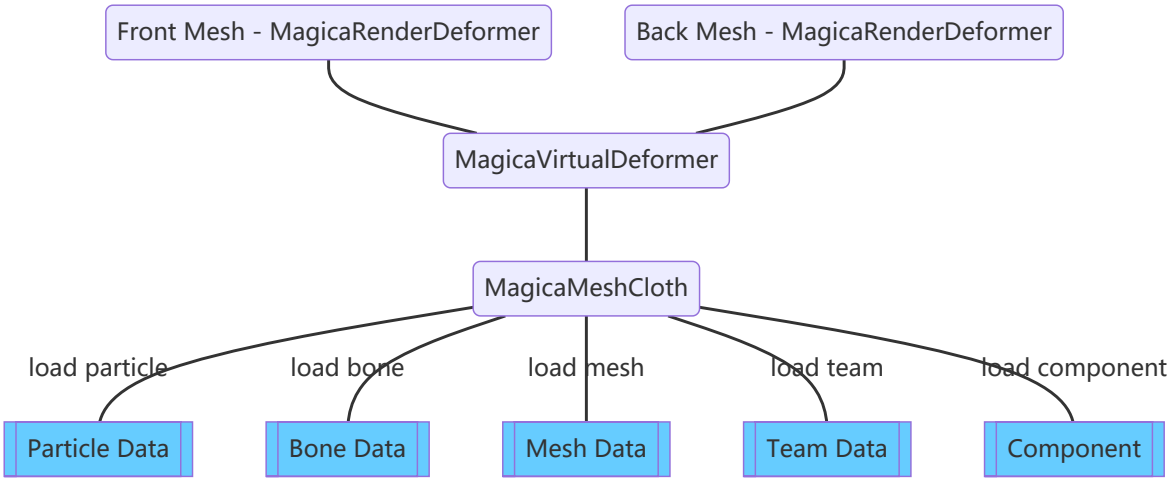
MagicaCloth v1.11.0 Developer Manual

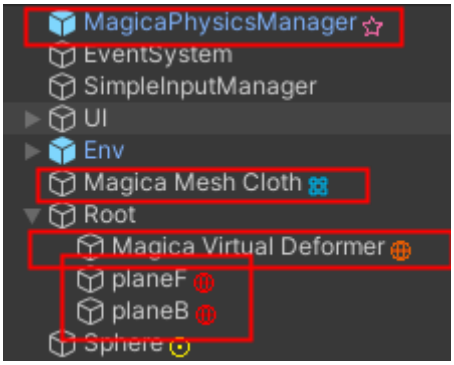
Content

Overview

Scheme



一个典型的MagicaCloth工程是这样的——



最下面的两个双面Mesh定义一块布料的顶点与Mesh结构，各自挂载一个Magica Render Deformer组件，通过面板操作传输到Magica Virtual Deformer中生成被简化的Virtual Mesh。在Magica Mesh Cloth中包含Magica Virtual Deformer以读取生成的Virtual Mesh获取原始数据。接下来在Magica Mesh Cloth中设置布料的各种初始物理属性，点击Create按钮使得在Editor阶段就完成布料的初始化生成。

进入Runtime阶段后，Magica Physcis Manager将加载MagicaMeshCloth中预生成的布料数据执行物理模拟循环。在循环结束后再次写入Magica Mesh Cloth进行渲染。

Files Structure

Magica Cloth v1.11.0 的文件结构大致如下

MagicaCloth

- Example
- Scripts
    - Core

        - API：一些API接口
        - Avatar：一些贴图相关逻辑
        - Cloth：布料相关模块
        - Define：一些定义
        - Deformer：形变器、用于生成Virtual Mesh
        - Physics：Runtime阶段的全部代码
    - ReductionMesh：网格消减。在运行前执行。

Base Concepts

Data Structure: `ChunkData`

> **主要内容：**
>
> Chunk，意为一大块东西。`ChunkData`定义了在一大片数据中一个数据开始点位与数据长度。`ChunkData`能够用于支持**大内存（数组）复用**以及**相同数据集中管理**。

```
/// File: ChunkData.cs
/// Line: 13 - 27

/// <summary>
/// データ配列のこのチャンクの開始インデックス
/// </summary>
public int startIndex;

/// <summary>
/// データ数
/// </summary>
public int dataLength;

/// <summary>
/// データ数内の使用されているローカルインデックス
/// (FixedMultiNativeListで使用)
/// </summary>
public int useLength;
```

> **主要用途：**
>
> MagicaCloth多采用提前划分一大片**固定内存**用于存储绝大部分相同类别的数据。例如，提前划分能够存储64个`float3`数据的内存，假设定义一个物体具有10个顶点，将其存储于第6~15个内存之中，则此ChunkData的开始Index为5，dataLength为10。

`ChunkData`主要主要用于支持FixedChunkNativeArray等。

Data Structure: `CurveParam`

> 主要内容：
>
> 插值数据结构，定义一个起点、一个终点以及一个插值权重。允许线性插值与Bézier插值。起点终点设置为一样时作用和普通的 `float` 一样。

```
/// File: CurveParam.cs
/// Line: 14 -17

public float sval;
public float eval;
public float cval;
public int useCurve;
```

FixedContainer: `FixedChunkNativeArray<T>`

> **主要内容：**
>
> 一个固定容器，通过ChunkData而具有复用、回收数据的功能。一个FixedChunkNativeArray同时包含两个NativeArray泛型容器，用于在Job中交替作为读写对象进行处理。

```
/// File: FixedChunkNativeArray.cs
/// Line: 18 - 39

/// <summary>
/// ネイティブリスト
/// </summary>
NativeArray<T> nativeArray0;

NativeArray<T> nativeArray1;

/// <summary>
/// ネイティブリストの配列数
/// ※ジョブでエラーが出ないように事前に確保しておく
/// </summary>
int nativeLength;

/// <summary>
/// 空インデックススタック
/// </summary>
List<ChunkData> emptyChunkList = new List<ChunkData>();

/// <summary>
/// 使用インデックスセット
/// </summary>
List<ChunkData> useChunkList = new List<ChunkData>();
```

> **主要用途：**
>
> MagicaCloth中最基本的几个容器类之一，与其余几个固定容器共同存储MagicaCloth的绝大部分数据。

Group Management: `PhysicsTeam` & `PhysicsTeamData`

> **主要内容：**
>
> `PhysicsTeamData` 内部主要定义了该组物理物体需要查询的碰撞组件列表。 `PhysicsTeam` 则是把同属于一组的物体的Particle Chunk。（例如：一块布料自己作为一个Team，则Particle Chunk内存储这个布料自己的粒子对应的Chunk）

`PhysicsTeamData`

```
/// File: PhysicsTeamData.cs
/// Line: 15 - 47

// チーム固有のコライダーリスト
[SerializeField]
private List<ColliderComponent> colliderList = new List<ColliderComponent>();

/// <summary>
/// 移動制限で無視するコライダーリスト
/// </summary>
[SerializeField]
private List<ColliderComponent> penetrationIgnoreColliderList = new
List<ColliderComponent>();

// ...

/// <summary>
/// 親アバターのコライダーを結合するかどうか
/// </summary>
[SerializeField]
private bool mergeAvatarCollider = true;

//================================================================================
=====
/// <summary>
/// ランタイムに追加されたコライダー
/// </summary>
private List<ColliderComponent> addColliderList = new List<ColliderComponent>();
```

