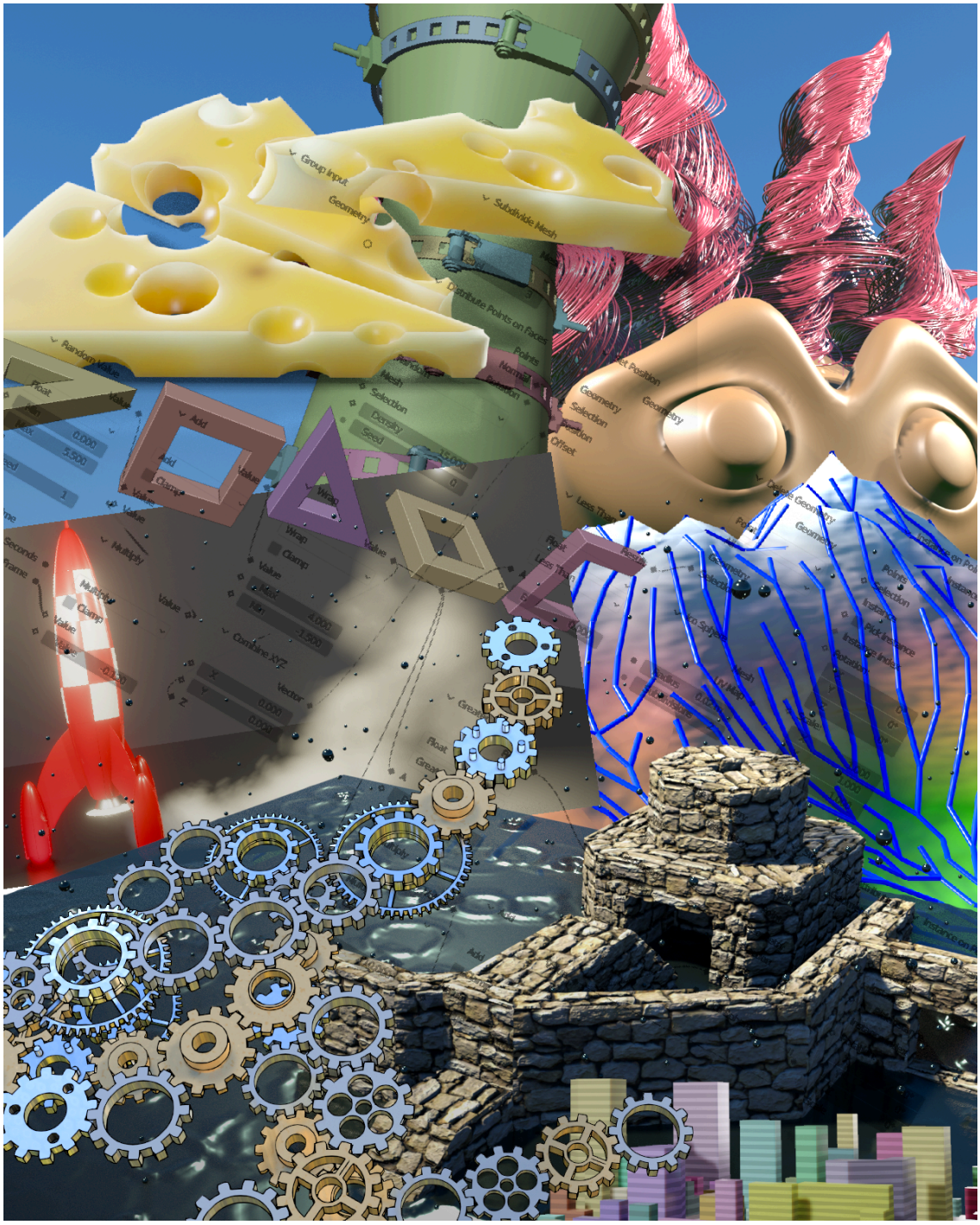
Blender ジオメトリノード 解説＆作例集 Vol.3

Geometry Nodes (for Blender 3.4 - 3.5)

Version 1.3

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. 3

Geometry Nodes (for Blender 3.4 and 3.5)

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

2023.04.01 ver. 1.0	2023.12.13 ver. 1.2
2023.07.04 ver. 1.1	2024.04.07 ver. 1.3

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。
その後数回改定を行なって、Blender 3.1 までに導入されたノードの機能を中心にして、様々な作例とその解説を記載しました。

その後も、Blender が 3.2 そして 3.3 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、Vol. 1 の本と少し別方向の作例や、Blender 3.2 以降に追加されたノードを使った作例集として作成をすることにしました。
ジオメトリノードの開発はその後も活発で、「そろそろ機能が出そろって、新ノードの追加も鈍くなってきたかな？」という状態にはまだまだ達していません。
本書 Vol. 3 では、Blender 3.4 そして Blender 3.5 で導入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。
第1.1版では、Blender 3.6 で増えた「シミュレーションノード以外」の機能についての解説や作例を少し加えています。
Blender 4.0, 4.1 への更新に伴って、第 1.2 版、1.3 版に更新をしました。

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 Vol. 1 の本 や Vol. 2 の本を読んでいるだろう…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。

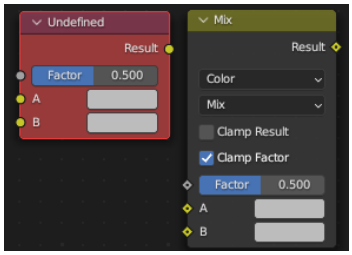
※本書に埋め込んである画像の一部は GIF アニメーションになっています。
.pdfとして書き出したファイルは、残念ながら静止画として最初のフレームが使われているだけなのですが、html版はブラウザーで見れば動いて見えるはずです。
また、**サンプルファイルは Blender 4.1 で保存をしました。**互換性に関する仕様として Blender 3.6 で開くことはできますが、それより前の Blender では開くことはできません。
Blender 3.6 で保存をすることで、Blender 3.5 以前の Blender で開くことが出来ます。(但し、必ずしも正常に動作するかは保証できないのですが…)

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 3.4 以降で注意するべき、ジオメトリノードやその周辺のも含めて変化を挙げておきます。

色の操作がいくつか変更

Blender 3.4 での地味ながら大きな変化として、Mix RGB ノードが Mix ノード(ミックス) に変更されていることがあります。
Mix の機能が拡張され、色だけではなく数値やベクトルにも対応するノードになっています。



3.3 以前に作った .blend ファイルを 3.4 以降に持って行く場合には自動で変換がされるのですが、
3.4 で保存した .blend ファイルは、3.3 ではノードが見つからないことになってしまいます。

Mixノードは、ジオメトリノードだけではなく シェーダーのノードでも共通の変更が行われているので、
マテリアルに関しても、3.4以降で作ったMixノードを使っているファイルを3.3で使えないということが起こります。

前のバージョンにさかのぼってファイルを読み込むのは、もとからあまり推奨されることではないのですが、
色関係のノードとして使う頻度が高いノードでの変更なので、要注意です。

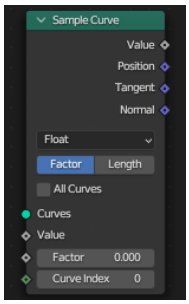
Transfer Attribute(属性転送)ノードも変更

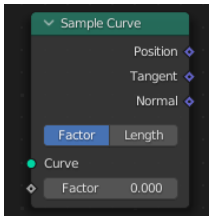
Blender 3.4 で、Transfer Attribute(属性転送)ノードが解体されて、
３つのノード群 (Sample Nearest , Sample Index, Sample Nearest Surface) に変更されました。
これらの詳細は、Sample Nearest 系ノードの章に記載をしています。

また、Blender 3.6 では類似したノードとして Index of Nearest(最近接インデックス)というノードも追加されています。
元から少し扱いの難しめのノードなので、一般的なユーザーは利用機会はそんなに多くないのですが、注意が必要です。

Sample Curve の強化

3.4 では カーブ上の位置や向きの情報を得るための Sample Curve の機能が強化されています。



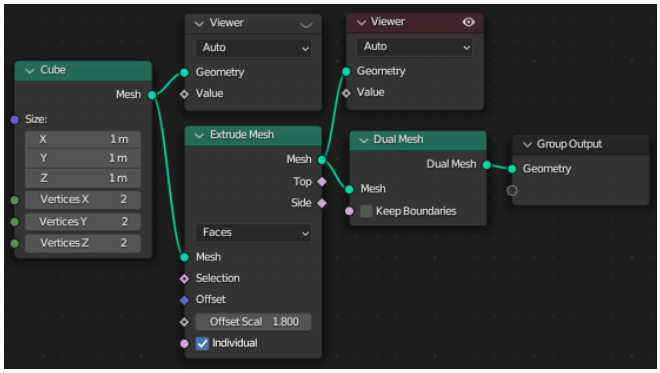


3.3 以前は、複数のスプラインがあってもまとめて[0-1]の範囲でFactorを指定しないといけませんでした。
つまり、２本の(同じ長さの)スプラインがあったら、それぞれに対して[0-0.5] と [0.5-1]が割り当てられるという具合です。

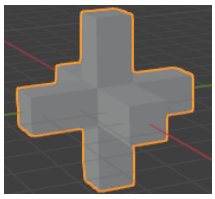
これは、複数のスプラインが存在するカーブでは使いづらく、その長さがまちまちだったりするとどうにもならなかったのですが、
3.4 からはスプライン（カーブ）の番号を別個に指定できるようになり、自由度が大幅に上がっています。
(旧来の挙動には、All Curves オプションを使います)
また、カーブ上の任意のアトリビュートなども評価できるようになっています。

ビューワーノードの強化

Blender 3.4 では ビューワーノードが強化されて、3Dビュー上で「途中の状態」を見ることができるようになっています。
以前はジオメトリノードで処理をしている途中の状態を、スプレッドシート上でデータを（数値で）見るのが基本的な用途でした。



新しい機能では、アクティブなビューワーノードの「目」のアイコンがチェックされていると、その状態が3Dビューに表示されるようになっています。

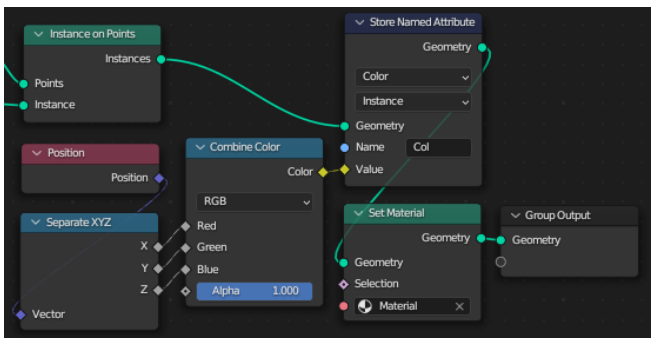


左図のような状態だと、Extrude(メッシュ押し出し)した状態が3Dビューに表示されるわけですね。

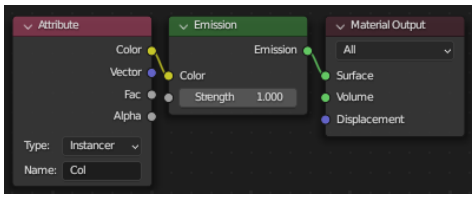
ジオメトリノードで編集中の状態が視認できるので、ややこしい処理をして「だんだん分からなくなってきた」ときに、とても役に立ちます。
これはあくまでプレビュー用なので、レンダリングしたり、別のジオメトリノードを追加したりすると、最後まで処理が進んだ Group Output での形が表示されます。

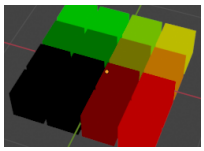
インスタンス単位でのアトリビュートの利用

Blender 3.3 までは、頂点や面といったジオメトリの要素が持つ属性だけをシェーダーで利用することができました。
そのため、「インスタンス単位での色違い」のような表現をするときも、すべての頂点(や面)に対しての色属性を作る必要があり、いささか無駄のある設定をしなければなりませんでした。
Blender 3.4 では、インスタンス単位の色属性などもシェーダーで利用できるようになり、そうした無駄な設定をしないで済むようになりました。



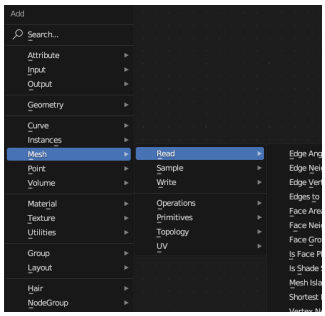
インスタンス単位の属性(Col)に色情報を収容したとします。
シェーダー側の Attribute ノードの設定を Instancer にすると、その色情報が使えます。





インスタンス単位での色違いなどが容易に実現できるようになりました。
今までも工夫して多少のノードを追加すれば実現ができたのですが、より単純で分かりやすくなっています。

メニュー構造の一新

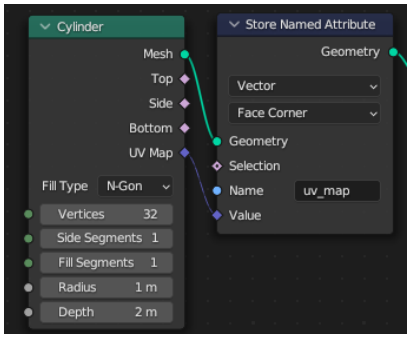


新しい機能のノードが増えるにつれて、ノード追加メニューの中身が当初のものから膨れ上がってきていました。
そのため、Blender 3.5 でメニューの並びが大幅に改定されています。

カテゴリーごとの分類が整理され、Curve(カーブ) や Mesh(メッシュ) といった特に基本となるカテゴリーでは、
サブメニュー内がさらに分類されて、 3 段階のメニュー構成になっています。
また、Field at Index(インデックスからフィールド)から Evaluate at Index(インデックスでの評価)のように名前が変更されたノードもあります。

