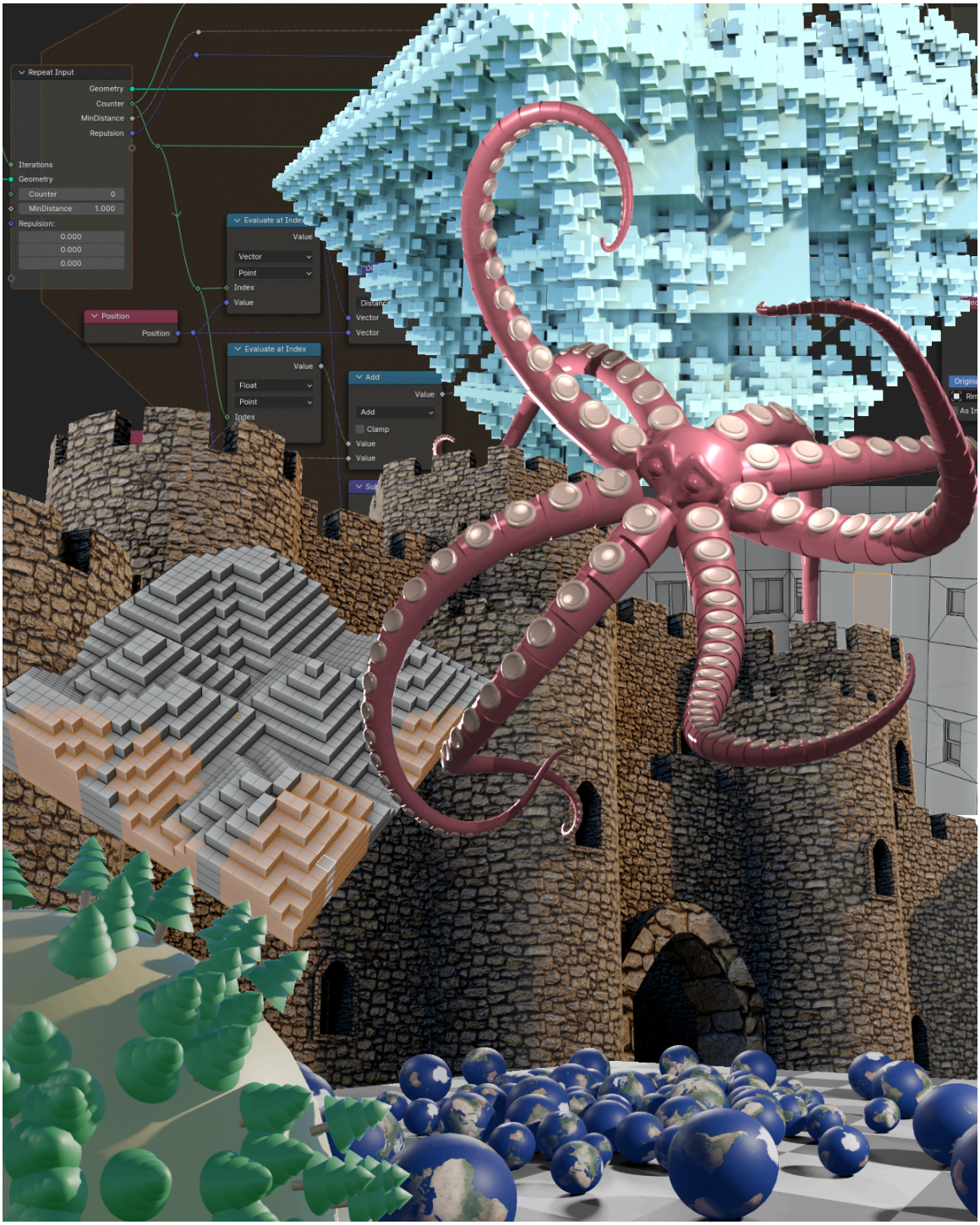
Blender ジオメトリノード 解説＆作例集 Vol.4

Geometry Nodes (for Blender 4.0)

Version 1.1

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

Blender ジオメトリノード 解説&作例集 Vol. 4

Geometry Nodes (for Blender 4.0)

作者 Twitter アカウント
Q@スタジオぽぷり

2023.11.16: ver. 1.0
2024.03.27: ver. 1.1

初めに

ジオメトリノードは、Blender 2.92 で導入された機能で、ノードを使ってオブジェクトに対して様々な操作をすることができます。
Blender の開発の中でもホットな分野で、その後も次々に新しい機能が実装されています。
本書のシリーズの最初、Vol. 1 は Blender 3.0 の時にリリースをしました。
その後も、Blender が 3.x そして 4.0 へと更新されるにつれて、強力なノードがいくつも追加されています。
そこで、Vol. 2 は、Blender 3.2 - 3.3 ごろに追加されたノードを中心とした使った作例集、
Vol. 3 は、Blender 3.4 - 3.5 ごろに追加されたノードを中心に解説と作例をまとめていきます。

Blender 3.6 では、待望のシミュレーションノード機能が実装され、ジオメトリノードは大きく強化されています。
しかし、このシリーズではシミュレーションノードまでは解説しきれないと考え、シミュレーションノードは別の解説本を作成することにしました。
本書 Vol. 4 は、Blender 3.6 以降 Blender 4.0 ごろに追加されたノードを中心に作例をまとめます。
第1.1版で、Blender 4.1 で追加されたノードについての解説を追加しました。

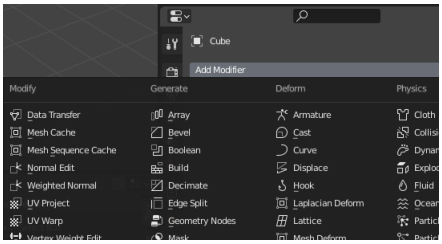
読者には、ある程度 Blender の操作に慣れている人を想定しています。
また、高校数学程度の、ベクトルや三角関数の知識…例えば内積や外積といったような…があった方が、理解がしやすいと思います。
簡単な注意点などは途中で説明を挟んでいきますが、基本的な操作法などは既に理解しているものとして説明していく点は注意してください。
読者は既に以前の本は読んでいるだろう…とは想定をしているのですが、
ジオメトリノードの基本部分の解説は、最初の章でざっとおさらいをしておこうと思います。
「その辺の基本部分は理解しているよ」という方は、最初は飛ばして進んでも大丈夫です。

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。

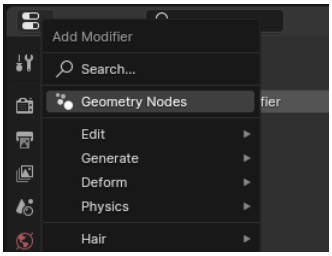
Geometry Nodes の変化

ジオメトリノード自体が、どんどん進化をしている最中なので、このシリーズを書いている最中にもいろいろと変化が起きています。
Blender 4.0 付近での重要な変化を幾つかあげておきます。

モディファイアメニューの更新

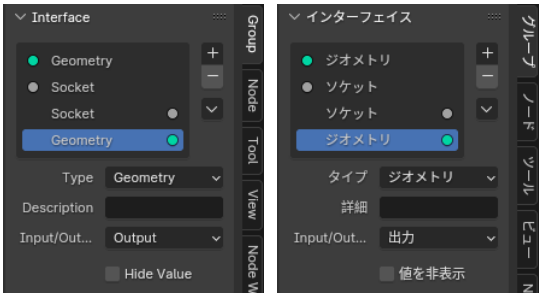


ジオメトリノードはモディファイアの１つとして実装されています。
今まで、モディファイア追加のメニューは、特徴的な大きな１枚のパネルでした。
ジオメトリノードはその中の１つして扱われています。



新メニューになり、モディファイアのカテゴリごとに２段の深さのメニューなりました。
その中で、ジオメトリノードはトップメニューとして特別に扱われていています。

アトリビュートの入出力

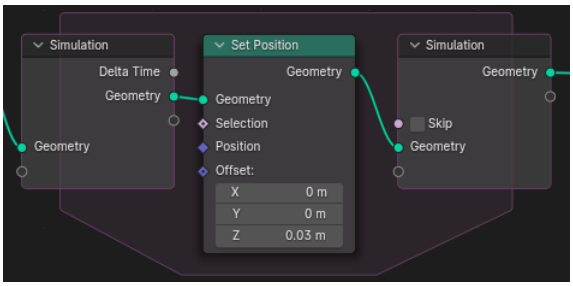


ジオメトリノードと、各種のアトリビュートとをやり取りするために使われる、
Group Input/Output ノードの設定用のパネルが Blender 4.0 で更新されています。
Inputs/Outputs という2つのパネルに分かれていたものが、Interface というノードに統一されました。

この更新は、増えすぎたソケットなどをまとめて閉じたり開いたりするパネル機能の追加に伴って行われています。
パネル機能については後のセクションで解説をします。

シミュレーションノード と Repeat Zone (リピートゾーン)

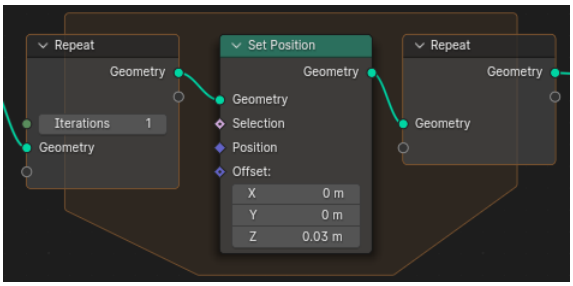
Blender 3.6 では、Simulation nodes(シミュレーションノード)によって、時間的に進化していくような効果の作成機能が大幅に強化されています。
シミュレーションノードに少し似ているのですが、Blender 4.0 で Repeat Zone (リピートゾーン) という繰り返し処理を行うノードが追加されています。



入力用と出力用の Simulation ノードに挟まれた部分の処理が、シミュレーションで実行される処理です。
ジオメトリに何らかの処理をして Simulation の出力まで行くと、
その結果を次の時間フレームの Simulation の入力に使う、という理屈で時間と共に進化していくシミュレーションが実現できます。
このシンプルなノード組みは、毎フレームごとに0.03だけZ軸方向に移動する、非常に単純なシミュレーションの例です。

シミュレーション機能にはこのシリーズでは深く踏み込みませんが、非常に強力な機能なので、
ジオメトリノードの習熟したら是非シミュレーションにも手を出してみて欲しいと思います。

Repeat Zone も似たような仕組みで、入力用の Repeat(リピート) と 出力用の Repeat に挟まれた処理を繰り返します。

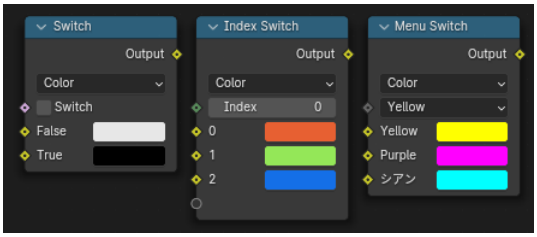


時間が進んだら処理をする代わりに、Iteration で設定された回数だけ処理を繰り返します。
左の図は　0.03 だけ移動するという手順を10回くりかえすノード組みです。
見た目はシミュレーションノードと似ていますが、色が微妙に異なっているのと繰り返しの回数(Iterations)の指定がついていることが分かります。

この例なら、「0.03ずつ10回移動するなら0.3移動すれば良いではないか」というところですが、もちろんもっと複雑なことをするのに役に立ちます。
本書で、Repeat Zone を使った作例をいろいろと扱ってみたいと思います。

Blender 4.1 で増えた機能や仕様の変更

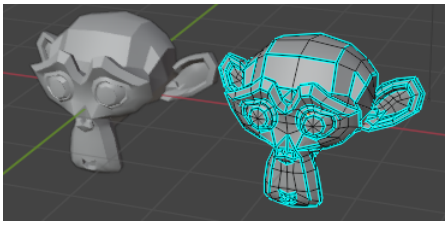
Blender 4.1 でも、様々な機能の追加や更新が行われました。



目立つ機能の追加としては、スイッチ機能の充実が挙げられます。
複数の要素からの選択をしたり、文字で選択肢を選ぶといった操作を行うことができます。

このほか、Blender 4.1 で追加されたノードは、本書では一つの章にまとめて加筆を行っています。

また、Blender 4.1 での大きな変更として、**旧来の Auto Smooth(自動スムーズ)機能が取り除かれて**、ジオメトリノードによる管理に移行しています。
Auto Smooth（的な処理をする場合）は自動的に（ブラックボックス的に）処理されるのではなく、面と辺に対しての Smooth/Sharp 設定で管理が行われます。



例えばスザンヌに Auto Smooth をすると今までは角度の浅い角が自動で滑らかに（急な角はシャープに）なったのですが、
Blender 4.1では、Smooth/Sharp の設定を角度に応じて実行するようにになり、
これに対して後からジオメトリノードで管理することも可能になっています。

