Taxiing and Wing Deploy by SPH（Smoothed particle hydrodynamics）

7, Feb., 2026

粒子法でトビウオの Taxiing と翅の展開を流体計算してみる．なお，OpenFOAM は VOF（Volume of Fluid）で計算している．後日要比較．魚は静止剛体として定義するが，粒子化されている．位置と速度を無理やり運動学で更新．衝突計算が楽．サンプルプログラム自体は，複数の剛体を動力学計算する solver も持っているので FSI もできが，いくつか問題があり後日検討．
気液二相流ではない．水上に空気はない．

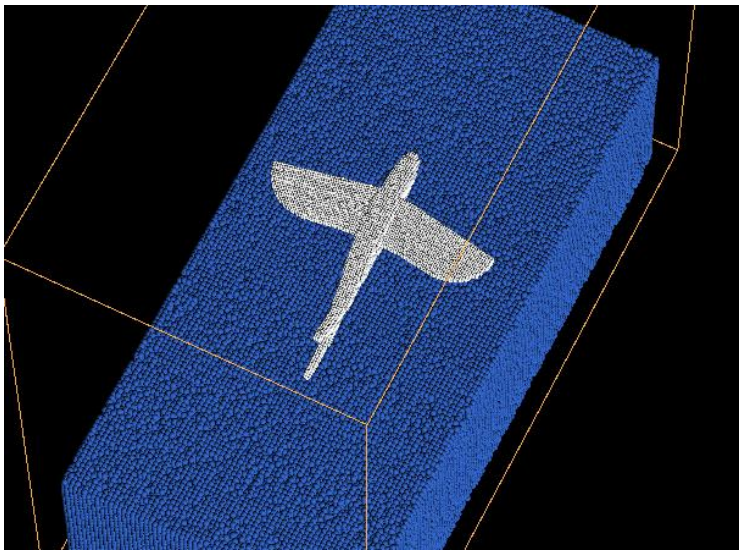
ここでは，剛体粒子の径と流体粒子の径を独立に定義できるようにした．本来剛体周りの流体粒子径のみアダプティブに変更したいがハードルが高いので後日．VOF などの格子法は物体周りの格子の細分化は簡単だが，粒子法は単純ではない．

Taichi 公式：
https://docs.taichi-lang.org

Taichi user interface 関係：
https://docs.taichi-lang.org/api/taichi/ui/

Taichi API reference：
https://docs.taichi-lang.org/api/

SPH の元ネタはここ：
https://github.com/erizmr/SPH_Taichi?utm_source=chatgpt.com
魚の全長は 20cm，翼幅 20cm，taxiing 周波数 10Hz，翼のラグ角 80deg
魚の粒子径は 2mm
流体の粒子径は 4mm



(1) 粒子法の計算スキーム私見＆覚書

ここでは，DFSPH(Divergence-Free SPH)を使っている．非圧縮性を安定に取り扱える．物性値が経験則でスケーリングされていて極めて直感的でない．粒子径を変更すると，それに伴い様々な物性値を調整しなければならない．なお，塚越先生の MPS(Moving particle semi-implicit method)は無い．

(1-1)　正しい物理量

．粒子半径 r[m]：particle_radius，tutorial のデフォルトは 0.01m．直径 d は particle_diameter.

．密度 ρ [kg/m^3]: density や rho など．デフォルトは 1000

この後のスケーリング物性値は，この値に適合するように決められている．

(1-2)　スケーリングされている物理量

．粘性 μ [Pa s]：水は 0.001 であるがデフォルトは 0.01．1cm の粒子半径に対して 10 倍になっている．
chatGPT 曰く，数値安定化にしか働かない．．．．混相流にしても同じ値が設定される．
密度だけ違う単相 SPH というらしい．

．表面張力 σ [N/m]：水は 0.0072 であるが，デフォルトは 0.01．chatGPT 曰く
σ /h のようなスケーリングになると言っている．とすると，カーネル距離 h が半分になったら表面張力がその分強くなるので，σ も半分にすればいいのか．．．．これも混相流にしても同じ値．

これは：

- カーネル半径 h
- 粒子径 dx
- 密度スケール
- dt

を全部含んだ
👉**「この解像度で“それっぽく見える”値」**

実務的には：

- 0.001 → ほぼ影響なし
- 0.01 → 水っぽいまとまり
- 0.05 → ゼリー / 金属液滴
- 0.1 以上 → 粒子が固まる

まとめ（重要）

- SPH の表面張力・粘性は
  物理定数ではなく、数値モデル係数
- 粒子径・h・dt に強く依存
- Taichi tutorial の混相流は
  「密度だけ違う単相 SPH」
- 本格混相流は別物（でもあなたなら行ける）

もし次に進むなら、
👉**「相間表面張力をどう定義するか」**
👉**「DFSPH で相ごとに粘性を分けると何が壊れるか」**

この辺、かなり面白いですよ。続けます？

(1-3) 粒子法用設定値

．カーネル半径 h[m]：supporrt_radius，対象粒子が周りの粒子から力学的影響を受ける範囲，デフォルトは，4r

．セル（グリッド）幅：grid_size，並列化のための計算対象範囲（組み合わせ爆発を防ぐ奴），デフォルトは=h．流体と剛体で異なる場合には大きい方に設定した．

．有効体積[m^3]：m_V=0.8d^3，球（=π/6*d^3）より大きく，立方体（=d^3）より小さい．0.8 はカーネル補正係数．「SPH 粒子 1 個が占めていると“みなす”体積」

．Solver flow

インデントが下がると子プロセスという意味．

solver = ps.build_solver()　　# particle_system　粒子を生成

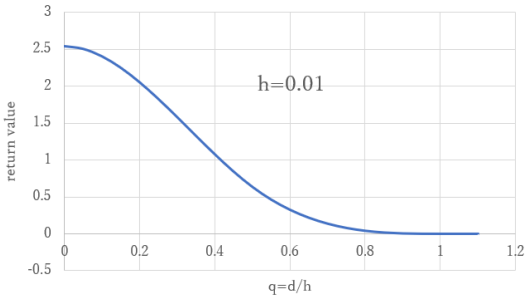
solver.step()　　# DFSPH　ソルバ

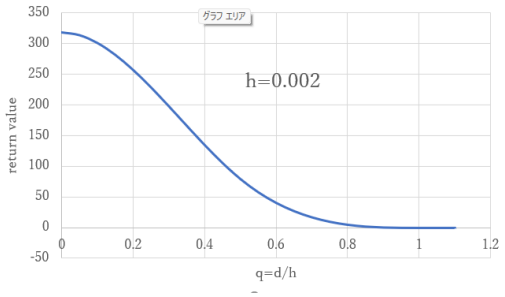
compute_densities()　　# 密度計算

cubic_kernel(0.0, h)　　# 3 次スプラインカーネル＝重み関数

$k = \frac{8}{\pi h^3}$, $q = \frac{d}{h}$, if $q < 1$, then $\begin{cases} k(6q^3 - 6q^2 + 1) & if \le q < 0.5 \\ 2k(1-q)^3 & else \end{cases}$