本文中にメニュー構成を表記するときには、3.5 以降のメニューに準じて表示をすることにします。

プリミティブのUV情報

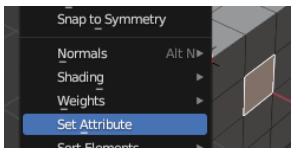


Blender 3.4 まで、ノード内で作成したプリミティブには uv_map というアトリビュート名でUV情報が収容されていました。
（たとえ必要ない場合でも自動で情報を持つようになっていました）

Blender 3.5 から UV Map というソケットから UV 情報を取り出すように変更されています。
それに伴って、必要な場合は明示的にアトリビュート情報を保持するようにノードを組まなければいけなくなっています。
3.4 以前の .blend ファイルを 3.5 で読むときには、自動で Store Named Attribute(名前付き属性収容)ノードが追加されます。

アトリビュートの編集

Blender 3.5 では、専用のGUIなどが伴わない簡易的なアトリビュートの編集機能が付きました。

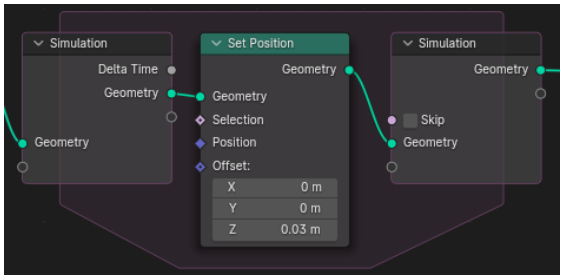


メッシュ編集時のメニューから、Mesh - Set Attribute(属性を設定)にあります。
現在アクティブなアトリビュートに対して、数値やそのほか色などの設定をすることができます。

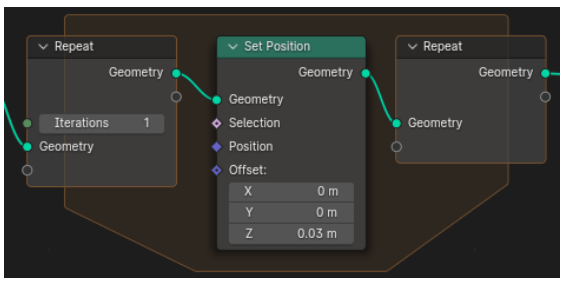
これによって、メッシュの頂点や面にあらかじめ手でパラメーターを与えて、ジオメトリノードでそれを利用することが容易になっています。
GUIを用いた編集手段などの充実が今後の早い時期での実装が目標になっているようです。

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。



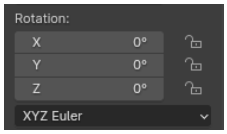
Simulation Input(シミュレーション入力) と Simulation Output(シミュレーション出力) に挟まれた部分の処理が、シミュレーションで実行される処理です。
ジオメトリに何らかの処理をして Simulation Ouput まで行くと、
その結果を次の時間フレームの Simulation Input に使う、という理屈で時間と共に進化していくシミュレーションが実現できます。
このシンプルなノード組みは、毎フレームごとに0.03だけZ軸方向に移動する、非常に単純なシミュレーションの例です。
シミュレーション機能にはこのシリーズでは深く踏み込みませんが、非常に強力な機能なので、
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、Repeat Input(リピート入力) と Repeat Output(リピート出力) 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。

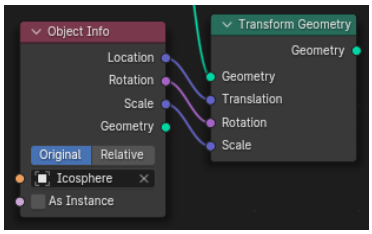
この例なら、「0.03ずつ10回移動するなら0.3移動すれば良いではないか」というところですが、
もちろんもっと複雑なことをするのに役に立ちます。

回転の表現



Blender 3.6 までは、ジオメトリノードの回転の表現にはオイラー回転の３成分のベクトル表記のみが使われていました。
XYZ軸回りの回転の程度を表す、デフォルトでオブジェクトの回転状態を表しているこの３成分のことです。

Blender 4.0 以降、Quaternion 表記に相当する回転の表現が追加で導入されています。
データのタイプとして Rotation (回転) となります。
Quaternion 表記では成分は４成分になり（人間目線だと）やや理解が難しいのですが、数学的には式がシンプルになり計算がしやすくなります。

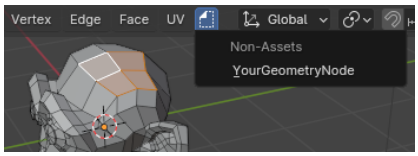


新たに Rotation 用に紫色のソケットが導入されて、Quaternion による回転状態をやり取りできます。
これに伴い、回転に関係するソケットが、ベクトル（青）から回転（紫）に変更されたノードが複数あります。

その分、Blender 4.0 ではどちらの表現での回転なのかの区別を厳密に使い分ける必要が生じたのですが、
Blender 4.1 以降では、ベクトル（オイラー回転）と回転のソケットをつなぐと自動で変換が行われるようになり、
区別が曖昧でも使いやすくはなりました。
※ ただ、混乱のもとなので、使い分けの意識は持っていた方が良いと思います。

ツールとしての利用

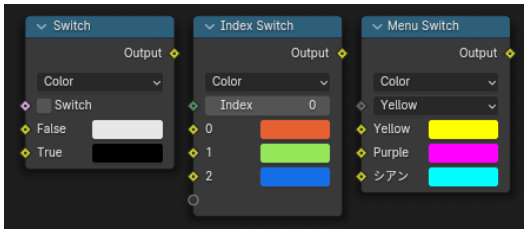
Blender 3.6 までは、ジオメトリノードはモディファイアとして使う以外の利用法はありませんでした。
Blender 4.0 で、オブジェクトやメッシュに即影響に使える機能としてジオメトリノードを使うことができます。
（モディファイアを追加した後適用して、すぐにメッシュの形状を変化させるようなイメージです）



適切に設定をすると、メニューの中から自分の作成したジオメトリノードを直接呼び出して編集作業に使うことが出来ます。
この Tool タイプのジオメトリノードに関しては、Vol.4 の本で詳しく解説をします。

スイッチの強化

Blender 4.1 ではスイッチ機能が強化されて、Index Switch(インデックススイッチ) と Menu Switch(メニュースィッチ)が追加されました。

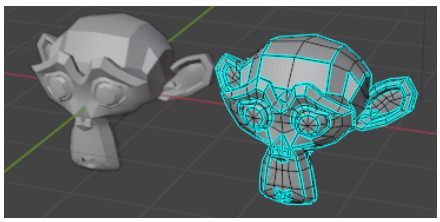


今まで、Switch ノードによって、True か False かの２択で値を使い分けるということが出来ました。
False なら黒、True なら白にする、というような使い方です。
Index Switch は従来の Swithc を機能強化したもので、任意の数の選択肢から選択するような使い方が可能になっています。

複数の選択をするような場面では今まで複数の Switch を組み合わせる必要があったのですが、
Index Switch １つで済むようなっています。

Auto Smooth(自動スムーズ)廃止

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。
従来の Auto Smoothのような自動処理をしたい場合には面と辺に対しての Smooth/Sharp 設定で管理を行います。



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設定するようになります。
今までの Auto Smooth のような自動処理をしたい場合には、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。

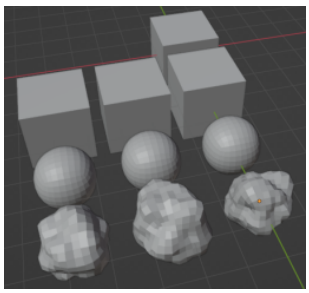
ジオメトリノードの基本

この章は、Vol.1 や Vol.2 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。
3冊目の Vol.3 に手を出している以上、ある程度は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。



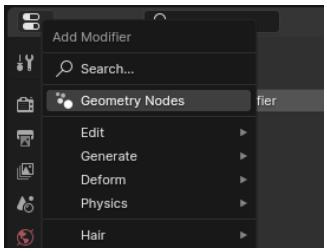
例えば、デフォルトキューブに対して

〇 Array(配列)
〇 Subdivision Surface(サブディビジョンサーフェス)
〇 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。

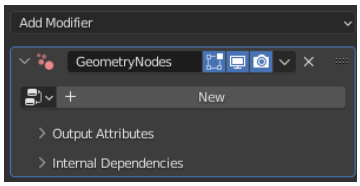
これらのモディファイアを後から取り除けば、オブジェクトは元の形状に戻るので「非破壊」というわけです。

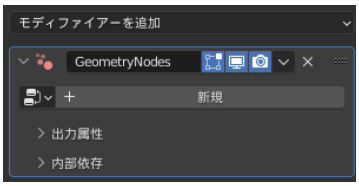
ジオメトリノードは、非常に高機能でカスタマイズが可能なモディファイア、と考えることができます。
モディファイアを追加するメニューの中に GeometryNodes の項目があるので追加をします。



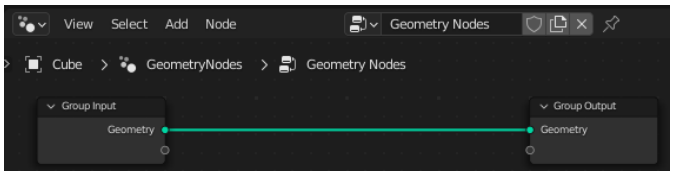
モディファイアのメニュー表示は、Blender 4.0 で更新されています。
ジオメトリノードは、メニュートップの選択しやすい位置に移動しました。

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。





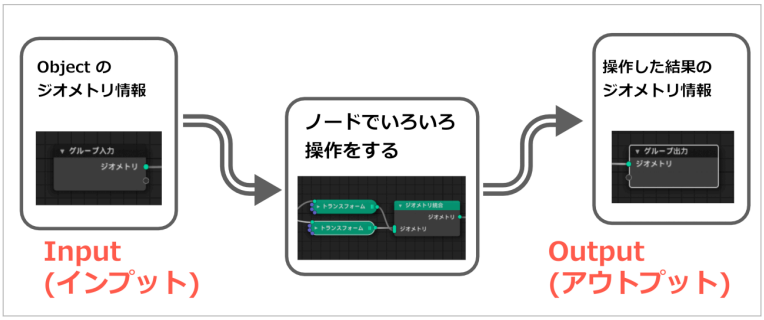
New(新規)ボタンを押すことで、新規のジオメトリノードが作成されます。
新規作成ではなく、既に使いたいノードが存在しているなら、それをメニューから選択すればよいわけです。



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。

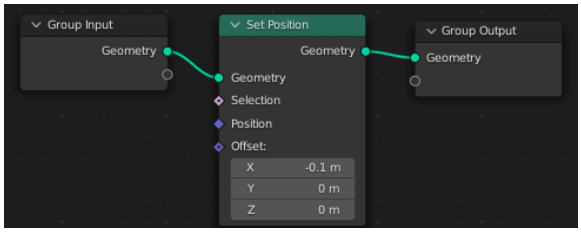
ジオメトリノードの編集

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。



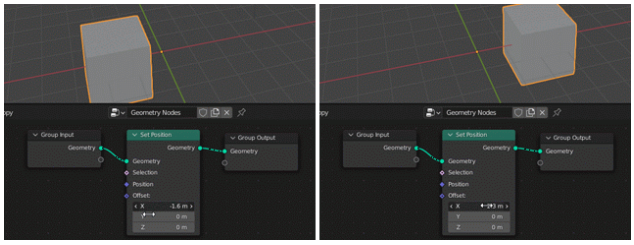
ジオメトリノードの基本形は、Group Input から Group Output の間に、様々な操作を挟み込む形になります。
Group Input (グループ入力) の Geometry (ジオメトリ) ソケットは、元々のメッシュ形状の情報です。
何もせずに直接 Input から Output に繋がれているのは「何もしないでそのまま」という状態です。
その間に、いろいろな操作を挟むことで、様々な編集が行えます。

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。

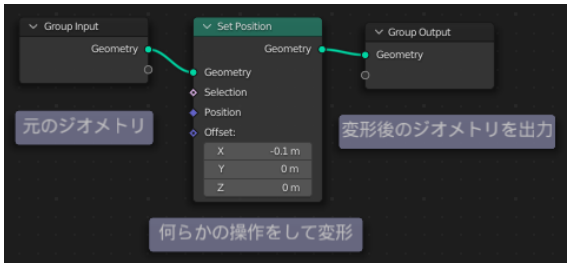


オブジェクト自体が動いているように見えますが、よく見ればオブジェクトの原点の位置は動いていません。
オブジェクトの位置はそのままで、メッシュが変形していることが分かります。

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

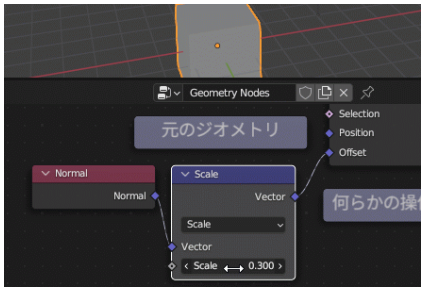
データの流れ(Geometry Node Fields)

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、

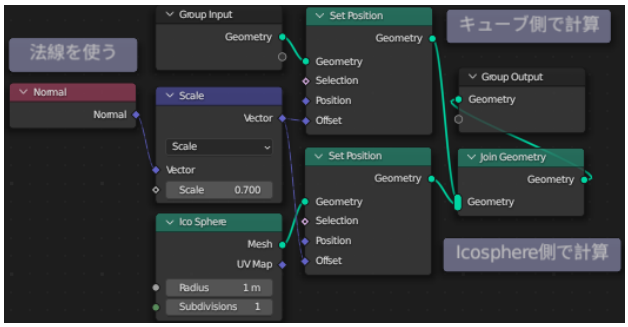
0. Set Position ノードで Offset のソケットに何かつながっている。
1. つながっているのはベクトルをスケールしたものだ。スケールと元のベクトルは何だ？
2. スケールは0.3だ。
3. ベクトルをたどると 法線(Normal)だ。