若干管理が煩雑になったとも言えますが、その分きめ細かく制御をすることも可能です。

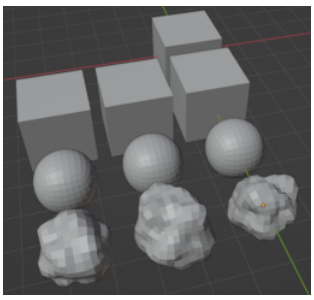
ジオメトリノードの基本

この章は、今までの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、（内容を縮小してより手短に）説明した内容です。
読者は既にジオメトリノードに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、Vol.1 等で確認をしてください。

ジオメトリノードの追加

モディファイア

ジオメトリノードは、モディファイアの一種として実装されています。
モディファイアは、元になるメッシュの形状を保ったまま（非破壊で）変形させるための機能です。



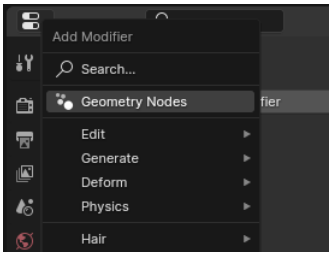
例えば、デフォルトキューブに対して

○ Array(配列)
○ Subdivision Surface(サブディビジョンサーフェス)
○ Displace(ディスプレイス)

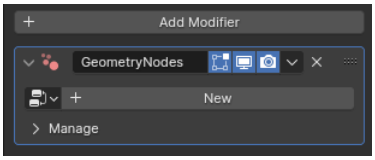
といったモディファイアを順番に追加すると、図のように（元の形状のデータはそのまま保ったままで）変形ができます。
モディファイアを複数使った組み合わせで、かなり複雑な操作をすることも可能です。

これらのモディファイアを後から取り除けば、オブジェクトは元の形状に戻るので「非破壊」というわけです。

ジオメトリノードは、非常に高機能でカスタマイズが可能なモディファイア、と考えることができます。
モディファイアを追加するメニューの中に GeometryNodes の項目があるので追加をします。

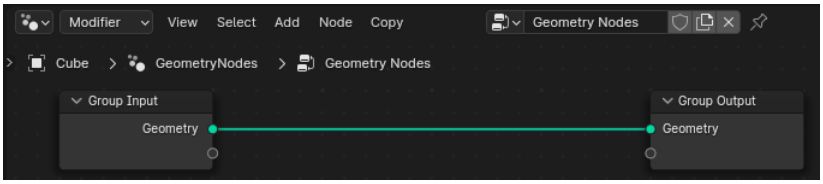


モディファイアのメニュー表示は、Blender 4.0 で更新されています。
ジオメトリノードは、メニュートップの選択しやすい位置に移動しました。



モディファイアは、最初は赤の無効マークで出てくるのですが、これは実際に使うジオメトリノードがまだ空のためです。
New(新規)ボタンを押すことで、新規のジオメトリノードが作成されます。

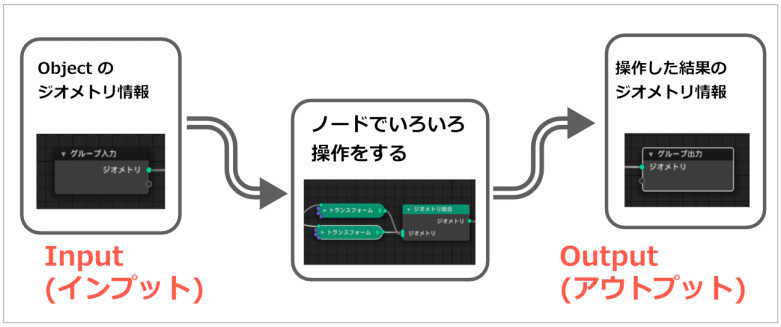
新規作成ではなく、既に使いたいノードが存在しているなら、それをメニューから選択すればよいわけです。



作成したノードの編集は、ジオメトリノードエディタで行います。
モディファイアとしてジオメトリノードを作成したので、
アイコン隣に表示されているモード選択が Modifier となっているのが目につきます。
（Blender 4.0 でモード選択が追加されました）

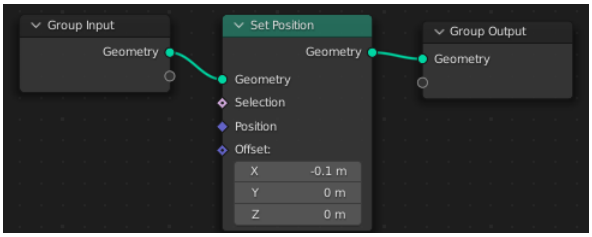
ジオメトリノードの編集

上の図のように、新規のジオメトリノードのデフォルト状態は、
緑の線が Group Input(グループ入力) と Group Output(グループ出力) のノード間をつないだ状態になっています。



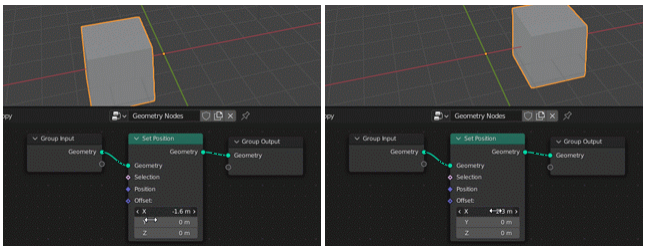
ジオメトリノードの基本形は、Group Input から Group Output の間に、様々な操作を挟み込む形になります。
Group Input (グループ入力) の Geometry (ジオメトリ) ソケットは、元々のメッシュ形状の情報です。
何もせずに直接 Input から Output に繋がれているのは「何もしないでそのまま」という状態です。
その間に、いろいろな操作を挟むことで、様々な編集が行えます。

もっとも簡単なノード編集として、間に Add - Geometry - Write(書込) - Set Position(位置設定)を挟み込んでみます。



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。

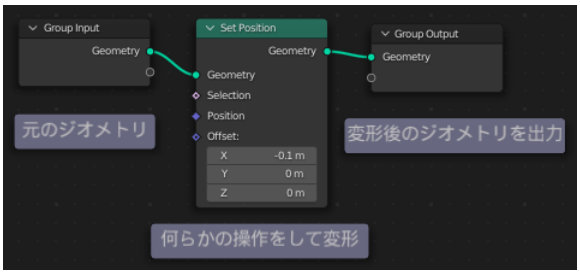


オブジェクト自体が動いているように見えますが、よく見ればオブジェクトの原点の位置は動いていません。
オブジェクトの位置はそのままで、メッシュが変形していることが分かります。

動画)SetPosition01.gif(pdfでは先頭のコマのみ表示されています)

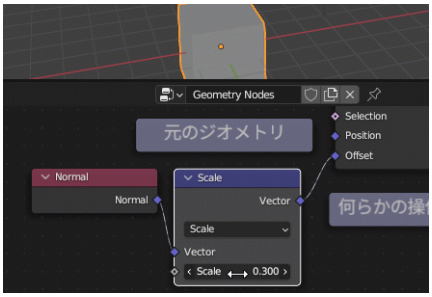
データの流れ(Geometry Node Fields)

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、

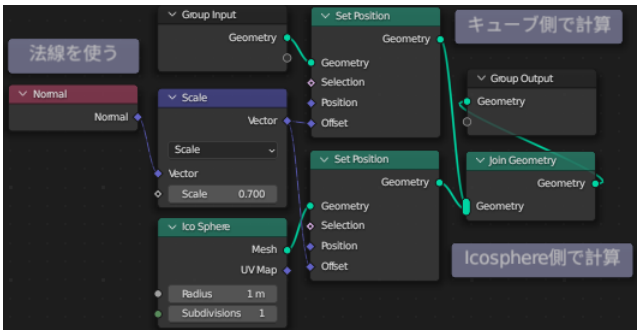
0. Set Position ノードで Offset のソケットに何かつながっている。
1. つながっているのはベクトルをスケールしたものだ。スケールと元のベクトルは何だ？
2. スケールは0.3だ。
3. ベクトルをたどると 法線(Normal)だ。

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）

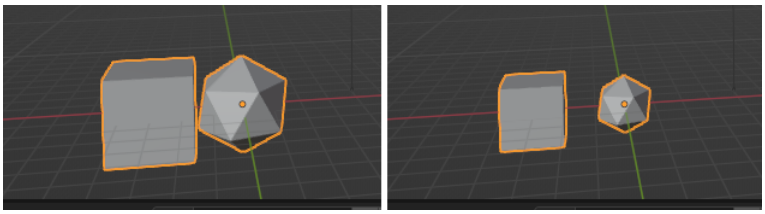
少しわかりづらい考え方なので、例を見てみます。
複数のメッシュを１つに合成する Join Geometry を使ってみましょう。



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。

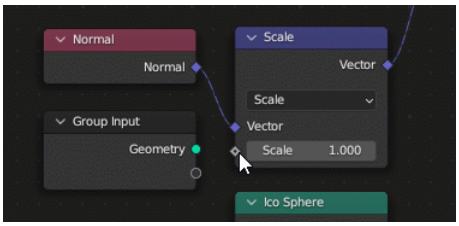


キューブと Ico球 それぞれで法線方向に膨らんでいます。
これは、Normal ノードが法線のデータそのものではなく「法線を使う」という情報を表すためです。
（法線が実際に評価されるのは、緑のライン(ジオメトリ)がつながっている Set Position のノードです。
そこで、「キューブの法線」「Ico球の法線」がそれぞれ別々に評価されているわけですね。）

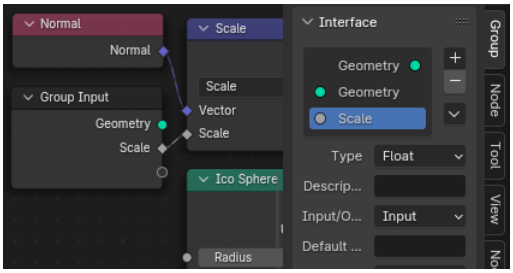
アトリビュートの入出力

今までの例では、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。

アトリビュートの入力

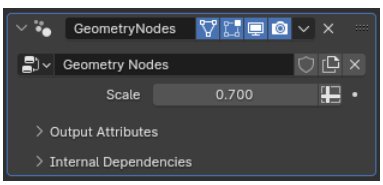


Group Input/Output のノードには、最初から Geometry のソケットが１つだけあります。
Group Input のノードの空のソケットにラインを繋ぐと、新しい入力用のソケットが作成されます。
動画)InputOutput01.gif(pdfでは先頭のコマのみ表示されています)



入出力に関する設定は、画面右側のInterface(インターフェイス) パネルが管理してます。
確認すると、入力出力にに対応した２つのソケットが表示されています。

左の図では Scale ノードに繋がる同名の(Slale)という入力ソケットが追加されています。
ソケットを追加する操作は、Interface パネルの(+)ボタンメニューから行うこともできます。

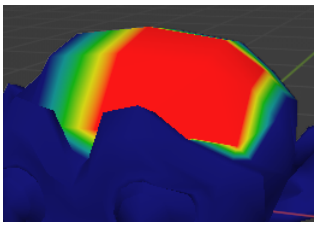


Group Input のソケットを増やすと、モディファイアのパネルに対応する入力欄が現れます。
Scale という名前の入力欄が表示されました。
ここで数値を入力することで、パラメータ違いのエフェクトなどを作ることができるわけです。

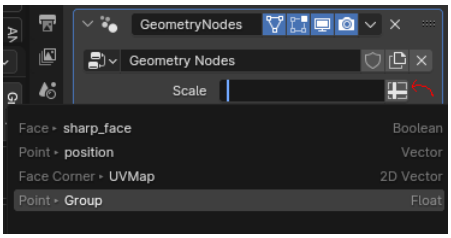
つまり、汎用性を持たせるようにノードを作っておけば、いろいろな場面で使いまわすことができて、便利なわけです。

この Interface(インターフェイス) パネルは、Blender 3.6 までは Input/Output という 2 つのパネルでした。
Blender 4.0 で 1 つの Interface パネルに統合されています。

ソケットの種類は、デフォルトでは Float(つまり普通の数値) ですが、用途に合わせて別のタイプ、
整数であったり、オブジェクトであったり、画像であったり…に変更することができます。

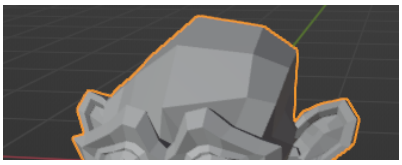


単一の数値だけではなく、頂点ウェイトやカラー属性のように、元のメッシュが持っている属性を入力にすることができます。
例えば、頂点ウェイト(デフォルト名 Group)を作って、一部だけ値を持たせてみます。



モディファイア側の入力欄で十字のマーク(スプレッドシートのアイコン)をクリックすると、
「数値の入力」と「属性の入力」モードが切り替わります。

直接文字で属性名(今回はGroup)を入力するか、選択肢から選ぶことができます。
属性入力の場合は、数値入力の様に「どこでも同一の数値」ではなくて、
頂点ウェイトの様に「場所（頂点）」によって違う値を入力することができます。



