ジオメトリノード シミュレーション 解説＆作例集

Geometry Nodes Simulation (for Blender 3.6)

Version 1.2

Q@スタジオぽぷり
@popqjp

作者 Twitter アカウント
Q@スタジオぽぷり

目次

■初めに … 3

■ジオメトリノードの基礎 … 4
ジオメトリノードの編集 … 4
アトリビュートの入出力 … 6

■ジオメトリノードシミュレーション基礎編 … 8
シミュレーションとは … 8
シミュレーション用のノードの組み方 … 9
パーティクル生成のような効果 … 12
キャッシュとシミュレーションの再計算 … 13
マテリアルとシミュレーション … 14

■位置のシミュレーション … 15
ランダムウォーク … 15
Dynamic Paint (ダイナミックペイント)的な制御 … 16
接触判定 … 17
Repeat Zone(リピートゾーン) … 19
懸垂線カーブ … 20
遅延とお化け … 23

■アトリビュートを使ったシミュレーション … 26
パーティクルの経過時間(Age) … 26
匿名のアトリビュート(ラベル名はある) … 28
前フレームの情報 … 29
色のダイナミックペイント的制御 … 31
カウンター … 32
お化け２、過去のジオメトリを使う … 34

■力学シミュレーション … 38
放物線で飛ぶパーティクル … 38
オイラー法(Euler Method) … 39
運動するパーティクルと衝突判定 … 39
カーブで軌跡を表示 … 41
パーティクルを生成するエミッター(Emitter) … 42
パーティクルとダイナミックペイント … 44
タイムストレッチ(Time Stretching) … 46

■振り子 … 47
ばね振り子 … 47
修正オイラー法 … 49
リープ・フロッグ法(Leap Frog 法) … 51
等速円運動 … 52
棒振り子 … 55

■ループとフラクタル … 56
メッシュアイランドから点 … 56
メンガーのスポンジ … 61
移動と反転によるフラクタル … 63

■平面上の波 … 65
波動方程式 … 65
境界条件の設定 … 67

■応用編 … 69

ライフゲーム(Conway's Game of Life) … 69
拡散反応(reaction diffusion)…っぽい模様を作る変形 … 72
レーザーカッター … 74

シミュレーションノード と Retiming(リタイミング) もしくは Speed Control … 77
Speed Control(スピードコントロール)でシーンの速度調整 … 78
Retiming(リタイミング)機能を利用しての速度調整 … 80
そのほかの Retimimg 機能 … 81

ツタの成長 … 82

従来のシミュレーションとの組み合わせ … 85
ソフトボディシミュレーション … 85

単純なクロスシミュレーション … 88

重力多体計算 … 92

多脚歩行 … 95

■サンプル.blendファイル … 101
■終わりに …102

ジオメトリノード シミュレーション 解説＆作例集

Geometry Nodes Simulation (for Blender 3.6)

作者 Twitter アカウント
Q@スタジオぽぷり

2023.06.29 ver 1.0
2023.12.30 ver 1.1
2024.04.25 ver 1.2

初めに

ジオメトリノードは、Blender 2.92 で導入されて以降さまざまな拡張がなされて、多くのことが行えるようになってきました。
SNSなどでも、驚くような作例が頻繁に投稿されています。
しかし、Blender 3.5 までのジオメトリノードには、シミュレーションによる時間進化を行うことができないという大きな弱点がありました。
時間進化？
動くエフェクトのジオメトリノードの作成は数多くあります。
ジオメトリノードの中で時間情報、つまり「今何フレーム目なのか」を知ることができますから、それに従って変化するエフェクトを作ることはできます。
また、他のオブジェクトの形状情報を得ることができますから「普通にアニメーションしている他のオブジェクトの形」でジオメトリノードを制御することもできます。
しかし今(Blender 3.5)まで「自分の過去(1フレーム前)の情報を得る」ことができなかったのです。

粒子シミュレーションは、前のフレームの粒子の位置を基準に、現在(次のフレーム)の粒子の位置を計算することで、物理法則に従うよう粒子の運動を計算します。
流体シミュレーションは、前のフレームの流体の形を基準に、現在(次のフレーム)の流体の形を計算することで、物理法則に従った流体の運動を計算します。
ジオメトリノードでは「前のフレームの情報を得る」ことができなかったので、そのような「過去の自分の状態から今の自分の状態を計算する」シミュレーションを行うことができなかったのです。

Blender 3.6 でシミュレーションノードが追加されたことによって、１フレーム前の自分の情報を得て、それを使ってジオメトリを計算することができるようになりました。
つまり、文字通りシミュレーションによる時間進化を行うことができるようになったのです。
この強力な機能を使った、様々な作例と解説を見ていくことにしましょう。

読者には、ある程度 Blender の操作に慣れている人を想定しています。
また、高校数学程度の、ベクトルや三角関数の知識…例えば内積や外積といったような…があった方が、理解がしやすいと思います。
また途中で１セクションほど、調子に乗って高校数学の範疇を離れて大学の理系で行いそうな内容も盛り込んだ章がありますが…そこは例外としてスキップしてしまっても大丈夫です。

ジオメトリノードの基本部分の解説は、最初の章でざっとおさらいをしておこうと思います。
「その辺の基本部分は理解しているよ」という方は、最初は飛ばして進んでも大丈夫す。

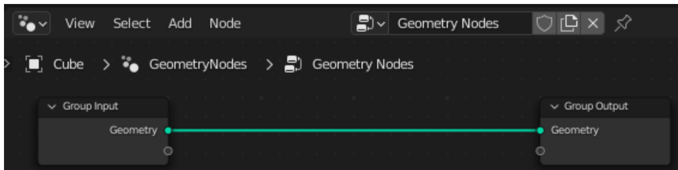
その後、Blender 4.0 になると Simulation Zone に似た Repeat Zone という便利なノードが追加されました。
ver. 1.1 ではそれも踏まえた追記やサンプルの修正を行っています。
ver. 1.2 は、引き続いて Blender 4.1 への対応とサンプルの追加を行っています。

本書に埋め込んである画像の一部は GIF アニメーションになっています。
.pdfとして書き出したファイルは、残念ながら静止画として最初のフレームが使われているだけなのですが、html版はブラウザーで見れば動いて見えるはずです。

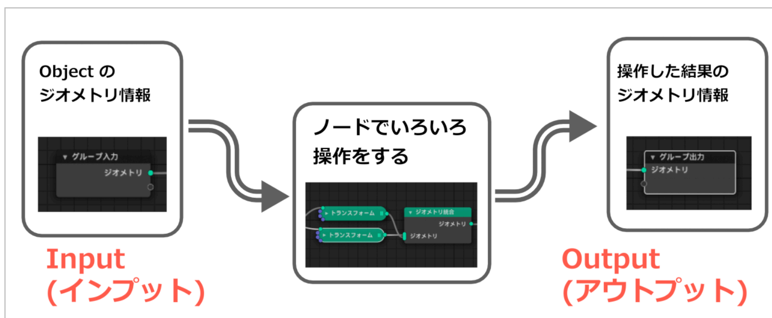
ジオメトリノードの基礎

この章は、ジオメトリノード解説の前シリーズでジオメトリノードの基本として説明した部分を、内容を縮小して手短に説明した内容です。
シミュレーションノードに手を出している以上、ジオメトリノードにはかなり慣れていると思いますから、「この辺は読み流す程度で充分」という読者がほとんどかと思います。
もう少し詳しくジオメトリノードの基礎を確認したい人は、前シリーズの解説などで確認をしてください。
（もちろんそれに限らず、世の中に出回っているどの解説でも大丈夫です）

ジオメトリノードの編集

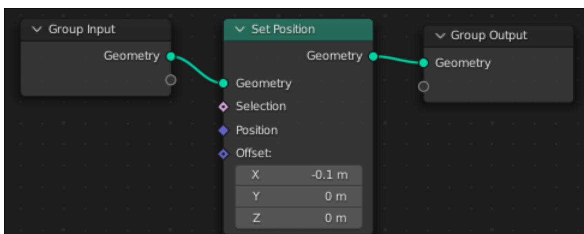


ジオメトリノードの編集は、ジオメトリノードエディタで行います。
初期状態の「何もしないジオメトリノード」がこの状態です。
Group Input(グループ入力) と Group Output(グループ出力) が緑のラインでつながっています。



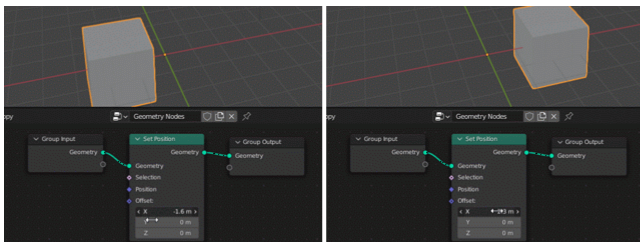
ジオメトリノードの基本形は、Group Input から Group Output の間に、様々な操作を挟み込む形になります。

Group Input の Geometry (ジオメトリ) ソケットから繋がる緑のラインは、元々のメッシュ形状の情報とその処理の流れを表します。
何もせずに直接 Input から Output に繋がれているのは「何もしないでそのまま」という状態です。
その間に、いろいろな操作を挟むことで、様々な編集が行えます。



もっとも簡単なノード編集として、Add - Geometry - Write(書込) - Set Position(位置設定)を使って見ましょう。

その名の通り、各頂点の位置を変更することができるノードです。
Position(位置) の場合は、各頂点の位置を直接指定しますし、
Offset(オフセット)を使えば、相対的に位置をずらすような使い方になります。
図のように x に -0.1 を指定すれば、すべての頂点が (-0.1, 0, 0) だけ平行移動します。

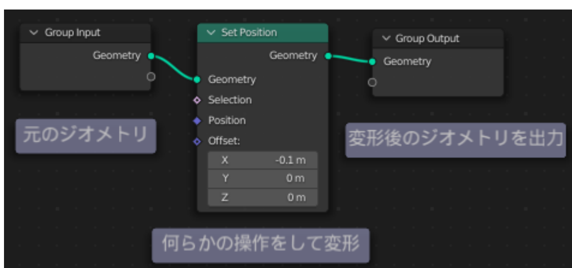


Offset の値を変更すれば、このようになります。

オブジェクト自体が動いているように見えますが、よく見ればオブジェクトの原点の位置は動いていません。
ジオメトリノードはオブジェクトの位置には影響を与えず、メッシュが変形していることが分かります。
動画)SetPosition01.gif(pdfでは先頭のコマのみ表示されています)

データの流れ(Geometry Node Fields)

Group Input の Geometry から、Group Output の Geometry まで、基本的に緑の線を左から右につないでいきます
この流れに沿って、間で様々な操作をするというのが基本です。



単純に頂点の位置を決まった量だけ動かす、ような場合は簡単です。
もっと複雑な指示をしたい場合はどうなるでしょう。

もう少し複雑に、「法線(ノーマル)方向に動かす」（つまり膨らませる）という操作をするノードを作ってみます。

Add - Geometry - Read(読込) - Normal(ノーマル) で法線情報を得るノードを配置します。
このノードを Offse(オフセット)につなげば頂点が法線方向に動きます。つまり膨らみます。
ところで、法線は長さが1と決まっているので
Add - Vector - Vector Math(ベクトル演算) で乗算をして好きな量だけ移動できるようにします。

素直にベクトルの掛け算ノードを使うと、数字が3つあって制御が面倒なので、Mutiply(乗算)ではなくて Scale (スケール)を使いました。
乗算でも Value(値)ノードから接続するなどで同じことはできます。
動画)Field01.gif (pdfでは先頭のコマのみ表示されています)

この時も、法線ベクトルのスケールを変更したものが Offset につながっている…と、左から右に情報が流れているように感じます。

ただ、処理の流れとしては本当のところは、Offset のソケットから上流にさかのぼって探っています。
Set Position(位置設定) ノードのところまで緑の線（ジオメトリ）の操作が進んだところで、
0. Set Position ノードで Offset のソケットに何かつながっている。
1. つながっているのはベクトルをスケールしたものだ。スケールと元のベクトルは何だ？
2. スケールは0.3だ。
3. ベクトルをたどると 法線(Normal)だ。

というように「実際にどれだけ動かせばよいのかな？」と、Offset の入力のソケットから逆にたどっていき、その計算結果を使う、という流れになります。

この時、逆に進んで辿りつくいた Normal ノードは「法線方向」を示すわけなのですが、
実際のデータ、すなわち頂点0の法線、頂点1の法線…というようなデータではなくて、
「法線を使う」、という情報だけがやり取りされています。

（公式サイトの Fileds に関する解説記事では、Callback という用語で説明がされています。
プログラマーであれば、コールバック関数という例えがしっくりくるかもしれません。
使うジオメトリの、「法線情報を得るという関数」が渡されている理解になります）

少しわかりづらい考え方なので、例を見てみます。
複数のメッシュを１つに合成する Join Geometry を使ってみましょう。

ノードを使って Icosphere を原点に配置します。
重なってしまうので、デフォルトキューブは少し動かして脇にどけておきましょう。

Group Input から繋がるデフォルトキューブと、Icosphere のどちらにも SetPosition を使って編集をします。
そして、Join Geometry で２つの形状を合成をします。

この時、編集に使う「法線を Scale で定数倍したもの」
という部分を、キューブ と Icosphere で共通にしておくことができます。

キューブと Ico球 それぞれで法線方向に膨らんでいます。
これは、Normal ノードが法線のデータそのものではなく「法線を使う」という情報を表すためです。
（法線が実際に評価されるのは、緑のライン(ジオメトリ)がつながっている Set Position のノードです。
そこで、「キューブの法線」「Ico球の法線」がそれぞれ別々に評価されているわけですね。）

要素ごとのパラメータと、１つだけのパラメータ

いろいろなノードを使って配置をするのに慣れてくると、ノードのソケットに菱形のものと丸いものがあることに気が付きます。
違いを意識する必要がある局面は少ないのですが、ソケットの形には実は意味があります。
例えば、ジオメトリを移動することができる、Set Position と Transform Geometry は機能の似ているノードです。

Set Position についているソケットは菱形をしているのに対し、Transform ノードには丸いソケットがついています。
これは「ただ一つの値」（もしくはベクトルなどのデータ）をとることができることを示しています。

Random Value(ランダム値)のノードなどは、頂点/辺/面といった要素に、ばらばらのランダムの値を与えます。
こうした要素ごとの値を表すのが菱形のソケットです。

ランダムのノードを、Transform(トランスフーム)のソケットにつなごうとするとエラーで赤く表示されます。
「実際に使うのは、どのランダムの値？」という情報が無いので、判断できずにエラーになるという理屈です。

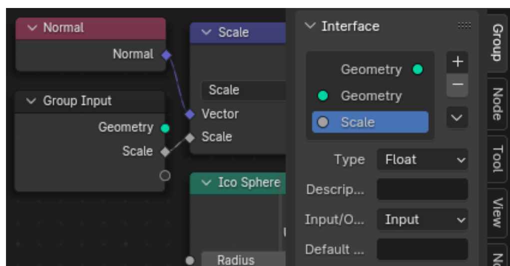
菱形のソケットは、各要素ごとにバラバラの値を受け取れますが、単一の値でも受け取ることが出来ます。
菱形の中に黒いポチがあるソケットは、単一の値を受け取っている状態になっています。
上の図では Random Value の Min や Max のソケットや、Set Position の Offset ソケットには黒ポチがあります。
これは、(0,0,0)という単一の値を使ってることを示しています。

本来は、頂点ごとに違う最小値や最大値やOffsetを渡すことも可能で、そうした場合にはソケットは黒ポチの無い菱形になります。
バラバラの値をやり取りしているのか、単一の値をやり取りしているのかは、ソケット同士をつないだ線が点線なのか実線なのかでも判断をすることができます。