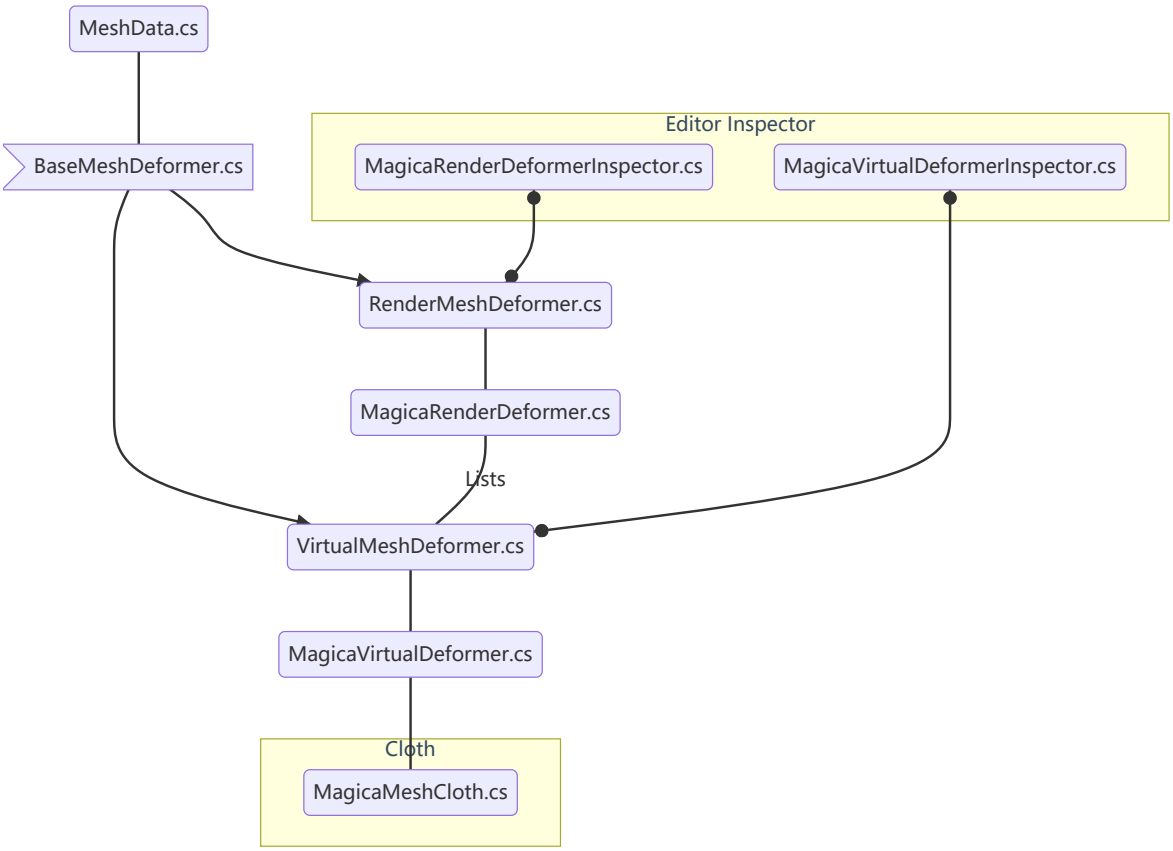
`PhysicsTeam`

```
/// File: PhysicsTeam.cs
/// Line: 62 - 65

/// <summary>
/// この物理チームで管理するパーティクル
/// </summary>
protected ChunkData particleChunk = new ChunkData();
```

Module - Deformer

Introduction

Deformer模块 主要负责从Unity Editor读取基本的顶点与Mesh等信息，调取 **RuductionMesh模块** 生成虚拟Mesh，并存储在 `MagicaVirtualDeformer` 中输入 `MagicaMeshCloth` 进入 **Cloth模块** 。



MeshData

> `MeshData` 类定义与存储了在属于预处理阶段最底层的数据，包括顶点、Mesh、子节点Mesh等等基础信息。

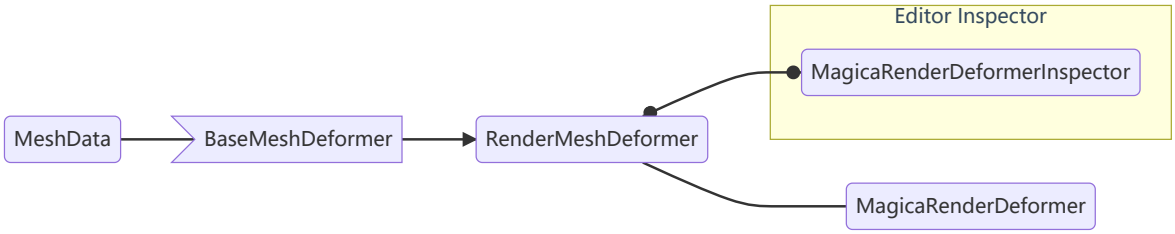
Recalculate Mode

- None
- Update Normal Per Frame
- Update Normal And Tangent Per Frames

```
/// File: MeshData.cs
/// Line: 34 - 102

/// <summary>
/// スキニングメッシュかどうか
/// </summary>
public bool isSkinning;

/// <summary>
/// 頂点数（必須）
/// </summary>
public int vertexCount;

/// <summary>
/// 頂点ごとのウエイト数とウエイト情報スタートインデックス
/// 上位4bit = ウエイト数
/// 下位28bit = スタートインデックス
/// </summary>
public uint[] vertexInfoList;

/// <summary>
/// 頂点ウエイトリスト
/// </summary>
public VertexWeight[] vertexWeightList;

/// <summary>
/// 頂点ハッシュデータ（オプション）
/// </summary>
public ulong[] vertexHashList;

/// <summary>
/// UVリスト（接線再計算用）
/// </summary>
public Vector2[] uvList;

/// <summary>
/// ライン数
/// </summary>
public int lineCount;

/// <summary>
/// ライン構成リスト（ライン数 x 2）
/// </summary>
public int[] lineList;

/// <summary>
/// トライアングル数
/// </summary>
public int triangleCount;

/// <summary>
/// トライアングル構成リスト（トライアングル数 x 3）
/// </summary>
public int[] triangleList;

/// <summary>
/// ボーン数
```

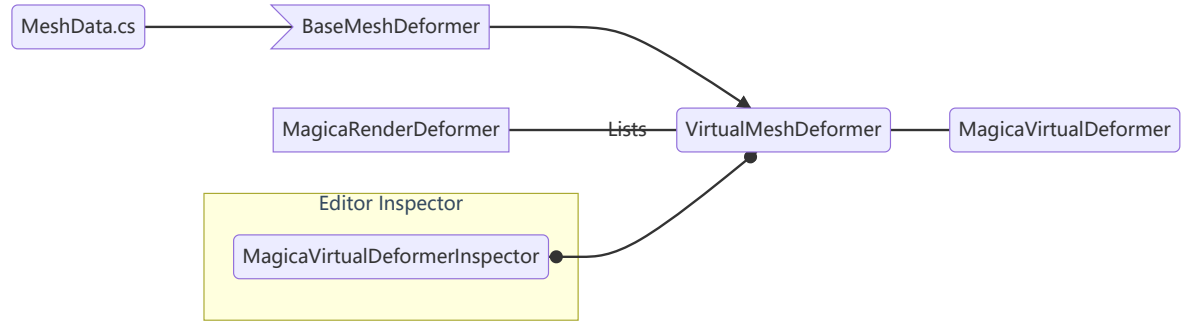
```
/// </summary>
public int boneCount;

/// <summary>
/// 仮想メッシュ頂点が属するトライアングル情報
/// 上位8bit = 接続トライアングル数
/// 下位24bit = 接続トライアングルリスト(vertexToTriangleIndexList)の開始インデックス
/// </summary>
public uint[] vertexToTriangleInfoList;

/// <summary>
/// 仮想メッシュ頂点が属するトライアングルインデックスリスト
/// これは頂点数とは一致しない
/// </summary>
public int[] vertexToTriangleIndexList;

// and more...
```