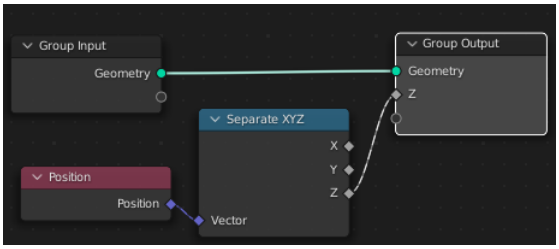
この例では、スザンヌの頭の一部だけが頂点ウェイトの値を持っているので、
頭の一部だけが変形するような操作が行えます。

ここまで、特に断りなく Attribut アトリビュート(属性)という名詞が出てきました。
メッシュの各頂点（もしくは面など）の持っている属性、つまり頂点ウェイトや旧頂点カラー（現カラー属性）もしくはUVマップといった情報は、アトリビュート(属性)と呼びます。

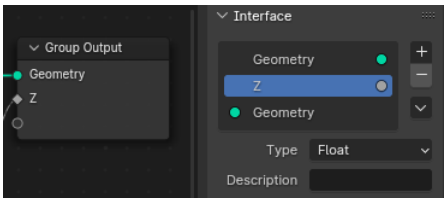
今までUVはUV、頂点ウェイトは頂点ウェイト、というように blender内部の仕組みはバラバラに作られていました。
ジオメトリノードの登場以降、こうしたパラメータなどはアトリビュート（属性）の一種として扱いが統一されつつあります。
ジオメトリノードで様々な操作を統一して行うことができるように改造の最中なのでしょう。
（Blender 3.x あたりのバージョンは、そうした統一の過程の過渡期になっているようです）
今後も、この流れが続いて、ジオメトリノードで扱うことの出来る属性が増えていくのではないか思います。

アトリビュートの出力と利用

既に編集されて用意された頂点ノードなどのアトリビュートを（入力として）ジオメトリノードで使うだけではなく、
ジオメトリノードで計算した結果をアトリビュートとして出力し、その後別の機能でそのアトリビュートを使うことができます。
そうした出力の仕組みを見てみます。
まず、先ほどと同様に頂点グループ「Gruop」を設定しておきます。

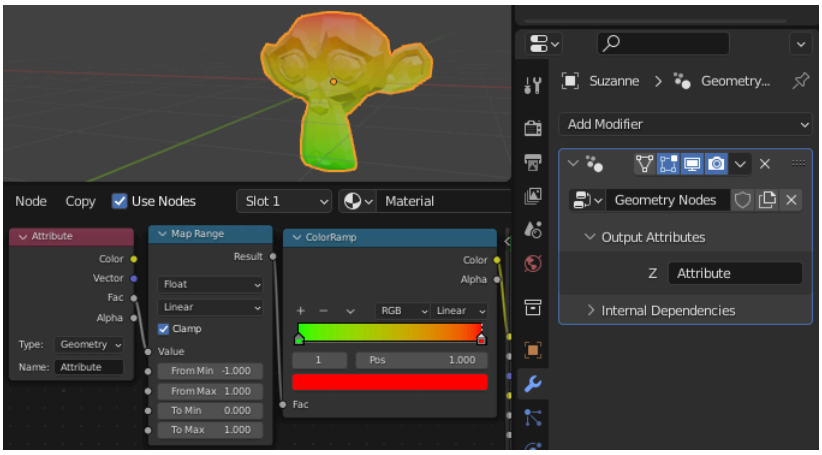


頂点の位置(Position)ノードから位置を得て、
Separate XYZ(XYZ分離)ノードで Z 成分を取り出しました。
Group Outuput につなげると、出力用のソケットが出現します。



Interface パネルを見ると、出力側のソケットが増えているのが確認できます。
もちろんGroup Output ノードの空ソケットにラインをつなぐだけではなく、
直接(+)ボタンを使って編集してソケットを増やすこともできます。

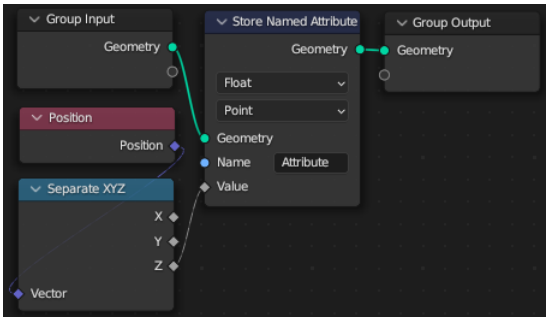
これで頂点の Z 座標成分を外部に取り出して利用することができます。
例えば、別のジオメトリノードを追加して入力として使ったり、シェーダーから利用するなどです。



ジオメトリノードを使って、頂点位置の Z 成分を、
Attribute という名前のアトリビュートとして出力しました。

この情報を、シェーダーのアトリビュート(Attribute)ノードから利用することができます。
[-1,1]の範囲でColor Ramp(カラーランプ)ノードで着色をした例です。

このアトリビュートの出力の仕組みを使うと、
ジオメトリノードによる効果と、シェーダーによる効果を結び付けることができます。
工夫次第で非常に強力な組み合わせになります。

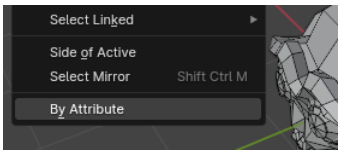


Group Output を使ってアトリビュートを出力することもできるのですが、
Store Named Attribute(名前付き属性収容) を使ってアトリビュートの出力をすることが出来ます。
ユーザーが属性名を変更する必要などの無い、名前決め打ちで良い場合には、
こちらの仕組みを使う方が使用時の手間が少なくて便利でしょう。

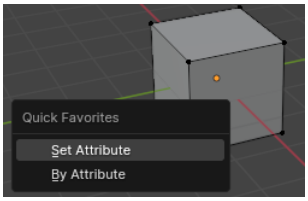
アトリビュートの編集

UVや頂点ウェイトなどと違い、汎用のアトリビュートには専用の GUI による編集ツールなどはありません。
簡易的なメニュー操作でのみ直接の編集が行えます。

メニューの Mesh - Set Attributes(属性を設定) で選択した要素のアトリビュートを直接入力します。



また、ブーリアンのアトリビュートに関しては、True の頂点/辺/面を選択する、
Select - By Attribute(属性) という操作が用意されています。
※ブーリアンの属性でしか使えず、Integer や Float の場合に使えないので少し不便です。



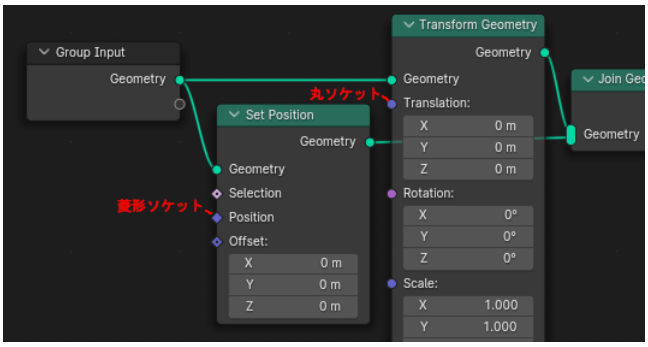
ジオメトリノードを多用する場合には、
これらのメニュー項目は、Quick Favorites(お気に入りツール)に登録しておくと便利ではないかと思います。

ひし形のソケット、丸いソケット

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ひし形のものと丸いものがあることに気が付きます。
初めのうちは違いを意識することは少ないのですが、ソケットの形には実は意味があります。

要素ごとのパラメータと、１つだけのパラメータ

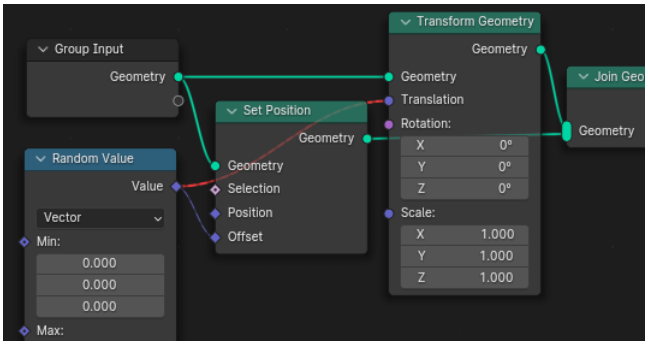
メッシュを平行移動で動かしたいときに、Transform (トランスフォーム)のノードを使うこともできますし、Set Position(位置設定)ノードを使うこともできます。



これらのノードのソケットをよく見ると、Transform ノードには丸いソケットが、
Set Position ノードには菱形のソケットがつながっています。

この時、丸いソケットは「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。
平行移動や、回転、スケールの変換などは、１つのパラメータで表現できるというわけです。

Set Position のノードは、すべての頂点を同じように動かして平行移動させるだけ…ではなくて各頂点をばらばらに動かすことができます。
上の図の例で Offset に (1, 0, 0)を設定をすると、「すべての頂点に対して」(1,0,0)の平行移動をしますが…



Random Value(ランダム値)などをつなぐと、各頂点に対してばらばらのランダムの値が与えられて、
頂点がそれぞれランダムに移動します。
このように、ただ１つだけの値ではなく「各頂点ごとのパラメータ」をやり取りする、ということ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。

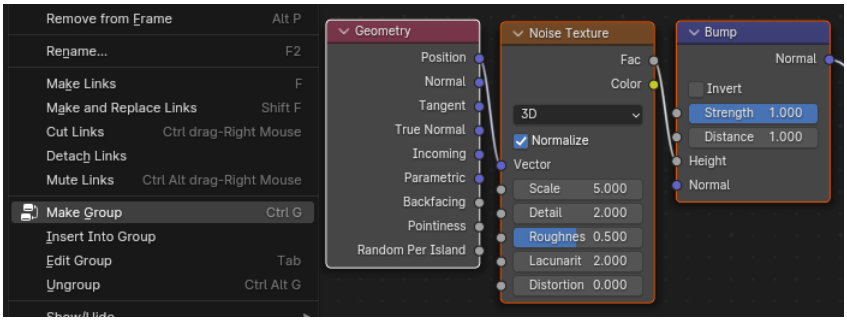
菱形のソケットは、各要素ごとにバラバラの値を受け取れますが、単一の値でも受け取ることが出来ます。
菱形の中に黒いポチがあるソケットは、単一の値を受け取っている状態になっています。
上の図の Random Value の Min や Max のソケットや、さらに上の図の何もつながっていない Offset ソケットには黒ポチがあります。
これは、(0,0,0)という単一の値を使ってることを示しています。

Positin のソケットのようにランダムな（頂点毎にバラバラの）値を繋いだ時には、黒ポチのない普通の菱形ソケットになります。
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や見た目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。

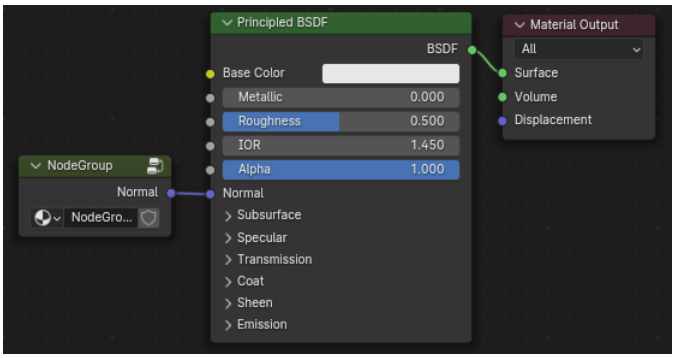
ノードグループとパネル

Blender 4.0 になって Inputs と Outputs の設定パネルは Interface パネルに整理されました。
単に2つのパネルが1つになっただけではなく、新しい機能が追加されています。
パラメーターが多くなりすぎるたばあいに、まとめて開いたり閉じたりするパネルを作成する機能です。



ノードグループ自体は、ジオメトリノードが導入される以前からノード編集で使うことが出来た機能です。
例えば、シェーダーのノード編集の際にノードが増えすぎて、幾つかのノードをまとめるときに使いました。
複数のノードを選択して、メニュー選択から Node - Make Group (Ctrl-G) でノードグループを作成ができます。