カーネル範囲内の粒子の計算．d は粒子間距離．横軸 q はカーネル距離比率．縦軸は重み．グラフにするとこんな感じ．離れるほど影響がなくなる．粒子径が小さいと，近づいたときに指数的に値が大きくなる．流体の物性値がスケーリングされるのはおそらくこれに依存している．径を小さくすとこの値が大きくなりすぎるため，柔らかくしないと（stiffness を小さくしないと）発散する（ただし，DFSPH は stiffness を必要としない）．1cm の粒子が 50%（=0.5cm）に近づくと重みは 0.7 ぐらい．0.2cm の粒子が 50%（=0.1cm）に近づくと重みは 70 ぐらいになる．1/5 にすると 100 倍→dt を相当小さくしなければならない．





compute_DFSPH_factor() # 勾配計算し，剛性を計算している．

divergence_solve()　　# 発散をとる計算

compute_non_pressure_forces()　　#表面張力，粘性，流体―剛体運動量交換

boundary_viscosity = 0.0 となっているので剛体の粘性は計算してない

predict_velocity()# 流速計算 v+＝a*dt

pressure_solve()　　# 近傍で圧力計算

compute_density_adv()　　#　＝V △v･a　　a はカーネルの導関数？

multiply_time_step(self.ps.dfsph_factor, inv_dt2)　　# 時間積分

advect() # 時間積分 p+＝v*dt

時間積分に advect を使っているが，移流項計算（advection）じゃないのね....

(1) CAD モデル

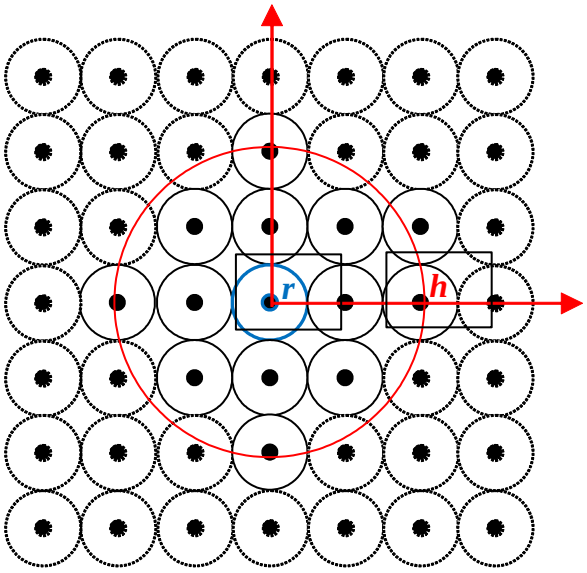
魚の剛体モデルは，CAD で作成して obj ファイルにする．単位は m

どうやら，obj フォーマットは，CAD などの inventor 系と CG での Maya 系があるようだ．vt が前者は 3 成分で後者は 2 成分. この元ネタはは後者を使っている. 以下をインストールしていれば回避できる．

> pip install networkx

翼厚は分厚くしてある．粒子半径 r を 1mm に設定するので，二個分の 4mm．検証していないが，粒子法の一般論として粒子が 2 層以上ないと正しく衝突計算ができない．下図の赤い領域に粒子が入ると，反力が計算される？　初期値においてこの範囲内に剛体や壁があると一気に計算が不安定になる．近傍のセルも h と同じ値．後日確認が必要．

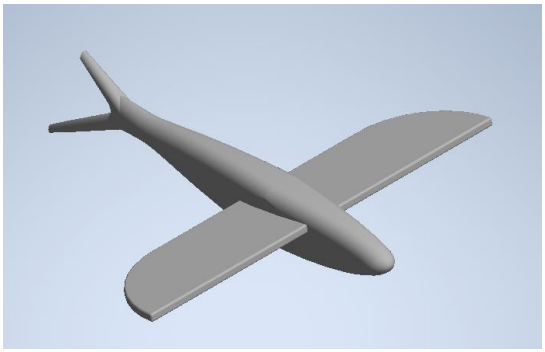
なお，最終的には，物体境界近傍の粒子径を動的に変更できないと正確に計算できない．



粒子半径 r，カーネル距離 h，セルサイズの関係

obj ファイル wiki

https://ja.wikipedia.org/wiki/Wavefront_.obj%E3%83%95%E3%82%A1%E3%82%A4%E3%83%AB



(2) 実行ファイル

data/models/fishbody.obj　　# 魚の CAD ファイル，obj 形式
data/scenes/fish.json　　# simulation の設定ファイル
config_builder.py　　# 粒子構築
DFSPH.py　　# DFSPH solver
particle_system.py　　# ps 変数構築．GPU を CPU 用にした．時代に逆行.
run_simulation.py　　# 実行ファイル
scan_single_buffer.py
sph_base.py
VTU.py　　# paraview 用 VTU 形式ファイル作成クラス
motion.py　　# モデルの運動を記述するクラス