Component - MagicaRenderDeformer



> 主要内容与用途：
>
> `RenderMeshDeformer` 继承自 `BaseMeshDeformer`，拥有 `MeshData` 成员。当被附加在一个具有Renderer的GameObject上时，`MagicaRenderDeformer` 将该物体的Mesh信息读入（**使用复制 Clone**）`MeshData` 中。

Component - MagicaVirtualDeformer

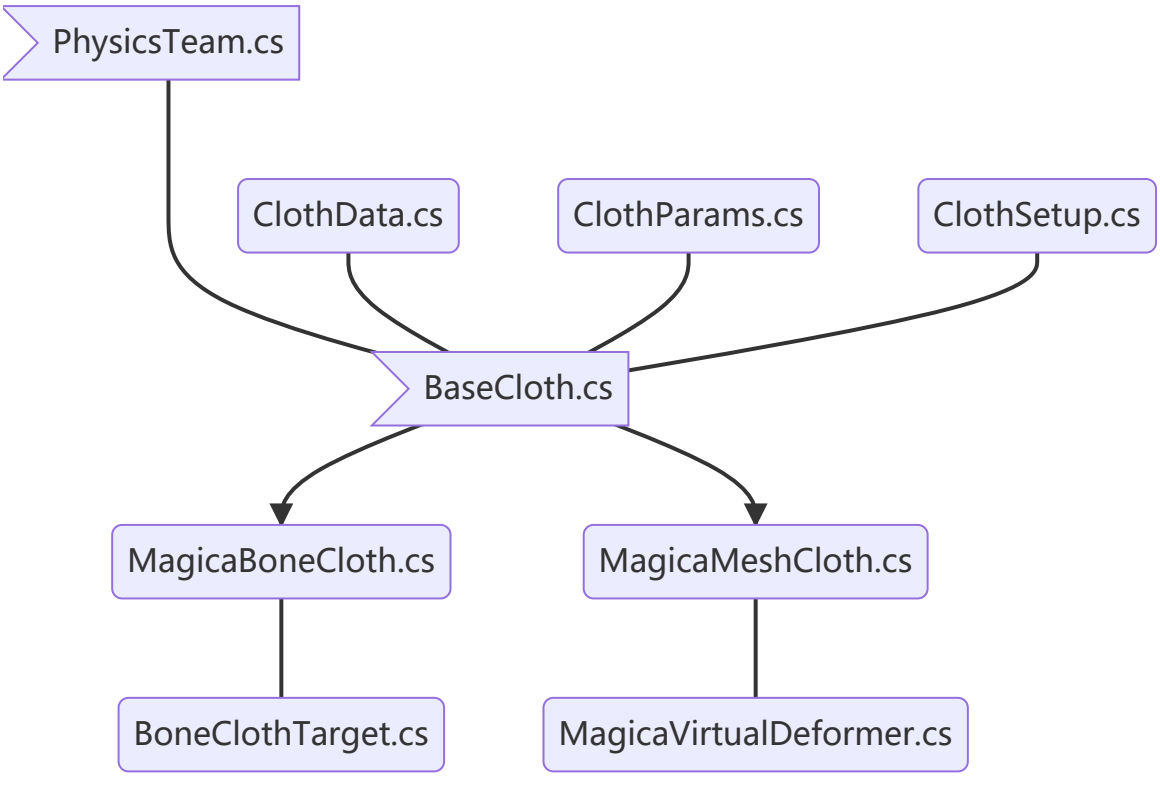


> 主要内容与用途：
>
> `VirtualMeshDeformer` 继承自 `BaseMeshDeformer`，拥有 `MeshData` 成员。通过 `MagicaVirtualDeformerInspector` 在Editor阶段就完成了Mesh Reduction生成Virtual Mesh存储在 `MeshData` 中。

Module - Cloth

Introduction

Cloth模块 主要用于初始化从 **Deformer模块** 接受的Raw Cloth数据，通过GUI调整Cloth的初始化参数，并将数据送入 **Physics Manager模块** 进入Runtime阶段进行模拟操作。



ClothData

> `ClothData` 是Runtime阶段之前数据最终的归处。在Editor编辑的数据将最终存储在这里，用于最终利用 `ClothSetup` 类将所有布料信息传输到 **Physics Manager模块** 执行物理模拟。

```
/// File: ClothData.cs
/// Line: 33 -

/// <summary>
/// メッシュの利用する頂点インデックスのリスト
/// これがそのままパーティクルとして作成される
/// クロスデータはこのリストのインデックスをデータとして指すようにする used vertex list
/// </summary>
public List<int> useVertexList = new List<int>();

/// <summary>
/// 頂点選択データ
/// SelectionDataクラスのInvalid/Move/Fixed/Extend値 selected vertex list
/// </summary>
public List<int> selectionData = new List<int>();

// and more ...
```

ClothParams

> `ClothParams` 定义了用于初始化布料所需要的参数，如：粒子质量、粒子半径、重力、空气阻力以及约束参数等等。

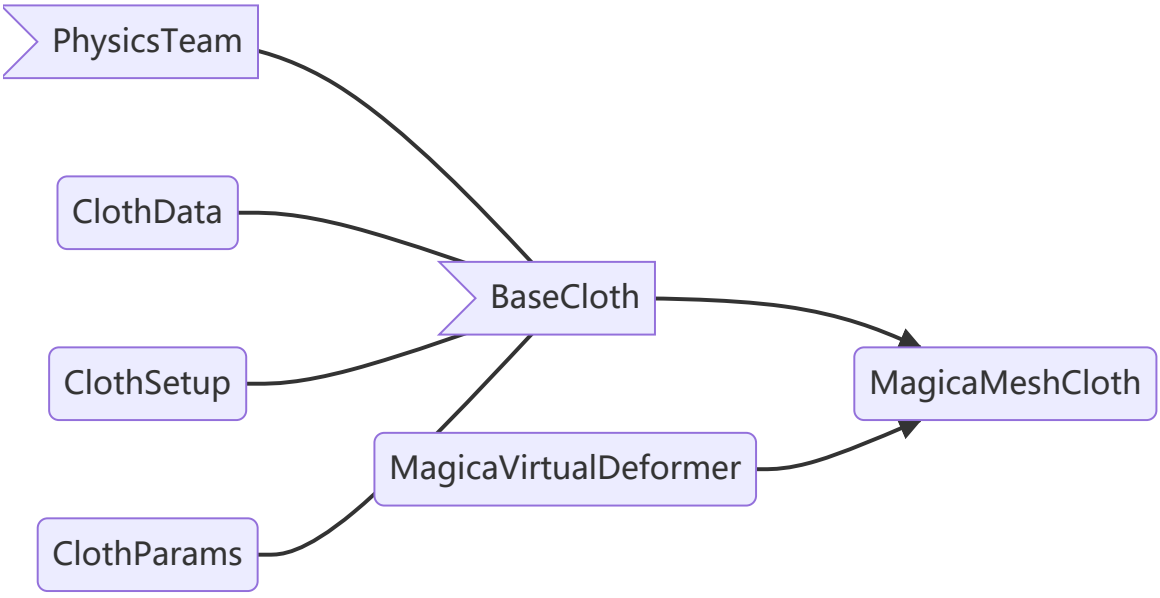
```
/// File: ClothParams.cs
/// Line: 30 - 44

// パーティクルサイズ
[SerializeField]
private BezierParam radius = new BezierParam(0.02f, 0.02f, true, 0.0f, false);

// パーティクルの重さ
[SerializeField]
private BezierParam mass = new BezierParam(1.0f, 1.0f, true, 0.0f, false);

// パーティクル重力加速度(m/s)
[SerializeField]
private bool useGravity = true;
[SerializeField]
private BezierParam gravity = new BezierParam(-9.8f, -9.8f, false, 0.0f, false);
[SerializeField]
private Vector3 gravityDirection = new Vector3(0.0f, 1.0f, 0.0f);

// and more ...
```