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）

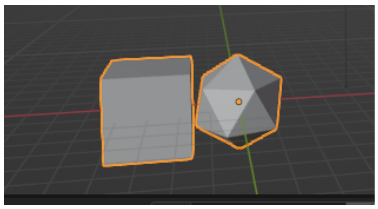
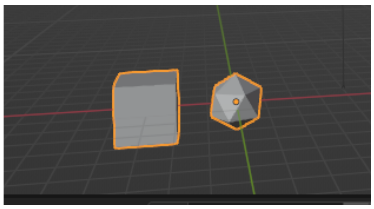
少しわかりづらい考え方なので、例を見てみます。
複数のメッシュを１つに合成する Join Geometry を使ってみましょう。



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。

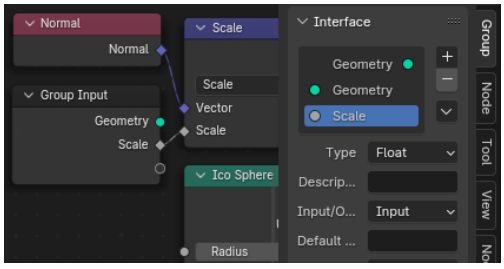



キューブと Ico球 それぞれで法線方向に膨らんでいます。
これは、Normal ノードが法線のデータそのものではなく「法線を使う」という情報を表すためです。
（法線が実際に評価されるのは、緑のライン(ジオメトリ)がつながっている Set Position のノードです。
そこで、「キューブの法線」「Ico球の法線」がそれぞれ別々に評価されているわけですね。）

アトリビュートの入出力

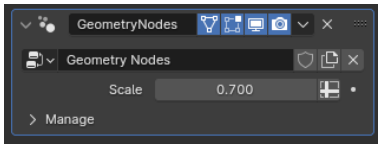
今までの例では、膨らませ方のパラメーターなどは、ノードの中で直接数値を編集していました。
それでは、「パラメータだけちょっと違う変形をする」ような場合などでもバリエーションの数だけノードを用意しないといけません。
そういう場合に、パラメータだけ変更する方法も用意されています。

アトリビュートの入力



もともと Group Input/Output のノードには、Geometry のソケットが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。

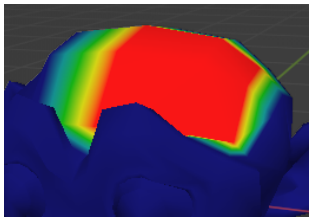
Group Input のノードの空のソケットにラインを繋ぐか、パネルの(+)ボタンメニューから、入力用のソケットを増やすことができます。
Scale のソケットからラインをつなぐと、同名(Scale)という入力ソケットが追加されました。



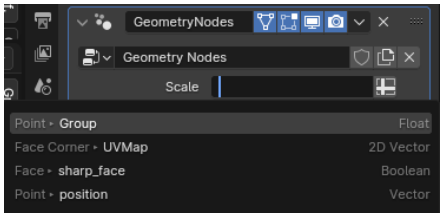
Group Input のソケットを増やすと、モディファイアのパネルにも対応する入力欄が現れます。
ここで、数値を入力することで、パラメータ違いのエフェクトなどを作ることができるわけです。

（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）

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。

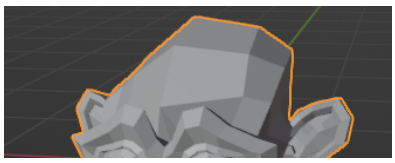


単一の数値だけではなく、元のメッシュが持っているアトリビュート(属性を入力にすることができます。
例えば、頂点ウェイト(デフォルト名 Group)を作って、一部だけ値を持たせてみます。



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。



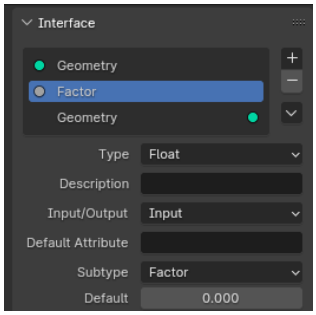
この例では、スザンヌの頭の一部だけが頂点ウェイトの値を持っているので、
頭の一部だけが変形するような操作が行えます。

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。

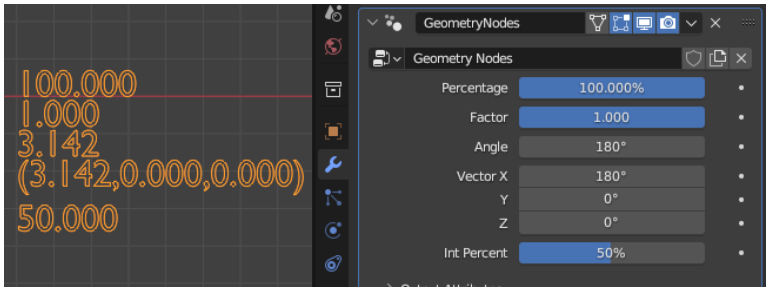
今までUVはUV、頂点ウェイトは頂点ウェイト、というように blender内部の仕組みはバラバラに作られていました。
ジオメトリノードの登場以降、こうしたパラメータなどはアトリビュート（属性）の一種として扱いが統一されつつあります。
ジオメトリノードで様々な操作を統一して行うことができるように改造しているということなのでしょう。
（Blender 3.x あたりのバージョンは、そうした統一の過程の過渡期になっているようです）

今までもバージョンが上がるとジオメトリノードで出来る操作の範囲が広がったりと、少しづつ変化してきました。
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。

blender 3.6 になり、入力の設定に Subtype(サブタイプ)が追加され、%表記や角度の表記を分かりやすくすることが出来るようになりました。



例えば、Float を入力する際に、サブタイプを Factor(係数)と設定することにより、
入力欄にスライダーを出すことができるようになっています。



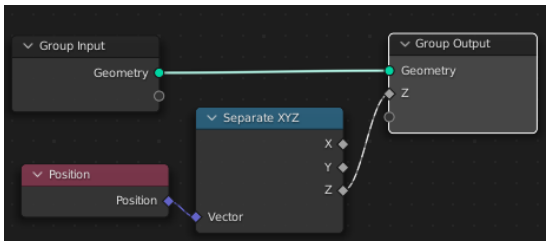
代表的なサブタイプ別の入力と、その結果入力される値を表示しました。

注意点として、%表記にしても 100%=1 ではなく値としては100が入力されることでしょうか。
100%を1と換算するにはジオメトリノード内で100で割る必要があります。

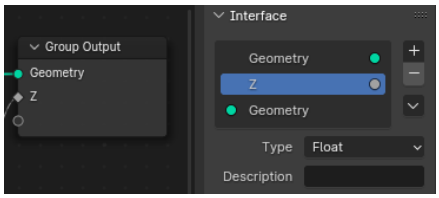
また、角度に関しては、入力は度(°)で行えますが、値はラジアンでの表現になります（180°=π=3.14159...）

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。

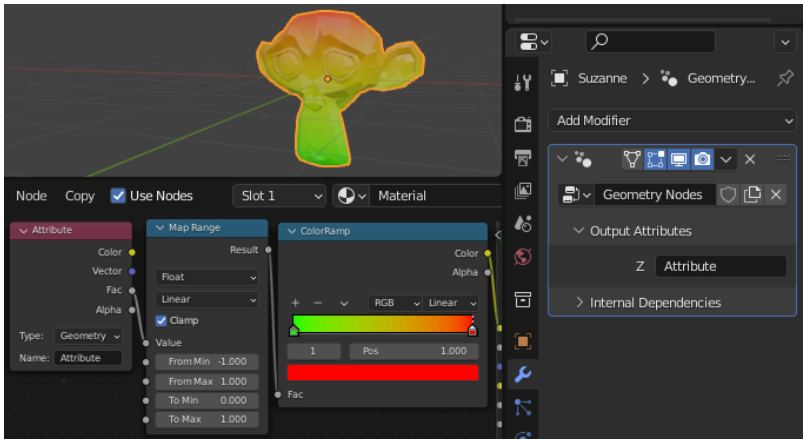


頂点の位置(Position)ノードから位置を得て、
Separate XYZ(XYZ分離)ノードで Z 成分を取り出しました。
Group Outuput につなげると、出力用のソケットが出現します。



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。

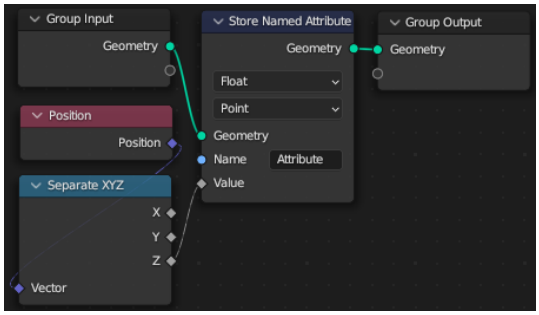
これで頂点の Z 座標成分を外部に取り出すことができます。
例えば、別のジオメトリノードを追加して、その入力として使ったり、シェーダーから利用するなどです。



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。



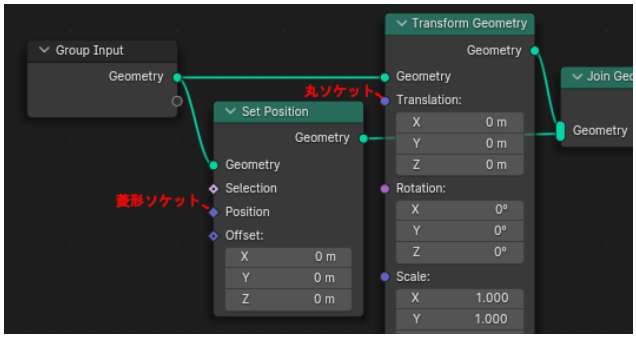
Group Output を使ってアトリビュートを出力することもできるのですが、
Store Named Attribute(名前付き属性収容) を使ってアトリビュートの出力をすることが出来ます。
ユーザーが属性名を変更する必要などの無い、名前決め打ちで良い場合には、
こちらの仕組みを使う方が使用時の手間が少なくて便利でしょう。

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

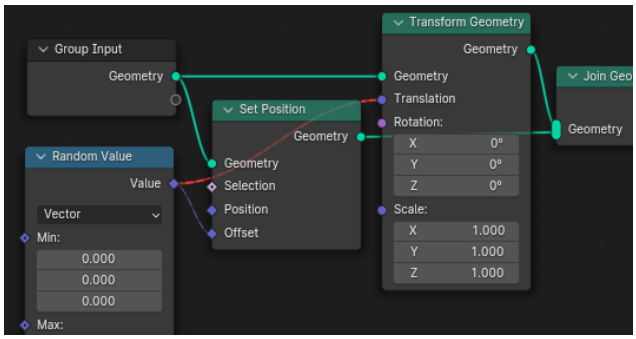
メッシュを平行移動で動かしたいときに、Transform Geometry (ジオメトリをトランスフォーム)を使うこともできますし、Set Position(位置設定)を使うこともできます。



これらのノードのソケットをよく見ると、Transform Geometry 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。
上の図の例では、(1, 0, 0)をノードで設定をすると、「すべての頂点に対して」(1,0,0)の平行移動をしますが…



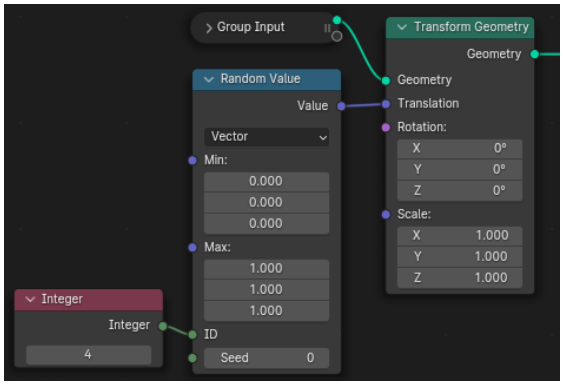
Random Value(ランダム値)などをつなぐと、各頂点に対してばらばらのランダムの値が与えられて、
頂点がそれぞれランダムに移動します。
このように、ただ１つだけの値ではなく「各頂点ごとのパラメータ」をやり取りする、ということを表すのが菱形のソケットです。

ランダムのノードを、Transform Geometry(ジオメトリのトランスフーム)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。



余談ですが、Random Value ノードの ID ソケットに、単一の値をつなげると出力はただ１つの値になります。
図では 4 の値を入力したのですが、これにより「頂点４に対して与えるはずのランダムの値」がただ１つ出力されることになります。

こういう場合は、Transform Geometry の丸いソケットにランダムをつなげることができて、 エラーにはならない…というわけですね。

Blender 3.4, 3.5 で主に追加されたノード

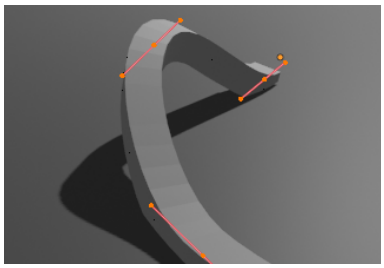
ここから先は、Blender 3.4 と 3.5 で追加されたノードや、増えた機能を使った作例を主に見てみます。
Blender 3.6 ではシミュレーションノードが増えていますが、ここでそれには言及しません。
シミュレーションノード以外では Index of Nearest(最近接インデックス)ノードが追加されています。

Curve

3.4 で増えたノードの中で顕著なのは、メッシュやカーブの形状に関する細かな情報を扱える Topology というカテゴリーです。
ところが、これが意外に複雑で使い方がややこしいものです。
頂点間のつながり次第で複雑な面を構成できるメッシュよりも、頂点が連続して連なっているだけのカーブの方が幾らか構造はシンプルです。
そこで、少し変則的ですが、カーブに関する機能から先に見て行こうかと思います。

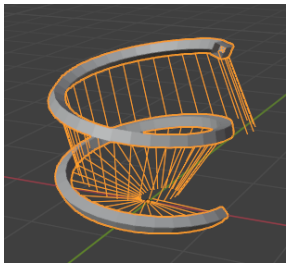
Set Curve Normal(カーブ法線設定)