Noise Texture と Bump ノードをを使ってランダムに凹凸を表現する部分を、
一つのノードグループにまとめます。

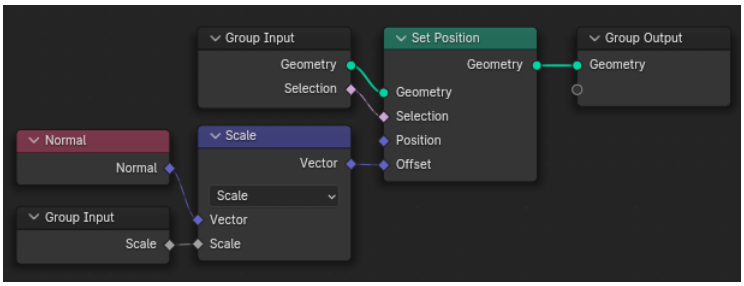


3つあったノードが１つのノードグループ(NodeGroup)にまとまってすっきりしました。

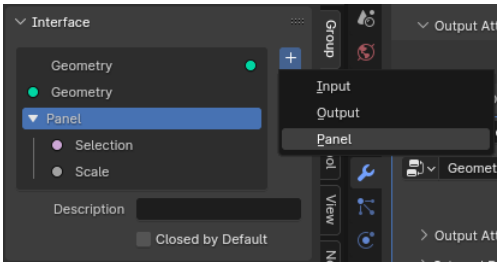
こうして凸凹を表現するノードグループの Normal ソケットを、いつもの Principled BSDF につなげば凸凹な表面を作れます…が、
おや、Blender 3.6 までで見慣れた 長大な Principled BSDF のパラメーター群が、すっきりとカテゴリー分けされています。

Subsurface や Specular などのカテゴリ分けされたパネルは、開いたり閉じたりできるようになっています。

この、ノードのソケットをパネルにカテゴリ分けする機能は、Blender 4.0 で導入された新機能です。
特に目立つのが（使用頻度の非常に高い)シェーダーの Principled BSDF だったので、例として最初に紹介しました。
ノードグループでも、このカテゴリ分けしたパネルを作る機能が使えるようになっています。

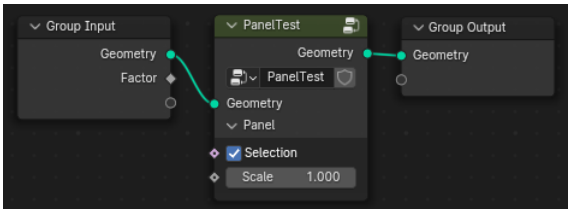


ジオメトリノード用のノードグループを作成してみます。
法線方向に頂点を移動して膨らませる機能の単純なノードグループですが、Geometry に加えて、Selection や Scale など複数のソケットを、Group Input に接続しました。



インターフェイスパネルには、接続した Input 用のソケット(Selction, Scale)が追加されています。
さらに「＋」ボタンを使いパネルを追加して、Selection と Scale をその下に配置しました。

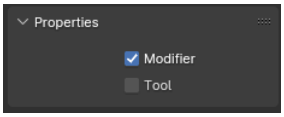
ソケットを移動して配置する手段は、Blender 4.0 時点でマウスドラッグしかないので注意します。
※)マウスドラッグで直感的に操作できる反面、以前あった上下に移動させるボタンは無くなってしまいました



Panel という名前のパネルの中に、Selection と Scale のソケットが収容されました。
クリックでパネルを開いたり閉じたりができるようになっています。

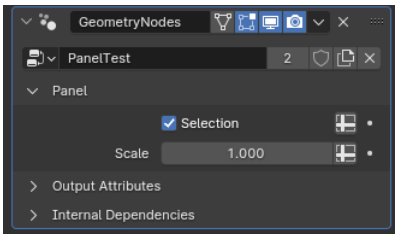
このノードグループを、PanelTest という名前に変更をしてみました。

ところで、こうして作成した「ジオメトリノードの中で使用するためのノードグループ」ですが、実は**「ジオメトリノードそのもの」**としても利用ができます。



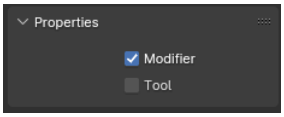
ノードグループの編集画面に入って、右側 Properties(プロパティ) パネルの Modifier にチェックを入れます。
※）この時「ジオメトリノード編集中」なのか、その中の「ノードグループ編集中」なのか間違えないようにしましょう

これで、PanelTest という名前で作ったノードグループ自体を、ジオメトリノードとして利用ができるようになります。



Selection と Scale を入力パラメーターに使えるジオメトリノードとして利用することもできるようになりました。

Blender 4.0 までは、モディファイアの中のパラメーターではパネルによるカテゴリー分けが実装されていませでした。
Blender 4.1 でモディファイアでのカテゴリー分けも実装されています。
これで、多数のパラメータを使うようなノードも管理がしやすくなっています。



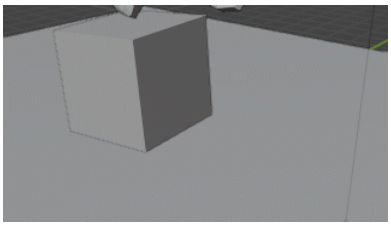
さて、先ほどのProperties(プロパティ) パネルですが、もう一つ気になる Tool という項目があります。
こちらは、Blender 4.0 のジオメトリノードの機能更新のもう一つの目玉の ジオメトリノードツールに関係する項目です。
後のセクションでじっくり確認してみましょう。

シミュレーションノード

Blender 3.6 におけるジオメトリノード強化の目玉は何と言ってもシミュレーションノードの追加でした。

従来のジオメトリノードの機能の基本は、毎フレーム何らかの計算や処理をして、その結果を表示するというものです。
例えばある時間フレームでメッシュの変形を行ったとして、次のフレームで「その変形をした結果を元に」さらに変形をするということはできませんでした。
シミュレーションノードの追加によって、そのような処理が可能になっています。
つまり
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…
というようなことが可能になります。

こうしたシミュレーションの例として見た目に分かりやすいのは剛体シミュレーションでしょう。



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、
ということを繰り返し計算することで、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)

この剛体シミュレーションの機能は、ジオメトリノードの機能ではありませんが、理屈は一緒です。
「前のフレームでの情報」を使うことで、様々な時間進化するシミュレーションの計算を行うことが出来ます。

さて、シミュレーションノードを使った作例は無数に考えられますから、シミュレーションノードだけ専用の別冊として本にしました。
ここでは、シミュレーションノードの基本の組み方と、ごく単純な作例、やや複雑な作例の２つだけ紹介をすることにします。

シミュレーション用のノードの組み方

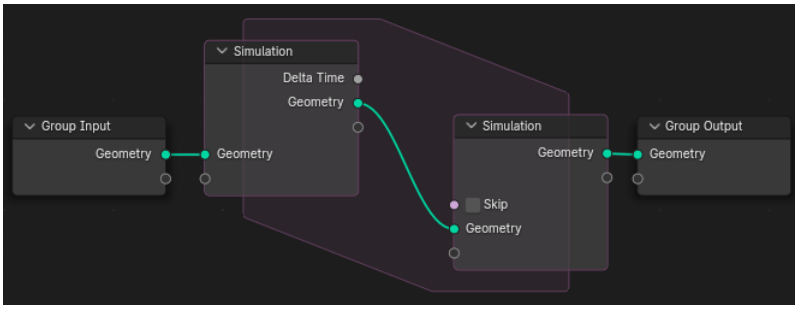
まずは、とても単純なシミュレーションノードを作成して、基本的な機能を確認してみましょう。

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、シミュレーション計算の本体を担います。
※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。

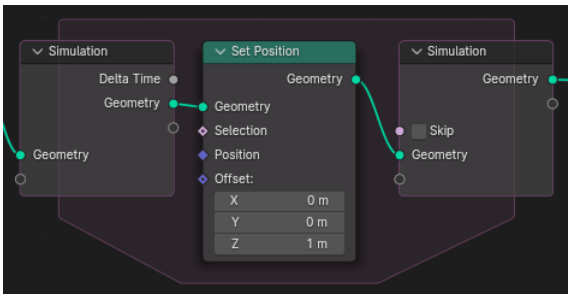
ジオメトリノードの構成の基本は、Group Input と Group Output の間にはさまれるようにノード群を配置することでした。
そこで、Group Input と Group Output の間に Simulation Zone を配置してみます。
この状態が「何もしない」シミュレーションを実行するノード組みになります。



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。

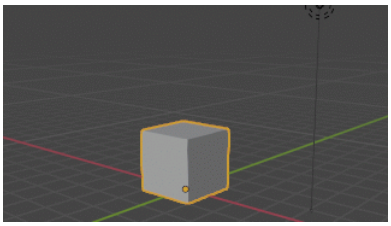
通常のジオメトリノードとの違いは、デフォルトキューブの情報は
Simulation Output のノードに到達した時点で一時的なメモリにも保存されるというところです。

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。

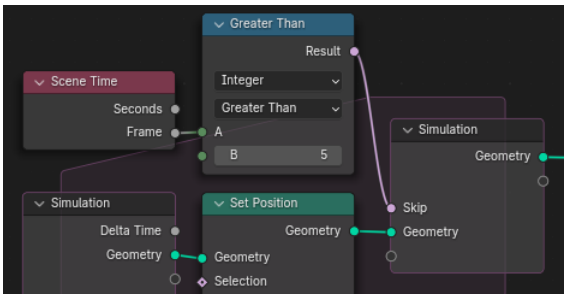


時間を進めていけば毎フレーム頂点が 1 ずつ上に移動していきます。
つまりこれで、単純な移動のシミュレーションが実現できたことになります。
動画)Sim01.gif(pdfでは先頭のコマのみ表示されています)

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。

最初のフレームでも、シミュレーションの内容が１ステップ分だけ実行されていることに気を付けます。
初期条件として設定した(Group Inputで入力される)頂点位置そのままではなく、１だけ上に移動した位置に頂点が表示されています。

Blender 4.0 では、新たに出力側の Simulation ノードに Skip(スキップ) というソケットが追加されています。



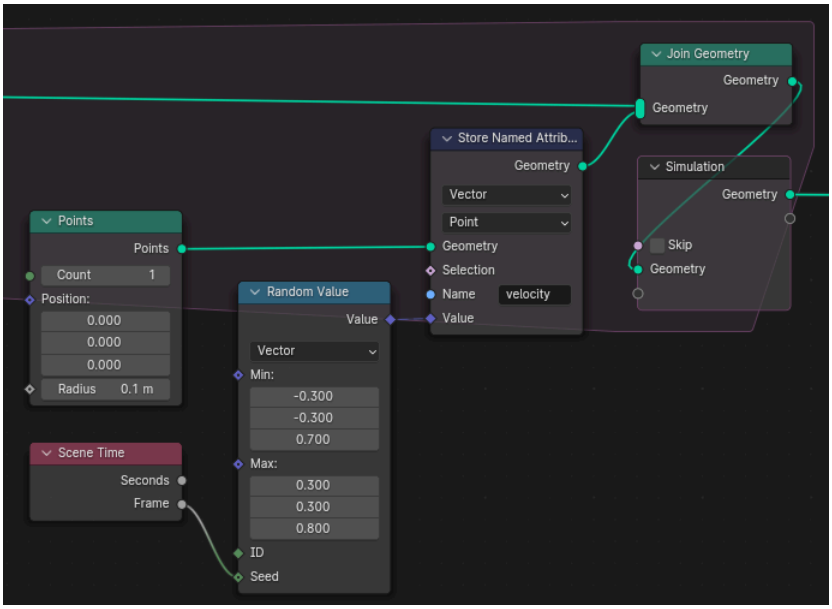
名前から予想できるように、このソケットを True を渡すと、
シミュレーションを実行せずにスキップして、そのままの状態で次のフレームにジオメトリを渡すことになります。

例えば、このように Frame が 5 より大きい(つまり6以上で) Skip するようにノードを組めば、
5フレーム目まではシミュレーションが実行されてキューブが動いていきますが、
6フレーム目以降はピタッと止まることになります。

単純に「編集中は余計な計算をしないようにシミュレーションを無効にする」という使い方もできますし、
このように条件を付けることで、途中でピタッと止まるような使い方をしても良いわけです。

少し複雑なシミュレーションノード

さて、シミュレーションノードの解説本であれば、徐々に複雑なノード組みへと進んでいくのですが、今回は一気に少し応用めいたノード組みへと進みます。
とはいっても、Vol.1 の本から順番に読み進めてくれた読者に皆さんなら、これぐらいの複雑さであればすんなりと理解できると思います。
パーティクルシミュレーションのように、パーティクルを噴出させて重力で放物線を描いて落下するようなシミュレーションをしてみます。

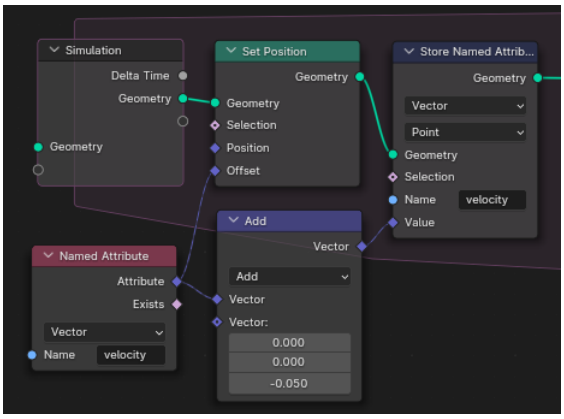


「パーティクル」と呼びましたが、その正体は単なる頂点です。
まず、パーティクル（頂点）を毎フレーム発生させる部分のノード構成を見てみます。

Points(ポイント) ノードによって毎フレームごとに新しく１つのパーティクル(頂点)を発生させます。
Join Geometry を使うことで、今までに発生させたパーティクルをそのまま保持するとともに、新しく発生させたパーティクルを（原点に）追加することになります。