こうしたソケットの種類の違いを意識しないといけない場面、というのは「あまり」ないのですが、
基礎知識として頭の片隅に置いておきたい区別の仕方です。

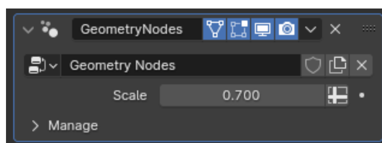
アトリビュートの入出力

今までの例だと、膨らませ方のパラメーターなどは、ノードの中で直接数値を編集していました。
それでは、「パラメータだけちょっと違う変形をする」ような場合などでもバリエーションの数だけノードを用意しないといけません。
そういう場合に、パラメータだけ変更する方法も用意されています。



もともと Group Input/Output のノードには、Geometry のソケットがそれぞれ１つだけあります。
画面右側のInterface(インターフェイス) パネルを見ると、それぞれに対応した２つのソケットが表示されています。

Group Input のノードの空のソケットにラインを繋ぐか、パネルの(+)ボタンメニューから、入力用のソケットを増やすことができます。
Scale のソケットからラインをつなぐと、同名(Scale)という入力ソケットが追加されました。

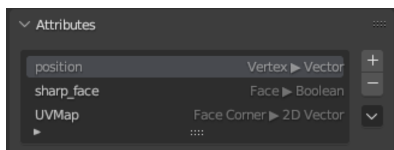


Group Input のソケットを増やすと、モディファイアのパネルにも対応する入力欄が現れます。
ここで、数値を入力することで、パラメータ違いのエフェクトなどを作ることができるわけです。

（つまり、汎用性を持たせるようにノードを作っておけば、
いろいろな場面で使いまわすことができるようになり、便利なわけです）

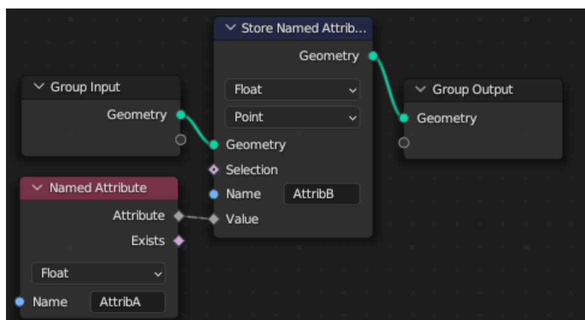
同様にして、Group Output のソケットを増やして、ジオメトリノードからアトリビュートの値を書き換えたり、新規に追加することができます。
そうして変更したアトリビュートは、シェーダーなどで利用することができるので、ジオメトリノードから間接的にマテリアルを制御することも可能なことになります。

単一の数値だけではなく、元のメッシュが持っているアトリビュート(属性)を入力にすることができます。
入力欄の右側にある十字のアイコン(スプレッドシートのアイコン)をクリックすると、文字入力が行えるようになり、名前で頂点ウェイトや、その他のアトリビュートを選ぶことができます。



メッシュが持つアトリビュートの一覧は、Attributes(属性)パネルから確認をすることが出来ます。
以前は UVMap やエッジのシャープやクリースの情報などメッシュが持つ情報は、内部で別々の扱いをしていてジオメトリノードで扱うことが出来ませんでした。
Blender 3.x や 4.x とバージョンが進むにつれて、汎用のアトリビュートとして統一的な扱いができるように内部の更新が進行中です。

Group Input や Group Output のノードにソケット追加し、モディファイアのパネルを使ってパラメーターの入出力を行う他に
Named Attribute(名前付き属性) と Store Named Attribute(名前付き属性収容) ノードを使う方法があります。

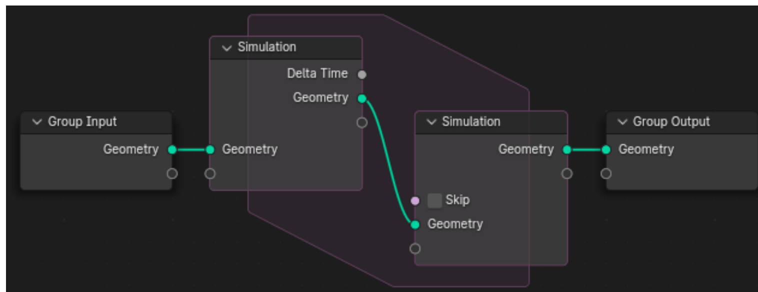


この例は、恐らくもっとも簡単なアトリビュートの読み書きの例です。
AttribA という名前のアトリビュートの内容を Named Attribute ノードで読み出し、
Store Named Attribute ノードを使って AttribB という名前で書き込みをしています。
つまり、アトリビュートの単純コピーをしています。
（あらかじめ AttribB が用意されていない場合は、新規に作成されます）

この Named Attribute は、シミュレーションを実行するときにも便利に使用することができ、頻出する機能ですので是非使い方を覚えてしまいましょう。

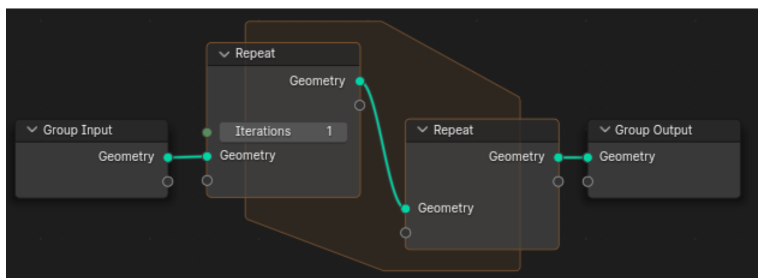
シミュレーションノード、リピートゾーン

ジオメトリノードは、Group Input と Group Output の間に複数のノードをはさんで、入力された Geometry に様々な処理を行って 出力をするというのが基本形になります。



Blender 3.6 で導入されたシミュレーション機能を使うには、
Group Input と Group Output の間にさらに、Simulation の入出力用の2つのノードで挟まれた領域(シミュレーションゾーン)を作ります。

Simulation Zone はノード追加メニューの Simulation のカテゴリーにあります。



Blender 4.0 で導入された Repeat Zone も、見た目からしてシミュレーションに似ています。
入出力の２つのノードで挟まれた領域(リピートゾーン)を作ります。

Repeat Zone はノード追加メニューの Utilities のカテゴリーにあります。

この領域の中に様々なノードを挟むことで色々なシミュレーションや繰り返しの処理を実行することができる仕組みです。
次の章から実際にジオメトリノードを使った作例を見てみましょう。

ジオメトリノードシミュレーション基礎編

シミュレーションとは

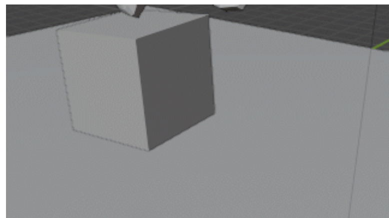
シミュレーションという言葉には、いくつか使われ方があります。
手元にあった英和辞典などで見てみると、語源的には「似ている」という言葉から始まって、「何かのふり」をする「擬態する」というような意味があるそうです。
サッカーで痛がっているふりをして、相手の反則をアピールすることを「シミュレーション」と呼びますが、これはどうやらこの語源に近い意味合いでしょう。
今の世の中で（理系には）一般的な使われ方でのシミュレーションには「訓練や実験のための模擬」という意味あいになるようです。
「物理現象のシミュレーション」にはこちらの意味あいがしっくりきます。

Blender の機能にも、剛体シミュレーション、流体シミュレーション、ソフトボディシミュレーションといった機能があります。
物理現象のシミュレーション、特に力学の多くのシミュレーションには「物事の時間進化を追う」という共通点があります。
※例外としては、力のつり合いを計算して、物事が静止して落ち着く平衡状態を計算する静力学のような計算もありますが、ここでは物事の動的なシミュレーションを考えましょう。

ジオメトリノードのシミュレーションは、ノードを使って「物事の時間進化を追う」ための計算を行うための機能です。
物事の時間進化を追うためには「今の状態」をもとに「未来の状態」を予測する必要があります。
Blender の場合は時間の進み方はフレーム単位ですから、

frame 1 の状態をもとに frame 2 の状態を計算する
frame 2 の状態をもとに frame 3 の状態を計算する
frame n の状態をもとに frame n+1 の状態を…以下繰り返し…

という処理をすることができれば、それがシミュレーションになります。



例えばBlenderの標準機能である「剛体シミュレーション」であれば、
あるフレーム(n)でのオブジェクトの位置や速度、を記憶して、
次のフレーム(n+1)でどのように動いたり、衝突して跳ね返るべきか、
ということを計算することで、このように複雑な動きが可能になっているわけです。
動画)RigidSim01.gif(pdfでは先頭のコマのみ表示されています)

このようなシミュレーションをしてみるとわかりますが、次のフレームを順番に計算していくので、フレームを飛ばして、いきなり未来へ進むようなことはできません。
順番に時間を進めることで、物理現象を再現することができます。

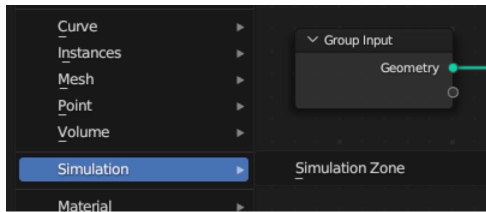
Blender 3.6 より前のジオメトリノードにはこのための機能、つまり、あるフレームの情報を次のフレームで使うための機能がありませんでした。
※トリッキーな工夫をすれば、無理矢理に行うことはできたようですが…そうした極端な例は外して考えましょう。

「シミュレーションであるかのように見える」エフェクトを作るような例はありましたが、
あくまで、Scene Time などの時間の情報や、キーフレームで設定したアニメーションをもとにして
「毎フレーム毎フレームごとに独立して」状態を計算して表示をするような仕組みになります。

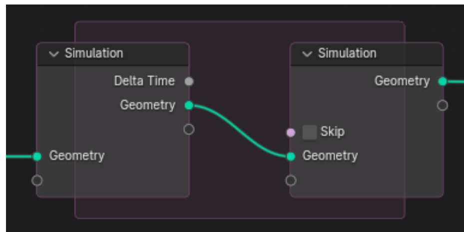
シミュレーションノードの導入により、
「あるフレーム(n)で計算した結果のジオメトリやアトリビュートを保持して」
「次のフレーム(n+1)でその結果を利用して、新しいジオメトリを計算する」
ということができるようになりました。
これによって、動的なシミュレーションが可能になります。

シミュレーション用のノードの組み方

まずは、とても単純なシミュレーションノードを作成して、基本的な機能を確認してみましょう。



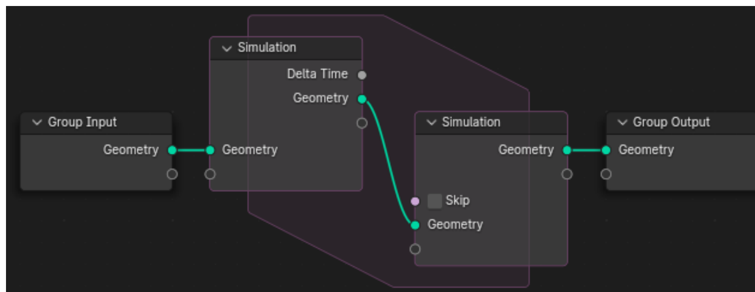
シミュレーション用のノードは、
Add - Simulation(シミュレーション) - Simulation Zone(シミュレーションゾーン)
で追加をすることができます。



シミュレーションの基本機能は、１つのノードではなく、
入力用と出力用でペアになった Simulation ノードで機能します。

このペアの間にはさまれた領域が Simulation Zone で、
ここの間に配置したノード群が、シミュレーションの計算の本体部分を担うことになります。
※Blender 4.0 までは Simulation Input, Simulation Output と表記されていましたが、
Blender 4.1 以降では Input/Output が省略されて表示がすっきりしています。

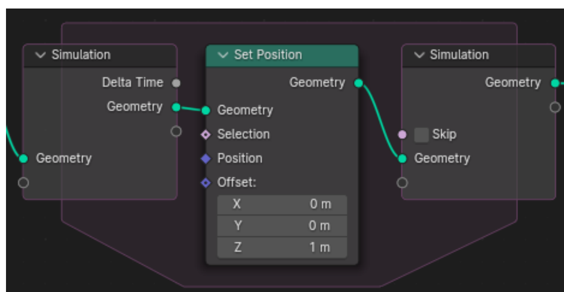
ジオメトリノードのノード構成の基本は、Group Input と Group Output の間にはさまれるようにノード群を配置することでした。
そこで、Group Input と Group Output の間に Simulation Zone を配置してみます。
この状態が「何もしない」シミュレーションを実行するノード組みになります。



このジオメトリノードを組んでアニメーション再生をしても、何も起きません。
Group Input ノードで入力されたデフォルトキューブの情報は、そのまま Group Output で出力されます。

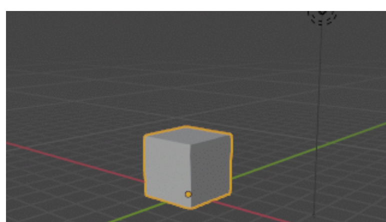
通常のジオメトリノードとの違いは、デフォルトキューブの情報は Simulation Output のノードに到達した時点で一時的なメモリにも保存されるというところです。

アニメーションでBlenderのシーンの時間が進み、次のフレームになったときに、
Simulation Input のノードの働きによって、Group Input ノードから入力されたジオメトリ情報ではなく、代わりに前のフレームでメモリに保存された情報が利用されることになります。
この「前の時間での情報」を利用して計算を行うことで、時間進化する系を表現することが出来る仕組みです。



もっとも単純なシミュレーションとして、毎フレームごとに上方向に 1 だけ頂点を動かすという事をしてみます。
Set Position ノードを使い、各頂点を移動しました。
シミュレーションではない普通の場合は 1 だけ頂点が動いてそれで計算は終了です。

シミュレーションゾーンの中で Set Position でメッシュを動かすと、メッシュの状態が Simulation Output に保存されます。
時間を一フレーム進めると、次のフレームではこの状態を Input として使い、
「前のフレームの状態から上方向に 1 動く」という操作が実行されます。



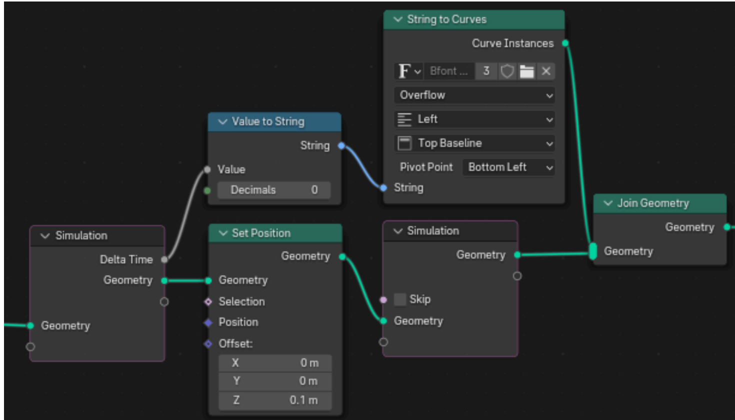
時間を進めていけば毎フレーム頂点が 1 ずつ上に移動していきます。
つまりこれで、単純な移動のシミュレーションが実現できたことになります。
動画)Sim01.gif(pdfでは先頭のコマのみ表示されています)

毎フレーム１動くと、かなり速いのでだいぶゆっくりと、カクカクした動きで表示をしました。

ただし、よく見ると最初のフレームで、デフォルトキューブが 1 だけ上に上がった状態です。
１フレーム目の Group Output ノードまで処理が進んだ時点で、既にシミュレーションゾーンで「１だけ上げる」という操作が実行されているで、そのようなことになるわけです。
これでは、Simulation Input ノードにつないだジオメトリ情報が「初期状態」になっていません。ちょっと不便です。
もう少し厳密に時間について考えるときには、Delta Time のソケットを利用することになります。

Simulation Zone と Delta Time

Delta Time は、前のフレームと今のフレームとの間に過ぎた時間の長さを得ることができるソケットです。
Delta Time のソケットの中身を見て内容を確認してみましょう。

