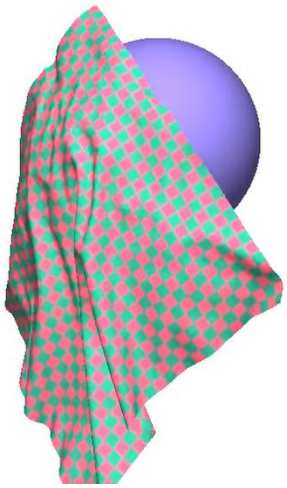
Cloth Falling

球に布を落とす tutorial のシミュレーション．球は剛体で動かない．布は，バネマスダンパ系の動力学モデル．インタプリタを高速化するパイソン特有のコードと，GPU 用のスキーム，動力学を単純化してそれっぽく見せるスキーム，いろいろ駆使されている．単位質量当たりの力学で解かれているため，質量の概念はない．

https://docs.taichi-lang.org/docs/cloth_simulation



(1) Source program

. clothFall.py

以下，単位は**「単位質量当たり」**．例えば，力:force は，[N/kg]になっている．

```
import taichi as ti             # ti という名前で使う定義
import os                       # linux のコマンドを使う宣言

from moviepy import ImageSequenceClip
import glob

ti.init(arch=ti.cpu)            # GPU じゃなくて CPU で使う

n = 128                         #  布の一辺の分割数．n*n の質点数になる
quad_size = 1.0 / n             #  布の粒子のサイズ［m］
dt = 4e-2 / n                   #  刻み時間［s］
substeps = int(1 / 60 // dt) #  刻み時間内のサブステップ“//”は割り算の商(Floor 除算)．余りは切り捨て

gravity = ti.Vector([0, -9.8, 0])      #  重力，y 方向
spring_Y = 3e4                  #  布の単位長さ×バネ定数［N/kg］: メッシュ長さで割ると単位質量当たりのバネ定数になる[N/m/kg]．単位質量当たりのバネの反力を，
                                F=k*Δx
                                で解くか，
                                F=K*ε：ε=Δx/L
                                で解くかの違い．
                                メッシュの長さに対応したそれぞれのバネ定数を計算するのが大変なのでこうしている．メッシュの長さが半分になると，バネ定数 k を倍にしなければならない．材力で考えると，F=ESε で，単位質量当たりにすると，F/m=ES/mε となって，k=ES/m と同じ．
```

```
dashpot_damping = 1e4              # 布の単位長さ＆単位質量当たりの粘性　[N/(m/s)/kg/m]．こちらはバネ定数と逆で，メッシュ長さが半分になると，この値も半分にしなければならない．
drag_damping = 1                   # 単位密度当たりの空気抵抗のダンパ［N/(m/s)/質量］

ball_radius = 0.3                  # 玉の半径　[m]
ball_center = ti.Vector.field(3, dtype=float, shape=(1, )) # データ構造定義．3 次元，浮動小数，shape=(1,) は「要素数が 1 の 1 次元配列」=結局，浮動小数の 3 次元ベクトルを 1 個だけ持つ 1 次元フィールドという意味
```

① `shape=(1,)`

- `shape` は配列の形状
- `(1,)` は **1次元・長さ1**

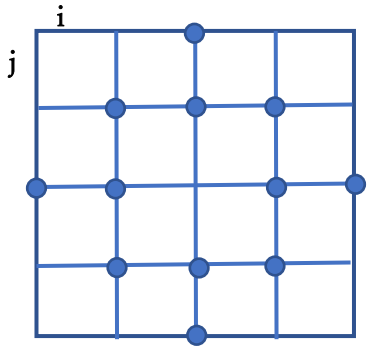
shape	意味
`(1,)`	要素が1個の1次元配列
`(3,)`	要素が3個の1次元配列
`(1, 1)`	1×1 の2次元配列
`(n, n)`	n×n の2次元配列