パーティクル（頂点）には、速度情報を持たせます。
Random Value を使ってランダムな速度情報の範囲を決めます。

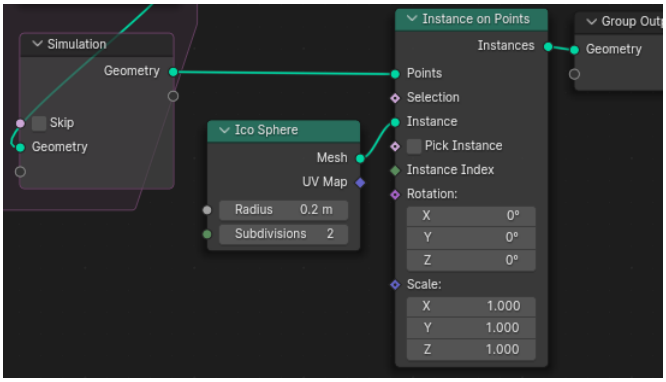
ところで、Pointsが毎フレーム発生させる頂点は１つだけです。
そのため、ID は最初は常に 0 になり、ランダムノードで発生するランダムは、シードが同じ場合常に(IDを元にした)同じ値になります。
そのため Seed 値には、Scene Time を使ってフレーム毎に別々のランダムが得られるように工夫します。
これをしないと、(IDが一緒なので)毎回同じ速度の粒子が発生してしまいます。
この IDが 0 の粒子は、Join Geometry で今までの粒子と一緒になる時に ID が再度割り振られることになります



発生したパーティクルはメッシュのジオメトリに追加されて、時間と共に数が増えていきます。
それらパーティクルを、Set Position ノードを使ってその速度に従って位置を動かしてやります。

それだけだと、単に直線状にまっすぐパーティクルが飛んでいくだけです。
Z軸の下方向に重力で引っ張られて速度が徐々に変化していくように、Store Named Attribute で velocity を上書きしてやります。
これで、全ての粒子は放物線を描いて落下していくようになります。

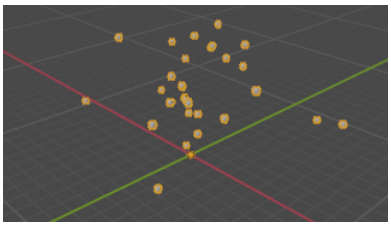
独立した頂点が存在しているだけだと、レンダリングして見ることができませんから、
最後に Instance on Points(ポイントにインスタンス)を使って Ico 球などを配置します。



この時、あくまでシミュレーションのゾーン内では 単なる複数のパーティクル(頂点)として処理だけを行い、
Simulaition Zone の外でインスタンス化しましょう。

Simulation Zone の中でインスタンス化した場合は、生成したインスタンスが次の時間フレームに引き継がれます。
すると引き継がれた Ico 球の頂点でさらに インスタンスを発生させたり…と、かなり困ったことが起こります。

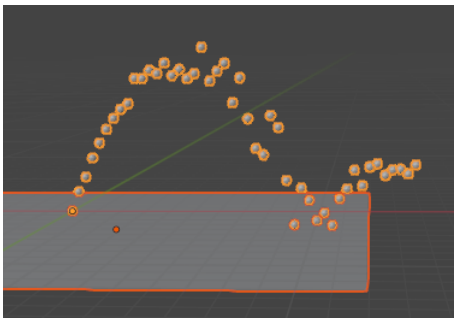
⚠ うっかりすると猛烈にインスタンスが発生してメモリを使いつぶすので要注意です。



原点から噴き出すパーティクルのシミュレーションが出来ました。
動画)ParticleEmittor01.gif(pdfでは先頭のコマのみ表示されています)

シミュレーションとしては比較的単純なものですが、速度と加速度の考え方を使ったり、毎フレーム頂点が増えたりと、シミュレーションノードで頻出するテクニックを使ったノード組みです。
このサンプルは、01_SIMBASIC/01_ParticleEmitter01.blend として同封しました。

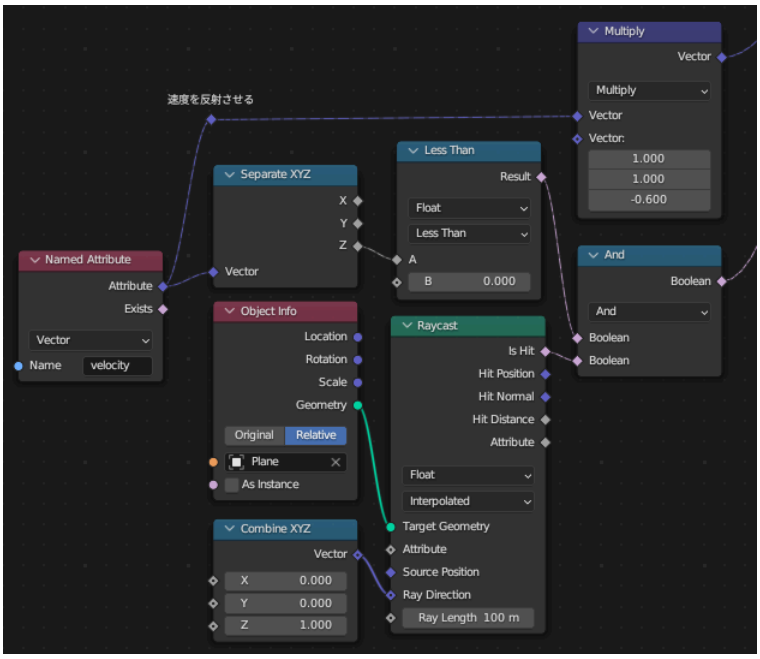
もう少し複雑なシミュレーション例として、衝突判定をして粒子の跳ね返りを表現してみます。



発生した粒子を（発生源の少し下に置いた）平面で跳ね返るようにします。

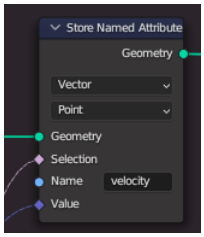
粒子はどのように噴き出しても良いのですが、静止画で跳ね返りがよくわかるように、
ランダムの範囲を調整して、ほぼ同じ方向に吹き出すように調整をしました。
これなら、横から見た静止画でも、跳ね返っている様子がよく分かります。

衝突判定の方法としては、
Geometry - Sample - Geometry Proximity(ジオメトリ近接) を使って対象のポリゴンまでの距離を測る方法があります。
ポリゴンまで一定の距離まで近づいたら衝突したとみなすのですが、
速度次第で衝突判定をすり抜けたりする危険があります。

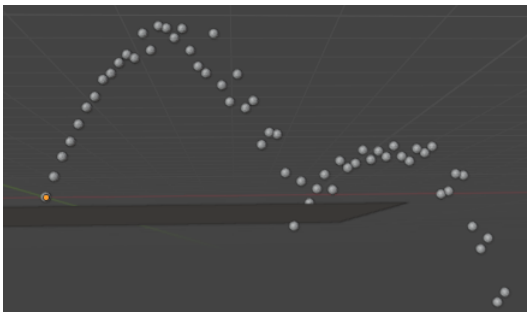


今回の例では、Geometry - Sample - Raycast (レイキャスト)をうまく利用して衝突判定してみます。
粒子の上方向にレイを飛ばし、衝突が起こったならば、粒子が平面の下側に潜り込んでいるという事ですから、この判定を衝突判定に使います。

衝突した粒子は速度を変化させて、跳ね返りを表現します。
しかし跳ね返った粒子が次のステップで平面の上に出なければ、次の時間フレームで即再衝突（誤判定）になってしまいます。
粒子の速度も同時に調べて、下向きの速度を持っている粒子だけが板に衝突したものとして、z方向の速度を -0.6 倍するようにしました。



衝突判定で True だった粒子の速度は、 Store Named Attribute で再度書き換えます。



板の下側に回り込んだ状態になって、はじめて衝突と判定して速度を変えるので、
その瞬間は板をすり抜けて下側に存在することになります。

通常はあまり気にならないですが…見た目の厳密さが必要な時には工夫が要ります。
簡単に誤魔化すならば、衝突判定にマージンを取るなどが選択肢になるでしょうか。
Raycast で衝突を判定する位置を、本来の粒子の位置から少し下にずらす等が考えられます。
(Source Position のソケットに、本来の粒子の位置からすこしずれた位置を与えることになります)

このサンプルは、01_SIMBASIC/02_ParticleEmitter02.blend として同封しました。

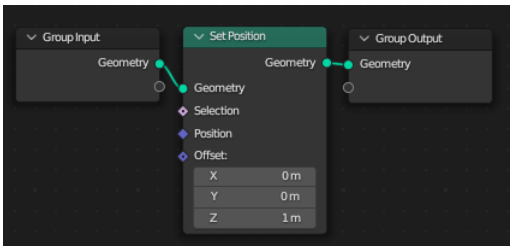
シミュレーションノードを使った様々な例などは、シミュレーションノードの作例本に譲るとして、本書では Blender 4.0 での機能追加の目玉であるループの処理に進むことにしましょう。

ループ処理(Serial Loop)

ジオメトリノード の Blender 4.0 での強化の目玉はループ(Loop)処理の導入です。
部分的に、Blender 3.6 で導入されたシミュレーションノードと似ている部分もありますが、それとはまた違った使い方をすることが出来ます。

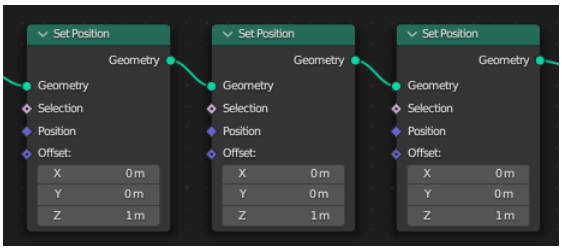
ループという用語を使いましたが、ノードの名前としては Repeat(繰り返し)という用語が使われています。
どちらの用語でも意味は通じますが、処理の名前としてはループの方が一般的に思えるので、ここでは用語としてループを使うことにします。
ジオメトリノードのループについて英語の文章などを見ると、シリアルループ(Serial Loop)という使い方を良く目にします。
シリアル(Serial)には「順次に」「順番に」というような意味があり、1回目の処理が終わったら次に２回目の処理を、2回目の処理を終わったら３回目の処理をする…というように順番に処理をするニュアンスが込められています。

シリアルループと対になる用語としてはパラレル(Parallel)ループがあります。
こちらは、パラレル(平行して)という意味合いから、処理１、処理２、処理3…を同時に行うような意味合いになります。



例えば、普通に「各頂点の位置を Set Position ノードで動かす」というような処理は、
各頂点に対して、パラレルなループ処理をしたこと考えることができます。

マルチスレッドで処理をしていなければ、実際にはもちろん順番に処理をしているのですが…、
考え方としては各頂点毎に(0,0,1)動かすという処理をしていて、それぞれの頂点どうしは関係がなく独立に処理をしているので、
パラレルループ処理をしていると考えることができます。



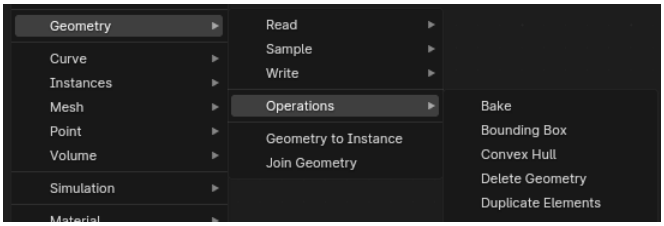
Set Position を3つ並べれば、3回シリアルループ処理をしたと考えることが出来ます。
1回目の処理の結果に対して、２回目の処理を行い、その結果に対して３回目の処理を行うというわけです。

そうした処理を100回行いたければ、今まではノードを100個並べる必要があったのですが…
ループ処理用のノードが実装されたことで、少ないノードを使ってそうした処理をすることができるようになりました。

※) 100回の処理をループ以外で行いたければ、本当はノードを100個並べる代わりに、
10回処理をするノードグループを作って、それを10個並べるという手もあります。
とはいえ、いずれにせよ 100回は 処理するノードを通過するようなノード組みをする必要があるわけですね。