ClothSetup

> `ClothSetup` 用于驱动整个MagicaCloth的数据初始化。
>
> 首先， `ClothSetup` 从 **Deformer模组** 中取出在Editor阶段生成的Virtual Mesh，并使用 `ClothParams` 中的参数对 `MagicaPhysicsManager` 中的成员进行初始化（包括写入顶点、Mesh数据，初始化约束等等），进入Runtime阶段。

Component - MagicaMeshCloth

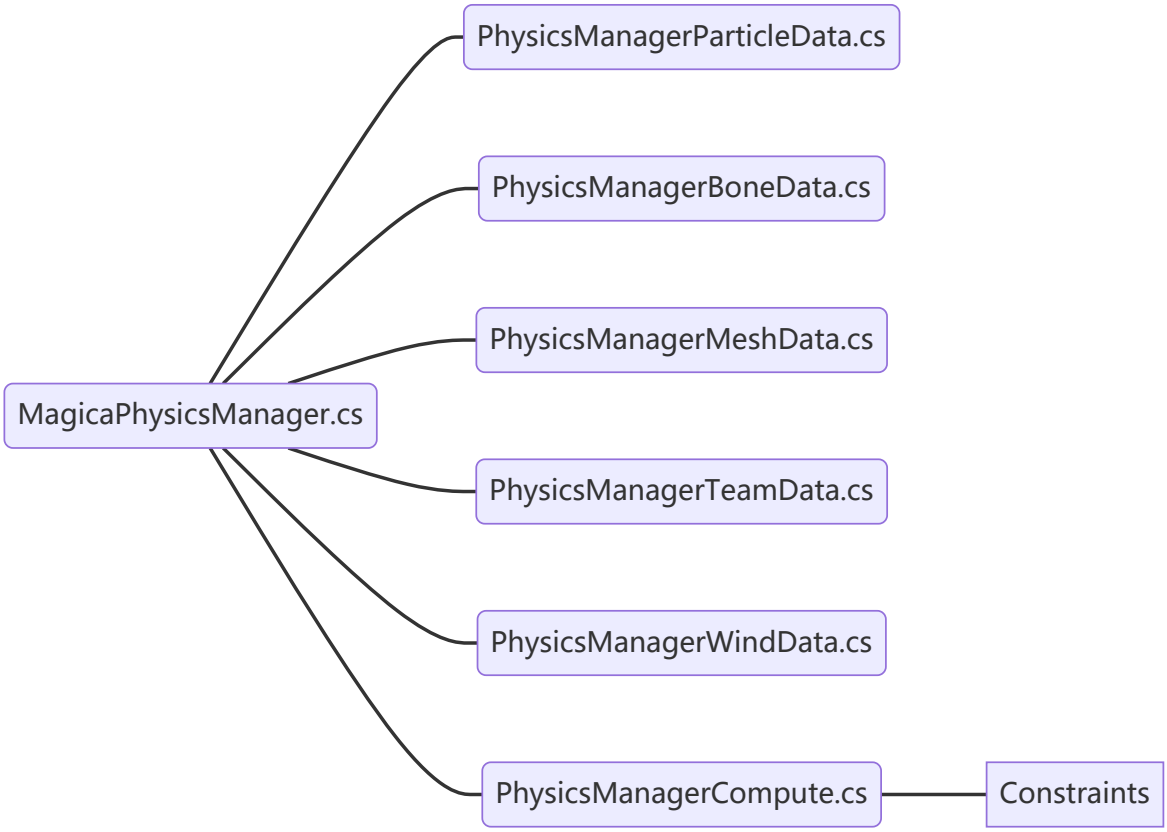
> 主要内容与用途：
>
> `MagicaMeshCloth` 所Attach的物体作为世界中的布料实体，拥有一系列完整的顶点与Mesh信息。通过 **Physics Manager** 进行模拟的顶点与Mesh信息最终也将写回 `MagicaMeshCloth` 所拥有的实例中进行渲染。