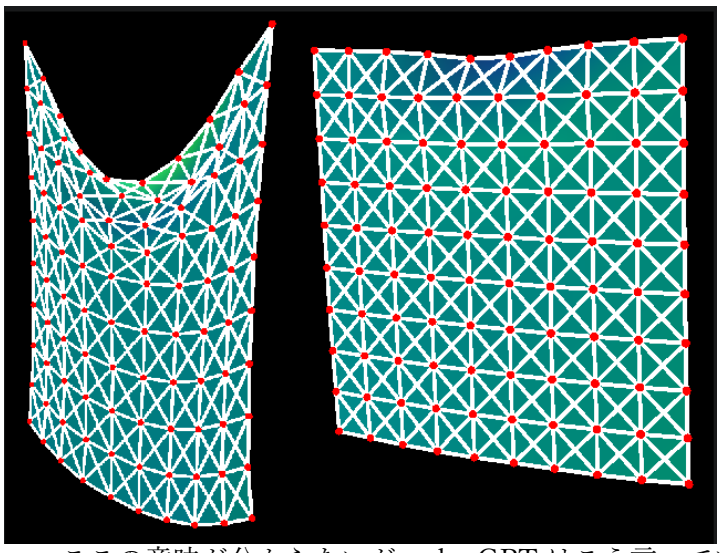
⚠️ `(1)` ではなく `(1,)`
→ **タプル**として認識させるためのカンマが必要です。

単純な一つの 3 次元ベクトルじゃなく，taichi の field として取り扱いたいという意味らしい．GPU の並列カーネルから直接アクセスするとかという話．

```
ball_center[0] = [0, 0, 0]         # 球の中心座標［m］

# 布の質点座標ベクトルの定義
x = ti.Vector.field(3, dtype=float, shape=(n, n))       # データ構造定義．3 次元，浮動小数，n*n 配列
v = ti.Vector.field(3, dtype=float, shape=(n, n))

num_triangles = (n - 1) * (n - 1) * 2          # 布を構成する三角形の数．四角は二個の三角形として定義される．
indices = ti.field(int, shape=num_triangles * 3) # 三角形の数×頂点数
vertices = ti.Vector.field(3, dtype=float, shape=n * n) # 布の頂点=質点の数
colors = ti.Vector.field(3, dtype=float, shape=n * n)   # 布の色：RGB の三成分を質点毎指定できる

bending_springs = False

@ti.kernel          # 以下の定義を taichi の kernel として定義する：インタプリタの python 関数じゃなく，JIT（just in time）でコンパイルする C++等相当のコードに変換（コンパイル）して並列実行できるようにしている．これを計算カーネルと呼ぶ．
def initialize_mass_points():
    random_offset = ti.Vector([ti.random() - 0.5, ti.random() - 0.5]) * 0.1
                    # 0.0 <= ti.random() <1.0 なので，-0.05～0.05 の乱数を返す．ちょっと初期値をずらしている．なお，ti.Vector([…])は 2 次元ベクトル
    for i, j in x:
        x[i, j] = [         # i, j の位置の質点の座標の初期化
            i * quad_size - 0.5 + random_offset[0], 0.6,
            j * quad_size - 0.5 + random_offset[1]
        ]
        v[i, j] = [0, 0, 0]  # 速度は 0


@ti.kernel
def initialize_mesh_indices():      # メッシュ番号の初期化
```

```
    for i, j in ti.ndrange(n - 1, n - 1): # 2 重 for 文
        quad_id = (i * (n - 1)) + j          # 四角形の ID．ひとつの四角形に対して二つの三角形があり，その頂点は 6 個ある
        # 1st triangle of the square   質点で囲まれた四角形の一つ目の三角形の頂点
        indices[quad_id * 6 + 0] = i * n + j
        indices[quad_id * 6 + 1] = (i + 1) * n + j
        indices[quad_id * 6 + 2] = i * n + (j + 1)
        # 2nd triangle of the square   二つ目
        indices[quad_id * 6 + 3] = (i + 1) * n + j + 1
        indices[quad_id * 6 + 4] = i * n + (j + 1)
        indices[quad_id * 6 + 5] = (i + 1) * n + j

    for i, j in ti.ndrange(n, n):        # 4 つごと色を変えている
        if (i // 4 + j // 4) % 2 == 0:
            colors[i * n + j] = (0.22, 0.72, 0.52)       # RGB, 0〜1
        else:
            colors[i * n + j] = (1, 0.334, 0.52)

initialize_mesh_indices()  #  質点初期化実行

spring_offsets = []            #  配列中身無し宣言
if bending_springs:            # Flase なのでここは無し
    for i in range(-1, 2):
        for j in range(-1, 2):
            if (i, j) != (0, 0):
                spring_offsets.append(ti.Vector([i, j]))

else:                          #  つねにこっち．5*5 成分
    for i in range(-2, 3):
        for j in range(-2, 3):
            if (i, j) != (0, 0) and abs(i) + abs(j) <= 2:          #
                spring_offsets.append(ti.Vector([i, j]))       # spring_offset に成分を加える
```