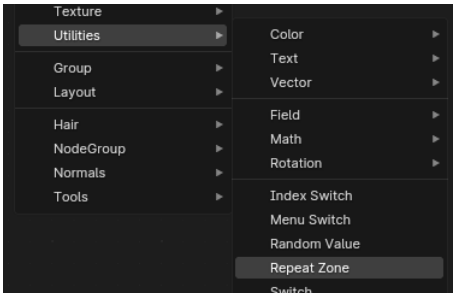
オペレーションのループ

実際にループ用のノード組みを作ってみましょう。
一番分かりやすいループの使い方は、何かの操作を繰り返し行うという使い方です。

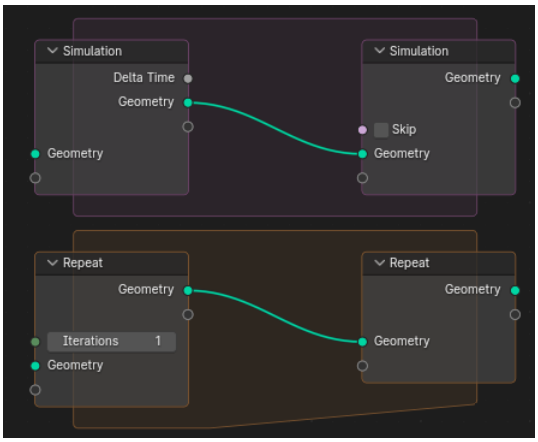


「操作を繰り返す」という意味で、オペレーションという言葉をタイトルに使いました。
実際、オペレーションのカテゴリーにそういった「操作」のノードが分類されていますが、
それら以外でも、単に「頂点を動かす」といった操作も含めてここではオペレーションと呼んでいます。

ということで、頂点を動かすだけの簡単なループ構造を作成して見ましょう。

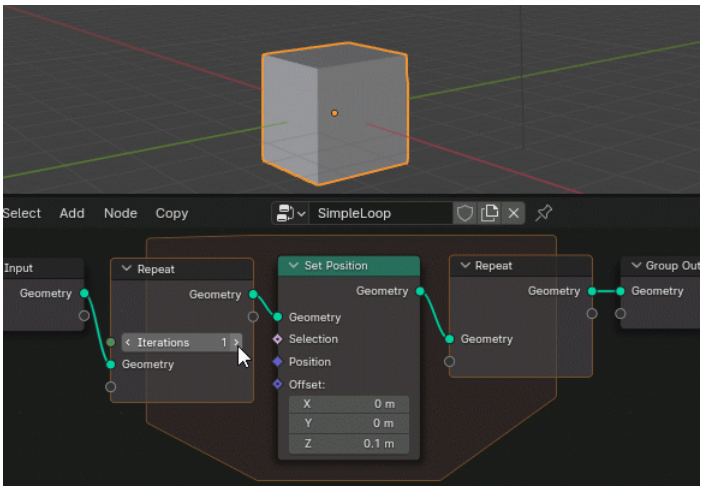


追加メニューの Utility(ユーティリティ)から Repeat Zone を選択すると、
ループ処理用のフレーム枠が作成されます。



Repeat ノードのペアと、くすんだオレンジ色のフレームが作成されます。
見覚えがあるようなフレーム枠ですね。
使っている色は違いますが、ジオメトリノードでシミュレーションをするときの Simulation Zone とかなり似ています。

シミュレーションの時には、Delta Time というソケットがありました。
Repeat の入力ノードには Delta Time の代わりに、Iterations という入力ソケットがあります。
これが、何回ループを行うかという回数を指定する欄になります。
初期状態では１ですから、１回だけフレームで囲まれた中の処理を行うことになるわけです。



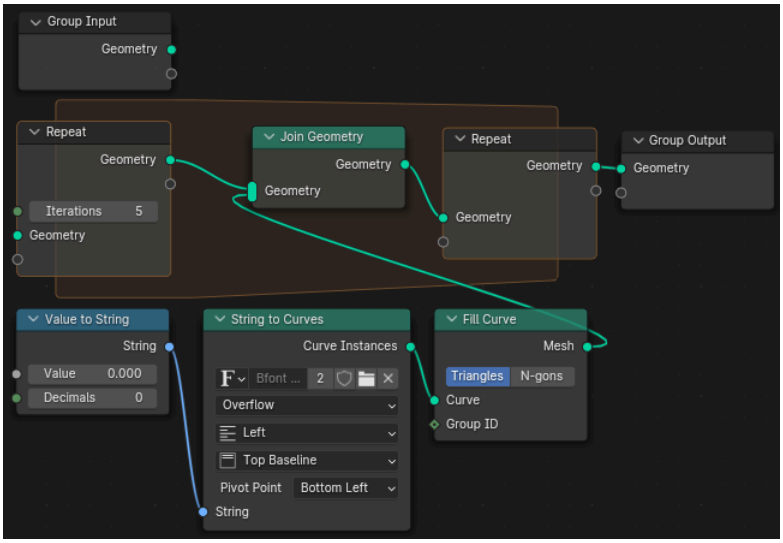
Group Input と Group Output の中に Repeat Zone を挟み込みます。
この辺のノードの組み方は、シミュレーションノードと同じです。

Input と Output の中に、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。

シミュレーションノードの時には、時間フレームを進めるたびに処理が行われて、時間と共にアニメーションさせることが出来ました。
Repeat Zone では、時間の進みとは関係なく、指定の回数だけ処理を繰り返します。
今回の例では、Iterations に入力した回数だけ上方向に移動することになります。
動画)LoopBasic01.gif (pdfでは先頭のコマのみ表示されています)

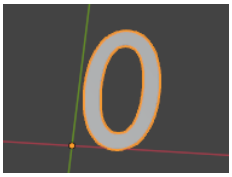
ループのカウンター

シミュレーションノードで、毎フレーム粒子を発生させるパーティクルのような処理をするときに、Join Geometry(ジオメトリ統合) ノードを使いました。
それによって「発生させた頂点を元のジオメトリに統合するという処理を毎時間フレームで行う」という理屈で、頂点数を増やしています。
同じように、ループ内で Join Geometry を使うことで、複数の形状を生成することが出来ます。

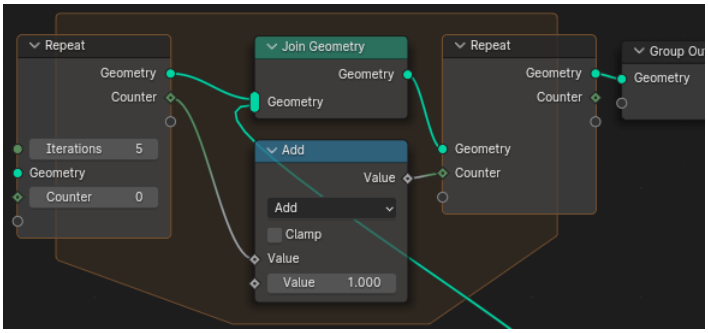


そこで、Value to String(値の文字列化) から String to Curves(文字列のカーブ化) のコンボで、複数の文字を作成してみます。
元々のジオメトリは使わないので、Group Input からの Geometry ソケットはどこにもつないでいません。

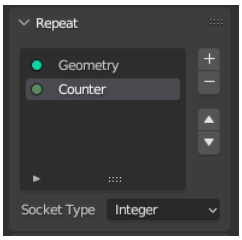
Iterations を5に設定すれば、5回文字列を発生されて、5個のテキストオブジェクトを配置することができるわけですが…



この状態だと、当然ながら同じテキストが同じ場所(原点)に５個重なるだけです。
[1, n] までの数字を表示したいような場合には、今何回目のループ処理なのかを知るための
カウンターの役をするパラメーターが必要です。

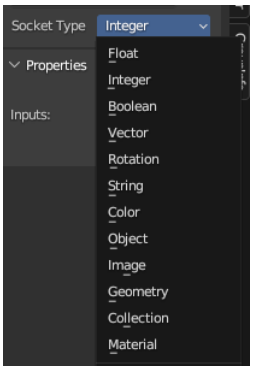


Repeat Zone 内で使えるパラメーターを増やすことが出来ます。
この例では、Counter という名前の整数(Integer)のパラメーターを追加しました。
Repeat の入力ノードのソケットから Add ノードにつないで１を足して
Repeat Output のソケットにつなぎます。
これによって、最初は 0 だった Counter が、処理するたびに１増えることになり、結果として今何回目の処理なのかを知るためのカウンターとして使えることになります。



パラメーターの追加は、編集画面右の Node タブにある Repeat パネルから行います。
パネルの「+」「-」ボタンでパラメーターを増減できます。

直接編集する以外にも、Repeat の入出力のノードの空きソケット部分にラインを繋げてパラメーターを増やすことも可能です。

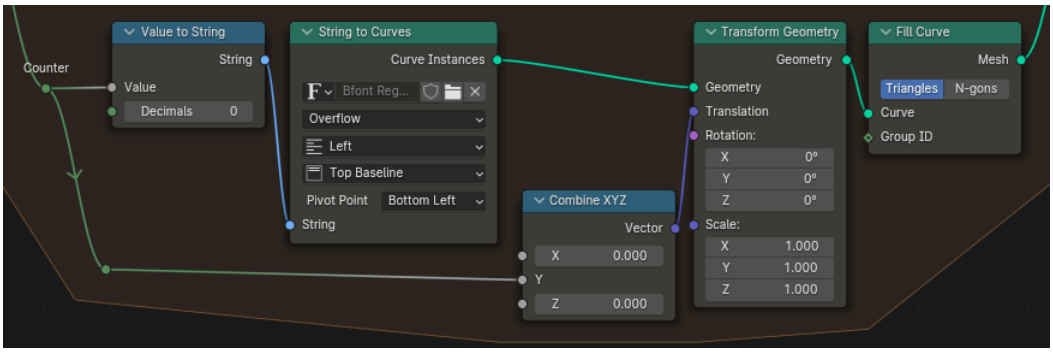


単純に「+」ボタンを押しただけだと、最初にあるソケットの Geometry を引き継いで緑のジオメトリ用のソケットになります。
使うパラメーターのタイプは、きちんと設定しておきます。

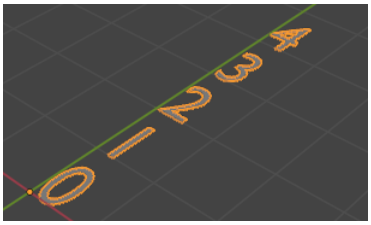
今回は整数のカウンターとして使うので、Integer(整数)を使います。
整数ではなく小数の Float でも実質的には問題ありませんが…、一応こういった変数の型はきちんと設定する習慣があった方が良いですね。

Blender 4.1 の時点では、ベクトルやマテリアル、文字列などノードで使用できるタイプは全て選択できます。
（マテリアル自体などをパラメーターとして使うような特殊な用途は、そうそう無いと思いますが…）

こうして作成したカウンター用のパラメーターは、文字の値を指定したり位置を調整するために使うことが出来ます。

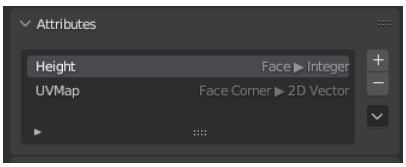


Value to String につないで文字の作成に、
Transform Geometry につないで移動に変数 Counter を利用しました。

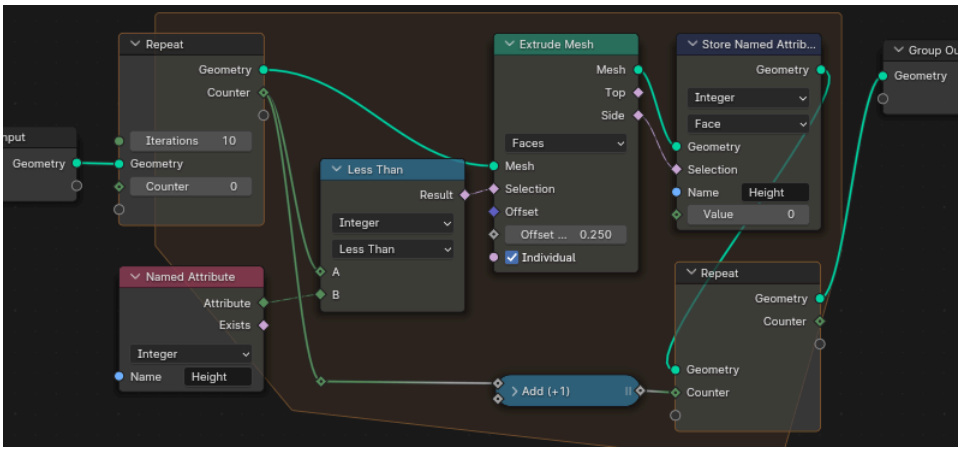


文字列を変更しながら並べることが出来ました。
もちろん100でも200でも値を並べることができます。
ループ処理が無ければ、テキストを作るノードを欲しい数だけ並べるしかなかったのですが、随分と手軽になっています。
この例は、02_LOOP/01_ShowCounter.blend として同封しました。

もう一つ、カウンター用のパラメーターを利用した作例を見てみましょう。
平面を分割したグリッドから、Extrude Mesh(メッシュ押し出し) を指定の回数だけ繰り返すノードを組んでみます。
地面から、にょきにょきと建物（…といっても只の四角ですが）を生やすような処理です。