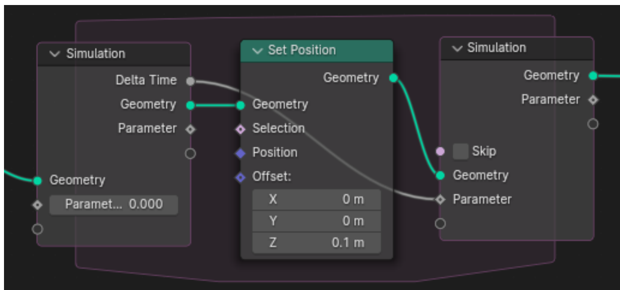


Value to String(値の文字列化)からの一連のノードを繋いで3Dビュー画面に数値を表示しようとしました。
しかし、このノードのつなぎ方は**間違い**です。

シミュレーションゾーンの中のパラメーターなどを、シミュレーションゾーンの外につないで、
（例えば今回の様に Join Geometry などで合成しようとして）混在させることはできません。
Blender 4.0 時点で別にエラーが出て実行できなくなるわけではないのですが、
シミュレーションゾーンの表示が消えてしまって、なにか問題が起きていることが分かります。

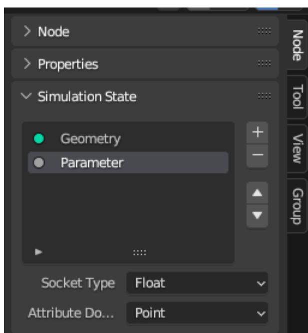
シミュレーション計算中に使うジオメトリや値などの情報、つまり Simulation の入力ノードからつながっているノードの流れは、Simulation の出力ノードを通さないと、他のノードの流れと混ぜることはできません。
今回は、入力ノードの Delta Time のソケットから伸びた(出力ノードを通っていない)ノードの流れと、Geometry のソケットから伸びた(出力ノードを通った)ノードの流れを Join Geometry で一緒にしようとしたので問題が起きたことになります。

しかし、今のままでは Simulation の出力ノードには Geometry のソケットしか無いので、ジオメトリ情報しかやりとりができないことになっています。



Simulation の入出力ノードにソケットを増やすことが出来ます。
左の図では、デフォルトのソケットに加えて Parameter というソケットが増えています。
本来は、シミュレーション中に利用するパラメーターを追加するための機能ですが、
今回は単純に Delta Time のソケットをつないで、値をシミュレーションゾーンの外に出す為に使います。

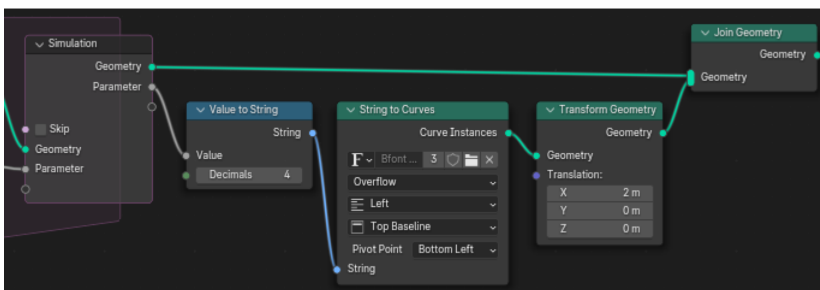
このソケットに、Delta Time の情報を繋ぐことで、
安全にシミュレーションゾーン内部の情報をシミュレーションゾーンの外に渡すことができます。



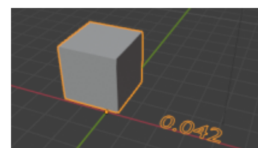
ソケットを増やすためにの操作として、空きのソケットに直接ノードを接続するか、
Simulation State(シミュレーション状態) のパネルで
「+」ボタンを押して直接ソケットを増やす操作をすることが出来ます。

この Simulation State のパネルは、Simulation Input か Simulation Output のノードを選択している時に表示されるパネルです。

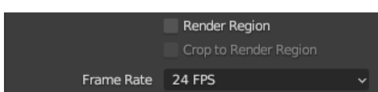
こうして無事に数字情報を取り出せましたが、そのままでは数字の中身がユーザーからは見えません。
理屈で数字の中身が分かっている時は良いのですが、デバッグをしたいようなときには数字を表示したくなります。
そうした場合は名前付きアトリビュートに書きこんでスプレッドシートの機能で見るか、もしくは
Utilities - Text - Valute to String(値の文字列化) ノードと、String to Curves(文字列のカーブ) ノードの組み合わせで、値を3Dビューに表示するという方法が使われます。



シミュレーションゾーンの外に、Delta Time の情報を取り出し、
Value to String ノードからの一連のノードで3Dビューに安全に表示をすることが出来ます。



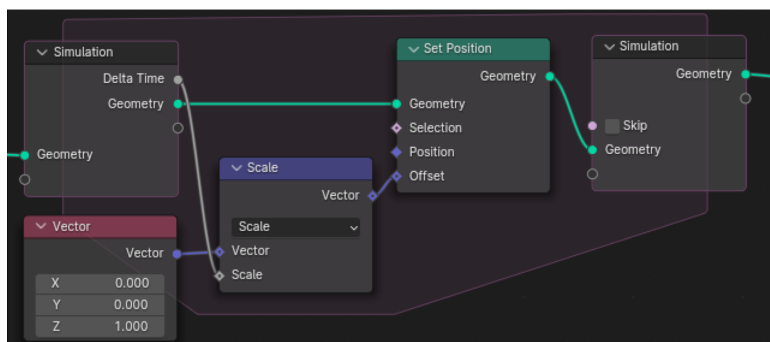
すると、最初のフレームでは0、それ以降のフレームでは0.042になっていることが分かります。
桁数を増やしてより厳密に表示すると、0.041666... となります。



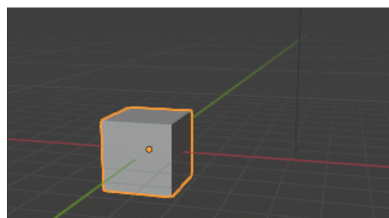
この値は、Blender のデフォルトでの fps 設定24、
つまり一秒間に24コマのアニメーションになっている時の、一コマあたりの秒数(1/24)です。

この Delta Time によって１フレーム当たりの時間変化の量がわかります。
最初のフレームでは 0 なので、これを指定した移動量に掛け算をすれば「最初のフレームでは移動をしない」ことになります。
つまりSimulation Inuput に入力した初期ジオメトリがそのままフレーム0の状態になるわけです。

また「（スローモーションなどを表現したいなどの理由で）フレームレートを変えてもそれに対応したシミュレーションの速度になる」
ということになり、時間に関する調整がしやすくなります。



つまり、Delta Time の情報を使ってこのようにノードを組むのが、
正しい時間進化の処理になるわけです。

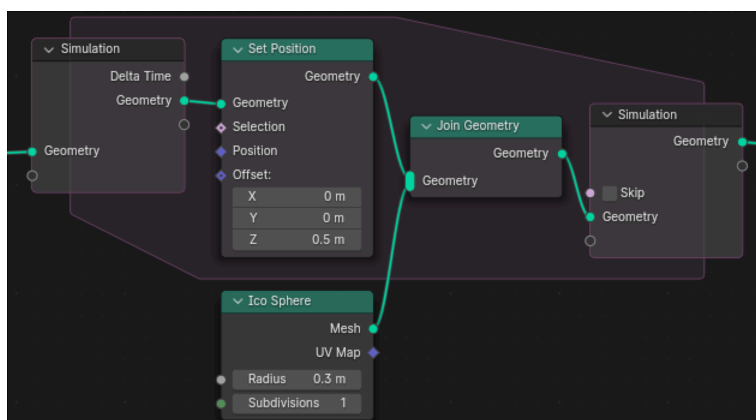


これで、24fpsの時には、24コマかけて、つまり1秒に1動くデフォルトキューブになりました。
動画)Sim01B.gif(pdfでは先頭のコマのみ表示されています)

ただし、正確にはなったのですが、実際のところ掛け算の分ノードが複雑になっていますし、
(0,0,1)として入力した速度は**1秒あたりの速度**になります。
つまり毎フレーム(0,0,1)動くのではなく、その1/24だけ動くようになっているので、
コマアニメーションとしての直感的な分かりやすさは少し犠牲になっています。

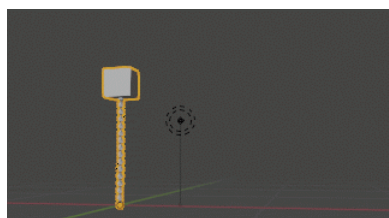
Delta Time を使うのも善し悪しで、シンプルに「毎フレームこれだけ処理をする」程度の厳密さで良ければ、Delta Time を無視するのも一つの方向性です。
この本で収録している作例では、Delta Time を使った場合、使わなかった場合などが混在している点に注意してください。
ここで使ったノード組みの幾つかは、01_BASIC/01_BasicSimulation.blendとして同封しました。

メッシュの追加



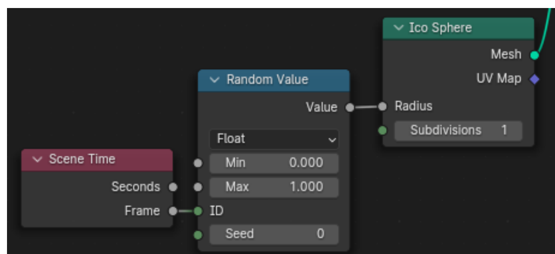
次に、頂点を動かすだけではなくて Join Geometry を使って球を作成してみます。
図で見ると、Ico Sphere のノード自体はシミュレーションの枠から外れています。
厳密にはJoin Geometry だけがシミュレーションとして実行されていることになります。
（解釈が違うだけで行うことは同じですが…）

このシミュレーションの枠内に設定されるノードは
「Simulation Input のソケットから直接つながるノード」
ということで区別されるようです。



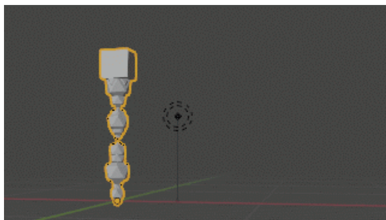
シミュレーションの仕組みを考えれば、毎フレーム原点に球が発生し、それが Set Position の影響で上に上昇していくはずです。
実際に時間を進めてみると…球が毎フレーム発生しています！
メッシュが生成されるようなシミュレーションも問題なく実行できることがわかります。
動画)Sim02.gif(pdfでは先頭のコマのみ表示されています)

上の動画だと、まるでキューブの軌跡が残っているように見えますが、本当は球はキューブと一緒に上に動いて行っているだけですね。



試しに発生する球のサイズをランダムにしてみます。

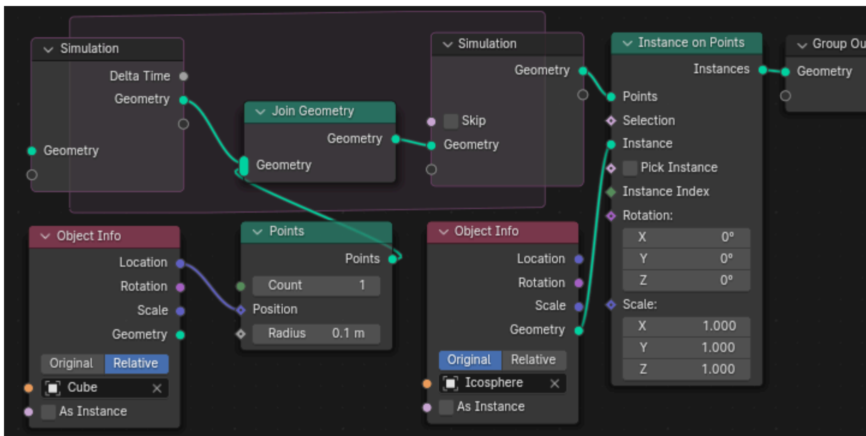
Icosphere の半径は丸ソケット（値を１つだけ受け付ける）なので、ランダムのIDにもシーンで１つの値を渡します。
このような用途には、Input - Scene - Scene Time を使うのが適しています。
（毎フレーム違うIDを与えるので、毎フレーム違うランダムになるわけです）



毎フレームランダムなサイズの球が発生して、キューブと一緒に上昇していることが分かります。
動画)Sim03.gif(pdfでは先頭のコマのみ表示されています)

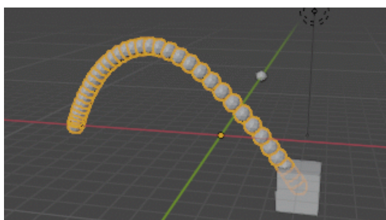
パーティクル生成のような効果

先ほどの例は軌跡のように見えて軌跡ではない効果だったので、実際にキューブの軌跡を残すようなシミュレーションを組んでみます。

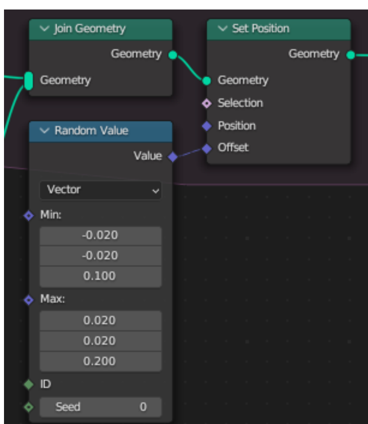


動かすキューブの他に軌跡用のオブジェクトを作り、そちらをジオメトリノードで管理するようにしましょう。
動くキューブキューブは制御用のオブジェクトということになります。
適当に空間を動かすアニメーションを設定します

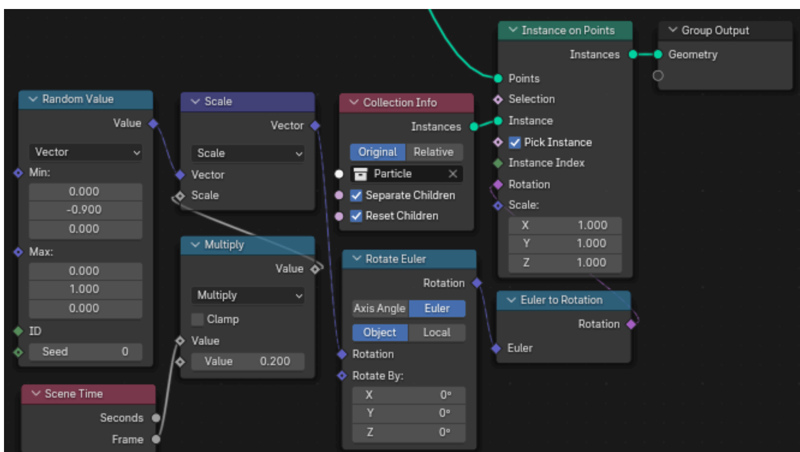
Cube の(相対的な)位置にポイントを１つ作成し、毎フレーム Join Geometry を使って追加をします。
これで、キューブの軌跡上にポイントが配置されていくはずですが、それだけでは表示が寂しいので、
Icosphere を別に作り、インスタンスを配置するようにしました。



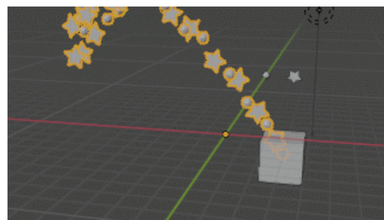
キューブの運動に沿ってポイントが発生する、ある意味単純なパーティクルのような効果が作成できます。
発生した点がその場を動かなければ、軌跡を表示するようなエフェクトになっているわけですね。
動画)Trajectory01.gif(pdfでは一部のコマのみ表示されています)



発生した点を毎フレームで動かしていけばよりパーティクルっぽいエフェクトになります。
全粒子が同じだけ動くとただの平行移動になってしまうので、ランダムを使って移動速度をばらけさせます。



点の位置にインスタンスを配置する部分も変えてみます。
Collection Info を使って複数のオブジェクトのバリエーションを付けてみます。
オブジェクトごとにランダムな回転を与えます。
回転に対してシーンの時間情報をかけ合わせれば、くるくる回るパーティクルになるわけです。

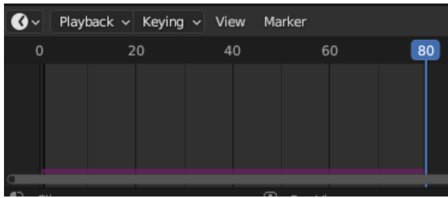


動画)Trajectory02.gif(pdfでは一部のコマのみ表示されています)