Module - Physics Manager

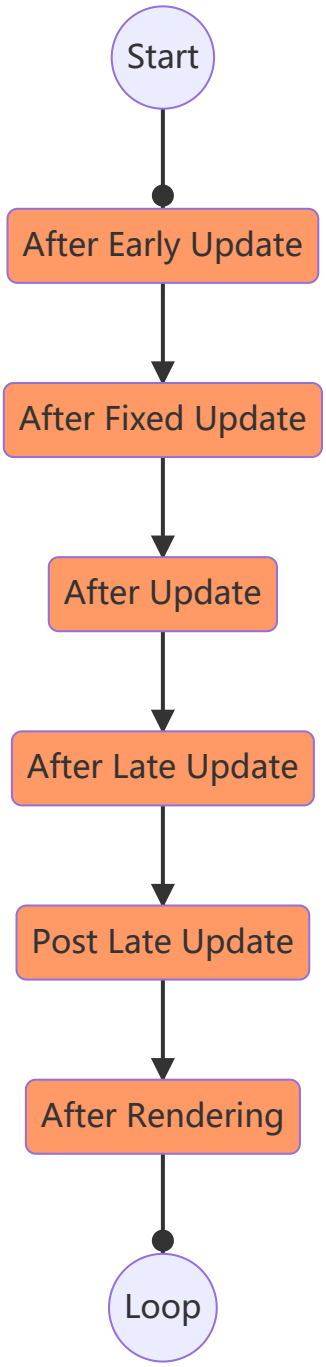
Introduction

> **Physics Manager模块** 定义与存储了所有的运行时（Runtime）数据与操作。通过包含以下六个子模块的实例推动整个物理模拟流程。



MagicaCloth Constom Player Loop

Magica Cloth自定义了一个PlayerLoop，生命周期大概如下图所示。其中，Compute物理模拟执行在After Late Update。



PhyicsManagerParticleData

> **主要内容：**
>
> `PhyicsManagerParticleData` 主要存储Runtime时所有的粒子信息。包括粒子的位置、旋转、速度等物理信息。需要注意的是，一些碰撞体以及其他非粒子的实体也共同存储在一个的顶点池中。

```
/// File: PhysicsManagerParticleData.cs
/// Line: 226 - 346

/// <summary>
/// フラグリスト flaglist
/// </summary>
public FixedChunkNativeArray<ParticleFlag> flagList;
```

```
/// <summary>
/// 所属するチームID（０＝グローバル） the teamID the particle belongs to (0 = global)
/// </summary>
public FixedChunkNativeArray<int> teamIdList;

/// <summary>
/// 現在座標リスト  current positions list
/// </summary>
public FixedChunkNativeArray<float3> posList;

/// <summary>
/// 現在回転リスト  current rotations list
/// </summary>
public FixedChunkNativeArray<quaternion> rotList;

/// <summary>
/// １つ前の座標リスト  last positions list
/// </summary>
public FixedChunkNativeArray<float3> oldPosList;

/// <summary>
/// １つ前の回転リスト  last rotations list
/// </summary>
public FixedChunkNativeArray<quaternion> oldRotList;

/// <summary>
/// １つ前の座標リスト（スロー再生用）  last positions list for slow play
/// </summary>
public FixedChunkNativeArray<float3> oldSlowPosList;

/// <summary>
/// 本来のローカル位置リスト  original local positons list
/// </summary>
public FixedChunkNativeArray<float3> localPosList;

/// <summary>
/// 本来のワールド位置リスト  original global positons list
/// </summary>
public FixedChunkNativeArray<float3> basePosList;

/// <summary>
/// 本来のワールド回転リスト   original global rotations list
/// </summary>
public FixedChunkNativeArray<quaternion> baseRotList;

/// <summary>
/// パーティクル深さリスト  particle depths list
/// </summary>
public FixedChunkNativeArray<float> depthList;

/// <summary>
/// 半径リスト  radius list
/// </summary>
public FixedChunkNativeArray<float3> radiusList;

/// <summary>
/// 復元トランスフォームリストへのインデックス  restore transform index list, -1 when
unnecessary
```

```
/// 不要な場合は(-1)
/// </summary>
public FixedChunkNativeArray<int> restoreTransformIndexList;

/// <summary>
/// 読み込み/書き込みトランスフォームリストへのインデックス  transform indexes list for
read/write purpose, -1 when unnecessary
/// 不要な場合は(-1)
/// </summary>
public FixedChunkNativeArray<int> transformIndexList;

/// <summary>
/// 現在の摩擦係数リスト  current frictiones list
/// </summary>
public FixedChunkNativeArray<float> frictionList;

/// <summary>
/// 現在の静止摩擦係数リスト  current static frictiones list
/// </summary>
public FixedChunkNativeArray<float> staticFrictionList;

/// <summary>
/// 現在の速度リスト  current velocities list
/// </summary>
public FixedChunkNativeArray<float3> velocityList;

/// <summary>
/// 接触コライダーID(0=なし)  collision linkId list, 0 when null
/// </summary>
public FixedChunkNativeArray<int> collisionLinkIdList;

/// <summary>
/// 接触コライダーの衝突法線
/// </summary>
public FixedChunkNativeArray<float3> collisionNormalList;

/// <summary>
/// 作業用座標リスト0  positions list0 for task
/// </summary>
FixedChunkNativeArray<float3> nextPos0List;

/// <summary>
/// 作業用座標リスト1  positions list1 for task
/// </summary>
FixedChunkNativeArray<float3> nextPos1List;

/// <summary>
/// 作業用座標リストの切り替えスイッチ  positions lists switch for task
/// </summary>
int nextPosSwitch = 0;

/// <summary>
/// 作業用回転リスト0  rotations list0 for task
/// </summary>
FixedChunkNativeArray<quaternion> nextRot0List;

/// <summary>
/// 作業用回転リスト1  rotations list1 for task
```

```
/// </summary>
FixedChunkNativeArray<quaternion> nextRot1List;

/// <summary>
/// 作業用回転リストの切り替えスイッチ  rotations lists switch for task
/// </summary>
int nextRotSwitch = 0;
```

PhysicsManagerBoneData

> **主要内容：**
>
> `PhysicsManagerBoneData` 为MagicaBoneCloth提供定义，大体结构与 `PhyicsManagerParticleData` 类似。

```
/// <summary>
/// 管理ボーンリスト
/// </summary>
public FixedTransformAccessArray boneList;

/// <summary>
/// ボーンフラグリスト
/// </summary>
public FixedNativeList<byte> boneFlagList;

/// <summary>
/// ボーンワールド位置リスト（※未来予測により補正される場合あり）
/// </summary>
public FixedNativeList<float3> bonePosList;

/// <summary>
/// ボーンワールド回転リスト（※未来予測により補正される場合あり）
/// </summary>
public FixedNativeList<quaternion> boneRotList;

/// <summary>
/// ボーンワールドスケールリスト（現在は初期化時に設定のみ不変）
/// </summary>
public FixedNativeList<float3> boneSclList;

/// <summary>
/// 親ボーンへのインデックス(-1=なし)
/// </summary>
public FixedNativeList<int> boneParentIndexList;

/// <summary>
/// ボーンワールド位置リスト（オリジナル）
/// </summary>
public FixedNativeList<float3> basePosList;

/// <summary>
/// ボーンワールド回転リスト（オリジナル）
/// </summary>
public FixedNativeList<quaternion> baseRotList;
```

```
/// <summary>
/// ボーンがUnityPhysicsで動作するかの参照カウンタ（1以上で動作）
/// </summary>
public FixedNativeList<short> boneUnityPhysicsList;

/// <summary>
/// ボーン未来予測位置リスト
/// </summary>
public FixedNativeList<float3> futurePosList;

/// <summary>
/// ボーン未来予測回転リスト
/// </summary>
public FixedNativeList<quaternion> futureRotList;

//================================================================================================
/// <summary>
/// 復元ボーンリスト
/// </summary>
public FixedTransformAccessArray restoreBoneList;

/// <summary>
/// 復元ボーンの復元ローカル座標リスト
/// </summary>
public FixedNativeList<float3> restoreBoneLocalPosList;

/// <summary>
/// 復元ボーンの復元ローカル回転リスト
/// </summary>
public FixedNativeList<quaternion> restoreBoneLocalRotList;

/// <summary>
/// 復元ボーンの参照ボーンインデックス
/// </summary>
public FixedNativeList<int> restoreBoneIndexList;

//================================================================================================
// ここはライトボーンごと
/// <summary>
/// 書き込みボーンリスト
/// </summary>
public FixedTransformAccessArray writeBoneList;

/// <summary>
/// 書き込みボーンの参照ボーン姿勢インデックス（＋1が入るので注意！）
/// </summary>
public FixedNativeList<int> writeBoneIndexList;

/// <summary>
/// 書き込みボーンの対応するパーティクルインデックス
/// </summary>
public ExNativeMultiHashMap<int, int> writeBoneParticleIndexMap;

/// <summary>
/// 読み込みボーンに対応する書き込みボーンのインデックス辞書
/// </summary>
```

```
Dictionary<int, int> boneToWriteIndexDict = new Dictionary<int, int>();

/// <summary>
/// 書き込みボーンの確定位置
/// 親がいる場合はローカル、いない場合はワールド格納
/// </summary>
public FixedNativeList<float3> writeBonePosList;

/// <summary>
/// 書き込みボーンの確定回転
/// 親がいる場合はローカル、いない場合はワールド格納
/// </summary>
public FixedNativeList<quaternion> writeBoneRotList;
```