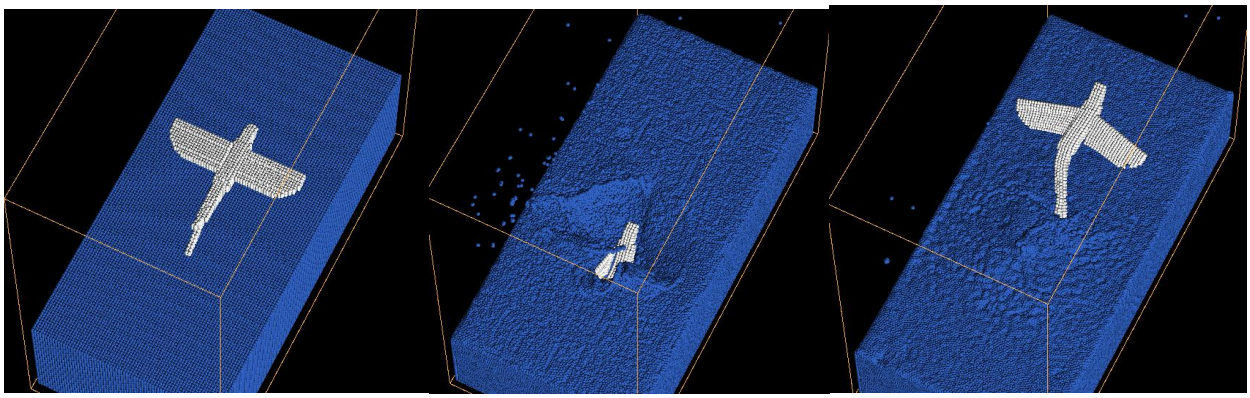
(1-1)　fish.json

json フォーマットのファイル．xml 形式
以下パラメータは，水っぽく見えるように調整しなければならない．今は適当．上記スケーリング物性値.

```
{
	"Configuration":
	{
	"domainStart": [0.0, 0.0, 0.0],	# 計算空間の設定．なぜか負の値はダメなようだ.
	"domainEnd": [0.6, 0.3, 0.6],
		"particleRadiusFLuid": 0.002,		# 流体粒子半径［m］
		"particleRadiusRigidRigid": 0.001,		# 剛体粒子半径
		"numberOfStepsPerRenderUpdate": 20,		# substep
		"density0": 1000,		# [kg/m^3]
		"surface_tension": 0.002,	# スケールされた表面張力．0.0072
		"viscosity": 0.002,	# スケールされた粘性係数．0.001
		"gravitation": [0.0, 0.0, -9.81],	# 重力
		"timeStepSize": 0.00005,		# dt [s]
		"exponent": 7,
		"boundaryHandlingMethod": 0,
		"exportFrame": true,	# 動画作成フラグ，window 表示しないときには false にする．最終的に，VTU ファイルを使って paraview で可視化できる.
		"exportPly": false,
		"exportObj": false,
		“simulationTime”: 0.8	# simulation 時間　[s]
	},
	"RigidBodies": [		# 剛体の魚
		{
	    "objectId": 1,		# 一匹目
			"geometryFile": "./data/models/fishbodyWithWing.obj", # CAD ファイルのパス
			"translation": [0.3, 0.15, 0.24],	# 粒子生成時の魚位置．流体粒子のない場所に設定する．初期魚座標原点
			"rotationAxis": [1.0, 0, 0],	# CAD モデル回転軸
			"rotationAngle": 0,	# CAD モデル回転角度
			"scale": [1, 1, 1],	# スケール
			"velocity": [0.0, 0.0, 0.0],	# 初速度［m/s］
			"density": 1000.0,	# 密度［kg/m^3］
			"color": [255, 255, 255],	# 色 RGB
			"isDynamic": false,	# 動力学計算の有無．ここでは運動学で計算するので false
			"fishLength": 0.2,	# 魚体長
			"fishWidth": 0.02,	# 魚ボディ幅
```

```
                "LEPos": 0.06,                  # 魚座標系からみた Leadingedge の x 位置.
                "wingspan": 0.2,                # 魚翼幅
                "wingchord": 0.04,              # 魚翼弦
                "waveLength": 0.2,              # carangiform 波長
                "freq": 10.0,                   # carangiform 周波数
                "amplitude": 0.02,              # carangiform 振幅
                "relax": 0.2,                   # 緩和係数 [s]
                "startPosition": [0.4, 0.15, 0.08],     # 魚泳ぎ始め位置
                "swimmingPosition": [0.227, 0.15, 0.22], # Taxiing 位置
                "pitchAngle": 30.0              } # ボディピッチ角
                ],
        "FluidBlocks": [                        # 流体粒子
                {
            "objectId": 0,
                "start": [0.008, 0.008, 0.008],         # 水の領域
                "end": [0.592, 0.292, 0.19],            # 0.19m が水面
                水の領域を境界よりも粒子径の 4 倍程度小さくしなければ発散するようだ.
                確かにプログラム中に 4*粒子径という記述がある. セル径依存するかも.
                start を[-1,-1,-1]のような負の領域から始められないのもよくわからず.
                "translation": [0.0, 0.0, 0.0],
                "scale": [1, 1, 1],
                "velocity": [0.0, 0.0, 0.0],
                "density": 1000.0,              # 密度も流体には二つある...
                "color": [50, 100, 200]
                }
        ]
}
```

魚の(1) "translation", (2) "startPosition", (3) "swimmingPosition"の位置関係は図の通り：
(1)は，流体と重ならないようにしなければならないので，一度水面上部に設定し，その後本来のシミュレーション開始位置の(2)まで沈め，そこから遊泳をスタートしている．これも気液二相流で計算するときには修正しなければならない（そもそも今は密度しか変えられないが....）.



(1-2) motion.py

剛体粒子を運動学で強制的に動かすルーチン．基本は単純な平行移動と回転の座標変換.

```
import taichi as ti
import math

@ti.data_oriented
class Motionclass:
```

```
    def __init__(self, fish):
        for i, rb in enumerate(fish):                    # json ファイルの変数を読み込む
            self.fishCenter = tuple( rb["translation"] )
            self.fishLength = float( rb["fishLength"] )
            self.fishWidth = float( rb["fishWidth"] )
            self.wingspan = float( rb["wingspan"] )
            self.wingchord = float( rb["wingchord"] )
            self.LEPos = float( rb["LEPos"] )                    # leading edge x pos = distance
from fish center to leadigedge
            self.pitchAngle = math.radians( float(rb["pitchAngle"]) )
            self.waveLength = float( rb["waveLength"] )
            self.fishVel = tuple( rb["velocity"] )
            self.fishAmp =   float( rb["amplitude"] )
            self.fishFreq =   float( rb["freq"] )
            self.relax =   float( rb["relax"] )          # relaxation time
            self.startPos = tuple( rb["startPosition"] )
            self.swimPos = tuple( rb["swimmingPosition"] )

    @ti.kernel    # *** One way FSI ***
    def carangiform( self, object_id: ti.template(), fx: ti.template(), fx0: ti.template(), fv: ti.template(),
n: ti.i32, t: ti.f32, dt: ti.f32 ):
        PI = ti.math.pi
        center = ti.Vector( self.fishCenter )
        amp = self.fishAmp
        length = self.fishLength
        waveL = self.waveLength
        wingL = self.wingspan/2.0 - self.fishWidth/2.0
        freq = self.fishFreq
        cycle = 1.0/freq
        startPos = ti.Vector( self.startPos )
        swimPos = ti.Vector( self.swimPos )
        relax = self.relax
        width = self.fishWidth
        pitch = self.pitchAngle * ti.min( 1.0, t/relax )
                    Ry = \begin{bmatrix} cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}
        Ry = ti.Matrix([ [ ti.cos(pitch), 0.0, ti.sin(pitch)],
                         [ 0.0,                1.0, 0.0              ],
                         [-ti.sin(pitch), 0.0, ti.cos(pitch)] ])

        for i in range(n):
            if object_id[i] == 1:    # if fish=rigid body: 0 means fluid, Caution: i is renumbered.
                bp0   = fx0[i] - center       # fish coordinate system
                bp1   = fx0[i] - center       # fish coordinate system
                disY = ti.abs( bp0.y ) - width / 2.0                    #  翼の付け根 y 位置
                # ********    current position ***********
                Y = \frac{A\left(x+\frac{L}{2}\right)}{L} * \sin(2\pi(\frac{x+\frac{L}{2}}{\lambda} - ft))
                dy = amp / length * ( bp0.x + length/2.0 ) * ti.math.sin( 2.0 * PI * ( ( bp0.x +
length/2.0 ) / waveL - freq * t ) )
                if 0.0 < disY:
                    dy *= ti.math.pow( 1.0 - disY/wingL, 2.0 )
                bp0.y += ( dy * ti.min( 1.0, t/relax ) )
                # ************* future position *************
                dy = amp / length * ( bp0.x + length/2.0 ) * ti.math.sin( 2.0 * PI * ( ( bp0.x +
length/2.0 ) / waveL - freq * (t+dt) ) )
                if 0.0 < disY:
                    dy *= ti.math.pow( 1.0 - disY/wingL, 2.0 )
                bp1.y += ( dy * ti.min( 1.0, t/relax ) )          # not accurate
```

```
                    # wing deployment
                    if ( 0.0 < disY ) and ( (-0.1*self.wingchord) < (bp0.x + self.LEPos) ) and ( (bp0.x +
self.LEPos) < (1.1*self.wingchord) ):
                        wp0 = bp0                                           # fish coordinate system
                        wp1 = bp1                                           # fish coordinate system
                        if 0.0 < wp0.y:  # right wing coordinate system
                            wp0 += ti.Vector([self.LEPos, -width/2.0, 0.0])          # shift for right wing
                            wp1 += ti.Vector([self.LEPos, -width/2.0, 0.0])          # shift for right wing
                        else:                       # left wing coordinate system
                            wp0 += ti.Vector([self.LEPos,    width/2.0, 0.0])        # shift for left wing
                            wp1 += ti.Vector([self.LEPos,    width/2.0, 0.0])        # shift for left wing
                        lag0 = 0.8 * ti.atan2( wp0.y, wp0.x )                   # ti.atan2(y, x), ti.atan2(0, 0)-
>0.0
                        if t < relax:
                            lag0 *= ( ( 1.0 - ti.math.cos( PI * t/relax ) ) / 2.0 )
                        elif t < (relax+2*cycle):
                            lag0 *= 1.0
                        elif t < (relax+3*cycle):
                            lag0 *= ( ( 1.0 + ti.math.cos( PI * (t-relax-2*cycle)/cycle ) ) / 2.0 )
                        else:
                            lag0 = 0.0
                        lag1 = 0.8 * ti.atan2( wp1.y, wp1.x )                       #      ti.atan2(y,      x),
ti.atan2(0, 0)->0.0
                        if (t+dt) < relax:
                            lag1 *= ( ( 1.0 - ti.math.cos( PI * (t+dt)/relax ) ) / 2.0 )
                        elif (t+dt) < (relax+2*cycle):
                            lag1 *= 1.0
                        elif (t+dt) < (relax+3*cycle):
                            lag1 *= ( ( 1.0 + ti.math.cos( PI * (t+dt-relax-2*cycle)/cycle ) ) / 2.0 )
                        else:
                            lag1 = 0.0
                        Rz0 = ti.Matrix([ [ti.cos(-lag0), -ti.sin(-lag0), 0.0],
                                          [ti.sin(-lag0),    ti.cos(-lag0), 0.0],
                                          [0.0,                0.0,              1.0] ])
                        Rz1 = ti.Matrix([ [ti.cos(-lag1), -ti.sin(-lag1), 0.0],
                                          [ti.sin(-lag1),    ti.cos(-lag1), 0.0],
                                          [0.0,                0.0,              1.0] ])
                        bp0 = Rz0 @ wp0
                        bp1 = Rz1 @ wp1
                        if 0.0 < wp0.y:
                            bp0 -= ti.Vector([self.LEPos, -width/2.0, 0.0])
                            bp1 -= ti.Vector([self.LEPos, -width/2.0, 0.0])
                        else:
                            bp0 -= ti.Vector([self.LEPos,    width/2.0, 0.0])
                            bp1 -= ti.Vector([self.LEPos,    width/2.0, 0.0])

                    # posture
                    fx[i] = Ry @ bp0 + center
                    Fx    = Ry @ bp1 + center

                    # Locomotion
                    if t < relax:
                      fx[i] += ( startPos - center ) * ti.min( 1.0, (1.0 - ti.cos(PI*t/relax))/2.0 )
                      Fx    += ( startPos - center ) * ti.min( 1.0, (1.0 - ti.cos(PI*t/relax))/2.0 )      #
not accurate
                    elif t < (relax+2*cycle):
                      fx[i] += ( ( startPos - center ) + ( swimPos - startPos ) * ( 1.0 - ti.cos(PI*(t-
relax)/(2.0*cycle)))/2.0 )
                      Fx    += ( ( startPos - center ) + ( swimPos - startPos ) * ( 1.0 - ti.cos(PI*(t-
```

```
relax)/(2.0*cycle)))/2.0 )  # not accurate
                    else:
                      fx[i] += ( swimPos - center )
                      Fx      += ( swimPos - center )    # not accurate



                    # ***** velocity ******** accuracy is very low because of "float"
                    fv[i] = ( Fx - fx[i] ) / dt
```