カーブの本体自体は「太さの無い線」なのでレンダリングはされません。
普通は太さ情報を持たせてメッシュ化して表示を行います。
円筒のような表現であれば「向き」は関係がありませんが、断面が四角などの形状にしてメッシュ化をする場合は、ねじりの基準をどうするのか決めないといけません。



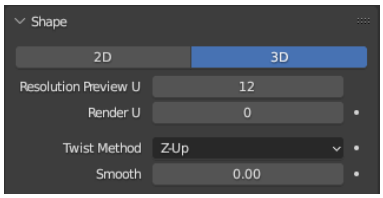
デフォルトの設定は、Minimum Twist(最小ツイスト) です。
3次元的にうねるようなカーブを作って試してみると、上面が垂直方向を向かずに傾くことが分かります。

Tiltを設定して、「ねじり」を指定するような場合はこの状態を基準として計算されるので、
任意の方向を向かせたいというような場合かなりやりずらいです。

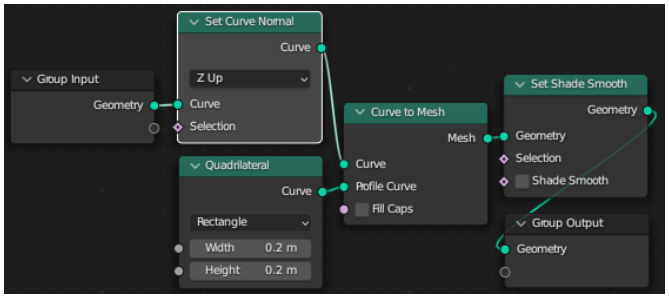


カーブの持つ捩じりの方向は、Normal(法線)のベクトル情報で得ることができます。
Vol.1 の本では、スパイラル状のカーブを作成して、その法線方向を見てみるという画像を収録しました。
この図は、「カーブの法線方向を Store Named Attribute 等で保持」して、
メッシュ化した後で「Extrude Mesh で頂点を保持した法線情報の方向に伸ばす」という方法でカーブの法線方向を線で見えるようにしたものです。

最初は水平方向を向いていた Normal 方向なのですが、Minimum Twist の設定だと螺旋運動にともなって微妙にねじれていき、下向きになっていきます。
このような微妙な変化があると、制御がしにくそうです。

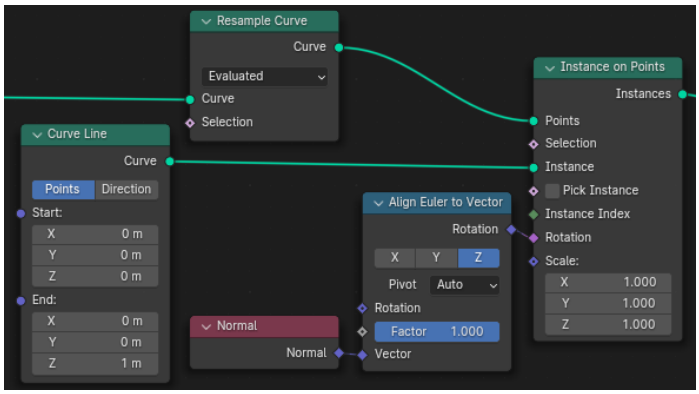


ジオメトリノード内でのメッシュ化ではなく、「従来のカーブの設定」を使って太さを持たせるときには、Z-Up などの設定があります。
ここの設定を変えるとカーブから作ったメッシュの向きを揃えることができます。
しかし、ジオメトリノード内で作ったカーブには3.3までこの設定をするノードがありませんでした。



3.4で追加された Curve - Operations(処理リスト) - Set Curve Normal(カーブ法線設定)を使って、
ジオメトリノード内で法線方向の設定を変更することができるようになりました。

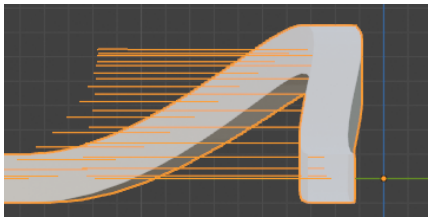
地味な機能ですが、これでカーブの向きの制御が一気に行いやすくなります。



折角ですので、ここでカーブに設定されている法線の向きをはっきりと見てみましょう。
Curve Line(カーブライン) ノードでZ方向を向いた直線を作成し、
Instance on Points (ポイントにインスタンス) で法線向きに配置してみます。

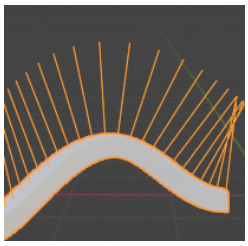
線の向きを法線方向に向けるために、Align Euler to Vector(オイラーをベクトルに整列)を利用します。

※ Blende 4.1 以降、青ソケット(オイラー角)の回転情報は、紫ソケットの回転につなぐと自動で変換が行われるため、
Euler to Rotation ノードを挟む必要はなくなりました



真横から見ると、確かに上面がZ方向に整列していることが分かります。
…が、Z-up という割に、法線方向が Z 軸の上側を向いていないですね。

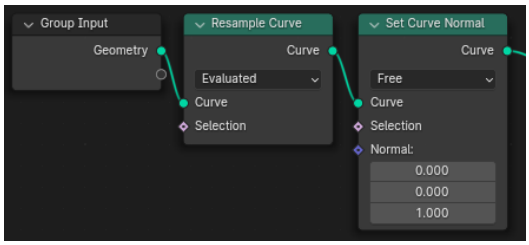
どうも、法線が Z 軸を向くというよりも、「法線を計算するための基準にZ軸を使っている」という意味のようです。
実際の法線そのものは基本的に水平方向を向くようです。

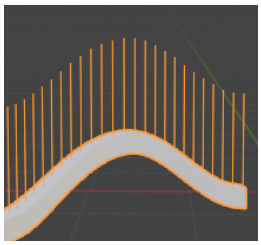


カーブ全体で Tilt の値を -90度にすると、法線方向が「上」を向きました。
（もちろん、カーブの湾曲に応じて法線の向きは変わるので、必ず真上を向くわけではなく、
「できるだけ上を向くように」ねじりの向きが揃うということですね）

Free(フリー) Normal

Blender 4.1 になり、カーブの法線方向の指定方法に Free(フリー)モードが追加されました。
Normal 方向を直接設定をすることが出来ます。

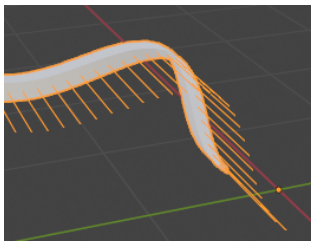




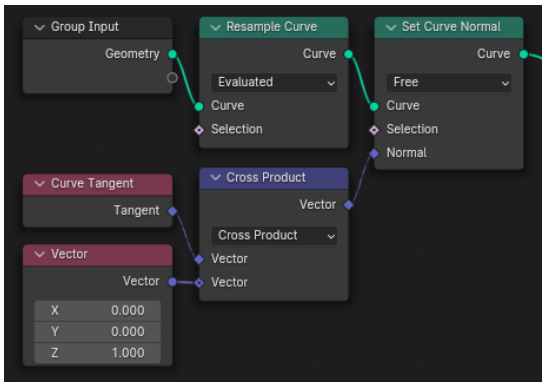
各点で Normal を設定するために、先に Resample をしている点に注意してください。
(先にコントロールポイントで Normal を設定して Resample で補間される挙動を防ぐためです)

Blender 4.1 beta の時点で、(0,0,1) を設定すれば法線はそのまま(0,0,1)になります。
※(0,0,2)を設定しても(0,0,1)になり、自動的に長さ１に調整されます

これは、考えてみると少し意外な実装です。
というのも、入力したベクトルがそのまま法線方向になるので、カーブの接線(Tangent)方向と法線が垂直になることが保証されません。

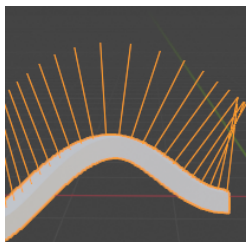


例えば同じカーブで(1,0,0)とほぼ横向きに設定すると、
カーブの断面がひしゃげてしまいました。
Tangent と Normal の方向が90度になっていないと、メッシュ化するときの断面の形状に影響が出てしまうようです。

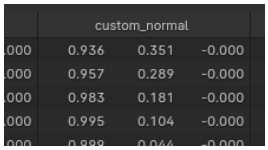


(0,0,1)を基準の方向として、必ず法線が Tangent と垂直になるように、
外積(Cross Product)を計算して Normal に設定するようにします。

この時に Tilt を -90 に設定すると、先ほどの Zup の時と同じような表示になります。



というか、実はこの**(0,0,1)方向を基準にして Tangent 方向に垂直な方向を計算して法線とする**、というモードが Zup モードの計算そのもののようです。
どおりで、Tilt が 0 のままの時に法線が横を向いたわけです。
(0,0,1)と垂直な方向を計算して法線としているのですから。



スプレッドシートでカーブの情報を確認すると、
Free モードで法線を設定したカーブには、custom_normal というアトリビュートが追加設定されていることが分かります。
従来の Normal を書き換えているのではなく、
ユーザーが設定したカスタム法線を追加する形での実装であるようです。

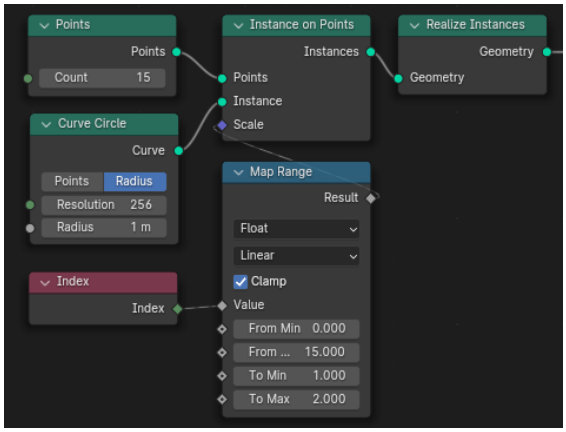
単に機能を確認して、画面上で見るだけのシンプルな作例を、01_CURVE/01_CurveNormal.blend として同封しました。
上の図は、Curve Line を使って線で向きを表示しましたが、作例中では Cylinder を使って太さのある円柱で法線の向きを表示しています。
この他、Set Normal を使って捩じり向きの調整が必要な場合の例として、最後の 応用の章に一つ作例 載せました。

Curve Topology(カーブトポロジー)

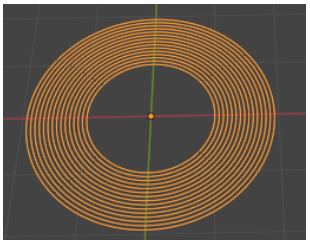
3.4 で新しく増えた Mesh Topology と Curve Topology のカテゴリーには、頂点とエッジや面に関する詳細な情報を得ることができるノードが複数収容されています。
但し、かなりマニアックなノード群で、メッシュやカーブのデータ構造に詳細に踏み込む必要が無ければ、あまり使う機会がなさそうなノードでもあります。
Mesh よりはシンプルなので、Curve に関する Topology の機能を先に見てみましょう。

Curve of Point(ポイントのカーブ)

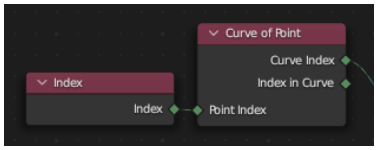
複数のスプラインを持つ場合を考えたいので、円を複数並べることにします。



Points と Instance on Points(ポイントにインスタンス) を使って、同心円上に円を配置しました。
円の半径を番号に応じて丁度良い程度に変化させるために、Map Range(範囲マッピング)を使っています。
インスタンスのままではその後の処理に問題があるので、最後にインスタンスを実体化します。

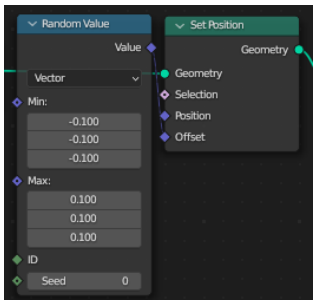


Curve of Point(ポイントのカーブ) は、カーブ上の点に対して、
「自分が所属してるカーブ（スプライン）の番号」
「それぞれのカーブ（スプライン）の中での番号」
を得ることができるノードです。

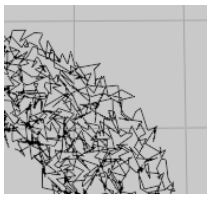


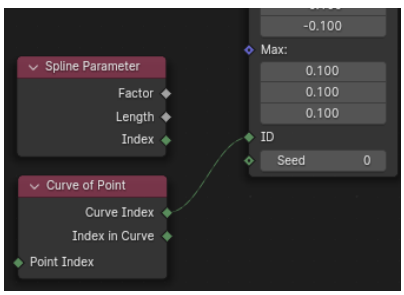
A of B (B の A) という名前のノードは日本語と英語で順番から逆というのがなかなか混乱を招く元ですが…
こればかりは仕方が無いので、慣れるしかありません。

文章での説明だけでは今一つ理解しにくいので、まず単純な例を見てみます。
同心円を動かすことをしてみましょう。

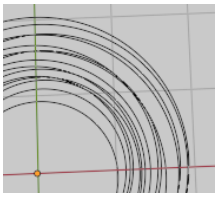


ポイント位置にランダム成分を加えると…素直な処理ではこのようにもじゃもじゃになってしまいます。
各点毎にランダムが与えられるのですから、当然ですね。

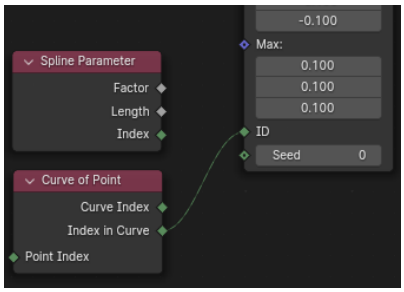




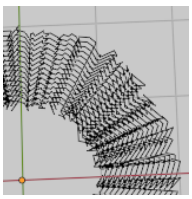
ランダムのIDに Curve of Point で得られるスプラインの番号(Curve Index)を繋げば、
「スプライン毎のランダム」な値を作ることができるので、同心円がズレた形状を作ることができます。