ここの意味が分からないが，chatGPT はこう言っている．

spring_offsets の中身（完全リスト）

距離1（構造バネ）

```
text

( 1,  0)
(-1,  0)
( 0,  1)
( 0, -1)
```

距離2（せん断・補助バネ）

```
text

( 2,  0)
(-2,  0)
( 0,  2)
( 0, -2)
```

斜め方向（せん断バネ）

```
text

( 1,  1)
( 1, -1)
(-1,  1)
(-1, -1)
```

👉 合計 12 個

```
text

spring_offsets = [
  (±1, 0), (0, ±1),
  (±2, 0), (0, ±2),
  (±1, ±1)
]
```

※ それぞれが `ti.Vector([dx, dy])`

```
@ti.kernel
def substep():
    for i in ti.grouped(x):          # 質点の番号
```

4. `ti.ndrange` と `ti.grouped`

```
python

for i, j in ti.ndrange(n-1, n-1):
```

- `ti.ndrange`
  → 多次元インデックスの並列ループ

```
python

for i in ti.grouped(x):
```

- `ti.grouped(x)`
  → `i` が `(i,j)` のベクトルになる
  - `i[0]`, `i[1]` でアクセス

2 次元配列を一次元で表すのか...

```
        v[i] += gravity * dt          # 重力による加速度の時間積分．分ける必要あるのか？
```

```
    for i in ti.grouped(x):                # 質点の番号
        force = ti.Vector([0.0, 0.0, 0.0])      # 力の初期化
        for spring_offset in ti.static(spring_offsets):  # 質点周りのバネの番号
            j = i + spring_offset                # ベクトル
            if 0 <= j[0] < n and 0 <= j[1] < n:
                x_ij = x[i] - x[j]
                v_ij = v[i] - v[j]
                d = x_ij.normalized()                    # 質点間単位ベクトル計算
                current_dist = x_ij.norm()               # 質点間距離（バネの長さ）計算
                original_dist = quad_size * float(i - j).norm() # バネの自然長計算
                # Spring force
                force += -spring_Y * d * (current_dist / original_dist - 1)         # F += k*dx: k の単位は上記の通り[N/kg]
                # Dashpot damping
                force += -v_ij.dot(d) * d * dashpot_damping * quad_size          # F += c*dv

        v[i] += force * dt                    # v += ΣF*dt になっている．すべて単位質量当たりの値．F の単位は，[N/kg]

    for i in ti.grouped(x):
        v[i] *= ti.exp(-drag_damping * dt)
```

これは次の微分方程式を **厳密解**で1ステップ進めています：

$$\frac{d\mathbf{v}}{dt} = -\gamma \mathbf{v}$$

- $\mathbf{v}$：速度
- $\gamma = \text{drag\_damping}$

厳密解

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t)\, e^{-\gamma \Delta t}$$

単純なオイラー法だと，

v(t+dt) = v(t) – c/m * v(t)*dt=(1-c/m*dt)*v(t)

programming code だと，

v *= (1-c/m*dt)