(1-3)　particle_system.py

json ファイルの設定から粒子を構築するプログラム．CAD モデルも粒子にしている．
マテリアルを流体(=1)と固体(=0)に分け，流体も固体も id で異なる物性を管理している（ここでは，流体が 0，剛体が 1)．混相流計算ができることになっているが，実際は粘性と表面張力は分けられていない．変えられるのは密度だけ．つまり，このままでは混相流計算はできない．また，マルチボディができるが，このプログラムでは剛体間の粒子径などは統一にした．

(1-4)　DFSPH.py

ソルバ．流体粒子径と剛体粒子径を独立して扱えるようにした．1, 2, 3 次元に対応している部分は，3 次元のみにした．カーネル距離は大きい方を採用するようにした．剛体の方が小さいので，剛体にとっては遠いところまで計算しなければならない無駄計算になる．支配方程式に関しては後日．

(1-5)　sph_base.py

粒子法の基礎計算スキーム．汎用性が高い部分は簡潔にして削除した．

(1-6)　Paraview 可視化用 VTU ファイル生成

中身は xml フォーマットになっている．
物性として，4 つ：
density
object_id
pressure
velocity
を出力させている．paraview 用に CAD モデルも吐き出した方がいいのかも．粒子が粗すぎて可視化画像がしょぼい．
また，ファイルサイズが大きすぎるので
write_vtuASCII
write_vtuBINARY
を用意した．バイナリバージョンは 1/5 ぐらいになる．本来 xml はアスキーなので，部分的にバイナリというトリッキーなことをしている．paraview では普通に読めた．また，剛体の密度は粒子すべてに初期値を出力するようにした．paraview での可視化をきれいにするため．実際は剛体も変動している．近傍粒子で計算しているため．その後，剛体の可視化は，object_id の方が便利なことが判明した．

```
<?xml version="1.0"?>
```

```
<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">
  <UnstructuredGrid>
    <Piece NumberOfPoints="813926" NumberOfCells="813926">     # 粒子数
      <Points>                                                 # 各粒子の三次元座標
        <DataArray type="Float32" NumberOfComponents="3" format="ascii">
          0.008227914571762085 0.008227914571762085 0.00800000037997961
…..
          0.5877098441123962 0.5877098441123962 0.15520766377449036
        </DataArray>
      </Points>
      <Cells>                                          # セル＝カーネル半径内の粒子番号か？
        <DataArray type="Int32" Name="connectivity" format="ascii">
          0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47
….
813912 813913 813914 813915 813916 813917 813918 813919 813920 813921 813922 813923 813924
813925
        </DataArray>
        <DataArray type="Int32" Name="offsets" format="ascii">       # これなんだろう？
          1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74
……
813913 813914 813915 813916 813917 813918 813919 813920 813921 813922 813923 813924 813925
813926
        </DataArray>
        <DataArray type="UInt8" Name="types" format="ascii">       # これなんだろう？　剛体 or
流体？
          1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
…….
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
        </DataArray>
      </Cells>
      <PointData Scalars="scalars">
        <DataArray type="Float32" Name="density" format="ascii">  # それぞれの粒子の密度？？
          525.5376586914062
          619.8544311523438
          619.4436645507812
……
618.0166015625
          524.6402587890625
        </DataArray>
        <DataArray type="Float32" Name="pressure" format="ascii">          # 粒子の圧力
          0.0
          0.0
          0.0
….
0.0
          0.0
        </DataArray>
        <DataArray type="Float32" NumberOfComponents="3" Name="velocity" format="ascii">#流
速
          0.02093004621565342 0.020930051803588867 0.0021960726007819176
          0.00899195671081543 0.00899195484817028 -0.024869199842214584
….
          -0.025897055864334106 -0.025897052139043808 -0.1137751042842865
          -0.03211219608783722 -0.03211219608783722 -0.12822650372982025
        </DataArray>
        <DataArray type="Int32" Name="object_id" format="ascii">
```

```
            0
            0
…
            0
          </DataArray>
        </PointData>
      </Piece>
    </UnstructuredGrid>
</VTKFile>
```