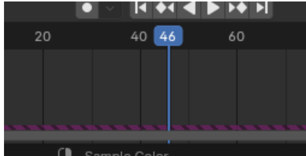
この例は、01_BASIC/02_Trajectory.blend として同封しました。

キャッシュとシミュレーションの再計算

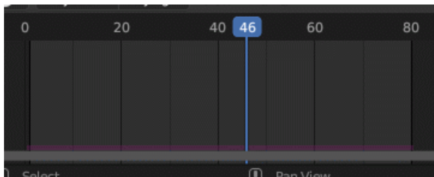
シミュレーションノードで計算した結果は、キャッシュとしてメモリ上に保持されます。



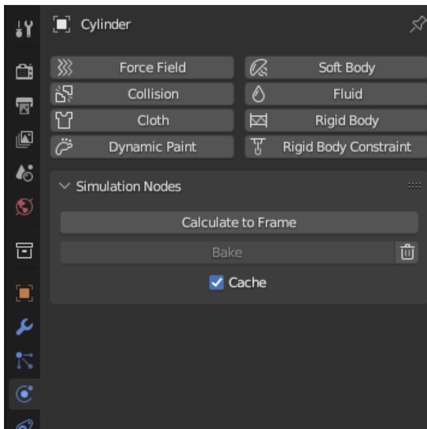
時間を進めると、計算が済んだフレームはタイムライン上に紫色でマークされます。
この部分にはメモリ上にデータが残っているので、
時間を進めたり戻したりが（再計算しなくて良いので）高速に行えます。
サンプルのような例だと計算は一瞬なので有難味はないですが、
重いシミュレーションの時などはキャッシュの有無で操作性が違うはずです。



ジオメトリノードのノード構成やパラメーターに変化があると、
「このキャッシュは最新のものではない」ということが分かるように、
色がやや暗くなり縞模様になります。



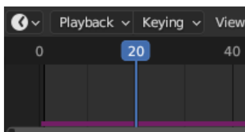
この状態でフレームをシーンの先頭に持って行くとキャッシュがクリアされ、
再生すれば新しいシミュレーションとして再計算されて、キャッシュに情報が溜まっていきます。



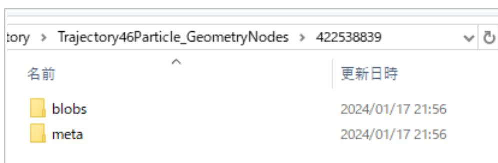
ジオメトリノードのキャッシュ情報の操作用のパネルも、他のシミュレーションと同じように、
プロパティパネルの中のシミュレーションのタブに表示されます。

全フレームのベイクとベイク/キャッシュデータの破棄、
現在のフレームまでの計算、そしてキャッシュ機能自体の有効/無効 が設定できます。

※キャッシュ機能を「わざわざ使わない」ようにする局面は少ないと思いますが、一応挙動を確認してみましょう。
キャッシュを使わない場合は、タイムラインで時間を飛ばしたりすると、飛ばした時間をスキップしてシミュレーションが進みます。
また、この場合は時間を戻すような操作をすると、初期状態に戻るような挙動になります。



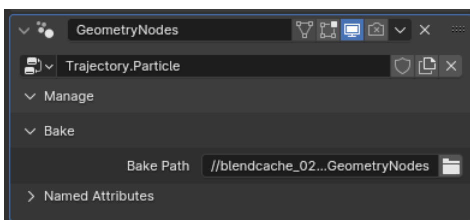
ベイクをすると、タイムライン上のマークの色がさらに一段明るくなります。
ベイクされたキャッシュデータは、メモリ上ではなくプロジェクトファイルのあるフォルダにファイルとして保存されます。



プロジェクトのファイル名や使ったオブジェクト名に応じたフォルダが作成され、
その中に毎フレームごとのジオメトリや、その他必要なメタデータなどが連番ファイルで保存されます。

重いシミュレーションなどであれば、必要に応じてベイクしたり削除したりしてください。
（この辺はパーティクルや流体などで重いシミュレーションをするときと同じ感覚です）

Blender 3.6 では bdata というフォルダ名、拡張子だったのですが、Blender 4.1 時点で blobs というフォルダ名、.blob という拡張子に変わっています。

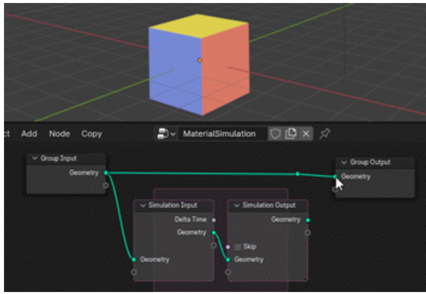


ベイクの保存先フォルダは、Manage(管理) - Bake(ベイク) の項目で確認することができ、
もし必要があれば、フォルダの変更（例えば一時利用のためのRAMディスク上のフォルダにする）等の操作ができます。

指定がなければ、プロジェクトファイル(.blend)と同じ階層にblendcache_filename というような形でフォルダが作られ、そこが保存先になります。

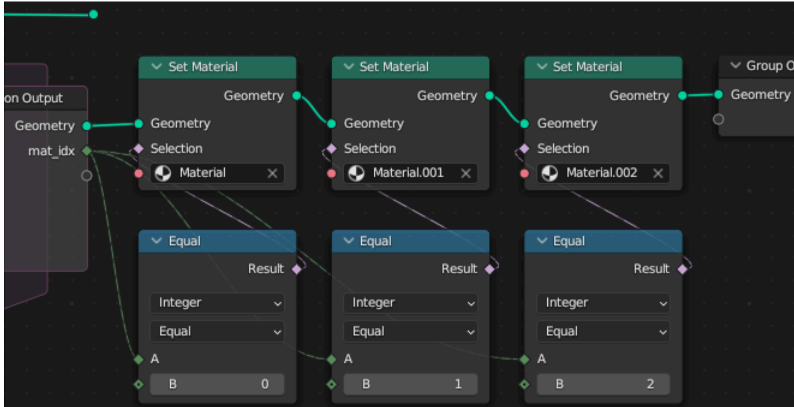
マテリアルとシミュレーション

Blender 4.0 時点で、シミュレーションノードが持つ明確な弱点が、マテリアル情報の扱いでした。
内部的なデータ構造がシミュレーションノードの実装と相性が悪いという理由で、シミュレーションを実行するとマテリアル情報が外れる仕様だったのです。

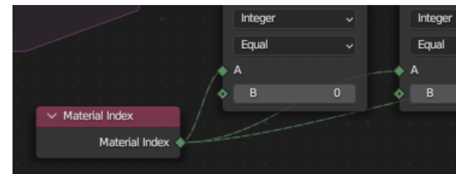


何もしないシミュレーションでも、シミュレーションゾーンを経由したとたんに、
マテリアル情報が外れてデフォルトの白色マテリアル表示になってしまいました。
動画)SetMaterial01.gif(pdfでは先頭のコマのみ表示されています)

このことは、当然改善すべき点として挙がっていて、内部的な改造が行われていたようです。
Blender 4.1 になってこの問題は解決されました。

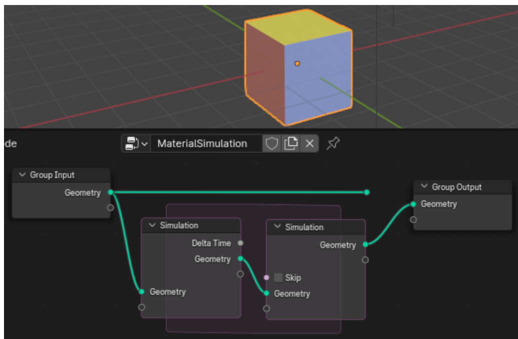


Blender 4.0 までは マテリアルの管理を手動で行わなければなりませんでした。
この図は Blender 4.0 までの場合のノード構成の参考です。
シミュレーションゾーンを通過した後で、パラメーターに従ってマテリアルを設定しています。



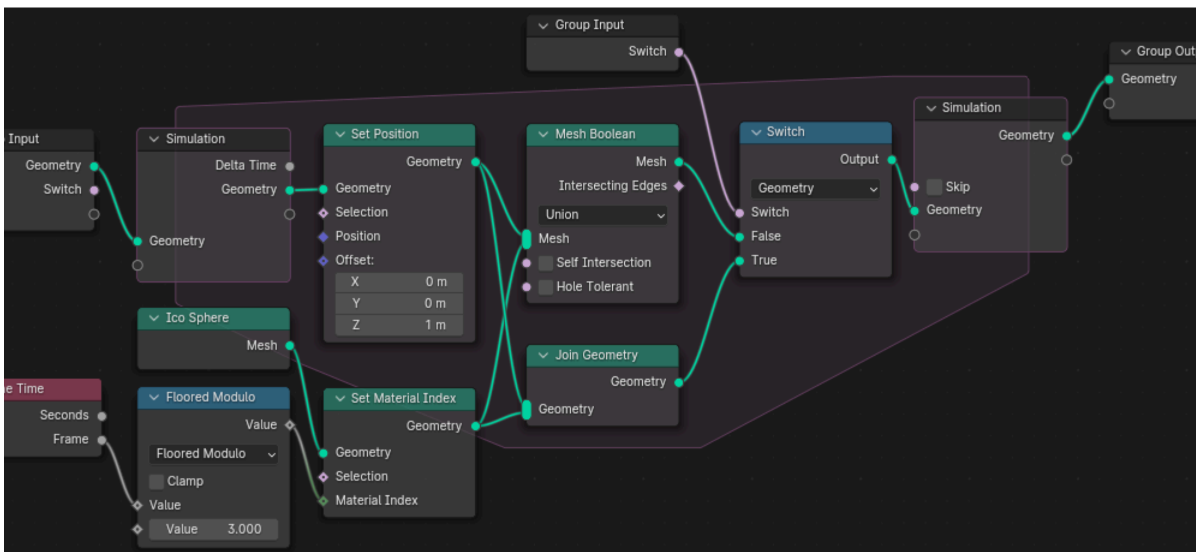
この時、Material Index を直接読んでマテリアルをセットしようとすると、おかしなことになるはずです。

Set Material の操作自体が、Material Index を書き換えてしまうので、次のSet Material で破綻が起きることがあるのです。
そのため、初期状態のマテリアルインデックスの情報を保持しておかないといけません。
Simulation Zone 内の変数(匿名アトリビュート)として記憶しておくか、名前付きアトリビュートを利用するかという方法を取ることになります。

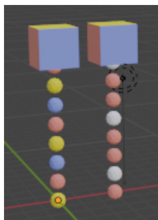


Blender 4.1 では、シミュレーションゾーンを通過しても元のマテリアル情報が維持されています。
頂点の移動など、変形をさせるシミュレーションで、複数のマテリアルの利用が行いやすくなりました。

シミュレーションゾーン内で、新たにメッシュを発生させるような場合には注意が必要です。
発生させたメッシュには、マテリアルの情報が無いので、やはり自分で管理をする必要があります。



マテリアルインデックスを変化させた Ico Sphere を追加するジオメトリノードを組んでみました。
ブーリアンの Union を使ってメッシュを追加した場合、Join Geometry を使って追加した場合の２通りを Switch で使い分けられるようにしています。



面白いことに、結果が大分違うことに気が付きます。
ブーリアンを使った場合には「インデックスに対応するマテリアルの情報が分かっている」メッシュ(Cube)が加工される、
という処理なので、インデックスに応じた色とりどりの Ico Sphere が並ぶことになりました。

Join Geometry の場合は、「インデックスに対応するマテリアルの情報がない」 Ico Sphere が追加されるという処理なので、挙動が違っています。

「マテリアル情報のないメッシュを Join した場合のマテリアルの扱い」は、本シリーズの Vol.1 の本で少し詳しく調べてあります。
マテリアルインデックスが 0 の場合とそれ以外の場合で挙動が違うので、このように 2 つおきに白い球になるような振舞いになりました。
ただし、そもそも「マテリアル情報のないメッシュ」を扱うのは設定し忘れのようなものですから、Set Material ノードを使うなどしてきちんとマテリアルを指定するべきでしょう。
この作例は、01_BASIC/03_Material.blend として同封しました。

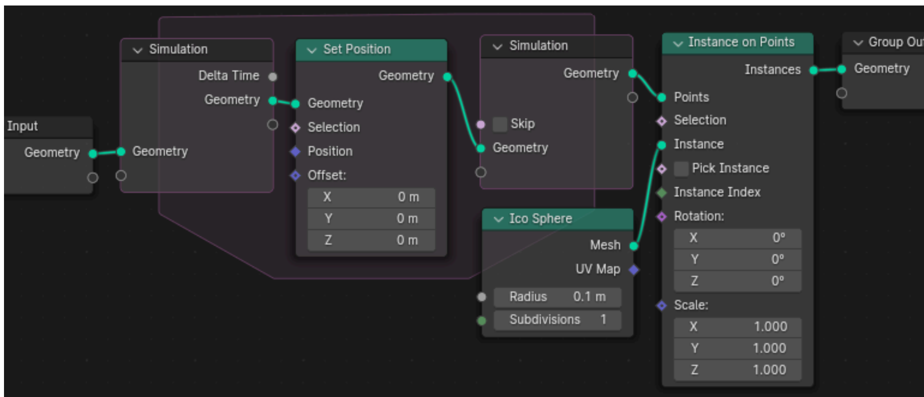
位置のシミュレーション

この先は、だんだんと複雑なシミュレーションに進んで行きます。
シミュレーションと言えば、剛体や液体といった物理のシミュレーションが思い浮かびます。
普通の（日常生活で見るような）物体は重さを持っているので、慣性の法則で動き出した物体はそのまま動き続けようとします。
そのため「現在の速度の情報」を持っていないと、次のステップでの位置が決められません。
ジオメトリノードのシミュレーションでも「速度」というパラメーターを導入しないと日常的な現象をうまく再現が出来ないことになります。

しかし、そうしたシミュレーションに進む前に、まず最初は頂点の位置だけを考えれば良い、より単純なシミュレーションから試してみましょう。

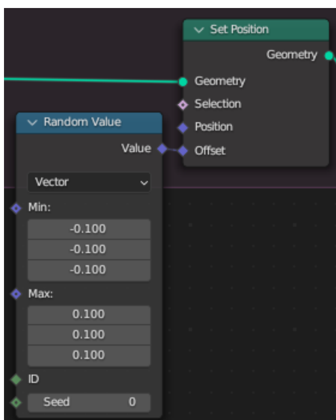
ランダムウォーク

ランダムウォークは、次のステップでの位置がランダムに移動するような動かし方です。
普通の物体には重さがあるので、速度と慣性（重いほど動かしづらく止めにくい）を考慮に入れないといけません。
しかし、液体中の微粒子のように速度情報が減衰してすぐに失われるような現象だと、慣性を考えなくとも良くなり、
ランダムウォークのようにフレーム毎の位置の変化だけを考えれば良くなります。
こうした運動としてブラウン運動などが有名ですね。
※)この説明はおそらく厳密に考えると正確な説明ではないのですが、ものすごく大雑把に言えば…

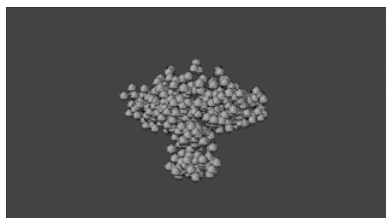


ということで、頂点が毎ステップ動くようなシミュレーションのフレームを考えます。
と言っても先ほどと同様に Set Position ノードを Simulation Input と Simulation Ouput に挟むだけですね。

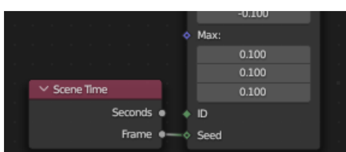
各頂点には Instance on Points(ポイントにインスタンス)を利用して球を表示するようにします。



毎ステップ頂点がランダム Offset で動くようにします。
しかし、このノードの組み方だと頂点ごとランダムに動くのですが、時間に対しては一定になります。
そのため、等速でランダム方向に動くようなことになります。