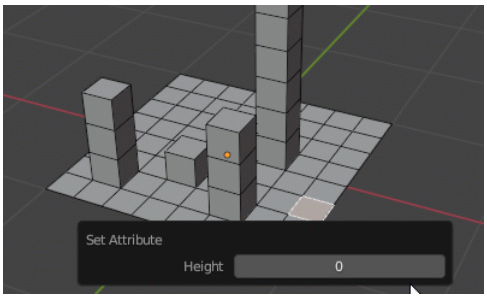


まず、処理をしたいメッシュオブジェクトに汎用アトリビュートを持たせることにしてみます。
平面のグリッドを押し出すと、その回数分だけ高くなるので、Height としました。
平面押し出しの回数として使用するので、Face が持つアトリビュートとして設定します。



カウンターの値が、各面で設定した Height よりも小さい場合だけ Extrude Mesh を実行するようにします。
そうすれば、Height が 1 の面では1回、 Height が 2 の面では2回と押し出しが実行されます。

但し、押し出しで作成された面の側面も Height の値を引き継ぎます。
つまり、上に押し出しを実行すると、次は側面から横向きにも押し出しが実行されてしまいます。
それを防ぐために、押し出しで作られた側面(Side)の面では、Height を 0 にするように Store Named Attribute を追加しました。

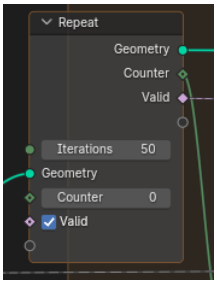


Set Attribute 等を使って、面の持つ Height の値を編集すれば、それに従ってニョキニョキと立方体が生えてきます。
先ほど、Iteration を 10 に設定したのでループは10回実行されます。
最大で10階までの高さになります。
もちろん、Iteration を増やせば、もっと高く伸ばすこともできるわけです。
動画)ExtrudeMesh01.gif (pdfでは先頭のコマのみ表示されています)

このサンプルは、02_LOOP/02_ExtrudeCounter.blend として同封しました。

ループとフラクタル

この「ニョキニョキと押し出す」ノード組みをもう少し発展させてみます。
先ほど面ごとに Height という高さ情報を持たせましたが、今度は「一定の確率で成長を打ち切る」という仕組みで高さを決めてみます。
この仕組みの場合は Height の代わりに「打ち切られたかどうか」という情報を面が持っていないといけません。
Valid という名前のパラメーターに持たせる事にしてみます。(Valid は「有効」という意味を持つ単語で、有効かどうかの判定ということでこの名前にしました)

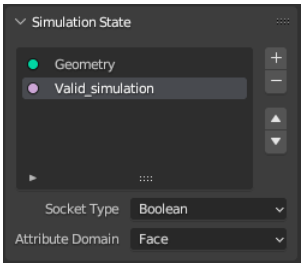


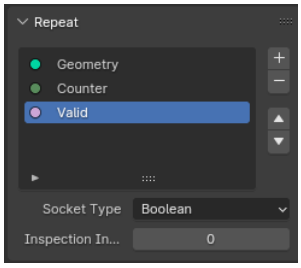
このパラメーターの持たせ方は３つ方法があります。
１．元から Attribute としてメッシュに持たせておく
２．Store Named Attribute を使って追加でメッシュに持たせる
３．Repeat Zone 内のパラメーターとして持たせる
今回は、３番の方法にしてみます。
Counter と同じようにして、Valid という名前のパラメーターを Repeat の入出力ノードで設定をします。

Counter の時は、全体で１つのパラメーターでした。
各頂点/面などがバラバラに Counter というパラメータを持てる、というわけではありません。
入力の Input ノードの右側のソケットをみると、Counter のソケットは菱形のソケットの中に点が打ってあり、確かに菱形のソケットとは区別がされています。

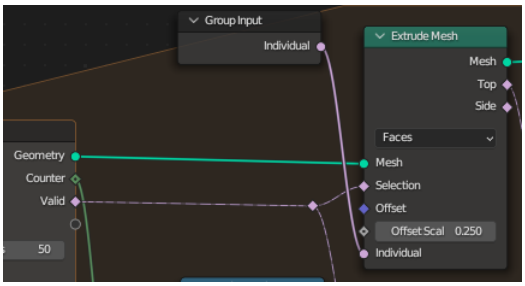
※ 菱形なので「各頂点が値を持つタイプ」にもなれるのですが、必要がないので全体で１つのパラメーターとして使われていることが分かります。
この時丸ソケットを使うノード、例えばValue to String などにつなげると、もはや各頂点が値を持つタイプとしては使えないので丸ソケットに変化します）

一方 Valid のソケットは（この後、面に対して行う処理のノードに繋がるので）自動で各面が持っているパラメーターと判定されて、菱形のソケットになっています。



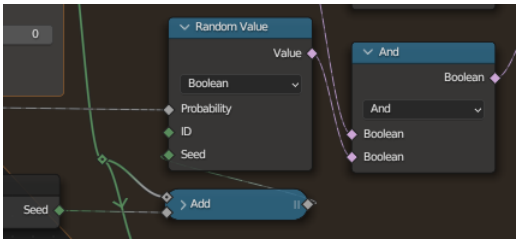


Blender 4.1 Alpha の時点では、この Valid というパラメーターが 各面/辺/点 のどれが担っているかという指定
つまり、ドメインタイプの指定は明示的に行えず、繋がっているノードのタイプから自動で判断されるようです。
Simulation Zone の時には、Attribute Domain の項目があり手動で指定できたのですが、ループ処理と少し違いがあるようです。



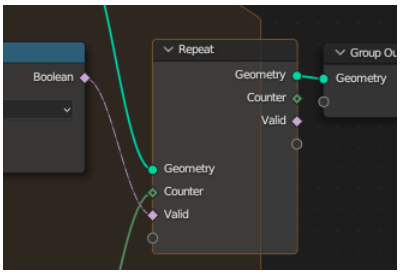
Valid のパラメーターが有効な面でだけ Extrude Mesh で押し出しをします。
Individual などのオプションは、モディファイアのパネルから調整できるように、Group Input につないでみました。

Random Value を使って、一定の割合で True だった Valid パラメーターが　False になるように処理をします。



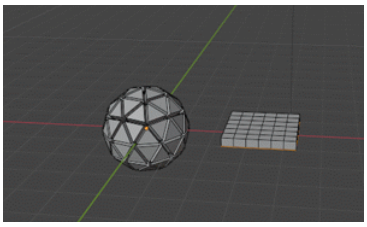
ブーリアン演算の And を利用しました。

この時、ID 値が一緒の面は何回目の処理でも同じランダムになってしまうので、シード値の方を変化させないといけません。
シード値に Counter を利用します。
また、モディファイアからもシード値をコントロールできると良いので、2つの値を足し合わせて使うことにしました。



一定の割合で False になった Valid の値は、次のループの処理で使うために
Repeat Output のソケットにつなぐことになるわけです。

先ほどの場合と同じように、Extrude したメッシュの横から成長が始まらないように、
Exturude Mesh の Top のソケットを And で利用することになります。

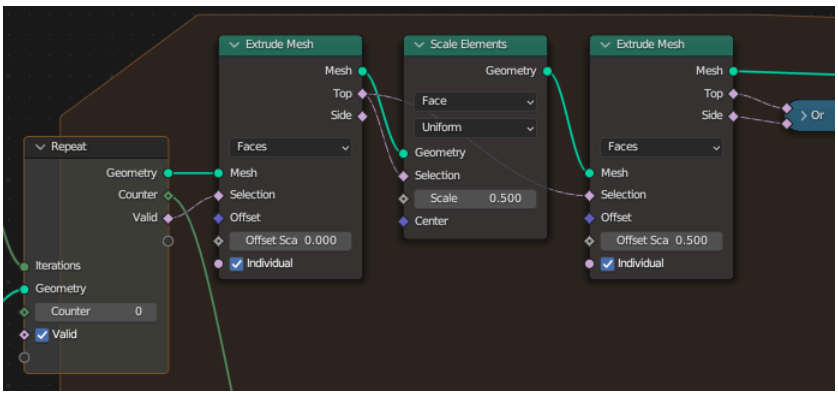


Icosphere やグリッド状のメッシュから Extrude Mesh の繰り返しで、
このような形状になります。
Valid が False になる確率や、そのランダムシードを変更させれば、このような動きになります。
動画)RandomExtrude01.gif (pdfでは先頭のコマのみ表示されています)

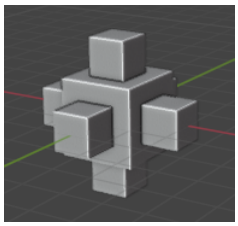
ランダムの出現順が面のIDとループのカウンター依存であり、完全にランダムというわけではないので、シード値を変えた時の挙動が少し特徴的ですね。

この例は、02_LOOP/03_RandomExtrude.blend として同封をしました。

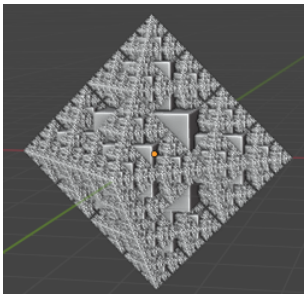
今回の例では Extrude Mesh した面の Top の面でだけ処理を繰り返すので、棒が伸びるような成長をしています。
形状の成長をコントロールしながら、側面からも Extrude Mesh していくようにすると、フラクタル状の構造が作れます。



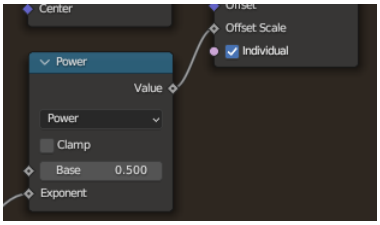
一度、押し出しの大きさを 0 にして面の押出をして、
押し出した面の Top を縮小、そして再度の押し出しをすると、
このように面の一部が飛び出る構造が作れます。



増えた面(つまり Top と Side の Or を取った面)でさらに押出をしていくことで、繰り返しのフラクタル形状が作れます。

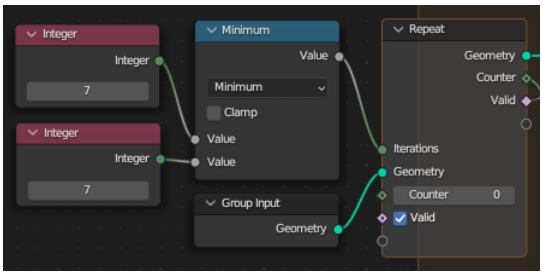


3段階と6段階押し出しを繰り返したループ処理で、このような形状になります。
自己相似形で細部の構造が出来ていくフラクタルになっていますが、
この時、押し出しの大きさは一定ではなく、処理の回数に従って小さくしていかないといけません。



Counter の数と数式のべき乗(Power)を利用して、ループが進むごとに押し出しのサイズを半分にしていきます。
今回は、Counter を利用しましたが、毎回1/2ずつ小さくなるようなパラメーターを作って利用しも良いですし、
処理する面の面積（の平方根）を使えば、カウンターの役目をするパラメーター無しでも適切な押し出し量を見積もることもできます。

ところで、こうしたフラクタル構造は最初のうち、数回のループで終わる時は良いのですが、ループが深くなるにつれて爆発的に頂点や面の数が増えていきます。
うっかりマウスを引っ掛けて、操作不能にしてしまったりメモリを使い切ってしまったりする危険があるので、安全装置を仕掛けておきたいところです。

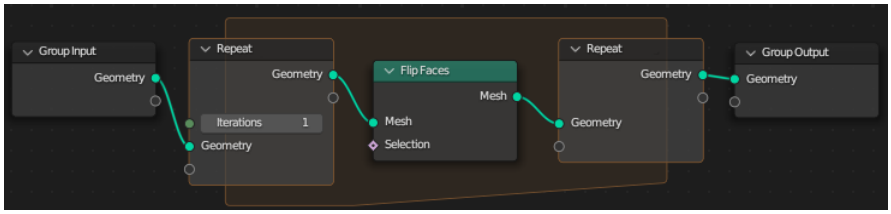


方法はいくつかあると思いますが、数式のMinimum(最小)と 2 つの入力を使うというのが簡単だと思います。
片方は、最大値として触れないでおいて、もう片方を操作するようにします。
今は 7 を設定しているので、片方を7以下にすればそれがループの回数になりますし、
うっかりマウス誤操作で片方を 100 とかにしても 7 までの制限がかかるので安心です。

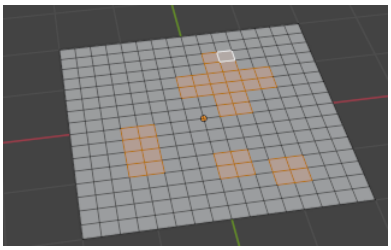
このサンプルは、02_LOOP/04_FractalExtrude.blend として同封しました。

計算や測定のみのループ

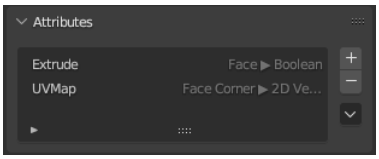
Repeat Zone のデフォルトの設定は Repeat の入力ノードから出力ノードにジオメトリのラインを結ぶものです。