この Index in Curve は、実は Curve - Read - Spline Parameter(スプラインパラメーター) で得られる Index と同じものになっています



ちなみに、Index in Curve を使うと「スプライン毎に見たポイントの番号」が得られるので、
ランダムのIDに使うと、スプライン毎に同じランダムのパターンになってこのようになります。

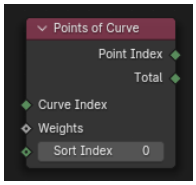


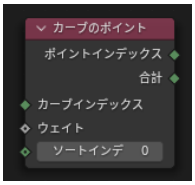
ポイントごとの処理をしている時は、普通に index ノードを使うと「そのポイントの番号」が得られます。
そうではなくて、「ポイント毎の処理をしている最中でもスプラインの番号を得られる」のがこのノードの役割です。

これらのトポロジー系のノードは単独での使い道はほぼ無いので、別のノードと組み合わせ使います。
次に、Curve of Point を他のトポロジー系のノードにつないで組み合わせてみます。

Points of Curve(カーブのポイント)

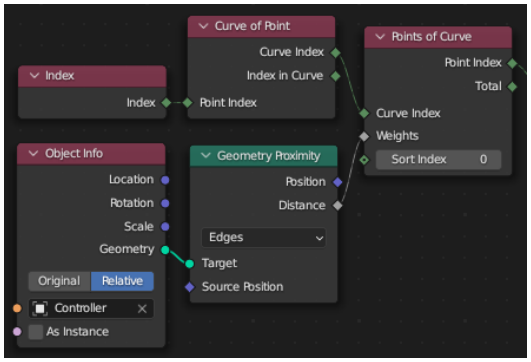
先ほどの同心円に対して、コントロール用のオブジェクトを近くに配置をして、
「それぞれのスプライン毎に、一番オブジェクトに近い点に物を配置する」ということをしてみます。
さて、この後はノードの名前からして混合しがちでややこしいのですが、注意して配置していきます…





Points of Curve(カーブのポイント)は、あるスプラインの中からポイントを一つ選びだすことができるノードです。
「選ぶ条件」は自分で決めることができ、「一番インデックスが若いポイント」や「z座標が一番大きいポイント」などを指示ができます。

別に「一番」何々でなく、２番目に何々、３番目に何々、といった選び方もできます。
その順番を指定するのが Sort Index(ソートインデックス)です。

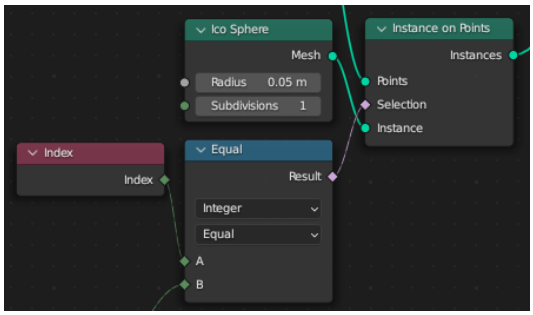


ポイントを選びだす条件を決めるのが Weights(ウェイト) と Sort Index(ソートインデックス)です。
スプライン内のポイントを、Weight 順に並べて n 番目のポイントのインデックスを得ることが出来る仕組みです。
デフォルトでは、Weights はインデックスそのものを使います。
Sort Index 0 だと、スプライン内で一番インデックスが若いポイント、つまり先頭のポイントの番号が分かります。

今回ここに、Geometry Proximity(ジオメトリ近傍)を使って
「コントロール用のオブジェクト Controller までの距離」を使ってみます。
Sort Index で 0 を指定しておくと、Weight の順番でソートした場合の最初の点の番号が得られます。

つまり、この時点で各頂点毎に、「自分の所属するスプライン(Curve Index)の中で」「一番コントロール用のオブジェクトに近い点の番号」が得られたわけです。

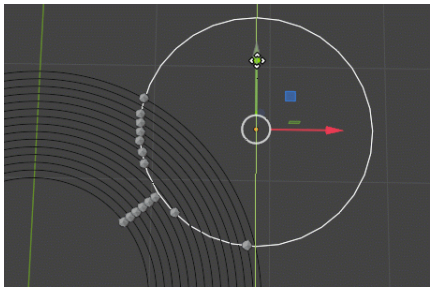
ここでトポロジー系のノードが難しいのは、点の「番号」の情報が得られても、それを直接使うようなノードがあまり無いことです。
Instance on Points を使って「一番近い点にインスタンスを置く」操作をしたければ、Selection という形に焼きなおさないといけません。



そこで、Instance on Points の Selection のソケットにつなぐために、
「自分のIndex」と比較をしてみます。

各ポイント毎に処理を行い「そのスプラインの中で、一番コントローラーとの距離が近いポイントの番号」
を得ているわけですから、

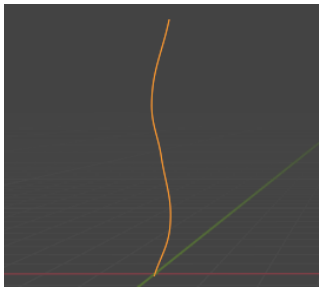
「自分のポイント番号と、得られたポイント番号が等しければ、つまり自分は最近点なので Selection を True にする」
ということをしたことになります。

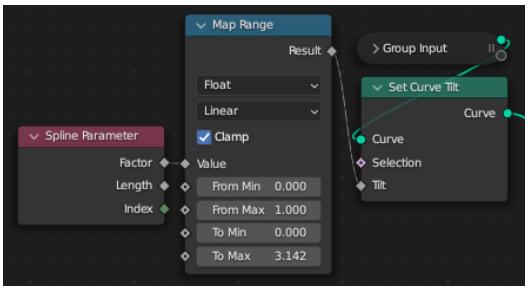


例えばコントロールオブジェクトを円にして、それを動かしてみます。
円周上で一番近い点にインスタンスが配置されていることが分かります。
球は実際には動いているのではなく、「カーブ上のポイントのうち一番近いもの」であるため、
移動の滑らかさは分割の細かさに依存することになります。
動画)Topology01.gif (pdfでは先頭のコマのみ表示されています)

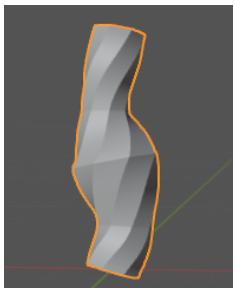
この例は、CURVETOPOLOGY/01_CurveTopologySimple.blend として同封しました。
とはいえ、少々地味すぎる作例なので、同じ仕組みでもう少しだけ派手な効果を作ってみます。

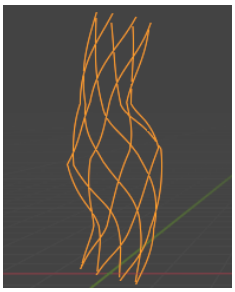
カーブに沿った最近点配置





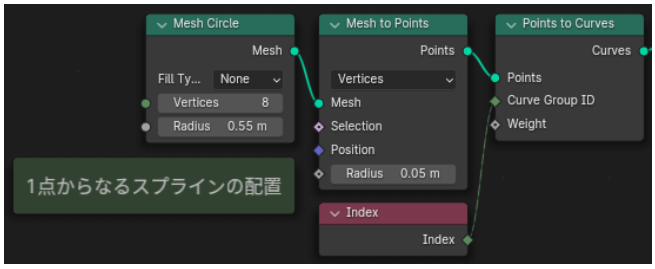
まず、インスタンスを配置する元になる形状を作ります。
ここでは適当に作成したカーブを元に少し複雑な形状を作成してみます。
カーブに沿ったTilt(傾き)を設定してやりました。





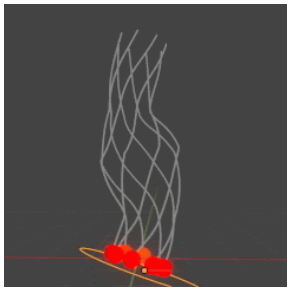
Curve to Mesh を使うと、断面の図形(カーブ)に応じて形を与えて、メッシュ化することができます。
普通は断面を円などにして、曲がった円柱（左)を作ることが多いのですが…

実はこの時に、「1点だけで構成されたスプライン」を断面として使うと
円柱のような太さを持った形状ではなく「線」を作成することができます。
そこで、複数の「１点だけのスプライン」を断面として使えば、
右の図のようなスプラインで構成された捩じった螺旋のような形状になります。



以前は「１点だけのスプライン」を作成する素直でシンプルな方法がなく、
多くのノードを組み合わせるしかありませんでした。

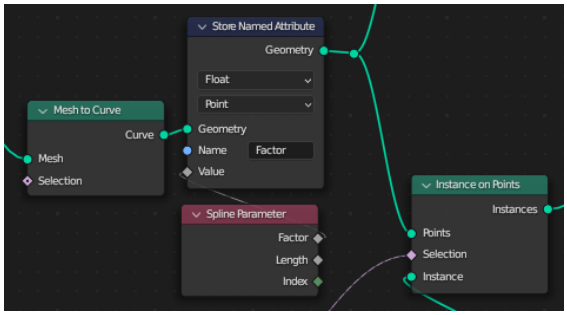
Blender 4.0 で、 Points to Curves が導入されたことで、
Curve Group ID のソケットを利用して、各点ごと独立したスプラインにすることができます。



後は先ほどの作り方と同じ理屈でノードを組めば、
コントローラーの動きに付随して、球がカーブに沿って移動する効果を作ることができました。
動画)Spiral01.gif (pdfでは先頭のコマのみ表示されています)

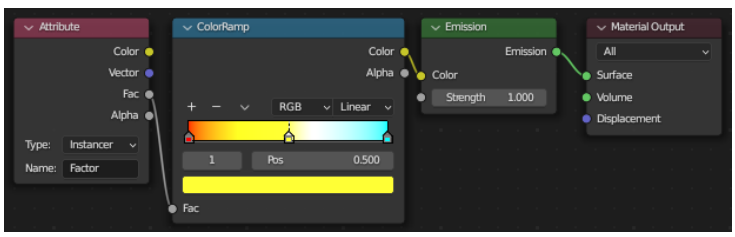
球の位置は、あくまでスプライン上で「一番近い位置」なので、
位置関係によっては滑らかに動けずに、時々ワープする場合があります。

位置によって色を付けるエフェクトを加えています。
このエフェクト部分のノードの組み方を確認してみます。



カーブの形状を作った段階で、Store Named Attribute(名前付き属性保存)などを使って、
Spline Parameter のファクターを保存しておきます。

このカーブ上の点にインスタンスを作成する際に、この属性はインスタンスにも引き継がれます。
Blender 3.3までは「インスタンスが持つ属性」はシェーダーで使えませんでした。
そのため、インスタンスが持つ属性にはあまり意味は無く、
「インスタンス実体化」を行い、球の各頂点が持つ属性などに変換をしておく必要がありました。

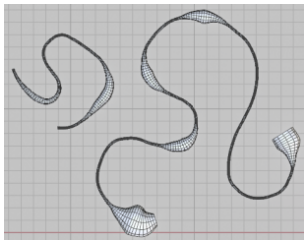


Blender 3.4 からは、
シェーダーの Attribute ノードのタイプを Instancer(インスタンサー)とすることで、
インスタンスの持つアトリビュート情報を使うことができます。
（単語のニュアンス的には「インスタンスの発生源の持っていた属性」ですかね）

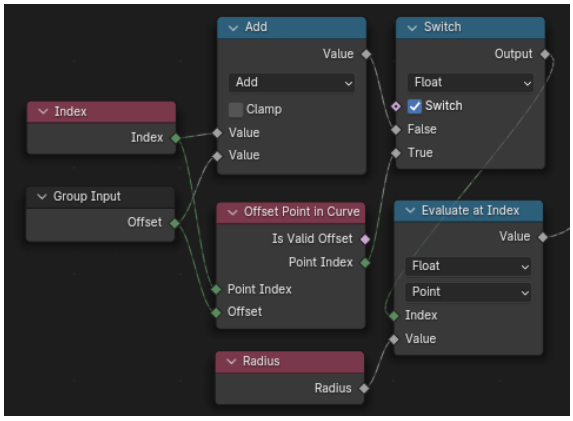
この例は、02_CURVETOPOLOGY/02_CurveTopology.blend として同封しました。

Offset Point in Curve(カーブ内ポイントオフセット)

Offset Point in Curve(カーブ内ポイントオフセット) は、ポイントのインデックスとオフセット量を与えると、
オフセットの分だけずれたポイント番号を返します。
…というと、単純にインデックスに Add(加算) をしてずらした場合と何が違うのだ？
と少し疑問に思うのですが、「ずらした番号が元の番号の範囲を超えた時」の振る舞いに少し違いがあるようです。



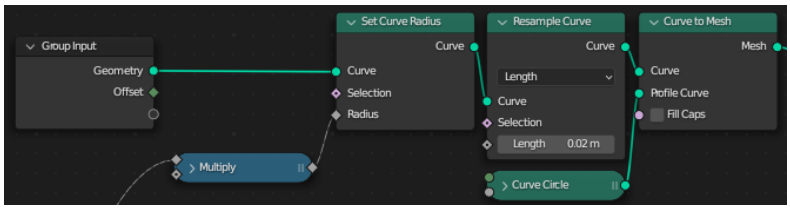
適当なカーブを作成して、場所ごとに半径に違いがあるように編集をしました。
これの半径の値を、ずれたインデックスで評価するようにして、違いを見てみます。



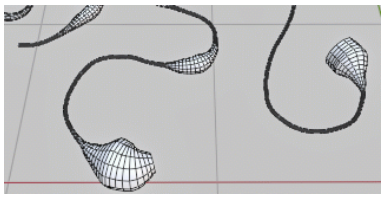
Index ノードに、単純 Add ノードで値(Offset)を加えた場合と、
Offset Point in Curve を使って Offset だけずらした場合の2通りのノードを用意しました。
Switch ノードを利用してこの2つの場合を切り替えて確認してみます。