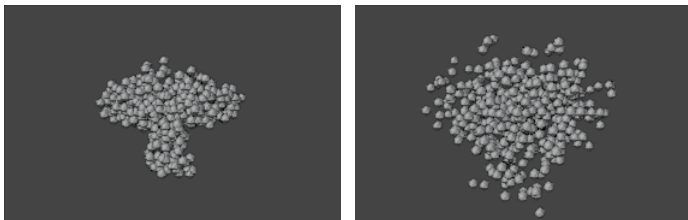


動画)Sim04.gif(pdfでは先頭のコマのみ表示されています)



毎フレームごとのランダムを作成する場合には、
Seed 値の方に毎フレーム違った値を入力します。

まぁつまり、普通は Scene Time のフレーム数を入力すれば良いわけですね。



頂点がランダムウォークするシミュレーションができました。
時間とともに、拡散してゆっくりと広がっていきます。
（実際にインクが静かな水の中で拡散して広がっていく現象に似た現象です）
動画)Sim05.gif(pdfでは先頭のコマのみ表示されています)

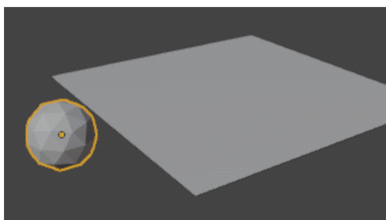
このサンプルは、01_BASIC/04_RandomWalk.blend として同封をしました。

Dynamic Paint (ダイナミックペイント)的な制御

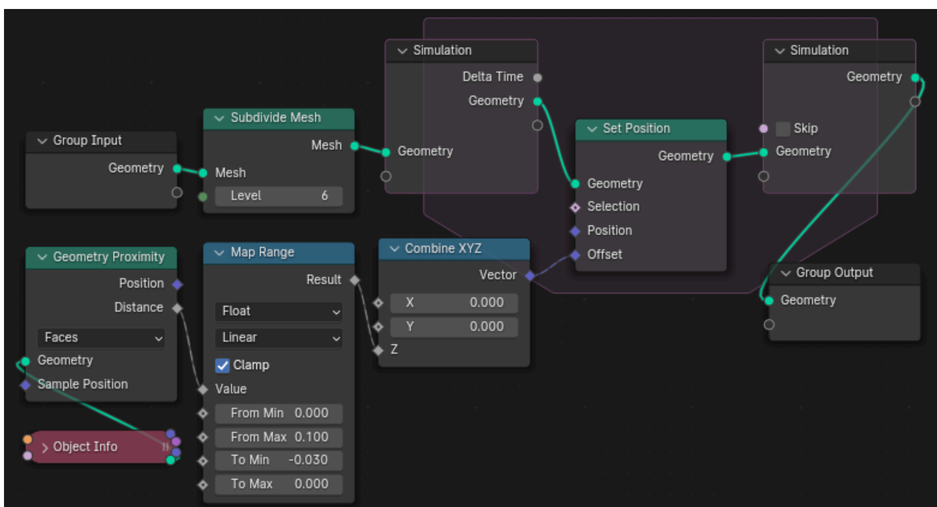
Blender の Dynamic Paint(ダイナミックペイント)は、オブジェクトの触れたところに色を付けたりくぼみをつけたりすることができる機能です。
地面をキャンバスに、靴をブラシに見立てて、キャンバスの上にブラシを置く（触る）と足跡が残る…というような効果を実現できます。

便利な機能なのですが、備え付きの機能であるために自由度はそんなに高くありません。
（正確にはそれなりの自由度はあるのですが、自在に操ろうとすると今度は設定がかなり煩雑になってしまいます）

ジオメトリノードによるシミュレーションを駆使すれば、ブラシに相当するオブジェクトが触れたという「状態が残る」わけですから、
ダイナミックペイントのような効果を自在にカスタムして実現することが出来ます。

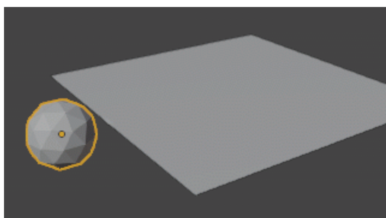


キャンバスに相当する平面と、ブラシに対応する球を作成します。
球にアニメーションを付けて、キャンバスの上を動くように設定します。
動画)DynamicPaint01.gif(pdfでは先頭のコマのみ表示されています)

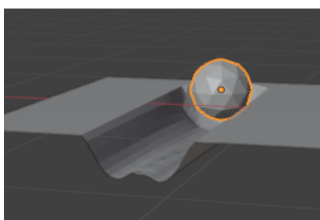


ブラシが接触した（近接した）頂点を下向きに動かすようにシミュレーションを組んでみました。
初期状態として、Subdivide Mesh(メッシュ細分化)で平面を6段階分割して
充分なメッシュの細かさになるようにしています。
※元の形状で既に細かく分割したものに、さらに分割をかけてしまうと大変なことになるので注意です…

Geometry - Sample - Geometry Proximity(ジオメトリ近接) を使ってブラシになる球との距離判定を行います。
近いところでは下方向に大きく動き、離れると移動量が 0 になるように、Map Range を使って調整します。

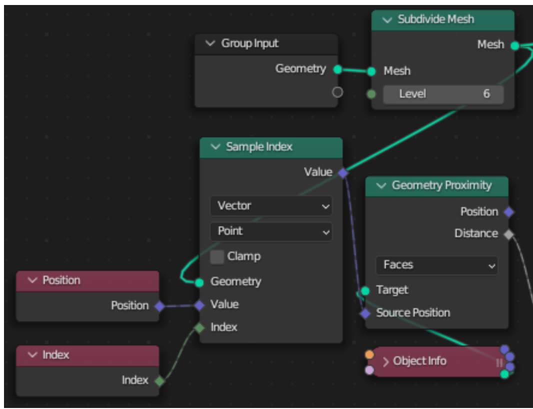


球の接触した場所に凹みが形成され跡が付きました。
動画)DynamicPaint02.gif(pdfでは先頭のコマのみ表示されています)



この凹みの深さは、球のサイズ程度までで自動的に止まっています。
これは、接触判定をしている形状が前ステップの時点での「変形後」の形状だからです。
へこめば、その分球から離れるのでそれ以上は変形しないことになるからです。

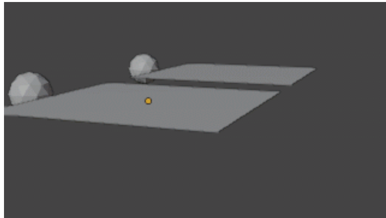
例えば「球が同じ場所に留まっていると効果が累積してどんどん溶けていく」ようなエフェクトにはなりません。



Geometry Proximity には、判定をするための位置を（実際の頂点の位置ではなく）別の場所に変える機能があります。
Source Position(ソースの位置)のソケットに、位置情報を渡せばその位置からの距離を測定します。

そこで、Sample Index(インデックスサンプル)を利用して
「変形前の初期の頂点位置」からの距離判定に変更してみます。

変形前と変形後で、頂点どうしのインデックスの並び順やメッシュの張り方などは変わらないので、
左のようなノード組みで、各頂点ごとの「変形前の位置」が分かることになります。

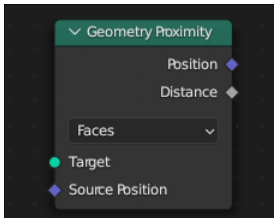


すると、球が（元）キャンバスの上で静止すると、ずっと接触をしている扱いになりますから、
その場で下方向の移動が続き、とろけているかのように変形が継続することになります。
動画)DynamicPaint03.gif(pdfでは先頭のコマのみ表示されています)

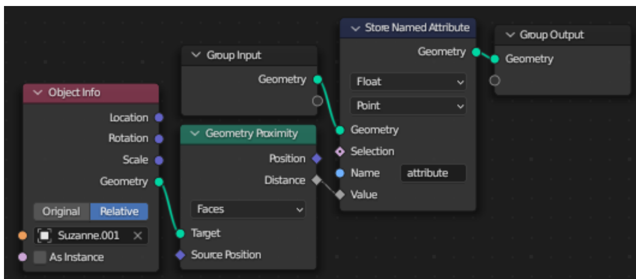
このサンプルは 01_BASICS/05_DynamicPaint.blend として同封しました。

接触判定

先ほどの Dynamic Paint での例でもそうですが、この先シミュレーションを用いてオブジェクトとオブジェクトの接触などを判定したい場合は数多く発生するでしょう。
そうした場合の判定法について幾つか確認をしておきます。

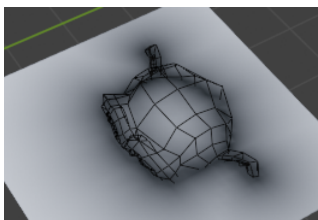


第一に使いやすいノードとしては、先ほども出てきた Geometry Proximity でしょう。
最も近い(頂点/辺/面)の位置と、そこまでの距離を得ることが出来ます。
接触判定に使うのであれば、頂点と頂点の間をすり抜けたりすると困るでしょうから、Faces(面)モードで使うのが普通だと思います。



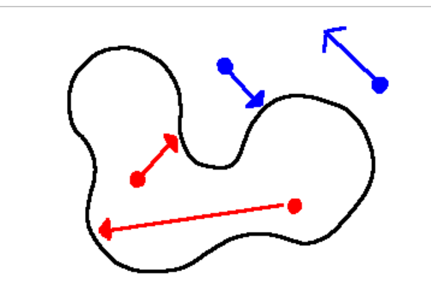
ところが、当然ではあるのですがポリゴン面までの距離が分かるだけだと、
オブジェクトの内部に居るのか、外部に居るのかという事は分かりません。

例えば、面からの距離を Store Named Attribute(名前付き属性収納)を使って保存したとします。
細かく切った平面を使って、あるオブジェクトの断面に距離に応じた色を付けてみます。



当然このようになり、スザンヌの内部と外部の区別はつかないことになります。
距離だけで衝突判定をすると、何かの拍子に面をすり抜けてオブジェクトの内部に入り込み、閉じ込められるような現象が起こりそうです。

ゲームなどで稀に起きたりする現象ですね…

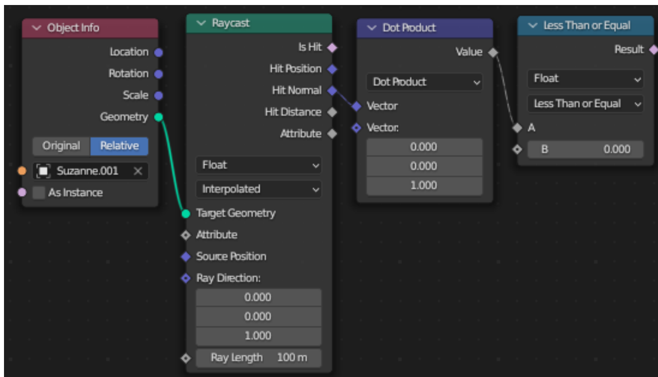


オブジェクトの内部かどうかの判定に Raycast を利用する方法があります。

きちんと閉じていて、重なったりめり込んだりしていなければ、ある点がメッシュの内部にあるのか外部にあるのか、次のように判定ができます。
(blenderに限らず、一般的なメッシュの判定アルゴリズムの話ですね)

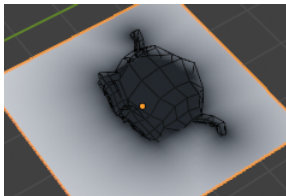
ある点から、好きな方向に光線を飛ばして、ポリゴンの裏側にぶつかれば、メッシュの内側(赤)。
ぶつからないか、ぶつかってもポリゴンの表側にぶつかったのであれば、メッシュの外側(青)。
という判定方法です。

「ポリゴンの表か裏か」をどのように判断するかは、一般的には難しいところもあるのですが、
ポリゴンの裏表が正しく組まれていれば、Raycast で飛ばした光線と法線が同じ方向を向いているか（内積が正になっているかどうか）で判断ができます。



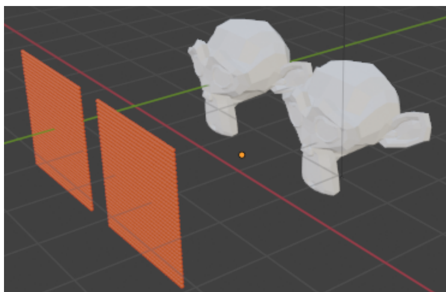
オブジェクトの内部を False に、外部を True にする判定用のノード組みです。
この判定方法は、光線を飛ばす向きはどちら向きでも構わないので、分かりやすく(0,0,1)を使いました。

ところで、内側か外側か以前に、衝突をしたかどうかは、本来は Is Hit(ヒットフラグ)で分かるのですが、
衝突しなかった場合には Hit Normal(ヒット法線)が(0,0,0)になります。
ということで、「内積が0を含めて0よりも小さい」という条件を判定するだけで、「ポリゴンは外側」という判定をしたことになります。



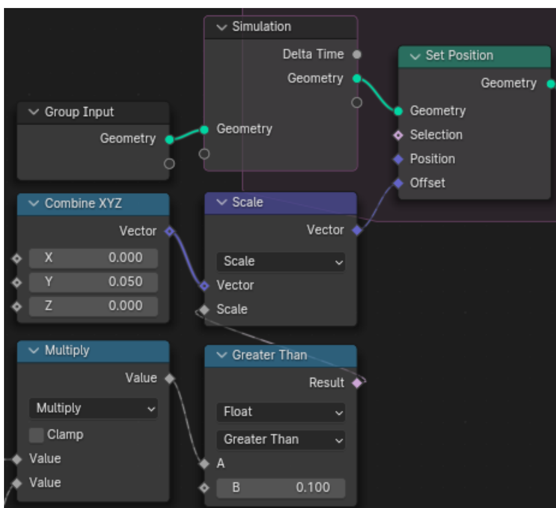
先ほどの距離情報とかけ合わせれば「内側はすべて0で、外側は表面までの距離」という情報を得ることが出来るわけです。
これで、多少粒子の移動が粗くても面をすり抜けるようなことが起こりません。

但し、メッシュはきちんと閉じている必要があります。
スザンヌは目の部分がきちんと閉じていないので、厳密にはこの判定法だと失敗する場合もあるのですが…
スザンヌ程度のメッシュの乱れであれば、大体の場合は大丈夫です。

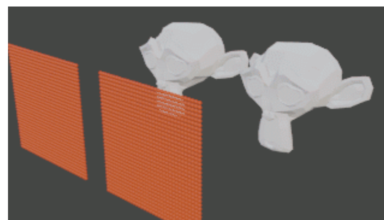


沢山の粒子を左側から飛ばし、スザンヌにぶつかったらそこで止まるというシミュレーションをしてみます。
細かく切ったメッシュ頂点に、小さな赤い球のインスタンスを配置しています。

頂点は毎フレーム移動するようにノードを組むのですが、頂点がスザンヌの内部に入り込んだら、
もしくは距離が一定の値以下になったら、そこで止まるようにします。



左下の Multiply ノードには Geometry Proximity によるポリゴン面までの距離と、
オブジェクトの内部か外部かの判定がつながっています。
ここでは、面までの距離が0.1よりも小さくなったら移動距離が0になるように ベクトルの Scale 演算を使っています。



動画)Collision01.gif(pdfでは先頭のコマのみ表示されています)

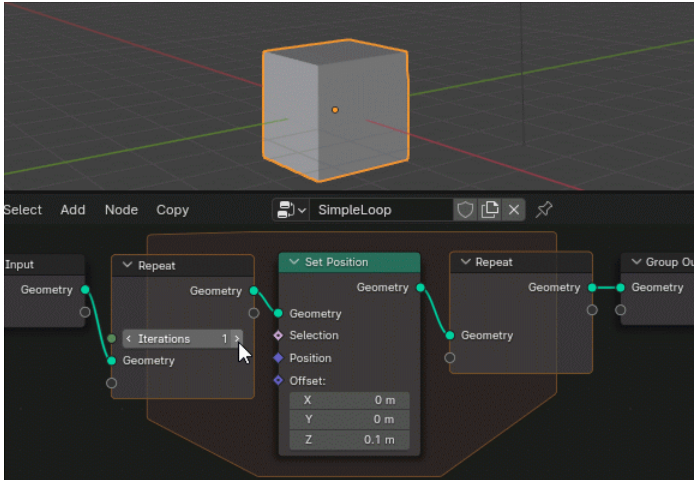
動画の左側は、Raycast による判定のみを利用して、距離のマージンが無い場合です。
これはこれで、スザンヌの表面で粒子が半分めり込んでしまいますから、あまり良くありません。