(1-7)　run_simulationkikut.py

これが実行ファイル．ちょっと煩雑なので要点だけ．

```
import os                                   # linux コマンドを使う．ここでは mkdir
import argparse
import taichi as ti                         # taichi のメイン関数
import numpy as np                          #  数学関数他
from config_builder import SimConfig        # from A import B は，A.py ファイルから B 関数を使うという意味
from particle_systemkikut import ParticleSystem
from moviepy import ImageSequenceClip
import glob
from VTU import VTUclass                    # VTU.py ファイルから，VTU クラスをインポートする
from motion import Motionclass              # taxiing motion を記述したクラス

# ti.init(arch=ti.gpu, device_memory_fraction=0.5)

ti.init(arch=ti.cpu)                        # GPU がないので CPU にする

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='SPH Taichi')
    parser.add_argument('--scene_file', default='', help='scene file') #  プロンプトに” --scene_file”と書いた後のファイルを読み込む
    args = parser.parse_args()
    scene_path = args.scene_file
    config = SimConfig(scene_file_path=scene_path)          # json ファイル読み込み
    vtu = VTUclass()                                        # VTU ファイル用クラス読み込み
    scene_name = scene_path.split("/")[-1].split(".")[0]    # json ファイル名切り出し
    substeps = config.get_cfg("numberOfStepsPerRenderUpdate") # substep
    output_frames = config.get_cfg("exportFrame")           #  画像書き出し  or not
    dt = config.get_cfg("timeStepSize")
    simulationTime = config.get_cfg("simulationTime")       #  シミュレーション時間
    series_prefix = "{}_output/particle_object_{}.ply".format(scene_name, "{}")
    if output_frames:
        os.makedirs("image", exist_ok=True)                 # image ディレクトリ作成
    os.makedirs("VTU", exist_ok=True)                       # VTU ディレクトリ作成

    fish = config.get_rigid_bodies()        #  魚変数．json からパラメータを読み込む
    motion = Motionclass( fish )            # motion クラス読み込み

    ps = ParticleSystem(config, GGUI=True)          # particle variables
    solver = ps.build_solver()                      # 4 means DFSPH
    solver.initialize()

    # Invisible objects       これ，なんだろう？
    invisible_objects = config.get_cfg("invisibleObjects")
    if not invisible_objects:
        invisible_objects = []
```

```
    if output_frames:       # window 出力．動画がいらなければいらない．
        window = ti.ui.Window('SPH', (800, 600), show_window = True, vsync=False)
        scene = ti.ui.Scene()
        camera = ti.ui.Camera()
        camera.position(2.0, 1.0, 4.0)
        camera.up(0.0, 0.0, 1.0)
        camera.lookat( 0.0, 0.0, -0.5)
        camera.fov(8)
        scene.set_camera(camera)

        canvas = window.get_canvas()
        movement_speed = 0.02
        background_color = (0, 0, 0)  # 0xFFFFFF
        particle_color = (1, 1, 1)


    # Draw the lines for domain  枠の描画だが，邪魔かも
    x_max, y_max, z_max = config.get_cfg("domainEnd")
    box_anchors = ti.Vector.field(3, dtype=ti.f32, shape = 8)
    box_anchors[0] = ti.Vector([0.0, 0.0, 0.0])
    box_anchors[1] = ti.Vector([0.0, y_max, 0.0])
    box_anchors[2] = ti.Vector([x_max, 0.0, 0.0])
    box_anchors[3] = ti.Vector([x_max, y_max, 0.0])

    box_anchors[4] = ti.Vector([0.0, 0.0, z_max])
    box_anchors[5] = ti.Vector([0.0, y_max, z_max])
    box_anchors[6] = ti.Vector([x_max, 0.0, z_max])
    box_anchors[7] = ti.Vector([x_max, y_max, z_max])

    box_lines_indices = ti.field(dtype=ti.i32, shape=(2 * 12))  # dtype=ti.i32 means "int"

    for i, val in enumerate([0, 1, 0, 2, 1, 3, 2, 3, 4, 5, 4, 6, 5, 7, 6, 7, 0, 4, 1, 5, 2, 6, 3, 7]):
        box_lines_indices[i] = int(val)

    cnt = 0
    t = 0                       #  時間

    while t < simulationTime:
        print(f"time={t:.3f}")      #  現在時間を 3 桁で．
        for i in range(substeps):    # solver
            motion.carangiform(  ps.object_id,  ps.x,  ps.x_0,  ps.v,  ps.particle_max_num,  t,  dt  ) #
carangiform で剛体粒子を移動．id=1 が剛体．リナンバリングされているので注意．x が位置．x0 が初
期値，v が速度，num が粒子数，t が現在時間，dt が時間刻み
            solver.step() #  ソルバ
            t += dt;

        # visualize for window
        if output_frames:           #  画像出力
            ps.copy_to_vis_buffer(invisible_objects=invisible_objects) #  描画のためのコピー
            scene.set_camera(camera)
            scene.point_light((2.0, 2.0, 2.0), color=(1.0, 1.0, 1.0))
            scene.particles(ps.x_vis_bufferFluid,                          radius=ps.particle_radiusFluid,
per_vertex_color=ps.color_vis_bufferFluid) #  流体粒子の登録
            scene.particles(ps.x_vis_bufferRigid,                          radius=ps.particle_radiusRigid,
per_vertex_color=ps.color_vis_bufferRigid)  #  剛体粒子の登録
            scene.lines(box_anchors, indices=box_lines_indices, color = (0.99, 0.68, 0.28), width = 1.0)
            canvas.scene(scene)
            window.save_image(f"./image/frame{cnt:06}.png")
            window.show()
```

```
        # for paraview visualization       VTU ファイル掃き出し
        vtu.write_vtuBINARY(f"./VTU/SPH{cnt:06}.vtu", pos)

        cnt += 1

    # collect frames and write movie
    if output_frames:                       #  動画作成
        files = sorted(glob.glob("./image/frame*.png"))
        if len(files) > 0:
            clip = ImageSequenceClip(files, fps=60)
            clip.write_videofile("out.mp4", codec="libx264", audio=False)
```