ずれたインデックスを利用するには Evaluate at Index を使います。
Radius ノードを接続して、カーブの半径情報を得ることにします。

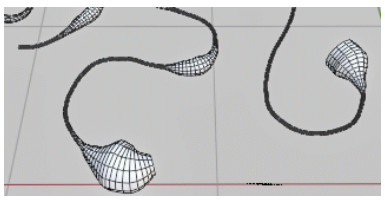
ズレた位置の半径情報を、Set Curve Radius(カーブ半径設定)で元のカーブの半径情報に上書きします。



見やすくするために、半径に定数値をかけたり、再サンプリングして解像度を細かくしたりする調整も行います。
Offset の値を変化させて行けば、半径を評価する場所がズレていくので、ふくらみが動いていくことになります。
この時のポイントインデックスは、スプライン毎のインデックスではなく、カーブ全体でのインデックスです。
複数のスプラインがあるときは、ふくらみは次のスプラインに乗り移っていくことになります。

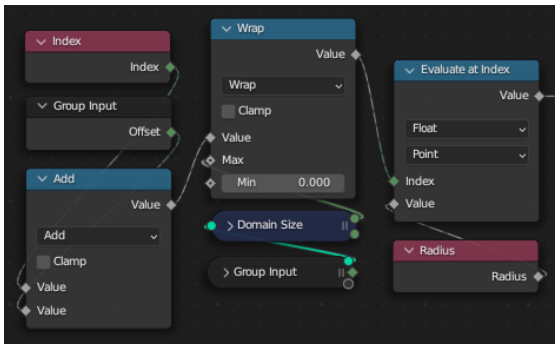


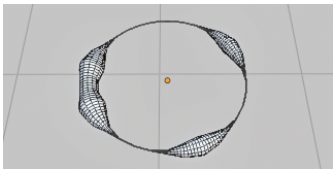
Offset Point in Curve を使った場合は、
ずらした結果がカーブの範囲外になると、カーブ端で上限や下限があるように出力するようです。
カーブの端で、太さ一定の範囲が広がっていきます。
（範囲外の評価かどうかは、Is Valid Offset(有効なオフセット)で確認できます）
動画)Offset01.gif (pdfでは先頭のコマのみ表示されています)



一方、単純に index に足し算をしただけであれば、
Evaluate at Index のノードには 範囲外の index もそのまま入力されます。
その際には、単純に半径 0 と評価がされるようです。
半径 0 の範囲が広がっていきます。
動画)Offset02.gif (pdfでは先頭のコマのみ表示されています)

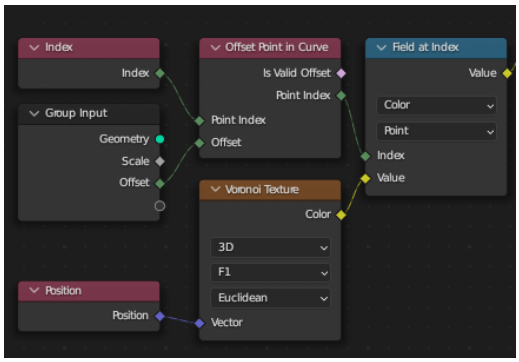
そのため、閉じたカーブなど範囲外のインデックスを周期的に繰り返して処理するような場合、
折角の新ノードですが使わずに、インデックスに単純な足し合わせをして、数式の Modulo(剰余)系のノードで処理をする方が素直です…



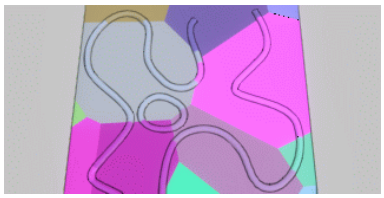


カーブ内の頂点の数で繰り返しをしたいので、数式ノードの Wrap ノードを使っています。
頂点数は、Domain Size(ドメインサイズ)ノードで得ることが出来ます。
動画)Offset03.gif (pdfでは先頭のコマのみ表示されています)

元からカーブが持つ属性である「半径」を使った上の例だけでは、素直すぎて今一つ面白みが無いので、
代わりにボロノイテクスチャの色を評価してみます。



背景には同じボロノイを使った板を置いています。輪郭線は Line Art を使いました。

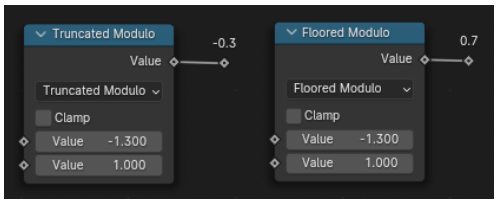


動画)Offset04.gif (pdfでは先頭のコマのみ表示されています)
Offset Point in Curve を使うことで、ずれたインデックスの位置でボロノイの色の評価ができます。
Offset を変化させると、カーブ線に沿ってずらして評価をしていることが分かります。

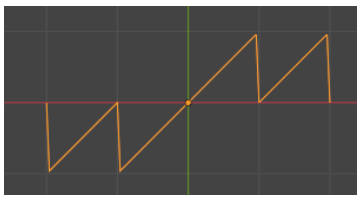
これらの図を作った例は、02_CURVETOPOLOGY/03_CurveOffset.blend として同封しました。

Modulo(剰余)ノード

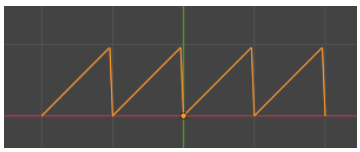
Blender 4.0 で、数式ノードにちょっとした更新があり、
Modulo (剰余) が Truncated Modulo(剰余(切り捨て)) と Floored Modulo(剰余(床)) の2つのノードに分割されました。
Blender 3.6 までの Modulo は Truncated Modulo に対応しており、Blender 3.6 の .blend ファイルを読むと Modulo は Truncated Modulo に変換されます。



今までの Modulo は、負の値に関して使いづらいところがありました。
-1.3 を 1 で割った余りは、「1.3 を 1 で割った余り」の正負を逆にした計算でした。



原点付近で y = Truncated Modulo(x, 1) をグラフにすると、このように原点で対称なグラフになります。
これでは、正の場合と負の場合で挙動が変わってしまうので、場合分けしないといけない場面が多くなります。



そこで、新たに「自分と同じかそれより小さい最大の整数」との差を計算する、Floored Modulo が導入されました。
-1.3 と 1 で割った余りは、-1.3 と -2 との差で 0.7 となるという理屈です。
この場合は、グラフは正負に関わらずこのようなのこぎり状で、原点で挙動が変化しません。

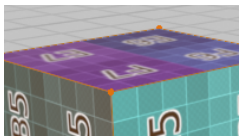
負の値も関係するような剰余を使いたい場合には、Truncated Modulo ではなく Floored Modulo を使う方が間違いが少なくて便利だと思います。

Mesh Topology(メッシュトポロジー)

Mesh Topology は、カーブトポロジーの時と同じように、メッシュのデータ構造の詳細を得ることができるノード群です。
カーブの時と違って、頂点、エッジ、面、そして面コーナーの 4 種類の概念が関係するために、さらに混乱しがちなノードです。
そこで、少しマニアックなのですが効果を簡単に確認できるノードを最初に見てみます。

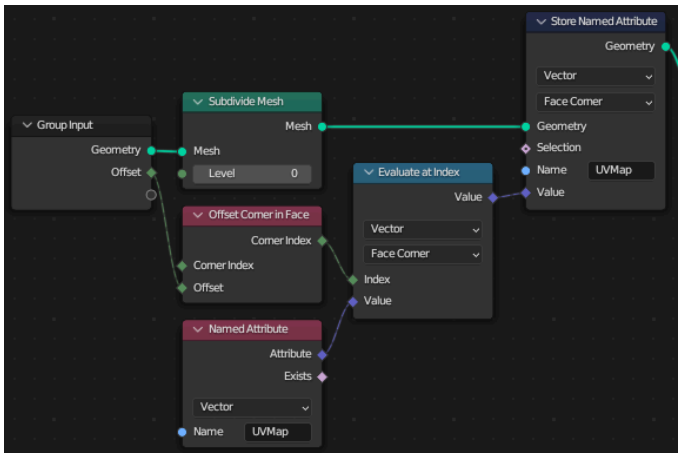
Offset Corner in Face(面内コーナーオフセット)

Mesh - Topology - Offset Corner in Face(面内コーナーオフセット)は、面コーナーをずらした番号を知ることができます。
面コーナーが重要になる機能の代表は UV マップです。
UV マップを使って、面ごとに違うテクスチャを貼ることができます。



１つの角の頂点は、点としては1点ですが、点が持つべきUV情報は1つだけではありません。
各面にバラバラのテクスチャを貼りたいような場合には、
それぞれの面に対応した３つのUV値を持っていないといけません。

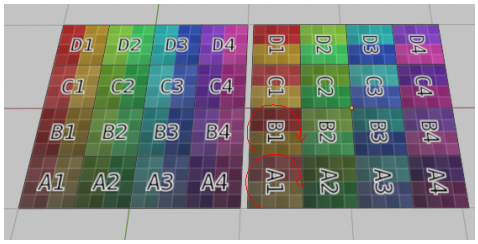
頂点がUVデータを持つというよりも、各面の隅（の頂点）がデータを持つ、という事になるので、「面コーナー」というわけです。
UV マップを張り替えるノードを組んでみます。



Store Named Attribute を使って、UVMap のデータを上書きしています。
Offset Corner in Face ノードに Offset の値を与えると、その分だけずれた面コーナーが得られます。
四角い板におなじみのカラーグリッドを貼って確認します。

例えばOffset を1にすると、index 0 の場所では 番号 1 が出力されます。
index 1 の位置の UVMap の値が Evaluate at Index(インデックスでの評価)を使って得られ、
それが index 0 のUVMap に上書きされるわけです。

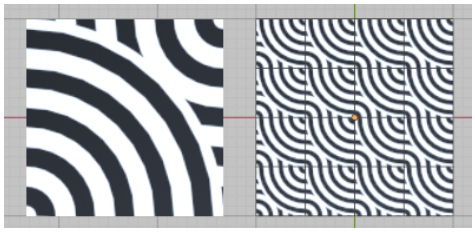
同じように 1は2に、2は3に対応するUVに書き換えられます。
そして順番に4は5…ではなくて0に戻ります。
面の周りをぐるりと一周すると、元のコーナーに戻ってくる実装になっています。
四角い面のコーナーが1つずつずれたわけですから、UVが、すなわちテクスチャが90度ずれて表示されました。



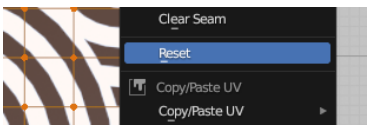
この時、Subdivide Mesh をかけて、面を16分割をしてみるとこうなります。

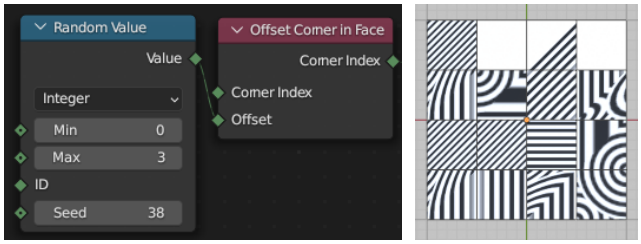
分割した16のタイルそれぞれに対して同じことが起こるので、
カラーグリッドの位置が変わらずに、個々の向きだけが90度回転しました。

カラーグリッドの例を見るからに、繰り返しのタイルの処理に便利そうです。
真四角(等多角形）以外のタイルだと、UVの位置をずらすと形状が歪んでしまいそうですから、四角いタイルを敷き詰めることにしましょう。
同心円を組み合わせたような形状（下図）を作りました。
この形状は、よく見ると上下左右がつながる仕組みになっています。



そのため、四角タイルそれぞれの UV をリセットして、[0,0]-[1,1]の範囲の四角にすると、
敷き詰めるようなUVで利用ができます。

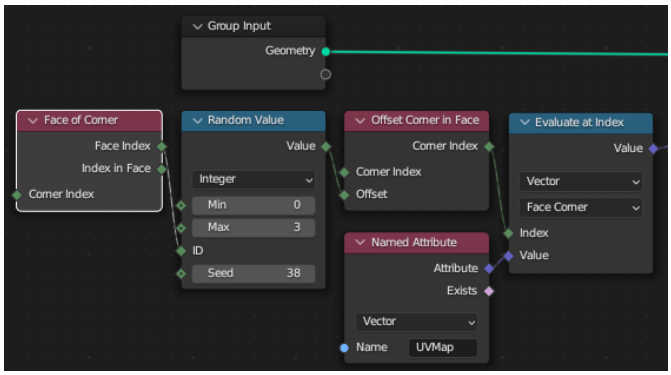




そこで、オフセットの値にランダムを導入すると…
うわー、それぞれの頂点（コーナー）毎にランダムになるので、
UVの形状が崩れて無茶苦茶になってしまいます。
コーナーごとではなく、面ごとのランダムでないといけません。

Face of Corner(コーナーの面)

幸い「今扱っているコーナーは、どの面のコーナーなのか？」を得るノードが用意されています。
Face of Corner(コーナーの面)は、上の例のような状況で役に立ちます。



Face fo Corner によって、「今処理しているコーナーが、何番目インデックスの面に所属するコーナーなのか」が分かるので、ランダムに使うIDのソケットにつなぐことで、面ごとにランダムが得られます。

Random Value ノードの IDは、デフォルトでは「今処理している要素のインデックス」です。
そのため、最初はコーナーのIndex を使っており、全コーナーでバラバラなランダム値になっていたわけです。
今面ごとの値をIDに渡しているので、同一の面では同一のランダム値になります。