近接する面までの距離の測定と、オブジェクトの内部と外部の判定、どちらも考慮した方がより安定した衝突判定になると考えられます。
この例は、01_BASIC/06_Collision.blend として同封をしました。

Repeat Zone (リピートゾーン)

Blender 4.0 で導入された Repeat Zone は、繰り返し処理を行うためのノードの仕組みです。
繰り返し処理は、様々な場面で登場する処理で、当然シミュレーションをするときにも便利に使える場面があります。
仕組み自体も Simulation ノードとかなり似ています。ここで、次の作例に進む前に Repeat Zone についての簡単な解説をしておきましょう。

追加メニューで Utility(ユーティリティ) - Repeat Zone(リピートゾーン) を選択すると、繰り返し処理用のフレーム枠が作成されます。
Simulation Zone の後に実装された機能なので、見た目や仕組みはかなり似ています。

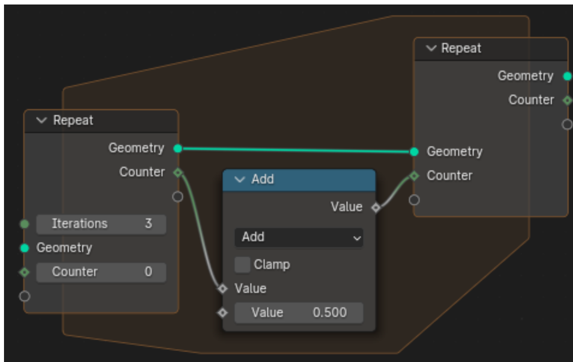


よく見ると、フレームの枠がややオレンジ色っぽい見た目の違いがあります。

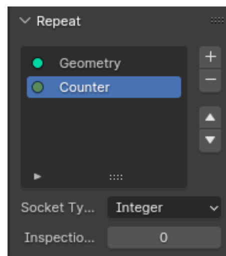
Group Input と Group Output の中に Repeat Zone を挟み込みます。
Repeat の出力ノードまで届いた Geometry は、Repeat の入力ノードまで戻されて次の処理を行うという形で繰り返し処理が行われます。
Repeat の入力/出力ノードの間に、Set Position(位置設定)ノードを挟み込んでみました。
Offsetのパラメーターを(0, 0, 0.1)にすれば、１回処理すると頂点が0.1だけ上に移動することになります。

繰り返しの回数は、Iterations のパラメーターで指定ができます。
5を指定すれば、5回繰り返して (0,0,0.5)までデフォルトキューブのメッシュが移動するわけです。
動画)RepeatZone01.gif(pdfでは先頭のコマのみ表示されています)

シミュレーションノードの場合は、出力ノードで保存された情報を、次の時間フレームの入力として利用するという事になりますが、
Repeat Zone の場合はフレームを跨がないでその場で繰り返し処理を行うというところが大きな違いになるわけです。

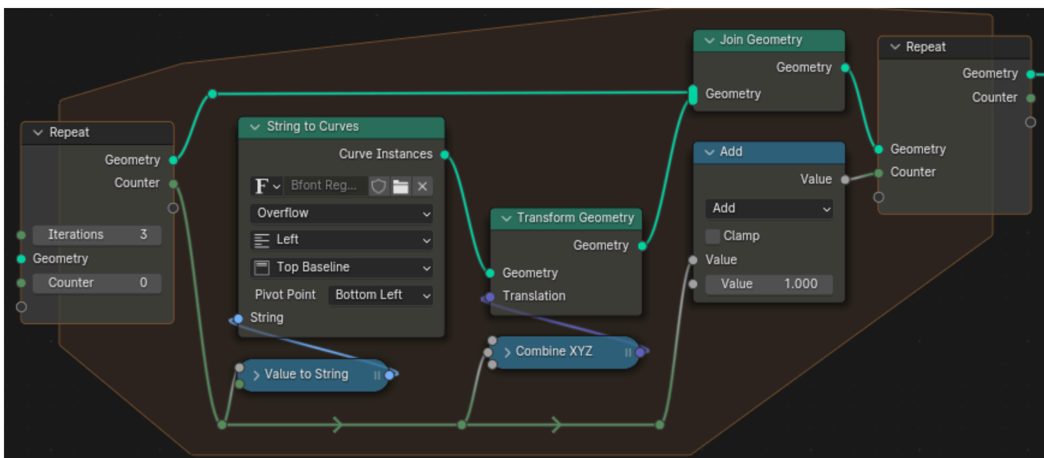


Repeat の入力/出力ノードにソケットを追加して、ループ中で使う変数を設定することが出来ます。



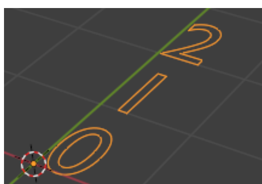
このように 変数Counterを追加設定して毎ループごとに１づつ追加するようにノードを組めば、
今何回目のループ処理なのかが分かるようになります。

後述するように、この変数は[「匿名のアトリビュート」]と理解することができます。



カウンターの数字に合わせて文字を表示するならば、このようなノード構成になります。
ループごとに Join Geometry で新しい文字を追加していくことで、複数の文字が表示される仕組みです。

そのままだと文字が原点に重なってしまいますから、Counter に合わせて位置もずらすようにします。



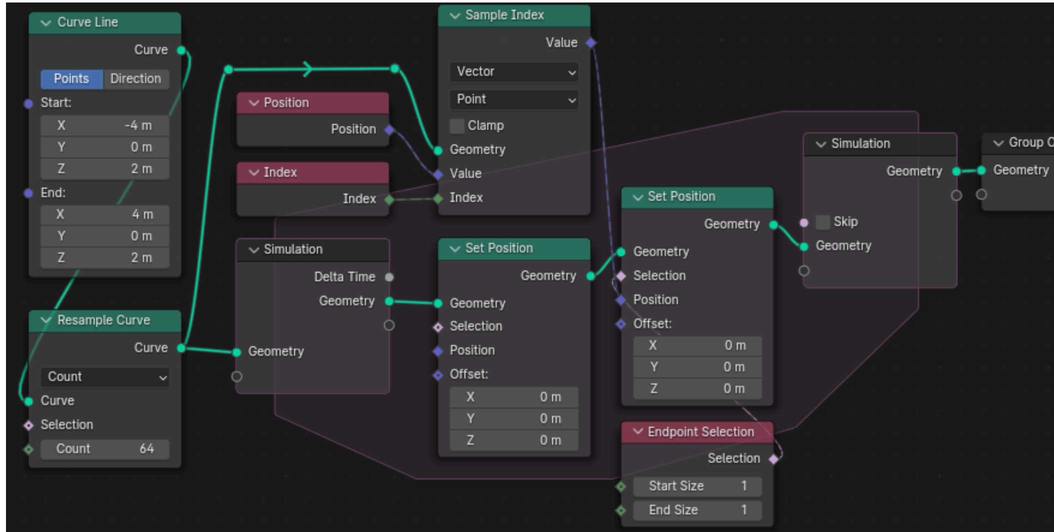
このように毎回違った文字を表示することができました。
Repeat Zone は組み合わせて使うことで、様々な応用が考えられる非常に便利な仕組みです。
また、考え方が似ているので、Simulation Zone の理解にも応用が効きます。
（フレームを跨ぐことが無いので、より分かりやすいとも言えます）
是非使いこなせるようになりましょう。

懸垂線カーブ

紐の両脇をつまんで、真ん中が垂れるようなシミュレーションをしてみます。
本当は、紐の運動だって紐の質量と慣性が重要なのですが、ここでは紐が最終的に落ち着く、垂れた形だけに興味があるとして慣性を無視します。

左の写真のように、紐（写真は鎖ですが）の両脇を固定して垂らしたときにできる曲線を懸垂線と呼びます。
（近似的には放物線の形をしているのですが、厳密には少し違った形をしています）

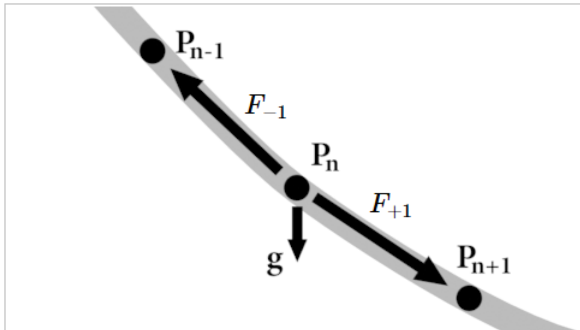
シミュレーション用のノードを組みます。
まずは紐の両脇を固定するところだけ先に作ることにします。



紐自体は、Curve - Primitive - Curve Line で1本のカーブを作成し、Resample Curve によって頂点数を64まで増やしました。

シミュレーションの最後で、先頭と最後の２頂点だけ、初期のカーブでの頂点の位置をコピーしました。
Sample Index(インデックスで評価)を利用して初期カーブの頂点位置を取得します。
先頭と最後の２頂点だけを選ぶのには Endpoint Selection(端を選択)を使います。

力のつり合いを（簡易的に）考えます。紐を複数の点(頂点)でつながったばねのように考えます。
ある頂点 Pn は前の点からは Pn-1 の方向に引っ張られ、次の点からは Pn+1 の方向に引っ張られます。その強さをそれぞれ F_{-1} と F_{+1} としましょう。
この頂点の間がばねでつながれていると思えば、その強さは距離で決まるので、位置の差分を取れば力が分かります。
そして、各点には重力からの力(g)も下(0,0,-1)方向加わります。



バネの自由長を L として、質量やばね定数などを1と考えて無視して簡略化します。
Pn から見た Pn-1 方向は、Pn-1 - Pn をその長さで割って、長さ１のベクトルにすれば表現できます。

その方向に F_{-1} の力を受けていることになります。逆方向の F_{+1} 同様です。
各点の受ける力は

$$\frac{\mathbf{P_{n-1}} - \mathbf{P_n}}{|\mathbf{P_{n-1}} - \mathbf{P_n}|}F_{-1} + \frac{\mathbf{P_{n+1}} - \mathbf{P_n}}{|\mathbf{P_{n+1}} - \mathbf{P_n}|}F_{+1} + \mathbf{g}$$

と表現できます。
※|A| のようにベクトルを縦棒で挟む記号は、ベクトルの長さを表します。

F_{-1} と F_{+1} は、自由長が L のばねと考えれば、（ばね定数や質量などを１として簡略化すれば）

$$F_{-1} = |\mathbf{P_{n-1}} - \mathbf{P_n}| - L$$

$$F_{+1} = |\mathbf{P_{n+1}} - \mathbf{P_n}| - L$$

となります。
元の式に代入すると、一部は分子と分母が打ち消しあって整理できるので

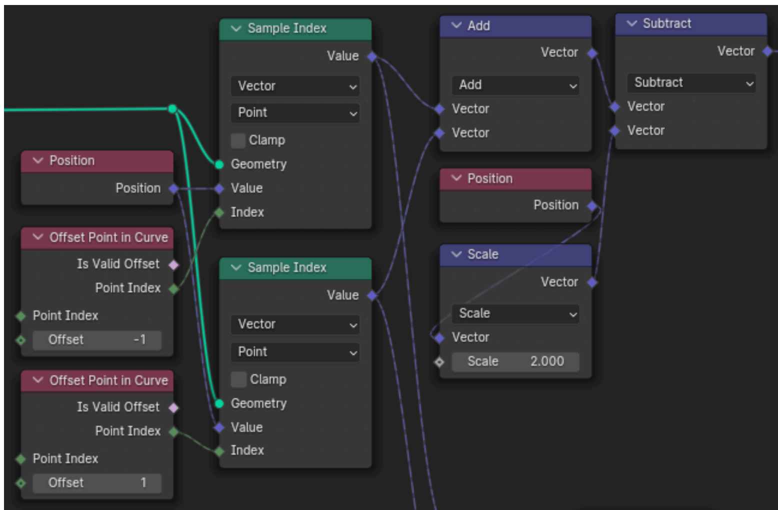
$$\mathbf{P_{n-1}} + \mathbf{P_{n+1}} - 2\mathbf{P_n} - \mathbf{Q} + \mathbf{g}$$

となります。自由長Lに関係する部分は、うまく打ち消しあわないので Q という文字にまとめました。Qは

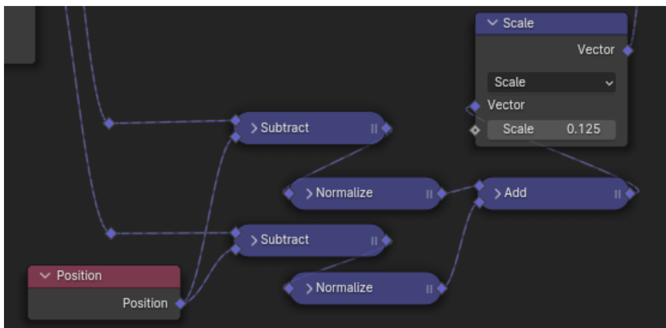
$$\mathbf{Q} = \left(\frac{\mathbf{P_{n-1}} - \mathbf{P_n}}{|\mathbf{P_{n-1}} - \mathbf{P_n}|} + \frac{\mathbf{P_{n+1}} - \mathbf{P_n}}{|\mathbf{P_{n+1}} - \mathbf{P_n}|}\right)L$$

とやや複雑ですが、分数の部分は実際には差分のベクトルを長さ１に調整しただけなので、
ノードに落とし込むときには、Normalize ノードを使えば良いので簡単です。

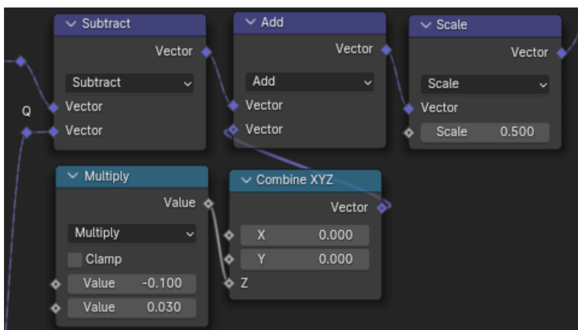
本当の物理であれば、受けた力に従って速度が変化し、その速度に従って運動するので紐は勢いでびよんびよん動きます。
しかし、つり合いを求めるのであれば速度を無視して、力に比例して各点を少しづつ動かしていけば、最終的に力の釣りあう形状まで求めることができます。
例えて言えば強烈に粘性が効いて、速度が毎フレーム0に戻っているような事態に相当します。
そうした状況でも、紐の形状はゆっくりと釣り合いの形状(静的平衡状態)まで近づいていきます。



Pn+1 と Pn-1、そして Pn の位置を得て力の前半部分を計算します。
Pn+1 と Pn-1はこれはつまり「次のインデックスの位置」と「前のインデックスの位置」なので、
Curve - Topology - Offset Point in Curve(カーブ内ポイントオフセット)と、
Geometry - Sample - Sample Index(インデックスサンプル)
を利用すれば情報が得られます。



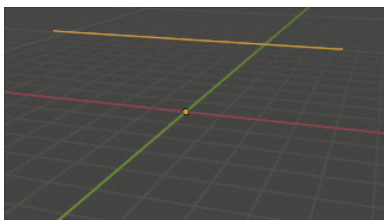
Q で表現した部分も計算します。
先ほどの Pn+1, P, Pn-1 を使った四則演算が主なので、ノードを閉じて省スペース化しています。
最後に、自由長さ L の値で掛け算が必要です。
今回最初に長さ8のカーブを64頂点数化したので、頂点間の距離は 約1/8 ということで 0.125 としました。
正確には、64頂点の間は63の辺で結ばれているので、0.125ではないのですが…