(3) 実行

taichi をアクティベイト

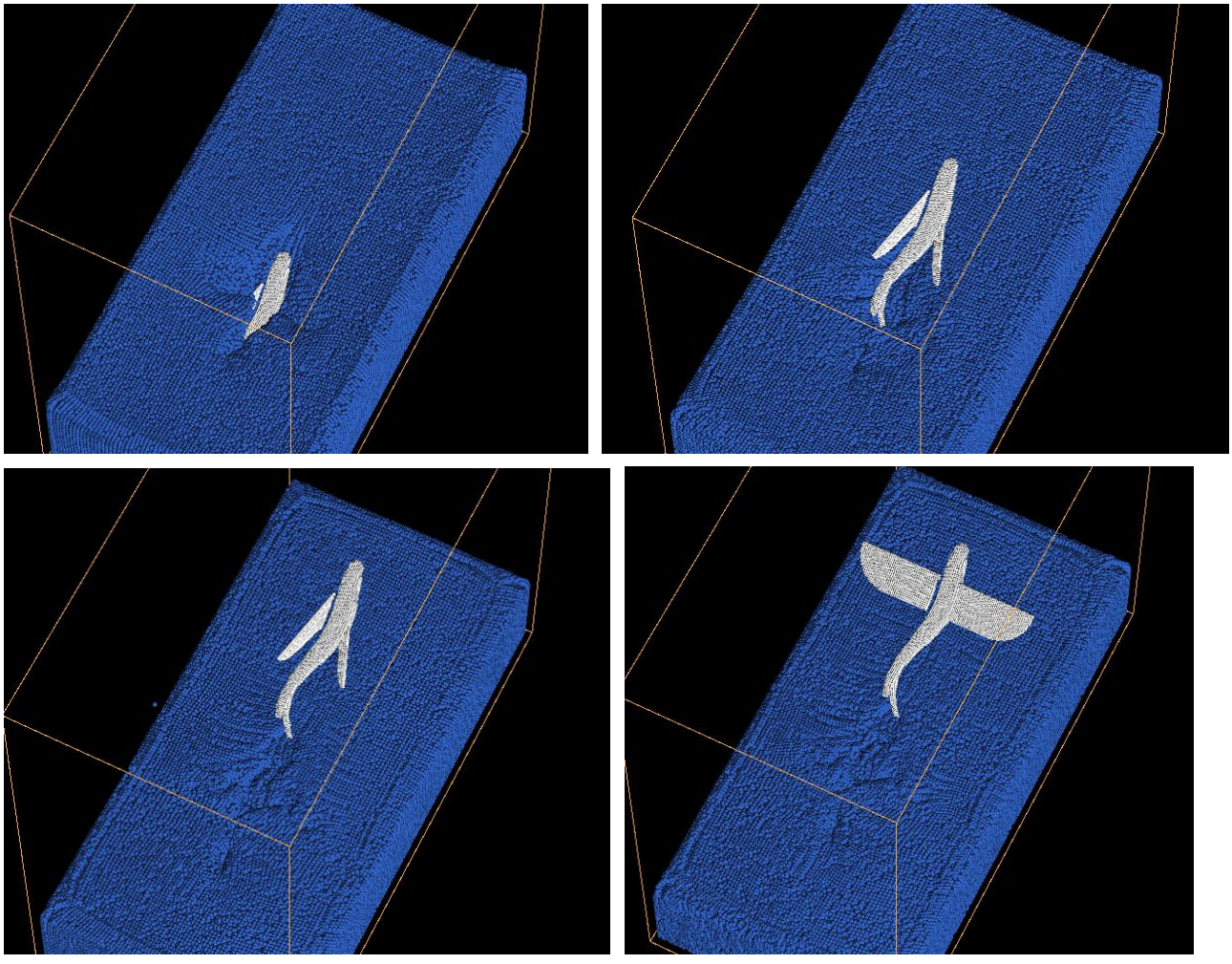
> source taichi_env/bin/active

してから,

> python `run_simulation.py --scene_file ./data/scenes/fish.json`

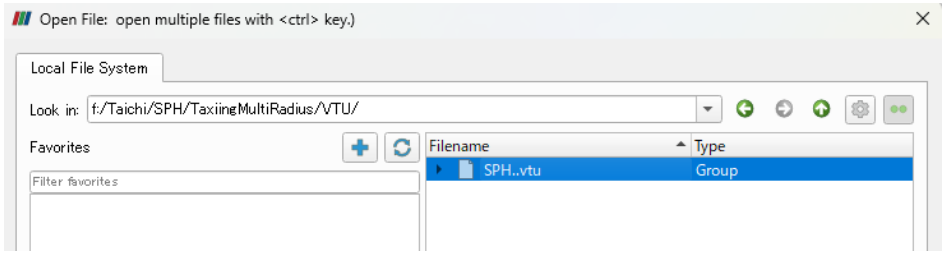
を実行する.

"exportFrame"を true にしていると，以下の画像が時系列で出力される．粒子径 2mm の画像．0.8 秒のシミュレーションに 3 時間ぐらいかかる．おそらく GPU なら 100 倍速い．そういうアルゴリズム．精度検証は必要．床の摩擦など未知．おそらく摩擦無しの slip と同等.