PhysicsManagerMeshData

> **主要内容：**
>
> `PhysicsManagerMeshData` 主要存储所有模拟物体的Mesh信息，并提供在模拟结束后写回渲染网格过程的方法。

PhysicsManagerTeamData

> **主要内容：**
>
> `PhysicsManagerTeamData` 主要存储这所有模拟物体的Team，定义与管理各个Team之间的物理特性、**约束关系**与**碰撞关系**。

Critical Fields

```
/// File: PhysicsManagerTeamData.cs
/// Line: 305 - 313

/// <summary>
/// チームデータリスト
/// </summary>
public FixedNativeList<TeamData> teamDataList;

public FixedNativeList<CurveParam> teamMassList;
public FixedNativeList<CurveParam> teamGravityList;
public FixedNativeList<CurveParam> teamDragList;
public FixedNativeList<CurveParam> teamMaxVelocityList;

/// Line: 316 - 378

/// <summary>
/// チームのワールド移動回転影響
/// </summary>
public FixedNativeList<WorldInfluence> teamWorldInfluenceList;

/// <summary>
/// チームごとの判定コライダー
/// </summary>
```

```
public FixedMultiNativeList<int> colliderList;

/// <summary>
/// チームごとのチームコンポーネント参照への辞書（キー：チームID）
/// nullはグローバルチーム
/// </summary>
private Dictionary<int, PhysicsTeam> teamComponentDict = new Dictionary<int,
PhysicsTeam>();
```

PhysicsManagerWindData

> **主要内容：**
>
> `PhysicsManagerWindData` 主要是定义风力的一些信息，内容比较少。

WindData

主要参数

```
/// File: PhysicsManagerWindData.cs
/// Line: 53 - 66

/// <summary>
/// 風量 wind force magnitude
/// </summary>
public float main;

/// <summary>
/// 乱流率(0.0-1.0)
/// </summary>
public float turbulence;

/// <summary>
/// 現在の風の方向（ここが計算で使用される）
/// </summary>
public float3 direction;
```

WindGeneration


```
/// File: PhysicsManagerWindData.cs
/// Line: 301 - 329

// コンポーネント姿勢
var bpos = bonePosList[wdata.transformIndex];
var brot = boneRotList[wdata.transformIndex];

// 風量による計算比率
float ratio = wdata.main / 30.0f; // 風速30を基準

// 周期（風向きが変わる速度）
```

```
float freq = 1.0f + 2.0f * ratio; // 1.0 - 3.0

// 風向きのランダム角度
float rang = 15.0f + 15.0f * ratio; // 15 - 30

// ノイズ参照
var noisePos1 = new float2(bpos.x, bpos.z) * 0.1f;
var noisePos2 = new float2(bpos.x, bpos.z) * 0.1f;
noisePos1.x += elapsedTime * freq; // 周期（数値を高くするとランダム性が増す）2.0f?
noisePos2.y += elapsedTime * freq; // 周期（数値を高くするとランダム性が増す）2.0f?
var nv1 = noise.snoise(noisePos1); // -1.0f～1.0f
var nv2 = noise.snoise(noisePos2); // -1.0f～1.0f

// 方向のランダム性
var ang1 = math.radians(nv1 * rang);
var ang2 = math.radians(nv2 * rang);
ang1 *= wdata.turbulence; // 乱流率
ang2 *= wdata.turbulence; // 乱流率
var rq = quaternion.Euler(ang1, ang2, 0.0f); // XY
var dir = math.forward(math.mul(brot, rq)); // ランダムはローカル回転
wdata.direction = dir;
```

PhysicsManagerComponent

> **主要内容：**
>
> `PhysicsManagerComponent` 内存储了 MagicaVirtualDeformer、MagicaVirtualDeformer 与 MagicaMeshCloth 的实例，以便于将每一次模拟的结果写回这些实例用于更新场景物体。

```
/// File: PhysicsManagerComponent.cs
/// Line: 13 - 18

/// <summary>
/// すべてのコンポーネントのセット
/// これは初期化の成否に関係なく無条件で登録されるので注意!
/// 初期化完了の有無は comp.Status.IsInitSuccess で判定する
/// </summary>
private HashSet<CoreComponent> componentSet = new HashSet<CoreComponent>();
```

PhysicsManagerCompute

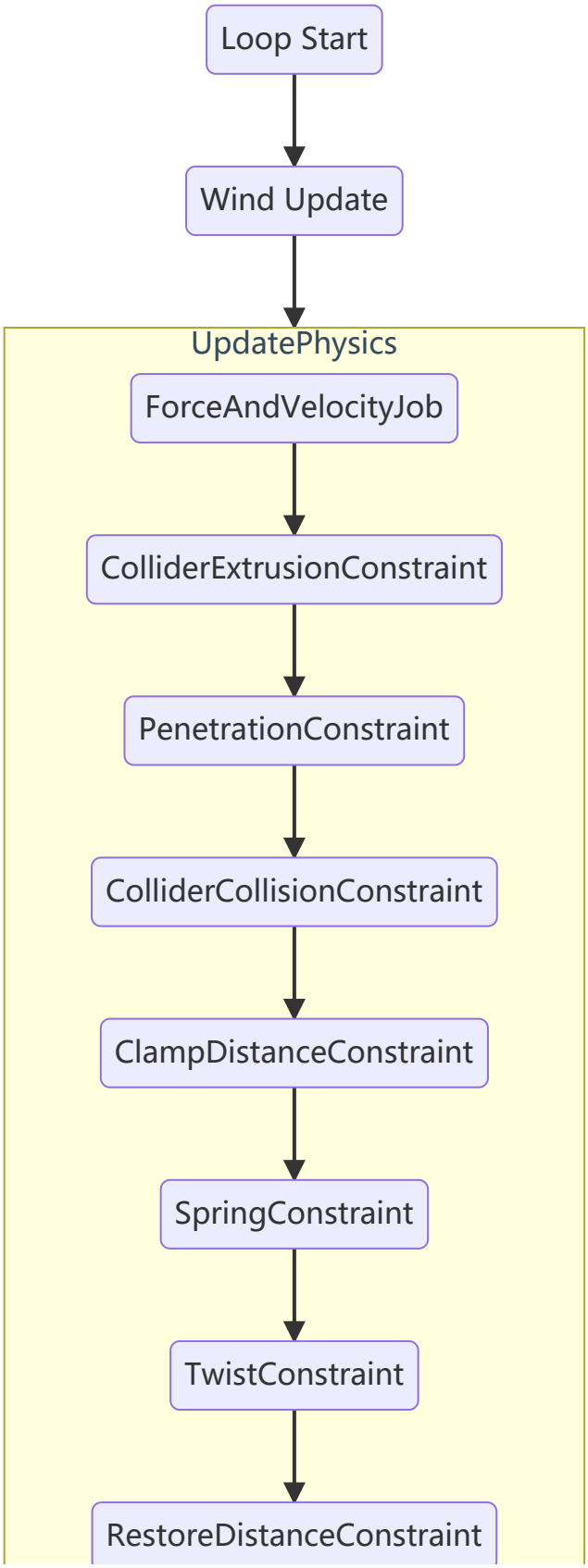
> **主要内容：**
>
> `PhysicsManagerCompute` 是整个物理模拟框架的核心，采用标准 Position Based Dynamics 模型。