最後に合計して、重力 g の分も足し合わせます。

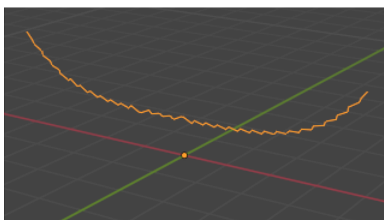
見た目として程よくばねが伸びて懸垂線になるように値を調整しました。
g を小さい値で微調整するには、ノードの小数点の表示桁数が3桁でやりづらいので、
Multiply ノードを利用しています。

あまり一ステップで大きく動かないように、Scale で0.5倍程度しておき、
この力の分だけ、毎ステップ頂点の位置を動かしていきます。



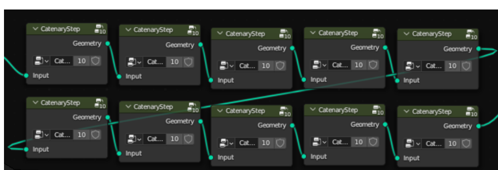
時間を進めると、かなりゆっくりとカーブが垂れていき懸垂線に近づいていきます。
動画)Catenary01.gif(pdfでは先頭のコマのみ表示されています)

このまま時間を大きく進めると、懸垂線で釣り合うようになりますが、
250フレームぐらい進めてもまだ最終状態に落ちつきません。

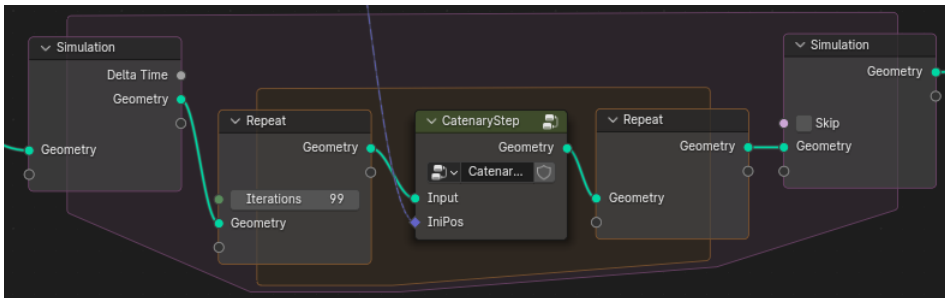


しかし一度にもっと大きく進化させようと、最後にScaleノードで0.5倍した部分をもっと大きく変更すると、
このようにギザギザの形が発生して計算が破綻してしまいます。
もう少し大きくすると、頂点がどこかへ吹っ飛んでいくと思います。
このようなシミュレーションでは、早く進めようとして１ステップに大きく動かすと、
こういった不安定が発生して計算が破綻することが良くあります。

こうした現象を防ぐために、１ステップで進む変化をある程度抑える必要があります。
しかし、変化量が少ないとなかなかシミュレーションが進まず、無駄にフレーム数が必要です。
例えば時間進化１フレームあたり10ステップ分ぐらい計算を繰り返すことが出来れば、不安定を起こさないで10倍時間を早く進めることが出来ます。

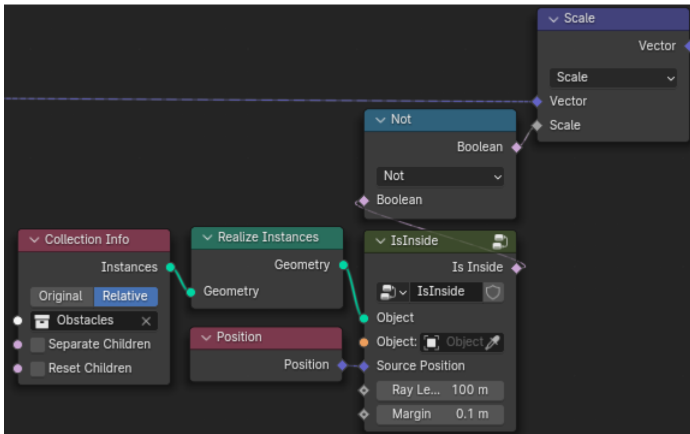


「ループ機能」は Blender 3.6 時点では無かったので、ノードグループを数珠つなぎに並べて自力でループ（みたいな）処理をする必要がありました。



Blender 4.0 以降は、ノードグループにまとめた処理を Repeat Zone で繰り返すだけで簡単です。99回ループさせるといった無茶も簡単に行えます。
(上から伸びているのは、カーブの両端を固定するための初期位置情報です)

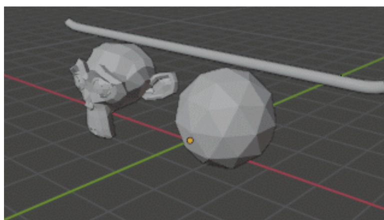
この状態のサンプルは、02_CATENARY/CurveCatenary_01.blend として同封しました。



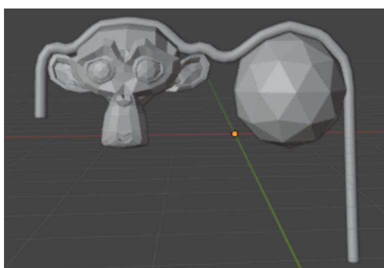
さて折角ですから、ただ紐を垂らすだけではなくて障害物に反応するようにしてみましょう。
障害物と接触しているか判定するノードをノードグループの中に追加します。

障害物用のオブジェクトを Obstacles という名前のコレクションにまとめておきます。
Collection Info を使ってコレクションをまとめたオブジェクトを作り、Realize Instances で実体化をしておきます。
（衝突判定はインスタンスのままではできません。メッシュにしておく必要があります）

先ほどの接触判定のセクションの理屈を使って、接触（衝突）判定をするノードグループ(IsInside)を組みました。
それを使い、メッシュから0.1よりも近づいた場合に、False をかけ合わせることで、紐が受ける力を 0 にするようにノードを組みます。



Curve to Mesh を使い太さ0.1のメッシュにすれば、
接触判定をする距離と丁度良くなります。
障害物にぶつかったところで動きが止まるような懸垂線を実現できました。
動画)Catenary02.gif(pdfでは先頭のコマのみ表示されています)



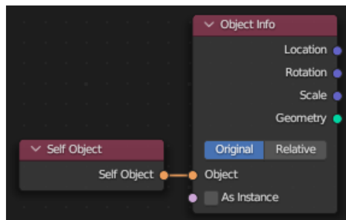
両端を固定する仕組みの部分を Mute(ミュート) などで無効化すれば、
このように両端が垂れるような状態に落ち着きます。

釣り合った平衡状態の形で良ければ、最後のフレームでジオメトリノードを適用してやれば、どんな形の障害物に対しても上に乗っかった紐の形状を作れるわけです。
このサンプルは、02_CATENARY/CurveCatenary_02.blend として同封しました。

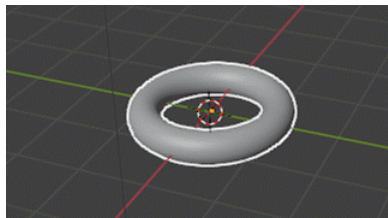
※このやり方で計算した懸垂線は、紐の長さは大分伸びていることに注意は要ります。
バネの自由長を初期状態の長さとしているわけですが、これだけ垂れた状態で釣り合っているという事は、かなりばねが伸びないと自身の重さを支えられないこいうことが分かります。
ばね定数を大きくして強力なバネに相当する計算をすれば、より伸びづらい状態での懸垂線になりますが、わずかな動きで大きく力が変化するようになるため、
その分時間刻みを細かくしていかないと、シミュレーションが破綻しやすくなります。

遅延とお化け

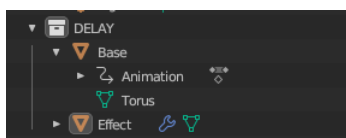
位置の情報を使ったシミュレーションとして、オブジェクトが遅れてついてくるような、遅延効果を作成してみましょう。
簡単な遅延効果は自分自身の「遅れている」位置情報と、「遅延が無ければ本来あるべき」位置情報があれば作成することができます。



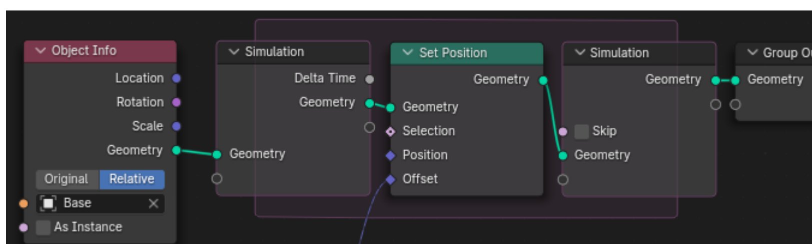
Self Object(オブジェクト自身) と Object Info の組み合わせ使えば、本来あるべき位置情報が得られます。
そのためオブジェクト１つに適切なジオメトリノードを設定することで、遅延効果を演出することが出来るのですが…



動画)Delay01.gif(pdfでは先頭のコマのみ表示されています)
自分自身を動かしているので、自分の原点も動いて行ってしまいます。
遅延効果はグローバルな座標で遅れてついてくるような効果なので、
ローカル座標とグローバル座標の関係がややこしくなってしまいます。
(ジオメトリノードで作成した形状は、自身のローカル座標で表示されますが、自身の原点も動いている状況です。)

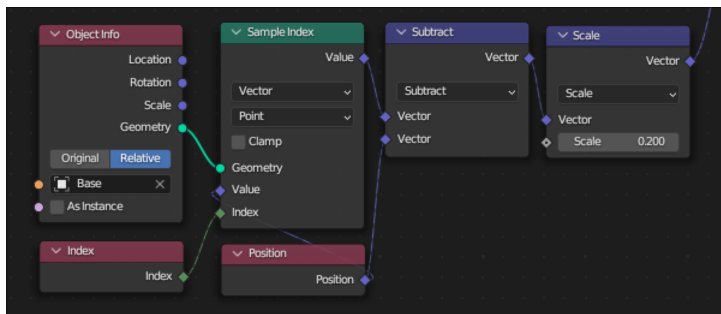


そこで、今回のサンプルでは元になる Base オブジェクトと、
エフェクト用に、原点に固定したエフェクトとの2つに分けます。
Effect 側にジオメトリノードを設定し、Base のオブジェクトについていく効果を作成してみましょう。



原点にある Effect オブジェクトから見れば、
Base オブジェクトの Relative(相対)で見た Object Info の情報が、
グローバルな座標での情報そのままです。

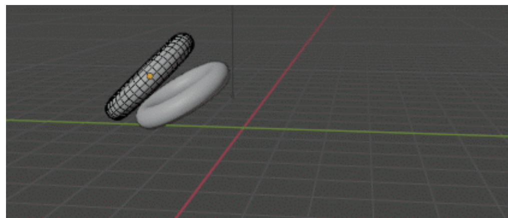
そこで、Simulation 入力ノードに、そのジオメトリデータを直接繋いで初期条件にします。
Effect オブジェクトは自分自身のデータは全く使わないわけですね。



初期条件で Base Object のジオメトリを Effect 側にコピーしているので、
インデックス構造などは同一になっています。
Effect オブジェクトの頂点の Index 番号が分かれば、対応する Base オブジェクトの情報が得られます。

Base Object の Relative の位置を Sample Index すれば、
Base オブジェクトの（遅延が無ければ現在あるべき）各頂点のグローバルな座標が分かります。

後は(遅延している)自分の頂点の位置を、本来あるべき頂点の位置の方へ動かしてやれば良いわけです。
ここでは、引き算をして差分をとり、その20%分だけ近づいていくことにしました。



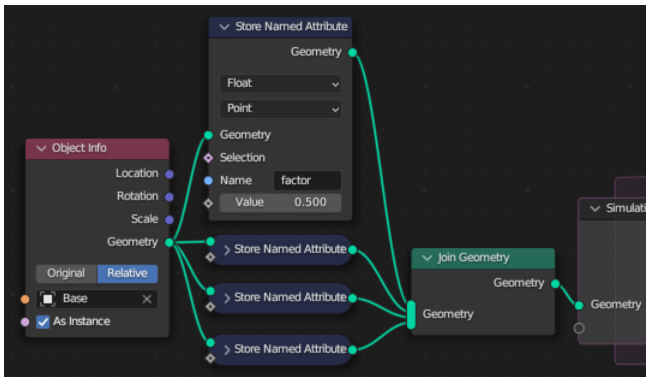
Base オブジェクトにアニメーションを付けて、ついてくる Effect の効果を見てみます。
区別しやすいように、Base 側にはワイヤーフレームも表示しています。
動画)Delay02.gif(pdfでは先頭のコマのみ表示されています)

ワイヤーフレームの無い Effect が遅れてついてくるのが分かります。
Scale で指定したファクターを1に近づけるほど、素早く本来の位置に近づくので遅延が少なくなります。

このサンプルは 03_GHOST/01_Delay_01.blend として同封しました。

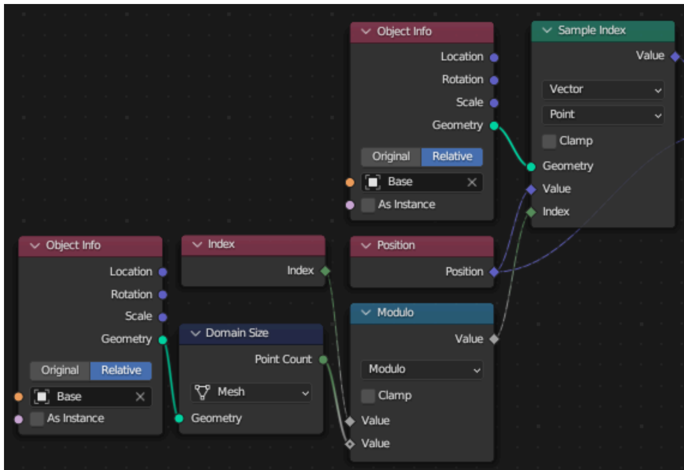
多少気を付ける必要があるのは、エフェクトは直線的に最短距離で追ってくるので、 回転運動している時などは内側の最短コースをつくことになり、全体的に小さくなってしまう事です。
そのため、移動のカーブに沿って弧状についてきてほしいエフェクトなどには向いていません。
例えば、刀を振ったときに、軌跡の弧に沿ってエフェクトを置きたいような場合です。
これを防ぐには、過去数フレームの情報などを使って、曲線で動いた頂点はその曲線を追いかけるようにする必要がありますが、ここではそこまでは踏み込まないことにします。

この Delay 効果を複数使って、残像のような効果を作成してみます。



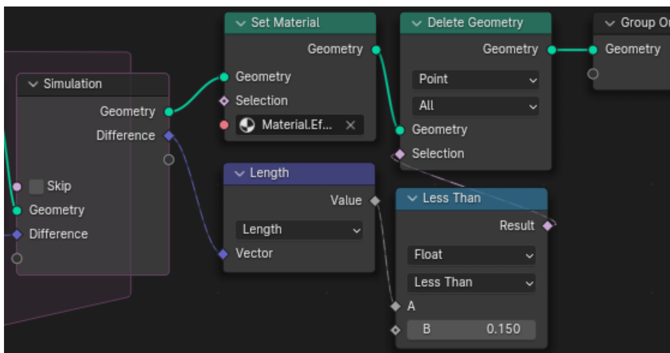
遅延した残像がどれぐらいゆっくりついてくるのかは、
先ほどのノード組みで、頂点の移動量を Scale したファクターが決めます。
さっきは、ファクターを 0.2 にしていました。

左の図では Geometry を分岐させて、それぞれに Store Named Attribue しています。
これでアトリビュート違いの複数のジオメトリが作成できるので、Join Geometry でひとまとめにします。



頂点数が残像の数だけ増えているので、それに対応するために、
Sample Index に繋がる index の情報を調整します。
元の頂点数以上の Index の頂点は２つ目３つ目以降の残像です。