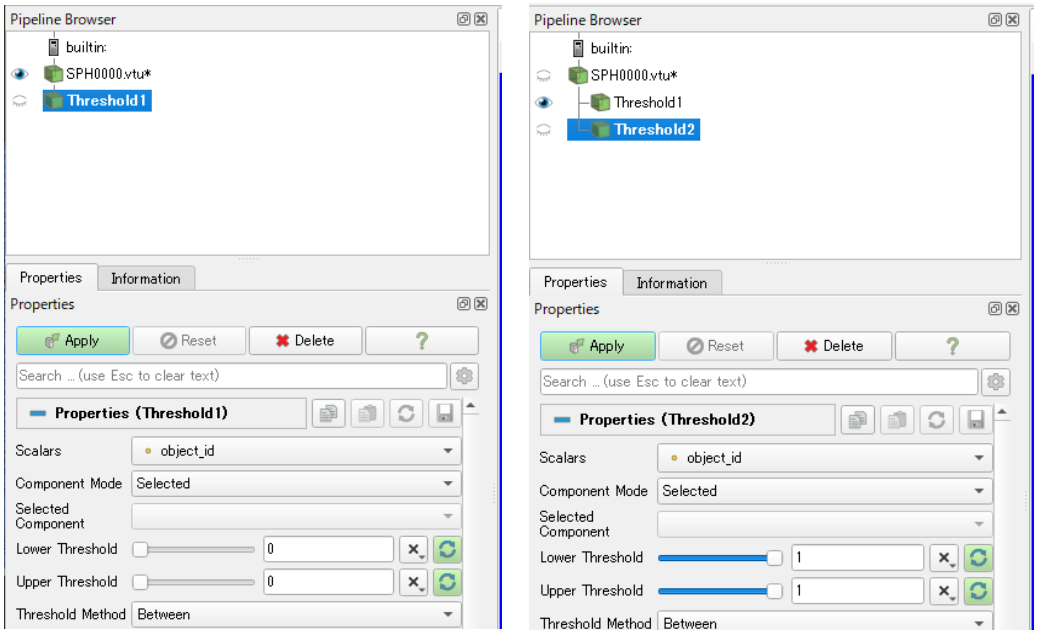


(4) Paraview での可視化

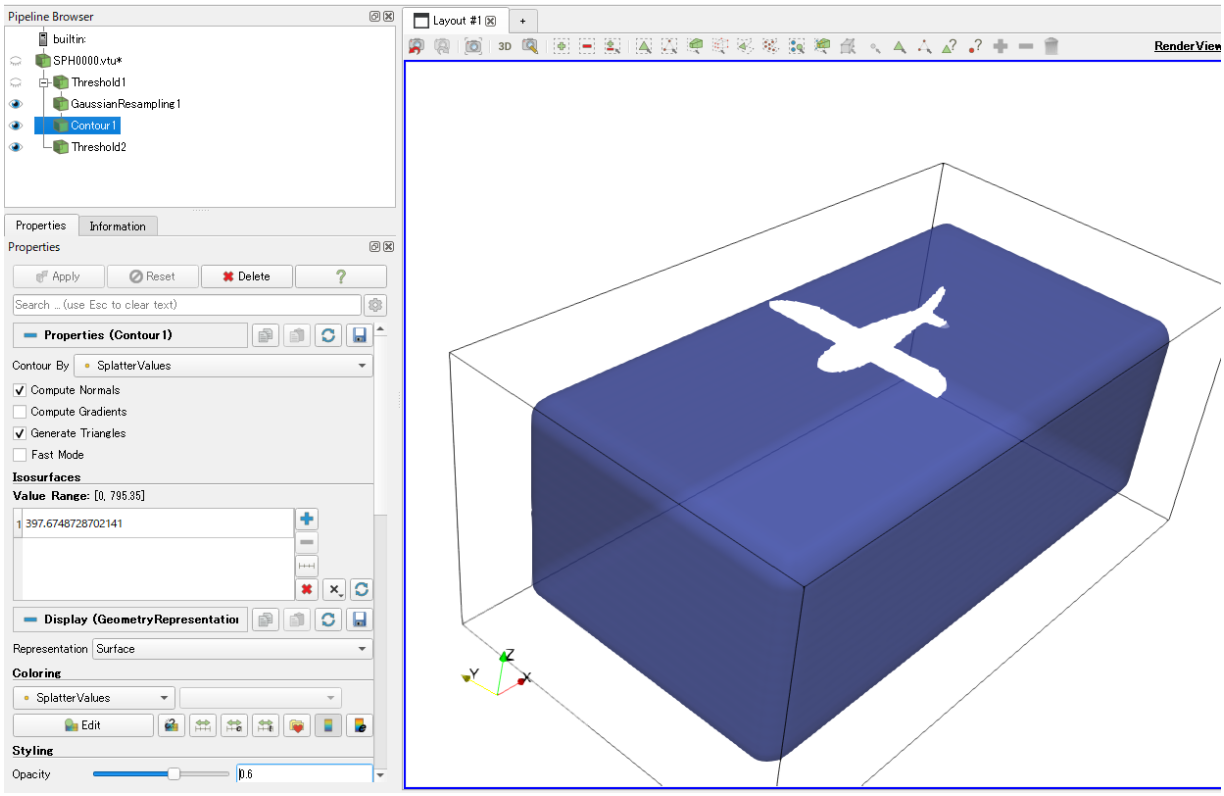
Paraview を立ち上げ，File -> open から折りたたまれてる vtu を一気に読み込んで Apply する．



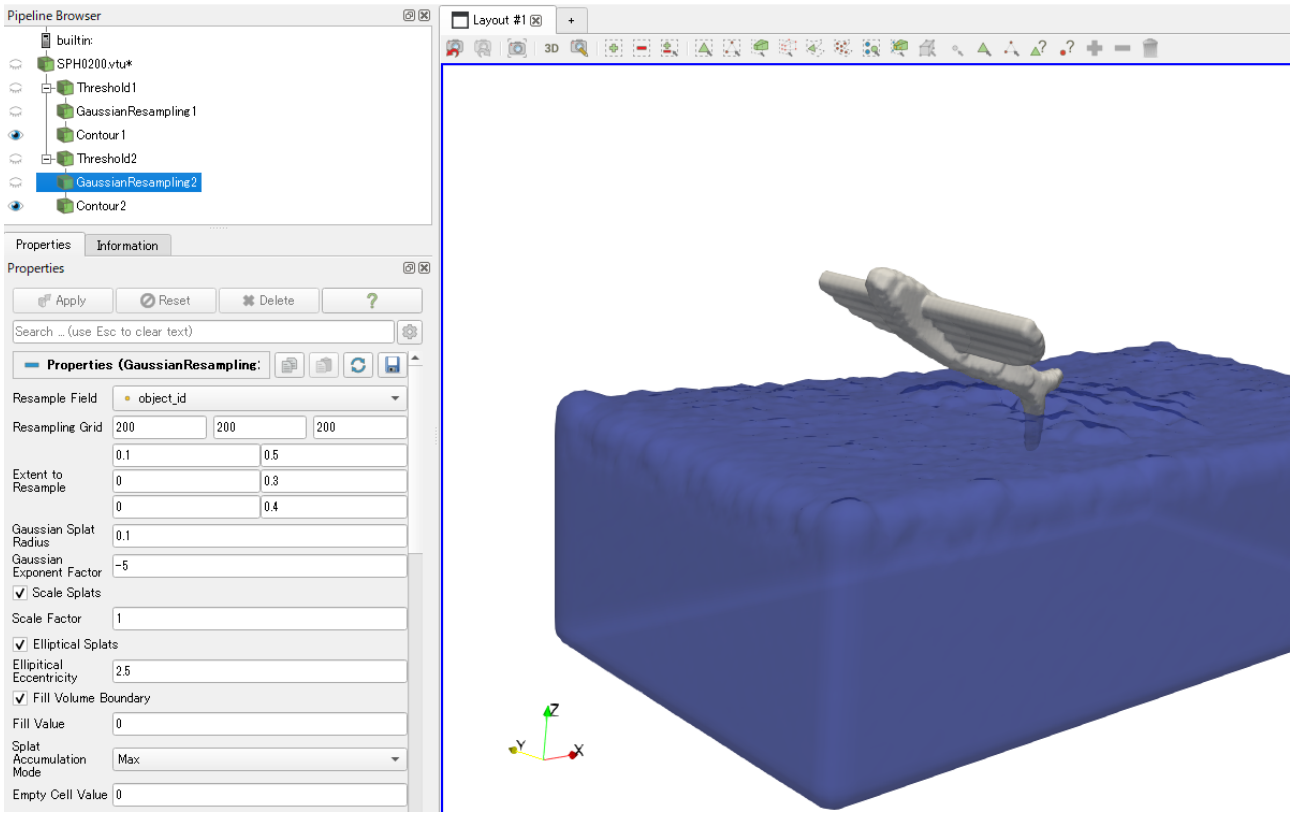
Filters->Alphabetical->Threshold で Acalar=object_id，Lower Threshold=0, Upper Threshold=0 で Apply する．0 は流体．vtu をクリックし，同様にして剛体も分けておく．剛体は 1