で数値積分するが，これだと dt が粗いとき右辺がマイナスになる可能性があり数値的に不安定になる．よって，ここだけ解析的に積分して厳密解として解いて代入している．

v *= exp(-c/m*dt)

粗い dt でも数値計算が安定するようにするざっくり力学の定番．イメージは，

慣性項＝粘性項＋剛性項

の積分するときに，粘性項の一部だけ厳密解を使い，その他は dt に依存した陽解法にするような感じ．さらに言うと，このスキームだと，剛性項によって更新された速度に対して粘性項を計算している．

```
    offset_to_center = x[i] - ball_center[0]          # 球と質点との衝突計算
    if offset_to_center.norm() <= ball_radius:
        # Velocity projection
        normal = offset_to_center.normalized()  # 単位法線ベクトル
        v[i] -= min(v[i].dot(normal), 0) * normal            # v と n の内積．逆向きなら 0．面に法線方向の速度成分だけを 0 にしている．めり込みもせず，跳ね返りもせず，衝突した瞬間に法線方
```

```
                                                    向速度が 0 になるので力学的には全く正しくないが，布なら球にめり込むはずもなく尤もらしく見えて計算速度も速くそれっぽくも見える．押しつけ力が無いのでクーロン摩擦も設定できない．
            x[i] += dt * v[i]                       # 質点の位置を更新

@ti.kernel
def update_vertices():
    for i, j in ti.ndrange(n, n):
        vertices[i * n + j] = x[i, j]               # 頂点変数に質点位置を代入

window = ti.ui.Window("Taichi Cloth Simulation on GGUI", (1024, 1024),      #描画ウィンドウ生成
                      vsync=True)
canvas = window.get_canvas()                                                #
canvas.set_background_color((1, 1, 1))                                      # 背景色
scene = ti.ui.Scene()
camera = ti.ui.Camera()                                                 # カメラ視線はデフォルト

current_t = 0.0                                                 # 初期時間
initialize_mass_points()                                        # 質点位置初期化

k = 0
os.makedirs("image", exist_ok=True)                             # linux で image ディレクトリを生成

while current_t < 1.5:                                  # 1.5 秒のシミュレーション
    for i in range(substeps):
        substep()
        current_t += dt
    update_vertices()                                   # 頂点更新

    camera.position(0.0, 0.0, 3)                        # カメラコンフィグ：位置
    camera.lookat(0.0, 0.0, 0)                          # 視点
    scene.set_camera(camera)                            # カメラ生成

    scene.point_light(pos=(0, 1, 2), color=(1, 1, 1))   # 光源位置と色
    scene.ambient_light((0.5, 0.5, 0.5))                # 周囲の RGB の強さ
    scene.mesh(vertices,                                # 物体の色
               indices=indices,
               per_vertex_color=colors,
               two_sided=True)

    # Draw a smaller ball to avoid visual penetration   動かない球の設定
    scene.particles(ball_center, radius=ball_radius * 0.95, color=(0.5, 0.42, 0.8))
    canvas.scene(scene)
    window.show()

    window.save_image(f"./image/frame{k:02d}.png")          # カメラの静止画の png 保存
    k+=1

files = sorted(glob.glob("./image/frame*.png"))         # ファイル名
clip = ImageSequenceClip(files, fps=60)                 # 60fps で保存
clip.write_videofile(                                   # mp4 で動画の保存
    "out.mp4",
    codec="libx264",
    audio=False
)
input()                                                 # 入力待ち
```

(2) 実行

taichi_env が存在するディレクトリで以下を実行して Taichi をアクティベイト
> source taichi_env/bin/activate

clothFall.py が存在するディレクトリで python を実行

> python3 clothFall.py

布の落下を描画しながら計算が進み,

. image ディレクトリ：時系列の静止画 png ファイル
. imugu.ini：テキストの結果ファイル
. out.mp4：mp4 で動画

を生成する.