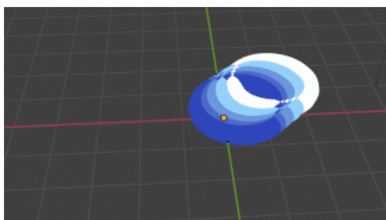
Domain Size を使えば頂点数が分かりますから、Floored Modulo(剰余(床))ノードを使って余りを計算して、
2番目、3番目の残像でも、1番目の残像と同じように、元メッシュの対応する頂点情報を得られるようにしました。



最後に残像の factor に応じて色を変えられるようにマテリアル設定などを行います。

このとき残像に半透明などを使って見え方の工夫をしないと、
残像が本体を隠してしまって本体が目立たない…といった問題がでることもありそうです。

今回は、本体と残像の位置の差もシミュレーションノードから出力しました。
差分が小さい時には残像を削除して出てこないようにして、
本体が残像に隠れる状況が発生しづらくする工夫をしています。



色違いの残像を複数配置するエフェクトです。
動画)AfterImage01.gif(pdfでは先頭のコマのみ表示されています)

このサンプルは 03_GHOST/02_Delay_02.blend として同封しました。

追記：
今回は４つの残像を表示しています。Join Geometry で 4つのジオメトリをまとめているからです。
Blender 4.0 では Repeat Zone があるので、簡単に100個の残像を作成するというようなこともできます。
（実用的かはさておき…）
サンプルファイルに一部追加をして、Repeat Zone で残像を増やすことも可能にしました。
元々の仕組みとは、Switch ノードで切り替えることができるようになっています。

ループ処理の中で、Join Geometry を繰り返すことでジオメトリを増やしていきます。
ループごとに値が増えていく Counter 変数を用意することで、それに従って factor のアトリビュートを設定する仕組みです。

残像の数を24個にしたらこのようになりました。
factor の値の取り方を、Float Curve などを使ってもう少し良い感じになるように工夫ができそうです。

お化け…手描きモーションブラー

お化けは、手描きのアニメーションでモーションブラーのような効果を表現するときの手法の名前（俗称？）です。
素早い動きをする物体の輪郭線をグニャグニャと歪ませたりするのが基本ですが、形自体を大きく変化させて、躍動感を増したり描き手の工夫次第で様々な派生があります。
遅延の仕組みを利用して、お化け効果のような変形を再現してみましょう。
考え方の基本は単純で、遅延の効果を頂点ごとに変化させれば良いことになります。

移動のファクターは、１の時に完全に追随して遅延がゼロ、0の時に全く移動せずに頂点が元の位置に残るようになっていましたから、
頂点の一部だけが良い具合に遅延するように、ランダムの後に Float Curve で手動で丁度良い遅延具合を探ります。

しかし、ランダムに遅延させるだけだと進行方向側もぼこぼこになってしまい、今一つです。
そこで、移動方向の反対側だけが遅れてついてくるように調整をしておきます。

移動方向と法線の Dot Product(内積) を計算しておき、遅れてくる方向側だけ遅延を有効にします。

この時、Normal ノードで得られるのは「変形している形状」での法線なので、場合によっては問題があるかもしれません。
Sample Index を使い「本来の形状」での法線情報を得てそれ使うべきかもしれませんが、
サンプルを作成して試してみたところでは、あまり問題は発生していない様でしたので、Normal ノードをそのまま使用してみました。

移動方向の反対側だけで一部の頂点が引き延ばされて、
簡単なお化けエフェクトの様に遅延をしています。
動画)Ghost01.gif(pdfでは先頭のコマのみ表示されています)

このサンプルは 03_GHOST/03_Ghost.blend として同封しました。

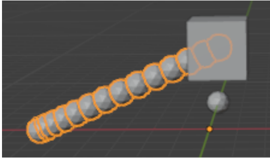
エフェクトの細かさなどは、取り残される頂点の密度等に依存します。
そのため、Subdivide Surface 等で頂点の密度を調整したりする必要があるかもしれません。
また、素早く弧を描くような動作に対しては、直線的なお化けしか表現できないので、向いていないこともあります。
しかし、簡単な仕組みを考えれば充分に効果的な手法と言えそうです。
もっと凝ったお化けエフェクトの追及は、後のセクションで行ってみましょう。

アトリビュートを使ったシミュレーション

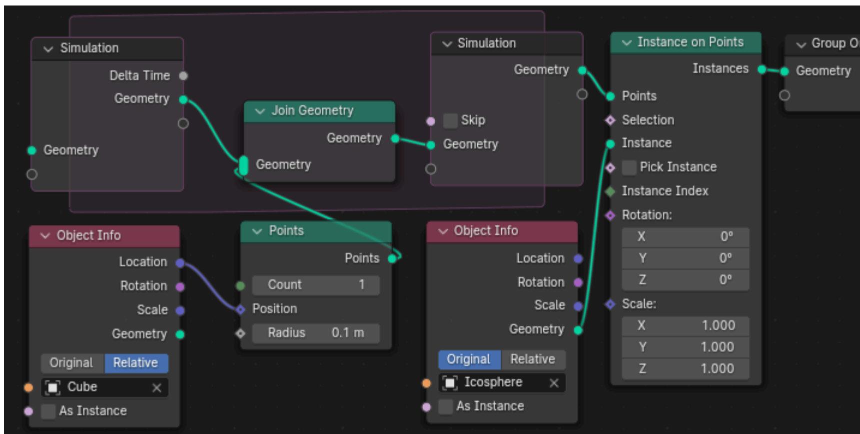
今までは、頂点などの位置情報だけを使ったシミュレーションの例を見てみました。
「今の位置の情報」を使って「次のステップの位置」を決めるという計算になっているものです。
それらは、単純な例から複雑な例に順番に進めるように（少々わざとらしい位に）位置情報だけを使う例を集めたものです。
もう少し複雑な例として、今度は位置以外の情報を使った例を見てみましょう。

パーティクルの経過時間(Age)

先ほど、オブジェクトの軌跡に沿ってパーティクルを生成するようなシミュレーションの例を見てみました。
生成したパーティクル（のような働きをする頂点）に、発生からの経過時間(Age)の情報を持たしてみましょう。
任意の情報を頂点に与えるために、Named Attribute (名前付き属性)に関するノードを駆使することになります。

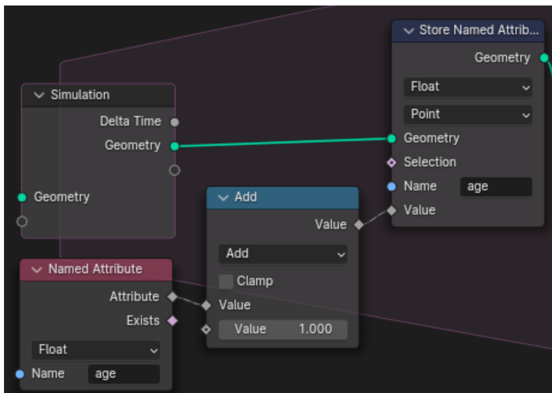


パーティクルを生成する基本形は先の例と同様です。
まず、キューブをアニメーションで動かして、その後ろにパーティクルのような軌跡を残すようにしましょう。



キューブとは別に軌跡を表示するためのオブジェクトを作り、ジオメトリノードを設定します。
制御用に「Cube」オブジェクトを作成して動かします。
「Cube」の位置に Points によって毎フレーム点を作成して Join Geometry で追加をしていけば、軌跡に沿って頂点が並びます。

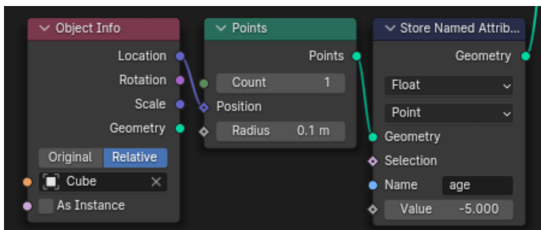
ただし頂点を配置するだけだとレンダリングしても見えません。
軌跡に並んだ頂点の位置に Icosphere のインスタンスを配置して表示する仕組みです。



汎用の情報、つまりアトリビュートを使ったシミュレーションをするときの基本形はこの図のようになります。
Store Named Attribute(名前付き属性収容) を使って頂点(や辺や面)にアトリビュートを与えます。
そして、そこに与える情報とも同じ名前を使った Named Attribute(名前付き属性) からノードを繋ぎます。

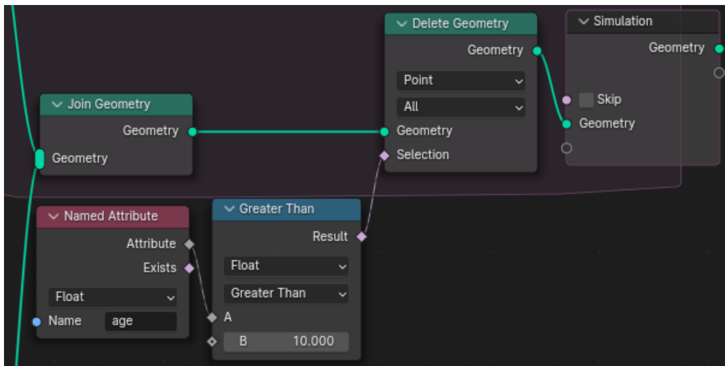
この例では、一回シミュレーションのステップが進むときに「今持っているage情報」に１が足されて収容されるので、
次の時間フレームの時には頂点ごとの Age が１だけ増えているという仕組みです。

この時、「前ステップで生まれたばかりの頂点」は Store Named Attribute で値を与えられていないので「理屈上はまだ Age 情報を持っていない」点に気が付いた人は鋭いです。
そういう場合には Age は 0 として扱われます。
そのため「生まれた時は 0 そして時間フレームが１進むごとに1づつ年を取っていく」という進化をします。

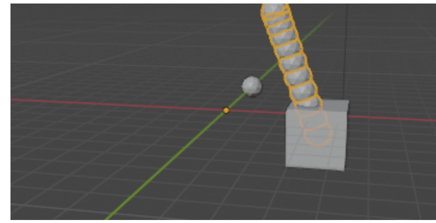


初期の値を 0 ではなくしたい場合もあります。
今回のような場合であれば、頂点を生成するときにもそこで初期値として別の値を age に与えることもできる訳です。

初期値にいろいろな値を渡せるので Age のように0から増えていくのではなく、
例えば逆に「ライフポイント」のようなアトリビュートを最初に与え、
時間とともに減っていくような考え方の仕組みを作ることもできるわけです。

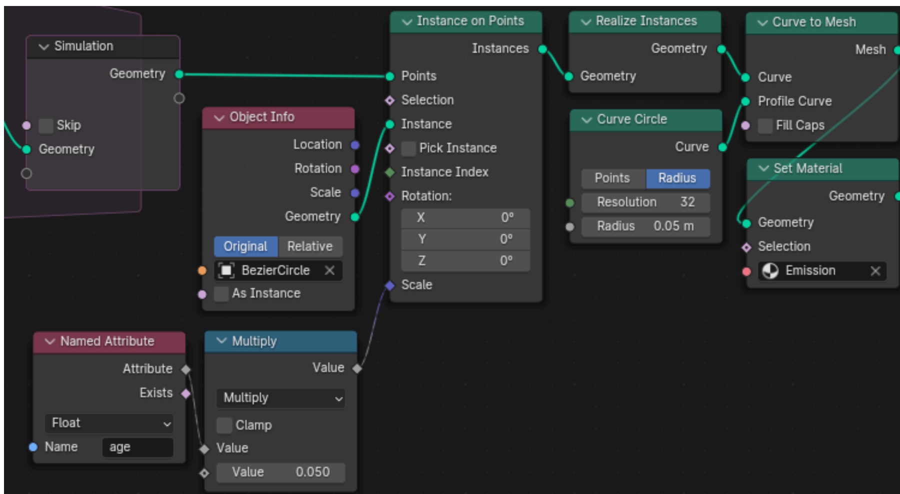


例えば、Age が 10 を超えたら消滅するような効果にしてみましょう。
Delete Geometry を使って条件を満たす頂点が消えるようにノードを組みました。


動画)Age01.gif(pdfでは先頭のコマのみ表示されています)

初期値を-5に設定した場合は、誕生から15フレーム目で消滅することになります。
ここまでのサンプルを 04_ATTRIBUTE/01_Age_01.blend として同封します。
これだけだと少々効果としては単純なので、もう少し凝ったエフェクトにしてみます。

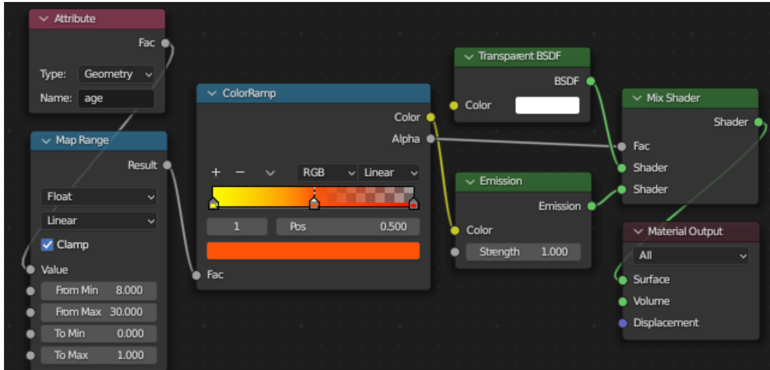
単にオブジェクトが軌跡上に配置されるのではなく、波紋が広がるようなエフェクトを作成してみます。



まず、頂点の位置に配置するインスタンスを変えてみます。
エフェクト用のインスタンスを Icosphere から カーブによる円(Bezier Circle)にしました。
円の Scale を age による大きさに依存するようにして、円が age とともに大きくなるようにします。

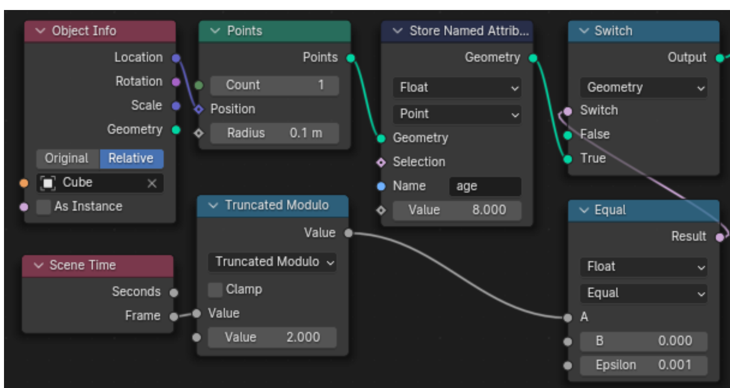
太さ情報が無いままだとレンダリング時に見えませんから、
Realize Instance によって実体化して、Curve To Mesh によって太さを持ったメッシュに変換をします。

折角ですので時間と色による演出を加えるために Emission という名前のマテリアルを作成してセットしました。



Emission のマテリアルは、age の違いによって色（と透明度）が変化するように、
Attribute ノードと ColorRamp ノードを繋いで調整します。

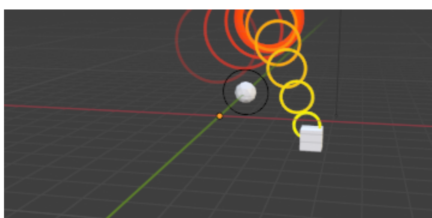
この時、MapRange を使って生成から消滅までの間の age が [0-1] の範囲になるように調整をします。
今回の例では、エフェクトがいい感じに見えるように、age は生成時が 8 で 30 になると消滅するようにしています。
（数値は目で見ていい感じになる、という理由で決めた値です）



波紋が毎フレーム発生すると、密に発生しすぎてあまりエフェクトとして上手くありません。
（前後のフレームに発生した波紋との区別がつきにくくなります）

そこで、2フレームに２回だけ波紋、つまり配置の元になる頂点が発生するように
Scene Time ノードと 数式の Truncated Modulo(剰余(床)) ノード、Switch ノードを組み合わせます。

また、波紋発生時に既にある程度の大きさを持つように、発生時の Age を 8 に設定しました。



波紋が発生するエフェクトになりました。
動画)Age02.gif(pdfでは一部のコマのみ表示されています)

このサンプルは 04_ATTRIBUTE/Age_02.blend として同封しました。