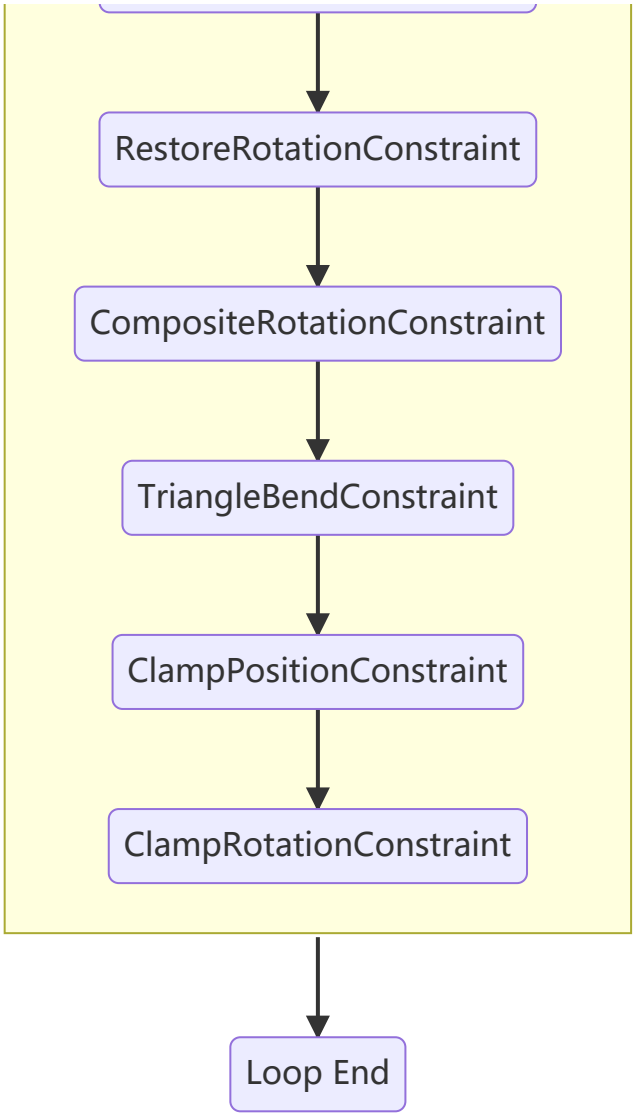
Critical Fields

```
/// File: PhysicsManagerCompute.cs
/// Line: 30, 52

List<PhysicsManagerConstraint> constraints;

List<PhysicsManagerWorker> workers;
```

Critical Methods

```
public void UpdateStartSimulation(UpdateTimeManager update);
```





Position Based Dynamics Scheme

(1) **forall** vertices i
(2) initialize $\mathbf{x}_i = \mathbf{x}_i^0$, $\mathbf{v}_i = \mathbf{v}_i^0$, $w_i = 1/m_i$
(3) **endfor**
(4) **loop**
(5) **forall** vertices i **do** $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{\text{ext}}(\mathbf{x}_i)$
(6) dampVelocities$(\mathbf{v}_1, \ldots, \mathbf{v}_N)$
(7) **forall** vertices i **do** $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
(8) **forall** vertices i **do** generateCollisionConstraints$(\mathbf{x}_i \rightarrow \mathbf{p}_i)$
(9) **loop** solverIterations **times**
(10) projectConstraints$(C_1, \ldots, C_{M+M_{\text{coll}}}, \mathbf{p}_1, \ldots, \mathbf{p}_N)$
(11) **endloop**
(12) **forall** vertices i
(13) $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$
(14) $\mathbf{x}_i \leftarrow \mathbf{p}_i$
(15) **endfor**
(16) velocityUpdate$(\mathbf{v}_1, \ldots, \mathbf{v}_N)$
(17) **endloop**

$$\triangle \mathbf{p_i} = -\mathbf{s}\frac{\mathbf{n} * \mathbf{w_i}}{\mathbf{\Sigma_j w_j}} \nabla_{\mathbf{p_i}} \mathbf{C}(\mathbf{p_1}, \ldots, \mathbf{p_n})$$

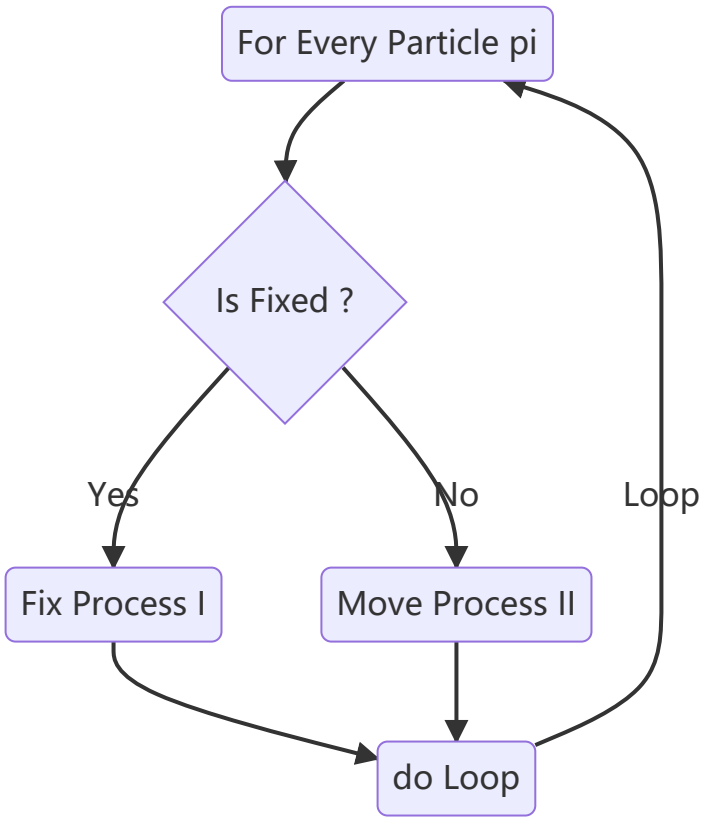
其中

- $ \mathbf{ C(p_{1},...,p_{n}) } $: 约束函数，用于约束一个顶点集合$\mathbf{p_1, ..., p_n}$。
- $\nabla_{p_i} C(p_1, ..., p_n)$: 约束函数对顶点$\mathbf{p_i}$取梯度。
- $\mathbf{w}$: 权重，通常为质量的倒数，取$\frac{1}{\mathbf{m}}$。
- $\mathbf{s}$: scale张量，是一个标量，其中

$$\mathbf{s} = \frac{\mathbf{C(p_1, ..., p_n)}}{\mathbf{\Sigma_j} ||\nabla_{\mathbf{p_i}} \mathbf{C(p_1, ..., p_n)}||^2}$$

ForceAndVelocityJob

Process Diagram



Move Process Kernel

辛欧拉方法进行状态预测（Symplectic Euler） (PBD-5,6,7)

- 速度预测：$\mathbf{v_{t+1}} = \mathbf{v_t} + \triangle \mathbf{t} * \mathbf{f(x, t)/m}$
- 位移预测：$\mathbf{x_{t+1}} = \mathbf{x_t} + \triangle \mathbf{t} * \mathbf{v_{t+1}}$

其中，

- $\mathbf{v_{t+1}}$：下一时刻的预测位移
- $\mathbf{x_{t+1}}$: 下一时刻的预测速度
- $\mathbf{v}$：此时刻的预测位移
- $\mathbf{x}$: 此时刻的预测速度
- $\triangle \mathbf{t}$：时间间隔，通常写作$\mathbf{h}$，区别于帧间隔
- $\mathbf{m}$：顶点质量，通常以$\frac{1}{\mathbf{m}}$形式出现，以避免频繁的除法操作
- $\mathbf{f(x, t)}$：此时刻该顶点受到的合外力