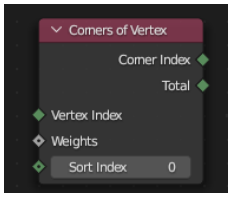
（スプライン毎のランダムを得た時の例と似ていますね）

意図したように、各面ごとにランダムに回転した模様が得られました。
四角タイルのランダム敷き詰めは、汎用性が高そうなテクなので、
知っていて損は無さそうなノウハウに感じます。

これらのサンプルは、03_MESHTOPOLOGY/01_OffsetCorner.blend として同封しました。

Corners of Vertex(頂点のコーナー)

さて、ここからもう一段ややこしい使い方のノードになるので、じっくりと行きましょう…

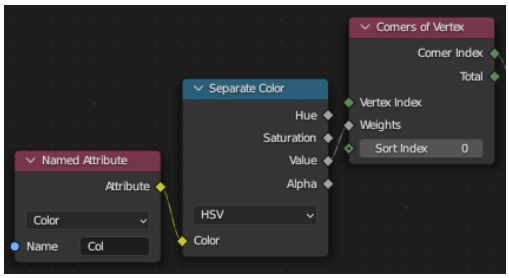


Corners of Vertex(頂点のコーナー) は、ある処理している頂点のインデックスを渡すと、その頂点が所属しているコーナーのインデックスを返します。

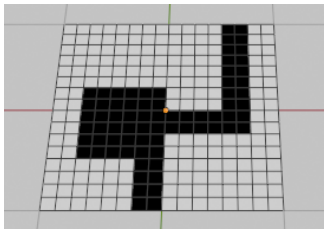
一つの頂点は、複数の面に所属しているので、複数の面コーナーを兼ねています。
「インデックス番号を返す」という処理は、そのうちの１つしか返せないのでおかしいな。どのコーナーを返すのだろう？ と気が付いた人は鋭いです。

複数の候補から、１つを選ぶためにここでも Weights(ウェイト) と Sort Index(ソートインデックス)のソケットが必要になっています。
Points of Curveで、１本のスプラインの中のポイントを１つだけ選ぶときに、Weight と Sort Index のソケットを使いました。

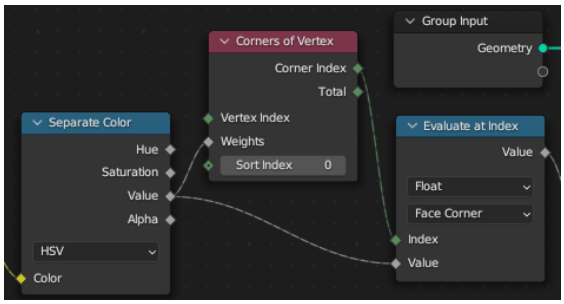
ここでも同じように複数の面コーナーから１つを選ぶためにこれらのソケットを使います。
面コーナー毎に何らかの値を Weights に渡します。
それをソートしたときに、先頭にくる面コーナーを返すという仕組みです。
(もしくは Sort Index の番号を0以外にすると、n番目の面コーナーを指定できます)



分かりにくいので実例を見てみます。
面コーナーごとに値を持つような属性は、UVの他に、カラーアトリビュートなどがあります。
そこで、カラーアトリビュート Col の明るさをウェイトに渡します。
すると、各頂点毎のソートの先頭、つまり「最も明るさが小さい」コーナーの番号が分かります。

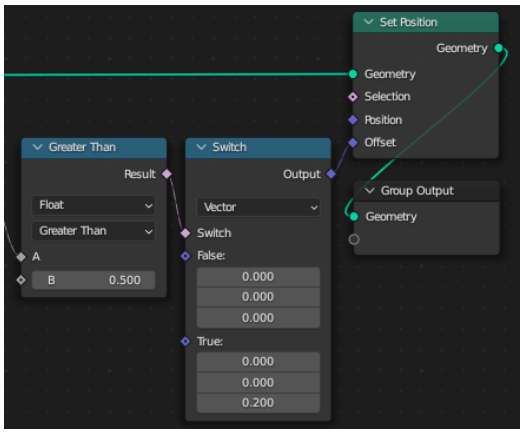


カラーアトリビュートの明るさの差がくっきり区別できて、
なおかつ形状がはっきり分かる…というような例として、このようにカラーアトリビュートを塗り分けた形状を例にしてみましょう。

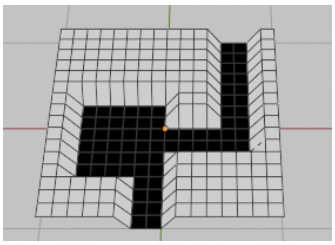


さて、各頂点毎に「最も明るさが小さいコーナーの番号」が分かりましたが、
番号が分かっただけでは、その後の処理で上手く使えません。

そこで、分かった番号で Evaluate at Index を行って、再度カラーの評価を行います。
これで、各頂点毎に「最も明るさが小さいコーナーの明るさ」が分かりました。



例えば、最小の明るさが 0.5 よりも大きい頂点をZ方向の上に動かす…というような処理をしてみます
すると、このようにゲームのダンジョンの壁のような変形をすることができました。

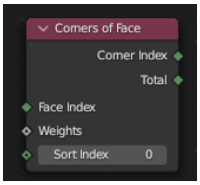


黒に接している頂点は、必ず最小値が0になるので動きません。
「黒に少しも接していない頂点だけ」が浮かび上がります。

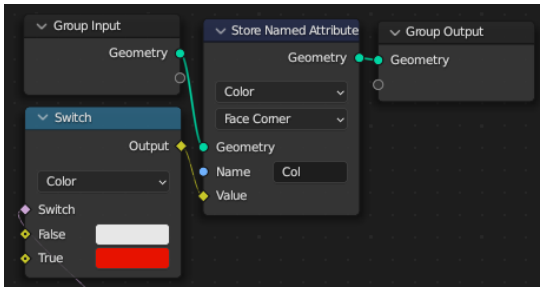
この変形処理程度であれば、わざわざ Corners of Vertex(頂点のコーナー)を駆使しなくても、行えそうな気がします。
少し大仰な仕組みでのノード組みです。
かなり複雑な細かい条件を必要とする場面でなければ、なかなか使いどころがないと思われるノードですが、Weight の理屈と使い方の簡単な例といったところです。
この例は、03_MESHTOPOLOGY/02_CornersOfVertex.blend として同封しました。

Corners of Face(面のコーナー)

似たような機能を持つノードに Corners Of Face(面のコーナー)があります。
これも、先の Corners Of Vertices と同じように Weight と Sort Index のソケットがあるのですが、何度か使い方を見ているので理解はだいぶ簡単になります。

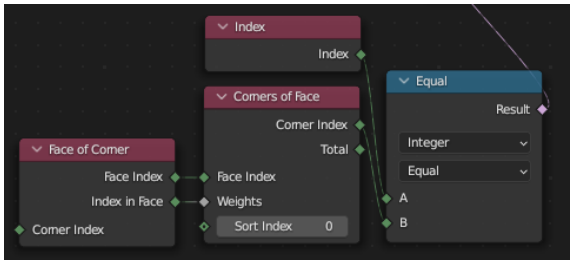


一つの面は、複数の取り囲む面コーナーを持っていますが、一つの面のindex の入力に対して、出力される面コーナーの index は一つだけです。
面の中のどのコーナーを選ぶのか？というところに、Weight のソケットを使います。



例として、
「各面のうち1つのコーナーのカラーアトリビュートを赤く塗る」
というノードを組んでみます。
まずは、各 Face Corner に白か赤の色を渡すノードを組みました。

ここで少し注意するのは、**白か赤かの判定をするのは「面コーナー」毎の処理**
というところです。



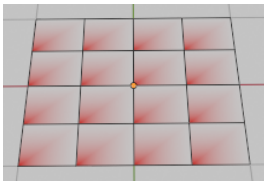
そのため、Corners of Face に渡すための Face Index は、
「今処理をしている面コーナーの属する面の Face Index」になります。
先ほど出てきた、Face of Corner の出番ですね。

そして、ソートに使う Weight は、index in Face つまり面ごとの面コーナーの並び順を使うことにしました。

この結果返って来るのは、「今処理をしている面コーナーが属している面の中で一番若い面コーナーの番号」です。
面コーナー毎の処理と、面に対しての処理が絡むので、ややこしいですね。
そして、この後には比較用の Equal ノードがつながっています。

このあたりは考え方がややこしいのですが、
もし「今処理をしている面コーナー番号」がこの「計算結果の番号」と等しければ、
今処理をしている面コーナーは、面の中で一番若いコーナーです。（なので赤くします。）

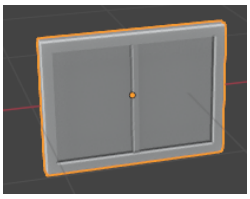
そうでなければ白くします。
この比較のために Equal ノードが必要になったわけですね。



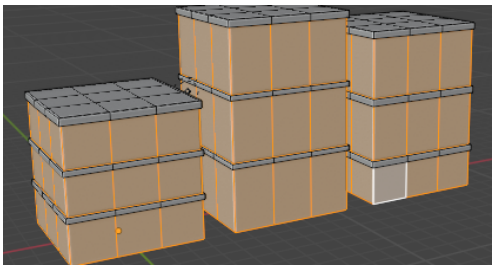
各面ごとに一番若いコーナーが赤く塗られました。
左下の隅が一番若いコーナーになっているわけです。
この例は、あまり実用は感じられない例ではありますが…
03_MESHTOPOLOGY/03_CornersOfFace.blend として同封しました。

面の底辺を基準に配置

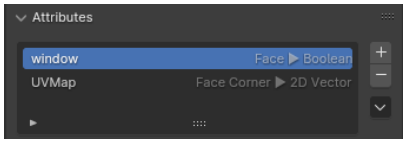
もう少し実用的な例として、ビルの壁面に窓を配置するときに、CornaersOfFace を使った工夫を見てみます。



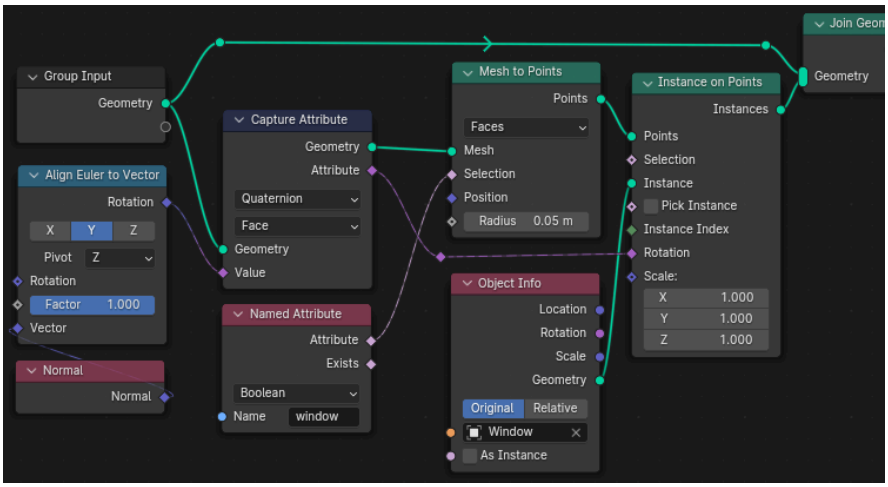
まずは、壁面に張り付けるための窓を作成しておきます。



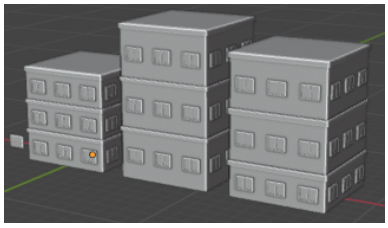
次いで、窓を貼り付ける簡単な建物を作成しました。
どの壁面に窓を貼り付けるか、アトリビュートを使って指定をするようにします。



アトリビュート window で指定された面を、一旦 Mesh to Points を使って点に変化させて、Instance on Points を使って窓を配置します。



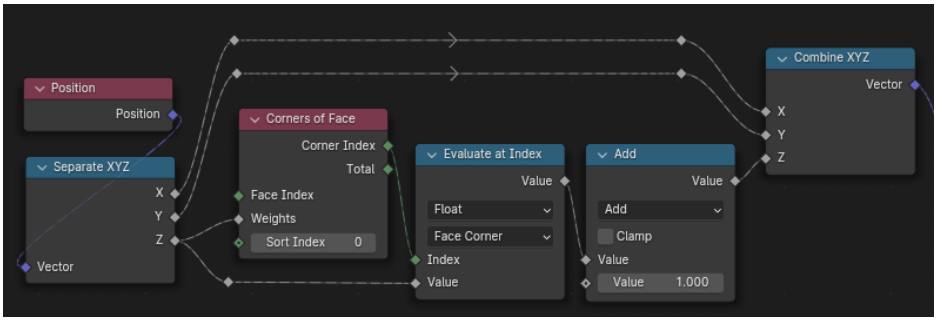
点は法線の向きの情報が無いので事前に Capture Attribute ノードを使い、法線方向を向くための回転情報を保持しておきます。



「面の中央に」窓が配置されます。
この場合、天井が高い建物と低い建物で、窓の高さが変わってしまいます。

細かいことを考えると、本当はこれは人間の背丈に合った窓の高さになるように、
建物の床からの高さを基準に配置したいところです。

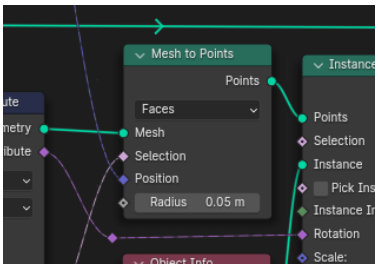
そこで、Corners of Face を使います。



評価用の Weights に位置のZ成分を使います。
すると、Sort Index 0 のコーナーというのは、Weights の一番小さいコーナーです。
つまり底辺を作るコーナー（のどちらか）の番号が Corner Index として取得できるわけです。