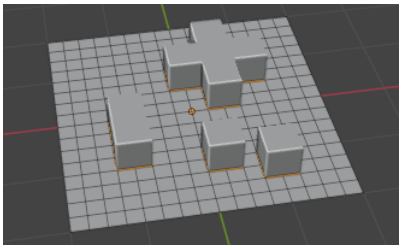
ですから、ループ処理はその間に何らかの Geometry を加工するノードを挟んで「加工を所定の回数繰り返すもの」と思ってしまいがちです。
しかし、ジオメトリのラインを結ばずに、何らかの計算や測定をするだけのループ処理を作ることもできます。



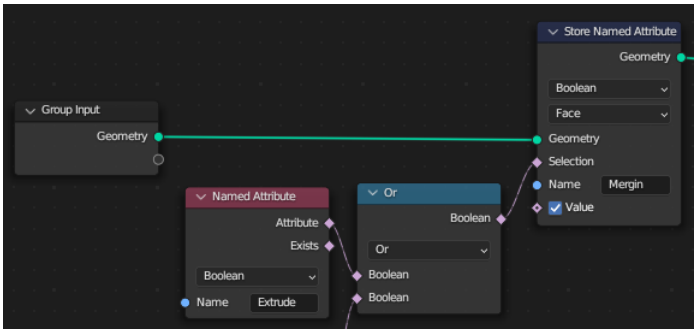
先ほどのメッシュ押し出しを繰り返す処理と似たセッティングをまた行ってみましょう。
グリッドに切った平面の一部を選択して、アトリビュートを設定します。
(見やすくするためにワイヤーフレーム表示を重ねています)



今回、1段階だけ Mesh Extrude をするので、Face の持つ Boolean のアトリビュートとして設定をします。
Extrude という名前にしました。

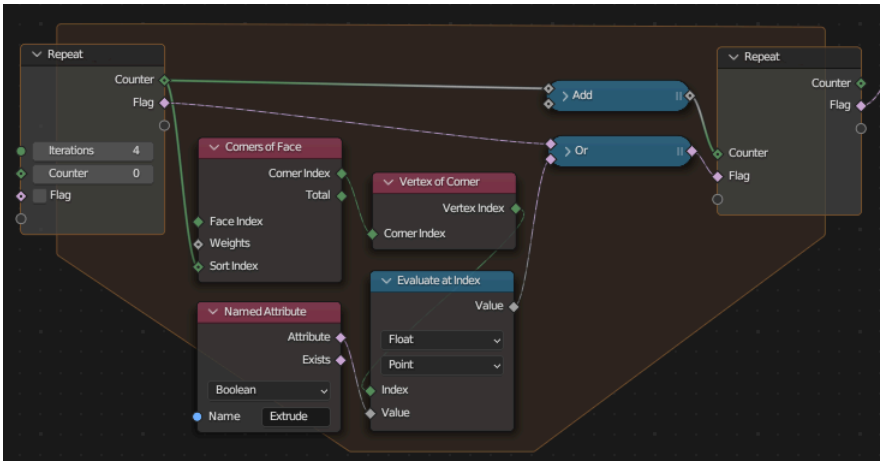


当然、Extrude が True な面を押し出しすれば、このような形状になります。
やけにシンプルですが、これは建物を表しているつもりです。
この周りに、一段高くなる歩道を配置してみます。



この建物の立っている Extrude 面に「隣接している面」に対して、Mergin という名前で属性を追加してみます。
この Mergin を利用して、建物の周りを押し出して一段高くしてみます。
Extrude に「隣接している面」とは、つまり「隣が Extrude である面」です。
この判定のためにループ処理を使います。

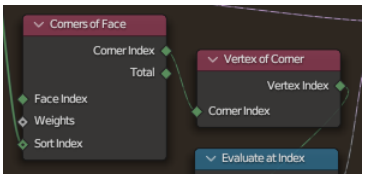
まず、Store Named Attribute を使って Selection のソケットで指定した面だけ Mergin が True になるようにノードを組みました。
ここに繋がる判定処理にループを作ります。



Repeat の入出力のノードの間に、Counter と Flag というパラメーターだけをつないだ Repeat Zone を作成します。
この Flag のソケットを、上の図の Store Named Attribute につなぎます。
（正確には、その手前の Or ノードにつなぎます）

Iterations を 4 に設定しました。
Repeat Zone 内のノード組みによって、各面ごとに計算される Flag というパラメーターを計算するために、
「各面ごとでそれぞれ」4回ループする計算が行われることになります。

では、具体的に Repeat Zone の中で何が計算されているのか確認してみましょう。

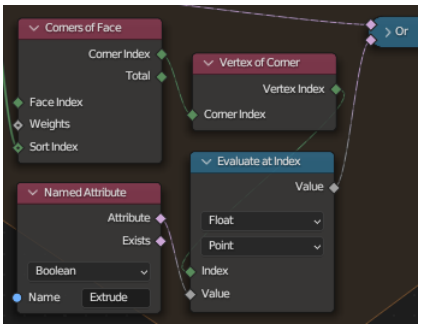


Counter のソケットから Corners of Face(面のコーナー) のノードの Sort Index(ソートインデックス) のソケットにラインがつながっています。
これによって、今計算している面の周囲の４つのコーナーの番号が分かります。

カウンターが 0 の時は一番若い番号のコーナー、カウンターが 1 の時は２番目に若い番号のコーナー…
というように、ループ処理１回ごとに1つのコーナーの番号が分かります。

この Corners of Face のノードは、使いこなすのが難しいノードです。
詳しい説明は、Vol.2 の本のトポロジーの章などで行っているのですが、Appendixに簡単な解説を載せておきます。

さてポリゴンの角である「コーナーの番号」が分かっただけだと、まだ頂点の番号などは分からないので、さらに Vertex of Corner(コーナーの頂点)ノードを繋ぐことで、ポリゴンの角の「頂点番号」が分かります。
「ポリゴンの角」と「頂点」が別物で、番号付けなども独自に行われているというのは、UVマップやカラーアトリビュート（旧頂点カラー）で出てくる重要な考え方なので、注意しておきたいところです。

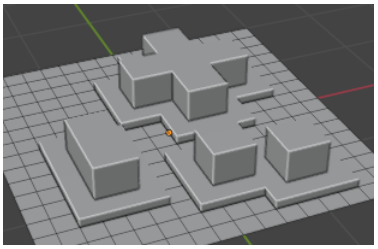


頂点の番号が分かったら、Evaluate at Index(インデックスでの評価)でその頂点のアトリビュートを得ることが出来ます。
Named Attribute を使って、アトリビュートの Extrude の情報を得ます。

この時、Extrude は面の持つ Boolean アトリビュートなので、それを頂点で評価した時にどうなるか、少し気になります。

面が持つアトリビュートを頂点で評価する場合は、基本的に平均値で評価されます。
ある頂点に接している4つの面が 1,0,0,0 と値を持っていたら、平均の0.25になります。
少なくとも　Float 型で、実数の値のアトリビュートの時にはそうなります。

しかし、今回 Extrude はブーリアンで False か True しか持たないので、False と True の平均をどう評価するのかは、すぐには分かりません。
このあたりの詳細が気になる人は、Appendixで少し詳しく述べているので確認をしてみましょう。
結論から言うと、今回のノードの組み方だと隣接する面のどれかが True であれば、True として評価をされます。
Flag のパラメーターは、周囲の4頂点でこの値を評価して、Or をとっているわけですから 「周囲の頂点のどれかに隣接する面の Extrude が True であれば True」になるというわけです。

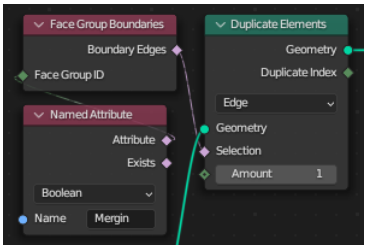


Mergin が True と評価された面を少し押し出します。

言葉にするとややこしかったですが、ある面の4隅の頂点で、それに隣接する面が1つでも Extrude になっているかどうかを判定したことになります。
4隅の頂点でそれぞれ判定するために、4回のループが必要だったわけです。
この判定で「隣接する面」が分かるので、建物の周囲が１ブロック分高くかさ上げされているような表現になりました。

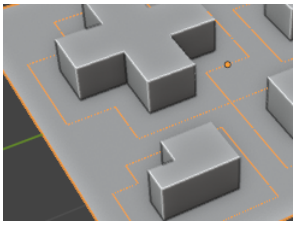
このサンプルを、02_LOOP/05_MakeMergin1.blend として同封しました。

折角ですから、もう少しだけ凝ったセットアップをしてみます。
今の例は、建物の周りに歩道があって少しかさ上げされている状況（のものすごいシンプルなもの）を模しているつもりです。
歩道の代わりに柵を巡らせてみます。



先ほど設定したアトリビュートの Mergin が True の平面と False の平面があります。
そうしたアトリビュートの値ごとに、境界のエッジを取り出すノードが
Face Group Boundaries(面グループ境界)です。

さらに、そうして(情報としてだけ)取り出したエッジ部分を実際に辺からなるジオメトリとして取り出す為に、
Duplicate Elements(要素コピー)を利用しました。

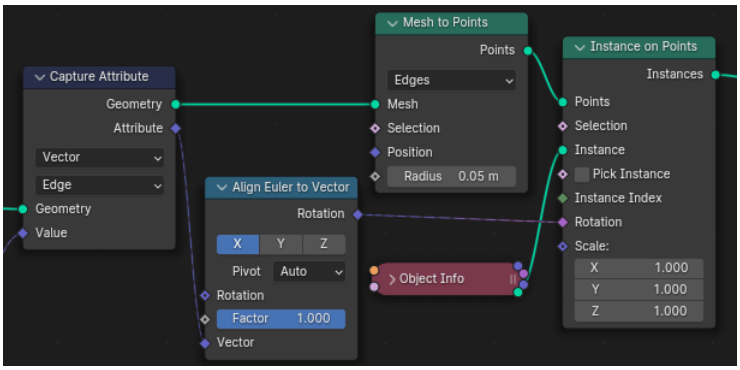


Join Geometry などを使って、元の形状と同時に表示をすれば、
建物部分を取り囲むように、辺が形成されているのが確認できます。

この辺に沿って、柵を作成します。
凝ったつくりをしようとすれば大変ですが、今回は非常に簡単に、
グリッドの長さに合わせたサイズで柵のパーツを作りました。

これを、辺ごとに、向きを合わせて配置をすれば柵になります。
一様なグリッド形状が元になっているので、単純に各辺の中心に柵を置けば良いことになります。
（パーツを繋ぎ合わせるだけなので、角の処理などは汚くなりますが…）

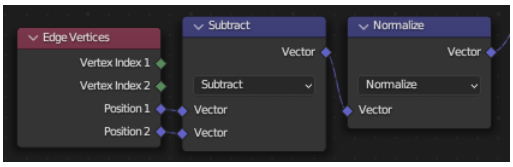
しかし、おなじみの Instance to Points(ポイントにインスタンス) は、辺ではなく頂点位置にインスタンスを置いてしまいます。



そこで、事前に Mesh to Points(メッシュのポイント化) を利用して「辺の中心位置に点を置く」変換をしておきます。

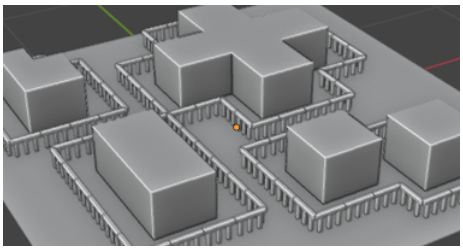
さて、問題は柵の向きです。
そのままだと縦横の区別がつかないので、（頂点に変換して情報を失ってしまう前に）
辺の向きを Capture Attribute(属性キャプチャ)ノードを使って保持しておくことにします。
そして、Align Euler to Vector ノードで、柵にエッジの向きに合わせた回転をさせるようにします。

Capture Attribute ノードにつなぐための辺の向きの情報は、１つのノードでは得られないので、組み合わせて計算します。



Edge Vertices ノードを使うと、辺の両端の２頂点の位置が分かるので、向きを知るに引き算をします。
長さ１の単位ベクトルにするために、さらに数式ノードの Normalize ノードも加えておきます。

※)実際には「長さ１のベクトル」でなくても向きは分かるので、Normalizeノードは必要無いのですが…
「向き情報として使うためのベクトル」を作る時は後の計算で「長かったり短かったりの長さ情報が悪さをしないように」長さ１にしておく習慣があった方が安心です。



建物の周辺に作った歩道部分が、今度は段差ではなくて柵として形成されました。
このサンプルは、02_LOOP/06_MakeMergin2.blend として同封しました。

ただし、見ての通り連続して繋がっていないパーツの集合なので、あまり美しくはありません。
例えばエッジで境界を作った後に、カーブ化をして連続した手すりを作り、長さに応じて支えを配置したりなど、
より凝った柵を作る方法などが考えられます。