> 注：辛欧拉积分的特征是先进行速度积分再进行位移积分，区别于**前向欧拉（Forward Euler）**与**隐式欧拉（Implicit Euler）**

Code

```
/// File: PhysicsManagerCompute.cs
/// Line: 712 - 791

// 速度計算(質量で割る)
velocity += (force / mass) * updateDeltaTime;

// 速度を理想位置に反映させる
nextPos = oldpos + velocity * updateDeltaTime;
```

ColliderExtrusionConstraint

碰撞体挤出约束，解决顶点与碰撞体紧贴时的误碰撞问题。

Code

```
/// File: ColliderExtrusionConstraint.cs
/// Line: 145 - 151

var ev = fpos - nextpos;
var elen = math.length(ev);
if (elen < 1e-06f)
{
    // コライダーが動いていない
    return;
}
```

ColliderCollisionConstraint

主要的碰撞检测。当被检测点进入碰撞体表面的内部，即可对被检测点出发碰撞响应。沿着碰撞点法线方向执行约束解算。

Kernel

- Constrant Function: $\mathbf{C(p)} = (\mathbf{p} - \mathbf{q_c})\mathbf{n_c}$

其中，

- $\mathbf{C(p)}$：当前粒子约束函数
- $\mathbf{p}$：当前粒子的预测位置
- $\mathbf{q_c}$：碰撞表面的点位置
- $\mathbf{n_c}$：碰撞表面的点法线

支持碰撞类型

- PlaneColliderDetection
- CapsuleColliderDetection
- SphereColliderDetection
- （未实装）BoxColliderDetection

ClampDistanceConstraint

intro

用于防止（主要作用于BoneCloth）点距离根节点过远/过近。

Kernel

- $ \mathbf{C(p) = min < ||p - r|| < max}$

其中，

- $\mathbf{p}$：当前粒子的预测位置
- $\mathbf{r}$：当前粒子所约束的根节点位置

code

```
/// File: ClampDistanceConstraint.cs
/// Line: 266 - 279

// 現在のベクトル
float3 v = nextpos - nextpos2;

// 復元長さ
float length = data.length; // v1.7.0
if (useAnimatedDistance)
{
    // アニメーションされた距離を使用
    length = math.distance(basepos, basePosList[pindex2]);
}
length *= team.scaleRatio; // チームスケール倍率

// ベクトル長クランプ
v = MathUtility.ClampVector(v, length * gdata.minRatio, length * gdata.maxRatio);
```

RestoreDistanceConstraint

Intro

恢复距离约束（RestoreDistanceConstraint），即距离约束（DistanceConstraint），用于保持约束中的两个粒子之间的相对运动距离。在偏离松弛距离过大时基于惩罚函数修正位移，以保证约束函数误差最小。

Kernel

- $ \mathbf{C(p_{1} , p_{2}) = ||p_{1} - p_{2}|| - d} $
- $\nabla_{\mathbf{p}_1}\mathbf{C} = \frac{\mathbf{p_1}-\mathbf{p_2}}{||\mathbf{p_1}-\mathbf{p_2}||}, \nabla_{\mathbf{p}_2}\mathbf{C} = -\frac{\mathbf{p_1}-\mathbf{p_2}}{||\mathbf{p_1}-\mathbf{p_2}||}$
- $\mathbf{s} = \frac{C(p_1,p_2)}{\nabla^2_{p_1}C+\nabla^2_{p_2}C} = \frac{||p_1-p_2||-d}{2}$

其中，

- $\mathbf{C(p_1, p_2)}$：当前粒子约束函数
- $\mathbf{p_1}$：当前粒子的预测位置
- $\mathbf{p_2}$：目标约束粒子的预测位置
- $\mathbf{d}$：松弛距离

因此，修正张量$\mathbf{\Delta p_i}$为

$$\Delta \mathbf{p}_1 = -\frac{\mathbf{w}_1}{\mathbf{w}_1 + \mathbf{w}_2}(||\mathbf{p}_1 - \mathbf{p}_2|| - \mathbf{d})\frac{\mathbf{p}_1 - \mathbf{p}_2}{||\mathbf{p}_1 - \mathbf{p}_2||}$$
$$\Delta \mathbf{p}_2 = \frac{\mathbf{w}_1}{\mathbf{w}_1 + \mathbf{w}_2}(||\mathbf{p}_1 - \mathbf{p}_2|| - \mathbf{d})\frac{\mathbf{p}_1 - \mathbf{p}_2}{||\mathbf{p}_1 - \mathbf{p}_2||}$$

Code

约束数据结构

```
/// File: RestoreDistanceConstraint.cs
/// Line: 31 - 44

/// <summary>
/// 計算頂点インデックス
/// </summary>
public ushort vertexIndex;

/// <summary>
/// ターゲット頂点インデックス
/// </summary>
public ushort targetVertexIndex;

/// <summary>
/// パーティクル距離(v1.7.0)
/// </summary>
public float length;
```

约束解算流程

```
/// File: RestoreDistanceConstraint.cs
/// Line: 459 - 492

// 現在の距離
float3 v = tnextpos - nextpos;
float vlen = math.length(v);
if (vlen < 0.00001f)
    continue;


// 復元距離
float rlen = data.length; // v1.7.0
if (useAnimatedDistance)
{
    // アニメーションされた距離を利用する
    rlen = math.distance(bpos, basePosList[tindex]);
}

// チームスケール倍率
rlen *= team.scaleRatio;

float clen = vlen - rlen;

// 重量差
float tdepth = depthList[tindex];
float tmass = gdata.mass.Evaluate(tdepth);
float tfriction = frictionList[tindex];
// 摩擦分重量を上げ移動しにくくする
tmass += tfriction * FrictionMass;
```

```
float m1 = tmass / (tmass + mass);

// 強さ
m1 *= stiffness;

// 移動ベクトル
float3 add1 = v * (m1 * clen / vlen);
```

TriangleBendConstraint

Intro

Kernel

- $\mathbf{C}(\mathbf{p_1},\mathbf{p_2},\mathbf{p_3},\mathbf{p_4}) = \arccos(\mathbf{n_1}\cdot\mathbf{n_2}) - \phi_0$

其中:

$$\mathbf{n_1} = \frac{(\mathbf{p_2}-\mathbf{p_1})\times(\mathbf{p_3}-\mathbf{p_1})}{||(\mathbf{p_2}-\mathbf{p_1})\times(\mathbf{p_3}-\mathbf{p_1})||}$$
$$\mathbf{n_2} = \frac{(\mathbf{p_2}-\mathbf{p_1})\times(\mathbf{p_4}-\mathbf{p_1})}{||(\mathbf{p_2}-\mathbf{p_1})\times(\mathbf{p_4}-\mathbf{p_1})||}$$

FixPositionJob

Process Diagram

Kernel

根据约束映射后的计算出的位移差，更新当前迭代时刻的速度$\mathbf{v_{t+1}}$ (PBD-12,13,14,15)

Code

```
/// File: PhysicsManagerCompute.cs
/// Line: 843 - 944

// 速度更新(m/s)
velocity = (nextPos - pos) / updateDeltaTime;
velocity *= teamData.velocityWeight; // 安定化用の速度ウエイト
```