Evaluate at Index も利用して、その番号での Z 座標成分を得て、もともとの位置の Z 成分と差し替えます。

ただし、そのままだと床の高さに窓を作ってしまうので、Add で加算をして 1m 高い位置に窓を配置することにしましょう。



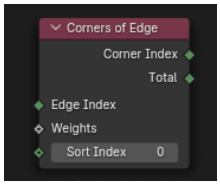
線が交錯してややこしくなってしまいましたが、
最初に使った Mesh to Points のノードの Position につなぐと、
「面の中央」の代わりに今計算した位置に点を配置することになります。
これで、計算した位置に窓を配置されるようになります。

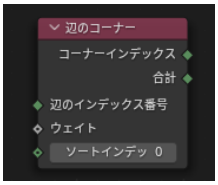
微妙な違いですが、最初の窓の配置と違って、面の中央ではなく、
床面を基準にした高さに窓が配置されていることが分かります。
これで、不等間隔の変則的な建物でも、窓の高さは人の背丈に合わせたような配置ができるわけです。
この作例は、03_MESHTOPOLOGY/04_SetWindow.blend として同封しました。

もう少し複雑な使い方をする例は、さらにこの後の作例で見てみます。

Corners of Edge(辺のコーナー) と Vertex of Corner(コーナーの頂点)

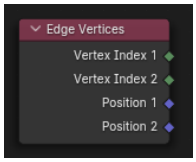
Corners of Edge は、実は他のトポロジー関連のノードと違い、Blender 3.4 から遅れることしばらくして Blender 4.0 で追加されたノードです。
今まで見てきたように、何とか of 何とか(何とかの何とか)という組み合わせのノードの中に、漏れがあって（？）後から追加されたものです。





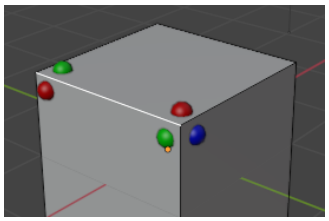
機能は今までの例からみて予想できると思います。
ある番号の辺(Edge)に対して、その両端に相当する「面コーナー」の番号、
の中の一つを選んで取得できます。

少しややこしいですね。



似たようなノード自体は既にありました。
エッジの両端の頂点と、その位置を取得できる、Mesh - Read - Edge Vertices(辺の頂点)と、
何が違うのか？便利なところってどこ？と気になるところです。

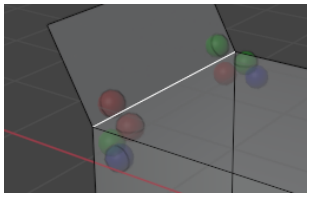
このノードで得られるのは「面コーナー」で、頂点そのものではありません。
面コーナーは、頂点のようですが「どの面に所属しているか」の区別があります。
一つの頂点は、複数の面コーナーを兼ねていることになります。



例えば、立方体の手前の角の頂点は、3つのポリゴンの角を兼ねているので、面コーナー番号としては３つの番号を持てるはずです。

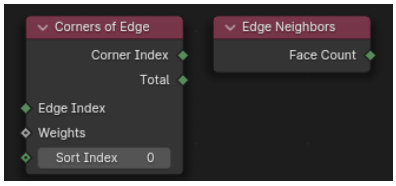
選択して白く表示されている辺についての Corners of Edge ががどの面コーナーの番号を返すのかというと、
赤、もしくは緑で示したペアの面コーナー番号です。
エッジが面を構成していない青い面コーナーは Corners of Edges では得られない仕様になっています。

通常の閉じたメッシュのエッジは、必ず２つの面に繋がっています。
そのため、取得できる面コーナーの候補は４つ(赤x2と緑x2)あるとも考えられますが…
頂点を共有する面コーナーはカウントせず、どちらの面からの面コーナーも含まれるようにする仕様になっていて、赤もしくは緑の2つの面コーナーの番号が得られます。
では、赤と緑のペアのどちらが得られるか…に関しては、内部データの頂点とエッジの収容順によって決まるっているので、どちらの組みでも大丈夫なように考えます。



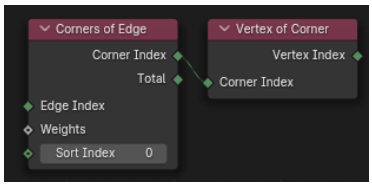
また、閉じていない面や、左の図のようにヒレのような面が生えて3つの面からなるような
特殊な(Non-manifoldな)辺の場合は、状況が変わるので少し注意が必要です。

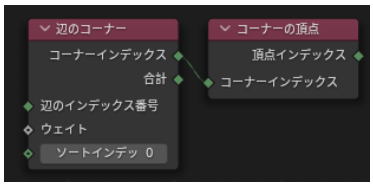
こうした特殊な場合について、もう少し詳しい挙動は Vol.4 で記載をしました。
※参考（Blender の Corners of edege マニュアル）



このノードには、Corner Index の他に、Total というソケットがあります。
これは、候補になる Corner Index の数…つまり、辺に繋がっている面を返します。
つまり、これは、Mesh - Read - Edge Neibors(辺の共有面数)と同じことになります。
閉じているメッシュであれば常に2になりますし、メッシュの端などでは1になります。

さて、面コーナーの番号(Corner Index)のままでは扱いづらいことも多いです。
頂点番号に変換して、頂点として扱いたい場合がよくあります。



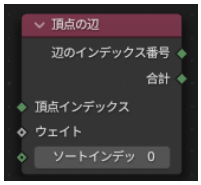


面コーナーの番号(Corner Index)から頂点番号へと変換をするには、
Vertex of Corner(コーナーの頂点) を使います。

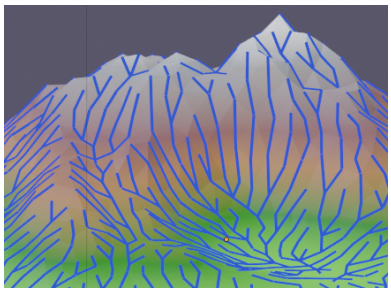
これらも単独ではなかなか使い道が無くて、組み合わせて使うようなノードです。
次の Edge of Vertex(頂点の辺)ノードと組み合わせて少し凝ったノード構成のサンプルを見てみます。

Edges of Vertex(頂点の辺)

Edges of Vertex(頂点の辺) は、ある頂点に繋がる複数のエッジ…のうちの一つの番号を返すノードです。

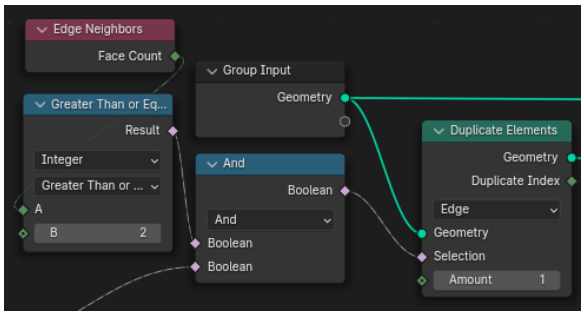


似たノードが続いてきたので、頂点に繋がる複数の辺（エッジ）のうち「どのエッジを返すのか？」の判定に Weight が必要なこともすんなり理解ができると思います。



今回、地形のようなメッシュから、
「勾配の大きい方のエッジを選択して抜き出す」
ノードを組んでみます。
Vol.2 でも Shortest Edge Paths(最短辺パス)などを使って似たようなエッジ抽出を行いましたが、
最短ルートではなく、傾きを判定に利用することにより、より水の流れに近い線を引くことになります。

※但し、後で解説をしますが、まだ正確な評価をするのに足りない機能があり、
今回の例は近似的なものになります



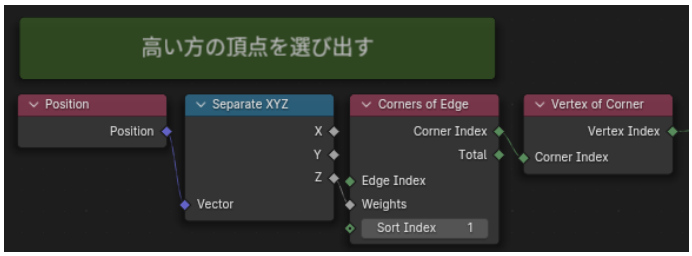
地形の形状は、既にあるものとします。
（Vol.2 のShortest Edge Paths の解説あたりで使ったテクを使って作ります）
地形を作る形状から、条件に合ったエッジを抜き出す為に、Duplicate Elements(要素コピー)
ノードを使うことにします。

ここで、Edge のタイプで処理をするので、これから先の処理…
(正確には、「このSelectionに繋がるこの前の処理」)は
全てのエッジ毎に行われる処理 であることに注意します。

まず、地形の端にあるエッジは、あまり意味が無いので最初に除外をするようにしました。
地形の端では、エッジにつながる面が 1 枚しかありません。
Edge Neibors で測定して、それ以上の面がつながっている場合だけ Selection に繋がるようにノードを組んでいます。

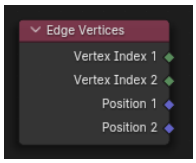
次に、勾配が大きいエッジを Edges of Vertex(頂点の辺)を利用して探していきます。
ところが、Edges of Vertex のノードを使うのに必要なのはエッジ番号ではなく頂点番号です。
処理中のエッジの持つ頂点番号を知らないといけません。

ここで「各エッジは2つの頂点から出来ている」という事が話をややこしくします。
2点の位置を比較して、「高い方の頂点の番号を使う」ことにします。

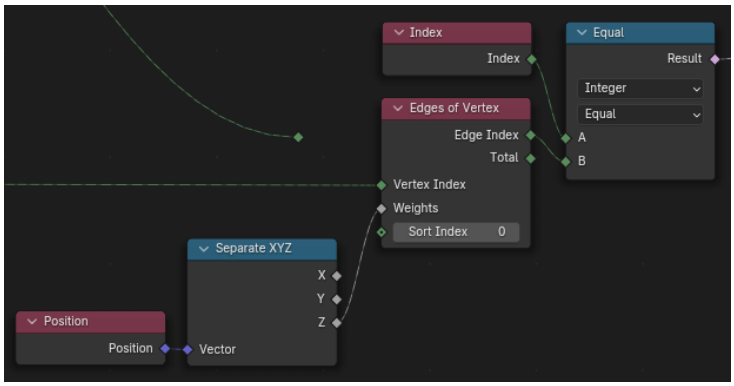


「高い方の頂点の番号を使う」部分のノードはこのようになります。
Corners of Edge(辺のコーナー)のウェイトに 位置の z 成分を渡せば良いわけです。
z 成分の順にソートされるので、「高い方」の、Sort Index は 1 になります。

※地形の縁は除外しているので、辺には必ず面（面コーナー）が 2 つあることを利用しています。
地形の縁まで含めると、例外などを考えないといけません。



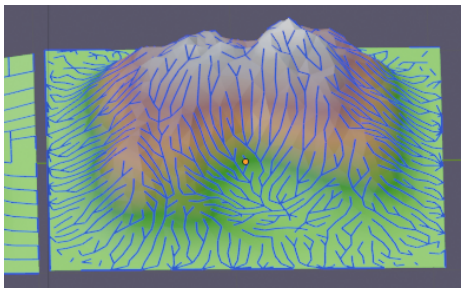
Edge Vertices を使っても同じことはできるのですが、Vertex Index 1 と 2 のどちらが高い方かは分かりません。
2つの頂点の比較をして、Switch ノードを使った切り替えが必要になったりするので複雑になってしまいます。



Edges of Vertex にこの頂点番号を繋ぐと、
この「高い方の頂点」を起点にして、繋がっている周囲のエッジを調べます。

Weight に位置の z 座標を使うことにします。
ある頂点の周りのエッジは、頂点を一つ共有しています。
位置が一番低いエッジが、そのまま「最も低い点に繋がっているエッジ」になっているはずです。
但し、番号が分かっただけではまだ直接は使えません。

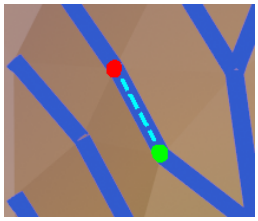
起点とした頂点が今処理しているエッジの「高い方の頂点」だったことを思いだせば…
「今処理しているエッジの番号」と「今回の処理で得られた番号」が等しければ、
「今処理しているエッジは上の頂点から水が流れてくる、条件を満たすエッジ」という事になるので、Selection に繋ぐことができるということになります。



あとは抜き出した線をメッシュ化して太さを持たせれば、
（近似的な）勾配に応じた線を引くことができます。

このサンプルは、03_MESHTOPOLOGY/05_SetLowEdgeSimple.blend に同封しました。

ところで、今回頂点から流れる水路(?)の方向を判定するのに、エッジの「位置そのもの」を使いました。
本当に勾配で判定をしたければ、位置ではなくて、頂点どうしの「高度差」つまり引き算した値を使って計算をしなければいけないはずです。
この計算では、傾きが緩くても長いエッジ…などがあると、傾きがきつくても短いエッジよりも優先して判定されてしまいます。



赤い点を起点として、水の流れ下るエッジを判定するために、水色のエッジのWeight 計算がされます。
同じように緑の点を計算している最中にも、水色のエッジのWeight 計算がされます。
この時、Weight に傾きを与えて評価したい場合は、赤い点から見た場合と緑の点から見た場合では Weight を逆に評価をしないといけないはずです。
さもないと、水が下から上に流れるような判定が起こってしまいます！

ところが、Edgess of Vertex のノードで判定するため、水色のエッジの Weight の処理をしている最中は、
どちらの点から呼び出されているのか…を区別する手段が（多分まだ）ありません。
ということで、エッジの頂点の差分を取った傾きを上手く扱うことができず、そうした心配のない頂点の位置情報そのものを使いました。

頂点の位置で判定している場合は、このような向きの問題は発生しません。
赤い点から見た水色の線は、周囲よりも低いので、水の流れる先の候補に挙がります。
そして、緑の点から見た水色の線は、周囲のエッジよりも高いところにあるので、
水の流れる先には判定されない、ということになるのです。