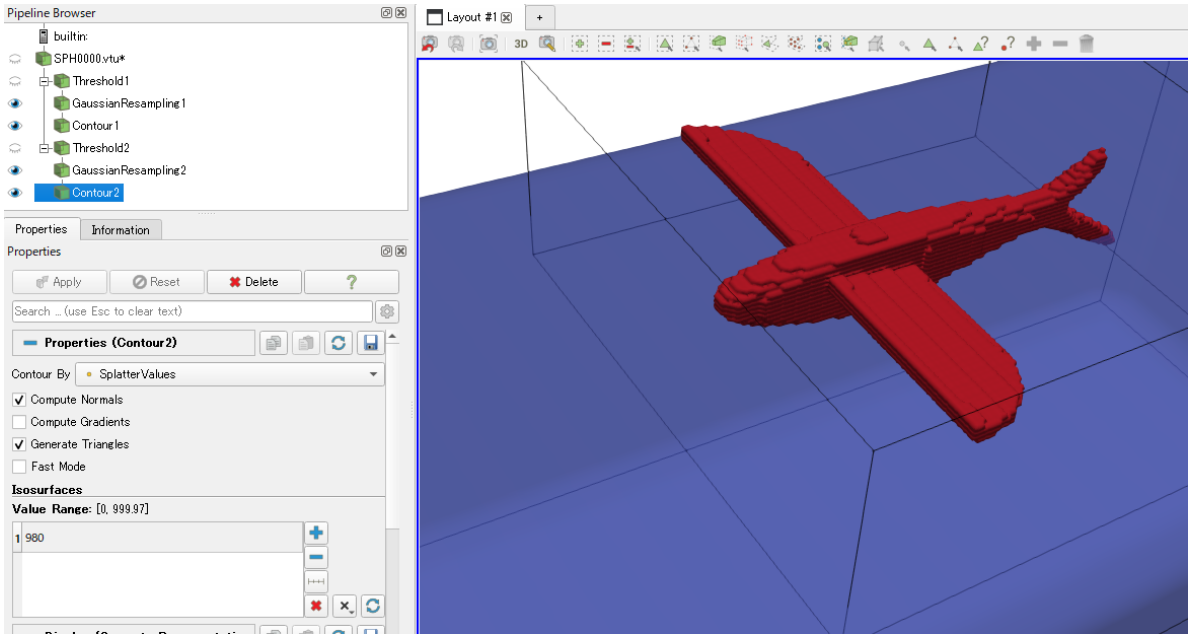


Threshold1（流体）を選択し，Filter->Alphabetical->Gaussian Resampling で空間解像度（Resampling Grid）を 200, 200, 200 にして Apply する．density をコンタ図にすると流体を可視化できる．

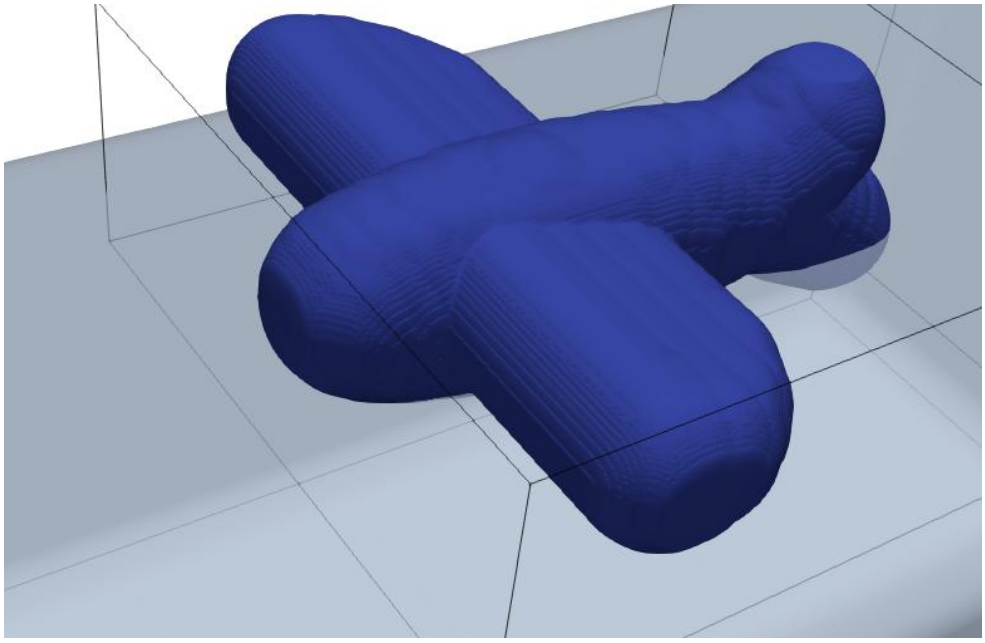


同様に，トビウオも表示する．Gaussian Resampling を 200, 200, 200 にして，object_id を Contour の値にして，0.95 にしてこんな感じ．そもそもの粒子が粗いのでこの辺が限界．翼厚が 2 粒子．また，Contour を density にして 980 は，粒子径あたりで描画している．プログラムの方で剛体粒子の密度は常に 1000 を吐き出している．なお，Resampling の範囲は，初期値に依存するのではみ出ないように Resampling Grid を大きくしておく．ここでは，x: 0.1 to 0.5, y: 0 to 0.3, z: 0 to 0.4.





コンタの値を小さくすると膨張する．おそらく粒子径の 4 倍程度まで．試しに 10（1000 の 1%）にするとこんな感じ．おそらく粒子が 4 倍ぐらいに膨らんだ状態で描画している．



時系列描画に関しては Dam break を参照.

Paraview での VTU
https://discourse.paraview.org/t/understanding-vtu-file-format/508

(5) その他資料
print(dis(ps))すると，ps には以下のメンバが定義されている．いくつかの変数は追加してある．

'GGUI',
'__class__',
'__delattr__',
'__dict__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__getstate__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'_add_particles',
'_copy_to_vis_buffer',
'_data_oriented',
'_write_grid_prefix_from_numpy',
'acceleration',
'acceleration_buffer',
'add_cube',
'add_particle',
'add_particles',
'build_solver',
'cfg',
'color',

'color_buffer',
'color_vis_buffer',
'compute_cube_particle_num',
'copy_to_numpy',
'copy_to_vis_buffer',
'counting_sort',
'density',
'density_adv',
'density_adv_buffer',
'density_buffer',
'dfsph_factor',
'dfsph_factor_buffer',
'dim',
'domain_end',
'domain_size',
'domain_start',
'domian_end',
'dump',
'flatten_grid_index',
'fluid_particle_num',
'for_all_neighbors',
'get_flatten_grid_index',
'grid_ids',
'grid_ids_buffer',
'grid_ids_new',
'grid_num',
'grid_particles_num',
'grid_particles_num_temp',
'grid_size',
'initialize_particle_system',
'is_dynamic',
'is_dynamic_buffer',
'is_dynamic_rigid_body',
'is_static_rigid_body',
'load_rigid_body',
'm',
'm_V',
'm_V0',
'm_V_buffer',
'm_buffer',

'material', # 0 or 1　で固体と流体(1?)を区別している
'material_buffer',
'material_fluid',
'material_solid',
'num_rigid_bodies',
'object_collection',
'object_id',
'object_id_buffer',
'object_id_rigid_body',
'padding',
'particle_diameter',
'particle_max_num',
'particle_num',
'particle_radius',
'pos_to_index',
'prefix_sum_executor',
'pressure',
'pressure_buffer',
'rigid_rest_cm',
'simulation_method',
'solid_particle_num',
'support_radius',
'update_grid_id',
'v', # 速度(vx, vy, vz)
'v_buffer',
'x', # 位置(x, y, z)
'x_0',
'x_0_buffer',
'x_buffer',
'x_vis_buffer']