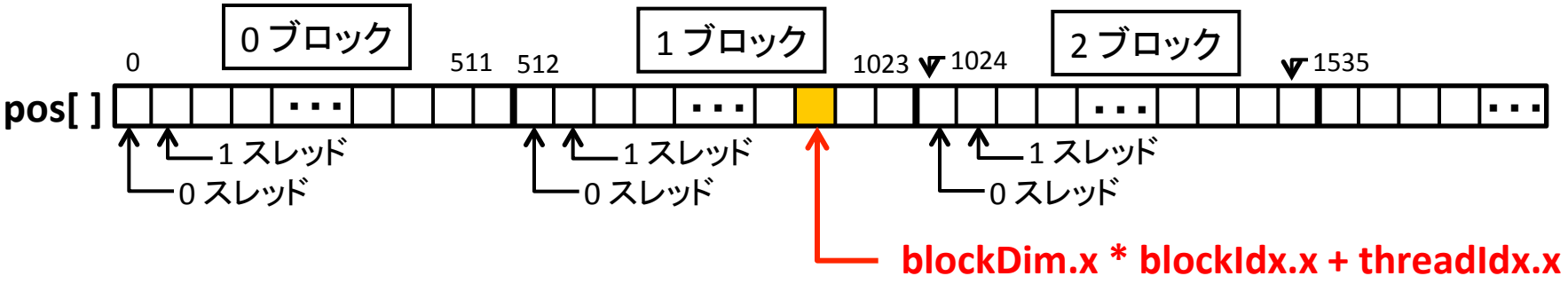
```
// GPU用ルンゲ･クッタ法
__global__ void d_RungeKutta(unsigned int
    num_particles, float (*pos)[2],
    float time, float dt)
// unsigned int num_particles;
// float (*pos)[2];
// float time;
// float dt;
{
    unsigned int index;
    float xn, yn, p1, q1, p2, q2;
    float p3, q3, p4, q4;
    float x, y, t;

    // 対象粒子の決定.
  index = blockDim.x * blockIdx.x
        + threadIdx.x;
    if (index >= num_particles)
        return;

    xn = pos[index][0];
    yn = pos[index][1];

    // 1段目.
  p1 = d_U(xn, yn, time);
    q1 = d_V(xn, yn, time);
```

```
// 2段目.
x = xn + 0.5f * p1 * dt;
y = yn + 0.5f * q1 * dt;
t = time + 0.5f * dt;
p2 = d_U(x, y, t);
q2 = d_V(x, y, t);

// 3段目.
x = xn + 0.5f * p2 * dt;
y = yn + 0.5f * q2 * dt;
t = time + 0.5f * dt;
p3 = d_U(x, y, t);
q3 = d_V(x, y, t);

// 4段目.
x = xn + p3 * dt;
y = yn + q3 * dt;
t = time + dt;
p4 = d_U(x, y, t);
q4 = d_V(x, y, t);

// 粒子位置の更新.
pos[index][0] = xn + (p1 + 2*p2
        + 2*p3 + p4) / 6.0f * dt;
pos[index][1] = yn + (q1 + 2*q2
        + 2*q3 + q4) / 6.0f * dt;
}
```

ルンゲクッタ法：カーネル関数（2/2）

```
__global__ void d_RungeKutta(unsigned int num_particles,
        float (*pos)[2], float time, float dt)
{
        unsigned int index;
        float xn, yn, p1, q1, p2, q2, p3, q3, p4, q4;
        float x, y, t;

        index = blockDim.x * blockIdx.x + threadIdx.x;
        if (index >= num_particles)
                return;

        xn = pos[index][0];
        yn = pos[index][1];
                ....
```

1個blockを追加したので，生成されるindexはnum_particles以上の可能性がある．



描画処理，display関数

```
// 表示.
void display(void)
{
    unsigned int i;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(left, right, bottom, top, -100.0, 100.0);
    glViewport(0, 0, window_width, window_height);

  // 粒子位置の更新.
   // runCPUKernel(); // CPU処理.
   runGPUKernel(); // GPU処理.

    // 点群の描画.
  glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_POINTS);
    for (i = 0; i < NUM_POINTS; i++)
        glVertex2fv(h_point[i]);
    glEnd();

    // 画像の更新.
  glutSwapBuffers();
    glutPostRedisplay();
}
```

粒子位置の更新処理を，CPU処理（runCPUKernel）からGPU処理（runGPUKernel）へ変更

後処理，cleanUp関数

- 計算後，確保してあったデバイス側のメモリーを解放する必要がある．
- 後処理用の関数を用意する．

```
// 後処理．
void cleanUp(void)
{
        cudaFree(d_point);
        cudaDeviceReset();
}
```

デバイス側のメモリーの解放と同時に，生成されたスレッドも，**cudaDeviceReset**関数で消去

main関数

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowPosition(INIT_X_POS, INIT_Y_POS);
    glutInitWindowSize(INIT_WIDTH, INIT_HEIGHT);
    glutCreateWindow("2D Particle Simulation");
    glutDisplayFunc(display);
    glutReshapeFunc(resize);
    glutKeyboardFunc(keyboard);
    atexit(cleanUp);

    // 粒子を初期位置に配置.
  setInitialPosition();

    // OpenGLの設定.
  if (!initGL())
        return 1;

    // アニメーション描画のループ.
  glutMainLoop();
        return 0;
}
```

cleanUp関数をatexit関数に登録すると，処理終了時に必ずcleanUpが実行される

発展

- 次の微分方程式による粒子の挙動を可視化せよ.

$$dx/dt = u(x, y, t)$$

$$dy/dt = v(x, y, t)$$

ただし

$$u(x,y,t) = \cos(\pi t/2)\sin(4\pi(x+0.5))\sin(4\pi(y+0.5))$$
$$v(x,y,t) = \cos(\pi t/2)\cos(4\pi(x+0.5))\cos(4\pi(y+0.5))$$